UWA Semester 2, 2019 – CITS5506 The Internet of Things Project Report: IoT Weather Station

Nathan Scott (18913101@student.uwa.edu.au)
Tracy Hong (10124352@student.uwa.edu.au)
Guobei Zhang (22063324@student.uwa.edu.au)

Abstract: There is demand for inexpensive, localised weather monitoring devices particularly in areas not well covered by existing on-ground weather stations and where weather patterns are distinct from those in their wider surroundings. To explore how such demand might be met, an Internet-connected remote weather station powered by battery with solar energy harvesting was designed, built and tested in this project. Main objectives were to develop an alternative to commercially available weather stations which provides Internet connectivity and reliable performance at a lower cost. The developed system focuses on detecting wind speed and direction and also measures temperature and humidity. Measurements are accessible online in real-time and measurement intervals can be set to vary according to time of day (based on ambient light levels). WiFi connection was used after other options were ruled out due to either cost or reliability. Power supply issues were encountered and mitigated in part by adjusting various elements of the system.

Keywords: Internet of Things; IoT; Weather Station; Remote Weather Monitoring; Wind Speed; Wind Direction; Temperature; Humidity.

1 INTRODUCTION

1.1 Description of Problem Targeted by Project

Monitoring of the natural environment has always been of interest to human civilisation. Weather, in particular, affects our day-to-day lives and can also affect our health and business interests. Remote sensing from aerial vehicles and satellites, combined with supercomputing resources, provide us with readily available and fairly accurate weather forecasts and current condition snapshots, while on-ground weather stations fill in the gaps by providing ongoing measurements of actual conditions as opposed to predictions.

It is well-known that local weather systems and effects can create microclimates which may result in significantly different wind patterns in some areas than are predicted from the abovementioned remote sensing. As a result, on-ground weather stations are a very valuable tool for understanding local weather systems. Unfortunately, there may not be a weather station in the particular area about which detailed weather observations are desired. This issue is especially acute in developing countries, where there are fewer weather stations than are found in developed countries. Take Bali, Indonesia for example, there are only a small handful of operational weather stations on the island which make their data available on the internet [1]. Parts of the island have strong local diurnal weather effects that can lead to local winds being completely different to the prevailing trade winds blowing across the open ocean nearby. These differences tend to be strongest in the morning to mid-day. The strength and duration of the local effects are unpredictable. Some days they are weak, some days they are strong and can last for either a few hours, or a longer period such as from early morning into early afternoon, while on other days the local wind remains calm all day despite the open ocean trade winds being strong [2]. Note that while a reference has been given, the foregoing statement is primarily based on local knowledge and field experience.

1.2 Proposed Solution

For people interested to know real-time weather conditions in an area not covered by an existing on-ground weather station, one solution is to have their own weather station. Our project group set out to build such a station and deploy it, primarily for measuring wind speed and direction. Our main goals were to make a weather station that:

- is relatively inexpensive to build and operate;
- operates reliably;
- is weather-proof;
- has a low-profile appearance;
- is easy to deploy;
- is battery powered with solar energy harvesting; and
- has RF connectivity.

Part 2 of this report outlines the design developed for our weather station. Part 3 describes implementation details for our system, providing a “how to” guide for those wishing to replicate it. Part 4 discusses results from deploying our system.

2 DESIGN OUTLINE

There are some cheap consumer grade weather stations available that meet some of our criteria, however none of those have communications ability that enable them to be deployed at a remote location. Those stations typically come with a custom short range radio link connecting the outdoor station to an indoor receiver unit, and as such they are targeted to on-premises use. The cheapest of these type of systems cost around $150. More accurate, reliable and robust professional grade stations with longer range communications options have much higher price tags, costing thousands of dollars or more.

The solution proposed here is a simple, low-cost and reasonably reliable weather station that can connect to either WiFi or the telecommunications network. Although the prototype presented in our report and demonstration only has WiFi connectivity, adding GSM/CDMA/LTE connectivity would be relatively easy except the power considerations could be challenging as discussed in the “Power” section in Part 3.

2.1 System Architecture

Central to our proposed solution is a simple wind speed and direction meter mounted on an elevated wooden frame. These sensors are connected to an ATMega328P microcontroller unit (**MCU**) placed inside an insulated weather-proof enclosure. The enclosure also houses a 3.7V lithium polymer (**LiPo**) battery, a 5V solar panel, a solar/battery power manager/regulator that controls battery charging from the solar panel as well as supplying a constant 5V to the MCU and the peripherals. The peripherals consist of a 5V to 3.5V DC/DC converter and an ESP8266 WiFi module board. The sensor regularly connects to a web server and posts measurements which are stored in a database. The data can then be accessed by the end user via a web browser dashboard. This architecture is shown in Figure 1.

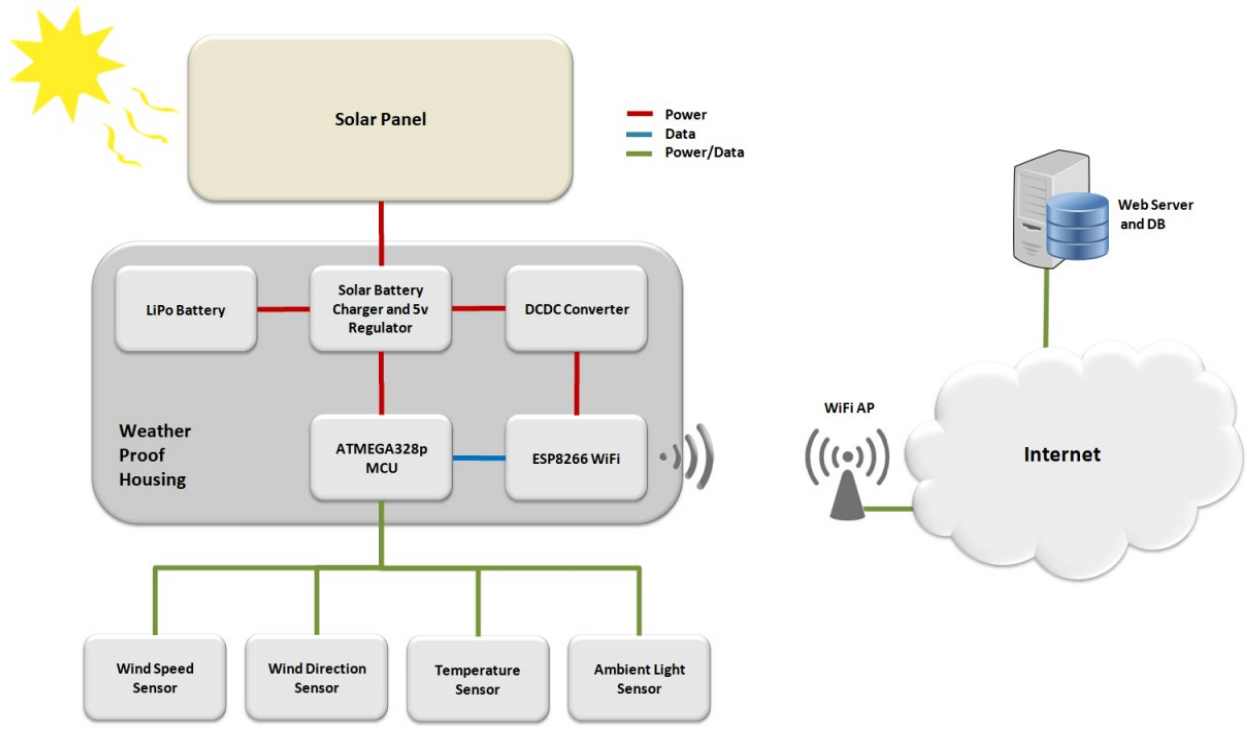

Figure 1. Schematic for IoT weather station.

2.2 Operation

We set the MCU to have three phases as follows and to cycle through these indefinitely:

(1) The MCU powers on the sensors and takes readings for around 30 seconds. The current draw in this phase is around 30mA.

(2) The MCU powers off the sensors, powers up the WiFi board and sends data to the server for around 30 seconds. The current draw in this phase is around 150mA.

(3) The MCU powers off the WiFi board and goes into a deep sleep mode for a varying amount of time (from 5 to 15 minutes) depending on the ambient light level. The current usage in this phase is around 4mA which is significantly lower than in phases 1 and 2, thus conserving power when the sensing and transmitting components of the device are not in use.

3 IMPLEMENTATION DETAILS ("HOW TO" GUIDE)

3.1 Hardware and connections

Figure 2 depicts the hardware and connections for the entire weather station.

