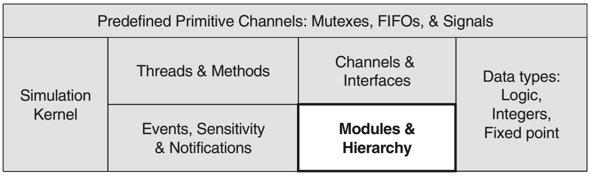
Chapter 11
Communication

Ports



This chapter describes SystemC's facilities for implementing connectivity, which enables orderly communication between modules.

11.1 Communication: The Need for Ports

Hierarchy without the ability to communicate between modules is not very useful, but what is the best way to communicate? There are two concerns: safety and ease of use. Safety is a concern because all activity occurs within processes, and care must be taken when communicating between processes to avoid race conditions. Events and channels are used to handle this concern.

Ease of use is more difficult to address. Let us dispense with any solution involving global variables, which are well known as a poor methodology. Another possibility is to have a process in an upper-level module. This process would monitor and manage events defined in instantiated modules. This mechanism is awkward at best.

SystemC takes an approach that lets modules use channels inserted between the communicating modules. SystemC accomplishes this communication with a concept called a port. Basically, a port is a pointer to a channel outside the module.

For simplicity, this chapter only covers the **sc_port**<*T*>. We will cover an alternate and related concept, the **sc_export**<*T*>, in a later chapter. An **sc_export**<*T*> is a pointer to a channel inside another module.

Consider the following example (Fig. 11.1):

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_11, © Springer Science+Business Media, LLC 2010

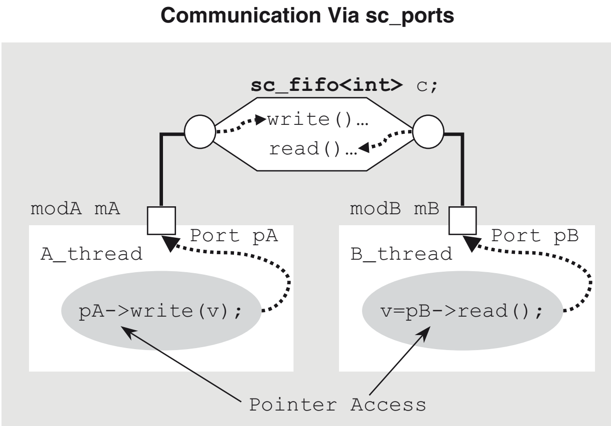


Fig. 11.1 Communication via ports

The process `A_thread` in module `modA` communicates a value contained in local variable `v` by calling the **`write`** method of the parent module's channel `c`. Process `B_thread` in module `modB` may retrieve a value via the **`read`** method of channel `c`.

Notice that access is accomplished via pointers `pA` and `pB`. Notice also that the two processes really only need access to the methods **`write`** and **`read`**. More specifically, `modA` only needs access to **`write`**, and **`modB`** only needs access to **`read`**. This separation of access is managed using a concept known as an interface, which is described in the next section.

11.2 Interfaces: C++ and SystemC

C++ defines a concept known as an abstract class. An abstract class is a class that is never used directly, but it is used only via derived subclasses. Partly to enforce this concept, abstract classes usually contain pure virtual functions. Pure virtual functions are not allowed to provide an implementation in the abstract class where they are defined as pure. This restriction in turn compels any class derived from the abstract class to override all the pure virtual functions, or in other words, the class derived from the abstract class must provide an implementation for all the pure virtual functions.

The following (Fig. 11.2) diagram illustrates the concept. Pure virtual functions are identified by 1) the keyword **`virtual`** and 2) the `=0;` to indicate they're pure.

