Introduction

TinyOS is an open source operating system designed for microcontrollers and wireless applications. TinyOS was originally designed at UC Berkley in an effort to build a small lightweight operating environment for microcontrollers used in wireless sensor applications. Low-cost embedded sensor networks are a growing field of study and TinyOS is primarily intended to be a tool for developing small efficient applications.

To meet this goal, TinyOS uses an extension to the C programming language called NesC. NesC is a precompiler that looks at the connections between the parts of the code and decides what is being used and therefore what needs to be compiled. Thus it is possible to build a large resource of support tools into TinyOS, without having them take up memory and processor time unless they are needed for the specific application.

In this memorandum, we will provide a basic guide for using TinyOS to build applications for the Harrisboard2.0. This guide includes: a basic overview of TinyOS, an explanation of the development environment, a walk-through for building your first application and specification for the various components currently supported by TinyOS on the Harrisboard2.0. Additionally, we provide links to several other TinyOS resources to provide a more thorough understanding.

Capabilities of TinyOS

Before diving in to working with TinyOS, take a moment to check that it provides the tools you need to build your application. TinyOS on the PIC processor is a port of the operating system from the default platforms and does not have as complete support as some of the other platforms do. (You can extend the support for TinyOS on the PIC as well.) Currently, TinyOS for the Harrisboard2.0 supports: timers, Serial Peripheral Interface (SPI) transmit, UART (untested), LED's, and the Pulse Width Modulation (PWM) unit. (Of course you also have direct access to any of the hardware you need, but the port contains modules to simplify the use of these modules.)

Development Environment

TinyOS is designed to be built in a POSIX (i.e. Unix-like) operating system. MPLAB (the PIC compiler) runs on Windows. Thus in order to use TinyOS, we have built a flash drive image of TinyOS on Cygwin, a POSIX emulator for Windows. To run the cygwin environment, download the zipped image and unzip it on to the root of a flash drive (requires at least 256MB). If you're on one of the lab computers, you should see the drive mounted at "E:\". Run the file "E:\cgywin.bat". (If the drive is mounted at another drive letter, you'll need to edit cygwin.bat first changing the line "set CYGROOT=E:" replacing "E:" with the flash drive's mount letter.) After a moment, you will be given a command line prompt. (You can ignore errors referring to missing java packages.) The TinyOS build tree is in "/opt/tinyos-1.x" ("cd /opt/tinyos-1.x"). There you will find several directories: "apps/" contains the different TinyOS applications, the "tos/" directory contains the files making up the TinyOS system, the "contrib/" directory contains files contributed by teams around the world to the TinyOS project. Files specific to the implementation on the Harrisboard2.0 are found in the "tos/platforms/harrisb/" directory.

Compiling

The compilation process for TinyOS uses the POSIX make tools. First, in the application directory for the app to be built, the user types in "make harrisb"; this command executes the NesC compiler using the TinyOS files for the harrisboard platform. If there are any errors in the code they will show up at this point. In the current environment, even a successful compilation will yield a string of errors ending in an “Error: unknown opcode `opcode’; make: *** [build/harrisb/main.exe] Error 1”, but these errors may be ignored.

(These errors occur because the make script tries to compile the code generated by NesC using the gcc compiler for a different processor.) If the compile was successful, the program should have created an "app.c" file in the build/harrisb/ sub-directory of the application directory. Before we compile using MPLAB, we need to make a few changes so the program will compile on the PIC. These changes are made using a perl script that can be run by typing ./convert_tcm.pl "./build/harrisb/app.c" "./build/harrisb/app_pic.c". This creates the file "app_pic.c" that can then be included in an MPLAB project and compiled for the PIC using the C compiler. (Don't forget to include the PIC hardware definitions header.) There is a sample project in the "proj/" directory on the flash drive.

Program Structure

TinyOS applications have a basic structure consisting of Configuration files usually titled *"appname*.nc" and files containing the implementation code called "*appname*M.nc".

