Resource Arbitration

TEP:	108
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Kevin Klues
Author:	Philip Levis
Author:	David Gay
Author:	David Culler
Author:	Vlado Handziski

Abstract

This memo documents the general resource sharing mechanisms for TinyOS 2.x. These mechanisms are used to allow multiple software components to arbitrate access to shared abstractions.

1. Introduction

TinyOS 1.x has two mechanisms for managing shared resources: virtualization and completion events. A virtualized resource appears as an independent instance of an abstraction, such as the Timer interface in TimerC. A client of a Timer instance can use it independently of the others: TimerC virtualizes the underlying hardware clock into N separate timers.

Some abstractions, however, are not well suited to virtualization: programs need the control provided by a physical abstraction. For example, components in 1.x share a single communication stack, GenericComm. GenericComm can only handle one outgoing packet at a time. If a component tries to send a packet when GenericComm is already busy, then the call returns FAIL. The component needs a way to tell when GenericComm is free so it can retry. TinyOS 1.x provides the mechanism of a global completion event which is signaled whenever a packet send completes. Interested components can handle this event and retry.

This approach to physical (rather than virtualized) abstractions has several drawbacks:

- If you need to make several requests, you have to handle the possibility of a request returning FAIL at any point. This complicates implementations by adding internal states.
- You have no control over the timing of a sequence of operations. One example of when this can be a problem is timing-sensitive use of an A/D converter. You need a way to pre-reserve the use of the ADC so that its operations can be run at the exact moment they are desired.
- If a hardware resource supports reservation, you cannot express this via this software interface. For instance, I2C buses have a concept of "repeated start" when doing multiple bus transactions, but it is not clear how to use this in TinyOS 1.x's I2C abstraction.
- Most TinyOS 1.x services do not provide a very convenient way of monitoring an abstraction's availability for the purpose of retries, nor very clear documentation of which requests could happen simultaneously.

It should be clear that a single approach to resource sharing is not appropriate for all circumstances. For instance, requiring explicit reservation of a resource allows programs to have better timing guarantees for access to an A/D converter. If a program does not need precise timing guarantees, however (e.g., when measuring temperature in a biological monitoring application), this extra resource reservation step unnecessarily complicates code and can be handled nicely using virtualization. The following section introduces the concept of resource classes in order to address this issue. The sharing policy used by a particular resource abstraction is dictated by the resource class it belongs to.

2. Resource Classes

TinyOS 2.x distinguishes between three kinds of abstractions: *dedicated*, *virtualized*, and *shared*. Components offer resource sharing mechanisms appropriate to their goals and level of abstraction.

> **Note**
> It is important to point out that Hardware Presentation Layer (HPL) components of the HAA[1] can never be virtualized, as virtualization inevitably requires state. Depending on their expected use, HPL abstractions can either be dedicated or shared. For example, while hardware timers are rarely multiplexed between multiple components, buses almost always are. This can be seen on the MSP430 microcontroller, where the compare and counter registers are implemented as dedicated resources, and the USARTs are shared ones.

2.1 Dedicated

An abstraction is *dedicated* if it is 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. Examples of dedicated abstractions include interrupts and counters.

Dedicated abstractions MAY be annotated with the nesC attribute @atmostonce or @exactlyonce to provide compile-time checks that their usage assumptions are not violated.

Please refer to Appendix A for an example of how a dedicated resource might be represented, including the use of the nesC @exactlyonce attribute.

2.2 Virtualized

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. As virtualization usually requires keeping state that scales with the number of virtualized instances, virtualized resources often use the Service Instance pattern[3], which is based on a parameterized interface.

Virtualization generally provides a very simple interface to its clients. This simplicity comes at the cost of reduced efficiency and an inability to precisely control the underlying resource. For example, a virtualized timer resource introduces CPU overhead from dispatching and maintaining each individual virtual timer, as well as introducing jitter whenever two timers happen to fire at the same time. Please refer to Appendix A for an example of how such a virtualized timer resource might be implemented.

2.3 Shared

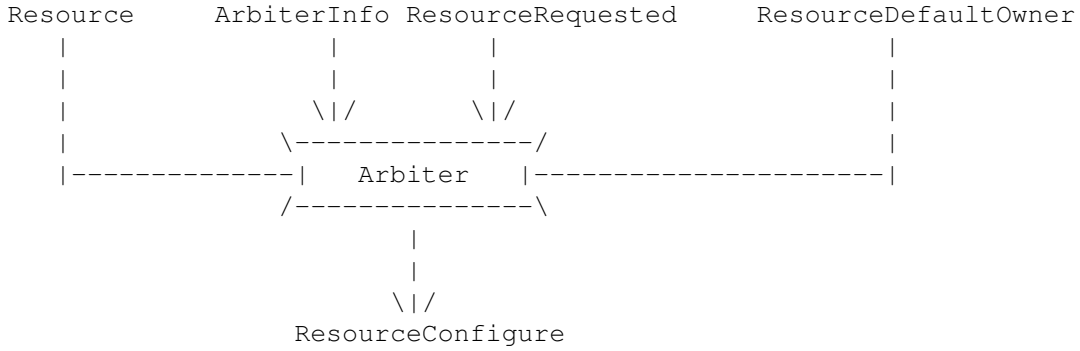
Dedicated abstractions are useful when a resource is always controlled by a single component. Virtualized abstractions are useful when clients are willing to pay a bit of overhead and sacrifice control in order to share a resource in a simple way. There are situations, however, when many clients need 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 2.x, 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. The following section is dedicated to describing the arbiter and its interfaces.

