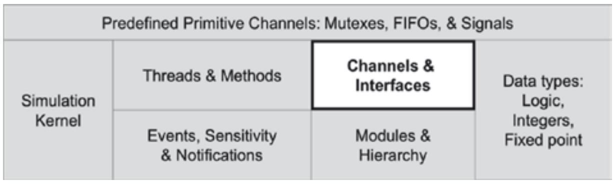
Chapter 13 Custom Channels and Data


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

Primitive & Hierarchical

We've already covered much of the syntax of SystemC. Now, we will focus on some of the more abstract concepts from which SystemC derives much of its power. This chapter illustrates how to create a variety of custom channels including: primitive channels, custom signals, custom hierarchical channels, and custom adaptors. Of these, custom signals and adaptors are probably the most commonly encountered.

13.1 A Review of SystemC Channels and Interfaces

In this section, we will review two of the four most important aspects of SystemC, channels and interfaces. The other two, modules and ports, have already been discussed in detail. Remember, SystemC channels implement communication between modules. SystemC interfaces provide an API and a means to allow independence of modules from the mechanisms of communication channels.

The basic structure of a channel is a class that inherits from one or more interfaces and a SystemC base class. The interface makes a channel usable with ports. Channels come in two flavors: primitive and hierarchical. Channels must inherit either from **`sc_prim_channel`** or **`sc_channel`**, which defines them as either primitive or hierarchical, respectively. This distinction in these latter two SystemC base classes is one of distinct capabilities and features. In other words, **`sc_prim_channel`** has capabilities not present in **`sc_channel`** and vice versa.

Primitive channels are intended to provide very simple and fast communications. They contain no hierarchy and no ports; primitive channels do not contain simulation processes. Primitive channels have the ability to use the evaluate-update paradigm as they inherit some specialized methods. These channels are discussed in the following section.

By contrast, hierarchical channels can have their own ports and processes, and they can contain hierarchy as the name suggests. In fact, hierarchical channels are really just modules that implement one or more interfaces. Hierarchical channels are intended to model complex communications buses such as PCI, HyperTransport™, AMBA™, or AXI™. Custom hierarchical channels are discussed later in this chapter.

Channels are important in SystemC because they enable several concepts:

- Appropriate channels enable safe communication among concurrent simulation processes.
- Channels in conjunction with ports clarify and delineate the relationships of communication (producer vs. consumer, master vs. slave, initiator vs. target).

Interfaces are important in SystemC because they enable the separation of communication from processing and allow independent refinement of the communication and functionality of a system.

13.2 The Interrupt, a Custom Primitive Channel

We discussed events in Chapter 6, Concurrency, and we saw how processes can use events to coordinate activities. We introduced hierarchy and ports in Chapter 10, Structure. This section answers the question of how we can provide a simple event or interrupt between processes located in different modules. Obviously, this interrupt is not the usual preemptive interrupt as used in software; instead, it is a handshake signal between modules or processes.

One approach might take a channel that has an event and simply use the side effect. For example, this approach could use sensitivity to **sc_signal**<**bool**> by use of the **value_changed_event()** method. However, using side effects is unsatisfying. Let us see how we might create a custom channel just for this purpose.

A proper channel must have one or more interfaces to implement. The ideal interface provides only the methods required for a particular purpose. For our channel, we'll create two interfaces: one interface for sending events, eslx_interrupt_gen_if, and another interface for receiving events, eslx_interrupt_evt_if. To allow and simplify use in static sensitivity lists, we'll specify a **default_event()**.

The interfaces are shown in the next figure (Fig. 13.1). Notice that interfaces are required to inherit from the **sc_interface** base class. Also, notice in eslx_interrupt_evt_if that **default_event()** has a specific calling signature. This signature is required for **default_event()** to be recognized by **sensitive**.

```
class eslx_interrupt_gen_if: public sc_interface {
public:
  virtual void notify() = 0;
  virtual void notify(sc_time t) = 0;
};
```

```
class eslx_interrupt_evt_if: public sc_interface {
public:
  virtual const sc_event& default_event() const = 0;
};
```

Fig. 13.1 Examples of custom channel interfaces

Next, we look at the implementation of the primitive channel shown in Fig. 13.2. The implementation has four features of interest.

