Nick Repetti

CPE 301

Spring 2013

Wireless Sensor Network – Final Project

Overview

The goal of this project is to create a Wireless Sensor Network using the AVR Atmega 328P microcontroller. The AVR will read a value from a temperature sensor and do an A2D conversion of it. The value will be displayed on a 10 bit LED bar, in binary. Then it will transmit this data wirelessly to another AVR, which will, based on the temperature value, send a PWM to a RGB LED. If it is cool, the LED will be blue, and if it is hot, the LED will be red. This project does some other operations as well, such as waking up from a sleep mode, and storing the A2D value in a flash memory location using a cyclic buffer.

I was not able to get the wireless transreceiver working, so the data is not sent wirelessly, but all of the other operations still work. Reading from a temperature sensor, an A2D is done through an interrupts. This temperature value is then displayed in binary on the LED bar. Based on that value, a PWM is sent to color a RGB LED to be “warm” or “cool.”

Components

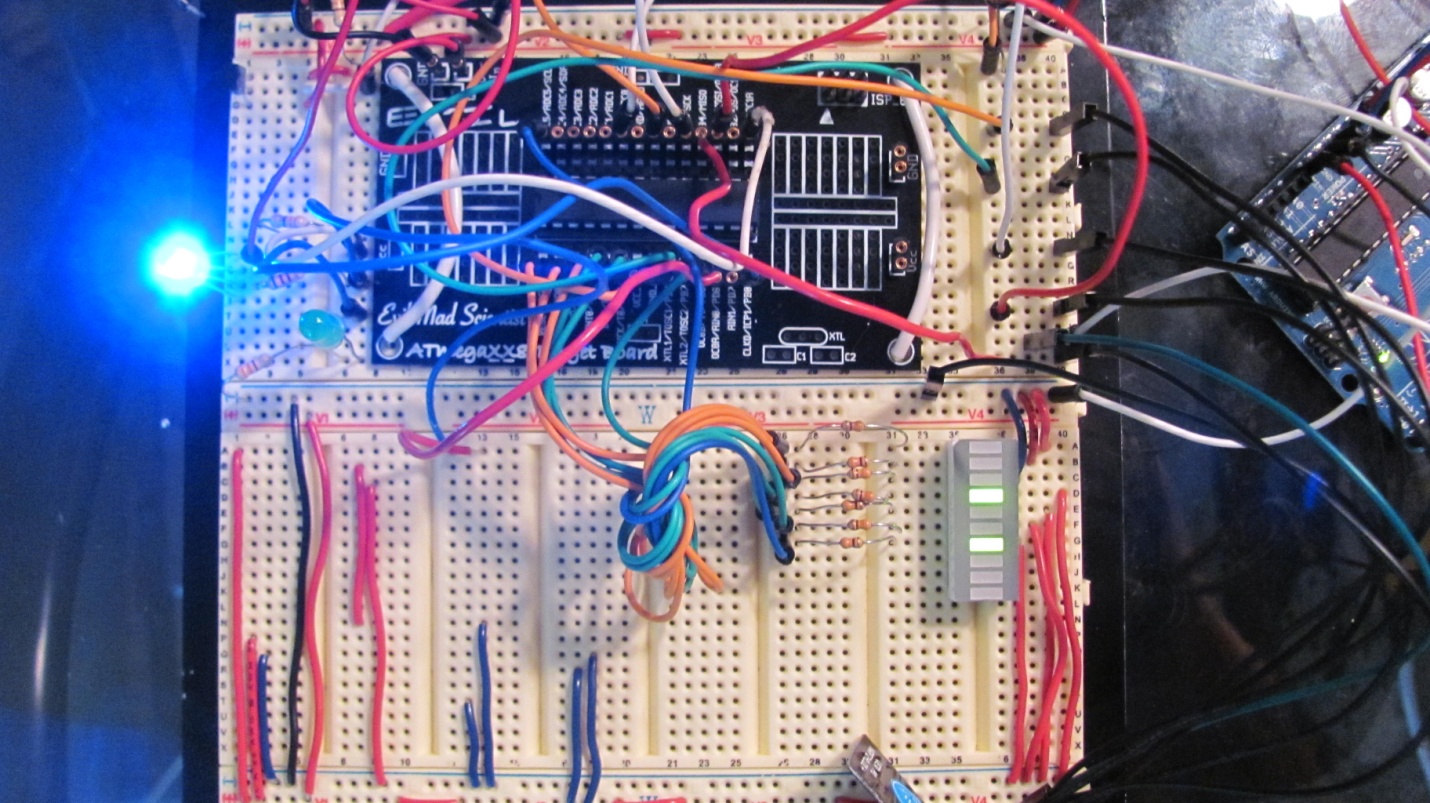
Here is a list of all of the components used for a working project:

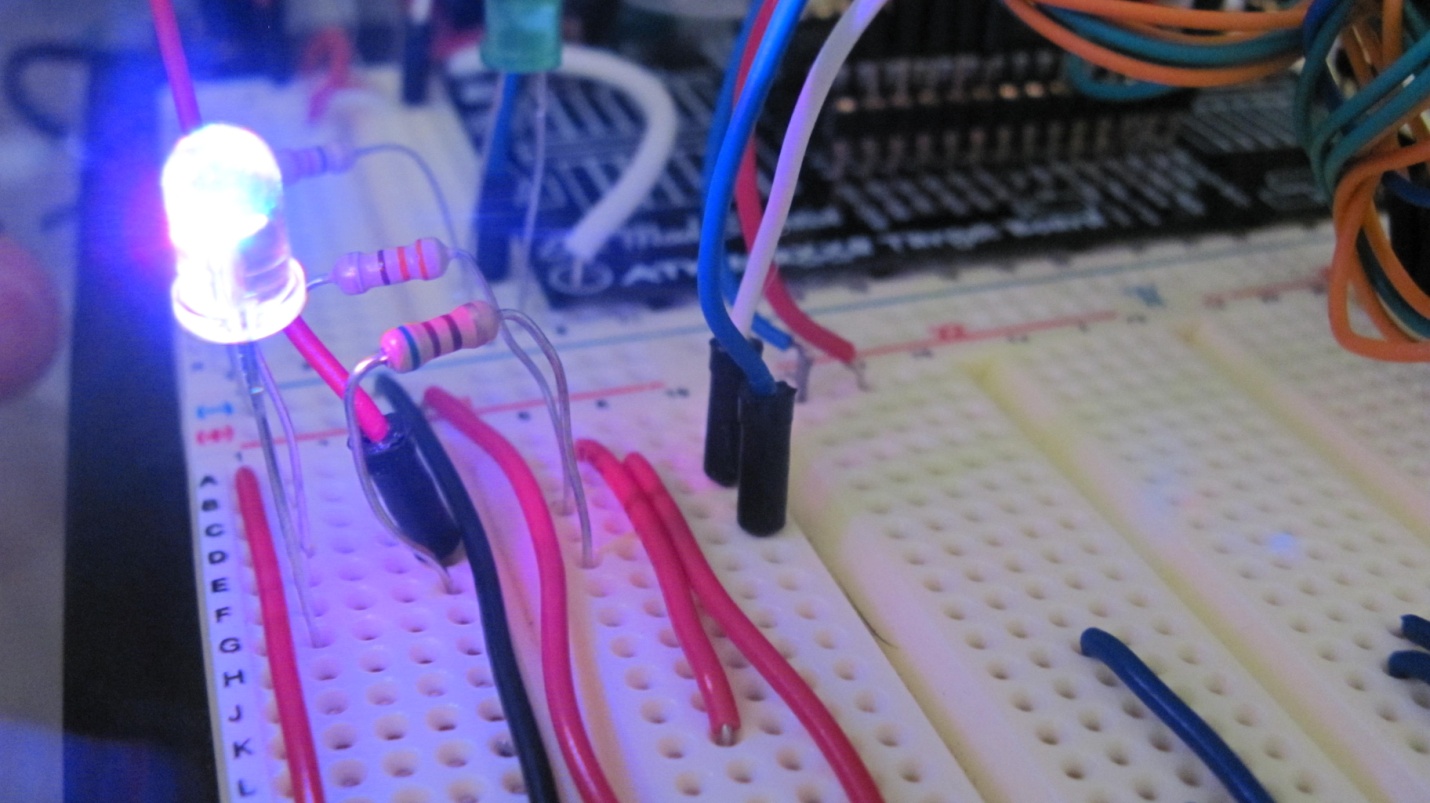
* AVR ATmega 328p Microcontroller (x2)
* RGB LED (x1)
* 10-bit LED Bar (x1)
* Green LED (x1)
* 100 uF Capacitor (x1)
* LM34 Temperature Sensor (x1)
* 330 Ohm Resistor (x8)
* nRF24L01 Wireless Transreceiver (x2)

My project only uses one AVR, and only 7 bits of the LED bar are being used (As the temperature will not exceed 127 degrees F.), and the Green LED is used for indicating a successful wireless transmission of data.

Images

Below are two images of the board. The first one shows the complete board layout, and the second one shows the RGB LED changing to red after a threshold of 80 degrees has been met. Also in the first image, I was using an Arduino as a power supply because I was working from home, as it had both 5v and 3.3v outputs, and could be powered by a 9 volt battery.





Code

Below I will go in depth of how my code works.