3. Resource Arbiters

Every shared resource has an arbiter to manage which client can use the resource at any given time. Because an arbiter is a centralized place that knows whether the resource is in use, it can also provide information useful for a variety of other services, such as power management. An arbiter MUST provide a parameterized Resource interface as well as an instance of the ArbiterInfo interface. The Resource interface is instantiated by different clients wanting to gain access to a resource. The ArbiterInfo interface is used by components that wish to retrieve global information about the status of a resource (i.e. if it is in use, who is using it, etc.). An arbiter SHOULD also provide a parameterized ResourceRequested interface and use a parameterized ResourceConfigure interface. It MAY also provide an instance of the ResourceDefaultOwner interface or any additional interfaces specific to the particular arbitration policy being implemented. Each of these interfaces is explained in greater detail below:



3.1 Resource

Clients of an arbiter request access to a shared resource using the Resource interface:

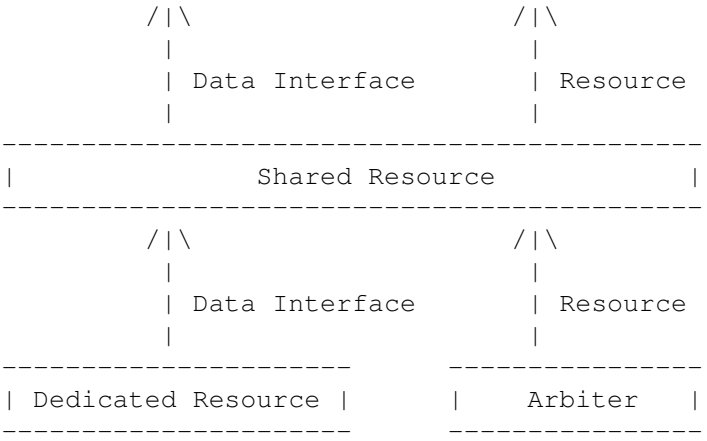
```
interface Resource {
  async command error_t request();
  async command error_t immediateRequest();
  event void granted();
  async command error_t release();
  async command bool isOwner();
}
```

A client lets an arbiter know it needs access to a resource by making a call to request(). If the resource is free, SUCCESS is returned, and a granted event is signaled back to the client. If the resource is busy, SUCCESS will still be returned, but the request will be queued according to the queuing policy of the arbiter. Whenever a client is done with the resource, it calls the release() command, and the next client in the request queue is given access to the resource and is signaled its granted() event. If a client ever makes multiple requests before receiving a granted event, an EBUSY value is returned, and the request is not queued. Using this policy, clients are not able to monolopize the resource queue by making multiple requests, but they may still be able to monopolize the use of the resource if they do not release it in a timely manner.

Clients can also request the use of a resource through the immediateRequest() command. A call to immediateRequest() can either return SUCCESS or FAIL, with requests made through this command never being queued. If a call to immediateRequest() returns SUCCESS, the client is granted access to the resource immediately after the call has returned, and no granted event is ever signaled. If it returns FAIL, the client is not granted access to the resource and the request does not get queued. The client will have to try and gain access to the resource again later.

A client can use the isOwner command of the Resource interface to check if it is the current owner of the resource. This command is mostly used to perform runtime checks to make sure that clients not owning a resource are not able to use it. If a call to isOwner fails, then no call should be made to commands provided by that resource.

The diagram below shows how a simple shared resource can be built from a dedicated resource by using just the Resource interface provided by an arbiter.:



An arbiter MUST provide exactly one parameterized Resource interface, where the parameter is a client ID, following the Service Instance pattern[3]_. An arbitrated component SomeNameP MUST #define SOME_NAME_RESOURCE to a string which can be passed to unique() to obtain a client id. This #define must be placed in a separate file because of the way nesC files are preprocessed: including the SomeNameP component isn't enough to ensure that macros #define'd in SomeNameP are visible in the referring component.

Please refer to Appendix B for an example of how to wrap a component of this type inside a generic configuration. Wrapping the component in this way ensures that each Resource client is given a unique client ID, with the added benefit of properly coupling multiple components that all need to refer to the same client ID.

