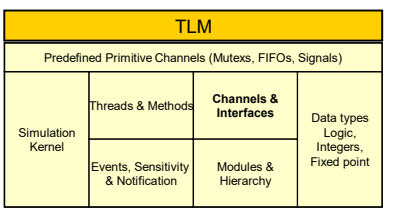
Plan

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

POLYTECH NICE SOPHIA
UNIVERSITÉ CÔTE D'AZUR

Transaction Level Modeling

- TLM Introduction
- TLM Interfaces
- TLM Channels
- Example



TLM

- TLM Standardization Alliance
  - June 2004 : OSCI / OCP-IP
  - Common TLM API
- Companies endorsing TLM standard within press release:
  - Cadence, CoWare, Forte, Mentor, Philips, ST, Synopsys
  - Atrenta, Calypto, Celoxica, Chip Vision, ESLX, Summit, Synfora
  - OCP-IP
- Why ?
  - Integrate Hw & Sw models
  - Early platform for Sw development
  - Early system exploration and verification
  - Verification reuse
- TLM version
  - 1.0 : Standard release (June 2005)
  - 2.0 : Draft release (Nov. 2006)
  - 2.3 : Build-in SystemC

TLM API Goals

- Support design & verification IP reuse
- Usability
- Safety
- Speed
- Generality
  - Abstraction levels
  - Hw / Sw prototyping
  - Several communication architectures (bus, packet, NoC …)
  - Different protocols

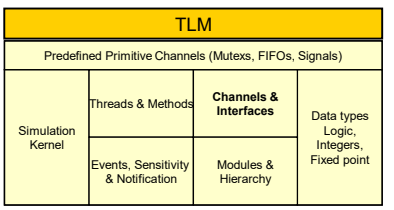
Key concepts

- Focus on SystemC interface classes
  - Define small set of generic, reusable TLM interface
  - Different components implement same interfaces
- Object passing semantics
  - similar to sc_fifo, effectively pass-by-value
  - Avoids problems with raw C/C++ pointers
  - Leverage C++ smart pointers and containers where needed
- Unidirectional versus Bidirectional dataflow
  - Unidirectional interfaces are similar to sc_fifo
  - Bidirectional is possible by using Unidirectional interfaces
  - Separates requests from responses
- Blocking versus non-blocking
- Use sc_port and sc_export

POLYTECH NICE SOPHIA
UNIVERSITÉ CÔTE D'AZUR

Transaction Level Modeling

- TLM Introduction
- TLM Interfaces
- TLM Channels
- Example



TLM Interface style

- same as sc_fifo
- blocking / non-blocking
  - SC_THREAD : blocking & non-blocking (wait calls)
  - SC_METHOD : non-blocking only
- Tranfers
  - Unidirectional
  - Bidirectional
- TLM Tag
  - C++ Trick
  - Allow us to implement more than one version interface

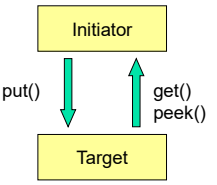
```
template<class  T>
class tlm_tag
{ };
```


TLM Interface style

- **Nonblocking:** Means function implementations *can never* call wait().
- **Blocking:** Means function implementations *might* call wait().
- **Unidirectional:** data transferred in *one* direction
- **Bidirectional:** data transferred in *two* directions
- **Poke/Peek:** Poke overwrites data and can never block. Peek reads most recent valid value. Poke/Peek are similar to write/read to a variable or signal.
- **Put/Get:** Put queues data. Get consumes data. Put/Get are similar to writing/reading from a FIFO.
- **Pop:** A pop is equivalent to a get in which the data returned is simply ignored.
- **Master/Slave:** A master initiates activity by issuing a *request*. A slave passively waits for requests and returns a *response*.

TLM Interface
Unidirectional Interfaces

- Blocking Interfaces (SC_THREAD)
  - put() : from Initiator to Target
  - get(), peek() : from Target to Initiator



get

```
template < typename T >
class tlm_blocking_get_if : public virtual sc_interface
{
public:
  virtual T get( tlm_tag<T> *t = 0 ) = 0;
  virtual void get( T &t ) { t = get(); }

};
```

put

```
template < typename T >
class tlm_blocking_put_if : public virtual sc_interface
{
public:
  virtual void put( const T &t ) = 0;
};
```

peek

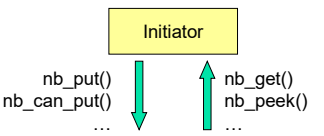
```
template < typename T >
class tlm_blocking_peek_if : public virtual sc_interface
{
public:
  virtual T peek( tlm_tag<T> *t = 0 ) const = 0;
  virtual void peek( T &t ) const { t = peek(); }
};
```

get & peek

```
template < typename T >
class tlm_blocking_get_peek_if :
  public virtual tlm_blocking_get_if<T> ,
  public virtual tlm_blocking_peek_if<T>
{ };
```


