OCCN Communication Channels for Inter-Module Communication

Marcello Coppola[1], Stephane Curaba[1], Miltos Grammatikakis[2], Giuseppe Maruccia[1]and Francesco Papariello[1]
[1]ST Microelectronics, AST Grenoble Lab, 12 Jules Horowitz 38019 Grenoble, France
Emails: {marcello.coppola,stephane.curaba,giuseppe.maruccia,francesco.papariello}@st.com
[2]ISD S.A., K. Varnali 22, 15233 Halandri, Greece, Email: mdgramma@isd.gr

Abstract

The On-Chip Communication Network (OCCN) project provides an efficient, open-source, GNU-GPL licensed framework, developed within SourceForge for the specification, modeling, simulation, and design exploration of network on-chip (NoC) based on an object-oriented C++ library built on top of SystemC. This document mainly focuses on the implementation, operation and use of point-to-point (StdChannel) and multi-point (StdBus) channels existing in the OCCN library. In addition, the advanced user may be able to exploit current OCCN communication channel development methodology for implementing high-level, specialized on-chip communication protocols, thus increasing NoC design productivity.

1. OCCN Communication Channels for Inter-Module Communication

Communication channels are responsible for inter-module communication, transferring both signals and data according to a given communication protocol (called channel interface). In general, a channel interface may model both direct, point-to-point communication, as well as multi-access channels, such as crossbar, bus, multistage or multi-computer network, thus forming a complex network-on-chip.

The OCCN library currently provides only two channel interfaces: the `StdChannel` that deals with bi-directional, point-to-point communication, and the `StdBus` that deal with traditional, simplified, bus-based communications. For indirect connections among communication routers, such as those required for implementing crossbars, multistage, or direct networks, featuring a number of simultaneous connections among communication nodes, new OCCN models are currently being explored. However, implementation of simplified OCCN communication channels, provides a general SystemC-based inter-module communication design methodology that allows the design of other specialized, user-defined bus channels. For example, Figure 1 illustrates class inheritance for OCCN classes implementing point-to-point (`StdChannel`) and multi-point (`StdBus`) inter-module communication models. OCCN classes are shown in light yellow, while related SystemC classes are shown in green.