Appendix B also provides a complete example of how an I2C resource might be abstracted according to this pattern. For further examples see the various chip implementations in the tinyos-2.x source tree under tinyos-2.x/chips/

3.2 ArbiterInfo

Arbiters MUST provide an instance of the ArbiterInfo interface. The ArbiterInfo interface allows a component to query the current status of an arbiter:

```
interface ArbiterInfo {
  async command bool inUse();
```

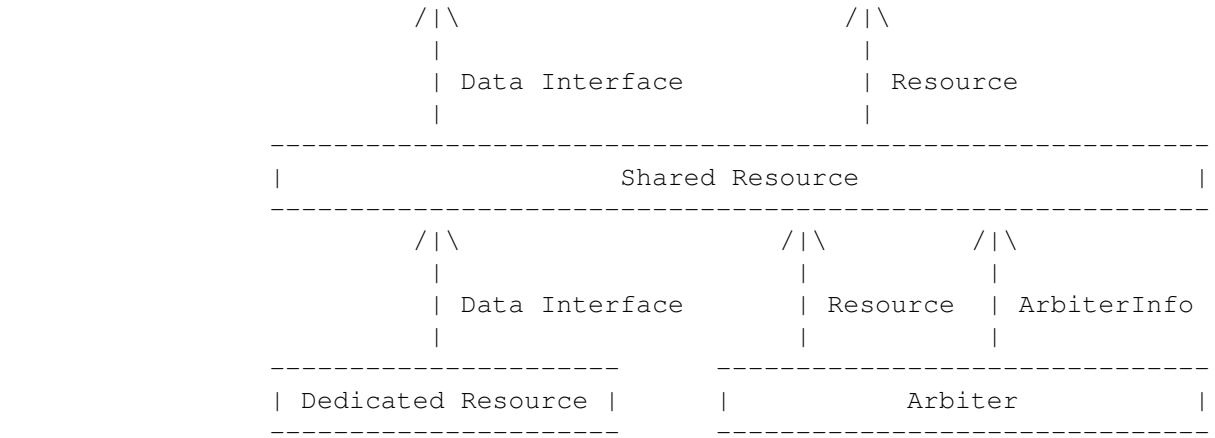
```
  async command uint8_t clientId();
}
```

In contrast to the parameterized Resource interface provided by an arbiter, only a single ArbiterInfo interface is provided. Its purpose is to allow one to find out:

- Whether the resource for which it is arbitrating use is currently in use or not
- Which client is using it.

One can view ArbiterInfo as an interface for obtaining global information about the use of a resource, while Resource can be viewed as an interface for obtaining local access to that resource.

The primary use of the ArbiterInfo interface is to allow a shared resource to reject any calls made through its data interface by clients that do not currently have access to it. For an example of how this interface is used in this fashion refer to Appendix B.:



3.3 ResourceRequested

Sometimes it is useful for a client to be able to hold onto a resource until someone else needs it and only at that time decide to release it. Using the ResourceRequested interface, this information is made available to the current owner of a resource:

```
interface ResourceRequested {
  async event void requested();
  async event void immediateRequested();
}
```

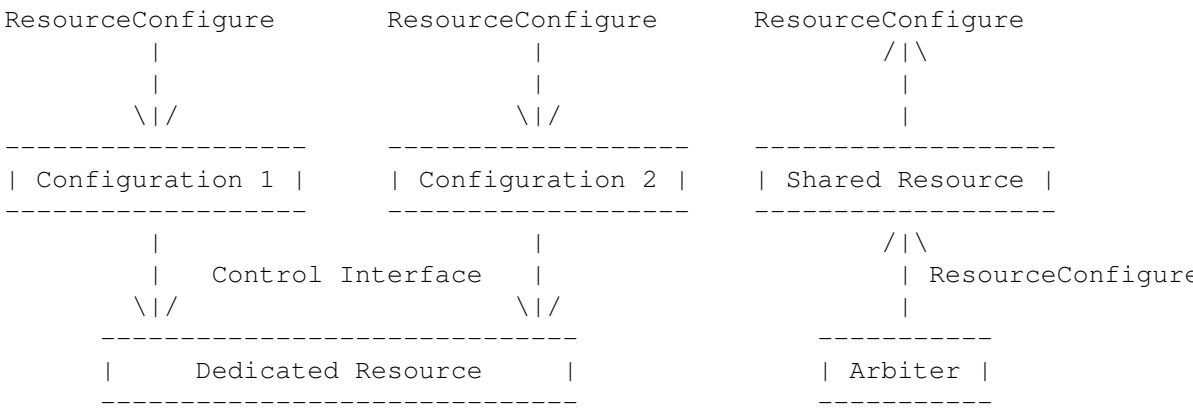
A requested event is signaled to the current owner of the resource if another client makes a request for the resource through the request() command of its Resource interface. If a request is made through the immediateRequest() command, then the immediateRequested() event is signaled.

