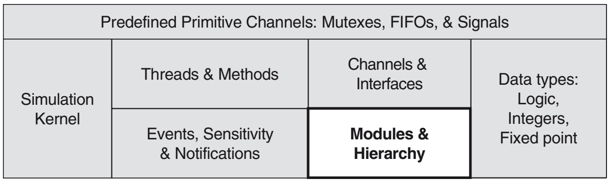
Chapter 12
More on Ports & Interfaces

Specialized & sc_export


Predefined Primitive Channels: Mutexes, FIFOs, & Signals
Simulation Kernel
Threads & Methods
Channels & Interfaces
Data types: Logic, Integers, Fixed point
Events, Sensitivity & Notifications
Modules & Hierarchy

This chapter continues our discussion of ports as we go beyond the basics and explore more advanced concepts. We start out with a look at some standard interfaces that can be used to build ports. Next, we discuss built-in specialized ports and their conveniences, especially with regard to static sensitivity. Finally, we present the concept of port arrays, and finish the chapter with a different type of port, **`sc_export`**<*T*>.

12.1 Standard Interfaces

SystemC provides a variety of standard interfaces that go hand in hand with the built-in channels discussed previously. This section describes these interfaces. This section is a more precise definition of the interface syntax, and it provides a basis for creating custom channels that will be discussed in the following chapter.

12.1.1 SystemC FIFO Interfaces

Two interfaces, **`sc_fifo_in_if`**<*T*> and**`sc_fifo_out_if`**<*T*>, are provided for the **`sc_fifo`**<*T*>channel. Together, these interfaces provide all of the methods implemented by **`sc_fifo`**<*T*>. In fact, the interfaces were defined prior to the creation of the channel. The channel simply becomes the place to implement the interfaces and holds the data implied by the functionality of a FIFO.

The interface, **`sc_fifo_out_if`**<*T*>, partially shown in the following figure, provides all the methods for output from a module into an **`sc_fifo`**<*T*>. The module pushes data onto the FIFO using **`write()`** or **`nb_write()`**. The **`num_free()`** indicates how many locations are free. The **`data_read_event()`** method may be used dynamically to wait for free space. We've discussed all of these methods in Chapter 8, Basic Channels, in the **`sc_fifo`**<*T*>channel discussion.

Notice in the following figure (Fig. 12.1) that the interface itself is templated on a class name just like the corresponding channel.

```
// Definition of sc_fifo<T> output interface
template <class T>
class sc_fifo_out_if: virtual public sc_interface {
public:
  virtual void write(const T& ) = 0;
  virtual bool nb_write(const T& ) = 0;
  virtual int num_free() const = 0;
  virtual const sc_event&
                data_read_event() const = 0;
};
```

Fig. 12.1 `sc_fifo` output interface definitions—abbreviated

The other interface, **`sc_fifo_in_if`**<*T*>, provides all the methods for input to a module from an **`sc_fifo`**<*T*>. The module pulls data from the FIFO using **`read()`** or **`nb_read()`**. The **`num_available()`** indicates how many locations are occupied, if any. The **`data_written_event()`** method may be used to dynamically wait for a new value to become available.

Again, all of these methods were discussed in Chapter 8 in the **`sc_fifo`**<*T*> channel discussion.

Here (Fig. 12.2) is the corresponding portion of the **`sc_fifo_in_if`**<*T*> interface definition:

```
// Definition of sc_fifo<T> input interface
template<class T>
class sc_fifo_in_if: virtual public sc_interface{
public:
  virtual void read( T& ) = 0;
  virtual T read() = 0;
  virtual bool nb_read( T& ) = 0;
  virtual int num_available()const = 0;
  virtual const sc_event&
            data_written_event() const = 0;
};
```

Fig. 12.2 `sc_fifo` input interface definitions—abbreviated

Something interesting to notice about the **`sc_fifo`**<*T*> interfaces is that if you use the **`read()`** and **`write()`** methods in your module and do not rely on the other methods, then your use of these interfaces is very compatible with the corresponding **`sc_signal`**<*T*> interfaces, which we will discuss next. In other words, you might want to simply swap out the interfaces and the channels; however, doing so would be dangerous. Remember **`sc_fifo`**<*T*>::**`read()`**[1] and **`sc_fifo`**<*T*>::**`write()`** are blocks waiting for the FIFO to empty; whereas, **`sc_signal`**<*T*>::**`read()`** and **`sc_signal`**<*T*>::**`write()`** arenon-blocking. This likely result is undesirable behavior.

[1]If you are having a hard time with this syntax, refer to the C++ scope resolution operator in Appendix A.

12.1.2 SystemC Signal Interfaces

Similar to sc_**fifo**<*T*>, two interfaces, **sc_signal_in_if**<*T*> and **sc_signal_inout_if**<*T*>, are provided for the **sc_signal**<*T*>channel. These two interface classes provide all of the methods provided by **sc_signal**<*T*>. Again, the interfaces were defined prior to the creation of the channel. The channel simply becomes the place to implement the interfaces and provides the request-update behavior implied for a signal.

