Power Management of Non-Virtualised Devices

TEP:	115
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x

Author:	Kevin Klues, Vlado Handziski, Jan-Hinrich Hauer, Phil Levis

Abstract

This memo documents how TinyOS 2.x manages the power state of physical (not virtualised) abstractions.

1. Introduction

TinyOS platforms have limited energy. A unified power management strategy for all devices and peripherals is not feasible, as they vary significantly in warm-up times, power profiles, and operation latencies. While some devices, such as microcontrollers, can efficiently calculate their lowest possible power state very quickly, others, such as sensors with warm-up times, require external knowledge to do so.

In TinyOS 1.x, applications are responsible for all power management. Low-level subsystems, such as an SPI bus, are explicitly powered on and off by higher level abstractions. This approach of deep calls to `StdControl.start()` and `StdControl.stop()` introduces strange behaviors and can get in the way of power conservation. Turning off the radio on the Telos platform, for example, turns off the SPI bus and therefore prevents the flash driver from working. Additionally, the microcontroller stays in a higher power state for the SPI bus even when it is inactive.

TinyOS 2.x defines two classes of devices for power-management: microcontrollers and peripherals. TEP 112 documents how TinyOS 2.x manages the power state of a microcontroller [TEP112]. Unlike microcontrollers, which typically have several power states, peripheral devices typically have two distinct states, *on* and *off*. This TEP is dedicated to documenting how TinyOS 2.x controls the power state of peripheral devices.

The term “peripheral device” refers to any hardware device which arbitrates access with the mechanisms described in [TEP108] . These devices are not virtualised in the sense that access to them must be explicitly requested and released by their users.

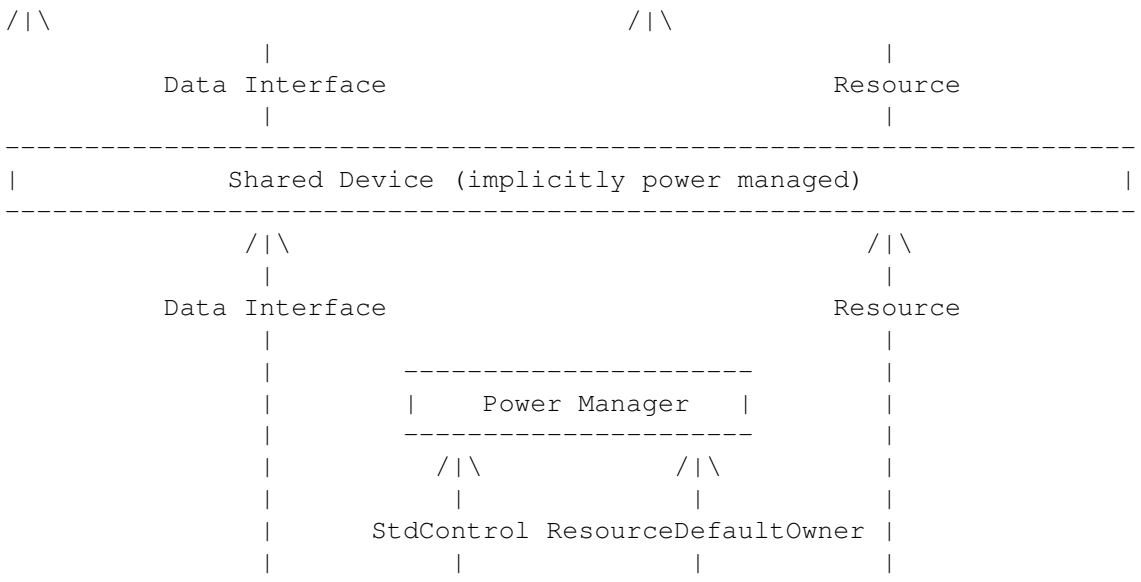
2. Power Management Models

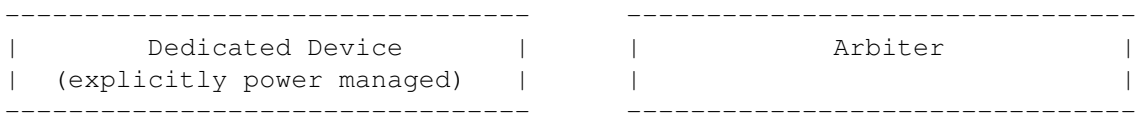
There are two different models to managing the power state of a peripheral in TinyOS: *explicit power management* and *implicit power management*.