First, the channel must inherit from **`sc_prim_channel`** and both of the interfaces we defined previously.

Second, the constructor for the channel has similar requirements to an **`sc_module`**; the constructor must construct the base class **`sc_prim_channel`**. This class has a single constructor that requires an instance name string.

Third, the channel must implement the methods compelled by the pure virtual functions in the interfaces from which it inherits. Thus, this channel must implement two versions of the **`notify()`** method and one **`default_event()`** method.

Fourth, we specify a private implementation of the copy constructor to prevent its use. Simply put, a channel should never be copied. This feature of the implementation provides a compile-time error if copying is attempted.

```
#include "eslx_interrupt_evt_if.h"
#include "eslx_interrupt_gen_if.h"

class eslx_interrupt
: public sc_prim_channel
, public eslx_interrupt_evt_if
, public eslx_interrupt_gen_if
{
public:
  // Constructors
  explicit eslx_interrupt()
  :sc_prim_channel(
              sc_gen_unique_name("eslx_interrupt"))
  {}//end constructor
  explicit eslx_interrupt(sc_module_name nm)
  :sc_prim_channel(nm)
  {} //end constructor
  // Methods
  void notify() { m_interrupt.notify(); }
  void notify(sc_time t) { m_interrupt.notify(t); }
  const sc_event& default_event() const
                { return m_interrupt; }
private:
  sc_event m_interrupt;
  // Copy constructor so compiler won't create one
  eslx_interrupt( const eslx_interrupt& rhs)
  {} //end copy constructor
};
```

Fig. 13.2 Example of custom interface implementation (AKA channel)

13.3 The Packet, a Custom Data Type for SystemC

Creating custom primitive channels is not very common; however, instantiating an **`sc_signal`**<*T*> channel or an **`sc_fifo`**<*T*> is very common. SystemC defines all the necessary features for both of these channels when used with built-in data types.

For custom data types, SystemC requires you to define several methods for your data type.

The reasons for the required methods are easy to understand. As an example, both channels support read and write methods, which involve copying the custom data type. For this reason, SystemC requires the definition of the assignment operator (i.e., `operator=()`). Also, `sc_signal`<*T*> supports the method `value_changed_event()`, which implies the use of comparison. In this case, SystemC requires the definition of the equality operator (i.e., `operator==()`).

Finally, there are two other methods required by SystemC, streaming output (i.e., `ostream& operator<<()`) and `sc_trace()`. Streaming output allows for a pleasant printout of your data structure during debug. The trace function allows all or parts of your data type to be used with the SystemC trace facility. This function enables viewing of trace data with a waveform viewer. We'll explain waveform data tracing in the next chapter.

Consider the following C/C++ custom data type (Fig. 13.3), which might be used for PCI-X transactions:

```
struct eslx_pcix_trans {
  int devnum;
  int addr;
  int attr1;
  int attr2;
  int cmnd;
  int data[8];
  bool done;
};
```

Fig. 13.3 Example of user-defined data type

This structure or record contains all the information necessary to fully communicate a PCI-X transaction; however, it is not usable with an `sc_signal`<*T*> channel or an `sc_fifo`<*T*>. Let's add the necessary methods (Fig. 13.4) to support this usage.

Note the `friend` declarations, which also need to be declared outside the class definition in the header file, are required because `ostream` and `sc_trace` are globally defined.

We provide example implementations of the latter two methods here (Fig. 13.5):

It should be noted that the `sc_trace()` method is optional; however, best practices suggest that you should always provide this method. Observe that this method is always expressed in terms of other traces that are already defined (e.g., the built-in ones).