An arbiter SHOULD provide a parameterized ResourceRequested interface to its clients, but is not required to. The client id of the parameterized ResourceRequested interface should be coupled with the client id of the Resource interface to ensure that all events are signaled to the proper clients. Please refer to Appendix B for an example of how this interface might be used.:

3.4 ResourceConfigure

The existence of the ResourceConfigure interface allows a resource to be automatically configured just before a client is granted access to it. Components providing the ResourceConfigure interface use the interfaces provided by an underlying dedicated resource to configure it into one of its desired modes of operation. A cleint then wires its shared resource abstraction to the component implementing the desired configuration. The configure command is called immediataely before the client is granted access to the resource, and the unconfigure command is called just before fully releasing it.:

```
interface ResourceConfigure {
  async command void configure();
  async command void unconfigure();
}
```



The arbiter SHOULD use a parameterized ResourceConfigure interface, with its client ID parameter coupled with the client id of its parameterized Resource interface. If an arbiter uses the ResourceConfigure interface, it MUST call ResourceConfigure.configure() on the granted client ID before it signals the Resource.granted()

event. Similarly, after a valid call to Resource.release(), it MUST call ResourceConfigure.unconfigure() on the releasing client ID. By calling ResourceConfigure.configure() just before granting a client access to a resource and calling ResourceConfigure.unconfigure() just before fully releasing it, it is guaranteed that a resource is always unconfigured before an attempt to configure it can be made again.

The commands included in this interface could have been made part of the standard Resource interface (and changed into callback events), but at a higher cost than keeping them separate. Introducing these new commands into the Resource interface would have lead to a large number of clients all including redundant configuration code, while using the call out approach to a separate component ensures that we only have a single instance of the code.

For an example of how configurations for the three different modes of the Msp430 Usart component can take advantage of the ResourceConfigure interface refer to Appendix B as well as section 4 on the use of cross-component reservation.

3.5 ResourceDefaultOwner

The normal Resource interface is for use by clients that all share the resource in an equal fashion. The ResourceDefaultOwner interface is for use by a single client that needs to be given control of the resource whenever no one else is using it. An arbiter MAY provide a single instance of the ResourceDefaultOwner interface. It MUST NOT provide more than one.:

```
interface ResourceDefaultOwner {
  async event void granted();
  async command error_t release();
  async command bool isOwner();
  async event void requested();
  async event void immediateRequested();
}
```

The Arbiter MUST guarantee that the client of the ResourceDefaulrClient interface is made the owner of the resource before the boot initialization sequence is completed. When a normal resource client makes a request for the resource, the ResourceDefaultOwner will receive either a requested() or an immediateRequested() event depending on how the request was made. It must then decide if and when to release it. Once released, all clients that have pending requests will be granted access to the resource in the order determined by the queuing policy of the arbiter in use. Once all pending requests have been granted (including those that came in while other clients had access to the resource), the ResourceDefaultOwner is automatically given control of the resource, receiving its granted() event in the process. The ResourceDefaultOwner interface also contains the same isOwner() command as the normal Resource interface, and the semantics of its use are exactly the same.

Although the ResourceDefaultOwner interface looks similar to a combination of the normal Resource interface and the ResourceRequested interface, its intended use is quite different. The ResourceDefaultOwner interface should only be used by clients that wish to have access to a resource only when no other clients are using it. They do not actively seek access to the resource, but rather use it to perform operations when it would otherwise simply be idle.

The primary motivation behind the definition of the ResourceDefaultOwner interface is to allow for an easy integration of power management for a resource with its

arbitration policy. Arbiters that want to allow a resource to be controlled by a particular power management policy can provide the ResourceDefaultOwner interface for use by a component that implements that policy. The power management component will receive the granted() event whenever the resource has gone idle, and will proceed in powering it down. When another client requests the resource, the power manager will be notified through either the requested() or immediateRequested() events as appropriate. It can then power up the resource and release it once the power up has completed. Note that if power up is a split-phase operation (takes a while), then calls by clients to immediateRequest() when in the powered down state will return FAIL. Please see the TEP on the Power Management of Non-Virtualized devices ([4]) for more details.

4. Cross-Component Reservation

In some cases, it is desirable to share the reservation of a single resource across multiple components. For example, on the TI MSP430, a single USART component can be used as an I2C bus, a UART, or an SPI connection. Clearly, on this chip, a reservation of the I2C bus implicitly restricts the corresponding UART and SPI services from gaining access to the resource. Enforcing such a policy can be accomplished in the framework described above by:

1) Creating a set of unique ids for each service using the shared resource.
2) Mapping these ids onto the ids of the underlying resource

