Version 2.0 Beta-3
Master-Slave Communication Library

- A SystemC Standard Library

Bugs and Suggestions

Please report bugs and suggestions about this document by going to:

http://www.systemc.org

Contents

Introduction

OSCI welcomes the development of different methodologies based on the infrastructure of the core language. This document outlines the master-slave communication library which is part of the OSCI standard library set for the core language.

SystemC 2.0 is organized as a core language, with the master-slave library and in the future other libraries (e.g., DataFlow) layered on top of the core. OSCI has tried to keep the core SystemC language as simple as possible, with a minimum set of language constructs to describe structure, communication, synchronization, behavior, and so forth. However, the core language does not specify a design methodology, nor is it well suited to address the needs of any particular application domain.

The master-slave communication library is targeted at systems that utilize master-slave bus communication protocols. Systems that consist of one or more processor cores, DSP's, peripheral devices and custom ASICS communicating over a set of buses are particularly well suited for this library. This library provides a proven path from functional-level executable specifications to RTL and enables interface synthesis tools for communication synthesis. The library introduces a sequential execution and communication semantics between processes[1], which is well suited for abstract functional modeling of sequential SW-SW communication, HW-SW interfaces, and HW-HW interfaces that are sequential in nature.

Using this library, complex system models can be built as an interconnection of sequentially communicating functional blocks where unnecessary implementation detail is abstracted. This allows quick model development and trade-off analysis for HW-SW partitioning and resource allocation decisions. In later stages of the design process, functional communication is refined to cycle-accurate bus protocol communication then followed by a refinement of the functional behavior inside the blocks to a cycle-accurate synthesizable form (RTL).

An important property of this model is that functional level abstract communication can be refined to cycle-accurate bus protocol communication while preserving the sequential communication and execution order of the functional level[2]. This property is a key enabler for interface synthesis tools.

1. Not the sequential semantics *within* a process are discussed but the sequential semantics *among* processes.

In the following section, we present modeling abstraction levels and a high-level view of the design flow. The library allows you to separate communication of a module from its internal behavior to a large degree (a complete separation is not always possible).

Benefits of Separation of Communication from Behavior

The two primary benefits of separating communication from behavior are the ease with which you can embed Intellectual Property (IP) blocks into a System on a Chip (SoC) and refine communication protocols.

Embedding of IP Blocks

IP blocks that have been encapsulated in an abstract communication model can be automatically embedded in a system through Interface Synthesis™. This relieves the designer of the details of bus timing, pin mapping, etc. of a particular IP and allows quick switching between alternative IP's.

Refining Communication

SystemC allows you to refine communication between modules by replacing the channels with a more refined model without affecting the communicating modules. For example, you can define an abstract communication protocol at the functional level, and then refine it to a FIFO communication link or to a bus communication. This enables modular design and design re-use.

2. This may seem odd since at the cycle accurate level clocked processes run concurrently. Nonetheless, the state progression in their FSM's preserves the sequential inter-process behavior of the functional level when a conform channel refinement (=order invariant transformation) is applied. Communication resource sharing and other optimizations may however alter this behavior.

Abstraction Levels and a High-Level Design Flow

The following abstraction levels are supported in this library:

- Untimed functional (UTF) level
- Timed functional (TF) level
- Bus-cycle-accurate (BCA) level
- Cycle-accurate (CA) level

Untimed Functional (UTF) Level

At this level, you create an executable specification of the system. The system is decomposed into functional modules that communicate over abstract communication channels in a mostly sequential form. Communication at the functional level is primarily point-to-point, which will later be refined to more complex forms when mapped to the architectural level. This mostly sequential communication is well suited for modeling hardware-software systems that communicate over buses with master-slave protocols. Such systems can usually be described with a few concurrent threads. In each concurrent thread a complex set of sequential communications is modeled among processes that run in the same thread of execution.

At the UTF abstraction level, data transactions and execution order are modeled accurately while time is not modeled. All processes execute in zero time but in a (mostly) defined order.

Timed Functional (TF) Level

At this level, a functional process is assigned a “run time” or duration in absolute time units. This is a useful abstraction for performance modeling, trade-off analysis for hardware-software partitioning, and for resource allocation. The system may not yet be clocked. At the TF level, you model a limited set of resources, which have a finite execution time.

A system may be described as an interconnection of timed and untimed processes.

The process execution order of UTF is not changed when the model is refined to TF. The TF transformation merely maps the execution order of UTF onto a time axis.

A process run time may reflect different things:

- A system constraint on the execution time of a process.
- An estimation for performance modeling.
- A time budget.

In SystemC, you assign a run time to a process through wait(delay) statements which put a process to sleep for a time duration of delay. The process continues execution after this delay.

Hardware-Software Partitioning

Design exploration, performance modeling and analysis together with hardware (HW) - software (SW) partitioning are done at the functional level since this is much easier and quicker at the functional level than at detailed implementation levels. After functional design, HW and SW modules are mapped to the chosen system architecture. SW modules are partitioned into tasks and inter-task communication and synchronization is implemented using a Real Time Operating System (RTOS). The SW flow in SystemC is currently under development. Therefore we will focus the rest of this discussion on the HW flow.

In the HW flow, the communication protocols between modules will be refined and functional modules transformed into cycle-accurate modules. This takes place over different design steps with a gradual refinement of abstraction levels. A mixture of abstraction levels is possible whereby some modules can be at a functional level while others are at a bus-cycle-accurate or RTL levels.

Bus-Cycle-Accurate (BCA) Level

At the BCA level, the communication between HW-SW and HW-HW processes in the system are modeled cycle-accurately with bus protocols, but process behavior may still be modeled at a functional level.

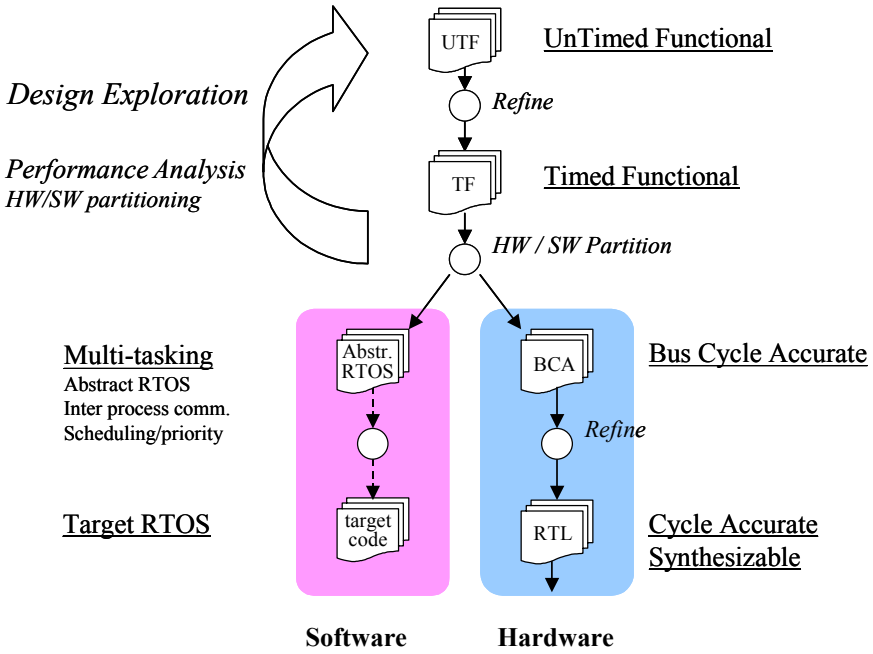
Examples of bus protocols are no-handshake, enable-handshake, and full-handshake protocols.

Cycle-Accurate (CA) Level

At the CA level, all hardware processes are now refined to the Register Transfer Level (RTL), which can be automatically synthesized to the gate level.

The SystemC design flow is shown in Figure 1.

FIGURE 1. SystemC Abstractions and Design Flow



Key Modeling Paradigms

The master-slave library is based on the following key modeling paradigms:

- A functional level abstract communication channel (sc_link_mp<T>), which defines a sequential (in-lined) execution and communication semantics between processes. These processes connect through abstract master-slave ports to the

sequential channel. As we will see, this communication model forms the basis for a model of re-use for sequential inter-process communication and execution.

- A concurrent communication and execution paradigm. Concurrent processes synchronize over events[1] or signals and communicate over signals and shared variables.
- A model for combining the above sequential and concurrent semantics in hierarchical channels.
- At the BCA level, abstract ports of the functional level are refined to bus ports, which are specialized by a bus protocol template parameter. A bus port is a hierarchical entity that groups together the specific terminals for data, address, and control signaling of a bus protocol.

In the following, we first introduce the sequential execution semantics of sc_link_mp<T> together with master-slave port classes at the functional level. Then the combined sequential-concurrent communication model is discussed. In the next section, refinement of functional level sequential communication to cycle-accurate bus protocol communication is presented.

Functional Level

We have seen how SC_METHOD, SC_THREAD, and SC_CTHREAD processes define a concurrent execution and communication semantics for RTL level communication with signals. This level of communication is too detailed and not suitable for functional modeling.

In this chapter, we will discuss an abstract communication and execution semantics, which is more appropriate for functional modeling. It is based on a new process type, called SC_SLAVE whose member method, which implements its behavior, can be invoked like a function from another process.

1. Events are high-level synchronization objects in SystemC. They should not be confused with gate-level events, which occur within a clock-cycle boundary. SystemC is designed for higher-level abstractions (RTL and functional level).

Model of Re-Use for Sequential Inter-Process Communication and Execution

Let's introduce the concepts with the following example of C code in which a function generate_data() produces a set of integer numbers and for each number calls the accumulate() function that accumulates the numbers and prints them out.

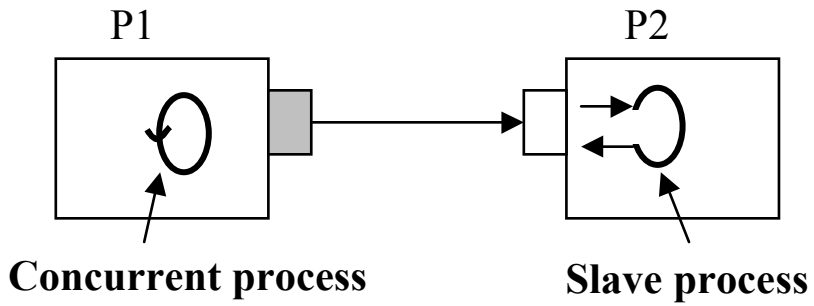
```
void generate_data (int &out)
  {
    for (int i = 0;  i < 10;  i++)
      {
        accumulate(out);
      }
  }

void accumulate (int &in1)
  {
    sum += in1;
    cout << "Sum =    "  << sum << endl;
  }
```

For re-usability of this sequential behavior, we'd like to partition the two functions into separate processes and encapsulate each in a module as shown in Figure 2. The intent is that this structural decomposition does not change the behavior of the original sequential C code. We have created a producer and a consumer module which encapsulate the generate_data() and accumulate() processes respectively. Since the two processes reside in separate modules, generate_data() can no longer invoke accumulate() directly[1]. Instead it writes to its master-output port which through the sequential channel sc_link_mp<T> invokes accumulate(). Through this structural encapsulation the two modules can now be re-used independently. The caller process connects through a master port (gray box in Figure 2) and the slave process through a slave port (white box in Figure 2) to the link. We have now defined the re-use model for sequential inter-process execution.

1. A direct invocation would violate the principle of encapsulation and prevent independent re-use of the two modules. The principle of encapsulation requires that a process communicates with the external world outside the module exclusively through its module ports.

In Figure 2, by convention a concurrent process is shown as an oval with an arrow while a slave process is an open oval with an ingoing and an outgoing arrow indicating that it is invoked by another process and returns after execution.