In some cases, it may be difficult to determine an appropriate representation. For `sc_trace()` as an example, `char`* or `string` have no real logical equivalent.

```
//FILE: eslx_pcix_trans.h
class eslx_pcix_trans {// previously as struct
  int devnum; // May need get and set methods
  int addr;   // and could rename member data to
  int attr1;  // match m_ naming conventions.
  int attr2;
  int cmnd;
  int data[8];
  bool done;
public:
  // Required by sc_signal<> and sc_fifo<>
  eslx_pcix_trans& operator=(
    const eslx_pcix_trans& rhs
  ) {
    devnum = rhs.devnum;  addr   = rhs.addr;
    attr1  = rhs.attr1;   attr2  = rhs.attr2;
    cmnd   = rhs.cmnd;    done   = rhs.done;
    for (unsigned i=0;i!=8;i++) data[i]=rhs.data[i];
    return *this;
  }
  // Required by sc_signal<>
  bool operator==(const eslx_pcix_trans& rhs)
                  const {
    return (
        devnum ==rhs.devnum && addr   ==rhs.addr
     && attr1  ==rhs.attr1  && attr2  ==rhs.attr2
     && cmnd   ==rhs.cmnd   && done   ==rhs.done
     && data[0]==rhs.data[0]&& data[1]==rhs.data[1]
     && data[2]==rhs.data[2]&& data[3]==rhs.data[3]
     && data[4]==rhs.data[4]&& data[5]==rhs.data[5]
     && data[6]==rhs.data[6]&& data[7]==rhs.data[7]
     );
  }
  friend ostream& operator<<(ostream& file,
                    const eslx_pcix_trans& trans);
  friend void sc_trace(sc_trace_file*& tf,
              const eslx_pcix_trans& trans,
              string nm);
};
```

Fig. 13.4 Example of SystemC user data type

In these cases, you may either convert to an unsigned fixed-bit-width vector (e.g., **sc_bv**), or omit it completely. However, remember that converting these representations is for ease of debug operations. Providing a complete and clear **sc_trace** is usually of much more value than you might originally think. The same can be said of appropriate representation for **ostream**.

You may also want to implement **ifstream** or **ofstream** to support verification needs.

As you can see, the added support is really quite minimal, and it is only required for custom data types.

```
//FILE: eslx_pcix_trans.cpp
#include "eslx_pcix_trans.h"
ostream& operator<<(ostream& os,
                    const eslx_pcix_trans& trans)
{
  os << "{" << endl << "  "
     << "cmnd: " << trans.cmnd   << ", "
     << "attr1:" << trans.attr1  << ", "
     …
     << "done:"  << (trans.done?"true":"false")
     << endl << "}";
  return os;
} // end
// trace function, only required if actually used
void sc_trace(sc_trace_file*& tf,
              const eslx_pcix_trans& trans,
              string nm)
{
  sc_trace(tf, trans.devnum,  nm + ".devnum");
  sc_trace(tf, trans.addr,    nm + ".addr");
  …
  sc_trace(tf, trans.data[7], nm + ".data[7]");
  sc_trace(tf, trans.done,    nm + ".done");
} // end trace
```

Fig. 13.5 Example of SystemC user data type implementation

13.4 The Heartbeat, a Custom Hierarchical Channel

Hierarchical channels are interesting because they're really hybrid modules. Technically, a hierarchical channel must inherit from **`sc_channel`**; however, **`sc_channel`** is really just a **`typedef`** for **`sc_module`**. Hierarchical channels must also inherit from an interface to let them be connected to an external **`sc_port`**<*T*>.

Why would you define a hierarchical channel? One use of hierarchical channels is to model complex buses such as PCI, AMBA, HyperTransport, or AXI. Another common use of hierarchical channels, adaptors, and transactors will be discussed in the next section.

To keep things simple, we'll model a basic clock or heartbeat. This clock will differ from the standard hardware concept that typically uses a Boolean signal. Instead, our heartbeat channel will issue a simple event. This usage would correspond to the **`posedge_event()`** used by so many hardware designs.