Here (Fig. 12.3) is a portion of the **sc_signal_inout_if**<*T*> interface

```
// Definition of sc_signal<T> input/output interface
template<class T>
class sc_signal_inout_if: public sc_signal_in_if<T>
{
public:
  virtual void write( const T& ) = 0;
};
```

Fig. 12.3 **sc_signal** input/output interface definitions—abbreviated

definition:

There are two rather interesting things to notice in the preceding interface. First, if you need an **sc_signal**<*T*> output interface, use the **sc_signal_inout_if**<*T*>. This interface lets a module have access to the value of an output signal directly through a read, rather than being forced to keep a local copy in the manner required by VHDL.

Previous versions of SystemC included an **sc_signal_out_if**<*T*>interface that was type defined to **sc_signal_inout_if**<*T*>. That interface has been deprecated, and so you may see this in older code.

The update portion of the behavior is provided as a result of a call to **request_update()** that is provided indirectly as a result of a call from **sc_signal**<*T*>::**write()**. The update is implemented with the protected **sc_signal**<*T*>::**update()** method call. The **sc_signal_in_if**<*T*> interface (Fig. 12.4) provides access to the results through **sc_signal**<*T*>::**read()**.

```
// Definition of sc_signal<T> input interface
template<class T>
class sc_signal_in_if: virtual public sc_interface {
public:
  virtual const sc_event&
                  value_changed_event() const = 0;
  virtual const T& read() const = 0;
  virtual bool event() const = 0;
};
```

Fig. 12.4 **sc_signal** input interface definitions

12.1.3 sc_mutex and sc_semaphore Interfaces

The two channels, **`sc_mutex`** and **`sc_semaphore`**, also provide interfaces (Figs. 12.5 & 12.6) for use with ports. It is interesting to note that neither interface provides any event methods for sensitivity. If you require event sensitivity, you must write your own channels and interfaces as discussed in the next chapter.

```
// Definition of sc_mutex_if interface
class sc_mutex_if: virtual public sc_interface {
public:
  virtual int lock() = 0;
  virtual int trylock() = 0;
  virtual int unlock() = 0;
};
```

Fig. 12.5 **`sc_mutex`** interface definitions

```
// Definition of sc_semaphore_if interface
class sc_semaphore_if: virtual public sc_interface
{
public:
  virtual int wait() = 0;
  virtual int trywait() = 0;
  virtual int post() = 0;
  virtual int get_value() const = 0;
};
```

Fig. 12.6 **`sc_semaphore`** interface definitions

12.2 Sensitivity Revisited: Event Finders and Default Events

Recall from Chapter 6, Concurrency, that processes can be made sensitive to events. Also recall from Chapter 8, Basic Channels, that standard channels often provide methods that provide references to events (e.g., **`sc_fifo`**::**`data_written_event()`**). Since ports are defined on interfaces to channels, it is only natural to want sensitivity to events defined on those channels.

For example, it might be nice to create an **`SC_METHOD`** process statically sensitive to the **`data_written_event()`** or perhaps to monitor an **`sc_signal`**<*T*> for any change in the data using the **`value_changed_event()`**. You might even want to monitor a subset of possible events such as a positive edge transition (i.e., **`false`** to **`true`**) on a **`sc_signal<bool>`**.

Using static sensitivity for the situations above has a hidden complexity. Ports are pointers that become initialized during elaboration, and they are undefined at the time when the **`sensitive`** method needs to know about them. SystemC provides a solution for this difficulty in the form of a special class, the **`sc_event_finder`**.

The **sc_event_finder** defers the determination of the actual event until after elaboration. Unfortunately, the **sc_event_finder** has a minor complication. An **sc_event_finder** must be defined for each event defined by the interface. Thus, it is commonplace to define template specializations of port/interface combinations to instantiate a port and to include an **sc_event_finder** in the specialized class.

Suppose you want to create a port with sensitivity to the positive edge event of a Boolean signal port using the **sc_signal_in_if<bool>::posedge_event()** member function as shown in the following example (Fig. 12.7):

```
 classeslx_port
   :public sc_port<sc_signal_in_if<bool>,1>
 {
public:
// Use a typedef to shorten syntax below
   typedef sc_signal_in_if<bool>if_type;
   sc_event_finder& ef_posedge_event() const {
     return *newsc_event_finder_t<if_type>(
                     *this,
                     &if_type::posedge_event
                   );
   }//end ef_posedge_event
 };
```

Fig. 12.7 Example of specialized port[2] implementing an event finder

Let's examine the preceding example. First, our custom event finder is inheriting from a specialized port on the second line. Second, to save some typing, we've created a **typedef** called if_type, which refers to the interface specialization. The new method, ef_posedge_event**()**[3], creates a new **event_finder** object and returns a reference to it. The constructor for an **sc_event_finder** takes two arguments: a reference to the port being found (***this**), and a reference to the member function, (**posedge_event()**) that returns a reference to the event of interest. The preceding example returns a reference to the event finder, which can be used by **sensitive**.