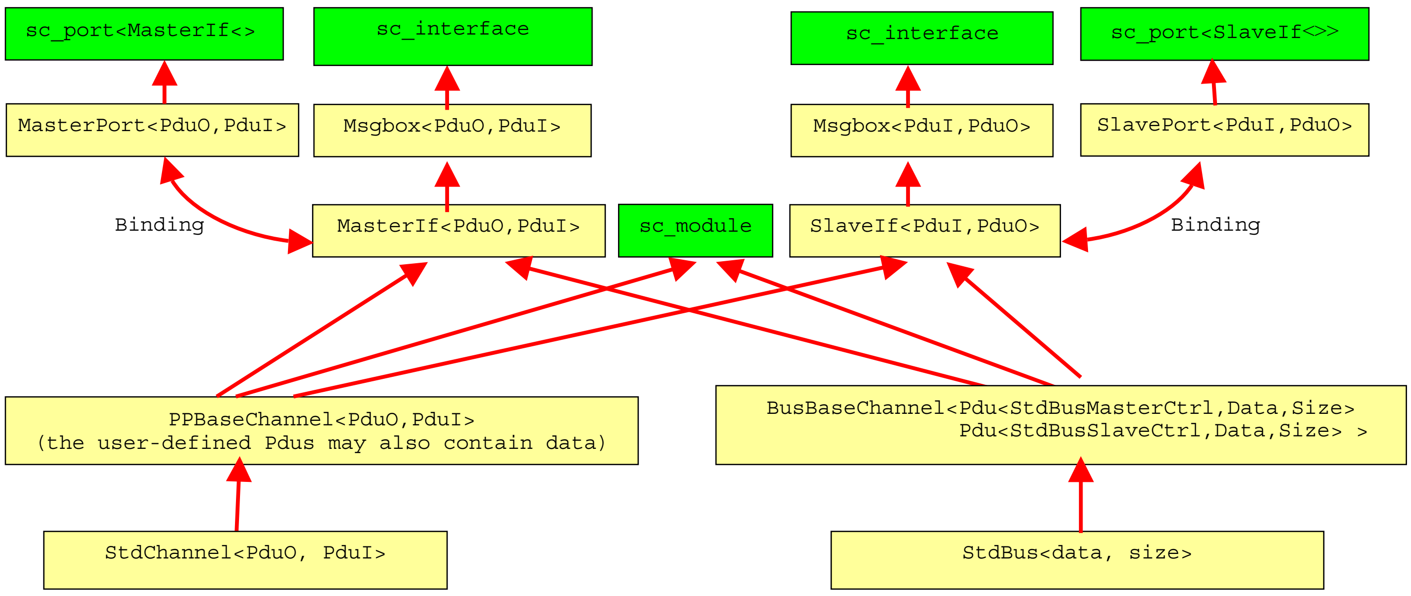

Figure 1. Class inheritance for point-to-point (`StdChannel`) and multi-point (`StdBus`) channels in OCCN.

From Figure 1, notice that OCCN channels are defined as template objects of a pair of

- user-defined Pdu classes, such as classes `PduO` and `PduI` of `StdChannel`,
- channel-specific Pdu classes, or
- mixed classes, such as `Pdu<StdBusMasterCtrl, Data, Size>` and `Pdu<StdBusSlaveCtrl, Data, Size>` of `StdBus`, where only `Data` and `Size` are user-defined.

Thus, in the usual case of Master/Slave communications, each channel can be parameterized through a set of user-defined or channel-specific signals and data coming out of the Master (and going to the Slave), and another set of signals and data coming out of the Slave (and going to the Master). These signals define the Module interface, and may include data, address and operation code fields in the Pdu header.