Because it's basic, the heartbeat is more efficient simulation-wise than a Boolean signal. Here (Fig. 13.6) is the header for our simple interface:

```
class eslx_heartbeat_if: public sc_interface {
public:
  virtual const sc_event& default_event() const = 0;
  virtual const sc_event& posedge_event() const = 0;
};
```

Fig. 13.6 Example of hierarchical interface header

It's no different than a primitive channel interface. Notice that we use method names congruent with **`sc_signal`**<*T*>. This convention will simplify design refinement. The careful design of interfaces is key to reducing work that is done later.

Let's look at the corresponding channel header (Fig. 13.7), which inherits from **`sc_channel`** instead of **`sc_prim_channel`** and has a process, **`SC_METHOD`**.

```
include "eslx_heartbeat_if.h"
class eslx_heartbeat
 :public sc_channel
 ,public eslx_heartbeat_if {
public:
  SC_HAS_PROCESS(eslx_heartbeat);
  // Constructor (only one shown)
  explicit eslx_heartbeat(sc_module_name nm
                         ,sc_time _period)
  :sc_channel(nm)
  ,m_period(_period)
  {
    SC_METHOD(heartbeat_method);
    sensitive << m_heartbeat;
  }
  // User methods
  const sc_event& default_event() const
  { return m_heartbeat; }
  const sc_event& posedge_event() const
  { return m_heartbeat; }
  void heartbeat_method(); // Process
private:
  sc_event m_heartbeat; // *The* event
  sc_time  m_period;    // Time between events
  // Copy constructor so compiler won't create one
  eslx_heartbeat( const eslx_heartbeat& );
};
```

Fig. 13.7 Example of hierarchical channel header

Let's see how it's implemented (Fig. 13.8):

```
#include <systemc>
#include "eslx_heartbeat.h"

void eslx_heartbeat::heartbeat_method(void) {
  m_heartbeat.notify(m_period);
}
```

Fig. 13.8 Example of hierarchical channel interface header

In the next chapter, we'll see the built-in SystemC clock, which has more flexibility at the expense of performance.

13.5 The Adaptor, a Custom Primitive Channel

Also known in some circles as transactors, adaptors are a type of channel specialized to translate between modules with different interfaces. Adaptors are used when moving between different abstractions. For example, an adaptor is commonly used between a testbench that models communications at the transaction level (i.e., TLM), and an RTL implementation that models communications at the pin-accurate level. Transaction-level communications might have methods that transfer an entire packet of information (e.g., a PCI-X transaction). Pin-accurate level communications use Boolean signals with handshakes, clocks, and detailed timing.

To make it easy to understand, we're going to investigate two adaptors. In this section, we'll see a simple primitive channel that uses the evaluate-update mechanism. In the following section, we'll investigate a hierarchical channel. For many of the simpler communications, an adaptor needs nothing more than some member functions and a handshake to exchange data. This setup often meets the requirements of a primitive channel. Many of the simpler adaptors could be of either type, since they don't require an evaluate-update mechanism.

We will now discuss an example design. The example design includes a stimulus (`stim`) and a response (`resp`) that is connected via an `eslx_interrupt` channel described in an earlier section. We now would like to replace `resp` with a refined RTL version, `resp_rtl`, which requires a **`sc_signal<bool>`** channel interface. The before and after example design is graphically shown in Fig. 13.9.