Now, the preceding specialization may be used as follows (Fig. 12.8):

```
 SC_MODULE(my_module) {
   eslx_port my_p;
   …
   SC_CTOR(…) {
     SC_METHOD(my_method);
     sensitive<< my_p.ef_posedge_event();
   }
   void my_method();
   …
 };
```

Fig. 12.8 Example of event finder use

[2] See next section for a discussion of specialized ports.

[3] The prefix ef_ is a convention. Some groups might prefer a suffix, _ef. In any case, a convention should be adopted.

A related and useful concept for sensitivity lists in SystemC is the ability to be sensitive to a port. The idea is that a process sensitive to a port, typically an **SC_METHOD**, is concerned with any change on that port. Obviously, this may be coded similar to Fig. 12.7 using **value_changed_event()** instead of **posedge_event()**.

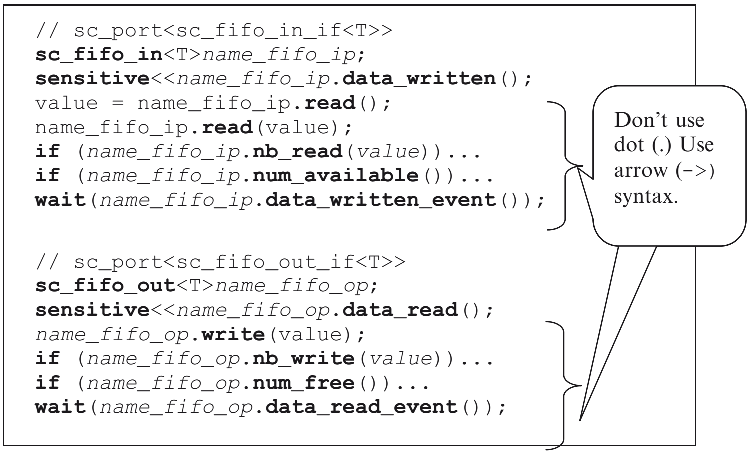
As a syntactical simplification, SystemC also allows specifying a port name if and only if the associated interface provides a method called **default_event()** that returns a reference to an **sc_event**. The standard interfaces for **sc_signal**<*T*> and **sc_buffer**<*T*> provide this method. If you design your own interfaces, you will need to supply this method yourself.

12.3 Specialized Ports

Event finders are not particularly difficult to code; however, they are additional coding and understanding the implementation is challenging. To reduce that burden, SystemC provides a set of template specializations that provide port definitions on the standard interfaces and include the appropriate event finders.

It is important to know the port specializations for two reasons. First, you will doubtless have need for the common event finders at some point. Second, you will encounter their use in code from other engineers.

Let's take a look at the syntax of FIFO specializations (Fig. 12.9):



```
// sc_port<sc_fifo_in_if<T>>
sc_fifo_in<T>name_fifo_ip;
sensitive<<name_fifo_ip.data_written();
value = name_fifo_ip.read();
name_fifo_ip.read(value);
if (name_fifo_ip.nb_read(value))...
if (name_fifo_ip.num_available())...
wait(name_fifo_ip.data_written_event());

// sc_port<sc_fifo_out_if<T>>
sc_fifo_out<T>name_fifo_op;
sensitive<<name_fifo_op.data_read();
name_fifo_op.write(value);
if (name_fifo_op.nb_write(value))...
if (name_fifo_op.num_free())...
wait(name_fifo_op.data_read_event());
```

Fig. 12.9 Syntax of FIFO port specializations

These specializations have a minor downside that has to do with how ports are to be referenced. Notice in the following syntax figures that methods such as **read()** are defined. Recall from the last chapter that processes invoke port methods using the pointer operator (->). With specialized ports, you may also use dot (e.g., my_sig.**read()**). This syntax has the unfortunate effect of creating bad habits that could cause you to stumble later.

You may still use the pointer methods in processes. With exception to the new sc_event_finder methods and initialization, we recommend you use the arrow form whenever possible.

GUIDELINE: Use dot (.) in the elaboration section of the code, but use arrow (->) in processes.

This style will help you differentiate port accesses from local channel accesses and reduce confusion.

Let's look at an example (Fig. 12.10) using the FIFO port specializations using the guideline:

Now we'll examine specialized ports (Fig. 12.11) for evaluate-update channels such as **`sc_signal`**<*T*>. We left out the obvious duplication of member functions

```
// Equalizer.h
SC_MODULE(Equalizer) {
  sc_fifo_in<double> raw_fifo_ip;
  sc_fifo_out<double> equalized_fifo_op;
  void equalizer_thread();
  SC_CTOR(Equalizer) {
    SC_THREAD(equalizer_thread);
      sensitive<< raw_fifo_ip.data_written();
  }
};
```

Only available in port specialization.

```
// Equalizer.cpp
void Equalizer::equalizer_thread() {
  for(;;) {
    double sample, result;
    wait(); // uses static sensitivity
    raw_fifo_ip->nb_read(sample);
    ... /* process data */
    equalized_fifo_op->write(result);
  }//endforever
}
```

Fig. 12.10 Example using FIFO port specializations

such as **`read()`** that are better handled using the pointer operator (->).

