# theWeather.system

12/10/2012-4/30/12
Designer: Mike Gurr and Todd Sukolsky

## Project Overview:

The system is meant to be placed outside, on the dock, of Boston University's DeWolfe Boathouse as a weather station. The measurements taken should be at minimum Humidity and temperature. If time allows wind speed and sun-exposure should also be measured. The system should be stand-alone, powered by a solar panel and backup battery. The backup battery is charged during the day if the sun is out and must be able to withstand a sunlight drought of up to one week. The method of communication with the device is through wireless transmission to a hub located in the boathouse coaching hallway, which is connected to a computer that will post current conditions to a website viewable on a web browser. The end goal is to create an iOS application that fetches data from the website and displays that information.

Planned Hardware and Implementation:

The system will use an Atmega324PA to collect data from a PCB temperature monitor as well as a thermistor and a humidity sensor. The humidity sensor outputs linear voltage and is made by Honeywell Electronics, the thermistor is N-type and is scaled down with a voltage divider into an ADC pin and, in the first REV, two PCB monitors will be implemented: TI's LM95071 and Analog Devices AD7476, both of which communicate over SPI. Readings will be taken in every twenty minutes and stored in EEPROM. One day will be cataloged in memory as well as data from the last year. The data required is average high for the day and average low for the night. This data will be used to map the seasons. The megaAVR also has a RTC being driven by a backup battery when the circuit is not on. The RTC will periodically wake up from sleep, check the time and date on the RTC, take measurements, store the measurements and then go back to sleep in order to save power.  The RTC is Microchip's MCP79410, which communicates over I2C.

Also onboard will be an ATtiny84A which is responsible for regulating voltage between the battery and solar panel. If possible, the ATmega324PA will be replaced by just one ATtiny84A to save space. Therefore, in the first implementation/rev there will be a parallel ATtiny to test SPI, I2C and thermistor data acquisition. If the ATtiny is comparable to the ATmega the ATmega will be scrapped and the ATtiny will be implemented.

The wireless chips to be used are RF transmitters that can send data one half mile to a parallel chip. The chip will feed into a BeagleBone or Raspberry Pi, preferably BeagleBone, which will then post the data to the web.

## Project Presented on Monday, April 29th:

On demo day the following parts of the projects were implemented: Temperature readings from both onboard SPI-by-wire chips (LM95071 and the AD7476), thermistor temperature readings and humidity readings, an email server that emails out current readings at 8AM, 12PM, 5PM and 9PM as well as the past week statistics every Sunday morning at 7AM. I2C communication with the ATtiny that also takes thermistor readings was not finished as Professor Giles needed more people to volunteer for the Monday presentations. Also, I2C communication to the RTC on the board was not debugged, however a RTC was implemented on the ATmega324PA

**Implementation-Real Time Clock:**

The ATmega324PA is responsible for collecting all temperature and sensor readings. The chip is set to an internal clock frequency of 8MHz and has one timer operating with a clock of sysClock/1024. This will cause an overflow of the 8-bit counter 30 times in one second, thus giving us a Real Time Clock that can be set through communication protocols. To implement the RTC a “clock” class was created which can be set with time strings over UART. Each time the timer overflows for the 30th time the clock class “addSeconds(1)” which increments the time by one second. The clock will increment days and hours and months correctly, however it does not take into the account Leap Years since the date can eventually just be updated with the external Real Time Clock chip or through UART communications with the server. Based on the Real Time Clock every twenty minutes readings are taken from all of the sensors and compiled into another class, the “thermostat” class.