FIGURE 2. In-Lined Execution



Structural decomposition of sequential behavior has the following advantages:

- Modular re-use of sequential behavior
- HW-SW partitioning at module boundaries
- Refinement of sequential communication channels with bus protocols

The SystemC source code for the producer-consumer example is given below.

```
SC_MODULE(producer)
{
  sc_outmaster<int> out1;   // master port
  sc_in<bool>   start;  // to kick-start the producer

  void generate_data ()
    {
      for (int i = 0;  i < 10;  i++)
        {
          out1 = i ;// this will invoke the slave;
        }
    }

  SC_CTOR(producer)
    {
      SC_METHOD(generate_data);  // concurrent process
      sensitive << start;
    }
```

```
};

SC_MODULE(consumer)
{
  sc_inslave<int> in1;  // slave port
  int sum;// declare as a module state variable

  void accumulate ()
    {
      sum += in1;
      cout << "Sum =    "  << sum << endl;
    }

  SC_CTOR(consumer)
    {
      SC_SLAVE(accumulate, in1); // slave process
      sum = 0;// initialize the accumulator
    }
};

SC_MODULE(top) // structural module
{
  producer *A1;
  consumer *B1;
  sc_link_mp<int> link1;

  SC_CTOR(top)
    {
      A1 = new producer("A1");
      A1.out1(link1);
      B1 = new consumer("B1");
      B1.in1(link1);
    }
};
```

In-Lined Execution Semantics[1] of the Sequential Channel sc_link_mp<T>

A slave process has a single slave port through which its behavior is invoked from a caller process. The slave method executes in-line with the caller process and returns to the caller after execution. The caller process is connected through a master port to a sequential channel sc_link_mp<T> that connects to the slave port of the slave process. The caller invokes the slave method by writing to (reading from) its master port a data value of type <T> if it is an output (input) port. The read or write in the caller blocks until the slave returns. A caller can read from (write to) the same input (output) master port more than once in a given execution.

The caller process can be of any type including slave and can have multiple master ports through which it can invoke other slaves. This can lead to in-lined execution chains with complex data dependent behavior. In our experience, complex systems can be described with only a few concurrent threads of execution with each thread having its own complex in-lined execution chains.

A slave process may read the data value if it has an input slave port and may write a data value if it has an output slave port, only once per invocation. Multiple reads or writes in a given slave invocation are illegal and will result in run time warnings. Read and write methods in a slave are non-blocking. The data value on the channel is immediately available to the other process, that is, sc_link_mp<T> does not follow the evaluate-update semantics discussed in the User Guide.

The last value written to an sc_link_mp<T> is stored until it is overwritten.

Processes in a module may share abstract ports. A process can access abstract- and signal-ports.

The connection over sc_link_mp<T> allows the caller to invoke the slave method without knowing its method pointer. This is essential for module re-use since a module behavior should not depend on its external connectivity.

1. In-lined execution is also called Remote Procedure Call (RPC), which is a term borrowed from Unix with similar semantics. Each concurrent thread defines a thread of execution. In-lined execution is equivalent to “Executing in the same thread of execution.”

We defined above the sequential execution semantics of point-to-point communication. In a multi-point channel, multiple master and slave ports can be inter-connected. Its semantics are described in “Sequential Execution Semantics in a Multi-Point Communication (sc_link_mp)” on page 15.

Fundamental Components of a Communication

Master-slave ports[1] codify in explicit form the fundamental (= orthogonal) components of a sequential communication in the port type and the port template parameters. These components are:

- Direction of a communication (in, out, inout)
- Initiator of a communication (master or slave)
- Data type of the data that is communicated
- Optionally a transaction index (= abstract address)
- Attributes of a communication channel for configuring a channel or a transaction. For example, to configure a communication for blocking with time-out or for non-blocking communication.
- Bus protocol at the bus-cycle-accurate (BCA) level

This explicit identification of the different components of a communication expresses communication intent and is a key enabler for interface synthesis tools.

Example:

```
    sc_inmaster<T>  my_abstract_port;  // abstract port
// the same port refined into a bus port with the
// enable-handshake protocol
    sc_inmaster< T, sc_enableHandshake<T> >  my_bus_port;
```

Master-Slave Port Syntax

Master and slave ports are specialized port classes. At the functional level, they are called abstract ports since the communication is abstract at this level.

1. Master and slave ports as a group are called abstract ports at the functional level and bus ports at the BCA level.

There are eight abstract port types. <T> is the data type that is communicated over the port.

```
sc_master<>, sc_inmaster<T>,
sc_outmaster<T>, sc_inoutmaster<T>

sc_slave<>, sc_inslave<T>,
sc_outslave<T>, sc_inoutslave<T>
```

where <T> can be any of the C++ built-in types (long, int, char, short, float, double, and so forth) or any of the SystemC types (sc_int<N>, sc_uint<N>, sc_bigint<N>, sc_biguint<N>, sc_bit, sc_logic, or fixed-point types), or a user-defined data type.

Examples of abstract port declarations:

```
sc_inmaster<int>              inp1;
sc_outmaster<sc_bigint<128> > out1;
sc_inoutslave<sc_lv<8> >      inout1;
```

Examples of Read and Write Methods of Abstract Ports

Read/write methods are supported in implicit and explicit form as illustrated below.

```
sc_inmaster<int>              inp1;
sc_outmaster<sc_bigint<128> > out1;

{ // a caller process
    int ival;
ival = inp1;
// blocking read from input port & invoke slave
ival = inp1.read();
// blocking read (explicit) & invokes slavesc_bigint<128>
bigival;
bigival = 1000;
// blocking write to output port & invoke slaveout1 = bigival;
// blocking write (explicit) & invoke slave
out1.write(bigival);

}
```

Control Flow Abstract Ports

Two specialized port classes, sc_master and sc_slave, only convey control flow without data flow direction or data type. An example would be to model an interrupt.

Examples:

```
sc_master<> mport;

sc_slave<>  sport;
```

In the calling process, a transaction is invoked by calling the function method on the master port, which will invoke the slave process connected to the master port. For the above example this would be:

```
mport();
```

It is illegal to write to or read from 'directionless' ports. Data type can be specified optionally but will have no effect.

sc_link_mp Syntax

Syntax:

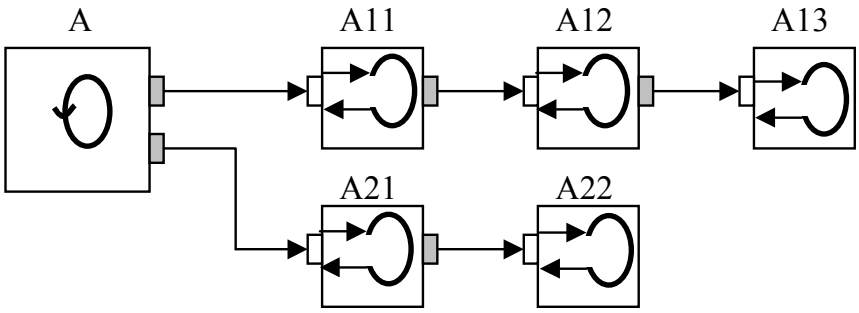
```
sc_link_mp<T> link1;
```

where <T> is the data type communicated over the link. Ports connected to a link must have the same data type as the link. sc_link_mp<T> can only bind to abstract ports.

Example: In-Lined Execution (RPC) Chains

Two RPC-chains each attached to a master port of a concurrent process are illustrated in Figure 3.

FIGURE 3. RPC Chain Example



The execution order in this example is A, A11, A12, A13, A21, and A22 in the assumption that the top most master port is accessed first.

RPC chains cannot form a loop since this would require a slave process to have more than one slave port.

Waiting in Slave Processes

A slave process does not have a static sensitivity list of its own. A slave process can synchronize with events and signals through embedded wait() statements (with static and dynamic sensitivity) provided that the slave process runs in-lined in a thread process. Otherwise, you will get a run-time warning. Statically sensitive wait()'s (those without an argument list) in a slave are sensitive to the sensitivity list of the thread in which it executes in-lined.

Slave Process Syntax

The slave process takes two arguments: *slave_method*, which implements the behavior of the slave process and *slave_port*, which must be an abstract port. slave_method must be a member method of the module class.

```
SC_CTOR(my_mod)
{
  // module constructor
  ...
  SC_SLAVE(slave_method, slave_port);
  // defines a slave process with its unique slave port;
  ...
}
```

Sequential Execution Semantics in a Multi-Point Communication (sc_link_mp)

In a multi-point (mp) communication, a master process communicates with multiple slaves as follows:

- A master performs an access to its master port, which will invoke all compatible slaves. Compatible slaves are explained below.
- Slaves are invoked sequentially but in an undefined order.
- Each slave executes and returns. Then, the next slave is invoked in a depth-first order.
- If a slave blocks at a wait(), then the remaining slaves in the list block as well because of the depth-first ordering.

Two concurrent processes accessing the same link over master ports access the link in a sequential but undefined order. This allows abstraction of any relationship of simultaneity between the two concurrent processes.

The sc_link_mp<T> object does not provide built-in arbitration or prioritization functions. It is a primitive (atomic) object that allows abstraction of such refinements. Complex bus models with bus arbitration and prioritization schemes can be built by composition from primitive links.

On a write operation to a sc_outmaster or sc_inoutmaster port, all slaves connected to the link with sc_inslave or sc_inoutslave ports will be invoked and may (optionally) read the data value on the link. This constitutes a broadcast communication.

On a read from an sc_inmaster or a sc_inoutmaster port, all slaves connected to the link with sc_outslave or sc_inoutslave ports will be invoked but one and only one must respond with a write operation to its slave port. Otherwise, a run-time warning will be generated and the last value written to the link will be retained (undefined

condition). The caller avoids clashes by providing an index (=address) with the transaction (see “Indexed Ports” on page 21).

A write to an input (master or slave) or a read from an output (master or slave) port is illegal which will result in a run-time warning.

The new value assigned to an abstract link is visible immediately[1] to all processes connected to the link.

Concurrent and Sequential Communication Combined in a Hierarchical Channel

A process may communicate over abstract and signal[2] ports. This forms the basis for combining concurrent and sequential master-slave communication in hierarchical channels.

The concurrent communication and execution model has been extensively discussed in the User Guide. For the sake of clarity, we will give here a brief summary of its semantics.

Concurrent processes have means for synchronization and means for data communication. Concurrent processes synchronize over events[3]. An event is the essential synchronization object in SystemC. A process can be made statically or dynamically sensitive to an event as discussed in the User Guide. Two concurrent processes communicate a data value essentially in two ways:

- Over shared variables when both processes reside inside the same module. A shared variable is a data member of a module. It has an immediate-update semantics. A shared variable should be protected by a mutex[4] if there is a risk of

1. It is possible to write non-deterministic behavior due to this immediate update semantics of sc_link_mp. In our opinion this does not take away from the usefulness of this sequential paradigm. Non-determinism is also a fact of life in multi-threaded programming, which did not prevent its widespread adoption. Non-determinism is avoided through careful design.
2. Ports (sc_in, sc_out, sc_inout) that connect to signals are called signal ports.
3. Events are pure synchronization objects. Signals additionally carry a data value.
4. SystemC is not pre-emptive. Therefore data corruption with unprotected access to shared memory cannot be modeled.

data corruption when two concurrent processes want to write its value simultaneously.

- Over signals when the two processes reside in the same or different modules.

If two processes, which reside in different modules, want to communicate over shared variables, then a hierarchical channel[1] must be used between them that contains the shared variables. A hierarchical channel is not a new type; it is a module. It is called hierarchical because it can have structural hierarchy and behavior. The two concurrent processes access the shared variables in the channel through slave processes of the channel. This is illustrated in the FIFO example shown in Figure 4.

In-lined (RPC) communication between two threads of execution is per definition not possible.