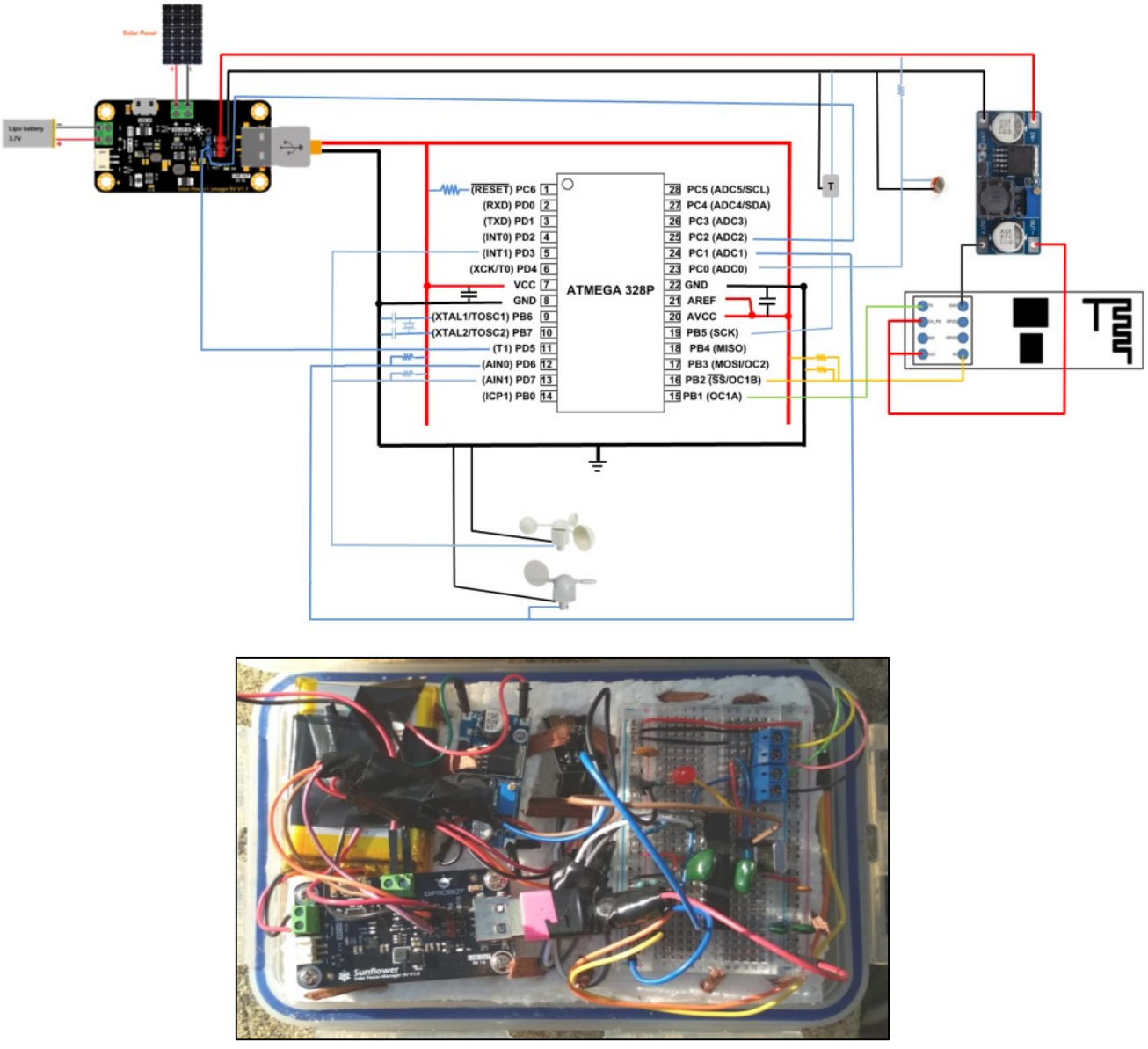

Figure 2. Schematic (top) and photograph (bottom) of the circuit.

3.2 Microcontroller

The system is controlled by a standalone ATMega328P MCU chip in a breadboard. The chip is the same as that found in an Arduino Uno board, and is used instead of a full Uno to reduce power consumption, as the Uno contains a lot of uncontrollable peripheral hardware such as the FTDI chip and voltage regulators.

We found that using an Uno the power consumption at 5V was around 50mA while active and 25mA in sleep mode, whereas using only the chip the consumption was around 30mA while active and 4mA in sleep mode.

Connections of the MCU:

- The MCU needs to be driven by a crystal oscillator with ground tied 22pF ceramic capacitors on both legs. The oscillator should be placed as close to the MCU pins as possible.
- Both pairs of VCC and GND pins also need to be decoupled by large ceramic or film capacitors to minimise power line noise. In this case 1uF capacitors were used along with smaller 100nF capacitors placed as close to the MCU pins as possible. We found that this is critical for the system to operate. The blog referenced at [2] provides more information.
- The reset pin needs to be tied to VCC through a 10K pullup resistor.
- The data Tx/Rx pins (9,10) to the ESP8266 WiFi module use software serial as the hardware serial pins (0,1) are needed for uploading sketches. The Tx pin (10) in particular needs to use a voltage divider (10k pullup resistor and 20k pulldown resistor) to drop the 5V output to 3.3V for the ESP8266. The Rx pin (9) does not need anything as it can read the incoming 3.3V from the ESP8266.
- The wind speed sensor works by closing a circuit every half turn, measured by connecting the output line from the sensor to a digital pin (with an internal 20K pullup resistor) and activating a falling level interrupt on that pin [4]. Every time the sensor circuit closes pulling the pin to ground, the interrupt would trigger and an interrupt handling routine would measure the time since the last interrupt and thus log the current wind speed.
- The wind direction sensor output line is connected to an analog input pin and tied to high via a 10K pullup resistor. For each of 16 available azimuths, the resistance of the sensor would change, thus the voltage needs to be measured and converted it to an azimuth value through some lookup functions.
- The battery level, temperature and humidity, and ambient light sensors are all connected to analog input pins and tied to high via 10K pullup resistors.
- One digital pin is used to control power the WiFi board: not to supply the power, only to signal to the regulator to activate and deactivate the power.
- One digital pin is used to control and supply power to the wind direction sensor, and another digital pin is used to supply power to the wind speed sensor: these components have very low current draw.

3.3 Uploading sketches to the MCU via an Arduino Uno

To upload sketches to the ATMega328P, an Arduino Uno (chip removed) is required and should be connected as shown in Figure 3.

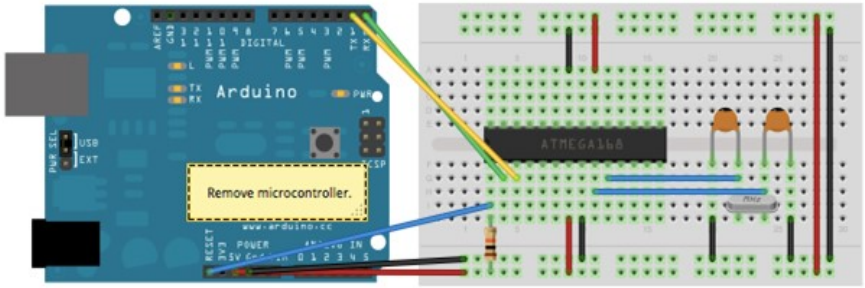

Figure 3. MCU connection for sketch uploading.

Once the board type is set to Arduino Nano instead of Arduino Uno and the processor type to ATMega328P, the sketch can be uploaded. Sometimes it may be necessary to put the removed chip back into the Uno and upload a sketch first, such as when there has been recent change of the bootloader or any of the low/high extended byte fuses, before using the aforementioned external method. Reference [5] provides a good guide.

3.4 Power

The power is supplied by a 3.7V 2000mAh LiPo battery and a small 5V 1.5W solar panel, connected to a DFRobot Sunflower battery charger and 5V regulator. This setup simultaneously charges the battery from the solar panel and provides a 5V regulated supply to the rest of the system. The Sunflower has a constant 5V regulated output as well as a user-controlled 5V regulated output that can be switched on and off via a control wire. The constant output powers the MCU and the user-controlled output powers the WiFi board via the DC/DC converter.

One of the main challenges with this project was sharing a power supply for the ATMega328P and the ESP8266. The ESP8266 drew a lot of transient current on power up before it reached its steady state of around 100-200mA. The cheap power supply did not handle this very well, resulting in voltage dips at the ATMega328P. The effect of this was unstable MCU operation, i.e. it would run for an indeterminate time (from 1 hour to 12+ hours) and then suddenly hang at which point only a hard reset or battery exhaustion

could force a reset of the MCU. The watch dog timer also became inoperative in this scenario. This issue was not completely solved during the project, however was minimised to the point that hang states where rare using the following adjustments:

- **The ESP8266** was repositioned physically to be as far away from the ATMega328P and breadboard (from around 2cm initially to around 12cm finally). This was to minimise electromagnetic energy interference to the breadboard connections. If possible the MCU would have been shielded by foil, but due to the chaotic nature of the breadboard wiring, this was not deemed practical. Obviously in a real world scenario the whole system would be on a custom PCB, leading to less of these types of issues.
- **The MCU oscillator** type was set to "full swing oscillator" which involved burning the low byte fuse [6]. We found this setting gave more voltage to the crystal oscillator causing it to oscillate to larger extremes and hence be less sensitive to electrical noise. The price for this was slightly higher power usage.
- **VCC-GND decoupling capacitors** were used on the breadboard. This helped to minimise the voltage dips experienced by the MCU.
- **The extended byte fuse was burned** to change the BOD (brown out detector) level which determines when the ATMega328P resets due to low voltage. Setting this to 4.3V helped to shed light on the actual power supply issue. The final setting was, however, the default of 2.7V.