In the **`sc_signal speicalized port`** syntax (Fig. 12.11), there are several features to note. First, there is the **`initialize()`** method. This method may be used at elaboration to establish the initial values of signal ports. This approach models start-up conditions properly, rather than waiting a delta-cycle and synchronizing processes at the start.

We also included one additional syntax that we especially don't like[4], the assignment operator (=), on the last line. When used, this operator can be especially confusing. Unless you realize the name on the left is a signal port, the behavior will seem bizarre. Remember that signals have an evaluate-update behavior; therefore the

[4] Some of this syntax was provided for backwards compatibility with earlier versions of SystemC (specifically 1.x).

```
// sc_port<sc_signal_in_if<T>>
 sc_in<T> name_sig_ip;
 sensitive << name_sig_ip.value_changed();

 // Additional sc_in specializations...
 sc_in<bool> name_bool_sig_ip;
 sc_in<sc_logic >name_log_sig_ip;
 sensitive << name_sig_ip.pos();
 sensitive << name_sig_ip.neg();

 // sc_port<sc_signal_out_if<T>>
 sc_inout<T> name_sig_op;
 sensitive << name_sig_op.value_changed();
 sc_inout_resolved<N> name_rsig_op;
 sc_inout_rv<N> name_rsig_op;
 sc_inout<T> name_rsig_op;
 sc_inout_resolved<T> name_rsig_op;
 sc_inout_rv<T> name_rsig_op;
 // everything under sc_in<T> plus the following...
 name_sig_op.initialize(value);
 name_sig_op = value; //  <-- DON'T USE!!!
```

Fig. 12.11 Syntax of signal port specializations

changes from this assignment are not reflected until the next delta-cycle. This is quite different from ordinary assignment, and very confusing to most programmers.

GUIDELINE: To avoid confusion, never use the assignment operator with **`sc_signal<T>`** or **`sc_port<sc_signal_inout_if`**`<T>`** `>`**. Instead, use the **`write()`** method.

Let's look at an example using signal port specializations (Fig. 12.12 & 12.13). This example is a typical hardware block, a 32-bit linear feedback shift register (LFSR) commonly used with built-in self-test (BIST). Notice the use of **`pos()`** and **`initialize()`**.

```
//FILE: LFSR_ex.h
SC_MODULE(LFSR_ex) {
  // Ports
  sc_in<bool> sample;
  sc_out<sc_int<32>> signature;
  sc_in<bool> clock;
  sc_in<bool> reset;
  // Constructor
  SC_CTOR(LFSR_ex) {
    // Register process
    SC_METHOD(LFSR_ex_method);
    sensitive<< clock.pos() << reset;
    signature.initialize(0);
  }
  // Process declarations & Local data
  void LFSR_ex_method();
  sc_int<32> LFSR_reg;
};
```

Fig. 12.12 Example of signal port specializations—header

```
//FILE: LFSR_ex.cpp
#include "LFSR.h"
void LFSR_ex::LFSR_ex_method() {
  if (reset->read() == true) {
  LFSR_reg = 0;
  signature->write(LFSR_reg);
  }
else {
    bool lsb =LFSR_reg[31]^LFSR_reg[25]^LFSR_reg[22]
             ^LFSR_reg[21]^LFSR_reg[15]^LFSR_reg[11]
             ^LFSR_reg[10]^LFSR_reg[ 9]^LFSR_reg[ 7]
             ^LFSR_reg[ 6]^LFSR_reg[ 4]^LFSR_reg[ 3]
             ^LFSR_reg[ 1]^LFSR_reg[ 0]
             ^ sample->read();
    LFSR_reg.range(31,1) = LFSR_reg.range(30,0);
    LFSR_reg[0] = lsb;
    signature->write(LFSR_reg);
  }//endelse
}
```

Fig. 12.13 Example of signal port specializations—implementation

12.4 The SystemC Port Array and Port Policy

The **sc_port**<*T*> provides additional template parameters we have not yet discussed: the array size parameter and the port policy parameter. The array size parameter allows the creation of a number of identical ports. This construct is referred to as a multi-port or port array. The port policy specifies whether zero, one, or all ports must be connected.

For example, a communication system might have a number of T1 interfaces all with the same connectivity. Another example might be an abstract hierarchical communication that may have any number of devices connected to it. The full **sc_port**<*T*> syntax follows (Fig. 12.14).

```
  sc_port<interface[,N[,POL]]> portname;
  // N=0..MAX Default N=1
  // POL is of type sc_port_policy
  // POL defaults to SC_ONE_OR_MORE_BOUND
```

Fig. 12.14 Syntax of **sc_port**<> declaration complete

N indicates the number of channels to be connected to the port. When N = 0, we have a special case that allows an almost unlimited number of ports. In other words, you may connect any number of channels to the port.