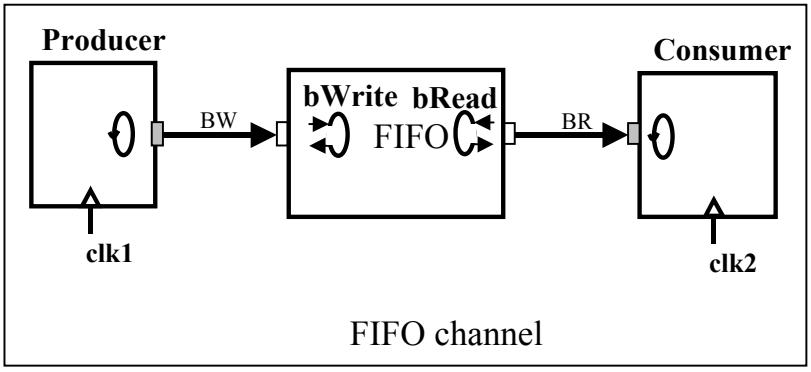
Example of a Hierarchical Communication Channel

An example of a hierarchical channel is depicted in Figure 4 where a producer and a consumer are concurrent processes, which communicate over a concurrent FIFO channel. The FIFO implements a blocking write (bWrite) and a blocking read (bRead) slave process to write to and read from the FIFO buffer. The buffer inside the FIFO is a shared variable object that the two slave processes access. BW and BR are sequential channels for in-lined execution. If the producer tries to write into the FIFO buffer while it's full, then the bWrite process will block the producer pro-

1. A signal is a special form of a concurrent channel. It has an evaluate-update semantics which models delta-delay propagation in a cycle-accurate hardware modeling abstraction.

cess until the bRead process frees up memory in the buffer. Similarly when the consumer attempts to read from an empty FIFO buffer.

FIGURE 4. Hierarchical FIFO Channel



Connectivity Rules and Examples

This section can be skipped without loss of continuity.

Point-to-Point (p2p) Communication Connectivity Rules

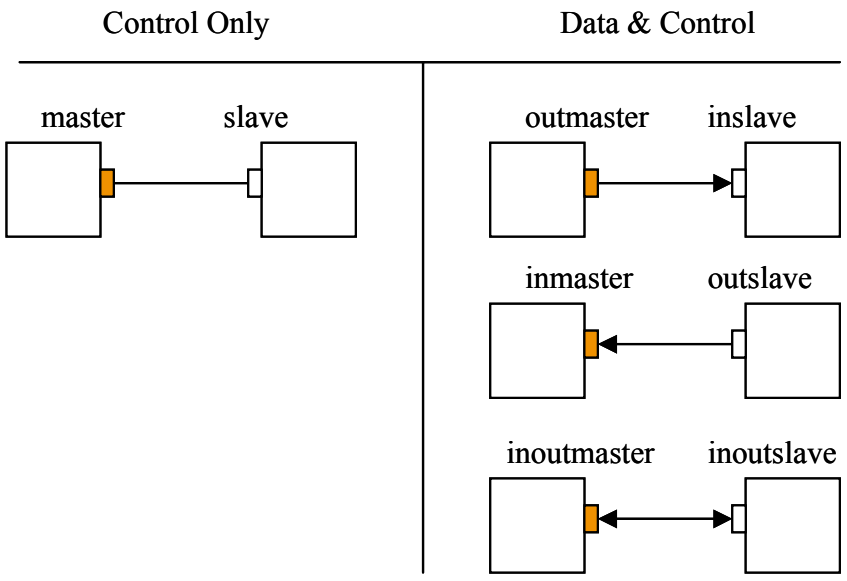
Abstract ports can be used in the following combinations in a p2p link:

```
sc_master -      --- sc_link_mp ---            sc_slave
sc_outmaster     --- sc_link_mp ---            sc_inslave
sc_inmaster      --- sc_link_mp ---            sc_outslave
sc_inoutmaster   --- sc_link_mp ---            sc_inslave
sc_inoutmaster   --- sc_link_mp ---            sc_outslave
sc_outmaster     --- sc_link_mp ---         sc_inoutslave
sc_inmaster      --- sc_link_mp ---         sc_inoutslave
sc_inoutmaster   --- sc_link_mp ---         sc_inoutslave
```

A master and a slave port must always be used as a pair at both ends of a p2p communication link. You need to match an output port with an input port. A pair of two

masters, two slaves, two outputs, or two inputs at the opposite ends of a p2p link is illegal. Figure 5 illustrates some legal p2p connections.

FIGURE 5. Some Legal p2p Connections



Multi-Point Link Connectivity Rules

The following connectivity rule is enforced for multi-point connections: with each master port connection, there should at least be one slave connection that can respond to the master. This rule is statically checked at the start of a simulation.

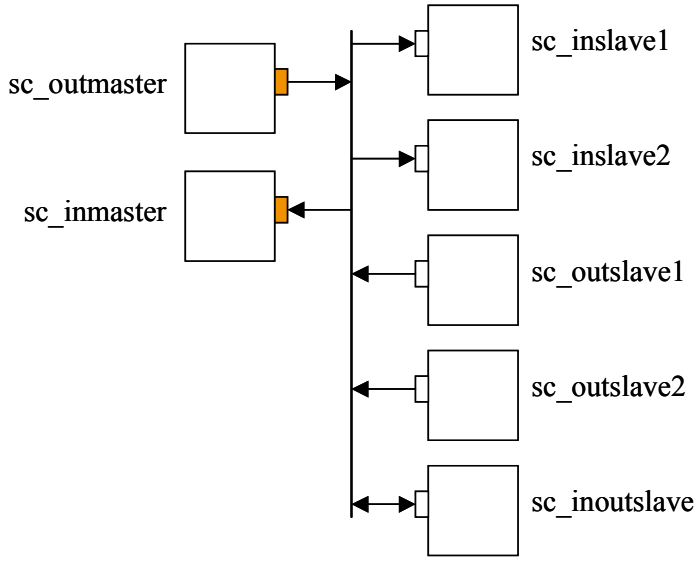
In practice, this means the following:

- With an sc_master port in a link there must at least be one sc_slave connection.
- With an sc_inmaster port in a link there must at least be one sc_outslave or one sc_inoutslave connection.
- With an sc_outmaster port in a link there must at least be one sc_inslave or one sc_inoutslave connection.
- With an sc_inoutmaster port in a link, there must at least be one sc_inslave and one sc_outslave or one sc_inoutslave connection.

Multi-Point Link Example

A multi-point link is illustrated in Figure 6, where two concurrent processes are connected to multiple slaves.

FIGURE 6. Multi-Point Link Example

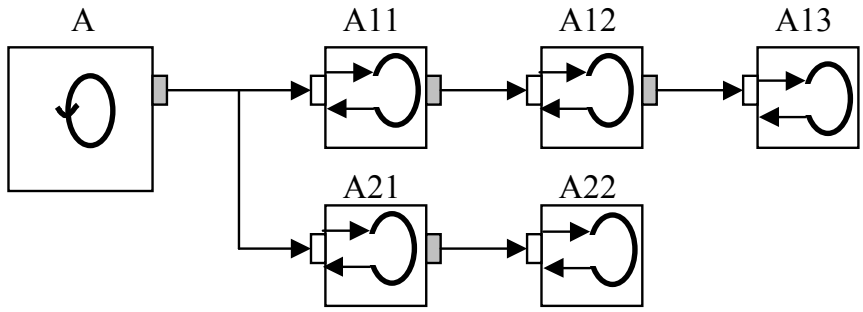


When the sc_outmaster writes to its sc_outmaster port, all sc_inslave and sc_inoutslave processes will be invoked. In this case, sc_inslave1, sc_inslave2 and sc_inoutslave will be invoked. Similarly a read operation from an sc_inmaster port will invoke all sc_outslave and sc_inoutslave processes. In the example below, sc_outlsave1, sc_outslave2 and sc_inoutslave will be invoked. However, only one slave should write a value to its output port.

Example: Execution Order in a Multi-Point Link

In Figure 7, we show two RPC chains A, A11, A12, A13 (chain1) and A, A21, A22 (chain 2) in a multi-point link. Valid execution orders are (depth-first): A, A11, A12, A13, A21, A22 or A, A21, A22, A11, A12, A13 .

FIGURE 7. Execution Order in a Multi-Point Link



Abstract Port Classes Detailed

This section can be skipped without loss of continuity.

Port Arrays

Arrays of ports are defined using the C array syntax. For example, the following is an array of sc_inmaster ports of type int, with 10 elements.

```
sc_inmaster<int> P1[10];
```

Indexed Ports[1]

Master and slave processes can perform data transactions that have an address, also called an index. Using this mechanism, a master process can write to or read from an address in a memory block in a slave process. Indexed ports are declared with

1. Indexed ports are an abstraction at the functional level. They do not exist at the BCA/CA levels.

an integer valued range parameter that specifies the upper limit of the address range, which always starts from 0[1]. Master and slave ports in a link must have matching address ranges. The master specifies the index value of a transaction as a master port index, while the slave reads this value using the get_address() method of its slave port. You cannot specify the index of a transaction at a slave port. It assumes the index of the master transaction. Access to an index outside the range is illegal and will result in a run time error.

Syntax:

```
sc_outmaster<T, sc_indexed<range> >port;

sc_inslave<T, sc_indexed<range> >   port;
```

Ports of type sc_master and sc_slave do not take a range parameter.

For example, you define an sc_outmaster port P2 of type int with an address range of (0,1023) as follows:

```
sc_outmaster<int, sc_indexed<1024> >P2;
```

The following illustrates an indexed write at address 68 on sc_outmaster P2.

```
SC_MODULE(my_master)
{
  sc_outmaster<int, sc_indexed<1024> > P2;
  int data;
  ...
  { // body of master process tied to port P2
    ...
    data = 10;
    P2[68] = data;
    ...
  }
};
```

In the following example, assume port pslave of my_slave is connected to port P2 above. The result of this transaction is that the value 10 is written into memory location 68.

```
SC_MODULE(my_slave)
```

1. This will change in a future release. You will then be able to specify a different range with each slave port with a non-zero lower limit.

```
{
  sc_inslave<int, sc_indexed<1024> > pslave;
  int memory[1024]; // memory block
  ...
  { // body of slave process triggered by pslave
    int index;
    index = pslave.get_address();
    // index gets value 68 of the master transaction
    memory[index] = pslave;
    // memory[68] is assigned the value 10
    // don't specify an index with pslave
    ...
  }
};
```

In the example above, a caller attached to master port P2 writes out a value with index 68. The slave process reads the index value of the transaction by invoking the member function get_address() of the pslave port.

Indexed Port Arrays

You can define an array of indexed ports. An indexed port array P2 with 10 members and an address range of (0,1023) for each member is specified in the following example:

```
sc_inmaster<int, 1024>  P2[10];
```

To read the value of port member 5 at address 68, you use the C++ syntax for two-dimensional arrays as shown in the following example. The first index indicates the member of the port array, and the second index indicates the address of the data.

```
int data;
data = P2[5][68];
```

Inout Ports

When a slave process is triggered by an sc_inoutslave port in response to a transaction initiated by a master process, the slave process needs to determine the direction of the data transfer requested by the master. This is done by calling the input() method of the slave port. This method returns true for an input transaction into the slave port and false for an output. The master always determines the direction of a

transaction by writing to or reading from its master port. A slave response that is incompatible with the master request will result in a run time error.