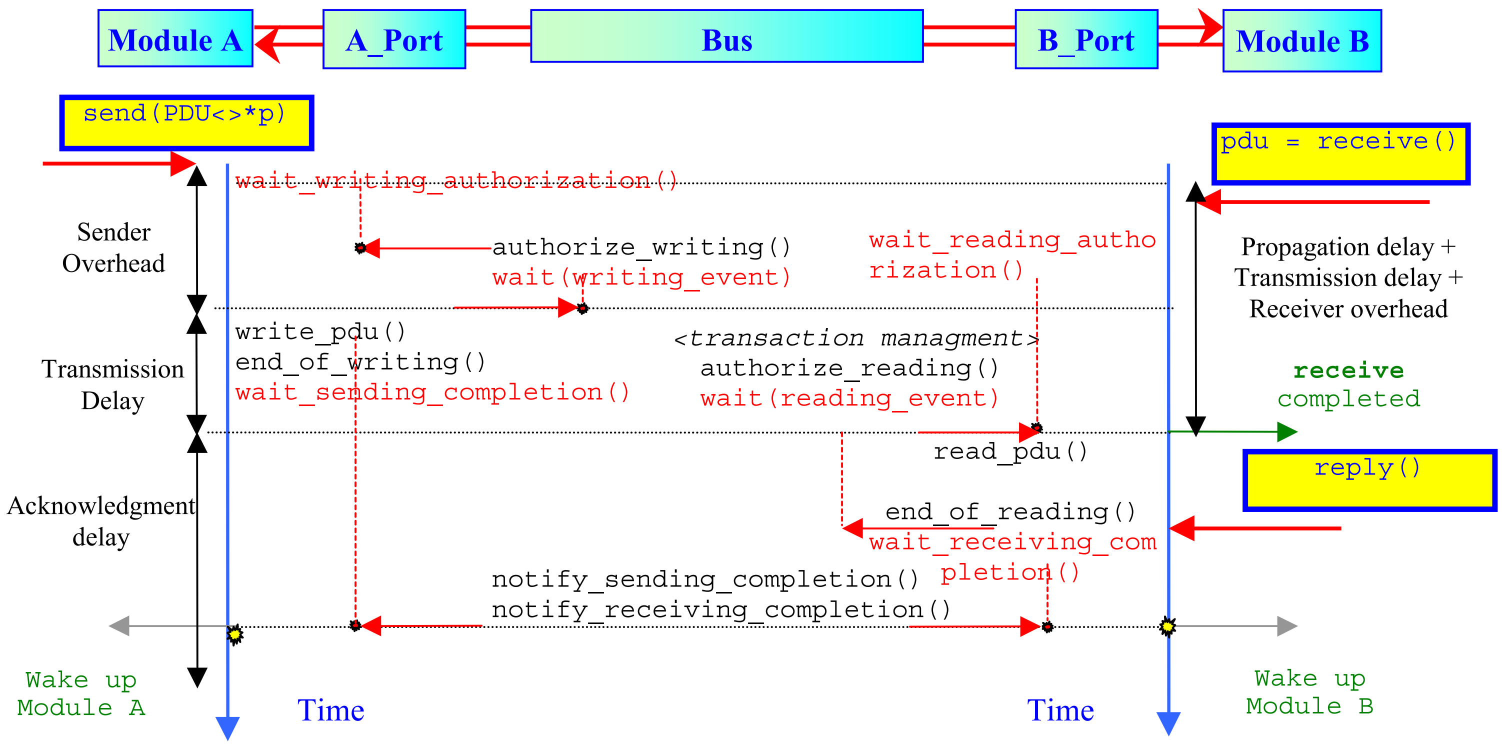

Figure 2. Synchronous inter-module communication using `send` and `receive/reply` in OCCN.

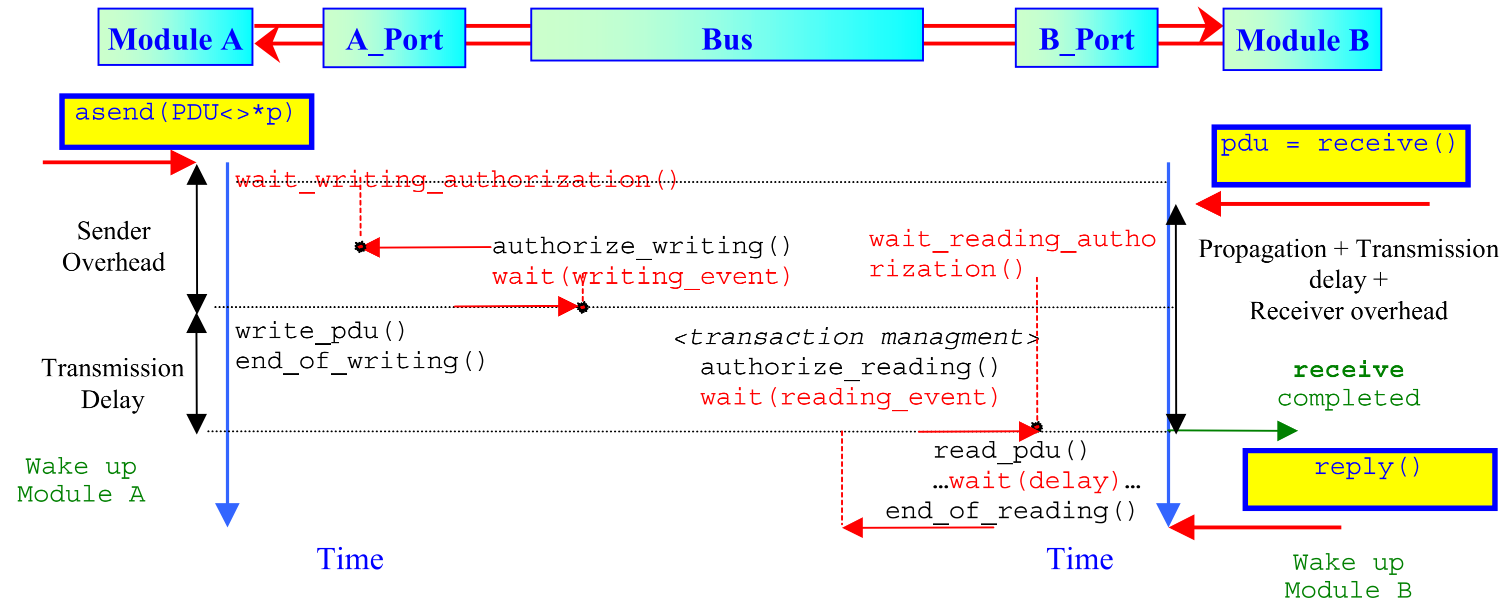

Figure 3. Synchronous inter-module communication using `asend` and `receive/reply` in OCCN.