Burning the bootloader and the fuses requires the wiring configuration shown in Figure 4 [7].

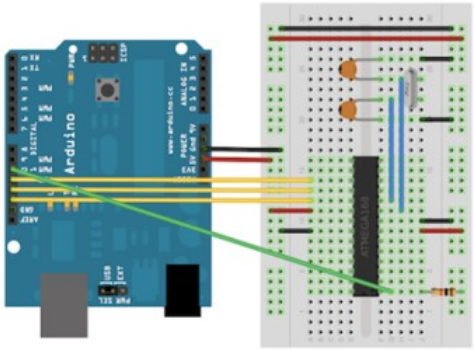

Figure 4. MCU connection to set BOD reset level.

3.5 Communications

The communications for our device are performed by the ESP8266 WiFi module. Although this tiny board is itself a microcontroller, for this task it is simply used as a slave WiFi unit. As it runs on 3.3V rather than the 5V used by the ATMega328P, the WiFi unit has to be supplied via a DC/DC converter which converts 5V to 3.3V. The data lines run between the ATMega328P and the ESP8266, enabling the ATMega328P to control the ESP8266 via AT commands to connect to a WiFi hotspot and HTTP POST the sensor readings to a web server. The readings are then processed and stored in a database for retrieval by the same web server application upon a dashboard GET request from an end user browser.

The baud rate of the WiFi unit had to be decreased from the default of 115200 bps to 9600 bps for use with software serial. The following command must be entered exactly (no spaces) to change the baud rate: "*AT+UART_DEF=9600,8,1,0,0*". Beware that if the wrong command is entered, the WiFi unit can become uncontrollable, requiring it to need re-burning of the bootloader [8]. The connections for testing AT commands (left) and burning the bootloader (right) are shown in Figure 5.

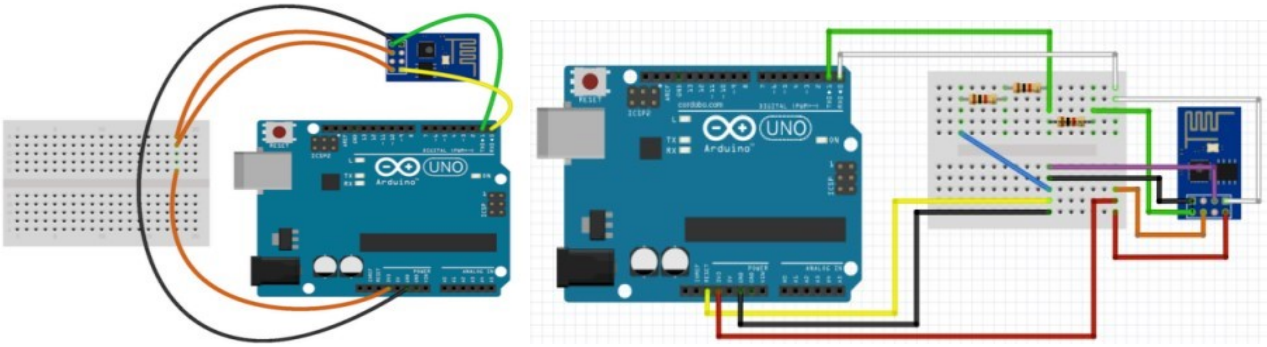

Figure 5. WiFi module connections for AT commands (left) and burning the bootloader (right).

3.6 Sensors

Our weather station uses the following sensors:

- ***Wind speed and wind direction sensors.*** These are analog devices that need to be mounted in a location free from wind obstructions (higher structures within 10m). These were the most difficult parts to obtain cheaply, we ended up ordering spare parts for a commercially available product and then cutting the rj12 connector to use the bare wires. Each sensor requires a ground and a pullup measurement wire.
  - The wind speed sensor (anemometer) closes a switch every half rotation, pulling the measurement wire voltage low temporarily every half rotation. To effect this, the wire needs to be connected to an interrupt enabled pin of the MCU as described in section 3.2 above. The interrupt triggered by this pin falling then calculates the wind speed based on the time passed since the previous interrupt.
  - The wind direction sensor varies the resistance based on the direction of the sensor, so the measurement wire voltage changes depending on the direction. This is sampled by an analog input pin periodically.
- ***Temperature and humidity sensor.*** This requires GND, VCC and a data wire. This device processes data internally and communicates via the data wire to a digital input pin.
- ***Ambient light sensor.*** This requires a ground and a pullup measurement wire.
- ***Battery level sensor.*** This requires only a measurement wire, which is a built-in function of the Sunflower power regulator.
- ***WiFi RSSI sensor.*** This is to measure received signal strength of the target WiFi AP, and requires no wire as it is measured by the WiFi module directly.

3.7 Weather-Proof Enclosure and Wind Sensor Frame

As the system needs to be deployed outdoors to measure its surroundings, a weather-proof enclosure is required. We set out to make an enclosure that is water-proof, easily serviceable, insulated against direct sunlight to avoid overheating of the MCU and battery, and is capable of having a weather-proof cable entry point. We used a premium water-tight silicone sealed click-lock Tupperware container. EPS foam was siliconed to the internal sides of the container to insulate it. A cable entry hole was burned in one side and an electrical entry boot with a downward facing pipe was siliconed in place. The solar panel was siliconed on the top of the housing, as we found that putting it inside halved the solar energy harvesting ability despite the Tupperware material being transparent.

The wind speed and direction sensors are mounted on a simple wooden frame, as shown below, to elevate them above the surrounding obstacles to wind movement.

Figure 6. Interior (left) and exterior (centre) of weather-proof enclosure, and wind sensor frame (right).

3.8 MCU Software

The software running the MCU system is a single Arduino sketch written in C (see Appendix 1). The general flow of the MCU code is as follows (elaborated on underneath Figure 7).

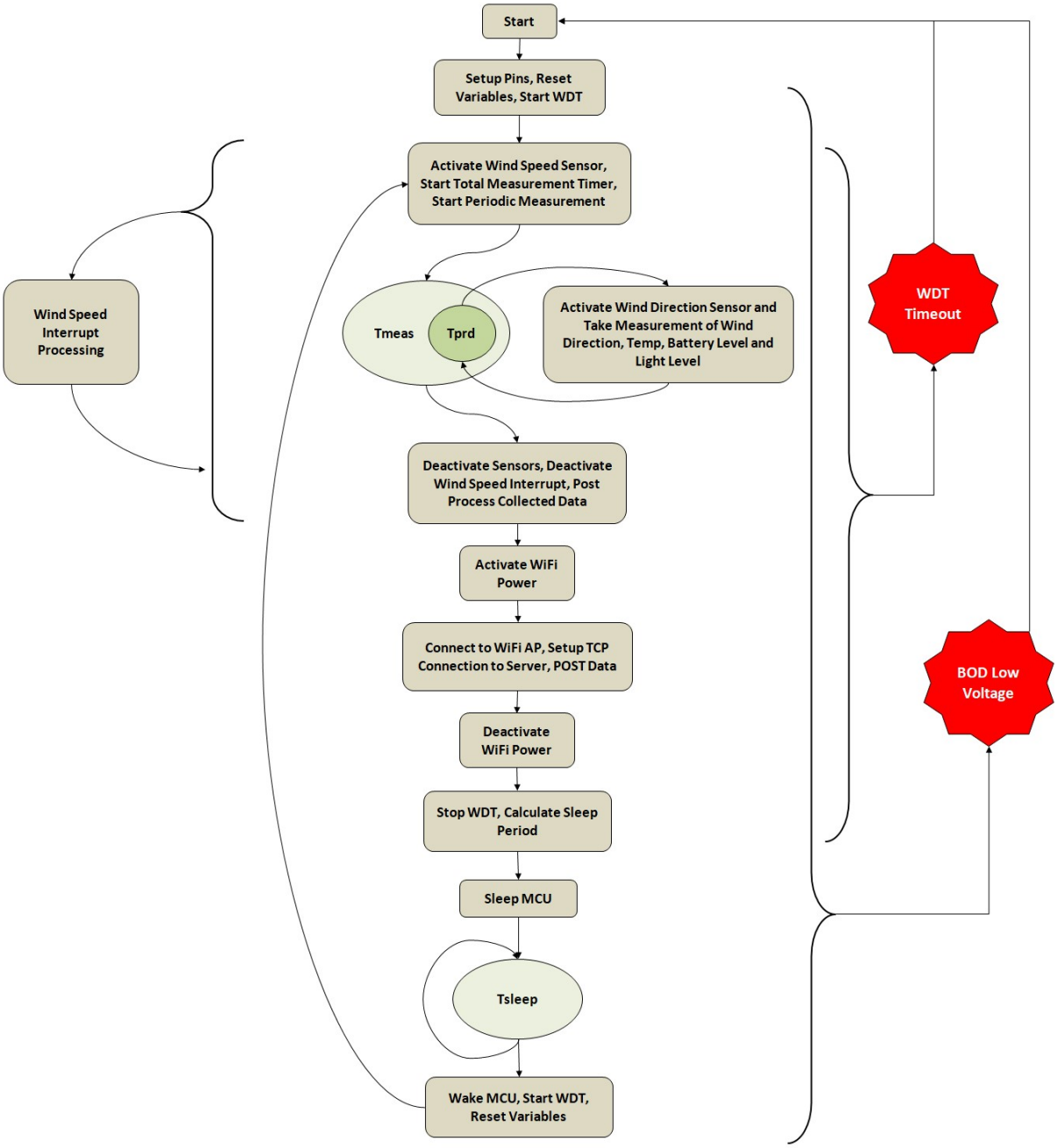

Figure 7. MCU software flow chart.