In the following example, the direction of the data transfer on port P1 is obtained by the P1.input() method.

```
sc_inoutslave<int> P1;
// the rest of the code is not shown
if ( P1.input() ) {
    val = P1;  // read from P1
} else {
    P1 = val;  // write to P1
}
```

Connectivity Rules for Abstract Ports in Hierarchical Designs

When modules are instantiated inside other modules, a child module port that connects to a parent module port must be of compatible type as follows:

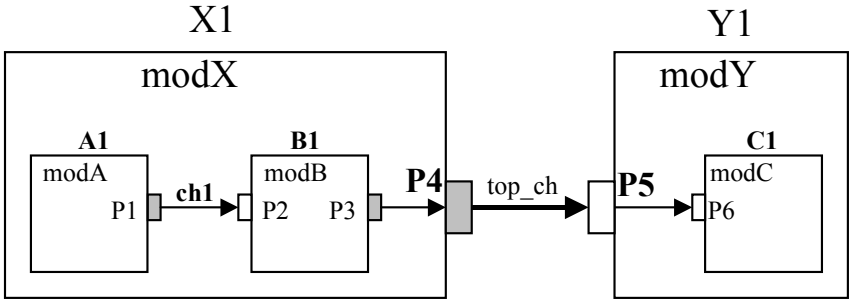
- A master port must connect to a master port, a slave port to a slave port.
- A child input port can connect to a parent input or inout port.
- A child output port can connect to a parent output or inout port.
- A child inout port can connect to a parent inout port.
- The data type of child and parent module must be the same.

These rules are checked at initialization by the SystemC class library.

Figure 8 illustrates how abstract ports and links are used in a hierarchical system. In this system, P4 is a representation of P3 at another level of hierarchy, so it retains

the same abstract protocol specification as P3. In module modY, P5 has the same relationship to P6. The code is also shown below.

FIGURE 8. Hierarchical Example




```
SC_MODULE(modA) {
  sc_outmaster<int> P1;
  ...
};

SC_MODULE(modB) {
  sc_inslave<int> P2;
  sc_outmaster<int> P3;
  ...
};

SC_MODULE(modC) {
  sc_inslave<int> P6;
  ...
};

SC_MODULE(modX) {
  sc_outmaster<int> P4;
  sc_link_mp<int> ch1;
  modA    *A1;
  modB    *B1;
  ...
  SC_CTOR(modX) {
    A1 = new modA("A1");
    A1(ch1);
    B1 = new modB("B1");
    B1(ch1, P4);
```

```
  }
};

SC_MODULE(modY) {
  sc_inslave<int> P5;
  modC *C1;
  ...
  SC_CTOR(modY) {
    C1 = new modC("C1");
    C1(P5);
  }
};

SC_MODULE(top) {
  sc_link_mp<int> top_ch;
  modX *X1;
  modY *Y1;
  ...
  SC_CTOR(top) {
    X1 = new modX("X1);
    X1(top_ch);
    Y1 = new modY("Y1");
    Y1(top_ch);
  }
};
```

Bus-Cycle-Accurate Level

Refining Communication with Bus Protocols

When you refine a design from the functional abstraction to the Bus-Cycle-Accurate (BCA)[1] or Cycle-Accurate (CA) HW abstraction levels, you do the following transformations:

- Sequential communication channels are refined to concurrent cycle accurate communication channels with a bus protocol.

1. In this discussion we use BCA to denote both BCA and CA level descriptions.

- Functional processes are refined to synchronous concurrent processes, which have clocks and resets.

Channel refinement is not as straightforward as simply replacing a functional channel with a bus protocol channel. This may work for simple point-to-point communication channels but does not for complex multi-point channels. Complex channels require design work, either manually or aided by an interface synthesis tool involving design of address decoders, multiplexers, bridges, protocol adapters, etc.

In this discussion, we will not describe the refinement process in detail, which is methodology and tool dependent and falls outside the scope of this discussion. Instead we present the language constructs at the BCA level.

We will illustrate this using two approaches for communication refinement, which are:

- Module refinement: the modules that communicate are refined for bus protocol communication.
- Channel refinement: the channel is refined for bus protocol communication while the modules remain unaffected.

Module Refinement

This method is illustrated in Figure 9 for the producer-consumer example that was discussed earlier in this chapter. In the example, the functional level communication is refined to a bus protocol communication with a full-handshake protocol as follows:

- Master and slave abstract ports are refined to bus ports[1] by specializing them with a bus protocol template parameter, in this case a full-handshake protocol.
- The producer and consumer processes are refined to clocked processes, which implement the full-handshake bus protocol.

A bus port has terminals for bus protocol signaling. Terminals are protocol specific and are defined in the bus protocol class (see “User-Defined Bus Protocols” on page 40). A bus port acquires its terminals from the bus protocol class with which it

1. Ports that have been specialized with a bus protocol are called bus ports or also hierarchical ports because they have port terminals (such as request, acknowledge, data) for bus protocol signaling in the physical layer.

is specialized. For example, the full-handshake protocol in our example has a *req* (request), *ack* (acknowledge) and *d* (data) terminals. Terminals are accessed as members of the port class[1]. A terminal is of type signal port. Hence it binds to signals. A bus port does not have read/write methods. All communication takes place through its terminals[2].

For example, you write p.req to access the req terminal of a bus port p.

```
sc_outmaster<T, sc_fullHandshake<T> > p;
// full-handshake bus port
```

The data type <T> must be specified twice[3], once as a template parameter of the port and once as a template parameter of the protocol.

Bus ports bind to sc_link_mp<T> objects, which do not have a bus protocol template parameter. The following things happen when a bus port binds to a sc_link_mp<T> object:

- The sc_link_mp<T> checks whether this is the first bus port that binds to the link. If it is, the link queries the port for its bus protocol.
- Then sc_link_mp<T> creates a bundle of signals that is defined by the bus protocol and binds them to the terminals of the port.
- For each subsequent bus port that binds to the link, the port terminals are bound to the corresponding signals in the link.

All ports connected to a link must have the same protocol type.

A bus port is bound as a whole to an sc_link_mp<T> object[4], not its terminals. The advantage of this is that port binding does not change when its protocol is changed.

---

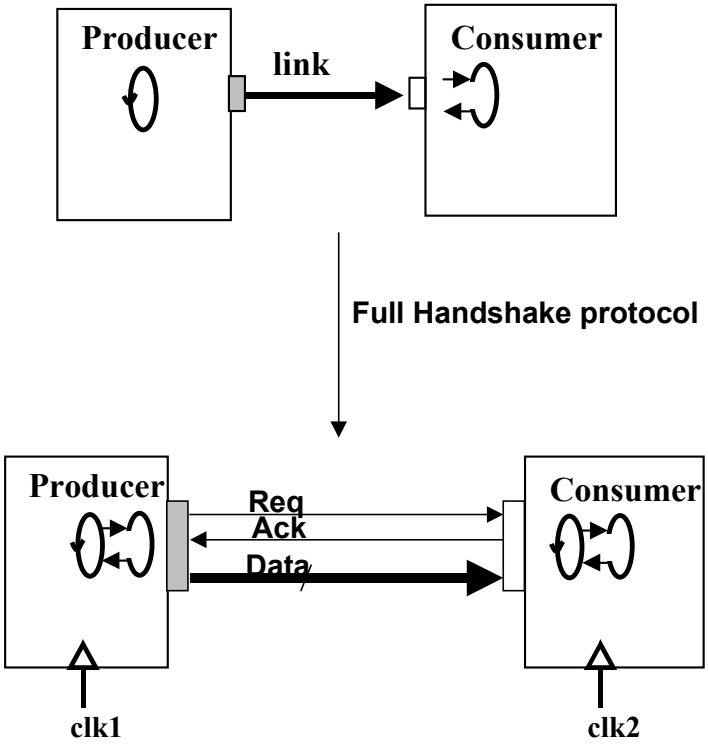
1. In SystemC v1.1beta release, port terminals were accessed as port-> terminal. This is still supported in this release but will be deprecated in future releases. The user is advised to use the port.terminal syntax instead.
2. This restriction will be removed in the future. A read/write method on a bus port will then read/write the port terminals according to the semantics of the bus protocol.
3. This double specification of <T> is an unfortunate artifact of the implementation. Both data types must be the same.
4. In the future you will be able to bind port terminals or signal ports to individual signals in a bus channel.

An sc_link_mp<T> object that is bound to bus ports does not have execution semantics as it did at the functional level. At the bus protocol level it only performs a signal creation and binding function as explained above.

SystemC comes with three pre-defined bus protocols: no-handshake, enable-handshake and full-handshake protocols. You can define your own bus protocols, as described in “User-Defined Bus Protocols” on page 40.

Port specialization with bus protocols allows tools to synthesize the channel from the different port protocols that are connected to it. This is more flexible than the channel refinement approach, which requires pre-defined channels for different channel configurations.

FIGURE 9. Module Refinement Example With a Full Handshake Protocol



The source code for the example of Figure 9 and after refinement is shown below.

```
SC_MODULE(producer)
{
  sc_outmaster<int> out1;
  sc_in<bool>       start;  // to kick-start the producer

  void generate_data ()
    {
      for (int i = 0;  i < 10;  i++)
        {
          out1 = i ;       // this will invoke the slave;
        }
    }

  SC_CTOR(producer)
    {
      SC_METHOD(generate_data);
      sensitive << start;
    }
};

SC_MODULE(consumer)
{
  sc_inslave<int> in1;
  int sum;              // declare as a module state variable

  void accumulate ()
    {
      sum += in1;
      cout << "Sum =    "  << sum << endl;
    }

  SC_CTOR(consumer)
    {
      SC_SLAVE(accumulate, in1);
      sum = 0;              // initialize the accumulator
    }
};

SC_MODULE(top) // structural module
{
  producer *A1;
  consumer *B1;
  sc_link_mp<int> link1;

  SC_CTOR(top)
```

```
    {
      A1 = new producer("A1");
      A1.out1(link1);
      B1 = new consumer("B1");
      B1.in1(link1);
    }
};
```

The code after communication refinement is shown below.

```
SC_MODULE(producer)
{
  sc_outmaster<int,sc_fullHndshk<int> > out1;
  sc_in<bool>       clk;

  void generate_data ()
    {
      for (int i = 0;  i < 10;  i++)
        {
           wait();
           out1.req = true;
           out1.data = i;
           while (!out1.ack) {
             wait();
             // wait for the clock in the producer
           out1.req = false;
           }
        }
    }

  SC_CTOR(producer)
    {
      SC_THREAD(generate_data);
      sensitive_pos << clk;
    }
};

SC_MODULE(consumer)
{
  sc_inslave<int, sc_fullHndshk<int> > in1;
  int sum;          // state variable

  void accumulate ()
    {
      while (1) {
```

```
        wait();
        if (in1.req) {
          sum += in1.data;
          cout << "Sum =    "  << sum << endl;
          wait();
          in1.ack = true;
          wait();
          in1.ack = false;
         }
       }
    }

  SC_CTOR(consumer)
    {
      SC_THREAD(accumulate);
      sensitive_pos << clk;
      sum = 0;// initialize the accumulator
    }
};

SC_MODULE(top) // structural module
{
  producer *A1;
  consumer *B1;
  sc_clock clk("clk",1000);
  sc_link_mp<int> link1;

  SC_CTOR(top)
    {
       A1 = new producer("A1");
       A1.out1(link1);
       A1.clk(clk);
       B1 = new consumer("B1");
       B1.in1(link1);
       B1.clk(clk);

    }
};
```

