Writing Low-Power Applications

From TinyOS Wiki

This lesson demonstrates how to write low power sensing applications in TinyOS. At any given moment, the power consumption of a wireless sensor node is a function of its microcontroller power state, whether the radio, flash, and sensor peripherals are on, and what operations active peripherals are performing. This tutorial shows you how to best utilize the features provided by TinyOS to keep the power consumption of applications that use these devices to a minimum.

Contents

Overview

Energy management is a critical concern in wireless sensor networks. Without it, the lifetime of a wireless sensor node is limited to just a few short weeks or even days, depending on the type of application it is running and the size of batteries it is using. With proper energy management, the same node, running the same application, can be made to last for months or years on the same set of batteries.

Ideally, one would want all energy management to be handled transparently by the operating system, relieving application developers from having to deal with this burden themselves. While attempts have been made in the past to provide such capabilities, most operating systems today still just provide the necessary primitives for performing energy management -- the logic of when and how to do so is left completely up to the application.

TinyOS provides a novel method of performing energy management directly within the OS. The method it uses is as efficient as it is elegant. Developers no longer have to struggle with code that explicitly manages the power state for any of the peripheral devices that it uses. To get a better idea of the complexity involved in doing something as simple as periodically taking a set of sensor readings and logging them to flash in the most energy efficient manner possible, take a look at the pseudo code found below.

```
Every Sample Period:
  Turn on SPI bus
  Turn on flash chip
  Turn on voltage reference
  Turn on I2C bus
  Log prior readings
  Start humidity sample
  Wait 5ms for log
  Turn off flash chip
  Turn off SPI bus
  Wait 12ms for vref
  Turn on ADC
  Start total solar sample
  Wait 2ms for total solar
  Start photo active sample
  Wait 2ms for photo active
  Turn off ADC
  Turn off vref
  Wait 34ms for humidity
  Start temperature sample
  Wait 220ms for temperature
  Turn off I2C bus
```

With the methods provided by TinyOS, hand-tuned application code that looks like that can be transformed into this:

```
Every Sample Period:
  Log prior readings
  Sample photo active
```

```
 Sample total solar
 Sample temperature
 Sample humidity
```

The pseudo code shown above is for concurrently sampling all of the the onboard sensors found on the latest revision of the tmote sky sensor nodes. Experiments have shown that using the methods provided by TinyOS, this application comes within 1.6% of the energy efficiency of the hand-tuned implementation -- even at sampling rates as fast as 1 sample per second. For more information on these experiments and the method TinyOS uses to actually manage energy so efficiently, please refer to the paper found here. (http://www.klueska.com/pubs/klues07sosp.pdf)

The rest of this tutorial is dedicated to teaching you how to write applications that allow TinyOS to manage energy for you in the most efficient manner possible. As a rule, TinyOS can manage energy more efficiently when I/O requests are submitted by an application in parallel. Submitting them serially may result in energy wasted by unnecessarily turning devices on and off more frequently than required.

Peripheral Energy Management

Compare the following two code snippets:

Parallel

```
event void Boot.booted() {
  call Timer.startPeriodic(SAMPLE_PERIOD);
}

event void Timer.fired() {
  call LogWrite.append(current_entry, sizeof(log_entry_t));

  current_entry_num = !current_entry_num;
  current_entry = &(entry[current_entry_num]);
  current_entry->entry_num = ++log_entry_num;

  call Humidity.read();
  call Temperature.read();
  call Photo.read();
  call Radiation.read();
}

event void Humidity.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
   current_entry->hum = val;
  }
  else current_entry->hum  = 0xFFFF;
}

event void Temperature.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
    current_entry->temp = val;
  }
  else current_entry->temp = 0xFFFF;
}

event void Photo.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
    current_entry->photo = val;
  }
  else current_entry->photo = 0xFFFF;
 }

event void Radiation.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
    current_entry->rad = val;
  }
  else current_entry->rad = 0xFFFF;
}

event void LogWrite.appendDone(void* buf,
                               storage_len_t len,
                               bool recordsLost,
                               error_t error) {
  if (error == SUCCESS)
    call Leds.led2Toggle();
}
```