TLM Interface
Unidirectional Interfaces

- Non-Blocking Interfaces (SC_METHOD, SC_THREAD)
  - from Initiator to Target
    - nb_put(), nb_can_put(), ok_to_put()
  - from Target to Initiator
    - nb_get(), nb_can_get(), ok_to_get()
    - nb_peek(), nb_can_peek(), ok_to_peek()



nb_get

```
template < typename T >
class tlm_nonblocking_get_if : public virtual sc_interface
{
public:
  virtual bool nb_get( T &t ) = 0;
  virtual bool nb_can_get( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const = 0;
};
```

nb_put

```
template < typename T >
class tlm_nonblocking_put_if : public virtual sc_interface
{
public:
  virtual bool nb_put( const T &t ) = 0;
  virtual bool nb_can_put( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const = 0;
};
```

nb_peek

```
template < typename T >
class tlm_nonblocking_peek_if : public virtual sc_interface
{
public:
  virtual bool nb_peek( T &t ) const = 0;
  virtual bool nb_can_peek( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const = 0;
};
```

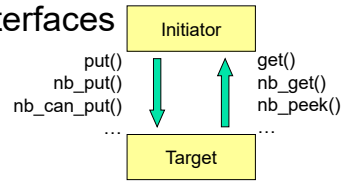
nb_get & nb_peek

```
template < typename T >
class tlm_nonblocking_get_peek_if :
  public virtual tlm_nonblocking_get_if<T> ,
  public virtual tlm_nonblocking_peek_if<T>
{ };
```

nb_ = non-blocking

TLM Interface
Unidirectional Interfaces

- Mixed Blocking / Non-blocking Interfaces
  - get(), put()
  - peek()



get / nb_get

```
template < typename T >
class tlm_get_if :
  public virtual tlm_blocking_get_if< T > ,
  public virtual tlm_nonblocking_get_if< T >
{ };
```

put / nb_put

```
template < typename T >
class tlm_put_if :
  public virtual tlm_blocking_put_if< T > ,
  public virtual tlm_nonblocking_put_if< T >
{ };
```

peek / nb_peek

```
template < typename T >
class tlm_peek_if :
  public virtual tlm_blocking_peek_if< T > ,
  public virtual tlm_nonblocking_peek_if< T >
{ };
```

get / nb_get / peek / nb_peek

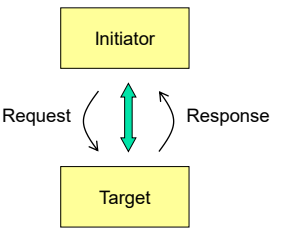
```
template < typename T >
class tlm_get_peek_if :
  public virtual tlm_get_if<T> ,
  public virtual tlm_peek_if<T> ,
  public virtual tlm_blocking_get_peek_if<T> ,
  public virtual tlm_nonblocking_get_peek_if<T>
{ };
```


TLM Interface
Bidirectional Interfaces

- Blocking Interface (SC_THREAD)
  - No Non-Blocking interface !
  - tlm_transport_if class
  - transport() method

Request Response

```
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_interface
{
public:
  virtual RSP transport( const REQ & ) = 0;
};
```



TLM Interface FIFO

- Based on implementation on sc_fifo
- tlm_fifo behavior
  - when you put a transaction into the tlm_fifo, you cannot get until the next delta cycle.
  - zero sized
  - infinite sized

```
template< typename T >
class tlm_fifo_debug_if : public virtual sc_interface
{
public:
  virtual int used() const = 0;
  virtual int size() const = 0;
  virtual void debug() const = 0;

  virtual bool nb_peek( T & , int n ) const = 0;
  virtual bool nb_poke( const T & , int n = 0 ) = 0;
};
```