The explicit model provides a means for a single client to manually control the power state of a dedicated physical device (as defined by [TEP108]). Whenever this client tells the device to power up or down it does so without delay (except for delays in the hardware of course). This model can be particularly useful when the control information driving the selection of the proper power state of a device relies on external logic contained in higher level components. The following section discusses the `StdControl`, `SplitControl`, and `AsyncStdControl` interfaces used to perform power management of this type.

The implicit model, on the other hand, provides a means for allowing the power state of a device to be controlled from within the driver for that device itself. Device drivers following this model are never explicitly powered up or down by some external client, but rather *require* some internal policy to be defined that decides exactly when their power states should be changed. This policy could exist natively on the hardware of the physical device itself, or be implemented on top of some lower level abstraction of a physical device adhering to the *explicit power management* model.

Shared devices (as defined by [TEP108]) can infer whether they should be on or off based on the interfaces they provide to their clients. For example, when a client requests the ADC, this implies the ADC should be on; if there are no requests of the ADC, this implies it should be off. Therefore shared devices do not need to provide a power control interface. They can use an implicit power management policy. Section 4.2 discusses this in more detail.:





3. Explicit Power Management

Just as in TinyOS 1.x, TinyOS 2.x has `StdControl` and `SplitControl` interfaces in order to control the on and off power states of explicitly managed peripherals. TinyOS 2.x also introduces a third interface, `AsyncStdControl`. A component representing a hardware device that can be powered on and off MUST provide one of these three interfaces. The selection of the right interface depends on the latencies involved in changing between these two states as well as the nature of the code (sync or async) executing any of the interface's commands.

3.1 Power Management with `StdControl`

Whenever the powerup and powerdown times of a non-virtualised device are negligible, they SHOULD provide the `StdControl` interface as defined below:

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

> **Note**
>
> Powerup and powerdown times on the order of a few microseconds have traditionally been considered negligible, and can be waited on using one of the *BusyWait* interfaces described in [TEP102]. Powerup and powerdown times on the order of a few milliseconds, however, should not be ignored, and MUST be hidden behind the use of the `SplitControl` interface described later on in this section. A general rule of thumb is that if waiting for powerup takes more than one hundred microseconds, SplitControl is probably more suitable.

An external component MUST call `StdControl.start()` to power a device on and `StdControl.stop()` to power a device off. Calls to either command MUST return either SUCCESS or FAIL.

Upon the successful return of a call to `StdControl.start()`, a device MUST be completely powered on, allowing calls to commands of other interfaces implemented by the device to succeed.

Upon the successful return of a call to `StdControl.stop()`, a device MUST be completely powered down, and any calls to commands of other interfaces implemented by that device that actually access the device hardware MUST return FAIL or EOFF.

If a device is not able to complete the `StdControl.start()` or `StdControl.stop()` request for any reason, it MUST return FAIL.

Based on these specifications, the following matrix has been created to describe the valid return values for any call made through the `StdControl` interface in one of the devices valid power states:

Call	Device On	Device Off
StdControl.start()	SUCCESS	SUCCESS or FAIL
StdControl.stop()	SUCCESS or FAIL	SUCCESS
operation	depends	FAIL or EOFF

Devices providing this interface would do so as shown below:

```
configuration DeviceC {
  provides {
    interface Init;
    interface StdControl;  //For Power Management
    ....
  }
}
```

3.2 Power Management with `SplitControl`

When a device's powerup and powerdown times are non-negligible, the *"SplitControl"* interface SHOULD be used in place of the *"StdControl"* interface. The definition of this interface can be seen below:

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

An external component MUST call `SplitControl.start()` to power a device on and `SplitControl.stop()` to power a device off. Calls to either command return one of SUCCESS, FAIL, EBUSY, or EALREADY. SUCCESS indicates that the device has now started changing its power state and will signal a corresponding completion event in the future. EBUSY indicates that the device is in the midst of either starting or stopping (e.g., it is starting when stop is called or stopping when start is called) and will not issue an event. EALREADY indicates that the device is already in that state; the call is erroneous and a completion event will not be signaled. FAIL indicates that the device's power state could not be changed. More explicitly:

Successful calls to `SplitControl.start()` MUST signal one of `SplitControl.startDone(SUCCESS)` or `SplitControl.startDone(FAIL)`.