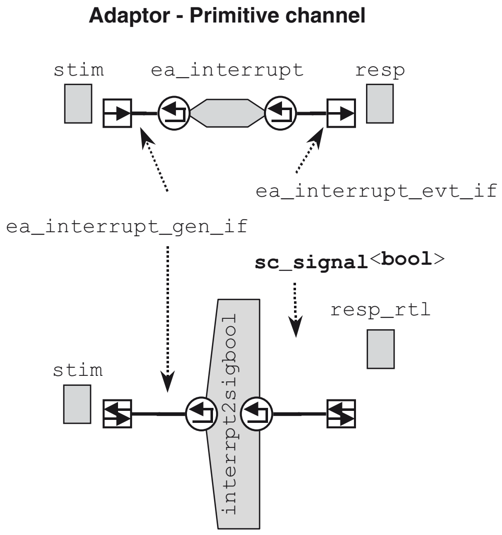


Fig. 13.9 Before and after adaptation

Here (Fig. 13.10) is the adaptor's header:

```
#include "eslx_interrupt_gen_if.h"
class interrupt2sigbool
: public sc_prim_channel
, public eslx_interrupt_gen_if
, public sc_signal_in_if<bool>
{
public:
  // Constructors
  explicit interrupt2sigbool()
  : sc_prim_channel(
      sc_gen_unique_name("interrupt2sigbool")) {}
  explicit interrupt2sigbool(sc_module_name nm)
  : sc_prim_channel(nm) {}
  // Methods for eslx_interrupt_gen_if
  void notify() {
    m_delay = SC_ZERO_TIME; request_update(); }
  void notify(sc_time t) {
    m_delay = t; request_update(); }
  // Methods for sc_signal_in_if<bool>
  const sc_event& value_changed_event() const
  { return m_interrupt; }
  const sc_event& posedge_event() const
  { return value_changed_event(); }
  const sc_event& negedge_event() const
  { return value_changed_event(); }
  const sc_event& default_event() const
  { return value_changed_event(); }
  // true if last delta cycle was active
  const bool& read() const {
    m_val = event(); return m_val;
  }
  // Did value change in the previous delta cycle?
  bool event() const {
    return (sc_delta_count() == m_delta+1);
  }
```

```
  bool posedge() const { return event(); }
  bool negedge() const { return event(); }
protected:
  // every update is a change
  void update() {
    m_interrupt.notify(m_delay);
    m_delta = sc_delta_count();
  }
private:
  sc_event m_interrupt;
  mutable bool m_val;
  sc_time  m_delay;
  uint64   m_delta;    // delta of last event
  // Copy constructor so compiler won't create one
  interrupt2sigbool( const interrupt2sigbool& );
};
```

Fig. 13.10 Example of primitive adaptor channel header

The first thing to notice is all the methods. Most of these are forced upon us because we are inheriting from the **sc_signal_in_if<bool>** class. Fortunately, most of them may be expressed in terms of others for this particular adaptor. Another way to handle excess methods is to provide stubbed **SC_FATAL**[1] messages with the assumption that nobody will use them.

The second feature of interest is the manner in which evaluate-update is handled. In the **notify()** methods, we update the delay and make a **request_update()** call to the scheduling kernel. When the delta-cycle occurs, the kernel will call our **update()** function that issues the appropriately delayed notification.

For the most part, this adaptor was simple. The hard part was obtaining a list of all the routines that needed to be implemented as a result of the interface. Listing the routines is accomplished easily enough by simply examining the interface definition in the Open SystemC Initiative library source code.

A third feature to note is the use of **sc_delta_count()**. It is used to determine that the interrupt event has occurred in the previous delta-cycle. The value returned by **sc_delta_count()** is incremented by one for each delta-cycle while the simulator is running.

Finally, for those not completely up on their C++, a comment on the **mutable bool**. The keyword **mutable** means changeable even if const. The **read()** method is defined in **sc_signal_in_if<bool>** interface, so we have to implement it. The **read()** method is defined as **const**, and it is required to return a **const** reference (&). We are using the member function **event()** to obtain a value, which is not a reference. So, we create a member data `m_val` to store the return value temporarily. Because the value is mutable, we are able to change it (even though the method is **const**) and return it as a **const** reference.

13.6 The Transactor, a Custom Hierarchical Channel

When a more complex communications interface is encountered; such as one that requires processes, hierarchy, or ports; then a hierarchical channel solution is required. The following processor interface problem demonstrates this type of channel.