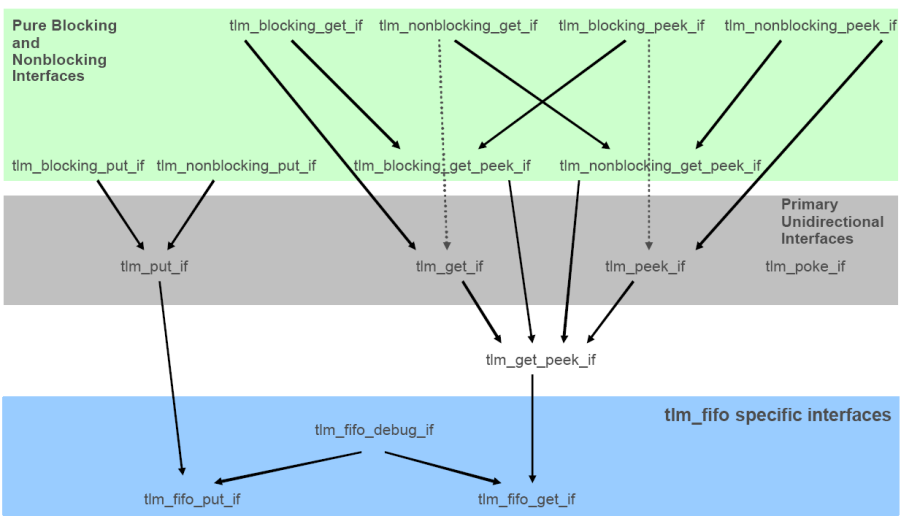
fifo_put

```
template < typename T >
class tlm_fifo_put_if :
  public virtual tlm_put_if<T> ,
  public virtual tlm_fifo_debug_if<T>
{ };
```

fifo_get

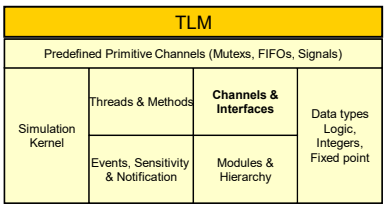
```
template < typename T >
class tlm_fifo_get_if :
  public virtual tlm_get_peek_if<T> ,
  public virtual tlm_fifo_debug_if<T>
{ };
```


Inheritance Diagram of Interfaces



Transaction Level Modeling

- TLM Introduction
- TLM Interfaces
- TLM Channels
- Example



TLM Channel
TLM FIFO Channel

- tlm_fifo<T>

```
template <class T>
class tlm_fifo :
  public virtual tlm_fifo_get_if<T>,
  public virtual tlm_fifo_put_if<T>,
  public sc_prim_channel
{
public:
    explicit tlm_fifo( int size_ = 1 )
      : sc_prim_channel( sc_gen_unique_name( "fifo" ) )
    …

    explicit tlm_fifo( const char* name_, int size_ = 1 )
      : sc_prim_channel( name_ )
    …

}
```

get
```
// tlm get interface
T get( tlm_tag<T> *t = 0 );
bool nb_get( T& );
bool nb_can_get( tlm_tag<T> *t = 0 ) const;
const sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const
…
```

peek
```
// tlm peek interface
T peek( tlm_tag<T> *t = 0 ) const;
bool nb_peek( T& ) const;
bool nb_can_peek( tlm_tag<T> *t = 0 ) const;
const sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const
…
```

put
```
// tlm put interface
void put( const T& );
bool nb_put( const T& );
bool nb_can_put( tlm_tag<T> *t = 0 ) const;
const sc_event& ok_to_put( tlm_tag<T> *t = 0 ) const
…
```

TLM Channels
TLM Request/Response Channel (1/2)