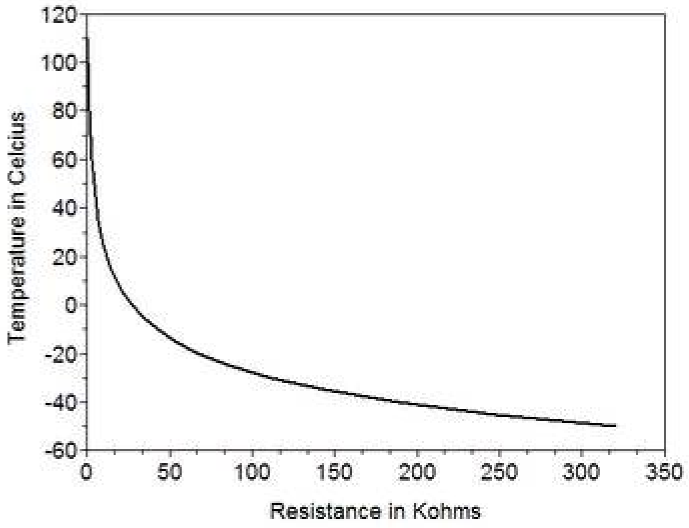
**Implementation-Temperature  and Humidity Sensing:**

The humidity sensor on-board is made by HoneyWell Electronics and outputs a linear voltage based on humidity. The device takes in 3.3V for power and outputs the reading to an ADC pin on the mega controller. The mega controller than samples the pin and converts the reading into a humidity based on the linear voltage scaling provided by HoneyWell in their data sheet. For more information see the devices data sheet or investigate the source code which has detailed explanations of the conversion factors.

As mentioned in the ‘Project Overview’ section, there are two on-board temperature sensors that communicate via SPI. One of these devices is connected to the main SPI ports on the ATmega324PA while another is set to communicate over the USART1 port that can be manipulated to work with SPI in three-wire mode. Because the TI chip, which communicates on the USART1 port, needs a chip select line, the line used was a normal GPIO line that is pulled high when the microcontroller is initialized and held high until communication needs to take place.

Both on-board sensors keep a 13-bit temperature reading that spans to the negative Celsius as well. This means to get one unit’s temperature reading an 8-bit SPI communication must be done. The TI chip sends the MSB first, so the first received byte must be shifted left in a uint16_t variable and the second byte bitwise or'ed with the variable. The same is true for the Analog Device 7476 chip. Conversions are then done to change the decimal conversion into Fahrenheit. For more information on those conversions see each chips datasheets and/or the implementation code.

To accompany the two on-board temperature readings is a thermistor that acts as a voltage divider that feeds into one of the ADC pins on the mega. The thermistor is an NTC thermistor, meaning the resistance increases as temperature decreases. A graph of the relative resistance is below.



To turn the voltage reading into an actual temperature several approximations were made. First off, a linear fit line was established for the temperature range of 0 Celsius to 40 Celsius. With this approximate slow the temperature can be found by subtracting 1024 (max possible ADC reading) by the ADC reading (since the voltage divider has the thermistor on the bottom of the tree) as well as a basic offset. Dividing the resulting value by the slope yields the temperature in Celsius and then the temperature is converted into Fahrenheit.

**Implementation-Data Tracking:**

The thermostat class is used to keep records of past temperature readings and running averages. When a new reading is taken, the results are taken and added to the class. If the temperature taken by the thermistor is above the high, the daily high is set to the new value and if the temperature is below the low then the daily low is set to the new value. Along with this adjustment the day's average temperature is recorded. To implement this the class must know how many new data points have been added. One way to implement this was with an array, another was with a custom vector class. The most dumbed down way to accomplish this was to keep track of how many readings were taken in a separate variable then weight the average with (numReadings-1)/numReadings and add that to the current temp/numReadings, which will yield an accurate new measurement.

It follows that the thermostat class keeps track of the current day as well as weekly averages. The day's data is reset when a new day happens, ie the internal RTC goes from 23:59:59 to 0:0:0 on its timing. There should then be a temperature reading done that starts new data for the day since every 20 minutes data should be taken. On that new day, the past days averages are added to the weekly average and a "days" counter is incremented to aid in the calculation of averages. The weekly statistics are cleared when the Raspberry Pi asks the microcontroller for the statistics. At this point, the time is set to 7AM, since that's when the Raspberry Pi asks for the data, and the statistics are reset. There are a few error prevention schemes in place to avoid uninitialized data from being sent when there has been a reboot just before a new week is bebun.