Suppose we have a testbench connected to an abstract model of a memory, and wish to replace the abstract memory with an RTL model. On one side, we have a testbench that needs to use simple transaction calls to verify the functionality of the memory. On the other side, we have a peripheral, an 8K x 16 memory. To not change the testbench, we insert an adaptor between the testbench and the RTL memory. This adaptor allows the testbench to convert transactions into pin-level stimulus.

[1] SC_FATAL is discussed in Chapter 14, Additional Topics.

Graphically, here (Fig. 13.11) are the elements of the design:

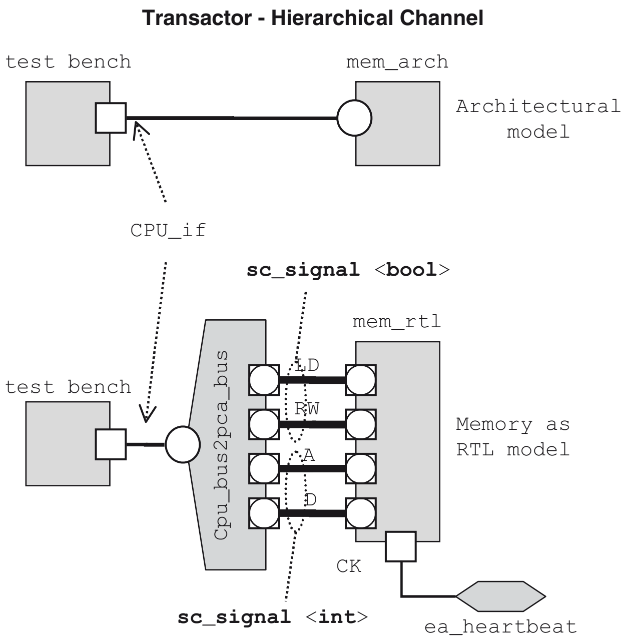


Fig. 13.11 Testbench adaptation using hierarchical channels

This figure actually has two hierarchical channels. The architectural model of the memory is a module implementing an interface, in this case the `CPU_if`. Our memory was designed to hang directly off the CPU.

Let's take a look at the CPU interface (Fig. 13.12):

```
class CPU_if: public sc_interface {
public:
  virtual void write(unsigned long addr
                    ,long data) = 0;
  virtual long read(unsigned long  addr) = 0;
};
```

Fig. 13.12 Example of simple CPU interface

The corresponding memory implementation is a straightforward channel (Fig. 13.13):

```
//FILE: mem_arch.h
#include "CPU_if.h"
class mem
: public sc_channel
, public CPU_if
{
public:
  // Constructors & Destructor
  explicit mem(sc_module_name nm
              ,unsigned long ba
              ,unsigned sz)
  :sc_channel(nm)
  ,m_base(ba)
  ,m_size(sz)
  { m_mem = new long[m_size]; }
  ~mem() { delete [] m_mem; }
  // Interface Implementations
  virtual void write(unsigned long addr
                    ,long data) {
    if (m_base <= addr && addr < m_base+m_size) {
      m_mem[addr-m_base] = data;
    }
  }//end write
  virtual long read(unsigned long  addr) {
    if (m_base <= addr && addr < m_base+m_size) {
      return m_mem[addr-m_base];
    } else {
      cout << "ERROR:"<<name()<<"@"<<sc_time_stamp()
           << ": Illegal address: " << addr << endl;
      sc_stop(); return 0;
    }
  }//end read
private:
  unsigned long m_base;
  unsigned      m_size;
  long*          m_mem[];
  mem_arch(const mem_arch&); // Disable
};
```

Fig. 13.13 Example of hierarchical channel memory implementation

Now, suppose we have the following timing diagram (Fig. 13.14) for the pin-cycle accurate interface:

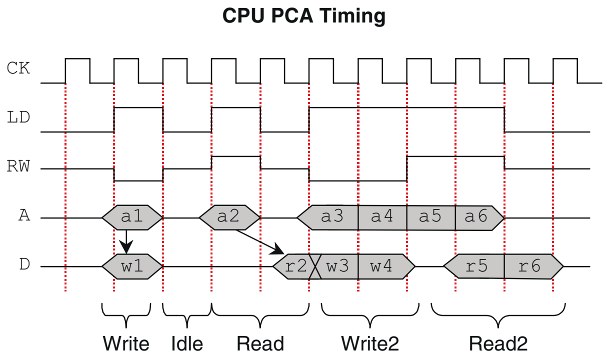


Fig. 13.14 CPU pin-cycle accurate timing

Notice that write transactions take place in a single clock cycle; whereas, the read transaction has a one-cycle delay for the first read in a burst. Also, this interface assumes a bidirectional data bus. Address and read/write have a non-asserted state. For this design, we'll allow this setup.

Here is the transactor's header (Fig. 13.15):

```
#include "CPU_if.h"
#include "eslx_heartbeat_if.h"
class cpu2pca
  :public sc_module
  ,public CPU_if
{
public:
  // Ports
  sc_port<eslx_heartbeat_if> ck; // clock
  sc_out<bool>               ld; // load/exec cmd
  sc_out<bool>               rw; // read high
                                 // write low
  sc_out<unsigned long>      a;  // address
  sc_inout_rv<32>            d;  // data
  // Constructor
  SC_CTOR(cpu2pca):FLOAT("ZZZZZZZZ") {}
  // Interface implementations
  void write(unsigned long addr
            ,long data);
  long read(unsigned long addr);
  // Useful constants
  const sc_lv<32> FLOAT;
private:
  cpu2pca(const cpu2pca&); // Disable
};
```

Fig. 13.15 Example of hierarchical transactor channel header

Clearly, with the preceding example, the basics of a module are present. Inheriting from `CPU_if` simply adds a few methods to be implemented, namely **`read()`** and **`write()`**.

An interesting point to ponder with channels (especially adaptors) is the issue of member function collisions. What if two or more interfaces that need to be implemented have identically named member functions with identical argument types?

There are two solutions. One solution is to modify the interface method in a renamed interface. This solution is ugly. Another solution is to isolate each interface to an **`sc_export`**<*T*>. This improved solution lets you use the implementation in a locally instantiated channel to complete your implementation.

Here (Fig. 13.16) is the implementation code for the transactor:

```
#include "cpu2pca.h"
enum operation {WRITE=false, READ=true};
void cpu2pca::write(unsigned long addr
                   ,long data) {
  wait(ck->posedge_event());
  ld->write(true);
  rw->write(WRITE);
  a->write(addr);
  d->write(data);
  wait(ck->posedge_event());
  ld->write(false);
}
long cpu2pca::read(unsigned long addr) {
  wait(ck->posedge_event());
  ld->write(true);
  rw->write(READ);
  a->write(addr);
  d->write(FLOAT);
  wait(ck->posedge_event());
  ld->write(false);
  return d->read().to_long();
}
```

Fig. 13.16 Example of hierarchical transactor channel implementation

The code for an adaptor can be very straightforward. For more complex applications; such as a PCI, AMBA, or AXI; the design of an adaptor may be more complex.

Because adaptors allow high-level abstractions to interface with lower-level implementations, they are very common in SystemC designs. Sometimes these hybrids are used as part of a design refinement process. At other times, they merely aid the development of verification environments. There are no fixed rules defining abstraction levels or how to use them.

13.7 Exercises

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

Exercise 13.1: Examine, compile, and run the `interrupt` example. Write a specialized port for this channel to support the method **`pos()`**.

Exercise 13.2: Examine, compile, and run the `pcix` example. Could this process of converting a **`struct`** to work with an **`sc_signal`** be automated? How?

Exercise 13.3: Examine, compile, and run the `heartbeat` example. Extend this channel to include a programmable time offset.

Exercise 13.4: Examine, compile, and run the `adapt` example. Notice the commented-out code from the adaptation of `resp` to `resp_rtl`.

Exercise 13.5: Examine, compile, and run the `hier_chan` example. Examine the efficiency of the calls. Extend the design to allow back-to-back reads and writes while using cycles efficiently.