**A2D Routine:**

ADCSRA |= ( 1 << ADPS2 ); // 16 Prescaler

ADMUX |= ( 1 << REFS0 ) | ( 1 << REFS1 ); // Internal 1.1 Voltage Reference

ADCSRA |= 1 << ADIE; // Enable interrupts

ADCSRA |= 1 << ADEN; // Enable ADC

sei(); // Enable interrupts

ADCSRA |= 1 << ADSC; // Start conversion

This code configures the A2D. I used interrupts and read the value that was on the ADC0 pin.

static volatile uint16\_t adc\_result; // Declare a global variable

// to hold the ADC result

ISR( ADC\_vect ){

uint8\_t adc\_low = ADCL;

adc\_result = ( ADCH << 8 ) | adc\_low; // Place ADCH:ADCL into a //variable(Only uses 10 bits)

ADCSRA |= 1 << ADSC; // Restart conversion

}

I declared a global variable to make it easier to use the A2D result outside of the main function. Once the A2D occurs, it triggers the interrupt, and then I store the value from ADCH/ADCL into the variable.

**A2D Storage Routine (Cyclic Buffer):**

#define BUFFER\_ONE \* (volatile unsigned char \*) 0x0115

#define BUFFER\_TWO \* (volatile unsigned char \*) 0x0116

#define BUFFER\_THREE \* (volatile unsigned char \*) 0x0117

#define BUFFER\_FOUR \* (volatile unsigned char \*) 0x0118

#define BUFFER\_FIVE \* (volatile unsigned char \*) 0x0119

To make it easier to store into the flash memory, I pre-defined 5 memory locations, which I will be using for the cyclic buffer.

int bufferCount = 1; // Which location of the buffer we are currently in

int transmitLimit = 0; // Tracks when 10 values needed to be averaged

int tempAverage = 0; // Variable to hold the average of 10 A2D results

int temp = (uint8\_t) ( adc\_result ); // Calibrate sensor, convert to a byte

// Based on what location the bufferCount is at, place the current temp into // the buffer

switch ( bufferCount ){

case 1:

BUFFER\_ONE = temp;

case 2:

BUFFER\_TWO = temp;

case 3:

BUFFER\_THREE = temp;

case 4:

BUFFER\_FOUR = temp;

case 5:

BUFFER\_FIVE = temp;

default:

break;

}

// After the temperature was stored into the buffer, increment the count

bufferCount++;

// Reset the buffer location once it reaches the end

if ( bufferCount == 6 )

bufferCount = 1;

// Only averaging 10 values, so reset the count once it gets to 10

if ( transmitLimit > 9 )

transmitLimit = 0;

tempAverage += temp; // Running sum to be averaged after 10

transmitLimit++; // Increment the current number of

if ( transmitLimit == 9 ){

tempAverage /= 10;

// Now this is where I would send the data

// ex: transmit(tempAverage);

tempAverage = 0;

}

This code converts the A2D result into an 8-bit variable. This works because the A2D result will at most be 8-bits, as the temperature sensor will never read above 255 degrees F. The cyclic buffer works as follows:

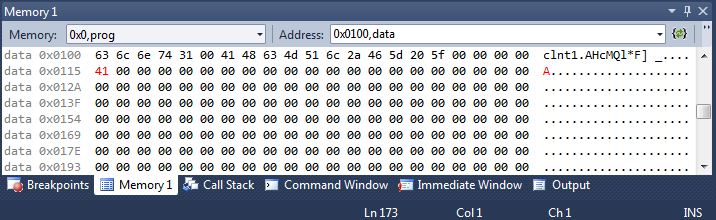
* The temperature is stored into buffer location 1
* Buffer count is increased (Now we’re looking at location 2)
* A new temperature is received, and stored into location 2
* Once buffer location 5 is filled, the next temperature reading is stored into location 1, overwriting the first value
* This continues forever, storing the values into locations 1-5, and then overwriting them

While the cyclic buffer is working, there is another counter that keeps track if 10 values have been read in yet. After 10 values have been read in (and then added together), I take the average of them and then transmit that average. Then the 10-counter is reset, so it can be used for the next 10 values.

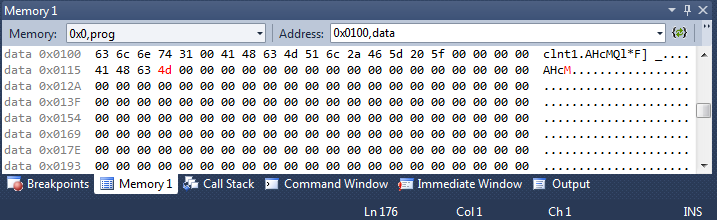
In order to simulate the cyclic buffer, I preset values to be placed into the buffer. This is done because I can not generate a real A2D result in the simulation. Below are some screenshots of the simulation for the cyclic buffer.

temperatureValues.png

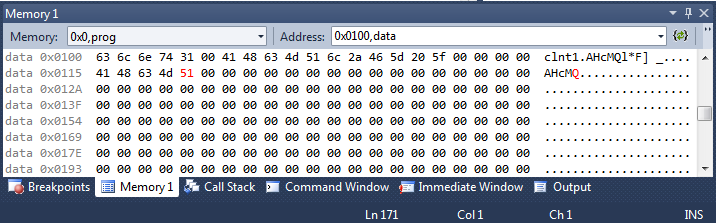
Here are the values I will be using to store into the Flash Memory using the Cyclic Buffer.