Successful calls to `SplitControl.stop()` MUST signal one of `SplitControl.stopDone(SUCCESS)` or `SplitControl.stopDone(FAIL)`.

Upon signaling a `SplitControl.startDone(SUCCESS)`, a device MUST be completely powered on, and operation requests through calls to commands of other interfaces implemented by the device MAY succeed.

Upon signalling a `SplitControl.stopDone(SUCCESS)`, a device MUST be completely powered down, and any subsequent calls to commands of other interfaces implemented by the device that actually access the device hardware MUST return EOFF or FAIL.

If a device is powered on and a successful call to `SplitControl.stop()` signals a `SplitControl.stopDone(FAIL)`, the device MUST still be fully powered on, and operation requests through calls to commands of other interfaces implemented by the device MAY succeed.

If a device is powered down and a successful call to `SplitControl.start()` signals a `SplitControl.startDone(FAIL)`, the device MUST still be fully powered down, and operation requests through calls to commands of other interfaces implemented by the device MUST return EOFF or FAIL.

If a device is not able to complete the `SplitControl.start()` or `SplitControl.stop()` requests they MUST return FAIL.

Calls to either `SplitControl.start()` when the device is starting or `SplitControl.stop()` while the device is stopping MUST return SUCCESS, with the anticipation that a corresponding `SplitControl.startDone()` or `SplitControl.stopDone()` will be signaled in the future.

Calls to `SplitControl.start()` when the device is started or `SplitControl.stop()` while the device is stopped MUST return EALREADY, indicating that the device is already in that state. The corresponding completion event (startDone for start or stopDone for stop) MUST NOT be signaled.

Calls to `SplitControl.start()` when the device is stopping or `SplitControl.stop()` while the device is starting MUST return EBUSY, indicating that the device is busy performing a differnet operation. The correspodning completion event (startDone for start or stopDone for stop) MUST NOT be signaled.

Call	**Device On**	**Device Off**	**Starting**	**Stopping**
SplitControl.start()	EALREADY	SUCCESS FAIL	SUCCESS	EBUSY
SplitControl.stop()	SUCCESS FAIL	EALREADY	EBUSY	SUCCESS
operation	depends	FAIL EOFF EOFF	FAIL EOFF SUCCESS	FAIL EOFF

Devices providing this interface would do so as shown below:

```
configuration DeviceC {
  provides {
    interface Init;
    interface SplitControl; \\ For Power Management
    ....
  }
}
```

Note

Other approaches were considered for the return values of `SplitControl.start()` and `SplitControl.stop()`. One such approach would have replaced EBUSY with SUCCESS when `SplitControl.start()` was called while in the process of stopping and `SplitControl.stop()` was called while in the process of starting. However, implementing such an approach adds unwanted complexity to a device driver. It is unreasonable to expect the implementor of each driver to implement this functionality.
Returning EBUSY is the most straightforward, unambiguous value that can be returned in such a situation. By returning EBUSY when a device is in a transitional state, the components built on top of a driver unambiguously know exactly why a call to `start()` or `stop()` did not succeed, and can take action accordingly. Since only ONE component should ever implement the `SplitControl` interface for a given device, it isn't unreasonable to expect them to keep track of this return value themselves. There is, of course, nothing preventing someone from creating a component on top of each driver implementation that implements things differently.

3.3 Power Management with `AsyncStdControl`

The commands and the events of the *"StdControl"* and the *"SplitControl"* interfaces are synchronous and can not be called from within asynchronous code (such as interrupt service routines, etc.). For the cases when the power state of the device needs to be controlled from within asynchronous code, the *"AsyncStdControl"* interface MUST be used in place of the *"StdControl"* interface. The definition of this interface can be seen below:

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

All of the semantics that hold true for devices providing the `StdControl` interface also hold for this interface.

