Using OSSS-Channels for HW/HW Communication

OFFIS - R&D Division Embedded Hardware-/Software-Systems

January 18, 2006

1 Introduction

This document can be regarded as a simple "tutorial" which introduces the key concepts and the usage of OSSS-Channels in modelling HW/HW communication. After presenting the key concepts we demonstrate the usage of an OSSS-Channel in a HW/HW point-to-point communication. This introductive example serves as a starting point for the implementation of a simplified I^2C protocol which is specified in the following section. This "tutorial" closes with some exercises which are thought to be helpful in understanding OSSS-Channels in a practical way.

2 The OSSS-Channel Approach

2.1 Key Concepts

The OSSS-Channel concept offers a mechanism to model communication between different modules by method calls. Data transferable by an OSSS-Channel is defined in our object oriented synthesisable subset called OSSS [1, 2]. Unlike channels known from SystemC/TLM which are always point to point connections, our channel approach allows the connection of multiple master and slave modules to a single OSSS-Channel.

The OSSS-Channel concept is based on the separation of port, interface and implementation like it is known from SystemC and encapsulates the user-defined protocols. The separation principle allows the designer to chose from different implementations of a channel as long as they implement the same method interfaces which are used by the modules. This makes it easy to explore different communication protocols for a certain design. This can be further assisted by storing OSSS-Channels in a library of protocol implementations for easy reuse.

The OSSS-Channel provides mechanisms to generate the necessary communication infrastructure between the connected modules. The internal structure consists of transactors, arbiters and a signal level interconnect network. The generation of the internal structure is invoked by the binding of an OSSS-Port of a module to an OSSS-Channel. The transactors offer a method based interface to the connected module and a signal based interface to the signal network inside the channel. Transactors implement the method interface and utilise the signal based interface to describe a signal level protocol. If more than one master is connected to the channel an arbiter handles the requests. The arbitration mechanism is specified by a user defined scheduler and a request/grant transactor.

Synopsis of the OSSS-Channel concept:

- offers a mechanism to model communication between different modules by method calls
- allows to transfer any valid C++/SystemC data types including classes (no pointers)
- offers to write user-defined protocols

- modelling view is based on the separation of port, interface and implementation like it is known from SystemC
- supports the automatic generation of the OSSS-Channel's internal communication network
  - transactor instances
  - arbiter instances with user defined scheduler and request/grant mechanism
  - address decoder with user defined address map
  - signal network including multiplexers
- allows to connect multiple master and slave modules to the same instance of an OSSS-Channel
- uses OSSS-Ports which invoke the generation of the communication network inside of an OSSS-Channel (by the binding operator)
- allows to create an own library of protocol implementations
- has a synthesis semantic

2.2 Single-Master/Slave Channels

As an introductive example, we will present an `osss_basic_channel` performing a simple point to point communication as shown in Figure 1. It consists of two modules (`Camera` and `Filter`) which communicate via an `osss_basic_channel`. The `Camera` initiates write transfers on the channel while the `filter` reads from the channel and consumes the transferred data.