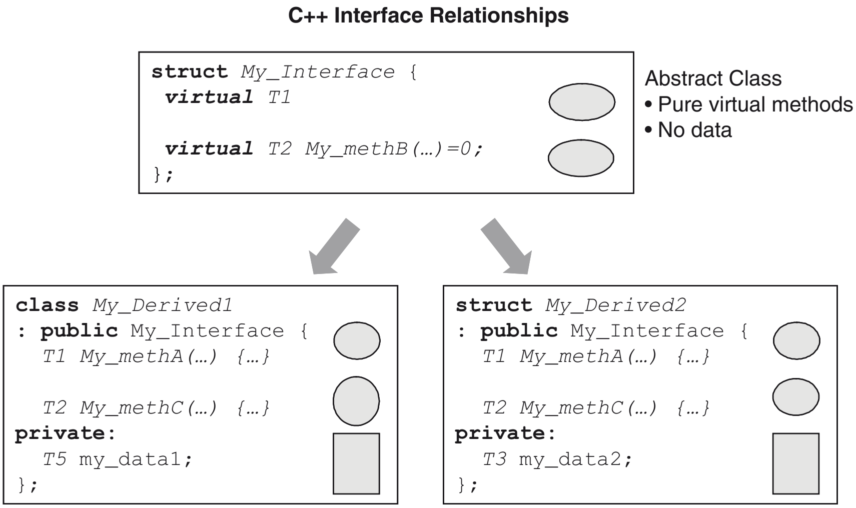


Fig. 11.2 C++ interface class relationships

If a class contains no data members and only contains pure virtual methods, it is known as an interface class. Here is a short example of an interface class:

The concept of interfaces has a powerful property when used with polymorphism. Recall from C++ that polymorphism is the following idea: A derived class can be processed by a function referencing the parent class.

```
class my_interface {
public:
  virtual void write(unsigned addr, int data) = 0;
  virtual int read(unsigned addr) = 0;
};
```

Fig. 11.3 Example of C++ interface

Consider the preceding figure (Fig. 11.3) of C++ interface class relationships. A function using `My_Interface` might access `My_methA()`. If the current object is of class My_Derived2, then the actual `My_methA()` call results in `My_Derived2::My_methA()`.

If an object is declared as a pointer to an interface class, it may be used with multiple derived classes. Suppose we define two derived classes as follows (Fig. 11.4):

```
class multiport_memory_arch: public my_interface {
public:
  void write(unsigned addr, int data) {
    mem[addr] = data;
  }// end write
  int read(unsigned addr) ) {
    return mem[addr];
  }//end read
private:
  int mem[1024];
};
```

```
class multiport_memory_RTL: public my_interface {
public:
  void write(unsigned addr, int data) {
    // complex details of RTL memory write
  }// end write
  int read(unsigned addr) ) {
    // complex details of RTL memory read
  }// end read
private:
  // complex details of RTL memory storage
};
```

Fig. 11.4 Example of two derivations from interface class

Now we write some C++ code to access the aforementioned derived classes.

```
void memtest(my_interface& mem) {
  // complex memory test
}

multiport_memory_arch fast;
multiport_memory_RTL  slow;
memtest(fast);
memtest(slow);
```

Fig. 11.5 Example of C++ interface

As seen in the preceding example (Fig. 11.5), the same code may access multiple variations of a design. You can think of an interface as the application programming interface (API) to a set of derived classes. This same concept is used in SystemC to implement ports.

DEFINITION: A SystemC interface is an abstract class that inherits from **`sc_interface`** and provides only pure virtual declarations of methods referenced by SystemC channels and ports. No implementations or data are provided in a SystemC interface.

We now provide the concise definition of the SystemC channel.

DEFINITION: A SystemC channel is a class that inherits from either **`sc_channel`** or from **`sc_prim_channel`**, and the channel should[1] inherit and implement one or more SystemC interface classes. A channel implements all the pure virtual methods of the inherited interface classes.

By using interfaces to connect channels, we can implement modules independent of the implementation details of the communication channels.