- Startup – Define the pins, reset the variables, and activate the watchdog timer (**WDT**) interrupt
- Measurement cycle
  - Activate the wind speed interrupt
  - Start the measurement cycle main timer
  - Start the periodic measurement timer
  - When the periodic measurement timer expires, a measurement is taken from the wind direction sensor as well as the battery, temperature and humidity, and light level sensors
  - WDT interrupt fires every 8 seconds, the program continues while the WDT total time is below the timeout threshold and when the threshold is reached the MCU is hard reset
  - Wind speed interrupt: during the measurement cycle if the sensor closes the circuit on a half rotation, it triggers the wind speed interrupt which is handled by an interrupt service routine (**ISR**)
  - When the main measurement timer expires the MCU deactivates the wind speed interrupt and exits the measurement cycle
- Post processing – After measuring the data, the MCU post processes it to prepare for transmission
- Data communications
  - WiFi module is powered on
  - Connect to the target AP
  - Set up TCP connection to the target server

- Send HTTP POST request with the data payload
- WiFi module is powered off

- Prepare to sleep the MCU
  - Stop the WDT interrupt
  - Calculate the sleep period from the ambient light level
  - Sleep the MCU for the given sleep period
- Wake the MCU reset the counters start the WDT
- Go back to “measurement cycle” above

3.9 Web Server and Database

The web server and database could be anywhere and on any platform. As a Raspberry Pi was available for the project, we connected it to a WiFi hotspot, enabled port forwarding on the AP, and set up a flask web server along with an SQLite database on the Raspberry Pi. As the particular AP used to host the server had a fixed IP, we could access this from the internet. The software for the web server is simply a flask application running on Python. SQLite comes with Python and enables databases and tables to be created and accessed from the command line. Appendix 2 sets out the actual code for this aspect of our project.

4 RESULTS AND DISCUSSION

4.1 Power Harvesting

As shown in the graph in Figure 8, a small 1.5W solar panel has enough harvesting ability to run the sensor unit indefinitely. The battery voltage in the figure is the terminal voltage of the LiPo battery, which has a low voltage cut-off of around 2.8V, below which level the battery shuts off to avoid damage. At full charge the battery reaches around 4.4V. Thus around 3V to 4.2V is the normal operating region indicating a level of battery charge. The voltage will drop evenly until it reaches around 3V below which it starts dropping fast.

Testing over a three-day period (4 to 6 October 2019) found that:

- On day 1, which was stormy with heavy cloud cover, a brief period of sunshine charged up the battery above 4V, and that battery level was maintained for the rest of the day which had very heavy clouds, only dropping when night fell and the solar unit harvested no more energy.
- Day 2 had scattered clouds, and the sunlight was enough to charge the battery to almost full.
- Day 3 had less cloud cover still and a higher starting voltage, the battery was quickly charged full.

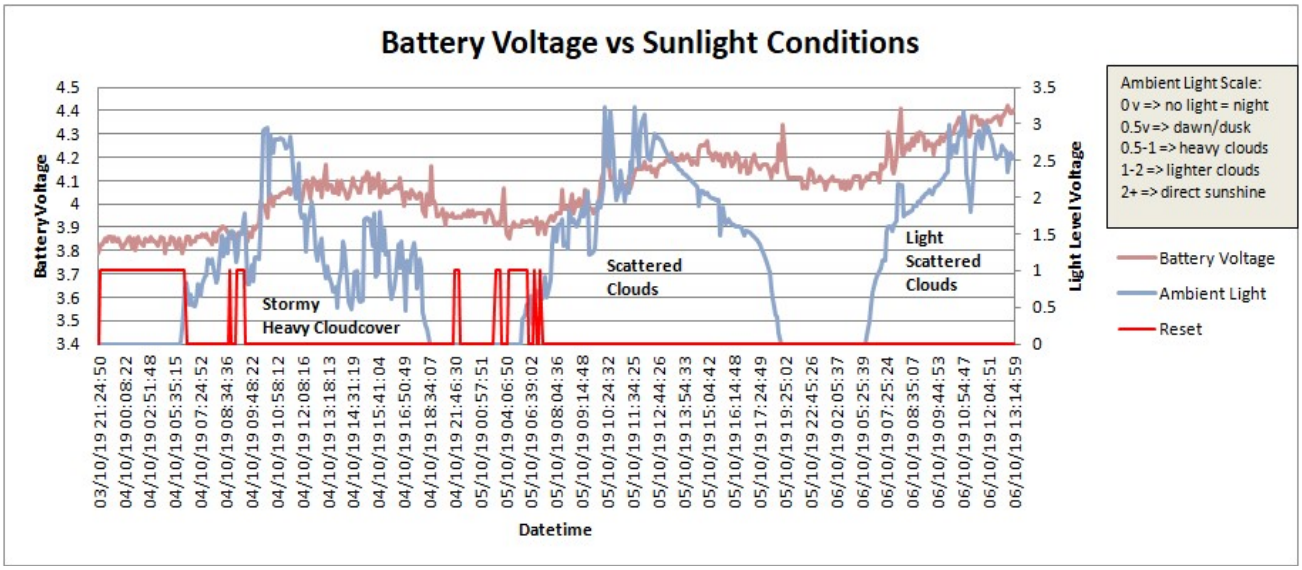

Figure 8. LiPo battery typical charge curve supplied by the DFRobot power manager.

From these results, it is clear that solar harvesting is a viable option for IoT devices and offers sufficient power to run a radio communications module if used with care (careful duty cycling). If used with a larger battery (the battery we used is very small) and/or a larger solar panel (the solar panel we used is only 8cm x 13cm), higher power RF communications could be deployed easily.

It can also be seen from the following figure that the MCU starts resetting at lower voltages, due to the current spike from the WiFi board starting up. This is a power supply issue and needs to be handled effectively for the unit to operate reliably.

4.2 Communications

We selected WiFi RF as the communications technology for our device because WiFi modules are cheap and readily available. One main issue with WiFi is that it is short range and thus only available within 20-30m of an AP. Another issue is WiFi uses ISM bands so can suffer from interference in crowded areas. For a longer range solution we found several options, these were considered but none met the cost criteria so were not pursued.

Cellular. Cellular technology is widely available and would enable the weather station to be deployed in a much larger range of areas. The issues are simply availability of cellular capable boards. 2G boards are the cheapest, but still cost upwards of $20. 3G-UMTS boards are $60+ and 4G-LTE boards are $100+. For this reason, cellular was not used for this project, though we consider would be the best choice for real world deployment.

LoRa. We had a discouraging experience of trying to connect a LoRa shield to an Arduino Uno and then connect to The Things Network (**TTN**), and concluded this technology is not appropriate for our project. The main issue we see is that TNN is a shared best effort network and its service quality is very patchy. Sometimes messages get though and sometimes they do not, despite having consistently good RF. For LoRa to be effective, one would be forced to buy a LoRa AP and then do the RF engineering to ensure a good link between the sensor system and the AP. All of this would end up costing more money and time than we had for the project. Furthermore, using a LoRa type technology forces one to use very compact data payload formats. This is good for not affecting network capacity when many devices connect, but is also troublesome. Using WiFi allowed us the luxury of simply sending as much ASCII format data as was desired, hence it was easier to implement.

4.3 Total Cost

We had a goal of creating an inexpensive device. Unavailability of the main sensor made this a difficult goal. The total hardware cost of our system is ~ $140 comprising the below items:

Description	Cost (approximate)
MCU + ancilliary equipment	$15
ESP8266 01 WiFi Board	$2
DFRobot Sunflower power manager	$12
DC/DC converter	$8
Battery	$15
Solar panel	$10
Wind sensors	$70 (note this is 50% of total cost)
Mounting and weather-proofing hardware	$10
Total	**$140**

Table 1. Cost Breakdown

To reduce the cost, the whole system could be designed into a custom PCB (printed circuit board), which would then require only a separate battery and solar panel. This would also enable the use of a smaller enclosure. The issue of obtaining wind sensors may be harder to solve. Building them from scratch may be an effective way of reducing cost, possibly using 3D printing [9]. Ultrasonic technology could also be investigated for wind speed and direction.