Serial

```
event void Boot.booted() {
  call Timer.startPeriodic(SAMPLE_PERIOD);
}

event void Timer.fired() {
  call Humidity.read();
}

event void Humidity.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad = 0xFFFF;
  call Temperature.read();
}

event void Temperature.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad = 0xFFFF;
  call Photo.read();
}

event void Photo.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad = 0xFFFF;
  call Radiation.read();
}

event void Radiation.readDone(error_t result, uint16_t val) {
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad = 0xFFFF;
  call LogWrite.append(entry, sizeof(log_entry_t));
}

event void LogWrite.appendDone(void* buf,
                               storage_len_t len,
                               bool recordsLost,
                               error_t error) {
  if (error == SUCCESS)
    call Leds.led2Toggle();
}
```

In the parallel case, logging to flash and requesting samples from each sensor is all done within the body of the `Timer.fired()` event. In the serial case, a chain of events is triggered by first calling `Humidity.read()` in `Timer.fired()`, sampling each subsequent sensor in the body of the previous `readDone()` event, and ending with all sensor readings being logged to flash.

By logging to flash and sampling all sensors within the body of a single event, the OS has the opportunity to schedule each operation as it sees fit. Performing each operation after the completion of the previous one gives the OS no such opportunity. The only downside of the parallel approach is that the application must manually manage a double buffer so that the values written to flash are not overwritten before they are logged. To save the most energy, however, the parallel version should always be used. Keep in mind that in both cases, the developer must also make sure that the sampling interval is longer than the time it takes to gather all sensor readings. If it is not, data corruption will inevitably occur.

Radio Power Management

By default, TinyOS provides low power radio operation through a technique known as *Low-Power Listening*. In low-power listening, a node turns on its radio just long enough to detect a carrier on the channel. If it detects a carrier, then it keeps the radio on long enough to detect a packet. Because the LPL check period is much longer than a packet, a transmitter must send its first packet enough times for a receiver to have a chance to hear it. The transmitter stops sending once it receives a link-layer acknowledgment or a timeout. The timeout is a few milliseconds longer than the receiver's check period. When a node receives a packet, it stays awake long enough to receive a second packet. Therefore, a packet burst amortizes the wakeup cost of the first packet over the follow-up packets. It is therefore more energy efficient to send packets in bursts when using low-power listening than sending individual packets at some fixed constant rate. Keep this in mind when developing applications that require low power operation.

Controlling the operation of low-power listening in TinyOS is provided through the use of the `LowPowerListening` interface. The Interface is supported by the cc1000, the cc2420 and the rf230 radios.

```
interface LowPowerListening {
 /**
  * Set this this node's radio wakeup interval, in milliseconds. After
  * each interval, the node will wakeup and check for radio activity.
  *
  * Note: The wakeup interval can be set to 0 to indicate that the radio
  * should stay on all the time but in order to get a startDone this
  * should only be done when the duty-cycling is off (after a stopDone).
  *
  * @param intervalMs the length of this node's Rx check interval, in [ms]
  */
 command void setLocalWakeupInterval(uint16_t intervalMs);
 /**
  * @return the local node's wakeup interval, in [ms]
  */
 command uint16_t getLocalWakeupInterval();
 /**
  * Configure this outgoing message so it can be transmitted to a neighbor mote
  * with the specified wakeup interval.
  * @param 'message_t* ONE msg' Pointer to the message that will be sent
  * @param intervalMs The receiving node's wakeup interval, in [ms]
  */
 command void setRemoteWakeupInterval(message_t *msg, uint16_t intervalMs);
 /**
  * @param 'message_t* ONE msg'
  * @return the destination node's wakeup interval configured in this message
  */
 command uint16_t getRemoteWakeupInterval(message_t *msg);
}
```

This interface is located in `tos/interfaces` in the standard TinyOS tree. Take a look at the comments for each command to familiarize yourself with how this interface can be used.