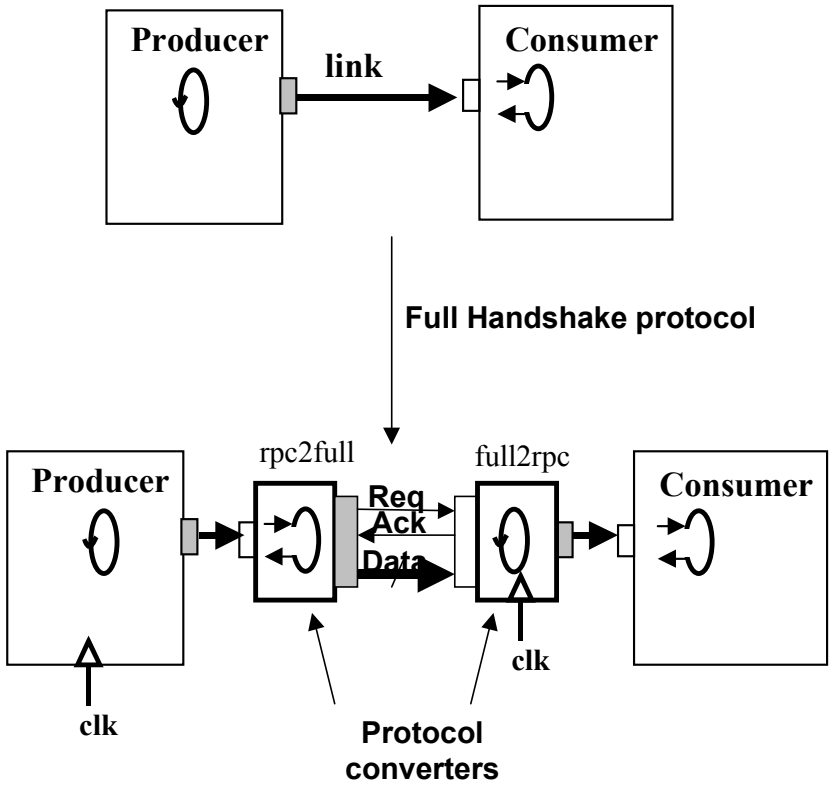
Channel Refinement

In the example below an abstract communication channel at the functional level is refined into a full-handshake protocol channel. Refinement is entirely done in the channel by inserting protocol conversion modules in the channel as shown in

Figure 10. This approach allows separation of communication and behavior. The protocol converters in this example convert a communication from an abstract into a bus protocol (rpc2full) communication or vice-versa (full2rpc).

The rpc2full converter is a slave process, which does not have a clock input of its own. It “gets” its clock from the calling process (producer). The full2rpc converter has a clock input which is passed on to the consumer module.

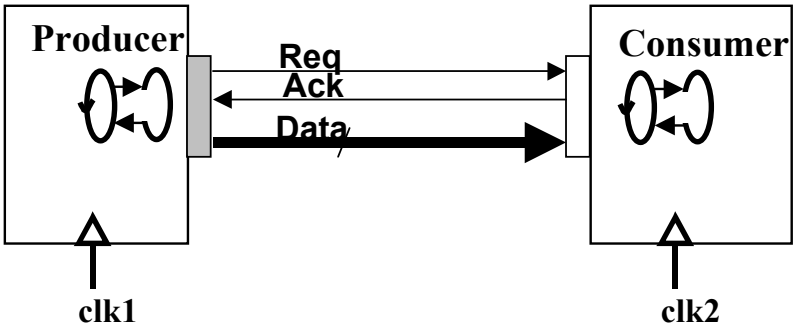
FIGURE 10. Channel Refinement Example



In the final step, protocol conversion modules are in-lined inside the producer and consumer modules as shown in Figure 11. In-lining merges the slave process function into the calling process such that abstract ports are removed as a result of this transformation. In the future, this in-lining will be done automatically through the

read/write methods of the bus port[1]. Note that after in-lining the channel refinement method yields the same end result as the module refinement.

FIGURE 11. After In-Lining of Protocol Converters



We're using the same producer/consumer example to illustrate the channel refinement method.

The following shows the code for the protocol converters and the top module after protocol converters are inserted.

```
SC_MODULE(rpc2Full) {
sc_inslave<int> in;
sc_outmaster<int, sc_fullHndshk <int> > out;
    void doit() { // executes in the thread of the producer
      wait(); // wait for the clock in the producer
     out.req = true;
      out.data = in;
      while (!out.ack) {
        wait();
      out.req = false;
      }
    }
    SC_CTOR(rpc2Full) {
      SC_SLAVE(doit,in);
```

1. This functionality will be provided in the future. Read/write methods will be member functions of the protocol class and implement the protocol's semantics. A bus port will access the read/write method of a protocol using the -> operator. For example: port->read() will access the read() method of the port's protocol.

```
    }

};
SC_MODULE(full2RPC) {
sc_outmaster<int> out;
   sc_in<bool> clk;
   sc_inslave<int, sc_fullHndshk <int> > in;
    void doit() {
      while (1) {
        wait();
        if (in.req) {
          out = in.data; // RPC call
          wait();
          in.ack = true;
          wait();
          in.ack = false;
         }
       }
    }
    SC_CTOR(full2RPC) {
      SC_THREAD(doit);
      sensitive_pos << clk;
    }

}


SC_MODULE(top) {
  producer *A1;
  consumer *B1;
  rpc2Full *adapter1;
  full2RPC *adapter2;
  sc_link_mp<int> link1;
  sc_link_mp<int> link2;
  sc_link_mp<int> buslink;
  sc_clock clk("clk",1000,SC_PS);
  SC_CTOR(top)
    {
      A1 = new producer("A1");
      A1.out1(link1);
      A1.clk(clk);
      B1 = new consumer("B1");
      B1.in1(link2);
      adapter1 = new rpc2Full("adapter1");
      adapter2 = new full2RPC("adapter2");
```

```
    adapter1.in(link1);
    adapter1.out(buslink);
    adapter2.in(buslink);
    adapter2.clk(clk);
    adapter2.out(link2);
  }
   };
```

Pre-Defined Bus Protocols

The following bus protocols are provided as a part of the SystemC class library:

```
sc_noHandshake<T>
sc_enableHandshake<T>
sc_fullHandshake<T>
```

The following illustrates the bus protocol signals, their type, and direction.

NoHandshake Protocol:

```
sc_outmaster<int, sc_noHandshake<int> >  Pwrite;
   Pwrite.d // d: out data

sc_inslave<int, sc_noHandshake<int> >  Pread;
   Pread.d  // d: in data

sc_inmaster<int, sc_noHandshake<int> >  Pread;
   Pread.d  // d: in data

sc_outslave<int, sc_noHandshake<int> >  Pwrite;
   Pwrite.d  // d: out data

sc_inoutmaster<int, sc_noHandshake<int> >  Pwrite_read;
   Pwrite_read.din  // din : in  data
   Pwrite_read.dout // dout: out data
   Pwrite_read.nRW  // nRW:  out control

sc_inoutslave<int, sc_noHandshake<int> >  Pread_write;
   Pread_write.din  // din : out data
   Pread_write.dout // dout: in  data
   Pread_write.nRW  // nRW:  in  control
```

Enable Handshake Protocol:

```
sc_outmaster<int, sc_enableHandshake<int> >  Pwrite;
```

```
    Pwrite.d  // d:  out data
    Pwrite.en // en: out control

sc_inslave<int, sc_enableHandshake<int> >  Pread;
    Pread.d  // d:  in data
    Pread.en // en: in control

sc_inmaster<int, sc_enableHandshake<int> >  Pread;
    Pread.d  // d:  in  data
    Pread.en // en: out control

sc_outslave<int, sc_enableHandshake<int> >  Pwrite;
    Pwrite.d  // d:  out data
    Pwrite.en // en: in  control

sc_inoutmaster<int, sc_enableHandshake<int> >  Pwrite_read;
    Pwrite_read.din  // din : in  data
    Pwrite_read.dout // dout: out data
    Pwrite_read.en   // en:   out control
    Pwrite_read.nRW  // nRW:  out control

sc_inoutslave<int, sc_enableHandshake<int> >  Pread_write;
    Pread_write.din  // din : out data
    Pread_write.dout // dout: in  data
    Pread_write.en   // en:   in  control
       Pread_write.nRW  // nRW:  in  control
```

Full Handshake Protocol:

```
sc_outmaster<int, sc_fullHandshake<int> >  Pwrite;
    Pwrite.d   // d:   out data
    Pwrite.ack // ack: in  control
    Pwrite.req // req: out control

sc_inslave<int, sc_fullHandshake<int> >  Pread;
    Pread.d   // d:   in  data
Pread.ack // ack: out control
    Pread.req // req: in  control

sc_inmaster<int, sc_fullHandshake<int> >  Pread;
    Pread.d   // d:   in  data
    Pread.ack // ack: in  control
    Pread.req // req: out control

sc_outslave<int, sc_fullHandshake<int> >  Pwrite;
```

```
    Pwrite.d   // d:   out data
    Pwrite.ack // ack: out control
    Pwrite.req // req: in  control

sc_inoutmaster<int, sc_fullHandshake<int> >  Pwrite_read;
    Pwrite_read.din  // din : in  data
Pwrite_read.dout // dout: out data
    Pwrite_read.ack  // ack:  in  control
Pwrite_read.req  // req:  out control
    Pwrite_read.nRW  // nRW:  out control

sc_inoutslave<int, sc_fullHandshake<int> >  Pread_write;
    Pread_write.din  // din : out data
    Pread_write.dout // dout: in  data
    Pread_write.ack  // ack:  out control
    Pread_write.req  // req:  in  control
    Pread_write.nRW  // nRW:  in  control
```

Figure 12 shows the terminals for pre-defined protocols.

FIGURE 12. Terminals of Pre-Defined Bus Protocols

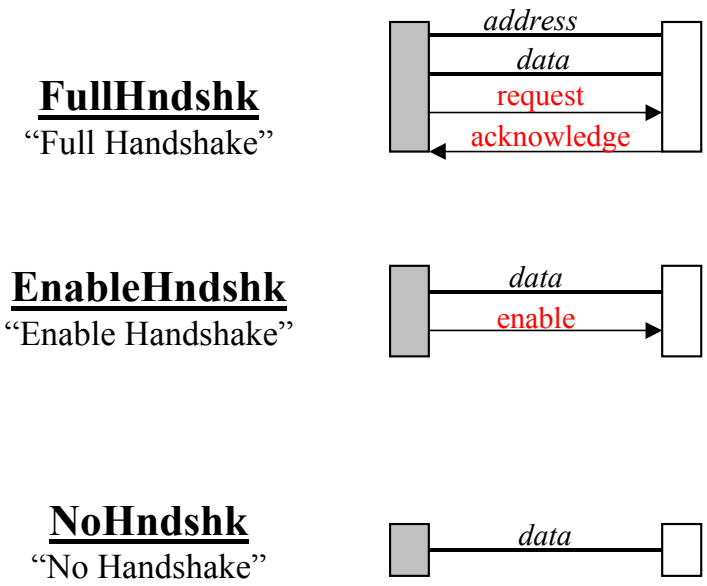
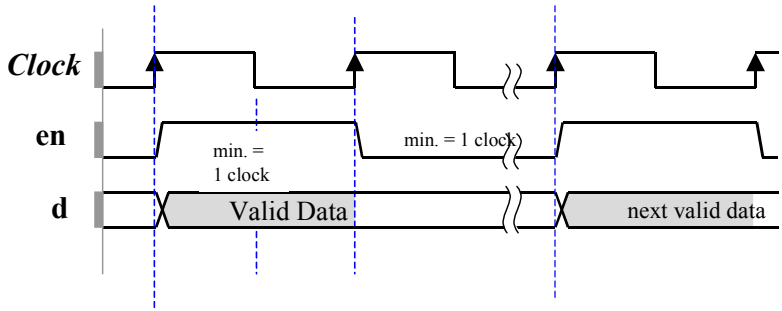


Figure 13 shows the timing diagram for the sc_enableHandshake protocol.

FIGURE 13. Enable-Handshake Timing Diagram



Tracing of Ports

You can trace abstract ports and terminals of bus ports for later viewing with waveform display tools. You can only trace ports after they have been bound to a signal. This happens after the simulation has started (after sc_initialize or sc_start). Therefore all trace statements for ports should follow a sc_initialize or a sc_start statement in sc_main. It is recommended not to put trace statements for ports in the constructor of a module since the port may not be bound yet to a signal when the trace statement executes. The trace function gives a warning message when the port is not connected (yet).