Clients connecting to these services do not know that that this mapping is taking place. As far as they are concerned, the only arbitration taking place is between other clients using the same service. In the MSP430 example, a single client of the I2C bus could be contending with a single client of the SPI connection, but they would probably have the same service level client ID. These two service level client ids would be mapped onto 2 unique resource ids for use by the shared USART component. The proper way to achieve this mapping is through the use of generic components. The example given below shows how to perform this mapping for the SPI component on the MSP430. It is done similarly for the UART and I2C bus:

```
#include "Msp430Usart.h"
generic configuration Msp430Spi0C() {
  provides interface Resource;
  provides interface SpiByte;
  provides interface SpiPacket;
}
implementation {
  enum { CLIENT_ID = unique(MSP430_SPIO_BUS) };

  components Msp430Spi0P as SpiP;
  Resource = SpiP.Resource[ CLIENT_ID ];
  SpiByte = SpiP.SpiByte;
  SpiPacket = SpiP.SpiPacket[ CLIENT_ID ];

  components new Msp430Usart0C() as UsartC;
  SpiP.UsartResource[ CLIENT_ID ] -> UsartC.Resource;
```

```
  SpiP.UsartInterrupts -> UsartC.HplMsp430UsartInterrupts;
}
```

The definition of the MSP430_SPIO_BUS string is defined in Msp430Usart.h. A unique id is created from this string every time a new Msp430Spi0C component is instantiated. This id is used as a parameter to the parameterized Resource interface provided by the Msp430Spi0P component. This is where the mapping of the two different ids begins. As well as *providing* a parameterized Resource interface (Msp430Spi0P.Resource), the Msp430Spi0P component also *uses* a parameterized Resource interface (Msp430Spi0P.UsartResource). Whenever a client makes a call through the provided Resource interface with id CLIENT_ID, an underlying call to the Msp430Spi0P.Resource interface with the same id is implicitly made. By then wiring the Msp430Spi0P.UsartResource interface with id CLIENT_ID to an instance of the Resource interface provided by the instantiation of the Msp430Usart0C component, the mapping is complete. Any calls to the Resource interface provided by a new instantiation of the Msp430Spi0C component will now be made through a unique Resource interface on the underlying Msp430Usart0C component.

This level of indirection is necessary because it may not always be desirable to directly wire the service level Resource interface to the underlying shared Resource interface. Sometimes we may want to perform some operations between a service level command being called, and calling the underlying command on the shared resource. With such a mapping, inserting these operations is made possible.

Having such a mapping is also important for services that need to explicitly keep track of the number of clients they have, independent from how many total clients the underlying shared resource has. For example, a sensor implementation that uses an underlying ADC resource may wish to power down its sensor whenever it has no clients. It doesn't want to have to wait until the entire ADC is free to do so. Providing this mapping allows the implicit power manager components described in TEP 115 to be wired in at both levels of the abstraction without interfering with one another. In this way, implementations of these components become much simpler, and code reuse is encouraged.

Implementations of components similar to this one can be found in the tinyos-2.x source tree in the tos/chips/msp430/uart directory

5. Implementation

Because most components use one of a small number of arbitration policies, tinyos-2.x includes a number of default resource arbiters. These arbiters can be found in `tinyos-2.x/tos/system` and are all generic components that include one of the two signatures seen below:

```
generic module SimpleArbiter {
  provides interface Resource[uint8_t id];
  provides interface ResourceRequested[uint8_t id];
  provides interface ArbiterInfo;
  uses interface ResourceConfigure[uint8_t id];
}

generic module Arbiter {
  provides interface Resource[uint8_t id];
```

```
  provides interface ResourceRequested[uint8_t id];
  provides interface ResourceDefaultOwner;
  provides interface ArbiterInfo;
  uses interface ResourceConfigure[uint8_t id];
}
```

The "Simple" arbiters are intended for use by resources that do not require the additional overhead incurred by providing the ResourceDefaultOwner interface.

For many situations, changing an arbitration policy requires nothing more than changing the queuing policy it uses to decide the order in which incoming requests should be granted. In this way, separating queuing policy implementations from actual arbitration implementations encourages code reuse. The introduction of the SimpleArbiterP and ArbiterP components found under tinyos-2.x/tos/system help in this separation. They can be wired to components providing a particular queuing policy through the use of the ResourceQueue interface.:

```
interface ResourceQueue {
  async command bool isEmpty();
  async command bool isEnqueued(resource_client_id_t id);
  async command resource_client_id_t dequeue();
  async command error_t enqueue(resource_client_id_t id);
}
```

An example of wiring a First-Come-First-Serve (FCFS) queuing policy to the SimpleArbiterP component using the ResourceQueue interface defined above can be seen below:

```
generic configuration SimpleFcfsArbiterC(char resourceName[]) {
  provides {
    interface Resource[uint8_t id];
    interface ResourceRequested[uint8_t id];
    interface ArbiterInfo;
  }
  uses interface ResourceConfigure[uint8_t id];
}
implementation {
  components MainC;
  components new FcfsResourceQueueC(uniqueCount(resourceName)) as Queue;
  components new SimpleArbiterP() as Arbiter;

  MainC.SoftwareInit -> Queue;

  Resource = Arbiter;
  ResourceRequested = Arbiter;
  ArbiterInfo = Arbiter;
  ResourceConfigure = Arbiter;

  Arbiter.Queue -> Queue;
}
```

This generic configuration can be instantiated by a resource in order to grant requests made by its clients in an FCFS fashion.