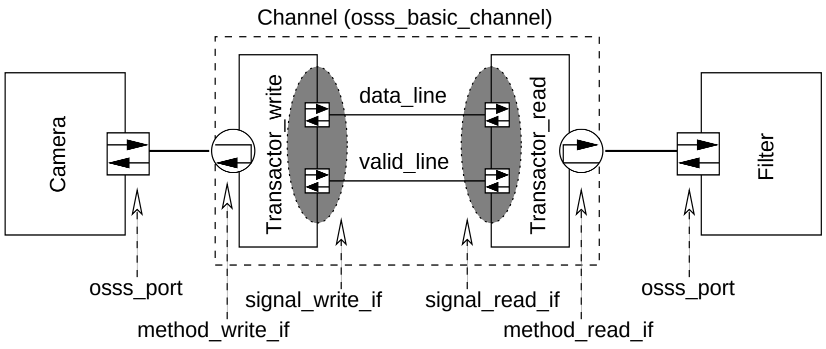


Figure 1: OSSS-Channel performing a simple point to point communication

The top module of the example design shown in Figure 1 can be found in Listing 1.

```
SC_MODULE(Top)
{
  sc_in<bool> clk;
  sc_in<bool> reset;

  Camera *cam;
  Filter *filter;
  Channel channel;

  SC_CTOR(Top)
  {
    cam = new Camera("cam");
    cam->clock(clk);
    cam->reset(reset);
    cam->output(channel);     // binds the cam module to the channel

    filter = new Filter("filter");
    filter->clock(clk);
    filter->reset(reset);
    filter->input(channel);  // binds the filter module to the channel
  }
};
```

Listing 1: Top module

```
SC_MODULE(Camera)
{

  sc_in<bool> clock;
  sc_in<bool> reset;

  osss_port< method_write_if > output; // port to the channel

  SC_CTOR( Camera )
  {
    SC_CTHREAD(main, clock.pos());
    watching(reset.delayed() == true);
  }

 private:

  void main()
  {
    Image i;
    wait(); // for resetting the system

    while(true)
    {
      i.clear();
      i.setImage(); // capture image (fills image object with data)

      output->write(i);
      wait();
    }
  }

};

SC_MODULE(Filter)
{
  sc_in<bool> clock;
  sc_in<bool> reset;

  osss_port< method_read_if > input; // port to the osss-channel

  Filter(sc_module_name moduleName, unsigned int waitCycles = 1)
    : sc_module(moduleName),
      m_waitCycles(waitCycles)
  {
    SC_CTHREAD(main, clock.pos());
    watching(reset.delayed() == true);
  }

 private:

  SC_HAS_PROCESS(Filter);

  void main()
  {
    Image i;
    wait(); // for resetting the system

    while(true)
    {
      input->read(i);

      // do something with the image i

      i.clear();
      wait(m_waitCycles);
    }
  }

  unsigned int m_waitCycles;
};
```

Listing 2: Camera and Filter modules

Listing 2 presents the `Camera` and the `Filter` modules. In this example the OSSS-Ports `output` and `input` of the `Camera` and `Filter` modules are bound to the channel (see Listing 1). Each port is declared using the `osss_port` whose template parameter specifies the abstract method interface for accessing the channel (see Listing 3). The implementation of the abstract interface classes is done in the transactors of the channel. In our example the `method_write_if` has a `write` method and the `method_read_if` has a `read` method.

```
class method_write_if
{
public:
  virtual void write(Image i) = 0;
};

class method_read_if
{
public:
  virtual void read(Image &i) = 0;
};
```

Listing 3: Method interfaces

```
class signal_write_if : public osss_signal_if
{
public:
  sc_out< sc_uint<8> > data_out;
  sc_out<bool> valid_out;

  OSSS_GENERATE {
    osss_connect(osss_reg_port(data_out), osss_shared_signal("data_line"));
    osss_connect(osss_reg_port(valid_out), osss_shared_signal("valid_line"));
  }
};

class signal_read_if : public osss_signal_if
{
public:
  sc_in< sc_uint<8> > data_in;
  sc_in<bool> valid_in;

  OSSS_GENERATE {
    osss_connect(osss_shared_signal("data_line"), osss_reg_port(data_in));
    osss_connect(osss_shared_signal("valid_line"), osss_reg_port(valid_in));
  }
};
```

Listing 4: Signal interfaces

The signal interfaces describe the RTL SystemC ports and the binding to the signal network inside of the channel as shown in Listing 4. The signal level interfaces described in `signal_write_if` and `signal_read_if` establish the low level communication between the modules. The communication protocol is described in the transactors of the channel (see Listing 5). They implement the the `write` and the `read` method by describing how the (serialized) data packets are mapped onto the signal level interface. In our example a user defined protocol with two ports (`data_{in/out}` and `valid_{in/out}`) is chosen, one to manage the control and one to transfer the data. These ports are bound to the channel's communication network in the `OSSS_GENERATE` section. In this section the ports of each transactor are connected by the `osss_connect` method. In our example the `data_out` port is bound to a shared signal called `data_line` which is bound to the `data_in` port in the other transactor. The same kind of point-to-point binding using the shared signal `valid_line` is performed for the `valid` ports of each signal interface. (see Listing 4).