Example of tracing:

```
SC_MODULE(slave_bca){
   //ports
   sc_inslave<int, sc_fullHandshake<int> > dat2;
   ...
};

int sc_main(int argc, char* argv[]) {
   slave_bca s1("slave_bca");

   ...
   sc_initialize();
   trace_file = sc_create_vcd_trace_file("trace");

   sc_trace(trace_file,s1.dat2->req,"BusRqst");
```

```
    sc_time t(-1);
    sc_start(t);

    sc_close_vcd_trace_file(trace_file);

};
```

User-Defined Protocols

This section can be skipped without loss of continuity.

This section describes how you can define your own protocols, which can be abstract type (for the functional level) or bus protocol (for the cycle accurate level) type.

User-Defined Bus Protocols

New bus protocols can be defined using the SC_PROTOCOL construct. A SC_PROTOCOL defines a structure that contains the terminal definitions for a bus protocol. As explained earlier, a bus port that is specialized with a bus protocol acquires its terminals from this bus protocol class. The terminals of a bus port (such as request, acknowledge, address, and data terminals) are used for bus protocol signaling at the RTL level. The terminals of a bus port are members of the bus port class. For example a bus port *p* with a data terminal *d* is accessed as *p.d* with the 'dot' member operator.

A terminal defines two things:

- A signal port type as one of sc_in, sc_out, or sc_inout type together with its data type
- The terminal type as one of data (TT_DATA), control (TT_CONTROL), or address (TT_ADDRESS)

Example:

```
sc_terminal_out<datatype,TT_DATA> d;
```

This terminal creates the following signal port object:

```
sc_out<datatype>   d;
```

The terminal type is TT_DATA. When a bus port is bound to an sc_link_mp<T> object, all the terminals of the bus port are bound to the corresponding signals in the link as explained before. The terminal type is used to bind the signal port of the terminal to the corresponding signal in the channel. In the example, port *d* will be bound to the data signal.

A bus protocol defines terminals per abstract port type (called port role in the BNF below) since the terminal directionality is in general port specific. For example, an enable terminal is configured as an output port for a master and as an input for a slave port.

For example, the SC_INMASTER_P macro (called port role in the BNF below) is used to define the terminals of an sc_inmaster port.

The terminals of a bus port are bound by sc_link_mp to the bus signals in the channel as follows:

- If a terminal of a given type (TT_DATA, TT_CONTROL, TT_ADDRESS) has a terminal_ id template argument (int type), then the terminal connects to the corresponding terminals (=with the same type and terminal id) of other ports in the link.
- If no terminal_id is specified then the terminals are connected by position: for example, the first TT_DATA of a bus port will connect to the first TT_DATA of other bus ports connected to the same sc_link_mp (similarly for TT_CONTROL and TT_ADDRESS).

Example:

In this example, a bus protocol for a new “enable-handshake” protocol is defined. The *d* terminal is specified with a terminal id while *en* and *nRW* terminals are specified without id to illustrate both possibilities.

```
template <class datatype>
SC_PROTOCOL(sc_MyEnableHandshake)
{
  SC_OUTMASTER_P { //defines terminals for
                   //sc_outmaster port type
    sc_terminal_out<datatype,TT_DATA,1> d;
    sc_terminal_out<bool,TT_CONTROL>  en;
  };

  SC_INMASTER_P {//defines terminals for
```

```
                //sc_inmaster port type
    sc_terminal_in<datatype,TT_DATA,1>  d;
    sc_terminal_out<bool,TT_CONTROL>  en;
  };

  SC_INOUTMASTER_P {//defines terminals for
                   //sc_inoutmaster port type
    sc_terminal_inout<datatype,TT_DATA,1>  d;
    sc_terminal_out<bool,TT_CONTROL>  en;
    sc_terminal_out<bool,TT_CONTROL>  nRW;
  };

  SC_OUTSLAVE_P {//defines terminals for
                 //sc_outslave port type
    sc_terminal_out<datatype,TT_DATA,1> d;
    sc_terminal_in<bool,TT_CONTROL>   en;
  };

  SC_INSLAVE_P {//defines terminals for
                //sc_inslave port type
    sc_terminal_in<datatype,TT_DATA,1>  d;
    sc_terminal_in<bool,TT_CONTROL>   en;
  };

  SC_INOUTSLAVE_P {//defines terminals for
                   //sc_inoutslave port type
    sc_terminal_inout<datatype,TT_DATA,1> d;
    sc_terminal_out<bool,TT_CONTROL>  en;
    sc_terminal_out<bool,TT_CONTROL>  nRW;
  };
};


SC_MODULE(Producer)
{
  sc_outmaster<int,sc_MyEnableHandshake> mstr;
  sc_in<bool> clk;
  int data;

  void send_data()
    {
      while (true)
        {
          mstr.d = data;  // in 1.1beta1 this
                          // would have been mstr->d
```

```
          mstr.en = true; // in 1.1beta1 this
                          // would have been mstr->en
          wait(); // wait for clock
        }
    }

  SC_CTOR...
};
```

User-Defined Abstract Protocols

You can define your own abstract protocols, which inherit from the base abstract protocol (sc_abstract). An abstract port has by default the base abstract protocol, which defines the in-lined execution semantics that we have discussed in “Untimed Functional (UTF) Level” on page 3. User-defined abstract protocols have the same semantics as sc_abstract but in addition they allow you to define your own protocol attributes.

An attribute is used to communicate information other than data[1] between processes connected to an abstract channel sc_link_mp<T>. Attributes are variables with a data type. Attributes are stored in the channel and accessed through abstract ports that have been specialized with the particular protocol. The new value written to an attribute is immediately visible to all ports on the channel. An attribute retains its value until it’s overwritten.

Attributes are used to configure a channel or a transaction on a channel, for example, to make an abstract communication blocking, non-blocking, or blocking with timeout. An abstract port reads or writes the attributes of the protocol as its data members. An attribute has a direction as one of out, in or inout, respectively indicating whether a port can write to it, read from it, or both.

For example, the following defines an attribute called *timeout* with data type *sc_time*. It is an output attribute, meaning that an abstract port can write to it.

```
sc_attribute_out<sc_time> timeout;
```

1. For tool interoperability, port attributes should not be used to communicate data. All data communication should take place through the read/write methods of ports.

You define attributes similarly to terminals in bus protocols. The only difference is that attributes are defined in port roles for abstract protocols. For example, in SC_OUTMASTER_RPC (port role for an abstract protocol) you define the attributes for an sc_outmaster port.

In the example below, we're creating a new abstract protocol (sc_timeout_rpc) that has a timeout and a status attribute. We use the new protocol in the FIFO example in which the blocking write (bWriteTimeout()) slave process times-out when the buffer in the FIFO cannot be written for a certain amount of time. The producer process configures the channel with the timeout attribute, writes to its master port, and invokes the bWriteTimeout in the FIFO. After the transaction, the producer checks whether the transaction was successful with the status attribute.

```
#include "systemc.h"
#include "sc_mslib.h"
template <class datatype>
SC_PROTOCOL(sc_timeout_rpc) {
  SC_MASTER_RPC {
    sc_attribute_out<sc_time> timeout;
    sc_attribute_in<bool> status;

  };
  SC_OUTMASTER_RPC {
    sc_attribute_out<sc_time> timeout;
    sc_attribute_in<bool> status;
  };
  SC_INMASTER_RPC {
    sc_attribute_out<sc_time> timeout;
    sc_attribute_in<bool> status;

  };
  SC_INOUTMASTER_RPC{
    sc_attribute_out<sc_time> timeout;
    sc_attribute_in<bool> status;

  };
  SC_SLAVE_RPC {
    sc_attribute_in<sc_time> timeout;
    sc_attribute_out<bool> status;

  };
  SC_OUTSLAVE_RPC {
    sc_attribute_in<sc_time> timeout;
    sc_attribute_out<bool> status;
```

```
  };
  SC_INSLAVE_RPC {
    sc_attribute_in<sc_time> timeout;
    sc_attribute_out<bool> status;

  };
  SC_INOUTSLAVE_RPC {
    sc_attribute_in<sc_time> timeout;
    sc_attribute_out<bool> status;

  };
};

SC_MODULE(Producer) {

  sc_outmaster<int,sc_timeout_rpc<int> > m;
  sc_in<int> data;
  sc_in<bool> clk;

  void doit() {
    sc_time  timeout(100, SC_US);
    while (1) {
      m.timeout = timeout;
      cerr << "master : send " << data << endl;
      m = data;
      cerr << "master status = " << m.status << endl;
      wait();
    }
  }

  SC_CTOR(Producer) {
    SC_THREAD(doit);
    sensitive << clk;
  }

};

SC_MODULE(fifo) {
  sc_inslave<int,sc_timeout_rpc<int> > s;
  buffer<int> buf;  //  fifo buffer class not shown

  void bWriteTimeout() {
    int d = s;
    cerr << " slave called with " << d <<
```

```
            " timeout " << s.timeout << endl;
    s.status = false;
    if buf.full() { // wait until !buf.full() or time-out
        //return a ref to a buffer event
        wait(s.timeout, buf.event_ref());
        if !buf.full() s.status = true; // not timed-out
      }

    else s.status = true;
    if s.status buf.put(d); // succesful
  }

  SC_CTOR(fifo) {
    SC_SLAVE(bWriteTimeout,s);

  }
};
```

BNF for Protocol Definitions

```
protocol_definition:
   SC_PROTOCOL ( identifier  ) {  port_protocols };

port_protocols :
   port_protocol
   port_protocols port_protocol

port_protocol :
   port_role { port_protocol_members };

port_role :
   SC_SLAVE_P       // port roles for bus protocols
   SC_INSLAVE_P
   SC_OUTSLAVE_P
   SC_INOUTSLAVE_P
   SC_MASTER_P
   SC_INMASTER_P
   SC_OUTMASTER_P
   SC_INOUTMASTER_P
   SC_SLAVE_RPC     // following are port roles
                    // for abstract protocols
   SC_INSLAVE_RPC
   SC_OUTSLAVE_RPC
   SC_INOUTSLAVE_RPC
   SC_MASTER_RPC
```

```
   SC_INMASTER_RPC
   SC_OUTMASTER_RPC
   SC_INOUTMASTER_RPC


port_protocol_members :
    /*empty*/
    port_protocol_member port_protocol_members

port_protocol_member :
   terminal_member
   attribute_member


terminal_member :
   terminal_type < datatype > identifier ;
   terminal_type < datatype ,terminal_kind > identifier ;
   terminal_type < datatype, terminal_kind, terminal_id >
identifier ;

attribute_member :
    attribute_type< datatype> identifier;
    attribute_type< datatype, attribute_id> identifier;


terminal_kind :
   TT_DATA
   TT_CONTROL
   TT_ADDRESS

terminal_id :
   number

attribute_id:
    number

terminal_type :
   sc_in_terminal
   sc_out_terminal
   sc_inout_terminal
   sc_in_terminal_rv
   sc_out_terminal_rv
   sc_inout_terminal_rv
```

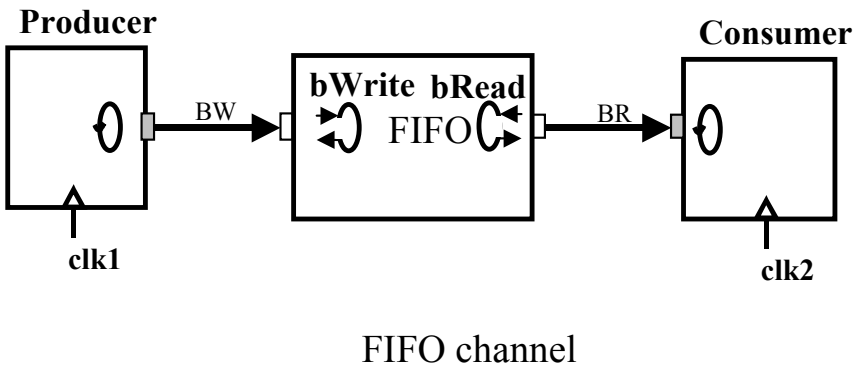
```
attribute_type :
   sc_in_attribute
   sc_out_attribute
   sc_inout_attribute
```

Examples

FIFO Model at the Functional Level

In this example, we illustrate how blocking RPC is used to model blocking write and read in a FIFO communication link. A producer process produces data items (integers in this example) and sends it through the FIFO link to the consumer. The producer runs at a higher speed than the consumer process such that the FIFO buffer will get full after some time. This will block the producer process since it will not be able to write into the FIFO buffer. As soon as the consumer removes an item from the FIFO buffer, the producer will be unblocked and write another item into the buffer.

