Resource Arbitration and Power Management

From TinyOS Wiki

Contents

Introduction

TinyOS distinguishes between three kinds of resource abstractions: **dedicated**, **virtualized**, and **shared**.

Two fundamental questions must be asked about each type of abstraction.

1. How can a client gain access to the resource provided through this abstraction?
2. How can the power state of that resource be controlled?

Components offer resource sharing mechanisms and power mangement capabilites according to the goals and level of abstraction required by their clients.

An abstraction is dedicated if it represents a resource which a subsystem needs exclusive access to at all times. In this class of resources, no sharing policy is needed since only a single component ever requires use of the resource. Resource clients simply call commands from the interfaces provided by the resource just as they would with any other TinyOS component. Resources of this type provide either an `AsyncStdControl`, `StdControl`, or `SplitControl` interface for controlling their power states. The definition of each of these interfaces can be found in `tinyos-2.x/tos/interfaces`.

```
interface AsyncStdControl {
  async command error_t start();
  async command error_t stop();
}
```

```
interface StdControl {
  command error_t start();
  command error_t stop();
}

interface SplitControl {
  command error_t start();
  command void startDone(error_t error);
  command error_t stop();
  command void stopDone(error_t error);
}
```

Currently, the power states of all dedicated resources are controlled by one of these three interfaces. They are only allowed to enter one of two logical power states (on/off), regardless of the number of physical power states provided by the hardware on top of which their resource abstraction has been built. Which of these interfaces is provided by a particular resource depends on the timing requirements for physically powering it on or off.

Virtual abstractions hide multiple clients from each other through software virtualization. Every client of a virtualized resource interacts with it as if it were a dedicated resource, with all virtualized instances being multiplexed on top of a single underlying resource. Because the virtualization is done in software, there is no upper bound on the number of clients using the abstraction, barring memory or efficiency constraints. The power states of a virtualized resource are handled automatically, and no interface is provided to the user for explicity controlling its power state. As they are built on top of shared resources, the reason their power states can be automatically controlled will become clearer after reading the following section.

Dedicated abstractions are useful when a resource is always controlled by a single component. Virtualized abstractions are useful when clients are willing to pay a bit of overhead and sacrifice control in order to share a resource in a simple way. There are situations, however, when many clients need precise control of a resource. Clearly, they can't all have such control at the same time: some degree of multiplexing is needed.

A motivating example of a shared resource is a bus. The bus may have multiple peripherals on it, corresponding to different subsystems. For example, on the Telos platform the flash chip (storage) and the radio (network) share a bus. The storage and network stacks need exclusive access to the bus when using it, but they also need to share it with the other subsystem. In this case, virtualization is problematic, as the radio stack needs to be able to perform a series of operations in quick succession without having to reacquire the bus in each case. Having the bus be a shared resource allows the radio stack to send a series of operations to the radio atomically, without having to buffer them all up in memory beforehand (introducing memory pressure in the process).

In TinyOS, a resource **arbiter** is responsible for multiplexing between the different clients of a shared resource. It determines which client has access to the resource at which time. While a client holds a resource, it has complete and unfettered control. Arbiters assume that clients are cooperative, only acquiring the resource when needed and holding on to it no longer than necessary. Clients explicitly release resources: there is no way for an arbiter to forcibly reclaim it.

Shared resources are essentially built on top of dedicated resources, with access to them being controlled by an arbiter component. In this way, **power managers** can be used to automatically control the power state of these resources through their `AsyncStdControl`, `StdControl`, or `SplitControl` interfaces. They communicate with the arbiter (through the use of a `ResourceDefaultOwner` interface), monitoring whether the resource is being used by any of its clients and powering it on/off accordingly. The figure below shows how an arbiter component and a power manager can be wired together to provide arbitration and automatic power management for a shared resource.

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-GiO1vY46' @ error/constitute.c/ReadImage/532. convert: missing an image filename `/tmp/transform_3f1ff0b16146-1.png' @ error/convert.c/ConvertImageCommand/3011.

