SID Simulator Component Developer’s Guide

SID Simulator Component Developer's Guide

Table of Contents

Chapter 1. Introduction

The SID Component Developer’s Kit (CDK) from Red Hat, Inc. provides you with the tools you need to create a simulated environment and to create simulated components. With these tools you can write and test software for an embedded system before the physical hardware is ready. The CDK provides:

- An API and tutorial to enable you to create your own simulated components
- A library of simulated hardware components (plug-ins) for commonly-used parts. For more information, refer to the *Simulator User’s Guide.*
- The ability for you to customize your environment.

This guide provides documentation for the Component Developer’s Kit. It guides you through the steps you need to take to develop your own simulation components.

About the SID Simulation Environment

The SID development environment enables levels of debugging and testing that are not readily available on real hardware. The SID simulation environment offers:

Greater Availability.

Early production hardware is often available in very limited quantities. SID allows a greater number of embedded software developers to access the tools they need to write and test software.

More Control.

The simulator can be easily stopped, reconfigured, and rerun. The system state can be saved and restored. This is not easy on real hardware.

Increased Debugging Ability.

The simulator provides a debugging environment that is not possible to create on real hardware. The simulator can show internal state of devices revealing information that cannot be captured with a logic probe. The simulator also provides more debugging control (with the use of triggerpoints, for example) than is possible on real hardware.

Increased Stability.

Early prototype hardware may have bugs or be unstable. The simulator enables comparative testing on early prototype hardware so that hardware bugs are easier to identify.

The SID Simulation environment provides continuity between developing and debugging on real hardware and on simulated hardware.

The Component Developer’s Kit (CDK)

The Component Developer’s Kit CDK is a toolkit for building new simulation components.

API Reference

All SID components adhere to a single API. To learn all about the SID API, refer to Chapter 2.

CDK Tutorial

To assist you in developing your own components, we provide a step-by-step tutorial that describes how we created a SID component of a simple timer chip.

For a detailed run-through of how to create a simulation component, refer to *CDK Tutorial*.

Document Conventions

The documentation uses the following general conventions:

Italic Font

Indicates a section name or a manual name.

Bold Font

Is used for the menu, window, and buttons names you see on your screen.

Plain Typewriter Font

Denotes code fragments, command lines, contents of files, and command names; also indicates directory, file, and project names where they appear in body text.

Bold, Italic Typewriter Font

Represents a variable for which you should substitute an actual value.

Chapter 2. API Reference

API Overview

Components

A SID component is a C++ object of type `sid::component` which calls, and is called by, other SID components. A complete simulation is a set of components connected together. Simulations are single-threaded, and there is no underlying "supervisory" simulation code. As the simulation runs, control repeatedly loops within a "root" component, calling into its connected components, which perform any local simulation tasks and in turn call their connected components before returning control to their caller. This cycle continues until the simulation is terminated.

Attributes

SID Components come equipped with a set of string-valued attributes, which can be queried and set during configuration or simulation, to inspect or modify the behavior of the component. Attributes generally provide information and control which is of interest to the simulation user, rather than to other components in the simulation. Conventionally, attributes are grouped into named "categories" by role, though attributes may have no category, and may in addition be accessed without regard to their category. Categories merely assist in treating a group of attributes similarly, for example in a simulator's graphical user interface.

Pins and Buses

SID models a component's communication connections as a set of buses and pins. These terms are intended to reflect the design of the hardware being simulated, and SID's API presents these terms as the abstract C++ classes `sid::bus` and `sid::pin`. A SID component may have many pins and many buses, each of which represents the *receiving* end of a connection to another component.

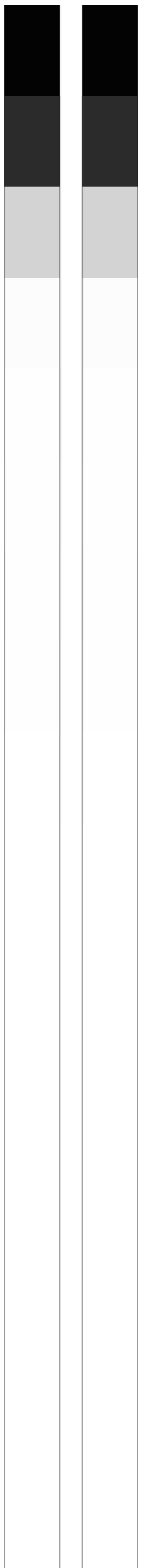
A pin represents a non-refutable, one-way connection through which an integer value may be "driven". Pins are best thought of as "triggers", which may carry an informative value but which do not have any sort of request/response mechanism built into them. A bus, on the other hand, represents a read/write interface through which integer values *and addresses* are transmitted. Every bus request returns a `sid::bus::status` object, which describes both the success or failure of the request and the latency of the bus transaction, as modeled by the recipient. Buses are best thought of as small windows into the recipient's memory or register set, through which the sender may attempt to read or write values.

Pins and buses do not in general exist beyond the scope of the component they belong do; a component is usually responsible for managing the lifecycle of its own pins and buses. Moreover components do not often set up their own connections. Instead, each component presents a set of "factory methods", which can be called with string names of buses or pins. The component must then choose to either construct a pin or bus "on demand", serve a pin and bus from a fixed set compiled into the component, or respond with a NULL pointer, indicating an unknown pin or bus was requested. A component may also be asked to place a pointer to a given pin or bus in a named slot within itself, which it will use as the *sending* end of a connection during simulation.

Slots for buses are referred to as "accessors", whereas slots for pins are simply "output pins".

Example 2-1. Bus Connection

It is very important to understand the relationship between sending and receiving ends of bus and pin connections, so we present an example here. Suppose we load a CPU component C and a memory component M. Suppose that C has an accessor named "system-mem" which it intends to use for reading and writing to some sort of memory device, and suppose that M has a bus named "data-bus" through which it expects to receive read/write requests. The scenario is depicted blow.



Recall that the accessor is the *sending* end of the connection we wish to make, and that the bus is the *receiving* end. Some people find the terms "master" and "slave" clearer than "sender" and "receiver", since information is permitted to flow both ways on a bus. In the master/slave terminology, the bus master makes bus *requests*; the bus slave *services* them. Whether the bus requests are reads or writes is irrelevant to this relationship. The following code will establish the connection:

```
sid::bus *memory_data_bus = M.find_bus ("data-bus");
  C.connect_accessor ("system-mem", memory_data_bus);
```

Most connections are not established by hand this way, but are done by a "configuration root" component, which reads a simple configuration command language, loads and instantiates components, hooks up buses to accessors, output pins to input pins, and sets attribute values. Each of these configuration commands is closely mirrored by a small sequence of SID API calls, such as in the above example.

Note that in this example only one component is transmitting signals to the memory's bus. This is not intrinsic to the connection mechanism; connections are usually just assignments to pointer variables within the sending component. There could just as easily be several components all sending to the same bus at various times during simulation. Additionally, while an accessor can only ever be connected to one bus, an output pin can be connected to *multiple* input pins; driving the output simply drives *all* the connected pins. Obviously this arrangement does not make sense for buses, in which each bus request results its own unique status code. To connect one accessor to multiple buses, one needs a bus mapper to arbitrate the connection. See the component reference manual for details on bus mappers.