All of the default queuing policies provided in tinyos-2.x along with the respective arbitration components that have been built using them are given below:

Queuing Policies:

- FcfsResourceQueueC
- RoundRobinResourceQueueC

Arbiters:

- SimpleFcfsArbiterC
- FcfsArbiterC
- SimpleRoundRobinArbiterC
- RoundRobinArbiterC

Keep in mind that neither the implementation of an arbiter nor its queuing policy can be used to explicitly restrict access to an underlying shared resource. The arbiter simply provides a standardized way of managing client ids so that shared resources don't have to duplicate this functionality themselves every time they are implemented. In order to actually restrict clients from using a resource without first requesting it, a shared resource must use the functionality provided by the ArbiterInfo interface to perform runtime checks on the current owner of a resource. Please refer to the section on the ArbiterInfo interface in Appendix B for more information on how such runtime checks can be performed.

6. Author's Address

Kevin Klues
503 Bryan Hall
Washington University
St. Louis, MO 63130

phone - +1-314-935-6355
email - klueska@cs.wustl.edu

Philip Levis
358 Gates Hall
Stanford University
Stanford, CA 94305-9030

phone - +1 650 725 9046
email - pal@cs.stanford.edu

David Gay
2150 Shattuck Ave, Suite 1300
Intel Research

Berkeley, CA 94704

phone - +1 510 495 3055
email - david.e.gay@intel.com

David Culler
627 Soda Hall
UC Berkeley
Berkeley, CA 94720

phone - +1 510 643 7572
email - culler@cs.berkeley.edu

Vlado Handziski
Sekr FT5
Einsteinufer 25
10587 Berlin
GERMANY

email - handzisk@tkn.tu-berlin.de

7. Citations

Appendix A: Resource Class Examples

Dedicated Resource

Timer 2 on the Atmega128 microprocessor is a dedicated resource represented by the HplAtm128Timer2C component:

```
module HplAtm128Timer2C {
  provides {
    interface HplTimer<uint8_t>   as Timer2     @exactlyonce();
    interface HplTimerCtrl8       as Timer2Ctrl @exactlyonce();
    interface HplCompare<uint8_t> as Compare2   @exactlyonce();
  }
}
```

Only a single client can wire to any of these interfaces as enforced through the nesC @exactlyonce attribute. Keep in mind that although the interfaces of this component

[1] TEP 2: Hardware Abstraction Architecture.

[2] TEP 102: Timers.

[3] Service Instance Pattern. In *Software Design Patterns for TinyOS.* David Gay, Philip Levis, and David Culler. Published in Proceedings of the ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05).

[4] TEP 115: Power Management of Non-Virtualized Devices.

[5] TinyOS Programming. http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf

are only allowed to be wired to once, nothing prevents the component wiring to them from virtualizing the services they provide at some higher level. If you are unfamiliar with how @exactlyonce and other nesC attributes are used to by the nesC compiler, please refer to section 9.1 of the TinyOS Programming Manual[5].

Virtualized Resource

The TimerMilliC component provides a virtual abstraction of millisecond precision timers to application components[2]. It encapsulates the required parameterized Timer interface through the use of a generic configuration. Clients wishing to use a millisecond timer need only instantiate a single instance of the TimerMilliC generic, leaving the fact that it is virtualized underneath transparent.:

```
generic configuration TimerMilliC {
  provides interface Timer<TMilli>;
}
implementation {
  components TimerMilliP;
  Timer = TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];
}
```

The actual parameterized Timer interface is provided by the chip specific HilTimerMilliC component. This interface is exposed through the TimerMilliP component which wires HilTimerMilliC to the boot initialization sequence:

```
configuration TimerMilliP {
  provides interface Timer<TMilli> as TimerMilli[uint8_t num];
}
implementation {
  components HilTimerMilliC, MainC;
  MainC.SoftwareInit -> HilTimerMilliC;
  TimerMilli = HilTimerMilliC;
}
```

Appendix B: Arbiter Interface Examples

Resource

Examples of how to use the Resource interface for arbitrating between multiple clients can be found in the tinyos-2.x source tree under tinyos-2.x/apps/tests/TestArbiter.

A specific example of where the Resource.isOwner() is used can be seen in the HplTda5250DataP component of the Infineon Tda5250 radio implementation:

```
async command error_t HplTda5250Data.tx(uint8_t data) {
  if(call UartResource.isOwner() == FALSE)
    return FAIL;
  call Usart.tx(data);
  return SUCCESS;
}
```

A call to the HplTda5250Data.tx command will fail if the radio does not currently have control of the underlying Usart resource. If it does, then the Usart.tx(data) command is called as requested.