POL is of type sc_port_policy and it is an enumerated type and has three legal values:

- **SC_ONE_OR_MORE_BOUND**
- **SC_ZERO_OR_MORE_BOUND**
- **SC_ALL_BOUND**

The value of POL enables different checking regarding the connectivity to the port. The default is **SC_ONE_OR_MORE_BOUND**. **SC_ZERO_OR_MORE_BOUND** allows a port with no connections. **SC_ALL_BOUND** requires that there are N and only N channels connected to the port (unless N=0 and then the checking associated with **SC_ONE_OR_MORE_BOUND** is used), which was the only implementation in earlier versions of SystemC.

An example with a drawing may help with understanding the use of multiports. In the figure (Fig. 12.15), four distinct channels connected to four ports T1_ip[0…3] on the left, and on the right, nine separate channels connect to nine ports request_op[0…8]. The block with the ports represents an imaginary switch.

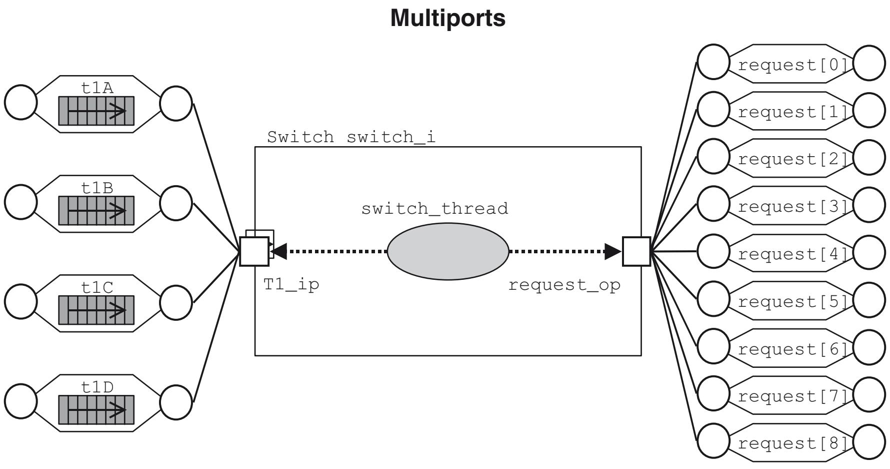


Fig. 12.15 Illustration of **sc_port**<*T*> array connectivity

Here is the header code (Fig. 12.16) for the switch in the above drawing:

```
//FILE: Switch.h
  SC_MODULE(Switch) {
    sc_port<sc_fifo_in_if< int>
            ,5
            ,SC_ONE_OR_MORE_BOUND
            > T1_ip;
    sc_port<sc_signal_inout_if<bool>
            ,0
            > request_op;
    ...
  };
```

Fig. 12.16 Example of **sc_port** array declaration

Channels are connected to port arrays the same way ordinary ports are connected, except port arrays have more than one connection. In fact, the basic port syntax simply relies on the default that N = 1. Each connection is assigned a position in the array on a first-connected first-position basis.

Here (Fig. 12.17) is the corresponding example for the connections:

```
//FILE: Board.h
#include "Switch.h"
SC_MODULE(Board) {
  Switch switch_i;
sc_fifo<int> t1A, t1B, t1C, t1D;
sc_signal bool> request[9];
  SC_CTOR(Board): switch_i("switch_i")
  {
    // Connect 4 T1 channels to the switch
    switch_i.T1_ip(t1A);
    switch_i.T1_ip(t1B);
    switch_i.T1_ip(t1C);
    switch_i.T1_ip(t1D);
    // Connect 9 request channels to the
    // switch request output ports
    for (unsigned i=0;i!=9;i++) {
      switch_i.request_op(request[i]);
    }//endfor
    ...
  }//end constructor
  ...
};
```

From preceding example.

Fig. 12.17 Example of **`sc_port`** array connections

The preceding example illustrates several things. First, a fixed port array of size 4 is connected directly to four FIFO channels. Second, an unbounded array is connected to an array of channels using a for-loop.

Access to port arrays from within a process is accomplished using the array syntax. This class also provides a method, **`size()`**, that may be used to examine the declared port size. This method is useful for situations where the array bounds are unknown (i.e., N = 0 or using **`SC_ONE_OR_MORE_BOUND`** or **`SC_ZERO_OR_MORE_BOUND`**).

Here (Fig. 12.18) is the code implementing the process accessing the multiports from within the Switch module:

Notice that the **`size()`** method requires the dot operator because it's defined in the specialized port class (e.g., request_op or T1_ip), rather than in the external channel (e.g., request[i] or t1A, t1B, t1C, t1D). On the other hand, port access to the channel uses the arrow operator as would be expected.

One current syntactical downside to **`wait()`** syntax may be seen in the preceding syntax. If you need to use "any event in the attached channels," current syntax requires an explicit listing.

```
//FILE: Switch.cpp
void Switch::switch_thread() {
  // Initialize requests
  for (unsigned i=0;i!=request_op.size();i++) {
    request_op[i]->write(true);
  }//endfor
  // Startup after first port is activated
  wait(T1_ip[0]->data_written_event()
      |T1_ip[1]->data_written_event()
      |T1_ip[2]->data_written_event()
      |T1_ip[3]->data_written_event()
  );
  while(true) {
    for (unsigned i=0;i!=T1_ip.size();i++) {
      // Process each port...
      int value = T1_ip[i]->read();
    }//endfor
  }//endwhile
}//end Switch::switch_thread
```

Fig. 12.18 Example of **`sc_port`** array access

There is an alternate possibility with dynamic threads. One could create and launch a separate thread to monitor each port and provide communication back via a shared local variable. We will examine this feature in the Advanced Topics chapter.

