Anthony Giordano

Phelipe Fernandes

1. Project Description

Develop a system that can monitor temperature at different depths in a lake at regular intervals. The

system should archive the data locally if no network connection is available. If a network connection is

available, transmit the data to a central repository. Develop a web interface that can display the data

collected at the central repository. Start with temperature sensor. Keep it simple, easily deployable and sturdy

2. Project Goals

-The buoy should be made using a raspberry Pi, temperature sensor, simple USB battery and otherwise household materials and readily available, inexpensive hardware (breadboard, DC motor, phone battery)

-The web interface should be easy to read, constantly updating and optimized for mobile devices, desktop or tablet.

-The website should be hosted and made using free, open source software whenever possible.

3. Your project implementation details.



Raspberry Pi running Raspian (lightweight Debian fork made for Raspberry Pi)

The circuit that controls the motor is mounted in a small breadboard that goes inside of a Tupperware container with the Raspberry pi and the batteries. The system uses two 9V batteries for the motor and a 5V battery for the Raspberry pi. For the flotation, we used two plastic bottles under a plastic surface.

4. How you broke up the project into components – describe each component

Temperature measurement:

We decided to use the same sensor used in class. The sensor DS18B20 communicates with the Raspberry Pi using the One-Wire protocol. It has 3 wires: ground, VCC and data. It needs 3.3V to work and a pull-up resistor on the data pin. In our circuit, we used the data pin #4 of the Raspberry Pi to make the connection to the sensor. After making the physical connection, we had to load two modules: one to activate the GPIO communication on the Raspberry and other to read the temperature using One Wire connection. We did that using the commands below:

>sudo echo “dtoverlay=w1-gpio” >> /boot/config.txt

>sudo modprobe w1-gpio

>sudo modprobe w1-therm

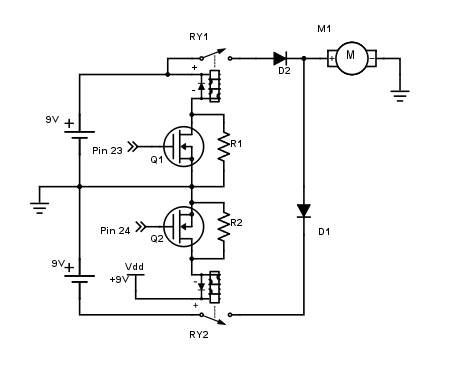
>sudo reboot

After the reboot, the Raspberry Pi is already able to read the temperature. The temperature measured by the sensor is shown in the file /sys/bus/w1/devices/28-0000069757fc/w1-slave. This file shows the MAC address of each sensor connected to the Raspberry Pi and the temperature times 1000 at the end of the line. It is necessary to format this number and save it on the database. To do it, the function *readTemp(unit)* gets the temperature from the file and returns the correct value. If the function receives 0 as parameter, it returns the temperature in degrees Celsius; if it receives 1 as parameter, it returns the temperature in degrees Fahrenheit.

Moving the sensor:

To move the sensor to different depths, we used a small DC Motor, controlled by two Relays and powered by two 9V batteries. Basically, the Raspberry Pi controls two mosfets channel N (Q1 and Q2). Those mosfets, activate the relays that activate the motor. Each mosfet controls a relay that is responsible to make the motor run clockwise or counterclockwise.

The code that controls the motor is very simple. In the circuit, the pin 23 is connected to the mosfet that activates the motor clockwise and makes the sensor go down and the pin 24 is connected to the mosfet that activates the motor counterclockwise and makes the sensor do up. The code makes the Raspberry Pi hold the pin 23 high and the 24 low for a few seconds to make the sensor does down; and the pin 24 high and the 23 low to make the sensor return to the surface.



Saving data:

After the Raspberry Pi moves the sensor, it waits a few seconds to stabilize the temperature, reads the temperature and saves in the database the date and time when this temperature was measured, the depth where the temperature was measured and the temperatures in degrees Celsius and Fahrenheit.

After all data is saved in database, another script starts to convert the database in a JSON file. It reads all data from the table and writes in a file. Now, the data can be used in the webpage, but it needs to become available.

The most efficient way to make that file available is to assign a online address to it. To do it, another script copies the JSON file to a server using the command *scp*.

All of this process, since the temperature measurement to saving the data in the cloud server, is executed by a main shell script that executes the python scripts. First, is call a python script that controls the motor, reads the temperatures and saves the data in the database. Then, it executes another python script that creates the JSON file and last it copies the file to the server.

Software:

The website was originally made using simple HTML with some CSS and JS libraries, using a graph plugin called CanvasJS that displayed randomzied data. CanvasJS was later scrapped in favor of Chart.js, a much more refined platform with a larger userbase and more expanded API, but adding real data to the graph wouldn't come until the last step of the project.

Getting the website on a network that could handle HTTP GET requests was the next step, using the microframework Flask running in an isolated python enviornment via virtualenv, this step proved to be relatively easy and efficient, and was chosen in favor of PHP due to peer advice warning me of PHP overhead.

Once we had the sensor updating a static url with plain text data of temperatures stored in a series of arrays containing the time, the depth, the temperature in celsius and fahrenheit, we could start implementing it into the chart.js graph. This proved to be the most troublesome step of the process, as issues with Javascript variable scoping and execution of functions was causing the JQuery GET request to not return until after the chart was rendered, leaving the data unused. This was eventually resolved, once I had the data in the script BEFORE chart execution, I could now write a script to parse it into arrays which could be passed to the chart.js plugin as can be seen in about.html javascript portion.

5. Good and Bad, what we would have done:

We were very pleased with the simplicity of the sensor, particularly cost efficiency and freeness of the software used. The website came out looking very professional albeit minimalistic. Given more time, we would have liked to implement:

-Solar panels for additional battery lifespan - limited by availability

-3D printed enclosure to make the project easier to replicate 1:1

-AJAX script to update the website graph without page reload

Group contribution Breakdown

Anthony Giordano - Web hosting, Webpage design, Data retrieval and parsing

Phelipe Fernandes - Sensor design and construction, Data upload and formatting