The implementation file contains two sections: the module description and the implementation. The Module description declares what interfaces the module provides and/or uses. (An interface describes a set of functions that can be called to provide a set of functionality. For example the SPI interface describes an init function, a get function and a put function.) The application generally "uses" interfaces like "Leds" or "SPI", but it also always "provides" the "StdControl" interface, which is used to start the program when the PIC is reset. The Implementation the function contains a set of functions that fit the interfaces described in the module description. This section can also contain generic C functions and declarations.

The Configuration file contains the instructions that control what modules will be used and what they will be referred to as in the application. The configuration first lists a set of components that will be used including *appname*M and then wires the module that "provides" interface "X" to the module that "uses" interface "X". For example if "LedsC" provides the "Leds" interface and your application "appM" uses "Leds" then you write "appM.Leds -> LedsC.Leds".

Blink

The Blink application demonstrates the use of timers and the LED interface provided for the PIC. In our port for the Harrisboard2.0 the red LED is assigned to port D1, the yellow LED is assigned to port D2, and the green LED is located at port D3. The Blink application is also a good place to get started in understanding the structure of a TinyOS app. For a full listing of the code you can look in the sample code section of this report. The code is also included in the apps folder included with the Cygwin package. The program utilizes a simple counter variable that is declared after the initialization by writing `uint8_t counter = 0`. In the StdControl init method the program initializes its needed components. In this case the LED module is set up using the `call Leds.init()` command. In the StdControl start method a repeating timer is set up by calling `Timer.start(TIMER_REPEAT, 500).` This creates a repeating timer with a period of 500ms. When the period is up a Timer.fired() event is returned. A method is created to implement this event and it increments the counter and toggles the LEDs. The LEDs can be controlled by using the `Leds.redOn(),Leds.redOff()`and `Leds.redToggle()` methods. A few major concepts that can be seen in this program are the ability to "call" methods, how to set up variables, and also how to handle an event returned by a called method. These concepts will be seen and used in all of the examples.

Bargraph

The Bargraph app is a simple application that shows how to use the ADC interface for analog to digital conversion and how to set the ports on the PIC directly from a TinyOS application. The application is designed to show on a bargraph display the relative magnitude of the 10-bit value that is read on port A0. In the StdControl.init() method the lines `call ADCControl.init();, call`

`ADCControl.bindPort(0,0);` and `ADCON1bits_PCFG3 = 1; //enables Analog on port0` set up the ADC to capture on port A0. The timer fires the `ADC.getData()` method which when ready returns the `ADC.dataReady(uint16_t data)` event, which includes the data read from that port. Bargraph then reads then sets up an 8-bit value stored in bar that can then be sent to Port D using `PORTD_register = bar` to set the assignment.

SPITest

SPITest demonstrates how to send data over the Serial Peripheral Interface. In this case a counter is incremented every two seconds and then the value of that counter is sent over the SPI. The SPI is set up using the `call SPI.init()` method when the program is first launched. Then when the timer fires the counter in incremented and sent over the SPI using the `call SPI.put(counter)` method. The event `SPI.get(uint8_t data)` can be used to get data sent over the SPI by another device and act upon it.

PWMTest

The PWM module sets up the PIC's pulse width modulation features and allows the user to change the period and duty cycle using a number of methods. The PWMTest demo sets up the PWM and increases the duty cycle using a counter. If the PIC is connected to an LED the brightness will slowly increase over time. When setting up the program `call PWM.init()` sets up the PWM with some default settings. Then when the timer fires `PWM.setOnPeriod(counter)` sets the duty cycle of the PWM to the increasing duty cycle. When the PWM fires it returns a `PWM.fired` result that can be used as a secondary timer.

Demo Application

Included in this report is a demo application called HBDemo that was used to demonstrate the functionality of TinyOS. This app used the PWM model to control a servomotor holding a small solar panel. The app would then use the PWM to check the relative voltage produced by the panel at each of 8 positions. When the max power was found the panel would move to that position and stay there until the voltage fell below a fraction of it's initial measured voltage. This application ran quite well and was relatively easy to develop given the tools provided by TinyOS.