Relationships

In addition to pins, buses and attributes, SID components may have named slots for holding pointers to other components. Such an association, when made in terms of the `sid::component` interface, is called a "relationship" of the pointer-holder. A component may hold many pointers to different components in a single relationship, and may have components placed in its relationship lists by the configuration process.

Relationships give components the ability to call related components' `sid::component` API, rather than simply communicate with them through `sid::bus` and `sid::pin` connections. If such a capability is not strictly required, the more conservative bus and pin communication APIs are generally preferred for the sake of minimizing potential bugs.

Component Libraries

Components are conventionally packaged into separate shared object (.so) or dynamic link library (.DLL) files, and loaded on demand into the SID process by the configuration component. Each such file may contain several component classes, but by convention such components are functionally similar. SID specifies an API for querying such files and instantiating their components. Any file which implements a component must support this API for SID to make use of it.

the `sid::component` interface

Any component must implement the following interface, though most of the methods can be implemented trivially, if not relevant. Additionally, there are several mix-in classes located in extra SID utility headers. These mix-ins are not part of the official component API, but cover a wide variety of common component implementation strategies and greatly simplify the task of writing a component; programmers are encouraged to read these headers for more details.

Attributes

```
vector<string> attribute_names ();

vector<string> attribute_names (const string& category);

string attribute_value (const string& name);

status set_attribute_value (const string& name, const string& value);
```

A component's attributes are listed with `attribute_names`. Once an attribute name is known it can be queried with `attribute_value` or set with `set_attribute_value`.

Pins

```
vector<string> pin_names ();

pin *find_pin (const string& name);

status connect_pin (const string& name, pin *pin);

vector<pin *> connected_pins (const string& name);

status disconnect_pin (const string& name, pin *pin);
```

A component's input and output pins are listed, by name, with `pin_names`. A pointer to an input pin can be obtained, by name, with `find_pin`. This pointer points to a `sid::pin` object, and is thus the *receiving* end of a pin connection. To complete the connection, this pin-pointer must be associated with an output pin name in the component acting as the *sending* end of the connection, using the `connect_pin` method. The input pins connected to a given output pins can be fetched with `connected_pins`, and individually disconnected from the output pin with `disconnect_pin`.

Buses

```
vector<string> bus_names ();

vector<string> accessor_names ();

bus *find_bus (const string& name);

status connect_accessor (const string& name, bus *bus);

status disconnect_accessor (const string& name, bus *bus);
```

```
vector<bus *> connected_bus (const string& name);
```

A component's buses are listed, by name, with `bus_names`; its bus accessors are listed with `accessor_names`. A pointer to a bus can be obtained, by name, with `find_bus`. This pointer points to a `sid::bus` object, and is thus the *slave* end of a bus connection. To complete the connection, this bus-pointer must be associated with an accessor name in the component acting as the *bus master*, using the `connect_accessor` method. The bus connected to an accessor can be fetched with `connected_bus`, and disconnected from the accessor with `disconnect_bus`.

Relationships

```
vector<string> relationship_names ();

status relate (const string& name, component *comp);

status unrelate (const string& name, component *comp);

vector<component *> related_components (const string& name);
```

A component's relationship names are listed with `relationship_names`. Once an relationship name is known components can be added to it with `relate` or removed from it with `unrelate`. The set of components currently in a given relationship name can be fetched with `related_components`

the `sid::bus` interface

Reading

```
status read (host_int_4 addr, little_int_1& data);

status read (host_int_4 addr, little_int_2& data);

status read (host_int_4 addr, little_int_4& data);

status read (host_int_4 addr, little_int_8& data);

status read (host_int_4 addr, big_int_1& data);

status read (host_int_4 addr, big_int_2& data);

status read (host_int_4 addr, big_int_4& data);

status read (host_int_4 addr, big_int_8& data);
```

To read from a bus, the bus master calls the overloaded function `read` with an address to read from, and a reference to a data word. Addresses are conventionally interpreted as *byte* addresses. If the read is successful, the data word will contain the word read; otherwise it will be unchanged.

Writing

```
status write (host_int_4 addr, little_int_1 data);

status write (host_int_4 addr, little_int_2 data);

status write (host_int_4 addr, little_int_4 data);

status write (host_int_4 addr, little_int_8 data);

status write (host_int_4 addr, big_int_1 data);

status write (host_int_4 addr, big_int_2 data);

status write (host_int_4 addr, big_int_4 data);

status write (host_int_4 addr, big_int_8 data);
```

To write to a bus, the bus master calls the overloaded function `write` with an address to write to, and a data word. Addresses are conventionally interpreted as *byte* addresses. The result of a write indicates whether the write was considered, by the bus slave, to be successful. Note however that the slave is not in any way required to reflect the contents of a successful write in terms of any future reads from the same address.

Status codes

All bus transactions are initiated by a bus master, and respond with a status code object. Such an object has the following fields:

`code`

a value from the `sid::bus::status_t` enumeration: either `ok`, `misaligned`, `unmapped`, or `unpermitted`.

`latency`

a scalar quantity, in unspecified units, describing the latency of the bus transaction.

The `sid::bus::status` class has simple zero-, one- and two-argument constructors, indicating general zero-latency success, success status alone (with zero latency) or status and latency of the bus transaction. For example, one can return from a bus transaction with any of the following statements:

```
return status ();           // status = ok, latency = 0

return status (unmapped);   // status = unmapped, latency = 0

return status (ok, 5);      // status = ok, latency = 5
```

the `sid::pin` interface

```
void driven(host_int_4 value);
```

The only operation one can perform on a pin is to drive a value to it; this may or may not have any effect, but it cannot fail.

the `sid::component_library` interface

Every component library must contain a `sid::component_library` structure in order for SID to load and instantiate its components. For example, assuming the definition of suitable functions `exampleListTypes`, `exampleCreate`, and `exampleDelete`, the following declaration would provide an interface for SID to instantiate components:

```
extern const component_library example_component_library;

const component_library example_component_library DLLEXPORT =
{
  COMPONENT_LIBRARY_MAGIC,
 & exampleListTypes,
 & exampleCreate,
 & exampleDelete
};
```

The precise type and meaning of these fields follows.

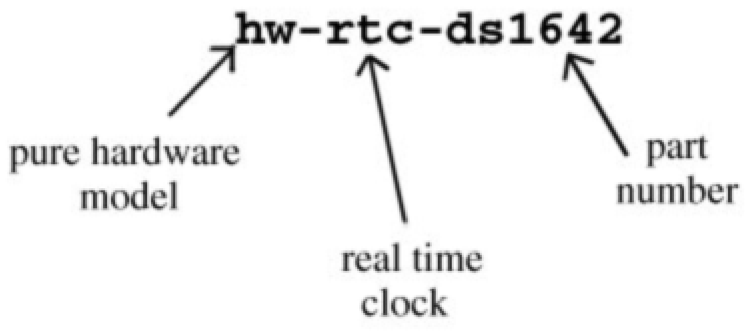
```
host_int_4 magic;

vector<string> (*list_component_types) ();

component* (*create_component) (const string& typeName);

void (*delete_component) (component* c);
```

The `sid::component_library` structure contains a magic number, for API version compatibility assurance, as well as pointers to 3 functions. The magic number should be initialized to the constant `COMPONENT_LIBRARY_MAGIC`, provided in SID's headers. The function pointed to by `list_component_types` must return a vector containing the string names of each type of component found in this library. These names are the same names used in the configuration command language, for example:



The function pointed to by `create_component` takes such a component name, and should construct a component of the given type and returns a pointer to it. If construction fails or an unknown type of component was requested, the function may return the value 0 to indicate failure. The function pointed to by `delete_component` should delete the provided component.