2.2.1 Channel with manual serialisation

```
class Channel : public osss_basic_channel
{
public:

  class Transactor_write : public method_write_if,
                           public signal_write_if
  {
  public:

    virtual void write(Image i)
    {
      valid_out = true;
      wait(2);
      valid_out = false;

      for (int n=0; n<Image::numPixels(); ++n)
      {
        data_out = i.getPixel(n);
        wait();
      }
    }
  };

  class Transactor_read : public method_read_if,
                          public signal_read_if
  {
  public:

    virtual void read(Image &i)
    {
      wait_until(valid_in.delayed() == true);
      wait_until(valid_in.delayed() == false);

      for (int n=0; n<Image::numPixels(); ++n)
      {
        i.setPixel( n, data_in.read() );
        wait();
      }
    }
  };

};

OSSS_REGISTER_TRANSACTOR(Channel::Transactor_write, method_write_if);
OSSS_REGISTER_TRANSACTOR(Channel::Transactor_read, method_read_if);
```

Listing 5: Channel with transactors and manual serialisation

Listing 5 shows the implementation of the channel. It provides two types of transactor classes, one for the `Camera` and one for the `Filter` module. Each transactor implements the method interface and inherits from the signal interface. The transactors describe the communication protocol by implementing the methods of the abstract interface classes. The `Transactor_write` implements the `write` method which sends the control signal and the data via the ports `valid_out` and `data_out` as defined in the `signal_write_if`. The `Transactor_read` implements the `read` method which receives the control signal and the transferred data via his `signal_read_if`.

The read and write transactors inside of the channel are automatically generated and connected to the signal network when an `osss_port` is bound to the channel. Consequently, in this simple example the channel contains a write transactor, a read transactor and a signal network.
A communication is initiated from the `main` process inside the `Camera` module by calling the `write` method on the `osss_port`. It induces the execution of the `write` method which is part of the `Transactor_write` inside of the channel. Therefore the data is transfered via the given signal interface. The `read` method implemented by the `Transactor_read` is invoked by a method call on the `osss_port` of the `Filter` module. The corresponding `read` method in the `Transactor_read` receives the written data on the `data_in` port.

2.2.2 Channel with "automatic" serialisation

A kind of "automatic" serialisation is achieved by making the object which should be transferred via the OSSS-Channel serialisable. I our example presented above the class `Image` is transfered via the channel and serialised in a manual for-loop. This class is made serialisable now by deriving it from `osss_serialisable_object`. Additionally the two pure virtual methods `serialise` and `deserialise` have to be implemented in the `Image` class. Listing 6 shows the serialisable `Image` class which declares the member variables `data` and `test_int` to be serialised and de-serialised.

```
class Image : public osss_serialisable_object
{
 public:

  Image()
  {
    // must be called to initialise the data structures in
    // osss_serialisable_object
    // (used to fill the vector and to know the size of the vector)
    serialiseObj();
  }

  /*
   * the serialise method declares all member variables of this class which
   * have to be serialised
   */
  void serialise()
  {
    // declares the serialisation of the array data of size NUM_BINS
    array(data, NUM_BINS);
    // declares the serialisation of the scalar test_int
    element(test_int);
  }

  /*
   * the deserialise method declares all member variables of this class which
   * have to be deserialised (this are typically the same members which are
   * declared in the serialise method) if multiple member variables are
   * serialised, the sequence of the declarations in the serialise and in the
   * deserialise method need to match
   */
  void deserialise()
  {
    // declares the deserialisation of the array data of size NUM_BINS
    restoreArray(data, NUM_BINS);
    // declares the deserialisation of the scalar test_int
    restoreElement(test_int);
  }

  ...

 private:

  sc_uint<8> data[NUM_BINS];
  int test_int;
};
```