A component using the Resource interface to implement an I2C service might look like this:

```
#include I2CPacket.h
configuration I2CPacketP {
  provides interface Resource[uint8_t client];
  provides interface I2CPacket<I2CAddrSize>[uint8_t client];
}
implementation {
  components new FcfsArbiterC(I2CPACKET_RESOURCE) as Arbiter;
  components I2CPacketImplP() as I2C;
  ...

  Resource  = Arbiter;
  I2CPacket = I2C;
  ...
}
```

where I2CPacketImplP contains the actual implementation of the I2C service, and I2CPacket.h contains the #define for the name of the resource required by the arbiter:

```
#ifndef I2CPACKETC_H
#define I2CPACKETC_H
#define I2CPACKET_RESOURCE "I2CPacket.Resource"
#endif
```

This service would then be made available to a client through the generic configuration seen below:

```
#include I2CPacket.h
generic configuration I2CPacketC {
  provides interface Resource;
  provides interface I2CPacket<I2CAddrSize>;
}
implementation {
  enum { CLIENT_ID = unique(I2CPACKET_RESOURCE) };

  components I2CPacketP as I2C;
  Resource = I2C.Resource[CLIENT_ID];
  I2CPacket = I2C.I2CPacket[CLIENT_ID];
}
```

In this example, an instance of the I2CPacket interface is coupled with an instance of the Resource interface on every new instantiation of the I2CPacketC component. In this way, a single Resource and a single I2CPacket interface can be exported by this component together for use by a client.

Clients of the I2C service would use it as follows:

```
module I2CClientP {
```

```
  uses interface Resource as I2CResource;
  uses interface I2CPacket<I2CAddrSize>;
} ...

configuration I2CClientC { }
implementation {
  components I2CClientP, new I2CPacketC();

  I2CClientP.I2CResource -> I2CPacketC.Resource;
  I2CUserM.I2CPacket -> I2CPacket.I2CPacket;
}
```

ArbiterInfo

In the implementation of the ADC component on the Msp430 microcontroller, a simple arbiter is used to provide a round robin sharing policy between clients of the ADC:

```
configuration Msp430Adc12C {
  provides interface Resource[uint8_t id];
  provides interface Msp430Adc12SingleChannel[uint8_t id];
  provides interface Msp430Adc12FastSingleChannel[uint8_t id];
}
implementation {
  components Msp430Adc12P,MainC,
             new SimpleRoundRobinArbiterC(MSP430ADC12_RESOURCE) as Arbiter,
             ...

  Resource = Arbiter;
  SingleChannel = Msp430Adc12P.SingleChannel;
  FastSingleChannel = Msp430Adc12P.FastSingleChannel;

  Msp430Adc12P.Init <- MainC;
  Msp430Adc12P.ADCArbiterInfo -> Arbiter;
  ...
}
```

In this configuration we see that the Resource interface provided by Msp430Adc12C is wired directly to the instance of the SimpleRoundRobinArbiterC component that is created. The ArbiterInfo interface provided by SimpleRoundRobinArbiterC is then wired to Msp430Adc12P. The Msp430Adc12P component then uses this interface to perform run time checks to ensure that only the client that currently has access to the ADC resource is able to use it:

```
async command error_t Msp430Adc12SingleChannel.getSingleData[uint8_t id]()
  if (call ADCArbiterInfo.clientId() == id){
    configureChannel()
    // Start getting data
  }
  else return ERESERVE;
}
```

```
async command error_t Msp430Adc12FastSingleChannel.configure[uint8_t id]()
  if (call ADCArbiterInfo.clientId() == id){
    clientID = id
    configureChannel()
  }
  else return ERESERVE;
}

async command error_t Msp430Adc12FastSingleChannel.getSingleData[uint8_t id
{
  if (clientID = id)
    // Start getting data
  else return ERESERVE;
}
```

In order for these runtime checks to succeed, users of the Msp430Adc12SingleChannel and Msp430Adc12FastSingleChannel interfaces will have to match their client id's with the client id of a corresponding Resource interface. This can be done in the following way:

```
generic configuration Msp430Adc12ClientC() {
  provides interface Resource;
  provides interface Msp430Adc12SingleChannel;
}
implementation {
  components Msp430Adc12C;
  enum { ID = unique(MSP430ADC12_RESOURCE) };

  Resource = Msp430Adc12C.Resource[ID];
  Msp430Adc12SingleChannel = Msp430Adc12C.SingleChannel[ID];
}

generic configuration Msp430Adc12FastClientC() {
  provides interface Resource;
  provides interface Msp430Adc12FastSingleChannel;
}
implementation {
  components Msp430Adc12C;
  enum { ID = unique(MSP430ADC12_RESOURCE) };

  Resource = Msp430Adc12C.Resource[ID];
  Msp430Adc12FastSingleChannel = Msp430Adc12C.SingleChannel[ID];
}
```

Since these are generic components, clients simply need to instantiate them in order to get access to a single Resource interface that is already properly coupled with a Msp430Adc12SingleChannel or Msp430Adc12FastSingleChannel interface.