Chapter 3. CDK Tutorial

This chapter details the API that all SID components must follow.

CDK Tutorial

This tutorial describes how the various parts of the Component Developer's Kit architecture can be used to create a complete and functional simulation model.

To provide some context, the tutorial guides you through the design and implementation of a sample component–a timer chip. This chip was chosen for several reasons:

- It is both simple to understand and simple to implement.
- It has both memory-mapped registers and pins, so it can be used to demonstrate the main functional interfaces in the simulator.
- It has a time-dependent aspect, which opens the door to more interesting design and implementation trade-offs.

The tutorial presents a series of steps leading from the initial examination of the chip specifications to the external configuration script that will be used to exercise the component.

Step 1: Define the Functionality Required

The first step in developing a component is to define its functional requirements. All chips have two aspects that must be represented in their software model

- The physical interface of buses, pins, and registers
- The behavioral interface to application software and the external world

This section describes the timer chip in enough detail to resolve the design and functional considerations.

Example

The particular timer used in this tutorial is a dual 16-bit down counter type, in which each timer can be pre-scaled and set in either free-running mode or periodic timer mode. For the purposes of this example, the specifications are taken from the on-board timers of the ARM CPU. However, the principles can be generalized to most timers, and in fact, the interface aspects apply to a broad range of peripherals.

The following details the timer registers:

Table 3-1. Registers

Address	Name	Read Location	Write Location
base$_i$ + 0x0	LOAD_REG	16-bit timer load value	
base$_i$ + 0x4	VAL_REG	16-bit timer load value	reserved
base$_i$ + 0x8	CTL_REG	8-bit timer control	
base$_i$ + 0xC	CLR_REG	reserved	interrupt clear

Table 3-2. Control Registers

31–8	7	6	5–4	3	2	1–0
0	Enable	Mode	0	Prescale		0
0	0 = Disabled 1 = Disabled	0 = Free-running 1 = Periodic	0	Bits 3–2	Divisor	0
0	0 = Disabled 1 = Disabled	0 = Free-running 1 = Periodic	0	00	1	0
0	0 = Disabled 1 = Disabled	0 = Free-running 1 = Periodic	0	01	16	0
0	0 = Disabled 1 = Disabled	0 = Free-running 1 = Periodic	0	10	256	0
0	0 = Disabled 1 = Disabled	0 = Free-running 1 = Periodic	0	11	Undefined	0

Each register is 32-bits wide. Since there are two timers, there are two base addresses: base 0 is selectable; base 1 = base 0 + 0x20.

The timers operate by counting down from the 16-bit value in LOAD_REG. The count rate can be varied by the pre-scale settings of the control register. The pre-scale is a divisor applied to the input clock. Therefore, with a pre-scale setting of 1, the timer counts down at the input clock rate; with a pre-scale value of 16, the timer counts down at 1/16 of the input clock rate. The current count can be read at any time from VAL_REG. Once the counter reaches zero, the timer takes one of two actions depending on the Mode setting, which is bit 6 of the control register. If the timer is in free-running mode, it will reload the current count value from LOAD_REG and start counting down again; if the timer is in periodic mode, it will generate an interrupt, and then reload its current count and continue.

The above set of behaviors forms the functional requirements of the simulated timer model. Specifically, the registers must be present at the prescribed addresses, and must appear to application software as if they were the real registers. Similarly, the interrupt and clock behaviors must be modeled accurately with respect to interrupt timing and pre-scale behavior.

Step 2: Design the Model

Once the functionality of the hardware is understood, it must be mapped onto the simulator's API and level of abstraction. There are some design decisions that can be made to raise or lower the level of abstraction. Many of these decisions trade performance of the simulated component against its fidelity to the actual hardware.

The timers in this tutorial are simple and therefore do not require many design decisions, but developing components such as CPUs or Ethernet controllers involves evaluating many trade-offs. As an example, a CPU with a hardware level 1 cache may be modeled without the cache, since cache simulation is time and space consuming and may not affect the correctness of the simulated application. This is a common motivation for reducing the accuracy of a component, since often the simulator is only required to behave with functional correctness from the target software's point of view.

Example

The simulator architecture possesses some basic properties that impact component implementation:

- Components communicate with other components only through pin and bus connections.
- Components do not have an internal concept of time.
- Components must not block (that is, suspend) the main thread of control.

In the case of the timer, the simulator framework dictates that the bus interface must be used to allow access to the registers; interrupt generation should use the pin interface; and the events that change the timer's counters must be externally generated.

There are several approaches to use when setting up the external timer generation of the timer model. The simulator includes an event scheduler component that maintains the concept of simulated time and supports a queue of time-ordered events. The timer's clocking is thus achieved by enqueueing a request to be notified at some specific future time.

- The simplest approach is to specify the future time as one system clock tick past the current time. This allows the component to examine its pre-scale state and decrement the internal counters appropriately. The problem with this approach is that it is inefficient–there is significant communication with the scheduler on every clock tick.
- A second approach is to set the notify time to the value of the pre-scaler, which, at best, can reduce the communication with the scheduler by up to a factor of 256, in this example.
- A third approach achieves minimal communication by calculating the pre-scaled total time, and requesting notification only when the internal counter value reaches zero.

One consideration in timing notification is that the timer must make its internal count value available at any intermediate time, since this is a functional requirement of supporting reads of VAL_REG. In the first two approaches, this requirement can be supported directly. In the third approach, intermediate counts do not have to be kept because the notification from the scheduler arrives only when the count is zero. Intermediate times can be supported on-demand, by querying the scheduler for the current time when a request for the intermediate count is received. If the start time of the current period is also saved, then the intermediate count can be calculated by dividing the elapsed time, which is the current time less the start time, by the pre-scale value.

This discussion is presented to illustrate the reasoning process that the component designer might use. For purposes of illustration, the remainder of the example will assume that the scheduler is set to notify the timer component at the pre-scale rate, the second option given above.

Step 3: Define the Component

The first implementation step is to define the component. All components derive from the abstract sid::component class that provides supervisory access and the methods needed to connect and initialize the model.

Example

The timer and its internal state can be defined as follows. For clarity, nested classes used for pins and buses will be omitted at this point:

```
class Timer: public virtual sid::component
{
public:
  Timer()
    :scheduler_pin(0), clock_pin(this), bus(this), enabled(false) { }

  // Provide implementations for abstract methods in sid::component.
  // See include/sidcomp.h.

  vector<string> pin_names();
  sid::pin* find_pin(const string& name);
  sid::component::status connect_pin(const string& name,
    sid::pin* pin);
  sid::component::status disconnect_pin(const string& name,
    sid::pin* pin);
  vector<sid::pin*> connected_pins(const string& name);

  vector<string> accessor_names();
  sid::component::status connect_accessor(const string& name,
    sid::bus* bus);
  sid::component::status disconnect_accessor(const string& name,
    sid::bus* bus);

  vector<string> bus_names();
  sid::bus* find_bus(const string& name);
  sid::bus* connected_bus(const string& name);

  vector<string> attribute_names();
  vector<string> attribute_names(const string& category);
  string attribute_value(const string& name);
  sid::component::status set_attribute_value(const string& name,
    const string& value);

  vector<string> relationship_names();
  sid::component::status relate(const string& rel,
    sid::component* c);
  sid::component::status unrelate(const string& rel,
    sid::component* c);
  vector<sid::component*> related_components(const string& rel);

private:
  // Data members that represent the timer’s internal state.

  bool enabled;
  sid::host_int_2 load_value, prescale, counter;
  enum timer_mode { PERIODIC, FREERUNNING } mode;
};
```

The Timer class contains all of the internal state needed to model a single timer. As a result, two instances of the class must be created, and mapped at appropriate addresses, to model the hardware. The internal state does not try to mimic the hardware layout; instead, it is organized to simplify the implementation for the component writer. In general, registers that store states as a bit field should be internally represented with separate member variables and they should only be assembled into a packed bit field when necessary.