Consider the following diagram (Fig. 11.6):

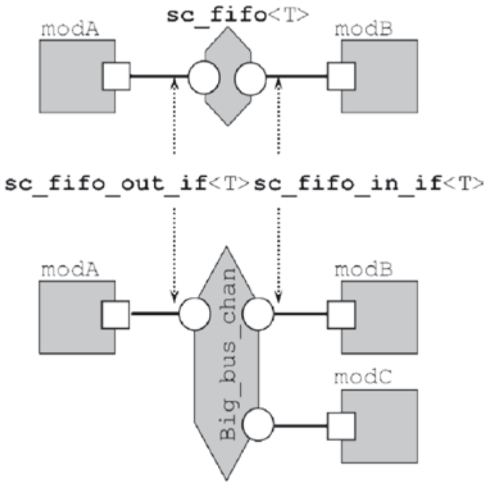


Fig. 11.6 The power of interfaces

In one design, modules `modA` and modB are connected via a FIFO. With no change to the definition of modA or modB, we can swap out the FIFO for a different channel. All that matters is for the interfaces used to remain constant. In this example, the interfaces are **`sc_fifo_out_if`**<*T*> and **`sc_fifo_in_if`**<*T*>. In the next few sections, the mechanics of using interfaces are described.

11.3 Simple SystemC Port Declarations

Given the definition of an interface, we now present the definition of a port.

DEFINITION A SystemC port is a class templated with and inheriting from a SystemC interface. Ports allow access of channels across module boundaries.

Specifically, the syntax of a simple SystemC port follows (Fig. 11.7):

[1]However, without an interface, a SystemC channel cannot be used with a SystemC port.

```
sc_port<interface> portname;
```

Fig. 11.7 Syntax of basic **sc_port**

SystemC ports are always defined within the module class definition. Here is a simple example (Fig. 11.8):

```
SC_MODULE(stereo_amp) {
  sc_port<sc_fifo_in_if<int> >  soundin_p;
  sc_port<sc_fifo_out_if<int> > soundout_p;
  …
};
```

Fig. 11.8 Example of defining ports within module class definition

Notice the extra blank space following the greater-than symbol (>). This is required C++ syntax when nesting templated classes.

11.4 Many Ways to Connect

Given the declaration of a port, we now address the issue of connecting ports, channels, modules, and processes. The following diagram (Fig. 11.9) illustrates the types of connections that are possible with SystemC:

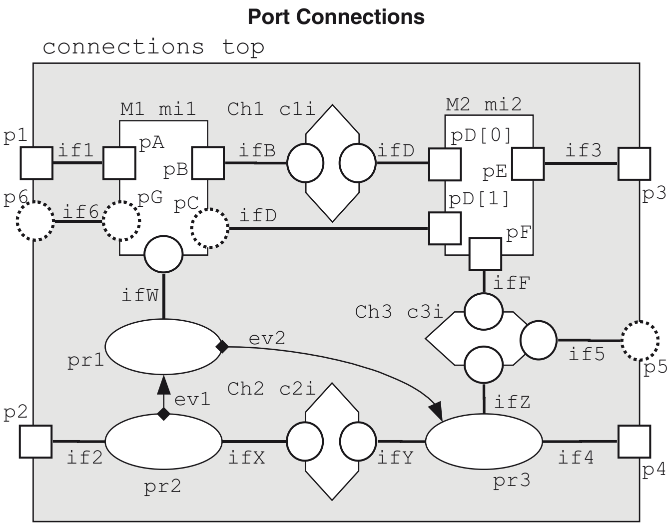


Fig. 11.9 Connectivity possibilities

This diagram (Fig. 11.9) is quite busy. Let's examine the pieces by name, and then discuss the rules of interconnection.

First, there are three modules represented with rectangles. The enclosing module instance is named `top`. The two submodule instances within top are named `mi1` and `mi2`.

Each of the modules has one or more ports represented with squares. Directional arrows within the ports indicate the primary flow of information. The ports for top are `p1`, `p2`, `p3`, `p4`, `p5`, and `p6`, which use interfaces named `if1`, `if2`, `if3`, `if4`, `if5`, and `if6`, respectively.

The ports for `mi1` are `pA`, `pB`, `pC`, and `pG`, which are connected to interfaces named `if1`, `ifB`, `ifD`, and `if6`, respectively.

Module `M1` also provides interfaces `ifW` and `if6`.

The ports for `mi2` are `pD[0]`, `pD[1]`, `pE`, and `pF`, which are connected to interfaces named `if3`, `ifD`, and `ifF`, respectively.

Next, three instances of channels represented with hexagonal shapes exist within `top`. These are named `c1i`, `c2i`, and `c3i`.