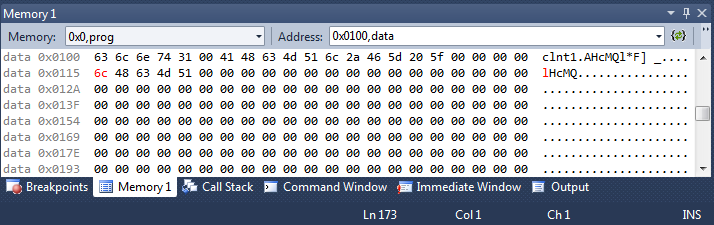
The buffer has started, and the first value of 65 (0x41) is placed into position 1.



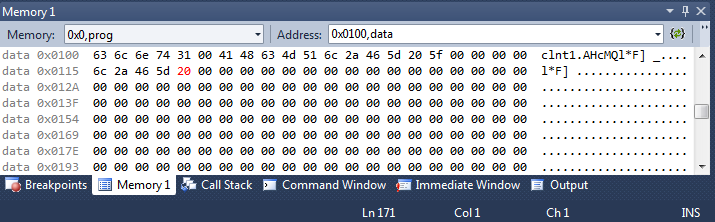
The buffer continues to get filled.



The buffer is now full



Since the buffer is full, it will now start overwriting values from the beggining.



The buffer has now gone through all 10 elements

**PWM Generation & LED Bar Display:**

In order to display the value to the LED bar, the following code is used.

int temp = (uint8\_t) ( adc\_result ); // Calibrate sensor, convert to a byte

PORTD = temp;

As stated earlier temp is just an 8-bit variable holding the value of the temperature reading. Then I send this to PORTD, which is connected to the LED bar, and this displays the correct temperature value in binary.

// Initialize Timer1/0

// Fast PWM Mode (8bit), Enable OC0A/OC1A/OC1B

TCCR0A |= ( 1 << COM0A0 ) | ( 1 << WGM01 ) | ( 1 << WGM00 );

TCCR1A |= ( 1 << COM1A0 ) | ( 1 << COM1B0 ) | ( 1 << WGM10 ) | (1<< WGM11 );

This code initializes Timer1/0. The red of the RGB led is connected to OC1A, and the blue is connected to OC0A. OC1A/OC0A is toggled every compare match from the Fast PWM. Now that the timers’ settings are initialized, they actually have to be started.

if ( adc\_result > 80 ){

TCCR0B = 0x00;

TCCR1B = 0x1D; // 1024 Prescaler

OCR1A = 0x7FFF; // 50% Duty Cycle

}

else{

TCCR1B = 0x00;

TCCR0B = 0x0D; // 1024 Prescaler

OCR0A = 0x1F; // Duty Cycle

}

Based on the temperature value, a PWM will be sent to either the Red, or the Blue line of the RGB LED. The threshold is set to 80 degrees F. When the temperature is above 80 degrees, I stop timer 0 (Which means turn off the blue channel), and then I start timer 1 (Which means turn on the Red channel). If the temperature is below 80 degrees, it will do the opposite (Turn off Red, Turn on Blue).

**Sleep Timer:**

#include <avr/sleep.h>

// Set the SM1 and SM0 Bits in the SMCR for power saver mode

set\_sleep\_mode(SLEEP\_MODE\_PWR\_SAVE);

cli(); // Turn off interrupts so it does not wake up from the ADC

TCCR2A |= 0x00; // Configure Timer2

TCCR2B |= 0x06; // Start Timer 2

The sleep.h file is used to clean up the code a little bit. It basically has functions built in the configure the SMCR, but done in a clean way. The sleep mode I use is the Power Saver, which wakes up on Timer 2 compare match. I disable interrupts so the AVR will only wake up from the Timer 2 overflow.

sleep\_enable(); // Start sleeping

sei(); // On wakeup, enable interrups again

The sleep\_enable(), starts the AVR to sleep. Once the timer2 match is set, the AVR wakes up. More sleep modes and different sleep times can be set by modifying Timer 2. Also, there is another function that you can call, sleep\_disable(). The stops the AVR from sleeping. With this, you can put a timer delay in between the enable/disable calls, or enable interrupts during this period.

**Altium:**

The next two pages have snapshots from my altium design. I have also included larger versions of these images in the email, along with the LIB and Schematic files.