Next, we proceed to briefly outline implementation issues, concentrating on the operation and use of existing OCCN communication channels. Operation of both `StdBus`, and `StdChannel` is based on the `send/asend` and `receive/reply` routines implemented by the module ports. Implementation of these functions is explained, assuming a bi-directional module A to module B transfer. These transfers are similar to the ones in `StdChannel` and `StdBus`. While in Figure 3, we concentrate on synchronous `send` communications, in Figure 4, we concentrate on asynchronous `asend` communications.

Notice that these figures don't show implementation of `timeout` management. Actually, timeouts are made preemptive by additional testing and implementation of special `cancel_sending` functions that directly translate into channel reset management.

In addition to the previously discussed `send/asend` and `receive/reply` functions, `backdoor_read(size, addr, buffer)` (and `write`) functions allow access to any Slave outside of simulation scope, i.e. simulation time is not advanced and the context is not changed. These functions are useful for loading programs, initializing data, or for debugging purposes, i.e. setting breakpoints or dumps. However, these functions implemented with the module(s) must first be binded to the corresponding port, e.g. using

```
  port->back_door_read_register(&backdoor_read/read, this);
  port->back_door_write_register(&backdoor_write, this);
```

1.1 The StdChannel for Point-To-Point Inter-Module Communication

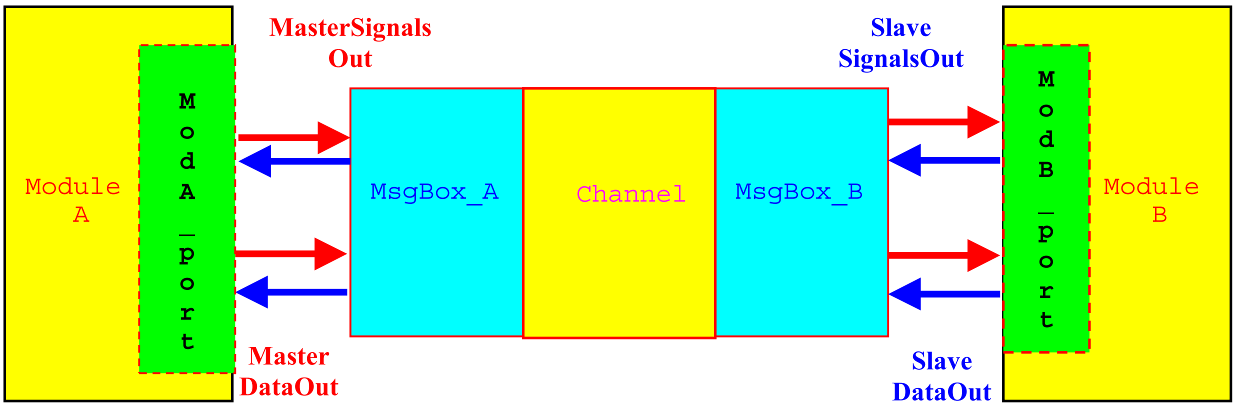

Figure 4. OCCN point-to-point, inter-module communication using the `StdChannel`

As shown in Figure 4, the `StdChannel` (called "Standard Channel interface"), described in "`StdChannel.h`", implements point-to-point communication between two modules (`A` and `B`). The channel allows for synchronous, bi-directional exchange of two Pdu data structures between the two modules. Both Pdu structures (called incoming and outgoing Pdus when referring to a single module) contain user-defined control signals and data, implemented separately and independently for each module. The bi-directional, point-to-point connection between each module and the channel is realized using two interfaces (`ModA_Port` and `ModB_Port`) located in the corresponding modules. These interfaces are compatible but not necessarily the same, e.g. signals and data handled by the ports may be different. Each interface is attached to a Message Box realizing basic functions for implementing control and arbitration for general communication protocols.