The Timer class also defines all of the abstract methods from the sid::component class that form the SID component API. Each of these methods must be implemented to fulfill the API. As you will see, these methods quite often can be implemented at a minimum if the component does not require all of the API's facilities.

Step 4: Define the Pins

The SID API only addresses the concept of input pins. An input pin is one which is made visible to other components, through the `pin_names()` and `find_pin()` methods. There is no concept of an output pin, except that it is a component's internal collection of other components' pin objects.

Example

The timer component will have two output pins: the interrupt pin and a control pin to be used in conjunction with the scheduler component. Since we do not wish to limit the number of other components that may be interrupted by the timer, it is necessary to maintain a "net list" of pin connections that can be iterated over each time a value is driven across the interrupt pin:

```
private:
  // A netlist, which tracks pins connected to the interrupt pin.
  typedef set<sid::pin*> netlist_t;
  netlist_t intpin_netlist;
```

The following methods do just that:

```
void
Timer::drive_interrupt(sid::host_int_4 value)
{
  // Iterate the netlist, driving the value to all pins connected to
  // the interrupt pin.

  for (netlist_t::const_iterator it = intpin_netlist.begin();
       it != intpin_netlist.end();
       it++)
    {
      (*it)->driven(value);
    }
}


// Return a list of pins that are connected to a named pin.
// We recognize "interrupt" and "divided-clock-control".

vector<sid::pin*>
Timer::connected_pins(const string& name)
{
  vector<sid::pin*> pins;
  netlist_t::const_iterator it;
  if (name == "interrupt")
    {
      for (it = intpin_netlist.begin(); it != intpin_netlist.end();
           it++)
        {
          pins.push_back(*it);
        }
      return pins;
    }
  else if (name == "divided-clock-control")
    {
      pins.push_back(scheduler_pin);
      return pins;
    }
  return vector<sid::pin*>();
}


// Connect a pin to a named pin.
// We recognize "interrupt" and "divided-clock-control".

// We allow multiple pins to be connected to the interrupt pin (with
// infinite fan-out!), so these are kept in a netlist.  For
// efficiency, the STL container chosen for the netlist ensures that
// no duplicate pin handles are stored.

sid::component::status
Timer::connect_pin(const string& name, sid::pin* pin)
{
  if (name == "interrupt")
    {
      // Add this pin to the netlist.
      intpin_netlist.insert(intpin_netlist.end(), pin);
      return sid::component::ok;
    }
  else if (name == "divided-clock-control")
    {
      // Reassign the scheduler pin.
      scheduler_pin = pin;
      return sid::component::ok;
    }
  return sid::component::not_found;
}

// Disconnect a pin from a named pin.

sid::component::status
Timer::disconnect_pin(const string& name, sid::pin* pin)
{
  if (name == "interrupt")
    {
      // Remove this pin from the netlist.
      if (intpin_netlist.erase(pin) > 0)
        return sid::component::ok;
    }
  else if (name == "divided-clock-control"& & scheduler_pin == pin)
    {
      // Elsewhere, we make sure to not use this pin if it’s null.
      scheduler_pin = 0;
      return sid::component::ok;
    }
  return sid::component::not_found;
```

```
}
```

The component will also have one input pin which is exported to other components: divided-clock-event. This pin, when connected to the scheduler component, can be used to deliver event notifications to the timer. We can create a specific pin for this, based on the `sid::pin` class:

```
class clock_pin_t: public sid::pin
  {
    // clock_pin_t is a specialized pin.
    // It calls timer->tick() whenever a value is driven.
  public:
    clock_pin_t(Timer* t): timer(t) { }
    void driven(sid::host_int_4 value) { timer->tick(); }
  private:
    Timer* timer;
  };
  friend class clock_pin_t;
  clock_pin_t clock_pin;

  // This method is called whenever the scheduler delivers an event,
  // because the "divided-clock-event" pin will be connected to the
  // scheduler.  It is a specialized pin with these fixed semantics.
  void tick();
```

And the following component API method is used to publicize this pin:

```
// Return a list of pin names which are visible to other components.

vector<string>
Timer::pin_names()
{
  vector<string> pins;
  pins.push_back("divided-clock-event");
  return pins;
}


// Find a pin of a given name.
// We only recognize "divided-clock-event".

sid::pin*
Timer::find_pin(const string& name)
{
  if (name == "divided-clock-event")
    return& clock_pin;

  return 0;
}
```

Step 5: Define the Bus

Like pins, buses work as demand-driven functional callbacks. The simulator abstracts buses to a read/write interface on address and data.

Example

The timer bus interface can be declared as follows. It should be nested within the Timer class to facilitate information hiding:

```
// register_bus is a specialized bus.
// It handles the majority of the component's functionality, since
// that is mostly controlled by the timer's register set.

class register_bus: public sid::bus
{
public:

 register_bus(Timer* t): timer(t) { }

  // Prototypes for bus read/write methods of all kinds.

  sid::bus::status read(sid::host_int_4 addr, sid::little_int_1& data);
  sid::bus::status read(sid::host_int_4 addr, sid::big_int_1& data);
  sid::bus::status read(sid::host_int_4 addr, sid::little_int_2& data);
  sid::bus::status read(sid::host_int_4 addr, sid::big_int_2& data);
  sid::bus::status read(sid::host_int_4 addr, sid::little_int_4& data);
  sid::bus::status read(sid::host_int_4 addr, sid::big_int_4& data);
  sid::bus::status read(sid::host_int_4 addr, sid::little_int_8& data);
  sid::bus::status read(sid::host_int_4 addr, sid::big_int_8& data);
  sid::bus::status write(sid::host_int_4 addr, sid::little_int_1 data);
  sid::bus::status write(sid::host_int_4 addr, sid::big_int_1 data);
  sid::bus::status write(sid::host_int_4 addr, sid::little_int_2 data);
  sid::bus::status write(sid::host_int_4 addr, sid::big_int_2 data);
  sid::bus::status write(sid::host_int_4 addr, sid::little_int_4 data);
  sid::bus::status write(sid::host_int_4 addr, sid::big_int_4 data);
  sid::bus::status write(sid::host_int_4 addr, sid::little_int_8 data);
  sid::bus::status write(sid::host_int_4 addr, sid::big_int_8 data);

private:
  Timer* timer;
};
friend class register_bus;
register_bus bus;
```

Each `read()` and `write()` method must be defined before a bus object may be instantiated. There is a read and write method for each byte order and each transaction width. To simplify this example, only 32-bit (4-byte) read and write methods will be supported for the timer's 32-bit register file. All of the remaining methods will return an error, but in a more complete implementation, should manage these smaller bus transactions when it makes sense.

```
sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::little_int_1& data)
{
  return sid::bus::unpermitted;
}
```

It is worth noting that all addresses seen by this component will be relative to 0. This is because the memory mapped address range is not known in advance. In a complete simulated system, the mapper component will handle the mapping from a memory mapped region into the device's address space.