FIGURE 14. FIFO Model at the Functional Level



FIFO channel

In the example code below, producer and consumer are implemented as clocked SC_THREAD processes to allow the two processes to run at different ‘speeds’. Producer is sensitive to the clock. Producer has a sc_outmaster port through which it invokes the blockingWrite() slave process in the FIFO. The blockingWrite() blocks when the buffer is full. The state of the buffer is tested on every sensitive

clock edge of the producer clock . Consumer is sensitive to a different clock. Consumer has an sc_inmaster port through which it invokes the blockingRead() slave process in the FIFO. The blockingRead() blocks when the buffer is empty. The state of the buffer is tested on every sensitive clock edge of the consumer clock. For the sake of brevity, the buffer class is not shown in the code below. The buffer has internally a ‘full’ and an ‘empty’ signal to keep the buffer’s state such that the buffer has a deterministic behavior, i.e. its behavior does not depend on which of blockingWrite() and blockingRead() processes checks first the state of the buffer in a given delta cycle. Simulation results below show a correct operation of the FIFO.

```
// file buffer.h
#define BUFFER_SIZE 10

template <class itemT>
class buffer
{
public:
  buffer(); // constructor
  virtual ~buffer() {}
  itemT get();          // get item from the top of buffer
  bool put(itemT);      // put item at the bottom of buffer
  bool full();          // buffer full
  bool empty();         // buffer empty

private:
  // private section not shown
};

// File Name : fifo.h
// A module with inslave and outslave abstract ports,
// and a fifo store

#ifndef SYSTEMC_H
#include "systemc.h"
#include "sc_mslib.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

#ifndef BUFFER_H
#include "buffer.h"
#endif
```

```
SC_MODULE(fifo)
{
  // ports
  sc_out<int> full;
  sc_out<int> empty;
  sc_inslave<int> Pwrite; // slave port
  sc_outslave<int> Pread; // slave port

  buffer<int> buf; // FIFO buffer
  int item;        // buffer item

  // slave methods

  void blockingWrite()
    {
      if (buf.full() )
        {
          do {wait();} // wait for sensitive edge
                       // of the producer clock
          while( buf.full());
        }

      // buffer is not full
      // write into buffer
      item = Pwrite; // read from slave port
      cout << "Writing into buffer: item = " << item <<
                                                    endl;
      buf.put(item);
    }

  void blockingRead()
    {
      cout << "\nfifo:blockingRead called" << endl;
      if (buf.empty())
        {
          do {wait();}
         // wait for sensitive edge of the consumer clock
          while (buf.empty() );
        }

      // buffer is not empty
      // read from buffer
      item = buf.get();
      cout << "  Item read = " << item << endl;
```

```
      Pread = item; // write to slave port
    }

  SC_CTOR(fifo)
    {
      SC_SLAVE( blockingWrite, Pwrite);
      SC_SLAVE( blockingRead, Pread);
    }
};


//
// producer.h  ::  producer
//

#ifndef SYSTEMC_H
#include "systemc.h"
#include "sc_mslib.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

SC_MODULE(producer)
{
  // port declaration
  sc_in<int> full;
  sc_outmaster<int> Pout; // refinable port
  sc_in_clk clk;

  // Internal variable
  int val;

  // outmaster process
  void producer_thread()
    {
      while (true)
        {
          val += 2;
          Pout = val;
          wait();  // wait for pos_edge clock event
        }
    }
```

```
  SC_CTOR(producer)
    {
      SC_THREAD(producer_thread);
      sensitive_pos << clk;
      val = 0;
    }
};


//
// consumer.h  ::  consumer module
//

#ifndef SYSTEMC_H
#include "systemc.h"
#include "sc_mslib.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

SC_MODULE(consumer)
{
  // declare ports
  sc_in<int>       empty;
  sc_inmaster<int> Cin;
  sc_in_clk        clk;

  // Internal variable
  int x;

  // inmaster process
  void consumerFunc()
    {
      while (true)
        {
          x = Cin;
          wait();  // wait for pos_edge clk
        }
    }

  SC_CTOR(consumer)
    {
      SC_THREAD(consumerFunc);
```

```
      sensitive_pos << clk;
    }
};


//
// top.cc : contains sc_main ; Instantiates FIFO design
//

#ifndef SYSTEMC_H
#include "systemc.h"
#include "sc_mslib.h"
#endif

#ifndef SC_XCOM_H
#include "sc_xcom.h"
#endif

#include "consumer.h"
#include "producer.h"
#include "fifo.h"

#ifndef BUFFER_H
#include "buffer.h"
#endif

int sc_main(int ac, char *av[] )
{

  // declare channels/signals

  sc_link_mp<int> BW;
  sc_link_mp<int> BR;

  // create clocks with diff frequencies
  sc_clock clock1 ("Clock1",5,SC_NS,0.5,0.3,SC_NS,true);
  sc_clock clock2 ("Clock2",40,SC_NS,0.5,0.1,SC_NS,true);

  // instantiate all blocks and connect
  // to channels, signals

  producer       p1("Master");
  fifo           f1("fifo");
  consumer       c1("Slave");
```

```
  p1.Pout(BW);
  p1.clk(clock1);
  f1.Pwrite(BW);
  f1.Pread(BR);
  c1.Cin(BR);
  c1.clk(clock2);

  sc_start(100);
  // return zero if no error
  return 0;
}
```

Output results from simulation of the FIFO example are shown below. Note that the FIFO buffer gets filled up by blockingWrite's starting from the left and gets emptied by blockingRead from the right. When the buffer is full, blockingWrite is blocked and unblocks when an item is removed from the buffer.

Simulation Output:

```
Writing into buffer: item = 2
  Array after shifting = 2 0 0 0 0 0 0 0 0 0

fifo:blockingRead called

fifo:blockingRead called
Writing into buffer: item = 4
  Array after shifting = 4 0 0 0 0 0 0 0 0 0
Writing into buffer: item = 6
  Array after shifting = 6 0 0 0 0 0 0 0 0 0
Writing into buffer: item = 8
  Array after shifting = 8 6 0 0 0 0 0 0 0 0
Writing into buffer: item = 10
  Array after shifting = 10 8 6 0 0 0 0 0 0 0
Writing into buffer: item = 12
  Array after shifting = 12 10 8 6 0 0 0 0 0 0
Writing into buffer: item = 14
  Array after shifting = 14 12 10 8 6 0 0 0 0 0
Writing into buffer: item = 16
  Array after shifting = 16 14 12 10 8 6 0 0 0 0
Writing into buffer: item = 18
  Array after shifting = 18 16 14 12 10 8 6 0 0 0

fifo:blockingRead called
Writing into buffer: item = 20
  Array after shifting = 20 18 16 14 12 10 8 0 0 0
```

```
Writing into buffer: item = 22
  Array after shifting = 22 20 18 16 14 12 10 8 0 0
Writing into buffer: item = 24
  Array after shifting = 24 22 20 18 16 14 12 10 8 0
Writing into buffer: item = 26
  Array after shifting = 26 24 22 20 18 16 14 12 10 8

fifo:blockingRead called
Writing into buffer: item = 28
  Array after shifting = 28 26 24 22 20 18 16 14 12 10
```

FIFO Example at the BCA Level

The same FIFO at the BCA level is shown below. Full-handshake protocol is used between the producer/ consumer threads and the FIFO. Enable-handshake protocol would not work here since the FIFO is blocking. The processes are shown after the protocol modules have been in-lined inside the respective modules, that is, the producer, consumer and the FIFO modules. For the sake of brevity, we did not show the consumer process, which is very analogous to the producer.

```
//
// producer_bp.h :: producer with fullHandshake protocol
//

#include "systemc.h"
#include "sc_mslib.h"

SC_MODULE(producer)
{
  // internal variable
  int val;
  int state;  // state variable of FSM

  // port declaration
  sc_outmaster<int, sc_fullHandshake<int> > Pout;
                                                  // bus port
  sc_in_clk clk;

  // outmaster process
  void producer_thread()
    {
      state = 0;
```

```
      Pout.req = 0;
      while (true)
        {
          wait();
          switch(state)
            {
            case 0:
              Pout.req = 0;
              if (Pout.ack == 0)
                {
                  state = 1;
                }
              break;
            case 1:
              Pout.req = 1;
              val += 2;
              Pout.d = val;
              state = 2;
              break;
            case 2:
              if (Pout.ack==1)
                {
                  state = 0;
                  Pout.req = 0;
                }
              break;
            }
        }
    }

  SC_CTOR(producer)
    {
      SC_THREAD(producer_thread);
      sensitive_pos << clk;
      val = 0;
    }
};


// File Name : fifo_bp.h
// A module with inslave and outslave fullHandshake bus
// ports, and a fifo store

#include "systemc.h"
#include "sc_mslib.h"
```

```
#ifndef BUFFER_H
#include "buffer.h"
#endif

SC_MODULE(fifo)
{
  // ports
  sc_in_clk wclk;
  sc_in_clk rclk;
  sc_inslave<int, sc_fullHandshake<int> > Pwrite;
   // slave port
  sc_outslave<int, sc_fullHandshake<int> > Pread;
   // slave port

  buffer<int> buf;
  int item;

  // slave methods
  void blockingWrite()
    {
      Pwrite.ack = 0;
      while(1)
        {
          wait();  // wait for wclk
          if (Pwrite.req)
            {
              if (buf.full() )
                {
                  // wait for buffer not full;
                  do {wait(); }  // wait for wclk
                  while( buf.full());
                }

              item = Pwrite.d; // read from slave port
              cout << "Writing into buffer: item = "
                     << item << endl;

              buf.put(item);
              Pwrite.ack = 1;
              while(Pwrite.req == 1)
                wait();  // wait for wclk
              Pwrite.ack = 0;
            }
        }
```

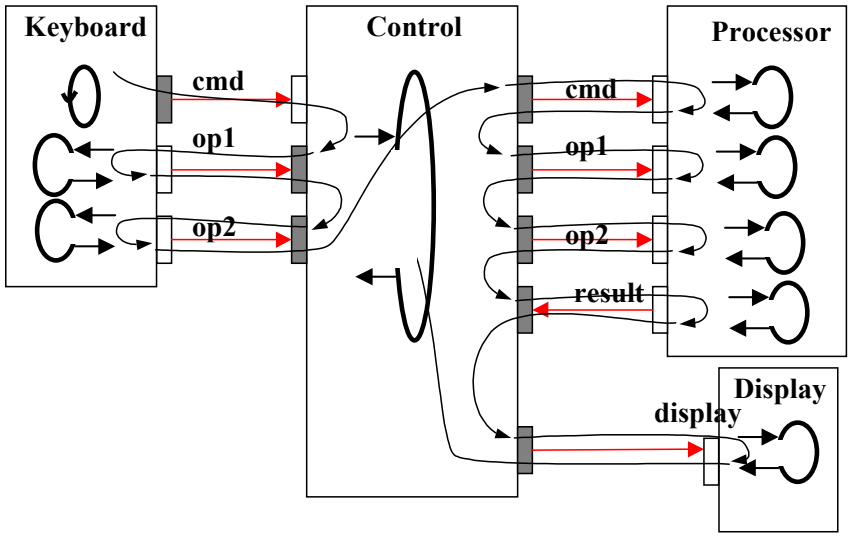
```
    }

  void blockingRead()
    {
      Pread.ack = 0;
      while(1)
        {
          wait(); // wait for clock
          if (Pread.req)
            {
              cout << "\nfifo:blockingRead called"
                   << endl;
              if (buf.empty())
                {  // wait for not empty buffer
                  do {wait();} // wait for rclk
                  while (buf.empty() );
                }

              item = buf.get();
              // cout << "  Item read = " << item << endl;
              Pread.d = item; // write to slave port
              Pread.ack = 1;
              while(Pread.req == 1)
              wait(); //wait for rclk
              Pread.ack = 0;
            }
        }
    }

  SC_CTOR(fifo)
    {
      SC_THREAD(blockingWrite);
      sensitive_pos << wclk;
      SC_THREAD(blockingRead);
      sensitive_pos << rclk;
    }
};
```

Example: Simple Arithmetic Processor

In this example, the key module (keyboard) serves as a test bench to generate inputs for a simple processor that takes a command for an arithmetic operation together

with two operands and calculates the results. The control module (ctrl) dispatches the commands coming from the keyboard to the processor, then gets the results from the processor and sends it to the display module for display. There is a single concurrent thread (key_input) in the key module. All other processes are slaves executing in this single concurrent thread. The single thread of execution is shown in Figure 15.