Specifications

PWM - Pulse Width Modulation

```
init()          //sets up basic PWM defaults
start()         //starts running PWM
pause()         //halts PWM
setPeriod()     //sets the length of the PWM period in cycles
setOnPeriod()   //sets the length of the pulse in cycles
fired()         //event called at the end of each period
```

SPI - Serial Peripheral Interface

```
init()          //sets up basic SPI defaults
stop()          //clears SPI defaults
put()           //sends a byte over SPI
```

```
   get()              //event called when a byte is received from SPI

ADC - A/D Converter
   getData()          //starts A/D conversion
   dataReady()        //event called when A/D is complete

ADCControl - A/D Control
   init()             //sets up basic A/D defaults
   setSamplingRate()  //sets A/D sampling rate
   bindPort()         //binds a physical A/D port to a TinyOS port
```

Additional Resources

This guide provides only he basics of the TinyOS system to get a better idea of how TinyOS operates, check out the TinyOS website: www.tinyos.net You can find a wide variety of documentation and tutorials as well as browsing the contributions from other teams around the world to the TinyOS project. (Check out the CVS repository and go to the contrib/ directory.) We recommend at least skimming through the TinyOS Programming guide (http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf) to get a better sense of the way TinyOS is put together before you dive into your project.

Sample Code

Blink.nc

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

BlinkM.nc

```
/**
 * Implementation for Blink application.  Toggle the red, yellow, and green
 *  LEDs based on a counter incremented whenever a Timer fires.
 **/
module BlinkM {
 provides {
  interface StdControl;
 }
 uses {
  interface Timer;
  interface Leds;
 }
}
implementation {

 uint8_t counter = 0;

 /**
  * Initialize the component.
  *
  * @return Always returns <code>SUCCESS</code>
  **/
 command result_t StdControl.init() {
  call Leds.init();
  return SUCCESS;
 }


 /**
  * Start things up.  This just sets the rate for the clock component.
  *
  * @return Always returns <code>SUCCESS</code>
  **/
 command result_t StdControl.start() {
  // Start a repeating timer that fires every 500ms
  return call Timer.start(TIMER_REPEAT, 500);
 }

 /**
  * Halt execution of the application.
  * This just disables the clock component.
```

```
  *
  * @return Always returns <code>SUCCESS</code>
  **/
 command result_t StdControl.stop() {
  return call Timer.stop();
 }

 /**
  * Toggle the LEDs in response to the <code>Timer.fired</code> event.
  *
  * @return Always returns <code>SUCCESS</code>
  **/
 event result_t Timer.fired()
 {
  //call Leds.redToggle(); - can be used to Togggle the red LED

  counter++;
  if (counter & 0x1) {
   call Leds.redOn();
  }
  else {
   call Leds.redOff();
  }
  if (counter & 0x2) {
   call Leds.greenOn();
  }
  else {
   call Leds.greenOff();
  }
  if (counter & 0x4) {
   call Leds.yellowOn();
  }
  else {
   call Leds.yellowOff();
  }
  return SUCCESS;
 }
}
```

Bargraph.nc

```
configuration Bargraph {
}
implementation {
  components Main, BargraphM, SingleTimer, LedsC, ADCC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BargraphM.StdControl;
  BargraphM.Timer -> SingleTimer.Timer;
  BargraphM.Leds -> LedsC;
  BargraphM.ADC -> ADCC.ADC[0];
  BargraphM.ADCControl -> ADCC.ADCControl;

}
```