A contrived example illustrates how the bus interface works:

```
  host_int_4 addr = 0;
  host_int_4 check, val = 0x5555;
  sid::bus* bus = component->find_bus("registers");

  bus->write(addr, val);  // write a value.
  bus->read(addr, check); // read back value.

  assert(val == check);
```

Other bus-related API methods that must be provided are shown below.

These methods enable supervisor components to connect to this component's bus:

```
// Return a list of bus names. We have just one-"registers".

vector<string>
Timer::bus_names()
{
  vector<string> buses;
  buses.push_back("registers");
  return buses;
}

sid::bus*
Timer::find_bus(const string& name)
{
  if (name == "registers")
    return& bus;
  return 0;
}

sid::bus*
Timer::connected_bus(const string& name)
{
  // No connected buses; return a null pointer.
  return 0;
}
```

Step 6: Add Scheduling Support

The scheduler is another SID component that is present in typical simulation systems. In order to request timed events to be delivered to a component, the component must negotiate them using a well-defined protocol. The component connects two pins to the scheduler:

```
divided-clock-event
divided-clock-control
```

Events are scheduled by communicating to the scheduler using this pin interface. To schedule an event at a future time, that time (as the number of seconds since UNIX epoch) must be carried across the `divided-clock-control` pin. If the most significant bit is set, then the event is to be delivered regularly with the specified frequency.

Example

To cancel an event subscription, the value 0 must be carried across the divided-clock-control pin. To simplify the implementation, these routines have been encapsulated within appropriately named methods:

```
// Schedule an event to be delivered at a later time.

void
Timer::schedule(sid::host_int_4 time)
{
  // The scheduler component tests bit 31 of a value carried on its
  // "control" pin.  If this bit is set, the event will be delivered
  // routinely at the specified interval.  Otherwise, the event will
  // only occur once.

  assert ((time&  0x80000000) == 0);
  assert ((time&  0x7FFFFFFF) != 0);

  if (scheduler_pin)
    scheduler_pin->driven(0x80000000 | time);
}


// Cancel any pending event.

void
Timer::cancel()
{
  // Cancel the event by driving a zero value to the scheduler.

  if (scheduler_pin)
    scheduler_pin->driven(0);
}

// Reset the schedule, in case the timer’s enable or divisor registers
// have been altered.

void
Timer::reset_schedule()
{
  cancel();

  if (!enabled)
    return;

  assert (prescale <= 2);
  unsigned divisor = 1 << (prescale * 4);

  schedule(divisor);
}
```

Step 7: Complete All Abstract Methods

Now all of the remaining abstract methods from the `sid::component` class need to be implemented. Often, these method bodies are trivial when the component does not utilize the entire API.

```
vector<string>
```

```
Timer::accessor_names()
{
  // No accessors.
  return vector<string>();
}

sid::component::status
Timer::connect_accessor(const string& name, sid::bus* bus)
{
  // No accessors; any name is unknown.
  return sid::component::not_found;
}

sid::component::status
Timer::disconnect_accessor(const string& name, sid::bus* bus)
{
  // No accessors; any name is unknown.
  return sid::component::not_found;
}

vector<string>
Timer::attribute_names()
{
  return vector<string>();
}

vector<string>
Timer::attribute_names(const string& category)
{
  return vector<string>();
}

string
Timer::attribute_value(const string& name)
{
  // No attributes-return the empty string for any attribute value.
  return string();
}

sid::component::status
Timer::set_attribute_value(const string& name, const string& value)
{
  // No attributes-return not_found regardless of attribute name.
  return sid::component::not_found;
}

vector<sid::component*>
Timer::related_components(const string& rel)
{
  // No related components.
  return vector<sid::component*>();
}

sid::component::status
Timer::unrelate(const string& rel, sid::component* c)
{
  // No related components; always unfound.
  return sid::component::not_found;
}

sid::component::status
Timer::relate(const string& rel, sid::component* c)
{
  // No related components; always unfound.
  return sid::component::not_found;
}

vector<string>
Timer::relationship_names()
{
  // No relations.
  return vector<string>();
}
```

Step 8: Complete the Model Functionality

For the timer, the functionality can be easily inferred from the hardware's functional description. For example, the bus' `read` call-back method would look something like the following:

```
// Handle 32-bit (little endian) reads.
// If the address is not 32-bit aligned or does not match any register
// address, return an error.

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::little_int_4& data)
{
  if (addr % 4 != 0)
    return sid::bus::misaligned;

  switch (addr)
    {
    case 0x0:
      data = timer->load_value;
      break;

    case 0x4:
      data = timer->counter;
      break;

    case 0x8:
      data =
        (timer->enabled << 7) |
        (timer->mode << 6) |
        (timer->prescale << 2);
      break;

    case 0xC:
      break;

    default:
      return sid::bus::unmapped;
    }

  return sid::bus::ok;
}
```

The address (`addr`) is a localized offset into the component's address space. The switch statement responds to these offsets by setting the load or control register value, or by clearing the interrupt, as appropriate.

To simplify the implementation and alleviate duplicated code, the big-endian version of `read()`:

```
// Handle 32-bit (big endian) reads.
// Just do a little endian read and rearrange the result.

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::big_int_4& data)
{
  sid::little_int_4 le_data;
  sid::bus::status st = read(addr, le_data);
  data.set_target_memory_value (le_data.target_memory_value ());
  return st;
}
```

This is how the `write()` method might look for a little-endian 32-bit read:

```
// Handle 32-bit (little endian) writes.
// If the address is not 32-bit aligned or does not match any register
// address, return an error.

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::little_int_4 data)
{
  if (addr % 4 != 0)
    return sid::bus::misaligned;

  switch (addr)
    {
    case 0x0:
      // A write to LOAD_REG.
      // Clear top 16 bits when loading a new value.
      timer->load_value = data&  0xFFFF;
      // Reset the counter value.
      timer->counter = timer->load_value;
      break;

    case 0x4:
      break;

    case 0x8:
      // A write to CTL_REG.
      timer->prescale = (data&  0x0C) » 2;
      timer->enabled = ((data&  0x80) == 0x80);
      timer->mode = ((data&  0x40) » 6) ? PERIODIC : FREERUNNING;
      timer->reset_schedule();
      break;

    case 0xC:
      timer->drive_interrupt(0);
      break;

    default:
      return sid::bus::unmapped;
    }

  return sid::bus::ok;
}
```

Again,

```
// Handle 32-bit (big endian) writes.
// Just rearrange the data and do a little endian write.

sid::bus::status
```

```
Timer::register_bus::write(sid::host_int_4 addr, sid::big_int_4 data)
{
  sid::little_int_4 le_data;
  le_data.set_target_memory_value (data.target_memory_value ());
  return write(addr, le_data);
}
```

Step 9: Configure the Connection

Now that the timer is implemented, it is time to hook it up and try it out. The simulator takes its configuration directions from a configuration file.