Devices providing this interface would do so as shown below:

```
configuration DeviceC {
  provides {
    interface Init;
    interface AsyncStdControl; \\ For Power Management
    ....
  }
}
```

Note

The `AsyncStdControl` interface should be provided whenever it might be necessary to allow a device to be powered on or off while running in async context. If it is anticipated that a device *will* not (or more likely *should* not) be powered on or off while in asynchronous context, then the `StdControl` interface SHOULD be provided instead. Components that wish to power the device on or off from within async context would then be required to post a task before doing so. In practice, `AsyncStdControl` is provided by low-level hardware resources, while `StdControl` is provided by higher level services built on top of these resources.

4. Implicit Power Management

While explicit power management provides the mechanism for changing power states, it does not specify a policy. This does not represent a large problem for the simple case of *dedicated* devices, but can become crucial for non-trivial cases involving complex interdependencies between devices controlled by multiple clients.

For example, if component *A* is a client of both component *B* and component *C*, what happens with *B* and *C* if `StdControl.stop()` is called on component *A* ? Should components *B* and *C* also be stopped? What about the reverse case where both *B* and *C* are clients of the single shared component, *A*? If devices *B* and *C* are shut off, should *A* be shut off as well? How can one decide when it is appropriate to cascade such powerup and powerdown requests?

The complex nature of the problem is evident from the number of unexpected behaviors in TinyOS 1.x involving `StdControl`. On several platforms, one of the SPI buses is shared between the radio and the flash device. On some of them, issuing `StdControl.stop()` on the radio results in a series of cascaded calls that result in SPI bus becoming disabled, rendering the communication with the flash impossible. Of course, the right policy would involve tracking the clients of the SPI bus and powering it off only once both the radio and the flash devices were no longer using it. Conversely, the SPI bus should be powered on whenever there is at least one active client.

The selection of the right power management policy to use is a complex task that depends on the nature of the devices in use, their interdependency, as well as on any specific application requirements. For cases when some of these features are known a-priori or are restricted in some sense, it is preferable that the system provide architectural support for enforcing a meaningful *default* power-management policy instead of passing that task on to the application programmer to be solved on a case-by-case basis. The following section discusses these power management policies and the components that implement them in greater detail.

4.1. Power Management Policies

Just as generic arbiters are offered in TinyOS 2.x to provide the arbitration functionality required by shared resources, generic power management policies are also offered to allow the power management of non-virtualised devices to be automatically controlled.

Through the use of the arbiter components described in [TEP108], device drivers implemented as shared resources provide the type of restricted resource interdependency where default power-management policies can be offered. The shared resource

class defined in Section 2.3 of [TEP108], provides a well defined component interdependency, where a single resource is shared among multiple clients. This relationship enables the definition of default power-management policies that can be used to automatically power the resource on and off.

The *Power Manager* component implementing one of these polices acts as the *default owner* of the shared resource device and interacts with it through the use of the ResourceDefaultOwner interface:

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

Acting as the default owner, the *Power Manager* waits for the `ResourceDefaultOwner.granted()` event to be signaled in order to gain ownership over the resource device.

Once it owns the device, the *Power Manager* is free to execute its power-management policy using the StdControl-like interface provided by the underlying resource. Different power managers can implement different policies. In the simplest case, this would involve an immediate power-down via one of the `stop()` commands. When the power-state transition involves non-negligible costs in terms of wake-up latency or power consumption, the *PowerManager* might revert to a more intelligent strategy that tries to reduce these effects. As pointed out in the introduction, one such strategy might involve the use of a timer to defer the power-down of the resource to some later point in time, giving any resource clients a window of opportunity to put in requests before the device is fully shut down.

Regardless of the power management policy in use, the *Power Manager* remains owner of the resource as long as the resource is not requested by one of its clients. Whenever a client puts in a request, the *Power Manager* will receive a `ResourceDefaultOwner.requested()` event (or `immediateRequested()` event) from the arbiter it is associated with. Upon receiving this event, the *Power Manager* MUST power up the resource through the StdControl-like interface provided by the lower level abstraction of the physical device. The *Power Manager* MUST release the ownership of the resource (using the `ResourceDefaultOwner.release()` command) and MUST wait until after the resource has been fully powered on before doing so.