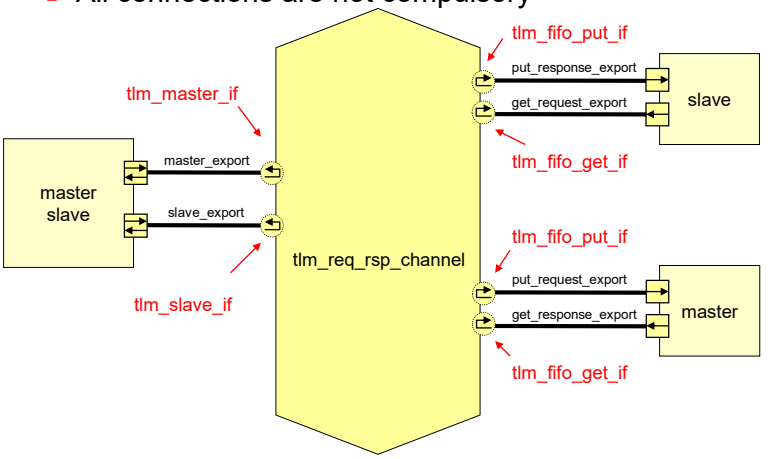
- tlm_req_rsp_channel<REQ, RSP>
  - Bidirectional channel
  - 2 FIFOS

```
template < typename REQ , typename RSP >
class tlm_master_if :
  public virtual tlm_put_if< REQ > ,
  public virtual tlm_get_peek_if< RSP >
{ };

template < typename REQ , typename RSP >
class tlm_slave_if :
  public virtual tlm_put_if< RSP > ,
  public virtual tlm_get_peek_if< REQ >
 {};
```

```
template < typename REQ , typename RSP >
class tlm_req_rsp_channel : public sc_module
{
public:
  // uni-directional slave interface
  sc_export< tlm_fifo_get_if< REQ > > get_request_export;
  sc_export< tlm_fifo_put_if< RSP > > put_response_export;

  // uni-directional master interface
  sc_export< tlm_fifo_put_if< REQ > > put_request_export;
  sc_export< tlm_fifo_get_if< RSP > > get_response_export;

  // master / slave interfaces
  sc_export< tlm_master_if< REQ , RSP > > master_export;
  sc_export< tlm_slave_if< REQ , RSP > > slave_export;


  tlm_req_rsp_channel( int req_size = 1 , int rsp_size = 1 )
  …

  tlm_req_rsp_channel( sc_module_name module_name ,
                       int req_size = 1 , int rsp_size = 1 )
  …
};
```


TLM Channels
TLM Request/Response Channel (2/2)

- Graphical Representation
  - All connections are not compulsory



TLM Channels
TLM Transport Channel (1/2)

- tlm_transport_channel<REQ, RSP>
  - Bidirectional channel
  - Each request is bound to one response
  - One place only

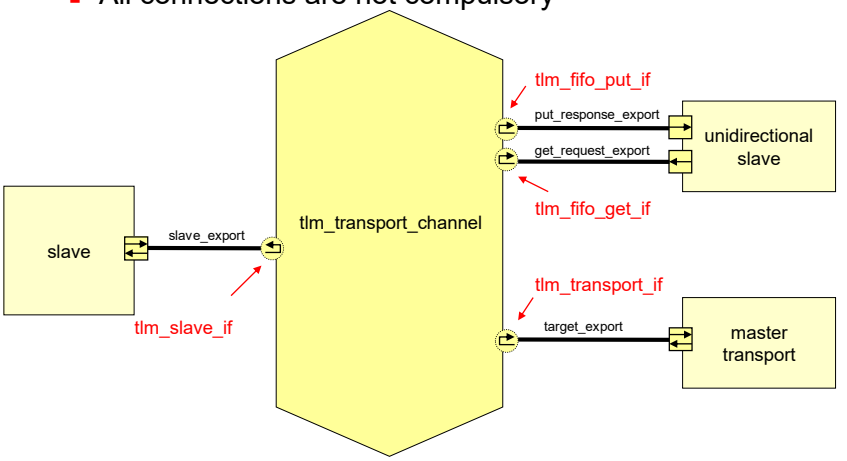
```
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_interface
{
public:
  virtual RSP transport( const REQ & ) = 0;
};
```

```
template < typename REQ , typename RSP >
class tlm_slave_if :
  public virtual tlm_put_if< RSP > ,
  public virtual tlm_get_peek_if< REQ >
{ };
```

```
template < typename REQ , typename RSP >
class tlm_transport_channel : public sc_module
{
public:
  // master transport interface
  sc_export< tlm_transport_if< REQ , RSP > > target_export;

  // uni-directional slave interface
  sc_export< tlm_fifo_get_if< REQ > > get_request_export;
  sc_export< tlm_fifo_put_if< RSP > > put_response_export;

  // slave interfaces
  sc_export< tlm_slave_if< REQ , RSP > > slave_export;

  tlm_transport_channel()
   …

  tlm_transport_channel( sc_module_name nm )
   …
};
```

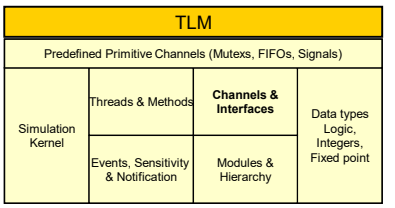

TLM Channels
TLM Transport Channel (2/2)