Figure 1: Arbiters and Power Managers

The arbiter component provides the `Resource`, `ArbiterInfo`, `ResourceRequested`, and `ResourceDefaultOwner` interfaces and uses the `ResourceConfigure` interface. The power manager doesn't provide any interfaces, but uses one of either the `AsyncStdControl`, `StdControl`, or `SplitControl` interfaces from the underlying resource, as well as the `ResourceDefaultOwner` interface provided by the arbiter. The figure below shows how these interface are then wired together with the implementation of a shared resource. Please refer to TEP 108 for more information on arbiters and TEP 115 for more information on Power Managers.

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-cVRdbLJi' @ error/constitute.c/ReadImage/532. convert: missing an image filename `/tmp/transform_7bcd4d52a84c-1.png' @ error/convert.c/ConvertImageCommand/3011.

Figure 2: Shared Resource Configuration

From this figure, we see that the only interfaces exposed to a client through the shared resource abstraction are the `Resource` and `ResourceRequested` interfaces provided by the arbiter as well as any resource specific interfaces provided by the resource itself. It also uses a `ResourceConfigure` interface, expecting it to be implemented on a client by client basis depending on their requirements. A client requests access to a shared resource through the `Resource` interface and runs operations on it using whatever resource specific interfaces are provided. A client may choose to wire itself to the `ResourceRequested` interface if it wishes to hold onto a resource indefinitely and be informed whenever other clients request its use.

The rest of this tutorial is dedicated to teaching users how to use shared resources and show them how wiring is done between all components that make them up.

Specifically, this tutorial will teach users how to:

1. Wire in a shared resource for use by a client.
2. Use the `Resource` interface to gain access to a shared resource.
3. Change the arbitration policy used by a particular shared resource.
4. Wire up a power manager for use by a shared resource.

Working with Shared Resources

This section shows you how to gain access to and use shared resources in TinyOS. It walks through the process of making a request through the `Resource` interface and handling the `granted` event that is signaled back. We will connect multiple clients to a single shared resources and see how access to each of them gets arbitrated. We also show how to hold onto a resource until another client has requested it by implementing the `ResourceRequested` interface.

To begin, go to the `tinyos-2.x/apps/tutorials/SharedResourceDemo` directory and install this application on a mote. After installing the application you should see three leds flashing in sequence.

Let's take a look at the different components contained in this directory to see whats going on. Start with the top level application component: `SharedResourceDemoAppC`

```
configuration SharedResourceDemoAppC{
}
implementation {
  components MainC,LedsC, SharedResourceDemoC as App,
  new TimerMilliC() as Timer0,
  new TimerMilliC() as Timer1,
  new TimerMilliC() as Timer2;
  App -> MainC.Boot;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
  App.Timer1 -> Timer1;
  App.Timer2 -> Timer2;

  components
  new SharedResourceC() as SharedResource0,
  new SharedResourceC() as SharedResource1,
  new SharedResourceC() as SharedResource2;
  App.Resource0 -> SharedResource0;
  App.Resource1 -> SharedResource1;
  App.Resource2 -> SharedResource2;
  App.ResourceOperations0 -> SharedResource0;
  App.ResourceOperations1 -> SharedResource1;
  App.ResourceOperations2 -> SharedResource2;
}
```

Other than the instantiation and wiring of the interfaces provided by the `SharedResourceC` component, this configuration is identical to the one presented in Lesson 1 for the Blink Application.

All shared resources in TinyOS are provided through a generic component similar to the `SharedResourceC` component. A resource client simply instantiates a new instance of this component and wires to the interfaces it provides. In this application, three instances of the `SharedResourceC` component are instantiated and wired to three different clients from the `SharedResourceDemoC` component. Each instantiation provides a `Resource`, `ResourceOperations`, and `ResourceRequested` interface, and uses a `ResourceConfgigure` interface. In this example, no wiring is done to the `ResourceConfigure` or `ResourceRequested` interface as wiring to to these interfaces is optional. The `ResourceOperations` interface is an **EXAMPLE** of a resource specific interface that a resource may provide to perform operations on it. Calls to commands through this interface will only succeed if the client calling them happens to have access to the resource when they are called.