Using this interface typically involves first setting a nodes local duty cycle within the `Boot.booted()` event of the top level application. For each packet the application wishes to send, the duty cycle of its destination is then specified as metadata so that the correct number of preambles can be prepended to it. The code snippet found below demonstrates this usage pattern:

```
event void Boot.booted() {
  call LPL.setLocalSleepInterval(LPL_INTERVAL);
  call AMControl.start();
}

event void AMControl.startDone(error_t e) {
  if(e != SUCCESS)
    call AMControl.start();
}

...

void sendMsg() {
  call LPL.setRxSleepInterval(&msg, LPL_INTERVAL);
  if(call Send.send(dest_addr, &msg, sizeof(my_msg_t)) != SUCCESS)
```

```
    post retrySendTask();
}
```

The `AMControl` interface is provided by `ActiveMessageC`, and is used, among other things, to enable the operation of low-power listening for the radio. Once `AMControl.start()` has completed successfully, the radio begins to duty cycle itself as specified by the parameter to the `setLocalSleepInterval()` command. Calling `setRxSleepInterval()` with a specific sleep interval then allows the correct number of preambles to be sent for the message specified in its parameter list.

Microcontroller Power Management

Microcontrollers often have several power states, with varying power draws, wakeup latencies, and peripheral support. The microcontroller should always be in the lowest possible power state that can satisfy application requirements. Determining this state accurately requires knowing a great deal about the power state of many subsystems and their peripherals. Additionally, state transitions are common. Every time a microcontroller handles an interrupt, it moves from a low power state to an active state, and whenever the TinyOS scheduler finds the task queue empty it returns the microcontroller to a low power state. TinyOS uses three mechanisms to decide what low power state it puts a microcontroller into: status and control registers, a dirty bit, and a power state override. Please refer to TEP 112 (http://www.tinyos.net/tinyos-2.x/doc/html/tep112.html) for more information.

As a developer, you will not have to worry about MCU power managment at all in most situations. TinyOS handles everything for you automatically. At times, however, you may want to use the provided power state override functionality. Take a look at `tos/chips/atm128/timer/HplAtm128Timer0AsyncP.nc` if you are interested in seeing an example of where this override functionality is used.

Low Power Sensing Application

A fully functional low-power sensing application has been created that combines each of the techniques found in this tutorial. At present, this application is not included in the official TinyOS distribution (<= 2.0.2). If you are using TinyOS from a cvs checkout, you will find it located under `apps/tutorials/LowPowerSensing`. Otherwise, you can obtain it from cvs by running the following set of commands from a terminal window:

```
cd $TOSROOT/apps/tutorials
cvs -d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroot/tinyos login
cvs -z3 -d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroot/tinyos co -P -D 2007-10-3 -d LowPowerSensing tinyos-2.x/apps/tu
```

Just hit enter when prompted for a CVS password. You do not need to enter one.

This application has been tested on telosb and mica2 platforms, but should be usable on others without modification. Take a look at the README file found in the top level directory for more information.

Related Documentation

- TEP 103: Permanent Data Storage (Flash) (http://www.tinyos.net/tinyos-2.x/doc/html/tep103.html)
- TEP 105: Low Power Listening (http://www.tinyos.net/tinyos-2.x/doc/html/tep105.html)
- TEP 108: Resource Arbitration (http://www.tinyos.net/tinyos-2.x/doc/html/tep108.html)
- TEP 109: Sensors and Sensor Boards (http://www.tinyos.net/tinyos-2.x/doc/html/tep109.html)
- TEP 112: Microcontroller Power Management (http://www.tinyos.net/tinyos-2.x/doc/html/tep112.html)
- TEP 114: SIDs: Source and Sink Independent Drivers (http://www.tinyos.net/tinyos-2.x/doc/html/tep114.html)
- TEP 115: Power Management of Non-Virtualised Devices (http://www.tinyos.net/tinyos-2.x/doc/html/tep115.html)
- Integrating Concurrency Control and Energy Management in Device Drivers (http://www.klueska.com/pubs/klues07sosp.pdf)

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