- Graphical Representation
  - All connections are not compulsory



Transaction Level Modeling

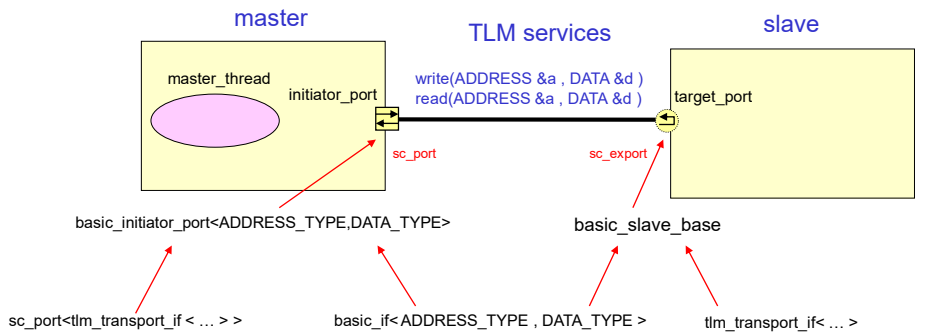
- TLM Introduction
- TLM Interfaces
- TLM Channels
- Example



TLM Layer

User Layer **Protocol-specific "convenience" API** **Targeted for embedded SW engineer** **Typically defined and supplied by IP vendors**	**amba_bus->burst_read(buf, adr, n);**
Protocol Layer **Protocol-specific code** **Adapts between user layer and transport layer** **Typically defined and supplied by IP vendors**	**req.addr = adr; req.num = n;** **rsp = transport(req);** **return rsp.buf;**
Transport Layer **Uses generic data transport APIs and models** **Facilitates interoperability of models** **Key focus of TLM standard** **May use generic fifos, arbiters, routers, xbars, pipelines, etc.**	**sc_port<tlm_transport_if<REQ, RSP> > p;**

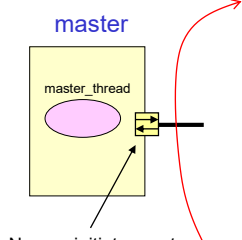
Master / Slave Example
Global View of the example*




```
template< typename ADDRESS_TYPE , typename DATA_TYPE >
class basic_if
{
public:
  virtual basic_status write( const ADDRESS_TYPE &a , const DATA_TYPE &d ) = 0;
  virtual basic_status read( const ADDRESS_TYPE &a , DATA_TYPE &d ) = 0;
};
```

* use of example_3_2 (TLM 1.0) with modifications

Master / Slave Example
Master Side – Interface (1/2)

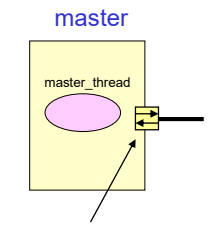



```
template < class ADDRESS , class DATA , int N = 1>
class basic_initiator_port :
  public sc_port<tlm_transport_if <basic_request< ADDRESS , DATA > , basic_response< DATA > > , N> ,
  public virtual basic_if<ADDRESS , DATA >
{
public:
  typedef  tlm_transport_if < basic_request< ADDRESS , DATA > , basic_response< DATA > > if_type;
  basic_initiator_port( const char *port_name ) :
    sc_port< if_type , N > ( port_name )  {}

virtual basic_status write( const ADDRESS &a , const DATA &d )
{
   basic_request<ADDRESS,DATA> req;
   basic_response<DATA> rsp;

   req.type = WRITE;
   req.a = a;
   req.d = d;

   rsp = (*this)->transport( req );
   return rsp.status;
 }

virtual basic_status read( const ADDRESS &a , DATA &d )
{
   basic_request<ADDRESS,DATA> req;
   basic_response<DATA> rsp;

   req.type = READ;
   req.a = a;

   rsp = (*this)->transport( req );
   d = rsp.d;
   return rsp.status;
 } };
```

Master Implementation of basic_if

use of tlm_transport_if

Master / Slave Example
Master Side – Module (2/2)



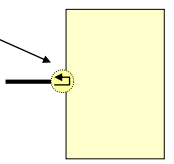
Name : initiator_port
Type : basic_initiator_port

```
class master : public sc_module
{
public:
 basic_initiator_port<ADDRESS_TYPE,DATA_TYPE> initiator_port;

 SC_HAS_PROCESS( master );

 master::master( sc_module_name module_name )
  : sc_module( module_name ) , initiator_port("iport")
 {
  SC_THREAD( master_thread );
 }

 void master_thread()
 {
  DATA_TYPE d;
  for ( ADDRESS_TYPE a = 0; a < 25; a++ ) {
   cout << "Writing Address " << a << " value " << a + 10 << endl;
   initiator_port.write( a , a + 10 );
  }

  for ( ADDRESS_TYPE a = 0; a < 25; a++ ) {
   initiator_port.read( a , d );
   cout << "Read Address " << a << " got " << d << endl;
  }
 }
};
```

use of basic_if
(Master Implementation)