FIGURE 15. Simple Arithmetic Processor




```
#include "systemc.h"
#include "sc_mslib.h"

# define ADD 1
# define SUBTRACT 2
# define MULTIPLY 3
# define MYMAX 5

SC_MODULE(key)
{
  // ports
  sc_outmaster<int > cmd;
  sc_outslave<int > op1;
  sc_outslave<int > op2;
  sc_in_clk clk;
```

```
static int array1[MYMAX];
static int array2[MYMAX];
int i, state;
int scratch;

void key_input ()
  { // implements concurrent process
    // runs forever
    while (true)
      {
        switch (state)
          {
          case 0:
            scratch = ADD;
            cmd = scratch;
            if (i == MYMAX-1)
              state = 1;
            break;
          case 1:
            cmd = SUBTRACT;
            if (i == MYMAX-1)
              state = 2;
            break;
          case 2:
            cmd = MULTIPLY;
            if (i == MYMAX-1)
              state = 0;
            break;
          }
        if (i == MYMAX-1)
          i = 0;
        else i++;
      }
  }

void slave_to_op1 ()
  {
    op1 = array1[i];
  }

void slave_to_op2 ()
  {
    op2 = array2[i];
  }
```

```
  // constructor
  SC_CTOR(key)
    {
      SC_METHOD(key_input); // concurrent process
            sensitive << clk;
      SC_SLAVE(slave_to_op1, op1);
      SC_SLAVE(slave_to_op2, op2);
      state = 0;
      i = 0;
    }
};


#include "systemc.h"
#include "sc_mslib.h"

SC_MODULE(ctrl)
{
  // ports
  sc_inslave<int > cmd_in;
  sc_inmaster<int > op1_in;
  sc_inmaster<int > op2_in;

  sc_outmaster<int > op1_out;
  sc_outmaster<int > op2_out;
  sc_outmaster<int > cmd_out;
  sc_inmaster<int > result;

  sc_outmaster<int > display;

  // internal variables
  int operand1;
  int operand2;
  int cmd_2;

  // slave "methods"
  void slave_to_cmd_in ()
    {
      operand1 = op1_in;
      operand2 = op2_in;

      op1_out = operand1;
      op2_out = operand2;
      cmd_out = cmd_in;
```

```
      display = result;

      cmd_2 = cmd_in;

      cout << "\n operand1 = " << operand1
           << "  operand2 = " << operand2
           << "  cmd = " << cmd_in;
    }

  SC_CTOR(ctrl)
    {
      SC_SLAVE(slave_to_cmd_in, cmd_in);
    }
};


#include "systemc.h"
#include "sc_mslib.h"

SC_MODULE(fun)
{
  // ports
  sc_inslave<int > op1;
  sc_inslave<int > op2;
  sc_inslave<int > cmd;
  sc_outslave<int > result;

  // internal variables
  int operand1;
  int operand2;
  int command;

  void slave_to_op1 ()
    {
      operand1 = op1;           // read operand 1
    }

  void slave_to_op2 ()
    {
      operand2 = op2;           // read operand 2
    }

  void slave_to_cmd ()
    {
      command = cmd;            // read command
```

```
    }

  void slave_to_result ()
    {
      switch (command)
        {
        case 1 :
          result = operand1 + operand2; // Add
          break;
        case 2 :
          result = operand1 - operand2; // Subtract
          break;
        case 3 :
          result = operand1 * operand2; // Multiply
          break;
        default :
          printf ("\nfun: Undefined operation\n");
          break;
        }
    }

  // constructor
  SC_CTOR(fun)
    {
      SC_SLAVE(fun,slave_to_op1, op1);
      SC_SLAVE(slave_to_op2, op2);
      SC_SLAVE(slave_to_result, result);
      SC_SLAVE(slave_to_cmd, cmd);
    }
};


#include "systemc.h"
#include "sc_mslib.h"

SC_MODULE(disp)
{
  // ports
  sc_inslave<int > din;

  // internal variables
  int first_time;

  // slave "method"
  void slave_to_din ()
```

```
    {
      if (first_time) first_time = 0;
      else cout << "     Result = " << din << endl;
    }

  // constructor
  SC_CTOR(disp)
    {
      SC_SLAVE(slave_to_din, din);
      first_time = 1;
    }
};


#include "systemc.h"
#include "sc_mslib.h"

#include "key.h"
#include "ctrl.h"
#include "fun.h"
#include "disp.h"

int key::array1[MYMAX] = { 1,2,3,4,5 };
int key::array2[MYMAX] = {1,3,5,7,9 };
int sc_main(int argc, char* argv[])
{
  sc_link_mp<int > cmd;
  sc_link_mp<int > in1;
  sc_link_mp<int > in2;
  sc_link_mp<int > op1out;
  sc_link_mp<int > op2out;
  sc_link_mp<int > cmd_out;
  sc_link_mp<int > result;
  sc_link_mp<int > display;
  sc_clock clock1("clock", 10, SC_NS);

  key k1("key");
  k1.cmd(cmd);
  k1.op1(in1);
  k1.op2(in2);
  k1.clk(clock1);

  ctrl c1("ctrl");
  c1.cmd_in(cmd);
  c1.op1_in(in1);
```

```
  c1.op2_in(in2);
  c1.op1_out(op1out);
  c1.op2_out(op2out);
  c1.cmd_out(cmd_out);
  c1.result(result);
  c1.display(display);

  fun f1("fun");
  f1.op1(op1out);
  f1.op2(op2out);
  f1.cmd(cmd_out);
  f1.result(result);

  disp d1("display");
  d1.din(display);

  sc_start(50);

  return (0);
};
```

Simulation Output:

```
operand1 = 0  operand2 = 0  cmd = 1     Result = 0

operand1 = 1  operand2 = 1  cmd = 1     Result = 2

operand1 = 2  operand2 = 3  cmd = 1     Result = 5

operand1 = 3  operand2 = 5  cmd = 1     Result = 8

operand1 = 4  operand2 = 7  cmd = 1     Result = 11

operand1 = 5  operand2 = 9  cmd = 2     Result = -4

operand1 = 1  operand2 = 1  cmd = 2     Result = 0

operand1 = 2  operand2 = 3  cmd = 2     Result = -1

operand1 = 3  operand2 = 5  cmd = 2     Result = -2

operand1 = 4  operand2 = 7  cmd = 2     Result = -3

operand1 = 5  operand2 = 9  cmd = 3     Result = 45
```

Simple Processor at the BCA Level

For the sake of brevity, we're showing only two modules of this example at the BCA level.

```
#include "systemc.h"
#include "sc_mslib.h"

# define ADD 1
# define SUBTRACT 2
# define MULTIPLY 3
# define MYMAX 5

SC_MODULE(key)
{
  // ports
  sc_outmaster<int, sc_fullHandshake<int> > cmd;
  sc_outslave<int, sc_enableHandshake<int> > op1;
  sc_outslave<int, sc_enableHandshake<int> > op2;

  sc_in_clk    clk;
  sc_in<int>   rstp;

  static int array1[MYMAX];
  static int array2[MYMAX];

  int i, state;

  void key_f ()
    {
      if (rstp == 0)
        {
          state = i = 0;
          cmd.d = 1;
          cmd.req = false;
        }
      else
        {
          switch (state)
            {
            case 0:
```

```
            cmd.req = false;
            if (cmd.ack == false)
              state = 1;
               else
              state = 0;
               break;
          case 1:
            cmd.req = true;
            cmd.d = ADD;
            state = 2;
            break;
          case 2:
            cmd.req = true;
            cmd.d = SUBTRACT;
            state = 3;
            break;
          case 3:
            cmd.req = true;
            cmd.d = MULTIPLY;
            state = 0;
            break;
          }

        if (op1.en == true)
          {
            if (i == MYMAX-1)
              i = 0;
            else
              i++ ;
            op1.d = array1[i];
          }
        if (op2.en == true)
          {
            op2.d = array2[i];
          }
      }
  }

  // constructor
  SC_CTOR(key)
    {
      SC_METHOD(key_f)
      sensitive << clk;
    }
};
```

```
#include "systemc.h"
#include "sc_mslib.h"

# define idle      0
# define transmit  1

SC_MODULE(ctrl)
{
  // ports
  sc_inslave<int, sc_fullHandshake<int> > cmd_in;
  sc_inmaster<int, sc_enableHandshake<int> > op1_in;
  sc_inmaster<int, sc_enableHandshake<int> > op2_in;

  sc_outmaster<int, sc_enableHandshake<int> > op1_out;
  sc_outmaster<int, sc_enableHandshake<int> > op2_out;
  sc_outmaster<int, sc_enableHandshake<int> > cmd_out;
  sc_inmaster<int, sc_enableHandshake<int> > result;

  sc_outmaster<int, sc_enableHandshake<int> > display;

  // sc_inslave<bool> rstp;
  // sc_inslave<bool> clk;
  //
  sc_in_clk clk;
  sc_in<int> rstp;

  static int state;

  void ctrl_f ()
    {
      cmd_in.ack = false;
      if (rstp == 0)
        {
          state = idle;
          cmd_in.ack = false;
        }
      else
        {
          if (cmd_in.req == true)
            {
              //state = transmit;
              cmd_out.en = true;
```

```
            op1_in.en = true;
            op2_in.en = true;
            op1_out.en = true;
            op2_out.en = true;
            result.en = true;
            display.en = true;

            int x = cmd_in.d;
            cmd_out.d = x;
            //  cout << "cmd =   "<< x ;

            x = op1_in.d;
            op1_out.d = x;
            // cout << " op1 = "<< x;

            x = op2_in.d;
            op2_out.d = x;
            // cout << " op2 = "<< x << endl;

            x = result.d;
            display.d = x;
            // cout << "    Result = " << x << endl;
            // cmd_in.ack = true;
          }
        else
          {
            int x = result.d;
            display.d = x;
            //  cout << "    Result2 = " << x << endl;
          }
      }
  }

  // constructor of the module
  SC_CTOR(ctrl)
    {
      SC_METHOD(ctrl_f);
      sensitive << clk;
    }
};
```

Index