4.4 Mounting Location

Given the system incorporates wind sensors, it is very sensitive to mounting location. The sensor needs to be mounted above nearby obstacles and at least 10m away from larger structures, ideally much further. Otherwise, the wind speed and direction measurements will be inaccurate due to eddies from the wind moving around solid structures.

4.5 Data

The optimal way to expose the data to the end user is via a web server and some kind of HTML-CSS-Javascript dashboard like the one shown in reference [10]. The data should be available via a dashboard where all the information is available in one page. The current web server only offers text based database access due to time constraints for completing the project. The following figure is a sample of the data collected over 3 days including 4 October when there was a spring storm that toppled a large tree on the UWA campus. The spike in wind speed can be seen just before 5pm at almost 40kmph.

Potential improvements would be functionality for real-time weather alerts via email, SMS or instant messaging. These alerts could be driven by simple threshold analysis or more complex predictive modelling based on continually learning models from historical data.

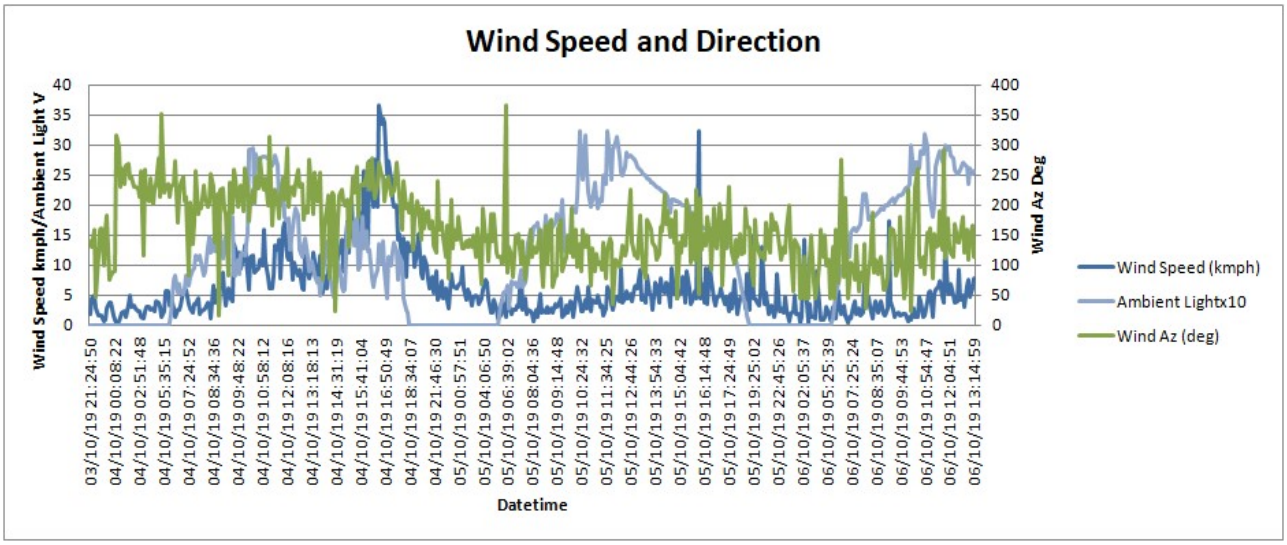

Figure 9. Wind speed and direction measurements from prototype device.

5 CONCLUSION

A main motivation for this project was to create an inexpensive device that would give the end user better knowledge of the real-time weather conditions – particularly wind speed and direction – at a target destination in order that they can plan to arrive at the location when the desired conditions are present. A successful device is likely to have a variety of consumer applications such as enabling better trip planning for visiting a faraway beach or trail for leisure, or work scheduling or risk assessment at a remote property (e.g. farm or construction site) where these tasks are affected by changes in weather conditions from day to day or at even smaller time intervals.

Through pursuing this project, we learned that making a standalone self-powered IoT field device is achievable with readily available battery and solar technology, however there are serious challenges in terms of connectivity, cost and reliability. We built a prototype device and demonstrated that it is operable at a basic level. Finding a suitable location to mount the device will be a challenge in the real world deployment phase, with not only physical constraints but also practical and legal considerations of obtaining permission from the owner of the land or structure to which one wishes the mount the device.

While small-scale IoT weather stations such as the one we built may be largely relegated to the realm of hobbyists due to the lack of perceived potential monetary gain, some of the base technologies (weather-proof enclosure, power self-sufficiency) used in our prototype could be used as a platform for developing a host of other IoT systems.

6 REFERENCES

[1] Author unknown. “WunderMap Sensor Network” search results for Bali island and surrounds, website of “Weather Underground”. https://www.wunderground.com/wundermap?lat=-8.65&lon=115.22&zoom=8&pin=&rad=1&rad.type=00Q&wxsn=0&svr=0&cams=0&sat=0&riv=0&mm=0&hur=0 (accessed Aug. 15, 2019).

[2] Author unknown (“Surfline Forecast Team”). “Mechanics: Keramas, Bali – The science behind Bali’s premier performance righthander,” May 25, 2018, website of “Surfline”. https://www.surfline.com/surf-news/mechanics-keramas-bali-indonesia-permier-performance-right-hander/26305(accessed Aug. 15, 2019).

[3] N. Gammon. “How to make an Arduino-compatible minimal board,” May 8, 2012, website of “Gammon Software solutions”. http://www.gammon.com.au/breadboard (accessed Aug. 15, 2019).

[4] K. Kessler. “Sparkfun Weather Station,” Jun. 21, 2012, website of “Kevin Kessler’s Various Projects”. https://blog.kkessler.com/2012/06/21/sparkfun-weather-station/ (accessed Aug. 15, 2019).

[5] Author unknown. “From Arduino to a Microcontroller on a Breadboard,” website of “Arduino”. https://www.arduino.cc/en/Tutorial/ArduinoToBreadboard (accessed Aug. 15, 2019).

[6] Author unknown. “ATMega328P datasheet,” website of “Sparkfun”. https://www.sparkfun.com/datasheets/Components/SMD/ATMega328.pdf (accessed Aug. 15, 2019).

[7] S. Mandal. “How to Change Fuse Bits of AVR Atmega328p - 8bit Microcontroller Using Arduino,” website of “instructables”. https://www.instructables.com/id/How-to-change-fuse-bits-of-AVR-Atmega328p-8bit-mic/ (accessed Aug. 15, 2019).

[8] Ravi (author surname unknown). “How to Update Flash ESP8266 Firmware – Flashing Official AT Firmware,” Dec. 16, 2017, website of “Electronics Hub”. https://www.electronicshub.org/update-flash-esp8266-firmware/ (accessed Aug. 15, 2019).

[9] Author unknown (“Arduino Team”. “A 3D-printed personal weather station,” website of “Arduino”. https://blog.arduino.cc/2018/04/09/a-3d-printed-personal-weather-station/ (accessed Aug. 15, 2019).

[10] Author unknown. “Weather Forecast For 10 Days” search results for Bali – Kuta Beach, website of “Windy App”. https://windy.app/forecast2/spot/331267/Bali+-+Kuta+Beach+/statistics (accessed Aug. 15, 2019).