Each channel implements one or more interfaces represented by circles with a bent arrow. The arrow is intended to indicate the possibility of a call returning a value. It is possible for a channel to implement only a single interface. Channel `c1i` implements interfaces `ifB` and `ifD`. Channel `c2i` implements interfaces `ifX` and `ifY`. Finally, channel `c3i` implements interfaces `if5`, `ifF`, and `ifZ`.

Last, there are three processes named `pr1`, `pr2`, and `pr3`. For this description, we don't need to know what type of processes (i.e., threads vs. methods). There are two explicit events, `ev1` and `ev2` used for signaling between processes.

From this diagram, several rules may be observed. As we already know, processes may communicate with processes:

- At the same level either via channels or synchronized via events
- Outside the local design module through ports bound to channels by way of interfaces
- In submodule instances via interfaces to channels connected to the submodule ports or by way of interfaces through the module itself of an ***`sc_export`***

Any other attempt at inter-process communication is either forbidden or dangerous.

Ports may connect via interfaces only to local channels, ports of submodules, or to processes indirectly.

There are a few interesting features that will be discussed later. First, module instance `mi1` implements an interface `ifW`. Second, port `pD` appears to be an array of size 2. This is known as a port array. Finally, port `p5` and port `pC` illustrate the **`sc_export`**.

As a summary, let's view this information in a tabular format (Table 11.1).

Table 11.1. Ways to interconnect

From	To	Method
Port	Submodule	Direct connect via **sc_port**
Process	Port	Direct access by process
Submodule	Submodule	Local channel connection
Process	Submodule	Local channel connection or via **sc_export** **or** interface implemented by sub-module[a]
Process	Process	Events or local channel
Port	Local channel	Direct connect via **sc_export**

[a]In this case, the module is also known as a hierarchical channel, which will be discussed later.

11.5 Port Connection Mechanics

Modules are connected to channels after both the modules and channels have been instantiated. There are two syntaxes for connecting ports: by name and by position. Due to the error-prone nature of positional notation (especially since the number of ports on modules tends to be large and changes), the authors strongly prefer connection by name. Here are both syntaxes (Fig. 11.10):

```
mod_inst.portname(channel_instance); // Named
mod_instance(channel_instance,…); // Positional
```

Fig. 11.10 Syntax of port connectivity

An example should help greatly. We'll use a simple video mixer example with a color space transformation. For this example, we will use two standard SystemC interface classes, **sc_fifo_in_if** and **sc_fifo_out_if,** which support read() and write(value), respectively. First, we introduce the module definitions (Fig. 11.11, 11.12 & 11.13).

```
//FILE: Rgb2YCrCb.h
SC_MODULE(Rgb2YCrCb) {
  sc_port<sc_fifo_in_if<RGB_frame> >    rgb_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > ycrcb_po;
};
```

Fig. 11.11 Example of port interconnect setup (1 of 3)

```
//FILE: YCRCB_Mixer.h
SC_MODULE(YCRCB_Mixer) {
  sc_port<sc_fifo_in_if<float> >         K_pi;
  sc_port<sc_fifo_in_if<YCRCB_frame> >  a_pi,b_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > y_po;
};
```

Fig. 11.12 Example of port interconnect setup (2 of 3)

```
//FILE: VIDEO_Mixer.h
SC_MODULE(VIDEO_Mixer) {
  // ports
  sc_port<sc_fifo_in_if<YCRCB_frame> >  dvd_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > video_po;
  sc_port<sc_fifo_in_if<MIXER_ctrl> >   control;
  sc_port<sc_fifo_out_if<MIXER_state> > status;
  // local channels
  sc_fifo<float>       K;
  sc_fifo<RGB_frame>   rgb_graphics;
  sc_fifo<YCRCB_frame> ycrcb_graphics;
  // local modules
  Rgb2YCrCb   Rgb2YCrCb_i;
  YCRCB_Mixer YCRCB_Mixer_i;
  // constructor
  VIDEO_Mixer(sc_module_name nm);
  void Mixer_thread();
};
```

Fig. 11.13 Example of port interconnect setup (3 of 3)