Modeling devices as shared resources and allowing them to be controlled in the way described here, solves the problems outlined in section 3 regarding how to keep track of when and how the powerdown of nested resources should proceed. The *Power Manager* component answers the question of how, and the combination of the power management policy being used and the reception of the `ResourceDefaultOwner.granted()` and `ResourceDefaultOwner.requested()` events answers the question of when. As long as the resource at the bottom of a large set of nested resource clients has been fully released, the power mananger will be able to power down the resource appropriately.

Using the model described above, a resource that uses one of these policies according to the *implicitly power management* model could be built as shown below:

```
module MyFlashP {
  provides {
```

```
    interface Init;
    interface SplitControl;
    interface Resource;
    interface FlashCommands;
    ...
  }
}
implementation {
  ...
}

generic module PowerManagerC(uint8_t POWERDOWN_DELAY) {
  provides {
    interface Init;
  }
  uses {
    interface SplitControl;
    interface ResourceDefaultOwner;
  }
}
implementation {
  ...
}

#define MYFLASH_RESOURCE "MyFlash.resource"
#define MYFLASH_POWERDOWN_DELAY       1000
configuration MyFlashC {
  provides {
    interface Init;
    interface Resource;
    interface FlashCommands;
  }
}
implementation {
  components new PowerManagerC(MYFLASH_POWERDOWN_DELAY)
  , FcfsArbiter(MYFLASH_RESOURCE)
  , MyFlashP;

  Init = MyFlashP;
  Resource = FcfsArbiter;
  FlashCommands = MyFlashP;

  PowerManagerC.ResourceDefaultUser -> FcfsArbiter;
  PowerManagerC.SplitControl -> MyFlashP;

}
```

This example implementation is built out of three components. The first component (`MyFlashP`) follows the *explicit power management* model for defining the interfaces to the physical flash device. The second component (`PowerManagerC`)

is the generic *Power Manager* component that will be used to implement the specific power management policy for this device. The third component (`MyFlashC`) is the configuration file that wires together all of the components required by the implementation of of the device as it adheres to the *implicit power management* model. It includes the `MyflashP` and `PowerManagerC` components, as well as an arbiter component for managing shared clients of the device. Notice how the *Power Manager* is wired to both the `ResourceDefaultUser` interface provided by the arbiter, and the `SplitControl` interface provided by the flash. All clients of this flash device are directly connected to the resource interface provided by the arbiter. As outlined above, the `PowerManagerC` component will use the events signaled through the `ResourceDefaultUser` interface to determine when to make calls to power the device up and power it down through the `SplitControl` interface.

4.2. Example Power Managers: PowerManagerC and DeferredPowerManagerC

TinyOS 2.x currently has two default power management policies that it provides. These policies are implemented by the various components located under tinyos-2.x/lib/power. The first policy is implemented using an *immediate* power control scheme, whereby devices are powered on and off immediately after they have been requested and released. The second policy is implemented using a *deferred* power control scheme, whereby devices are powered on immediately after being requested, but powered off after some small delay from being released. This delay is configurable to meet the varying needs of different device drivers.

Each policy has three different implementations for use by each of the `StdControl`, `SplitControl`, and `AsyncStdControl` interfaces.

For reference, each of the available components are listed below

Immediate Power Management:

- `StdControlPowerManagerC`
- `SplitControlPowerManagerC`
- `AsyncStdControlPowerManagerC`

Deferred Power Management:

- `StdControlDeferredPowerManagerC`
- `SplitControlDeferredPowerManagerC`
- `AsyncStdControlDeferredPowerManagerC`

5. Author's Address

Kevin Klues
444 Gates Hall
Stanford University
Stanford, CA 94305-9030

email - klueska@cs.stanford.edu

Vlado Handziski
Sekr FT5
Einsteinufer 25
10587 Berlin
GERMANY

phone - +49 30 314 23831
email - handzisk@tkn.tu-berlin.de

Jan-Hinrich Hauer
Sekr FT5
Einsteinufer 25
10587 Berlin
GERMANY

phone - +49 30 314 23813
email - hauer@tkn.tu-berlin.de

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

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

6. Citations

[TEP102] TEP 102: Timers.

[TEP108] TEP 108: Resource Arbitration.

[TEP112] TEP 112: Microcontroller Power Management.