Listing 6: Image class which implements the `osss_serialisable_object` interface

Listing 7 shows the usage of the "automatic" serialisation and de-serialisation of an `Image` object. Serialisation of objects starts with invoking the `serialiseObj` method (de-serialisation in contrast ends with invoking the `deserialiseObj`). After this request the object's attributes are written to a bit vector or a bit vector is initialized to take incoming data. The object is subdivided into chunks which can be submitted or received by the width of the denoted ports. The `writeChunkToPort` or `readChunkFromPort` method is executed in a loop until the serialisation or de-serialisation is completed.

```
class Channel : public osss_basic_channel
{
public:

  class Transactor_write : public method_write_if,
                           public signal_write_if
  {
```

```
  public:

    virtual void write(Image &i)
    {
      valid_out = true;
      wait(2);
      valid_out = false;

      // prepares the object i for serialisation
      i.serialiseObj();

      // writes the whole object i in chunks to the data_out port
      while (!i.complete())
      {

        // writes a chunk of the object i to the data_out port
        // the size of the chunk read from i depends on the
        // size of the port data_out
        i.writeChunkToPort( data_out );

        /* alernative way:
         * reads a chunk of size transfer_type (which is the type of the
         * interface of port data_out) from i and writes it to data_out
         *
         * data_out = i.readChunk< transfer_type >();
         */

        wait();
      }
    }
  };

  class Transactor_read : public method_read_if,
                          public signal_read_if
  {
  public:

    virtual void read(Image &i)
    {
      wait_until(valid_in.delayed() == true);
      wait_until(valid_in.delayed() == false);

      // reads the whole object i in chunks to the data_in port
      while (!i.complete())
      {
        // reads a chunk from the data_in port and stores it in the object
        // the size of the chunk stored in i depends on the size of the
        // port data_in
        i.readChunkFromPort( data_in );

        /* alternative way:
         * reads from the data_in port and writes the data to the object i
         *
         * i.writeChunk( data_in.read() );
         */

        wait();
      }
      // finallises the deserialisation of object i
      i.deserialiseObj();
    }
  };

};

OSSS_REGISTER_TRANSACTOR(Channel::Transactor_write, method_write_if);
OSSS_REGISTER_TRANSACTOR(Channel::Transactor_read, method_read_if);
```

Listing 7: Channel with transactors and "automatic" serialisation

2.3 Multi-Master/Slave Channels

The topic of multi-master/slave channels will not be covered in this "tutorial". For first concepts please refer to [3]. Multi-master/slave channels will be covered in the next "tutorial".

3 Modelling the I^2C Protocol using an OSSS-Channel

3.1 Introduction

To illustrate the use of the OSSS-Channel from a designers point of view, we have chosen the I^2C Bus as a first modelling experiment. The I^2C (**I**nter-**I**ntegrated-**C**ircuit-Bus) Protocol is primarily used for the communication between integrated circuits as the name already implies. The complete specification can be found in [5]. To keep the modelling of the protocol simple we have chosen a simple version[1] with only one bus master. The following short description of the I^2C protocol only considers the aspects which are important for the simplified version of the protocol.

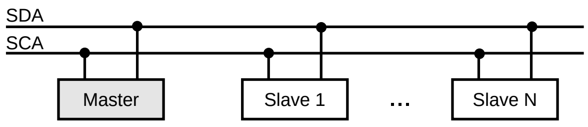


Figure 2: Organisation of the I^2C bus

The physical implementation of the I^2C bus is a bidirectional two wire bus. The SDA (**S**hift **DA**ta) line is used for the transportation of the data. The SCL (**S**hift **CL**ock) line is used for the synchronisation of the data.
In our modelling example only one bus master has to be connected to the bus. It controls the data transfer on the bus. Furthermore multiple slaves can be attached to the bus.
The bus master initiates each communication on the bus and drives the SCL signal, which determines the bus clock. The slaves can send or receive data as requested by the bus master.
While the SCL signal is only driven by the single master the SDA signal is either driven by the master or the slave depending on the direction of the data (the writer drives the SDA line).