Master / Slave Example
Slave Side – Interface (1/2)

Name : target_port
Type : if_type




```
template< class ADDRESS_TYPE , class DATA_TYPE >
class basic_slave_base :
  public virtual basic_if< ADDRESS_TYPE , DATA_TYPE > ,
  public virtual tlm_transport_if< basic_request< ADDRESS_TYPE , DATA_TYPE > ,
                                   basic_response< DATA_TYPE > >
{
public:
  typedef  tlm_transport_if< basic_request< ADDRESS_TYPE , DATA_TYPE > ,
                             basic_response< DATA_TYPE > > if_type;
  /* Transport Implementation */
  basic_response<DATA_TYPE> transport(const basic_request<ADDRESS_TYPE,DATA_TYPE> &request )
  {
    basic_response<DATA_TYPE> response;
    switch( request.type ) {
    case READ :
       response.status = read( request.a , response.d );
       break;
    case WRITE:
       response.status = write( request.a , request.d );
       break;
    default :
       response.status = ERROR;
       break;
    }
    return response;
  }
};
```

Implementation of tlm_transport_if

use of basic_if
(Slave Implementation, next slide)

Master / Slave Example
Slave Side – Module (2/2)

```
class slave :
  public sc_module ,
  public virtual basic_slave_base< ADDRESS_TYPE , DATA_TYPE >
{
public:
sc_export< if_type > target_port;

slave::slave( sc_module_name module_name , int k ) :
  sc_module( module_name ) ,  target_port("iport") {
  target_port.bind( *this);
  memory = new ADDRESS_TYPE[ k * 1024 ];
}

basic_status slave::write( const ADDRESS_TYPE &a , const DATA_TYPE &d ) {
  cout << name() << " writing at " << a << " value " << d << endl;
  memory[a] = d;
  return basic_protocol::SUCCESS;
}

basic_status slave::read( const ADDRESS_TYPE &a , DATA_TYPE &d ) {
  d = memory[a];
  cout << name() << " reading from " << a << " value " << d << endl;
  return basic_protocol::SUCCESS;
}

private:
  ADDRESS_TYPE *memory;
};
```

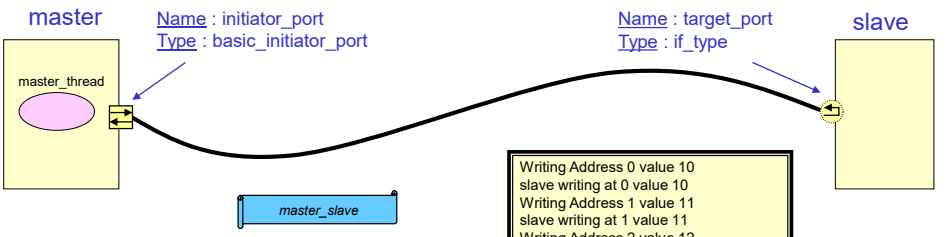
Name : target_port
Type : if_type

slave

sc_export of itself
(slave module)
basic_slave_base inherits of if_type

Slave Implementation of basic_if

Master / Slave Example
Simulation




```
int sc_main( int argc , char **argv )
{
  master master_i("master");
  slave slave_i("slave");

  master_i.initiator_port( slave_i.target_port );
  sc_start();

  return 0;
}
```

```
Writing Address 0 value 10
slave writing at 0 value 10
Writing Address 1 value 11
slave writing at 1 value 11
Writing Address 2 value 12
slave writing at 2 value 12
Writing Address 3 value 13
slave writing at 3 value 13
Writing Address 4 value 14
slave writing at 4 value 14
slave reading from 0 value 10
Read Address 0 got 10
slave reading from 1 value 11
Read Address 1 got 11
slave reading from 2 value 12
Read Address 2 got 12
slave reading from 3 value 13
Read Address 3 got 13
slave reading from 4 value 14
Read Address 4 got 14
```

References

- SystemC 2.2 / TLM 1.0 (http://www.systemc.org)
- Stuart Swan, “Introduction to Transaction Level Modeling in SystemC”, Cadence Design Systems, Inc, 2005
- Transaction Level Modeling in SystemC, Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez, Cadence Design Systems, Inc
- Towards a SystemC Transaction Level Modeling Standard, Stuart Swan, Adam Rose, John Pierce, June 2004
- TLM 1.0 : use of example_3_2