Now, let's look at interconnection of the preceding modules using both named (Fig. 11.14) and positional (Fig. 11.15) syntaxes.

```
SC_HAS_PROCESS(VIDEO_Mixer);
VIDEO_Mixer::VIDEO_Mixer(sc_module_name nm)
: sc_module(nm)
, Rgb2YCrCb_i("Rgb2YCrCb_i")
, YCRCB_Mixer_i("YCRCB_Mixer_i")
{
  // Connect
  Rgb2YCrCb_i.rgb_pi(rgb_graphics);
  Rgb2YCrCb_i.ycrcb_po(ycrcb_graphics);
  YCRCB_Mixer_i.K_pi(K);
  YCRCB_Mixer_i.a_pi(dvd_pi);
  YCRCB_Mixer_i.b_pi(ycrcb_graphics);
  YCRCB_Mixer_i.y_po(video_po);
}
```

Fig. 11.14 Example of port interconnect by name

Although slightly more code than the positional notation, the named port syntax is more robust, and tools exist to reduce the typing tedium.

```
SC_HAS_PROCESS(VIDEO_Mixer);
VIDEO_Mixer::VIDEO_Mixer(sc_module_name nm)
 : sc_module(nm)
 {
   // Instantiate
   Rgb2YCrCb_iptr   = new Rgb2YCrCb(
                          "Rgb2YCrCb_i"
                        );
   YCRCB_Mixer_iptr = new YCRCB_Mixer(
                          "YCRCB_Mixer_i"
                        );
   // Connect
   (*Rgb2YCrCb_iptr)( rgb_graphics
                     ,ycrcb_graphics
                    );
   (*YCRCB_Mixer_iptr)( K
                       ,dvd_pi
                       ,ycrcb_graphics
                       ,video_po
                      );
 }
```

Fig. 11.15 Example of port interconnect by position

The problem with positional connectivity is that of keeping the ordering correct. In large designs, middle- and upper-level modules frequently have a large number of ports (potentially multiple 10s), and it is common to add or remove ports late in the design. Using a positional notation can quickly lead to debug problems. That is why we recommend avoiding the positional syntax entirely, and always using a named port approach.

GUIDELINE: Whenever possible, use the named port interconnection style.

How does it work? Whereas the complete details require an extensive investigation of the SystemC library code, we can provide a short answer. When the code instantiating an **`sc_port`** executes, the **`operator()`** is overloaded to take a channel object by reference and saves a pointer to that reference internally for later access by the port. Thus, we recall a port is an interface pointer to a channel that implements the interface.

11.6 Accessing Ports From Within a Process

Connecting ports between modules and channels is of no great value unless a process somewhere in the design can initiate activity over the channels. This section will show how to access ports from within a process. The **`sc_port`** overloads the C++ **`operator->()`**, which allows a simple syntax (Fig. 11.16) to access the referenced interface.

```
portname->method(optional_args);
```

Fig. 11.16 Syntax of port access

Continuing the previous example, we now illustrate (Fig. 11.17) port access in action. In the following, `control` and `status` are the ports; whereas, K is a local channel instance. Notice use of the **`operator->`** when accessing ports.

```
void VIDEO_Mixer::Mixer_thread() {
  …
  switch (control->read()) {
    case MOVIE: K.write(0.0f); break;
    case MENU:  K.write(1.0f); break;
    case FADE:  K.write(0.5f); break;
    default:    status->write(ERROR); break;
  }
  …
}
```

Fig. 11.17 Example of port access

Ports feel and behave as if they were pointers. Indeed that is a good way to think of them even though this is not precisely correct. A mnemonic may help here. P is for port and P is for pointer. When accessing channels through ports, always use the pointer operator (i.e., ->).

11.7 Exercises

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

Exercise 11.1: Examine, compile, and run the `sedan` example. Which styles are simplest?

Exercise 11.2: Examine, compile, and run the `convertible` example. Notice the forward declarations of `Body` and `Engine`. How might this be an advantage when providing IP?

Exercise 11.3: Examine, compile, and run the `VIDEO_Mixer` examples. Change the port ordering, and insert a new port (with no functionality). What problems does this cause?

Exercise 11.4: In the VIDEO_Mixer port interconnect by name example, change the code from direct to indirect submodule instantiation.