Let's take a look at the `SharedResourceDemoC` to see how access is actually granted to a Resource.

```
module SharedResourceDemoC {
  uses {
    interface Boot;
    interface Leds;
    interface Timer as Timer0;
    interface Timer as Timer1;
    interface Timer as Timer2;

    interface Resource as Resource0;
    interface ResourceOperations as ResourceOperations0;
    interface Resource as Resource1;
    interface ResourceOperations as ResourceOperations1;

    interface Resource as Resource2;
    interface ResourceOperations as ResourceOperations2;
  }
}
```

Each pair of `Resource/ResourceOperations` interfaces reperesents a different client of the shared resource used by this application. At boot time, we put in a request for the shared resource through each of these clients in the order (0,2,1).

```
event void Boot.booted() {
  call Resource0.request();
  call Resource2.request();
  call Resource1.request();
}
```

Each of these requests is serviced in the order of the arbitration policy used by the shared resource. In the case of `SharedResourceC`, a Round-Robin policy is used, so these requests are serviced in the order (0,1,2). If a first-come-first-serve policy were in use, they would we be serviced in the order the were put in, i.e. (0,2,1).

Whenever a client's request for a resource has been granted, the `Resource.granted()` event for that client gets signaled. In this application, the body of the granted event for each client simply performs an operation on the resource as provided through the `ResourceOperations` interface.

```
event void Resource0.granted() {
  call ResourceOperations0.operation();
}
event void Resource1.granted() {
  call ResourceOperations1.operation();
}
event void Resource2.granted() {
  call ResourceOperations2.operation();
}
```

Whenever one of these operations completes, a `ResourceOperations.operationDone()` event is signaled. Once this event is received by each client, a timer is started to hold onto the resource for 250 (binary) ms and an LED corresponding to that client is toggled.

```
 #define HOLD_PERIOD 250

 event void ResourceOperations0.operationDone(error_t error) {
   call Timer0.startOneShot(HOLD_PERIOD);
   call Leds.led0Toggle();
 }
 event void ResourceOperations1.operationDone(error_t error) {
   call Timer1.startOneShot(HOLD_PERIOD);
   call Leds.led1Toggle();
 }
 event void ResourceOperations2.operationDone(error_t error) {
   call Timer2.startOneShot(HOLD_PERIOD);
   call Leds.led2Toggle();
 }
```

Whenever one of these timers goes off, the client that started it releases the resource and immediately puts in a request for it again.

```
event void Timer0.fired() {
  call Resource0.release();
  call Resource0.request();
}
event void Timer1.fired() {
  call Resource1.release();
  call Resource1.request();
}
event void Timer2.fired() {
  call Resource2.release();
  call Resource2.request();
}
```

In this way, requests are continuously put in by each client, allowing the application to continuously flash the LEDs in the order in which requests are being serviced. As stated before, the `SharedResourceC` component services these requests in a round-robin fashion. If you would like to see the requests serviced in the order they are received (and see the LEDs flash accordingly), you can open up the `SharedResourceP` component in the `apps/tutorials/SharedResourceDemo` directory and replace the `RoundRobinArbiter` component with the `FcfsArbiter` component.

RoundRobinArbiter

```
configuration SharedResourceP {
	provides interface Resource[uint8_t id];
	provides interface ResourceRequested[uint8_t id];
	provides interface ResourceOperations[uint8_t id];
	uses interface ResourceConfigure[uint8_t id];
}
implementation {
  components new RoundRobinArbiterC(UQ_SHARED_RESOURCE) as Arbiter;
  ...
  ...
}
```

FcfsArbiter

```
configuration SharedResourceP {
	provides interface Resource[uint8_t id];
	provides interface ResourceRequested[uint8_t id];
	provides interface ResourceOperations[uint8_t id];
	uses interface ResourceConfigure[uint8_t id];
}
implementation {
  components new FcfsArbiterC(UQ_SHARED_RESOURCE) as Arbiter;
  ...
  ...
}
```