12.5 SystemC Exports

There is a second type of port called the **`sc_export`**<*T*>. The export is similar to standard ports in that the declaration syntax is defined on an interface. However, this port differs in connectivity. The idea of an **`sc_export`**<*T*> is to move the channel inside the defining module, thus hiding some of the connectivity details and using the port externally as though it were a channel. The following figure (Fig. 12.19) illustrates this concept:

Contrast this concept with Fig. 11.9 where we originally investigated ports.

The observant programmer might ask, Why use **`sc_export`** at all? After all, one could just access the internal **`sc_channel`** instance name directly using a hierarchical access. That approach works only if the interior channel is publicly accessible. For an IP provider, it may be desirable to export only specific channels and keep everything else private. Thus, **`sc_export`**<*T*> allows control over the interface.

Another reason for using **`sc_export`**<*T*> is to provide multiple interfaces at the top level. For example, consider the situation where you wish to create a channel that has two distinct interfaces for **`sc_signal`**<*T*>. Normally, it is not possible to inherit more than once from the same base class. However, with **`sc_export`**<*T*>, it is now possible to do this. Without **`sc_export`**<*T*>, we are

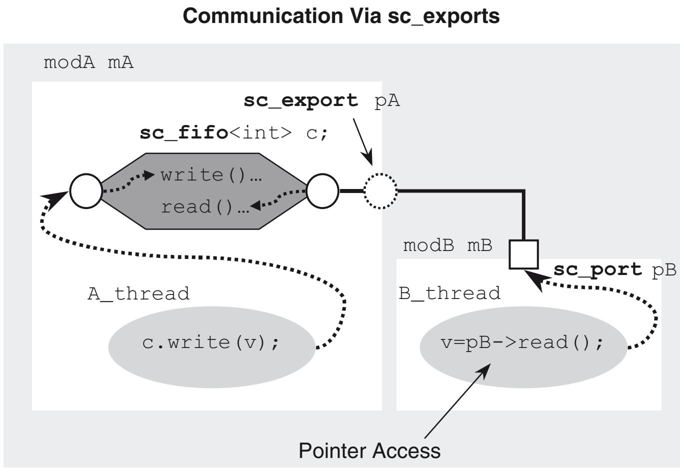


Fig. 12.19 How **`sc_export`** works

left to using the hierarchical channel, which allows for only a single top-level set of interfaces.

With an export, each **`sc_export`**<*T*> contains a specific interface. Since a connection is not required, **`sc_export`**<*T*> also allows creation of hidden interfaces. For example, a debug or test interface might be used internally by an IP provider, but not documented for the end user. Additionally, an export lets a development team or IP provider develop an instrumentation model. This model can be used extensively during architectural exploration and then dropped during regression runs when visibility is needed less and performance is key.

A hidden interface has the benefit of making the channel simpler to read and understand in the module where the IP is instantiated. The channel is easier to read since any programmatic instantiation of channels is hidden as well as much of the connectivity.

Another reason for using **`sc_export`**<*T*> is communications efficiency down the SystemC hierarchy. The **`sc_export`**<*T*> provides symmetry to the direction of C++ calls. Without **`sc_export`**<*T*>, SystemC ports would require additional channels and processes to allow an external process to pull information from a lower point in the model hierarchy. Efficiency is gained because **`sc_export`**<*T*> allows direct access to information (data) without intermediate channels.

The **`sc_export`**<*T*> syntax shown following (Fig. 12.20) **`sc_port`**<*T*>, but without the additional template parameters. This means that you cannot currently have an array for **`sc_export`**<*T*>.

```
sc_export<interface> portname;
```

Fig. 12.20 Syntax of **`sc_export`** declaration

Connectivity to an **sc_export**<*T*> requires some slight changes since the channel connections have now moved inside the module. Thus, we have (Fig. 12.21):

```
SC_MODULE(modulename) {
  sc_export<interface> portname;
  channel cinstance;
  SC_CTOR(modulename) {
    portname(cinstance);
  }
};
```

Fig. 12.21 Syntax **sc_export** internal binding to channel

Let's look at a simple example (Fig. 12.22). This example provides a process that is toggling an internal signal periodically. The **sc_export**<*T*> in this case is simply the toggled signal. For compactness, this example includes the entire module definition in the header.

```
SC_MODULE(clock_gen) {
  sc_export<sc_signal<bool>> clock_xp;
  sc_signal<bool> oscillator;
  SC_CTOR(clock_gen) {
    SC_METHOD(clock_method);
    clock_xp(oscillator); // connect sc_signal
                          // channel
                          // to export clock_xp
    oscillator.write(false);
  }
  void clock_method() {
    oscillator.write(!oscillator.read());
    next_trigger(10,SC_NS);
  }
};
```