Example

Here is a configuration file fragment that illustrates the use of the new timer component:

```
...
# components
new ARM7100 cpu
new arm-timer timer1
new arm-timer timer2
new mapper map
# pin connections
connect-pin timer1 interrupt -> cpu INTR
connect-pin timer2 interrupt -> cpu INTR
# bus connections
connect-bus cpu data-bus map access-port
connect-bus map [0xa0000-0xa0010] timer1 registers
connect-bus map [0xa0020-0xa0030] timer2 registers
...
```

The configuration file segment instructs the simulator to instantiate the following four components: a CPU, two timers, and an address mapper.

The strings used to identify the components are specified in a shared library wrapper described in the *API Reference* manual. Like the bus and pin names, they are arbitrary, and must be determined from the component documentation. The configuration file directs SID to connect the output pins of the two timers to the interrupt input pin of the CPU.

Note: The names `timer1` and `timer2` refer to the instance names defined in the script, while the name of the pin, `interrupt`, is available by virtue of the fact that the `sid::component::pin_names()` method names it.

The last part of the file connects the CPU data bus to the address mapper, and then adds the timers at address 0xa0000 and 0xa0020, using the address separation defined in the hardware specification (Figure 1).

This is just a fragment of the full configuration file, which must also define search paths to find the component libraries and add some memory so that a target program can be loaded and run.

Chapter 4. Theory of Operations for bridge-tcl Component

A bridge-tcl component is a shell that hooks all SID API calls to an embedded Tcl interpreter so that they can be handled as Tcl procedure calls. In addition, SID API calls are exposed to that interpreter, so the Tcl procedures can call back out to the C++ system. With these two capabilities, a user-provided Tcl package may become a first class SID component.

Objects such as bus, component, and pin pointers may be passed through Tcl scripts safely, because the bridging calls represent these as unique strings, and convert them back to C++ pointers automatically. Any pointers seen through incoming call arguments, or outgoing call return values, are transparently converted into unique long-lived opaque strings. This way, C++ pointers can safely pass through the Tcl bridge in both directions.

Unlike C++ components, Tcl scripts that run in a bridge-tcl do not have access to the `sidutil::` group of utility classes. This means that only low level operations are directly provided, and `sidutil::` abstractions would need to be rewritten (if needed) in tcl.

Incoming SID API Calls

Almost all incoming SID API calls are passed through verbatim to the embedded Tcl interpreter. (Exceptions are parametrized and noted below.) Plain types are mapped according to the table below: C++ object to Tcl for arguments, and Tcl to C++ for return values. If Tcl procedures by the appropriate names are not loaded into the interpreter by the time they are invoked from another SID component, a TCL ERROR message is printed to cerr, and a function-specific error indication is returned.

Calls belonging to `sid::pin` and `sid::bus` are similarly mapped to Tcl procedure calls. The C++ pin/bus object on which they are called is passed to the procedures as an extra argument. (C++ pin/bus objects may be constructed for a Tcl component through special callback functions, listed below.)

Functions with multiple outputs, like the `sid::bus::read` reference arguments, map to Tcl procedures returning a list with the mapped C++ return type as first element, and the output reference argument as second element.

C++ Type	**Tcl Type**
string	string
`vector<string>`	list of strings
component,bus,pin pointer	opaque strings
`{little,big,host}_int_{1,2,4,8}`	numeric integer - care with 64-bit ints
`component::status`	string: `ok, bad_value, not_found`
`bus::status`	string: `ok, misaligned, unmapped, unpermitted`
`vector<component*>`	list of opaque strings
`vector<pin*>`	list of opaque strings
0 (null pointer)	""

In `sid::component`:

Incoming C++ call	**Outgoing Tcl call**
`attribute_names()`	`attribute_names`
`attribute_names(category)`	`attribute_names_in_category $category`
`attribute_value(name)`	`attribute_value $name`
`set_attribute_value(name,value)`	`set_attribute_value $name $value`
`pin_names`	`pin_names`
`find_pin(name)`	`find_pin $name`
`connect_pin(name, pin)`	`connect_pin $name $pin`
`disconnect_pin(name, pin)`	`disconnect_pin $name $pin`
`connected_pins(name)`	`connected_pins $name`
`bus_names`	`bus_names`
`find_bus(name)`	`find_bus $name`
`accessor_names`	`accessor_names`
`connect_accessor(name,bus)`	`connect_accessor $name $bus`
`disconnect_accessor(name,bus)`	`disconnect_accessor $name $bus`
`connected_bus(name)`	`connected_bus $name`
`relationships()`	`relationships`
`set_related_components(rel,comps)`	`set_related_components $rel $comp1 $comp2 $comp3 ...`
`related_components(rel)`	`related_components $rel`

In `sid::pin`:

Incoming C++ call	**Outgoing Tcl call**
`driven`	`driven $pin`
`driven(value)`	`driven_h4 $pin $value`

In `sid::bus`, for `host_int_4` address and `{big, little}_int_Y` data types:

Incoming C++ call	**Outgoing Tcl call**
`read(address,data)`	`read_h4_{l,b}Y $address ** return [list $status $data] **`
`write(address,data)`	`write_h4_{l,b}Y $address $data`

Incoming C++ calls that access these parameters are not passed through to the embedded Tcl interpreter. Rather, they are processed in the bridge code attribute.

load!

Loads the given Tcl script into the interpreter using the source procedure pin

event!

Passes control to the tcl/Tk event loop, in non-blocking mode

Outgoing SID API Calls

Once a Tcl program is loaded into the interpreter, it is able to make outgoing SID API calls, not merely respond to incoming ones. All SID API functions are exposed to Tcl as procedure hooks, in a very symmetric way to the incoming calls. Simply, each function in the incoming set has a shadow: `sid::component::FUNCTION`, `sid::pin::FUNCTION` or `sid::bus::FUNCTION`, as appropriate. Each outgoing procedure takes a receiver handle (the same opaque string passed in an incoming call) as its first argument.

There is no checking that would prevent an outgoing SID API call from becoming recursive and referring to the originating component, either directly or indirectly. As for all other components, infinite recursion prevention is the responsibility of the component author.

Incoming	Outgoing
`attribute_value $name`	`sid::component::attribute_value $component $name`
`driven_h4 $pin $value`	`sid::pin::driven_h4 $pin $value`

There are some special outgoing functions that function as constructors for local object handles.

sid::component::this

Returns an opaque string handle to this component.

sid::pin::new

Returns an opaque string handle to a new private C++ pin, usable as a return value to find_pin.

sid::bus::new

Returns an opaque string handle to a new private C++ bus, usable as a return value to find_bus.