7 SOFTWARE APPENDIX 1

The actual code is shown below in logical sections. Part 1: Setup commands

```
#include <avr/sleep.h>
#include <avr/wdt.h>
#include <SoftwareSerial.h>
SoftwareSerial esp(9, 10);  //RX,TX
#include <dht.h>
dht DHT;

#define comms_rx_pin 0
#define comms_tx_pin 1
#define temp_meas_pin 2
#define w_spd_itr_pin 3
#define w_spd_pwr_pin 6
#define w_dir_pwr_pin 7
#define status_led_pin 8
#define comms_pwr_pin 13

#define light_sensor_meas_pin A0
#define w_dir_meas_pin A1
#define bat_meas_pin A2

#define KMPHPP 2.4    //kmph per pulse of the wind speed meter
#define BOUNCE_GUARD 500   //ignore wind speed pulses shorter than this in u-secs
#define VREF 5.1    ///vref should be = 5v = Vcc


//general program vars
String result;
bool trouble_flag = false;      //used to detect bad connectivity and wait it out
bool connected_flag = false;
bool http_ok_flag = false;
int i;    //global counter var, do nto use in functions

//basic control
bool testing = false;   //set to true for testing
bool wind_test = false;   //set to true for testing
bool startup_wifi_flag = false;    //set to true for the first time in a new ssid

//for wifi
String server_ip = "xxx.xxx.xxx.xxx";   //for external
int server_port = xxxx;
String post_uri = "/data";
String ssid = "xxxxxxxxx";
String pwd = "xxxxxxxxx";

//timing vars
//NOTE: one dump cycle = (sleep_reps * 8 + meas_period_s) * meas_reps
#define meas_reps 5  //in reps how many measurement cycles to run for each dump
int meas_period_s = 6;      //how long (s) to activate the wind sensors for every measurement cycle
int sleep_reps = 6; //how many 8s reps to sleep the board each measurement cycle,
int sleep_reps_min = 6;     //in reps of 8s
int sleep_reps_max = 20;     //in reps of 8s
int sleep_reps_delta = 100;    //in reps of 8s
int dump_rep_cnt = 0;
int meas_rep_cnt = 0;             //counts how many measurement reps have taken place

//wdt vars
int wdt_timeout_s = meas_period_s*meas_reps + 200; //sensor read time + max connect time
int wdt_timeout_reps = wdt_timeout_s/8;
volatile int wdt_timeout_rep_cnt = 0;

//sleep vars
bool sleep_flag = false;

//light sensor vars
double light_sensor_thd = 0.5;   //0.5;   //in volts
double light_sensor_v = 0.0;   //in volts

//battery meter vars
double bat_v = 0.0; //in volts
bool low_bat_flag = false;
double low_bat_thd = 3.3;

//temperature and humidity vars, temp in deg.C
double temp = 0.0;
double hum = 0.0;

//vars for use in the wind speed interupt routine
```

```
volatile unsigned long t_prev = 0.0;  //time of the previous pulse
volatile int pulse_cnt = 0;   //how many pulses
volatile double w_spd[meas_reps];
double w_spd_mean = 0.0;

//wind direction vars
double w_dir_mean = 0.0;
double w_dir_x[meas_reps];
double w_dir_y[meas_reps];

//These numbers are for a 5v Vref and a 5V Vcc and a 10K pullup resistor
int w_dir_raw_range[] = {66,84,92,127,184,244,287,406,461,600,631,702,786,827,889,946};
int w_dir_az[] = {1125,675,900,1575,1350,2025,1800,225,450,2475,2250,3375,0,2925,3150,2700};
```

Part 2: the interrupt handling code for the watchdog timer and sleep as well as the wind speed sensor.

```
//ISR for wind speed
//these need volatile: w_spd[], pulse_cnt, t_prev
void on_wind_speed_pulse_interupt(){
  //this processes the wind speed pulse from the sensor, the pulses occur once per sensor rotation
  unsigned long t_now = micros();
  //check if we are starting the meas cycle again
  if (t_prev == 0){
    t_prev = t_now;
  }
  else{
    //get the duration since last function call
    unsigned long duration = t_now - t_prev;
    //only process if the duration is longer than the bounce protect, needed to filter out sensor switch
bounce
    if (duration > BOUNCE_GUARD) {
      t_prev = t_now;   //only update t_prev if we accept the point a legit pulse
      pulse_cnt++;
      w_spd[meas_rep_cnt] += KMPHPP/((double)duration/1000000.0);
    }
  }
}

//ISR for wdt and for sleep
ISR (WDT_vect) {
  if (!sleep_flag) {
    //this is for the watchdog timer case
    wdt_reset();   //this should reset the curretn timer with all settings, not sure if it resets the
settings too
    wdt_disable();
    wdt_timeout_rep_cnt++;
    //Serial.println("Interrupt: " + (String)wdt_timeout_rep_cnt + " of " + (String)wdt_timeout_reps + "
Result = " + (String)(wdt_timeout_rep_cnt > wdt_timeout_reps));
    if (wdt_timeout_rep_cnt > wdt_timeout_reps){
      if (testing) Serial.println("TIMEOUT RESTARTING!!!");
      digitalWrite(comms_pwr_pin, LOW);
      wdt_enable(4); //we have timedout, so we need to reset
    }
    else{
      wdt_enable(9);   //set stadnard 8s wdt
      set_int_8s_no_reset();   //modify the 8s wdt
    }
  }
}

void set_int_8s_no_reset() {
  //you need to run this after you have enabled wdt to ensure you disable reset, if you do not, you will
reset on timer expiry
  //WDIE = 1: Interrupt Enable
  //WDE = 1 :Reset Enable
  //WDP3 = 0 :For 1000ms Time-out
  //WDP2 = 1 :For 1000ms Time-out
  //WDP1 = 1 :For 1000ms Time-out
  //WDP0 = 0 :For 1000ms Time-out
  //WDCE=4,WDE=3,WDIE=6,WDP3=5,WDP2=2,WDP1=1,WDP0=0
  // same settings for wdt and sleep => 8s, interrupts and no reset
  cli();
  wdt_reset();
  MCUSR &= ~(1<<WDRF);
  WDTCSR |= (1<<WDCE) | (1<<WDE);  // Set WDCE and WDE to enable changes.
  WDTCSR = (1<<WDIE) | (0<<WDE) | (1<<WDP3) | (0<<WDP2) | (0<<WDP1) | (1<<WDP0);
  sei();
}

//sleep stuff
//####################################################################
```

```
void sleep_mcu(int reps){
  if (testing) Serial.println("starting sleep");
  int j=0;
  sleep_flag = true;
  for (j=0;j<reps;j++){
    set_sleep_mode(SLEEP_MODE_PWR_DOWN); // Set sleep mode.
    cli();
    sleep_enable();
    set_int_8s_no_reset();
    sleep_bod_disable();
    sei();
    sleep_cpu();
    //######################################
    // After waking from watchdog interrupt the code continues to execute from this point.
    //######################################
    sleep_disable();
  }
  sleep_flag = false;
  if (testing) Serial.println("sleep finished");
}
```

Part 3: Variable reset functions and some trig helper functions

```
void reset_vars_dump_cycle(){
  //reset important variables after each data dump to WiFi
  int k;
  meas_rep_cnt=0;
  for (k=0;k<meas_reps;k++){
    w_spd[k] = 0.0;
  }
  for (k=0;k<meas_reps;k++){
    w_dir_x[k] = 0.0;
    w_dir_y[k] = 0.0;
  }
  temp = 0.0;
  hum = 0.0;
  bat_v = 0.0;
}

double correct_deg(double deg){
  if (deg < 0){
    return deg + 360.0;
  }
  if (deg >= 360){
    return deg - 360.0;
  }
  return deg;
}

double rad2deg(double rad){
  return correct_deg(rad*(180.0/M_PI));
}

double deg2rad(double deg){
  return M_PI*deg/180.0;
}
```

Part 4: Measure the Wind Direction, Temperature, Battery Level and Light Level

```
void read_wind(){
  if (testing) Serial.print("Doing Wind...");
  double az_deg = -1.0;
  int k = 0;

  //start measurements
  //activate wind speed sensor
  pinMode(w_spd_pwr_pin, INPUT_PULLUP);
  delay(100);
  attachInterrupt(digitalPinToInterrupt(w_spd_itr_pin), on_wind_speed_pulse_interupt, FALLING);
  pulse_cnt = 0;
  t_prev = 0;

  //turn on the wind direction sensor
  digitalWrite(w_dir_pwr_pin, HIGH);
  delay(500);
  int tmp_wait = 200;    //ms between each successive wind dir meas
  int tmp_cnt = 0;
  double w_dir_raw = 0;
  w_dir_raw=0.0;
  for(k=0; k<1000*meas_period_s/tmp_wait; k++){
    w_dir_raw += (double)analogRead(w_dir_meas_pin);
```

```
    delay(tmp_wait);
    tmp_cnt++;
  }
  //turn off the wind direction sensor
  digitalWrite(w_dir_pwr_pin, LOW);

  //deactivate the wind speed sensor
  detachInterrupt(digitalPinToInterrupt(w_spd_itr_pin));
  pinMode(w_spd_pwr_pin, OUTPUT);
  digitalWrite(w_spd_pwr_pin, LOW);

  //process the wind speed results
  //################################################
  if (pulse_cnt == 0) w_spd[meas_rep_cnt] = 0;  //test if wind speed is too low to accurately measure
  else w_spd[meas_rep_cnt] = w_spd[meas_rep_cnt]/(double)pulse_cnt;  //if ok

  //process the wind direction measurements
  //################################################
  w_dir_raw = w_dir_raw/(double)tmp_cnt;

  //this looks for the closest match, if it finds one within 4 points of the nominal, it goes for it straight
 away,
  //otherwise, it just finds the closest and returns that
  int min_delta=1024;
  int min_index=8;
  for (k=0;k<16;k++){
    int delta = min(abs(w_dir_raw - w_dir_raw_range[k]), abs(w_dir_raw+1023 - w_dir_raw_range[k]));
    if (delta < 4){
       az_deg = (double)w_dir_az[k]/10.0;
       break;
    }
    if (delta < min_delta){
      min_delta = delta;
      min_index = k;
    }
  }
  if (az_deg == -1.0){
    az_deg = (double)w_dir_az[min_index]/10.0;
  }

  //process the w_dir results
  w_dir_x[meas_rep_cnt] = sin(deg2rad(az_deg));
  w_dir_y[meas_rep_cnt] = cos(deg2rad(az_deg));

  if (testing) Serial.println((String)az_deg + ", " + (String)w_spd[meas_rep_cnt]);
}

void read_bat_lev(){
  int bat_meas = 0;
  bat_meas = analogRead(bat_meas_pin);
  bat_v += (double)VREF*((double)bat_meas+1.0)/1024.0;
}

void read_temp_hum(){
  //read the raw value
  DHT.read11(temp_meas_pin);
  temp += DHT.temperature;
  hum += DHT.humidity;
}
```