MSB							LSB
A_6	A_5	A_4	A_5	A_2	A_1	A_0	R/W

Table 1: I^2C address format

Each slave module has a unique address which consists of eight bits as shown in Table 1. A_6 down to A_0 are the address bits of each slave. The R/W bit is not part of the physical address but it is used by the master to indicate whether a read or a write transfer is initiated.

3.2 The I^2C Bus Protocol

The data transfer on the SDA line is serial (single signal) and synchronous to the SCL signal. When no communication takes place, both signals SDA and SCL are high. The master initiates a transfer with a *start condition* which wakes up all the attached slaves. A *start condition* is characterised by a falling edge of the SDA signal while the SCL signal remains high. The end of a communication is notified by the master performing the *stop condition.* A *stop condition* is characterised by a rising edge of the SDA signal while the SCL signal remains high. Figure 3 illustrates the start and the stop conditions in a waveform.

3.2.1 Data Transfer from Master to Slave

Table 2(a) shows the data transfer from the master to a slave. After setting the start condition the master initiates a write request to one of the slaves by transferring his address on the SDA line. He starts with the MSB, while the last bit indicates a read or a write transfer (setting the R/W bit to zero indicates a

[1]this simplified I^2C bus protocol has been taken from [6]

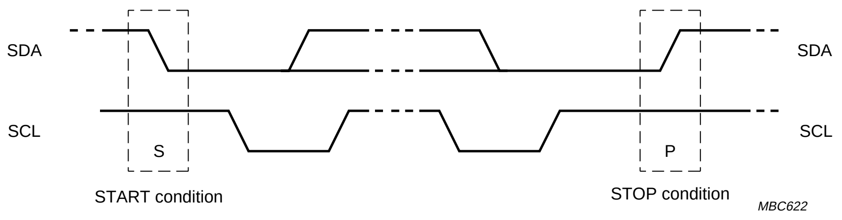


Figure 3: I^{2}C start and stop conditions [5]

write transfer). Since all slaves are woken up by the master's start condition, each slave compares the address with its own. If it matches, the slave acknowledges the reception of the address by pulling down the SDL line synchronous to the next clock event on the SCL line. Now the master sequentially clocks a data byte on the SDA line. The slave acknowledges the reception of the data byte in the same manner as he acknowledged the address reception by pulling down the SDA signal. The transmission finishes when either the slave does not accept the receipt of delivery (acknowledge = 1) and/or the master generates the stop condition.

(a) data transfer from master to slave

S	7 bit slave address	R/W=0	A	8 bit data	A	...	A	8 bit data	A/$\overline{\text{A}}$	P

(b) data transfer from slave to master

S	7 bit slave address	R/W=1	A	8 bit data	A	...	A	8 bit data	$\overline{\text{A}}$	P

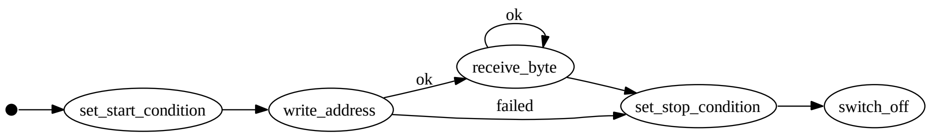
S	start condition	A	acknowledge	$\overline{\text{A}}$	not acknowledge	P	stop condition

Table 2: I^{2}C data transfer

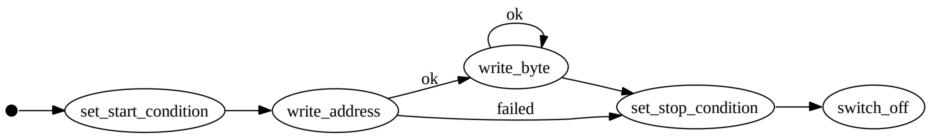
3.2.2 Data Transfer from Slave to Master