Appendix A. Sample Source Code

timer.cxx Source File

```
// Copyright (C) 2001 Red Hat
//
// Description: A complete version of the example given in the
// Component Developers’ Kit (CDK) reference guide.
//
// Noteworthy remarks about the code in this source file:
// * For clarity, the C++ "using" keyword is used sparingly and
//   all members
//   of the "sid::" namespace appear fully qualified.
// * Documentation for the Standard Template Library, plus any obscure
//   C++ syntax you may encounter can be discovered in the 3rd edition
//   of “The C++ Programming Language”, by Stroustrup.
//   ISBN 0-201-88954-4.

#include <sidcomp.h>
#include <sidso.h>
#include <sidtypes.h>

#include <set>
#include <string>
#include <vector>


// Wrap everything in a unique namespace
namespace timer_example
{


// Import standard types into this namespace.
using namespace std;


// None of the sid API routines is allowed to throw an exception.
#define NT throw()


class Timer: public virtual sid::component
{
public:
  Timer()
    :scheduler_pin(0), clock_pin(this), bus(this), enabled(false) { }

  // Provide implementations for abstract methods in sid::component.
  // See include/sidcomp.h.

  vector<string> pin_names() NT;
  sid::pin* find_pin(const string& name) NT;
  sid::component::status connect_pin(const string& name, sid::pin* pin) NT;
  sid::component::status disconnect_pin(const string& name, sid::pin* pin) NT;
  vector<sid::pin*> connected_pins(const string& name) NT;

  vector<string> accessor_names() NT;
  sid::component::status connect_accessor(const string& name, sid::bus* bus) NT;
  sid::component::status disconnect_accessor(const string& name, sid::bus* bus) NT;

  vector<string> bus_names() NT;
```

```
    sid::bus* find_bus(const string& name) NT;
    sid::bus* connected_bus(const string& name) NT;

    vector<string> attribute_names() NT;
    vector<string> attribute_names(const string& category) NT;
    string attribute_value(const string& name) NT;
    sid::component::status set_attribute_value(const string& name, const string& value) N

    vector<string> relationship_names() NT;
    sid::component::status relate(const string& rel, sid::component* c) NT;
    sid::component::status unrelate(const string& rel, sid::component* c) NT;
    vector<sid::component*> related_components(const string& rel) NT;

  private:
    // A netlist, which tracks pins connected to the interrupt pin.
    typedef set<sid::pin*> netlist_t;
    netlist_t intpin_netlist;

    // A handle to the scheduler's "control" pin.
    // The scheduler is a preexisting SID component which can schedule
    // and deliver events to a component via an "event" pin.
    // See schedule() for more details.
    sid::pin* scheduler_pin;

    // Schedule an event for some time in the future.
    void schedule(sid::host_int_4 time);

    // Cancel any pending events.
    void cancel();

    // Reschedule an event.
    void reset_schedule();

    // This method is called whenever the scheduler delivers an event,
    // because the "divided-clock-event" pin will be connected to the
    // scheduler.  It is a specialized pin with these fixed semantics.
    // Refer to class clock_pin_t below.
    void tick();

    // Drive a value on the interrupt pin (which propagates to all
    // connected pins).
    void drive_interrupt(sid::host_int_4 value);

    class clock_pin_t: public sid::pin
    {
      // clock_pin_t is a specialized pin.
      // It calls timer->tick() whenever a value is driven.
    public:
      clock_pin_t(Timer* t): timer(t) { }
      void driven(sid::host_int_4 value) NT { timer->tick(); }
    private:
      Timer* timer;
    };
    friend class clock_pin_t;
    clock_pin_t clock_pin;


    // register_bus is a specialized bus.
    // It handles the majority of the component's functionality, since
    // that is mostly controlled by the timer's register set.

    class register_bus: public sid::bus
    {
```

```
  public:
    register_bus(Timer* t): timer(t) { }

    // Prototypes for bus read/write methods of all kinds.

    sid::bus::status read(sid::host_int_4 addr, sid::little_int_1& data) NT;
    sid::bus::status read(sid::host_int_4 addr, sid::big_int_1& data) NT;
    sid::bus::status read(sid::host_int_4 addr, sid::little_int_2& data) NT;
    sid::bus::status read(sid::host_int_4 addr, sid::big_int_2& data) NT;
    sid::bus::status read(sid::host_int_4 addr, sid::little_int_4& data) NT;
    sid::bus::status read(sid::host_int_4 addr, sid::big_int_4& data) NT;
    sid::bus::status read(sid::host_int_4 addr, sid::little_int_8& data) NT;
    sid::bus::status read(sid::host_int_4 addr, sid::big_int_8& data) NT;

    sid::bus::status write(sid::host_int_4 addr, sid::little_int_1 data) NT;
    sid::bus::status write(sid::host_int_4 addr, sid::big_int_1 data) NT;
    sid::bus::status write(sid::host_int_4 addr, sid::little_int_2 data) NT;
    sid::bus::status write(sid::host_int_4 addr, sid::big_int_2 data) NT;
    sid::bus::status write(sid::host_int_4 addr, sid::little_int_4 data) NT;
    sid::bus::status write(sid::host_int_4 addr, sid::big_int_4 data) NT;
    sid::bus::status write(sid::host_int_4 addr, sid::little_int_8 data) NT;
    sid::bus::status write(sid::host_int_4 addr, sid::big_int_8 data) NT;

  private:
    Timer* timer;
  };
  friend class register_bus;
  register_bus bus;

  // Data members that represent the timer's internal state.

  bool enabled;
  sid::host_int_2 load_value, prescale, counter;
  enum timer_mode { PERIODIC, FREERUNNING } mode;
};


// Return a list of pin names which are visible to other components.

vector<string>
Timer::pin_names() NT
{
  vector<string> pins;
  pins.push_back("divided-clock-event");
  return pins;
}


// Return a list of pins that are connected to a named pin.
// We recognize "interrupt" and "divided-clock-control".

vector<sid::pin*>
Timer::connected_pins(const string& name) NT
{
  vector<sid::pin*> pins;
  netlist_t::const_iterator it;

  if (name == "interrupt")
    {
      for (it = intpin_netlist.begin(); it != intpin_netlist.end();
           it++)
        {
          pins.push_back(*it);
        }
      return pins;
    }
  else if (name == "divided-clock-control")
    {
      pins.push_back(scheduler_pin);
      return pins;
    }
  return vector<sid::pin*>();
}


// Connect a pin to a named pin.
// We recognize "interrupt" and "divided-clock-control".

// We allow multiple pins to be connected to the interrupt pin (with
// infinite fan-out!), so these are kept in a netlist.  For
// efficiency, the STL container chosen for the netlist ensures that
// no duplicate pin handles are stored.

sid::component::status
Timer::connect_pin(const string& name, sid::pin* pin) NT
{
  if (name == "interrupt")
    {
      // Add this pin to the netlist.
      intpin_netlist.insert(intpin_netlist.end(), pin);
      return sid::component::ok;
    }
  else if (name == "divided-clock-control")
    {
      // Reassign the scheduler pin.
      scheduler_pin = pin;
      return sid::component::ok;
    }
  return sid::component::not_found;
}

// Disconnect a pin from a named pin.

sid::component::status
Timer::disconnect_pin(const string& name, sid::pin* pin) NT
{
  if (name == "interrupt")
    {
      // Remove this pin from the netlist.
      if (intpin_netlist.erase(pin) > 0)
        return sid::component::ok;
    }
  else if (name == "divided-clock-control"& & scheduler_pin == pin)
    {
      // Elsewhere, we make sure to not use this pin if it’s null.
      scheduler_pin = 0;
      return sid::component::ok;
    }
  return sid::component::not_found;
}


// Find a pin of a given name.
// We only recognize "divided-clock-event".

sid::pin*
```

```
Timer::find_pin(const string& name) NT
{
  if (name == "divided-clock-event")
    return& clock_pin;

  return 0;
}


vector<string>
Timer::accessor_names() NT
{
  // No accessors.
  return vector<string>();
}


sid::component::status
Timer::connect_accessor(const string& name, sid::bus* bus) NT