Part 5: Post Process the measured data, does wind speed weighted wind direction vector addition here

```
void pre_dump_processing(){
  int k;

  //this does the wind speed
  w_spd_mean=0;
  for (k=0;k<meas_reps;k++){
    w_spd_mean += w_spd[k];
  }
  w_spd_mean = w_spd_mean/(double)meas_reps;

  double w_dir_x_mean = 0;
  double w_dir_y_mean = 0;
  w_dir_mean=0;
  //this does the wind dir, mean weighted by w_spd
  for (k=0;k<meas_reps;k++){
    if (w_spd_mean != 0.0) {
      //do an weighted mean
      w_dir_x_mean += w_dir_x[k]*w_spd[k];
      w_dir_y_mean += w_dir_y[k]*w_spd[k];
```

```
    }
    else{
      //do an unweighted mean
      w_dir_x_mean += w_dir_x[k];
      w_dir_y_mean += w_dir_y[k];
    }
  }
  w_dir_x_mean = w_dir_x_mean/(double)meas_reps;
  w_dir_y_mean = w_dir_y_mean/(double)meas_reps;
  w_dir_mean = rad2deg(atan2(w_dir_x_mean, w_dir_y_mean));

  //this does the temp/hum/batt
  temp = temp/(double)meas_reps;
  hum = hum/(double)meas_reps;
  bat_v = bat_v/(double)meas_reps;
}
```

Part 6: Adjust the sleep period

```
void adjust_sleep_cycle_length(){
  //handles low battery states
  if (bat_v < low_bat_thd) low_bat_flag = true;
  else if (bat_v > (low_bat_thd + 0.2)) low_bat_flag = false;

  // read the value from the sensor, returns int from 0-1023 maps from 0 to vref(5v):
  int light_sensor_meas = analogRead(light_sensor_meas_pin);

  //normal case
  if (!trouble_flag & !low_bat_flag){
    //adjust the sleep time based on the abient light, longer for lower light, shorter for more light
    light_sensor_v = (double)VREF*((double)light_sensor_meas + 1.0)/(double)1024;
    if (light_sensor_v <= light_sensor_thd){
      //sleep_reps = min(sleep_reps_max, sleep_reps + sleep_reps_delta);
      sleep_reps = sleep_reps_max;
    }
    if (light_sensor_v > light_sensor_thd){
      //sleep_reps = max(sleep_reps_min, sleep_reps - sleep_reps_delta);
      sleep_reps = sleep_reps_min;
    }
    //helps battery charge
    if (bat_v < 4.0){
      sleep_reps = sleep_reps*2;
    }
  }
  else {
    //connectivity trouble case
    if (trouble_flag & !low_bat_flag){
      int mult_factor = 2;
      //we increase the sleep to max quickly
      //sleep_reps = min(sleep_reps_max, sleep_reps + mult_factor*sleep_reps_delta);
      sleep_reps = sleep_reps_max;
      if (bat_v < 4.0){
        sleep_reps = sleep_reps*2;
      }
    }
    //low battery case
    else{
      sleep_reps = 3600/8/meas_reps;   //1 hour sleep as low_bat_flag is active
      for (int k=0;k<4;k++){
        digitalWrite(status_led_pin, HIGH);
        delay(1000);
        digitalWrite(status_led_pin, LOW);
        delay(1000);
      }
    }
  }
}
```

Part 7: WiFi handling functions => Setup, Connect, Post, Wait for Response

```
void reset_wifi() {
  if (testing) Serial.println("resetting wifi board");
  esp.println("AT+RST");
  get_result_fast(5000);
}

void set baud_rate(){
  //use this to set the baud rate
  digitalWrite(comms_pwr_pin, HIGH);
  esp.begin(115200);  //this assumes default baud rate is used by the module
  delay(1000);
```

```
  esp.println("AT+UART_DEF=9600,8,1,0,0");
  delay(1000);
  esp.end();
  delay(1000);
  esp.begin(9600);
  delay(3000);
  digitalWrite(comms_pwr_pin, LOW);
}

int setup_wifi_full(){
  //test the cipmux setting
  if (testing) Serial.println("setting up wifi board");
  esp.println("AT+CIPCLOSE");
  get_result_fast(500);
  esp.println("AT+CIPMUX=0");   // 0 for single TCP connection mode, 1 for multi TCP connection mode
  get_result_fast(500);
  esp.println("AT+CIPMODE=0");   // 0 for normal transfer mode
  get_result_fast(500);

  //these commands are stored in memory, so you only need once
  esp.println("AT+CWMODE_DEF=1");   // 1 for wifi client mode
  get_result_fast(500);
  esp.println("AT+CWAUTOCONN=1");   //set to 0 to not auto connect to ap on power on
  get_result_fast(500);

  //only need this once to set params, then do not need
  //board will automatically connect to the stored hotspot after this
  connect_wifi();

  //all ok
  return 1;
}

int setup_wifi_fast(){
  //test the cipmux setting
  if (testing) Serial.println("setting up wifi board");
  esp.println("AT+CIPCLOSE");
  get_result_fast(100);
  esp.println("AT+CIPMUX=0");   // 0 for single TCP connection mode, 1 for multi TCP connection mode
  get_result_fast(100);
  esp.println("AT+CIPMODE=0");   // 0 for normal transfer mode
  get_result_fast(100);
  //all ok
  return 1;
}

int connect_wifi() {
  if (testing) Serial.println("Disconnecting from current AP");
  esp.println("AT+CWQAP");
  get_result_fast(3000);

  //connect to the desired ssid
  if (testing) Serial.println("Connecting to target AP");
  esp.println("AT+CWJAP_DEF=\"" + ssid + "\",\"" + pwd + "\"");
  get_result_slow(4000, 3);
  if (result.indexOf("OK")!=-1 | result.indexOf("WIFI CONNECTED")!=-1){
    if (testing) {
      Serial.println("Connect OK");
      Serial.println("result: " + result);
    }
    return 1;    //connect ok
  }

  //could not connect error
  if (testing) {
    Serial.println("Could Not Connect Error");
    Serial.println("result: " + result);
  }
  return 0;
}


int http_post(){
  /*
  //list all available APs rssi and ch .. for testing
  esp.println("AT+CWLAPOPT=1,127");
  delay(1000);
  esp.println("AT+CWLAP");
  esp.println("AT+CIFSR");
  get_result_fast(5000);
  Serial.println(result);
```

```
  */

  //get the RSSI
  String rssi = "-1";
  esp.println("AT+CWJAP?");
  get_result_fast(1000);
  if (result.indexOf(",-")!=-1){
    rssi = result.substring(result.indexOf(",-")+1, min(result.length(),result.indexOf(",-")+5));
    rssi.trim();
  }

  //prepare the data payload
  String data_vals = "wspd_mean=" + (String)w_spd_mean +
              "&wdir_mean=" + (String)w_dir_mean +
              "&temp=" + (String)temp +
              "&hum=" + (String)hum +
              "&rep_cnt=" + (String)dump_rep_cnt +
              "&light_lev=" + (String)light_sensor_v +
              "&batt_lev=" + (String)bat_v +
              "&rssi=" + rssi;
  if (testing) Serial.println(data_vals);

  //start a TCP connection.
  if (testing) Serial.println("Establish TCP");
  esp.println("AT+CIPSTART=\"TCP\",\""+server_ip + "\","+server_port+",0");  //last value is the TCP keep ali
ve counter in s, 0 to disable
  get_result_fast(3000);   //5000;
  if (testing) Serial.println(result);
  if (result.indexOf("ERROR") != -1){
    return 0;
  }

  //test that we activated the TCP or it is already activated, then we are good to proceed
  if (result.indexOf("OK")!=-1 || result.indexOf("ALREADY")!=-1){
    String cmd1 = "POST " + post_uri + " HTTP/1.1";
    String cmd2 = "Host: " + server_ip + ":" + (String)server_port;
    String cmd3 = "Content-Type: application/x-www-form-urlencoded";
    String cmd4 = "Content-Length: " + (String)data_vals.length();
    String cmd5 = "";

    //need to add the carriage return and new line (2 chars onto each line)
    int len = cmd1.length() + 2 +
              cmd2.length() + 2 +
              cmd3.length() + 2 +
              cmd4.length() + 2 +
              cmd5.length() + 2 +
              data_vals.length() + 2;

    //determine the number of caracters to be sent.
    if (testing) Serial.println("Sending HTTP Msg");
    esp.print("AT+CIPSEND=");
    esp.println(len);
    //flush rx buffer
    get_result_fast(1000);   //2000;

    //send request
    esp.println(cmd1);
    delay(300);
    esp.println(cmd2);
    delay(300);
    esp.println(cmd3);
    delay(300);
    esp.println(cmd4);
    delay(300);
    esp.println(cmd5);
    delay(300);
    esp.println(data_vals);
    delay(3000);

    //close the connection
    esp.println("AT+CIPCLOSE");
    get_result_fast(1000);   //5000;
    if (result.indexOf("ERROR")==-1){
      if (testing) Serial.println(result);
      return 1;    //was send ok
    }
  }

  //send not successfull
  return 0;    //bad sending result error
}
```

```
//gets the result from the serial rx in difficult cases
String get_result_slow(int del, int cnt){
  int k;
  String res_new;

  result = "";
  delay(del);
  if (trouble_flag) delay(15000);
  for (k=0;k<20;k++){
    res_new = esp.readString();
    if (res_new == "") {
      cnt = cnt - 1;
      if (cnt == 0) break;
    }
    result += res_new;
    delay(1000);
  }
}

//gets the result from the serial rx
void get_result_fast(int start){
  result = "";
  delay(start);
  while(esp.available()){
    result = esp.readString();
    delay(500);
  }
}
```