Although each module may operate on a different clock frequency, all transfers occur synchronously within the clock environment of the `StdChannel`. Thus, `StdChannel` operation may be described as follows

- Module `A` initiates a transfer using a synchronous send. Then, the `StdChannel` thread `std_process_M_to_S` transfers the Pdu from `Msgbox A` (attached to the `StdChannel` interface connected to module `A`) to `Msgbox B` (attached to the `StdChannel` interface connected to module `B`). Since send is synchronous, Module `A` is blocked until module `B` posts a `receive` command and a positive edge of the internal `StdChannel` clock occurs.

- Once the Pdu is transferred to `Msgbox` B, module B is able to receive the incoming Pdu (control signals and data) by posting a `receive`. Module B also synchronizes with its port by issuing a `reply`.
- Soon afterwards module B prepares its own Pdu (new set of control signals and data), and initiates a transfer towards module A using a synchronous `send`. Then, the `StdChannel` thread `std_process_S_to_M` transfers the Pdu from `Msgbox` B to `Msgbox` A. Module B remains blocked until module A posts a `receive` command and a positive edge of the internal `StdChannel` clock is reached.
- Once the Pdu is transferred to Msgbox A, module A is able to receive the incoming Pdu (control signals and data) by posting a `receive`. Module A also synchronizes with its port by issuing a `reply` and the protocol completes.

Assuming no module delays, e.g. due to delayed `send`, `receive`, or `reply` operation, clock synchronization, arbitration, or congestion, a complete `StdChannel` transaction requires 1 clock cycle in each direction, i.e. two clock cycles for completion of all `StdChannel` protocol communications.

1.2 The StdBus for Multi-Point Inter-Module Communications

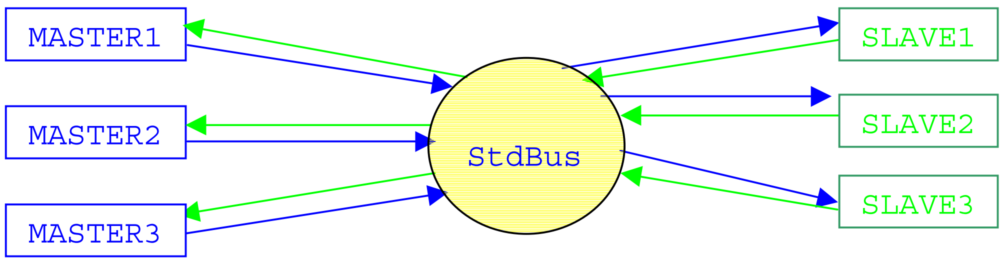

Figure 5. OCCN multi-point, inter-module communication using the `StdBus`.

As shown in Figure 5, Standard Bus interface (described in "`StdBus.h`") implements multi-point, bi-directional communication among several Master and Slave modules by forwarding channel-specific signals and user-defined data (size and type) among successive pairs of Master/Slave modules. For `StdBus`, the Master and Slave signals are **not** user-defined. While the predefined class `StdBusMasterCtrl` is used for connecting any Master module to the `StdBus`, **no** control signals are required by the Slave module, i.e. the corresponding `StdBusSlaveCtrl` class is empty, and thus only user-defined data can be transferred from Slave to Master.

```
class StdBusMasterCtrl {
  public:
    N_uint8  priority;
    N_uint   address;
    N_uint8  opcode;
    N_uint   be;
};
```

A Master module always initiates an StdBus transaction by transmitting (via a synchronous `send`) control signals and data to a particular Slave. Until the corresponding Slave module responds to the selected Master, and the Master acknowledges, no other Master may initiate another `StdBus` transaction to any Slave. Thus, `StdBus` channel implements many-to-many communication, by allowing consecutive locking of channel resources by various one-to-one Master/Slave pairs. Each Master/Slave pair operates based on a simple, high-level, bi-directional, point-to-point inter-module communication protocol using two StdBus port-internal message boxes, identified as `MasterMsgbox` and `SlaveMsgBox`. These ensure the required synchronization between the bus process and the Master and Slave processes.