{
  // No accessors; any name is unknown.
  return sid::component::not_found;
}


sid::component::status
Timer::disconnect_accessor(const string& name, sid::bus* bus) NT
{
  // No accessors; any name is unknown.
  return sid::component::not_found;
}


// Return a list of bus names. We have just one-"registers".

vector<string>
Timer::bus_names() NT
{
  vector<string> buses;
  buses.push_back("registers");
  return buses;
}

sid::bus*
Timer::find_bus(const string& name) NT
{
  if (name == "registers")
    return& bus;
  return 0;
}


sid::bus*
Timer::connected_bus(const string& name) NT
{
  // No connected buses; return a null pointer.
  return 0;
}


vector<string>
Timer::attribute_names() NT
{
  // No attributes; return an empty vector.
  return vector<string>();
}

vector<string>
Timer::attribute_names(const string& category) NT
{
  // No attributes, regardless of category. Return an empty vector.
  return vector<string>();
}

string
Timer::attribute_value(const string& name) NT
{
  // No attributes-return the empty string for any attribute value.
  return string();
}

sid::component::status
Timer::set_attribute_value(const string& name, const string& value) NT
{
  // No attributes-return not_found regardless of attribute name.
  return sid::component::not_found;
}


vector<sid::component*>
Timer::related_components(const string& rel) NT
{
  // No related components.
  return vector<sid::component*>();
}

sid::component::status
Timer::unrelate(const string& rel, sid::component* c) NT
{
  // No related components; always unfound.
  return sid::component::not_found;
}

sid::component::status
Timer::relate(const string& rel, sid::component* c) NT
{
  // No related components; always unfound.
  return sid::component::not_found;
}

vector<string>
Timer::relationship_names() NT
{
  // No relations.
  return vector<string>();
}



void
Timer::drive_interrupt(sid::host_int_4 value) NT
{
  // Iterate the netlist, driving the value to all pins connected to
  // the interrupt pin.

  for (netlist_t::const_iterator it = intpin_netlist.begin();
       it != intpin_netlist.end();
       it++)
    {
      (*it)->driven(value);
    }
}


// Schedule an event to be delivered at a later time.

void
Timer::schedule(sid::host_int_4 time)
{
  // The scheduler component tests bit 31 of a value carried on its
  // "control" pin.  If this bit is set, the event will be delivered
  // routinely at the specified interval.  Otherwise, the event will
  // only occur once.

  assert ((time&  0x80000000) == 0);
  assert ((time&  0x7FFFFFFF) != 0);

  if (scheduler_pin)
    scheduler_pin->driven(0x80000000 | time);
}


// Cancel any pending event.

void
Timer::cancel()
{
  // Cancel the event by driving a zero value to the scheduler.

  if (scheduler_pin)
    scheduler_pin->driven(0);
}


// Reset the schedule, in case the timer's enable or divisor registers
// have been altered.

void
Timer::reset_schedule()
{
  cancel();

  if (!enabled)
    return;

  assert (prescale <= 2);
  unsigned divisor = 1 << (prescale * 4);

  schedule(divisor);
}


// Handle 32-bit (little endian) reads.
// If the address is not 32-bit aligned or does not match any register
// address, return an error.

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::little_int_4& data) NT
{
  if (addr % 4 != 0)
    return sid::bus::misaligned;

  switch (addr)
    {
    case 0x0:
      data = timer->load_value;
      break;

    case 0x4:
      data = timer->counter;
      break;

    case 0x8:
      data =
        (timer->enabled << 7) |
        (timer->mode << 6) |
        (timer->prescale << 2);
      break;

    case 0xC:
      break;

    default:
      return sid::bus::unmapped;
    }

  return sid::bus::ok;
}


// Handle 32-bit (big endian) reads.
// Just do a little endian read and rearrange the result.

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::big_int_4& data) NT
{
  sid::little_int_4 le_data;
  sid::bus::status st = read(addr, le_data);
  data.set_target_memory_value (le_data.target_memory_value ());
  return st;
}


// Handle 32-bit (little endian) writes.
// If the address is not 32-bit aligned or does not match any register
// address, return an error.

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::little_int_4 data) NT
{
  if (addr % 4 != 0)
    return sid::bus::misaligned;

  switch (addr)
    {
    case 0x0:
      // A write to LOAD_REG.
      // Clear top 16 bits when loading a new value.
      timer->load_value = data&  0xFFFF;
      // Reset the counter value.
      timer->counter = timer->load_value;
      break;
    case 0x4:
      break;

    case 0x8:
      // A write to CTL_REG.
      timer->prescale = (data&  0x0C) » 2;
      timer->enabled = ((data&  0x80) == 0x80);
      timer->mode = (data&  0x40) ? PERIODIC : FREERUNNING;
      timer->reset_schedule();
      break;

    case 0xC:
      timer->drive_interrupt(0);
      break;

    default:
      return sid::bus::unmapped;
    }

  return sid::bus::ok;
}


// Handle 32-bit (big endian) writes.
// Just rearrange the data and do a little endian write.

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::big_int_4 data) NT
{
  sid::little_int_4 le_data;
  le_data.set_target_memory_value (data.target_memory_value ());
  return write(addr, le_data);
}

// For simplicity, bus accesses that are not 32-bits wide are not
// handled.  Ideally, different widths should be handled sensibly.
// For example, a one-byte write to location n+1, where n is 32-bit
// aligned, should behave as you might expect.

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::little_int_1& data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::big_int_1& data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::little_int_2& data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::big_int_2& data) NT
{
  return sid::bus::unpermitted;
}
```

```
sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::little_int_8& data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::read(sid::host_int_4 addr, sid::big_int_8& data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::little_int_1 data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::big_int_1 data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::little_int_2 data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::big_int_2 data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::little_int_8 data) NT
{
  return sid::bus::unpermitted;
}

sid::bus::status
Timer::register_bus::write(sid::host_int_4 addr, sid::big_int_8 data) NT
{
  return sid::bus::unpermitted;
}

// Called when the scheduled event arrives.
// Decrement the counter register and check for an interrupt
// condition-and if so, drive the interrupt pin.

void
Timer::tick()
{
  if (!enabled) return;

  counter-;
  if (counter == 0)
    {
      if (mode == PERIODIC)
        {
          counter = load_value;
          drive_interrupt(1);
        }
      else
        {
          // Rolls over from maximum value; no interrupts.
          counter = 0xFFFF;
        }
    }
}


// Return a list of component types supported by this library.

static vector<string>
TimerListTypes() NT
{
  vector<string> types;
  types.push_back("hw-timer-example");
  return types;
}

// Instantiate a component, given a specified component type.

static sid::component*
TimerCreate(const string& typeName) NT
{
  if (typeName == "hw-timer-example")
    return new Timer();

  return 0;
}

// Destruct a component instance.

static void
TimerDelete(sid::component* c) NT
{
  delete dynamic_cast<Timer*>(c);
}


} // end namespace


// This symbol is used by the library loader to validate the library
// and instantiate components of the types supported by this library.

extern const sid::component_library example_component_library;

const sid::component_library example_component_library DLLEXPORT =
{
  sid::COMPONENT_LIBRARY_MAGIC,
  & timer_example::TimerListTypes,
  & timer_example::TimerCreate,
  & timer_example::TimerDelete
};
```