Part 8: The actual Setup and Loop Functions to tie everything together

```
void setup()
{
  analogReference(VREF);

  if (testing) {
    Serial.begin(9600);
    Serial.println("############START#############");
  }
  pinMode(status_led_pin, OUTPUT);

  //indicator led on startup
  for (i=0;i<10;i++){
    digitalWrite(status_led_pin, HIGH);
    delay(100);
    digitalWrite(status_led_pin, LOW);
    delay(100);
  }

  //temperature setup
  DHT.read11(temp_meas_pin);
  delay(2000);

  //wind sensor setup
  pinMode(w_spd_pwr_pin, OUTPUT);
  digitalWrite(w_spd_pwr_pin, LOW);    //off the sensor initally
  pinMode(w_dir_pwr_pin, OUTPUT);
  digitalWrite(w_dir_pwr_pin, LOW);    //off the sensor initally

  //setup wdt
  wdt_enable(9);
  set_int_8s_no_reset();
  wdt_timeout_rep_cnt = 0;    //start the watchdog timer

  //wifi setup
  pinMode(comms_pwr_pin, OUTPUT);
  digitalWrite(comms_pwr_pin, LOW);
  esp.begin(9600);
  if (startup_wifi_flag){
    digitalWrite(comms_pwr_pin, HIGH);
    delay(1000);
    setup_wifi_full();
    digitalWrite(comms_pwr_pin, LOW);
    delay(1000);
  }
  wdt_timeout_rep_cnt = 0;    //start the watchdog timer
}

//#############################################################################
void loop() {
```

```
//test for a low battery situation
bat_v = 0;
for(i=0;i<3;i++){
  read_bat_lev();
  delay(200);
}
bat_v = bat_v/3.0;
if (testing) Serial.println(bat_v);
adjust_sleep_cycle_length(); //set the sleep reps based on the ambient light level and battery level
if (!low_bat_flag){
  //we have good battery voltage, so carry on
  reset_vars_dump_cycle();
  if (testing) Serial.println("Starting Measurements");
  for (i=0; i<meas_reps; i++){
    //#####DO MEASUREMENTS##################
    digitalWrite(status_led_pin, HIGH);
    delay(50);
    digitalWrite(status_led_pin, LOW);

    if (!wind_test) read_wind();
    else delay(2000);
    read_temp_hum();
    read_bat_lev();
    meas_rep_cnt++;
  }

  //######PROCESS AND DUMP############
  //################################################3
  //process measurements
  if (testing) Serial.println("Data Processing");
  pre_dump_processing();
  if (wind_test){
    //just randomly make up some numbers for testing
    w_spd_mean = (double)random(0,30);
    w_dir_mean = (double)random(0,359);
  }

  //################################################3
  //activate the comms and send
  if (testing) Serial.println("Activating Comms");
  digitalWrite(comms_pwr_pin, HIGH);
  delay(2000);  //give it time to connect
  setup_wifi_fast();

  //try to connect 3 times, then go into connect trouble mode where we wait with longer sleep intervals
  connected_flag = false;
  http_ok_flag = false;
  if (!trouble_flag){
    //try to post
    for (i=0;i<5;i++){
      if (http_post() == 1){
        http_ok_flag = true;
        break;
      }
    }
    if (!http_ok_flag){
      //could connect but could not send data
      trouble_flag = true;
      if (testing) Serial.println("HTTP fail, Trouble Flag Has Been Activated");
    }
  }
  else{
    //we are in trouble mode, so try only once and then give up
    if (connect_wifi() == 1){
      if (http_post() == 1){
        trouble_flag = false;
        if (testing) Serial.println("Trouble Flag Has Been Deactivated");
      }
    }
  }
  if (testing) Serial.println("Deactivating Comms");
  digitalWrite(comms_pwr_pin, LOW);
  delay(100);
  //increment dump rep count to know how many dump reps were done before the board died and reset itself
  dump_rep_cnt++;
}

//sleep the board
if (testing) Serial.println("Sleeping the Board");
wdt_disable();   //disable the wtc function
```

```
  delay(300);
  sleep_mcu(sleep_reps*meas_reps);   //go to sleep
  //##############################################3
  //###SLEEPING HERE###
  //##############################################3
   //prep the board after wakeup
  digitalWrite(status_led_pin, HIGH);
  delay(300);
  digitalWrite(status_led_pin, LOW);

  digitalWrite(comms_pwr_pin, LOW);
  delay(500);
  wdt_enable(9);
  set_int_8s_no_reset();
  wdt_timeout_rep_cnt = 0;    //start the watchdog timer
}
```

---

8 SOFTWARE APPENDIX 2

These are the SQL and data handling functions running on the RP.

```
from flask import Flask, url_for, escape, g, request, render_template, send_from_directory, jsonify
from flask_cors import CORS, cross_origin
import sqlite3
import json
from datetime import datetime as dt

def run_sql(db, sql, get_response=False, header=False):
    response = 'ok'
    try:
        with sqlite3.connect(db) as db_conn:
            cur = db_conn.cursor()
            cur.execute(sql)
        if get_response:
            response = cur.fetchall()

        if header:
            cols = [tuple([description[0] for description in cur.description])]
            return cols + response
        else:
            return response
    except Exception as e:
        return 'error: ' + str(e)

def data_processor(data, schema):
    #converts the incoming dict to a string for db update
    #check data format is ok
    if len(data) != len(schema_list): return None
    #note: dt.now() gives the current time, dt.utcfromtimestamp(0) gives the time at 0:0:0 1/1/1970
    dt_now = dt.now()
    datetime_str = dt_now.strftime('%d/%m/%y %H:%M:%S')
    secs_since_1970 = int((dt_now - dt.utcfromtimestamp(0)).total_seconds())
    data_str = '{0},"{1}"'.format(secs_since_1970, datetime_str)
    for item in schema:
        if item not in data: return None
        data_str += "," + str(data[item])
    return data_str
```

This creates the web app and defines the homepage

```
app = Flask(__name__, static_url_path='')
CORS(app)
db_path = '/home/pi/weather_ap/weather.db'
schema_list = ['wspd_mean', 'wdir_mean', 'temp', 'hum', 'rep_cnt', 'light_lev', 'batt_lev', 'rssi']

@app.route('/')
def homepage():
```

```
    return send_from_directory('static', 'index.html')
This is to handle POST requests from the MCU
@app.route('/data', methods = ['POST'])
def rx_data():
    if request.method != 'POST':
        return 'nok'

    #this will read in all the post data as a dict
        data = request.form
    print(data)
    #convert it to a string
    data_str = data_processor(data, schema_list)
    if data_str is None: return 'nok'
    #build the sql command and send it
    sql = 'insert into data values({0});'.format(data_str)
    response = run_sql(db_path, sql, get_response=False, header=False)
    return response
```

This is to handle user requests from the database

```
#usage: http://ip:port/db/24 ... or use last for the most recent value
@app.route('/db/<hrs>')
def show_db(hrs):
    if hrs == 'last':
        sql = 'SELECT * FROM data ORDER BY datetime_sec DESC LIMIT 1;'
    else:
        #this will show the data for the last N hours
        secs_since_1970 = int((dt.now() - dt.utcfromtimestamp(0)).total_seconds())
        time_thd = secs_since_1970 - 3600*int(hrs)
        sql = 'select * from data where datetime_sec > {0};'.format(time_thd)
    response = run_sql(db_path, sql, get_response=True, header=True)
    return render_template("sql_select.html", data = response);
```

This actually runs the flask webserver

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```