Looking through the rest of this component, you can see how its wiring matches the connections shown in Figure 2.

```
#define UQ_SHARED_RESOURCE   "Shared.Resource"
configuration SharedResourceP {
	provides interface Resource[uint8_t id];
	provides interface ResourceRequested[uint8_t id];
	provides interface ResourceOperations[uint8_t id];
	uses interface ResourceConfigure[uint8_t id];
}
implementation {
  components new RoundRobinArbiterC(UQ_SHARED_RESOURCE) as Arbiter;
  components new SplitControlPowerManagerC() as PowerManager;
  components ResourceP;
  components SharedResourceImplP;

  ResourceOperations = SharedResourceImplP;
  Resource = Arbiter;
  ResourceRequested = Arbiter;
  ResourceConfigure = Arbiter;
  SharedResourceImplP.ArbiterInfo -> Arbiter;
  PowerManager.ResourceDefaultOwner -> Arbiter;

  PowerManager.SplitControl -> ResourceP;
  SharedResourceImplP.ResourceOperations -> ResourceP;
}
```

Four different components are instantiated by this configuration:

```
components new RoundRobinArbiterC(UQ_SHARED_RESOURCE) as Arbiter;
components new SplitControlPowerManagerC() as PowerManager;
components ResourceP;
components SharedResourceImplP;
```

As we've already seen, the `RoundRobinArbiterC` component is used to provide arbitration between clients using `SharedResourceC`. The `SplitControlPowerManagerC` component is used to perform automatic power management of the resource to turn it on whenever a new client requests its use and shut it down whenever it goes idle. The `ResourceP` component is the implementation of a dedicated resource which provides a `SplitControl` interface and a `ResourceOperations` interface. This dedicated resource is wrapped by the `SharedResourceImplP` component in order to provide protected shared access to it. `SharedResourceImplP` wraps all the commands provided by the dedicated resource, and uses the `ArbiterInfo` interface to keep clients from calling them without first being granted access to the resource.

If you would like to see more examples of how to use the different arbiters and power managers provided in the default TinyOS distribution, please refer to the test applications located in `tinyos-2.x/apps/tests/TestArbiter` and `tinyos-2.x/apps/tests/TestPowerManager`. This tutorial has provided enough background information on how to use these components in order for you to sift through these applications on your own.

Conclusion

This tutorial has given an overview of how resource arbitration and mechanisms for performing power management on those resources is provided in TinyOS. It walked us through the steps necessary for:

1. Wiring in a shared resource for use by a client.
2. Using the `Resource` interface to gain access to a shared resource.

3. Changing the arbitration policy used by a particular shared resource.
4. Wrapping a dedicated resource and wiring in a power manager in order to create a shared resource.

While the power managers presented in this tutorial are powerful components for providing power management of shared resources, they are not the only power management mechanisms provided by TinyOS. Microcontroller power management is also preformed as outlined in TEP115. Whenever the task queue empties, the lowest power state that the microcontroller is capable of dropping to is automatically calculated and then switched to. In this way, the user is not burdened with explicity controlling these power states. The cc1000 and cc2420 radio implementations also provide "Low Power Listening" (LPL) interfaces for controlling their duty cycles. The LPL implementation for the cc2420 can be found under `tinyos-2.x/tos/chips/cc2420` and the LPL implementation for the cc1000 can be found under `tinyos-2.x/tos/chips/cc1000`. Take a look at lesson 16 to see how this interface is used.

Related Documentation

- TinyOS Programming Guide *Sections 6.2 and 7.4* (http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf)
- TEP 108: Resource Arbitration (http://www.tinyos.net/tinyos-2.x/doc/html/tep108.html)
- TEP 112: Microcontroller Power Management (http://www.tinyos.net/tinyos-2.x/doc/html/tep112.html)
- TEP 115: Power Management of Non-Virtualized Devices (http://www.tinyos.net/tinyos-2.x/doc/html/tep115.html)

< **Previous Lesson** | **Top** | **Next Lesson** >