Although all modules operate on their own clock, `StdBus` transactions occur synchronously within the clock environment of the channel. `StdBus` operation and arbitration principles are described as follows.

- Each Master module may initiate a synchronous `send` transaction request to a specific Slave module by appropriately defining and initializing the corresponding user-defined `data` field, and the following `StdBus`-specific control fields in the Master Pdu:
    - ➢ `opcode`, as either `OCCN_write` or `OCCN_read`,
    - ➢ `address` mapped to the address space of a given Slave; the address space for each Slave is defined at construction time using the port function `set_slave_address_range` (see the example provided later),
    - ➢ packet `priority`, and
    - ➢ `byte enable` defining which bytes within a packet are significant.
- `StdBus` arbitration, implemented as a thread within the `StdBus` channel process, selects a Master with a pending `OCCN_write` or `OCCN_read` request in a non-preemptive way. Selection is based on Master Pdu `priority`, and for equal `priority`, it is based on a simple round-robin order. At this point, `StdBus` is locked and all other Master requests become blocked until the selected communication transaction completes.
- After `StdBus` channel arbitration, the selected Master request becomes blocked, and its transaction fields (control signals and data) are saved to `MasterMsgbox` (attached to the `StdChannel` interface connected to Master modules).
- At the positive edge of the clock that the corresponding Slave is able to receive, information is transferred from `MasterMsgbox` to `SlaveMsgBox` (attached to the `StdBus` interface connected to Slave modules). After this operation, the `StdBus` thread is blocked until the Slave module posts a `receive`, in order to obtain the Master Pdu.
- Once the Slave module is able to obtain the Master Pdu using `receive`, it also synchronizes with the port by posting a `reply`.
- If the opcode is `OCCN_write`, then the corresponding Master module is ready to become unblocked when a positive edge of the internal `StdBus` clock occurs.
- If the opcode is `OCCN_read`, then additionally the following communication pattern is realized.
    - ➢ At first, after sending the acknowledgment, the Slave module prepares and transmits a Slave Pdu containing no control signals but only user-defined data (size and type) using `asend`. Notice that the Slave Pdu may contain different data, i.e. type and size, than the Master Pdu (either could be empty). Since asynchronous communication is used, the Slave module becomes unblocked when the channel, i.e. through `SlaveMsgBox`, obtains the data.
    - ➢ Then, the `StdBus` channel process transfers information from the `SlaveMsgBox` to the `MasterMsgBox`.

- Finally, the Master module is able to obtain the Slave Pdu from `MasterMsgBox` using `receive`. It also synchronizes with its StdBus port by posting a `reply` operation.