Fig. 12.22 Example of simple **sc_export** declaration

To use the above **sc_export**<*T*>, we provide (Fig. 12.23) the corresponding instantiation of this simple module.

```
#include "clock_gen.h"
…
clock_gen clock_gen_i(“clock_gen_i”);
collision_detector cd_i(“cd_i”);
// Connect clock
cd_i.clock(clock_gen_i.clock_xp);
…
```

Fig. 12.23 Example of simple **sc_export** instantiation

Another powerful possibility with **sc_export**<*T*> is to let interfaces be passed up the design hierarchy as illustrated in the next figure (Fig. 12.24).

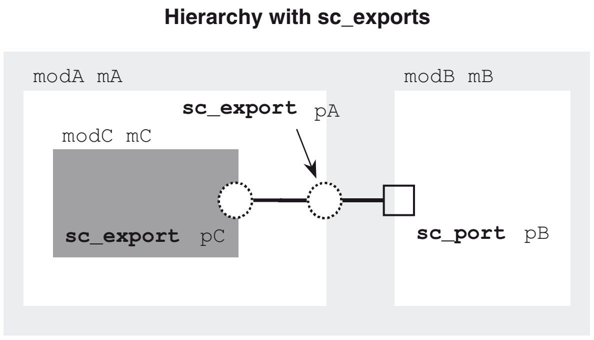


Fig. 12.24 **sc_export** used with hierarchy

Just like **sc_port**<*T*>, the **sc_export**<*T*> can be bound directly to another **sc_export**<*T*> in the hierarchy. Here (Fig. 12.25) is how to accomplish this binding:

```
SC_MODULE(modulename) {
  sc_export<interface> xportname;
  module minstance;
  SC_CTOR(modulename)
  , minstance("minstance")
  {
    xportname(minstance.subxport);
  }
};
```

Fig. 12.25 Syntax of **sc_export** internal binding to submodule

sc_export<*T*> has some caveats that may not be obvious. First, it is not possible to use **sc_export**<*T*> in a static sensitivity list. On the other hand, you can access the interface via the pointer operator (->). Thus, one can use **wait**(xportname->event()) on suitably defined interfaces accessed within an **SC_THREAD** process.

Second, as previously mentioned, it is not possible to have an array of **sc_export**<*T*> in the same manner as **sc_port**<*T*>. On the other hand, suitable channels may allow multiple connections, which may make this issue moot.

The following is an example of how an **sc_export**<*T*> might be used to model a complex bus including an arbiter to be provided as an IP component. First, let's look at the customer view (Fig. 12.26):

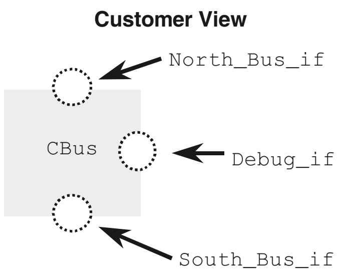


Fig. 12.26 Example of customer view of IP

This view would be defined using the following (Fig. 12.27) header file:

```
//CBus.h
#include "CBus_if.h"
class North_bus; // Forward declarations
class South_bus;
class Debug_bus;
class CBus_rtl;
SC_MODULE(CBus) {
  sc_export<CBus_North_if> north_p;
  sc_export<CBus_South_if> south_p;
  SC_CTOR(CBus);
private:
  North_bus* nbus_ci;
  South_bus* sbus_ci;
  Debug_bus* debug_ci;
  CBus_rtl*  rtl_i;
};
```

Fig. 12.27 Example of **sc_export** applied to a bus

Notice how the preceding code is independent of the implementation, and the end user is not compelled to hook up either bus. In addition, the debug interface is not provided in this example header. Here is the implementation view (Fig. 12.28):

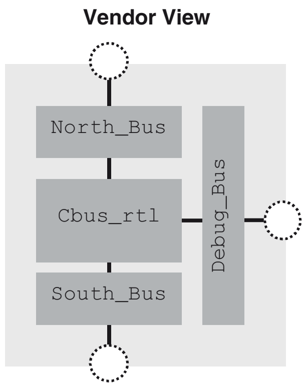


Fig. 12.28 Example of vendor view of IP

Here (Fig. 12.29) is the implementation code, which may be kept private:

```
//FILE: CBus.cpp
#include "CBus.h"
#include "North_bus.h"
#include "South_bus.h"
#include "Debug_bus.h"
#include "CBus_rtl_bus.h"
CBus::CBus(sc_module_name nm): sc_module(nm) {
  // Local instances
  nbus_ci  = new North_bus("nbus_ci");
  sbus_ci  = new South_bus("sbus_ci");
  debug_ci = new Debug_bus("debug_ci");
  rtl_i    = new CBus_rtl("rtl_i");
  // Export connectivity
  north_p(*nbus_ci);
  south_p(*sbus_ci);
  // Implementation connectivity
  …
}
```

Fig. 12.29 Example of `sc_export` applied to a bus constructor

In the preceding code, notice that the debug interface is not provided to the customer. Providing this interface would be an optional aspect of the IP that could be connected at compile time using an **`#ifdef`** or at run time using a "switch" from a control file or the command line.