Take a look in the tinyos-2.x source tree under tinyos-2.x/tos/chips/adc12 to see the full implementation of these components in their original context.

ResourceRequested

On the eyesIFXv2 platform, both the radio and the flash need access to the bus provided by the Usart0 component on the Msp430 microcontroller. Using a simple cooperative arbitration policy, these two components should able to share the Usart resource by only holding on to it as long as they need it and then releasing it for use by the other component. In the case of the MAC implementation of the Tda5250 radio component, however, the Msp430 Usart resource needs be held onto indefinitely. It only ever considers releasing the resource if a request from the flash component comes in through its ResourceRequested interface. If it cannot release it right away (i.e. it is in the middle of receiving a packet), it defers the release until some later point in time. Once it is ready to release the resource, it releases it, but then immediately requests it again. The flash is then able to do what it wants with the Usart, with the radio regaining control soon thereafter.

In the CsmaMacP implementation of the Tda5250 radio we see the ResourceRequested event being implemented:

```
async event void ResourceRequested.requested() {
  atomic {
    /* This gives other devices the chance to get the Resource
       because RxMode implies a new arbitration round.  */
    if (macState == RX) call Tda5250Control.RxMode()();
  }
}
```

Through several levels of wiring, the Tda5250 interface is provided to this component by the Tda5250RadioP component:

```
module Tda5250RadioP {
  provides interface Tda5250Control;
  uses interface Resource as UsartResource;
  ...
}
implementation {
  async command error_t Tda5250Control.RxMode() {
    if(radioBusy() == FALSE)
      call UsartResource.release();
      call UsartResource.request();
  }
  event void UsartResource.granted() {
    // Finish setting to RX Mode
  }
  ...
}
```

Although the full implementation of these components is not provided, the functionality they exhibit should be clear. The CsmaMacP component receives the ResourceRequested.requested() event when the flash requests the use of the Msp430 Usart0 resource. If it is already in the receive state, it tries to reset the receive state through a call to a lower level component. This component checks to see if the radio is in the middle of doing anything (i.e. the radioBusy() == FALSE check), and if not, releases the resource and then requests it again.

To see the full implementations of these components and their wirings to one another, please refer to the tinyos-2.x source tree under tinyos-2.x/tos/chips/tda5250, tinyos-2.x/tos/chips/tda5250/mac, tinyos-2.x/tos/platforms/eyesIFX/chips/tda5250, and tinyos-2.x/tos/platforms/eyesIFX/chips/msp430.

Resource Configure

The Msp430 Usart0 bus can operate in three modes: SPI, I2C, and UART. Using all three concurrently is problematic: only one should be enabled at any given time. However, different clients of the bus might require the bus to be configured for different protocols. On Telos, for example many of the available sensors use an I2C bus, while the radio and flash chip use SPI.

A component providing the SPI service on top of the shared Usart component looks like this:

```
generic configuration Msp430Spi0C() {
  provides interface Resource;
  provides interface SpiByte;
  provides interface SpiPacket;
  ...
}
implementation {
  enum { CLIENT_ID = unique( MSP430_SPIO_BUS ) };

  components Msp430SpiNoDma0P as SpiP;
  components new Msp430Usart0C() as UsartC;
  SpiP.ResourceConfigure[ CLIENT_ID ] <- UsartC.ResourceConfigure;
  SpiP.UsartResource[ CLIENT_ID ] -> UsartC.Resource;
  SpiP.UsartInterrupts -> UsartC.HplMsp430UsartInterrupts;
  ...
}
```

And one providing the I2C service looks like this:

```
generic configuration Msp430I2CC() {
  provides interface Resource;
  provides interface I2CPacket<TI2CBasicAddr> as I2CBasicAddr;
   ...
}
implementation {
  enum { CLIENT_ID = unique( MSP430_I2CO_BUS ) };

   components Msp430I2C0P as I2CP;
   components new Msp430Usart0C() as UsartC;
   I2CP.ResourceConfigure[ CLIENT_ID ] <- UsartC.ResourceConfigure;
   I2CP.UsartResource[ CLIENT_ID ] -> UsartC.Resource;
   I2CP.I2CInterrupts -> UsartC.HplMsp430UsartInterrupts;
   ...
}
```

The implementation of the ResourceConfigure interface is provided by both the Msp430SpiNoDma0P and the Msp430I2C0P. In the two different components, the

same Msp430Usart0C component is used, but wired to the proper implementation of the ResourceConfigure interface. In this way, different instances of the Msp430Usart0C can each have different configurations associated with them, but still provide the same functionality.

Take a look in the tinyos-2.x source tree under tinyos-2.x/tos/chips/msp430/usart to see the full implementation of these components along with the corresponding Uart implementation.