We include in Figure 6, as a useful reference for new channel implementations the main `StdBus` thread called `stdbus_process`.

```
template <class Data, int Size>
void StdBus<Data,Size>::stdbus_process() { // StdBus process
  N_int id_initiator = -1
  N_int id_target = -1;

  // enable read/write events for Masters
  for (N_uint i=0; i<masters.get_length(); i++) {
    masters[i]->enable_writing_event();
    masters[i]->enable_reading_event(); }

  // initialization: enable read/write events for Slaves
  for (N_uint i=0; i<slaves.get_length(); i++) {
    slaves[i]->enable_writing_event();
    slaves[i]->enable_reading_event(); }

  do { // infinite thread

    // arbitration: get next request by priority or round-robin
    // if no request exists, then wait for next Master write event
    id_initiator = get_next_request_initiator_id();
    if (id_initiator == -1) {
      wait(*masters_write_ev);
      id_initiator=get_next_request_initiator_id(); }

    // obtain Slave id for the selected Master's address
    id_target = get_slave_id_according_address
   (occn_hdr(*(masters[id_initiator]->get_write_pdu_ptr()), address));

    // wait until corresponding Slave is ready to receive
    if (!slaves[id_target]->is_reading_completed())
      wait(*slaves_read_ev);

    // Master Pdu is transferred to Slave Message Box
    swap_master_pdu(id_initiator,id_target);

    // Master may initiate new transaction, Slave may read Pdu
    masters[id_initiator]->authorize_writing();
    slaves[id_target]->authorize_reading();

    // synchronize between master and slave at end of Pdu reception
    wait(*slaves_read_ev);
    wait(clk.posedge_event());
    masters[id_initiator]->notify_sending_completion();
    slaves[id_target]->notify_receiving_completion();

    //if opcode is READ, then wait until asend is posted from Slave
    //                                   and receive from Master
    if (occn_hdr(*(slaves[id_target]->get_read_pdu_ptr()),opcode)
                                                   == OCCN_READ) {
      // wait until Slave transfers Pdu to Slave Message Box
      if (!slaves[id_target]->is_writing_completed())
        wait(*slaves_write_ev);
      // wait until Master is ready to read
      if (!masters[id_initiator]->is_reading_completed())
        wait(*masters_read_ev);

      // Transfer Pdu to Master Message Box
      swap_slave_pdu(id_initiator, id_target);

      // Slave may initiate new send (not really happening in StdBus)
      // Master may now read the Pdu
      slaves[id_target]->authorize_writing();
      masters[id_initiator]->authorize_reading();

      // Master may re-initiate a transaction, only after it has
      // read the Pdu, and positive edge of clock is reached
      wait(*masters_read_ev);
      wait(clk.posedge_event());
      masters[id_initiator]->notify_receiving_completion(); }
  } while(1);
}
```

Figure 6. Thread controlling `StdBus` channel.

In Figure 7, we connect two Master and two Slave modules through `StdBus`. For Slave modules, the function `set_target_address_range` provides the starting and end addresses that refer to the mapping of global addresses in the `StdBus` domain to a specific Slave module. Notice that no address range overlapping is allowed. Thus, a target may access a Slave module only if the address belongs to this address range. The channel process is able to access the user-defined mapping from an address to a Slave by calling the function `get_slave_id_according_address(addr)`.

```
 int main () {

 sc_clock clk(“main_clk”, 10, SC_NS); // Clock declaration

 StdBus<N_uint32, 1> my_StdBus("StdBus");  // StdBus

 // module instantiations
 ModuleMaster master1("MASTER1");
 ModuleMaster master2("MASTER2");
 ModuleSlave  slave1("SLAVE1");
 ModuleSlave  slave1("SLAVE1");

 // StdBus bindings
 master1.port(my_StdBus);
 master2.port(my_StdBus);
 slave1.port(my_StdBus);
 slave2.port(my_StdBus);

 // Clock bindings
 my_StdBus.clk(clk);

 // Slave address maps
 slave1.port->set_target_address_range(0x0000, 0x3FFF);
 slave2.port->set_target_address_range(0x4000, 0x7FFF);

 // start simulation
 sc_start(-1);
}
```

Figure 7. Modules connected to `StdBus` in `main.cc`.

Finally, in Figure 7, we outline the basic user-defined `StdBus` communication protocol, for modules `A` and `B`, as explained before.

```
Pdu<MyHeader,int> p;
Pdu<int> ret_p;
p = 0xAABBCCDD; // write operation
occn_hdr(p,priority) = HIGH;
occn_hdr(p,address)  = 0x100;
occn_hdr(p,opcode) = OCCN_WRITE;
port.send(p);
// read from same address
occn_hdr(p,opcode) = OCCN_READ;
port.send(p);
ret_p = *port.receive();
if (p != ret_p)
  cerr << 'Data sent != received';
  port.reply(); }
```

```
Pdu<MyHeader,int> p;
Pdu<int> ret_p;
p = *port.receive();
if(occn_hdr(p,opcode)==
                        OCCN_WRITE){
   Mem(occn_hdr(p,address) = p;
   port.reply(); }
else if(occn_hdr(p,opcode)==
                         OCCN_READ){
  ret_p = Mem[occn_hdr(p,address)];
  port.reply(1);
  port.asend(ret_p);
}
```

Figure 8. Initial initiator (Module A) and target (Module B) processes connected to `StdBus`.