BargraphM.nc

```
/**
 * Implementation for Bargraph application.  Displays relative magnitude read by
 * the ADC to Port D.
 **/
module BargraphM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
    interface ADC;
    interface ADCControl;
  }
}
implementation {

  uint8_t counter = 0;

  /**
   * Initialize the component.
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    call ADCControl.init();
    call ADCControl.bindPort(0,0);
    ADCON1bits_PCFG3 = 1; //enables Analog on port0
    return SUCCESS;
  }


  /**
   * Start things up.  This just sets the rate for the clock component.
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 500ms
    return call Timer.start(TIMER_REPEAT, 500);
  }

  /**
   * Halt execution of the application.
   * This just disables the clock component.
   * @return Always returns <code>SUCCESS</code>
```

```
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }


  /**
   * Call <code>ADC.getData</code> in response to the <code>Timer.fired</code> event.
   * @return Always returns call to <code>ADC.getData</code>
   **/
  event result_t Timer.fired()
  {
    return call ADC.getData();
  }

  /**
   * Sets the Port D register based on the data result returned by the
   * <code>ADC.dataReady</code> event
   * @return Always returns <code>SUCCESS</code>
   **/
  async event result_t ADC.dataReady(uint16_t data){
        uint8_t bar=0;
        int i;

        for(i=0;i<8;i++){
                if(data>=(i+1)*128){
                        bar = bar|(1<<i);
                }
        }
        PORTD_register = bar;

        return SUCCESS;
  }
}
```

SPITest.nc

```
configuration SPITest {
}
implementation {
  components Main, SPITestM, SingleTimer, SPIC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> SPITestM.StdControl;
  SPITestM.Timer -> SingleTimer.Timer;
  SPITestM.SPI -> SPIC;
}
```

SPITestM.nc

```
/**
 * Implementation for SPITest application sends the value of a counter to the SPI.
 **/
module SPITestM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface SPI;
  }
}
implementation {

  uint8_t counter = 0;

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call SPI.init();
    return SUCCESS;
  }


  /**
   * Start things up.  This just sets the rate for the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 500ms
    return call Timer.start(TIMER_REPEAT, 500);
  }

  /**
   * Halt execution of the application.
   * This just disables the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
```

```
  /**
   * Sends the counter value to the SPI in response to the <code>Timer.fired</code>
event.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
    call SPI.put(counter);
    counter++;
    return SUCCESS;
  }

  async event result_t SPI.get(uint8_t data){
        return SUCCESS;
  }
}
```

PWMTest.nc

```
configuration PWMTest {
}
implementation {
  components Main, PWMTestM, PWMC, SingleTimer;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> PWMTestM.StdControl;
  PWMTestM.Timer -> SingleTimer.Timer;
  PWMTestM.PWM -> PWMC;
}
```

PWMTestM.nc

```
module PWMTestM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface PWM;
  }
}
implementation {

  uint16_t counter = 0;

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call PWM.init();
    return SUCCESS;
  }


  /**
   * Start things up.  This just sets the rate for the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 2s
    return call Timer.start(TIMER_REPEAT, 2000);
  }

  /**
   * Halt execution of the application.
   * This just disables the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }


  /**
```

```
   * increment counter and set new on period
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
        counter++;
        call PWM.setOnPeriod(counter);
        return SUCCESS;
  }

  //dummy function - run at end of PWM period
  async event result_t PWM.fired()
  {
        return SUCCESS;
  }
}
```

HBDemo.nc

```
/**
 * HBDemo is a demonstration app designed for the Harrisboard used in
 * the Harvey Mudd College Eng 155 class.  It demonstrates SPI, PWM, and
 * A/D functionality by controlling a Hitec HS-322HD Servo attached to
 * a small solar cell.
 *
 * @author jlarson@hmc.edu & mjeffryes@hmc.edu
 **/

configuration HBDemo {
}
implementation {
  components Main, HBDemoM, PWMC, SingleTimer, LedsC, ADCC, SPIC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> HBDemoM.StdControl;
  HBDemoM.Timer -> SingleTimer.Timer;
  HBDemoM.PWM -> PWMC;
  HBDemoM.Leds -> LedsC;
  HBDemoM.ADC -> ADCC.ADC[0];
  HBDemoM.ADCControl -> ADCC.ADCControl;
  HBDemoM.SPI -> SPIC;
}
```