12.6 Connectivity Revisited

Let's review port connectivity. The following diagram (Fig. 12.30) is copied from the previous chapter. It should become second nature to understand how to accomplish all the connections illustrated.

All of the possible connections are illustrated in this one figure. This figure is a handy reference when reviewing the SystemC connection rules, which are listed below:

1. A process may communicate with another process in the same module using a channel. For example, process `pr2` to process `pr3` via interface `ifX` on channel `c2i`.
2. A process may communicate with another process in the same module using an event to synchronize exchanges of information through data variables instantiated at the module level (e.g., within the module class definition). For example, process `pr2` to process `pr1` via event `ev1`.

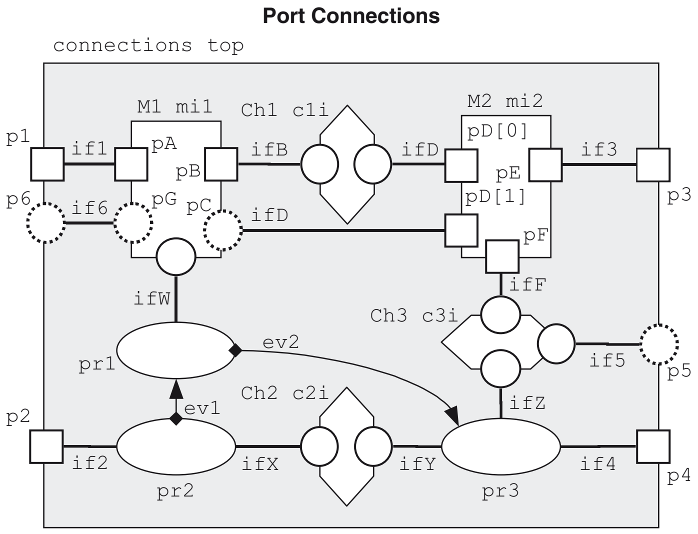


Fig. 12.30 Connectivity possibilities

3. A process may communicate with a process upwards in the design hierarchy using the interfaces accessed via **sc_port**<*T*>. For example, process pr3 via port p4 using interface if4.
4. A process may communicate with processes in submodule instances via interfaces to channels connected to the submodule ports. For example, process pr3 to module mi2 via interface ifZ on channel c3i.
5. An **sc_export**<*T*> may connect to another **sc_export**<*T*> via interfaces to local channels. For example, port p5 to channel c3i using interface if5.
6. An **sc_port**<*T*> may connect directly to an **sc_port**<*T*> of submodules. For example, port p1 is connected to port pA of submodule mi1.
7. An **sc_export**<*T*>may connect directly to an **sc_export**<*T*> of a submodule. For example, port p6 is directly connected to port pG of submodule mi1.
8. An **sc_port**<*T*> may connect indirectly to a process by letting the process access the interface. This is just a process accessing a port described previously. For example, process pr1 communicates with submodule mi1 through interface ifW.
9. An **sc_port**<*T*,*N*> array may be used to create multiple ports using the same interface. For example, pD[0] and pD[1] of submodule mi2.

Finally, we present an equivalent diagram (Fig. 12.31) to the preceding. In this diagram, channels appear as slightly thickened lines. An **sc_port**<*T*> is represented with a square containing a circle to indicate the presence of an interface. This style is often used to simplify the schematic representation at the expense of slightly hiding the underlying functionality. In the next chapter, we will investigate more complex channels known as hierarchical channels.

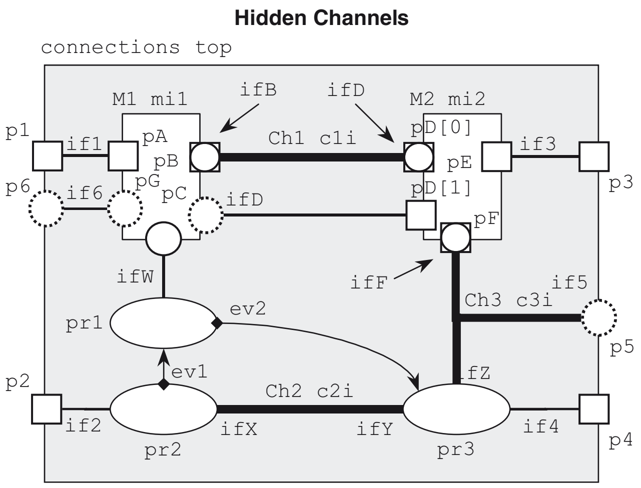


Fig. 12.31 Hidden channels

12.7 Exercises

For the following exercises, use the samples provided at www.scftgu.com.

Exercise 12.1: Examine, compile, and run the `static_sensitivity` example.

Exercise 12.2: Examine, compile, and run the `connections` example. See if you can identify all the connections shown in the figures in this chapter.