The number of days contributing to the weekly averages does not need to be reset after seven implicitly because the server will ALWAYS ask the system for the week statistics unless it is powered down. If it is powered down, then theWeather.system board will also be powered down so no foul is called and the number of days will be killed.

The system also implements the AVR's 1K of EEPROM by saving the average temp, average humidity, high and low for each day in a block. Each block has an initial offset and block size which can be incremented. Using simple management functions previous days can be loaded and restored to the current date if necessary. Currently there is no system IO for this, however it is intended to provide access to a monthly list/calendar of the past weather readings to contribute to the making of an almanac.

**Communication Protocols:**

To establish communications the Raspberry Pi sends an interrupt to the mega microcontroller to wake itself up from sleep. The mega then goes into a UART receive state machine, sends an Acknowledgement to the Pi and then receives a string with instructions for what readings to capture and what data to transmit. The string receiving state does not use interrupts, but only the control flags in the status register for the UART. The UART input and output is piped through an FTDI chip that then goes into a mini-USB mount which is plugged into one of the USB ports on the Raspberry Pi (usually shows up as ttyUSB0 on Pi). Each instruction string is terminated with a '.', so when the receive protocol receives a '.' it moves on to the next state which is logical comparison of the received string with choice options. If the received string matches one of the recognized characters a flag will be set to execute some functionality, otherwise the system will go back into sleep mode. There is a timeout set for UART receive protocol that spans five seconds and is dependent upon the Real Time Clock implemented in the "clock" class. If the timeout is hit, the system returns to sleep and waits for another interrupt.

**Email Server Integration- Raspberry Pi:**

The current method of communication to the outside world is through an email server setup on the Raspberry Pi. To get data, the RPi sends a string to the uC and then awaits data. Current requests are “STATS.”, “WEEK.”, “Hi.”, “save.”, “time.” and “T<real time>.” The latter method is used to set the time of the microcontrollers internal clock while Stats and Week are used to ask for temperature statistics. The script to execute all of this is a python script that opens a serial port and toggles a GPIO pin (initialized in a pinSetup script that creates the gpio in the /sys/class/gpio directory) to interrupt the uC. There is a cron script that executes the communication program at certain times of the day and once a week to ask for the current statistics and week statistics, respectively.

On boot, the Raspberry Pi connects to the internet behind my router with a static IP address, then sets the time of the uC so that the uC can take readings at the appropriate time. At the same time is starts a server that can be connected to using port forwarding that receives a connection, asks for an email, then checks to see if the email address is currently stored on the server or not. If it is not, the Pi gets the current stats from the uC and then sends a “welcome subscription” email to the new user. From then on the added email address will get the updates at certain times of the day. To remove a user from the list, the same program can be run and the email given will be checked against current emails. If that email already exists, it will delete the email address from the list.

**Assessment of Success and Future Steps:**

The first Rev of the project is considered a success because all important data was calculated and accurate to better than .5 degrees Fahrenheit. The system is not, however, complete because it still needs the integration of an RF chip that can communicate from the boathouse dock to a substation. Also, a weatherproof case needs to be modeled to store the device. For uC code, implementation of the day storing in EEPROM could be implemented, however the EEPROM is limited to 100,000 read/writes. Flash memory is much more durable so in RevB a small flash chip should be placed on-board that can store data. This will also enable more data to be stored, ideally up to one year for a full almanac entry. Overall, this initial implementation of theWeather.system was successful and needed before developing the stand-alone boards. Good proof-of-concept.

**Project Distribution:**

Todd started this project (board design and layout, basic prototyping of temperature readings) in December and January for a hobby, however Michael Gurr joined on in April to speed up the development process. The uC code was a joint effort where we both sat behind the computers and tested the code. Todd made the RasPi scripts, with input and testing feedback from Mike, as Linux has become a large hobby of his and he has a strong knowledge of python and it’s abilities in embedded systems.

**NOTE: This project has absolutely nothing to do with our Senior Design projects and was spawned completely as a hobby project. Luckily we were able to use it for something besides our own entertainment (although that’s mostly what it was).