HBDemoM.nc

```
/**
 * Implementation for HBDemo application.  Control a servo with a Solar
 * cell connected ot the A/D port.
 **/

//magic numbers
#define PERIOD 98
#define NUMPOS 8
#define SCAN 0
#define READ 1
#define FREEZE 2
#define FROZEN 3

module HBDemoM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
    interface ADC;
    interface ADCControl;
    interface PWM;
    interface SPI;
  }
}
implementation {
  //global variable initialization
  uint8_t state = SCAN;
  uint8_t cycler = 0;
  uint16_t threshhold = 0;
  uint16_t readingf = 0;
  uint16_t readings[NUMPOS] = {0,0,0,0,0,0,0,0};
  int position = -1;
```

```
  //calculate bargraph style display
  uint8_t calcBarData(uint16_t data){
    uint8_t bar = 0;
    int i;
    for(i=0;i<8;i++) {
        if (data>=(i+1)*128) {
                bar = bar | (1<<i);
        }
    }
    return bar;
  }

//calculate decimal equivalent to send over SPI
  uint8_t calcSPIData(uint16_t data){
    uint8_t spidata = 0;
    int i;

    for (i = 0; i < 3; i++) {
      if (data >= (i + 1) * 310) {
          spidata = spidata + 16;
          data = data - 310;
        }
    }
    for (i = 0; i < 8; i++) {
      if (data >= (i + 1) * 31) {
          spidata = spidata + 1;
        }
    }
    return spidata;
  }

  //find the maximum reading stored in the array
  int findMax(){
    int i;
    int maxindex = 0;
    for(i=1;i<8;i++){
        if(readings[i]>readings[maxindex]){ maxindex = i; }
    }
    PORTD_register = calcBarData(readings[maxindex]);
    call SPI.put(calcSPIData(readings[maxindex]));
    threshhold = readings[maxindex]*0.7;
    return maxindex;
  }

  /**
   * Initialize the components.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    call ADCControl.init();
    call ADCControl.bindPort(0,0);
    call SPI.init();
    call PWM.init();
    call PWM.start();
    return SUCCESS;
  }


  /**
   * Start things up.  This just sets the rate for the clock component.
```

```
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 1.0 s
    return call Timer.start(TIMER_REPEAT, 1000);
  }

/**
   * Halt execution of the application.
   * This just disables the PWM and clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
    return call PWM.pause();
  }


  /**
   * Move between states based on the <code>Timer.fired</code> event.
   *
   * SCAN: Sets PWM period to move to next position
   * READ: Takes a reading at the current position
   * FREEZE: Finds the position with max measured voltage and moves there
   * FROZEN: Check A/D results to see if below threshhold
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
   if(state == SCAN){
    position++;
    state = READ;
   }else if(state == READ){
     call ADC.getData();
     if(position == 7){
        state = FREEZE;
     }else{
        state = SCAN;
     }
   }else if(state == FREEZE){
     //call Timer.stop();
     position = findMax();
     readingf = 1028;
     state = FROZEN;
   }
   else if(state == FROZEN) {
     call ADC.getData();
     if ( readingf < threshhold ) {
        position = -1;
        state = SCAN;
     }
   }
   return SUCCESS;
  }

/**
   * Handles A/D data on the <code>ADC.dataReady</code> event.
   *
   * When in the FROZEN state the A/D handler saves
```

```
   * data to readingf.  When not it saves the data to an array
   * based on the current cell position.
   *
   **/
  async event result_t ADC.dataReady(uint16_t data) {
    PORTD_register = calcBarData(data);
    call SPI.put(calcSPIData(data));
    if (state == FROZEN ) {
        readingf = data;
        }
    else {
        readings[position] = data;
        }
    return SUCCESS;
  }

/**
  * Handles PWM
  *
  * When the PWM module reaches the end
  * of a cycle this function is called
  * This function sets the PWM to create
  * a longer cycle than usually possible
  **/
  async event result_t PWM.fired()
  {
        if(cycler<(position+4)){
                call PWM.setOnPeriod(0x03FC);
                CCP1CON_register = (CCP1CON_register | 0x30);
        }else if(cycler<PERIOD){
                call PWM.setOnPeriod(0x0000);
                CCP1CON_register = (CCP1CON_register & 0xCF);
        }else{
                cycler = 0;
        }
        cycler++;
        return SUCCESS;
  }

  //Dummy function for unimplemented SPI recieve
  async event result_t SPI.get(uint8_t data){
        return SUCCESS;
  }
}
```