Table 2(b) shows the data transfer from a slave to the master. After setting the start condition the master initiates a read request to one of the slaves by transferring his address on the SDA line. He starts with the MSB, while the last bit indicates a read or a write transfer (setting the R/W bit to one indicates a read transfer). Since all slaves are woken up by the master's start condition, each slave compares the address with its own. If it matches, the slave acknowledges the reception of the address by pulling down the SDL line synchronous to the next clock event on the SCL line. Now the slave sequentially clocks a data byte by the SDA line synchronous to the SCL signal. The master acknowledges (acknowledge = 0) the reception of the data byte on pulling down the SDA signal. The transmission finishes when the master refuses the acknowledgment (acknowledge = 1) followed by the generation of the stop condition.

Figure 4 shows the master protocol state machines and Figure 5 shows the slave protocol state machine which have to be implemented in the master/slave transactors of the OSSS-Channel.



(a) `master_receiving_byte`



(b) `master_send_byte`

Figure 4: Protocol state machines for the bus master

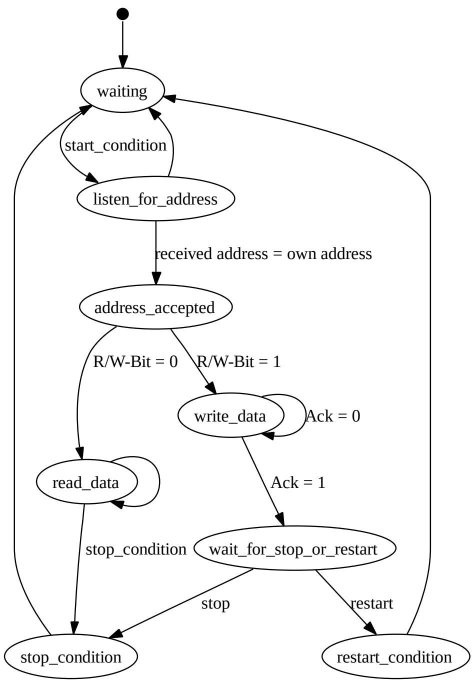


Figure 5: Protocol state machine for the slave (`slave_action_byte`)

4 Exercise

1. Get familiar with the concept of the `osss_basic_channel` as described in subsection 2.2 and [3, 4]
2. Use the `osss_basic_channel` to model a simple point to point communication (single master and single slave) which uses the simplified I^2C protocol described in section 3.
3. Simulate/Execute your design (since synthesis does not work yet).
4. After successfully simulating the single master to single slave communication, extend your system by a second slave.
5. Simulate/Execute your design extended by a second slave.
6. Specify a class which is serialisable (i.e. is derived from `osss_serialisable_object` and implements the `serialise` and the `deserialise` methods) and transfer it via your OSSS-Channel developed above.
7. Simulate/Execute your design and observe the serialisation of the transfered object.
8. Please report any bugs, remarks and modelling experiences.

References

[1] Eike Grimpe and Frank Oppenheimer, *Extending the SystemC Synthesis Subset by Object Oriented Features*, CODES + ISSS 2003, Marriott Hotel, Newport Beach, California, USA, October 2003

[2] Eike Grimpe, Bernd Timmermann, Tiemo Fandrey, Ramon Biniasch and Frank Oppenheimer, *SystemC Object-Oriented Extensions and Synthesis Features*, Forum on Design Languages FDL'02, Marseille, September 2002

[3] Kim Grüttner, Cornelia Grabbe, Frank Oppenheimer and Wolfgang Nebel, *Modelling and Synthesis of Communication Using OSSS-Channels*, 9. GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systeme", Dresden, February 2006

[4] *Specification of HW/SW Communication Design Methodology Based on Abstract Communication Models*, ICODES Deliverable D11, 2005

[5] Philips Semiconductors, *The I^2C-Bus Specification*, Version 2.1, January 2000

[6] Wolfram Putzke-Röming, *Durchgängiges Kommunikationsdesign für den strukturalen, objektorientierten Hardware-Entwurf*, Dissertation, Carl von Ossietzky Universität Oldenburg, April 2001