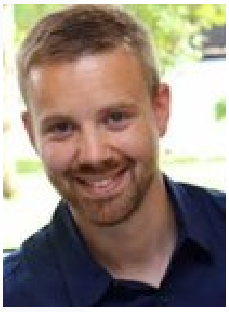
Anemometer

IOT windspeed measuring device

16th. of May 2021

Authors:



Kim Holmberg Christensen

s181554

Jørgen Drelicharz Greve

s181519

DTU - ELEKTRO

62547 - Embedded C/C++ Smart Applications

Contents

1 Introduction

1.1 Problem statement

The problem that we would like to solve is having a way to tell how windy it is outside right now. This could be done in a number of ways but the way we would like to do it is with the help of technology in the form of microcontrollers and other electronic devices. And preferably we would like to have that data available to us everywhere we go.

The first challenge we face is a sensor or device that can measure the current wind force also called an anemometer. This is a device that most often has either a propeller or scoops that can be moved by the wind and then some device that can measure that movement. In our case that device should be something that can communicate the movement to a PIC microcontroller.

The next couple of challenges we face is that the anemometer should be placed in a spot where the wind can hit it unobstructed. Often that means that the distance from a device that can communicate with the world wide web, a.k.a a router, is too great for wifi to reach and mains power would require a long extension lead. Therefore the anemometer will need to communicate via some kind of long distance wireless technology and preferably be powered by batteries and a solar panel.

The wireless technology mentioned above will then require a receiving device which is closer to the router like a basestation that can relay the data sent from the anemometer via the router and and out on the internet where it can be viewed on a webpage.

1.2 Methodology

Primarily our own understanding of electronics and some google research lays the basis for our choice of our anemometer design. We had the idea from the beginning that a DC motor could be used as a rotation measuring device but in our research we found that there were more frictionless methods of designing an anemometer.

2 Analysis

2.1 Anemometer

There is more than one way to engineer an anemometer. It is possible to use a small DC motor and then measure the voltage it outputs when turned by the wind, but unless you have a friction less motor a lot of energy will be lost due to friction which is why we did not choose this solution. Another way to do it could be to have a LED shine onto a photo resistor and have the rotor block the light every time it comes around. Then it would be possible to measure the time it takes to go one round form the dip in voltage over the photo resistor and then calculate the windspeed from that. Lastly and the way we chose to engineer our anemometer due to the simplicity and relatively friction less design, is to have two magnets turning around a hall sensor. One magnet with its north pole facing the sensor and one magnet with its south pole facing the sensor. When the north pole passes the sensor it outputs around 450mV and when the south pole passes it outputs about 0V. From that it is possible to time how long it takes to go one round and again to get the windspeed from that.

2.2 Rate of transmissions

An important subject in this project is the "smart application" concept which means that the system should be able to make decisions by it self. We chose that our system should be able to transmit data more often if the windspeed is high.

2.3 Internet communication

One of the primary goals of the assignment was connectivity to the internet. Since the COVID-19 situation limited our ability to access components at DTU we decided to go with *ESP8266 WIFI Wireless Module* for the transmission of measured data to the internet. This module seemed like the go to module for WiFi connectivity for many MCU projects.

2.4 Transmission between anemometer base and basestation

Working out how the data exchange between the Anemometer and the Basestation should happen a couple of soloutions was discussed.
First a wired connection was discussed, here Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver/Transmitter (UART) were the two types that came to mind. SPI communication are limited by a relatively short cable length(10 meters), so this solution was ruled out. UART are not as limited by cable length as SPI however the maximum cable length are given as a combination of BAUD-rate and the capacitance of the used cable. Furthermore it was discussed that interference in the cable could cause a loss of data. Lastly the setup of the Anemometer in an area with a minimum wind of disturbance, to ensure quality measurements, could require long cables. Due to these factors it was decided to go for a wireless communication solution for the project.

For wireless communication three solutions was discussed:

- Bluetooth
- WiFi
- RF

Due to limitations of time and resources it was decided to buy low-cost modules that met our requirements.
A Bluetooth module would meet the requirement of wireless communication, but the transmission range could be a problem, the range was $10m - 60m$. Bluetooth allowed connection to a PC or smartphone for debugging and data reception as well as security (Password) could enhance the security. The price for the modules were the highest of three solutions, even for the module with only $10m$ range were more than double the price of the other communication alternatives. Due to the price and range we ruled out Bluetooth for this project.

A WiFi module seemed to be a good choice, the *ESP8266 WIFI Wireless Module*, have an acceptable price as well as the range for these modules seems to work for distances between $25m$ and above $100m$ however we were not able to get any exact maximum distance only some estimations from other users around the world. We already planned to use this module for the IOT part of the project in the Basestation, but were cautious using this for the Anemometer due to the relative high operating current of $\approx 80mA$, hence the limitations of battery lifetime.

A RF module would come at a low price, with lower operating current than the *ESP8266* as well as a good range of $\approx 100m$ for the cheapest module and up to $1000m$ for a larger module. Furthermore it seemed like a good choice to handle the task of a stable, cheap and simple solution of exchanging data between the Anemometer and the Basestation.

2.5 Accessibility

For making the measurements available on the internet, we decided to go with the *ESP8266* module for reasons mentioned above. The *ESP8266* module can be utilized both as webserver and simply as a station that can transmit data to a server on the internet. We wanted to be able to collect a lot of data and keep it which is why we went with using the *ESP8266* module as a station that relays data to a SQL[1] database server. The data on the server can then be accessed from a webpage and used to whatever purpose you would like.

2.6 Power Supply

From the start it was decided to design the system with two MCU's, one Anemometer and one Basestation.

The Basestation was tought of as a relaying station where measured data was received from the anemometer and routed to the website via the *ESP8266* module. Due to the wireless communication with the two MCU, the Basestation could be arranged inside and close to a permanent power supply, therefore a battery power supply for this part of the project was ruled out.

For the Anemometer a permanent power supply could limit the locations for setup just at a wired data-line between the two MCU's were ruled out. Here a battery pack with rechargeable batteries and a solar panel would make a perfect fit for this project, but is still a low priority since this would just be a nice feature of the project.

2.7 Requirement specification

No.:	**Requirement:**	**Description:**	**Prior.:**	**Met:**
1	Windspeed	The system should be able to measure windspeeds between 0.5 m/s and 60 m/s.	high	√
2	Transmission rate	At higher windspeeds the measurement transmissions should increase.	high	√
3	Transmission type	Wireless transmission between anemometer and basestation.	med	√
4	IOT (Internet Of Things)	Should be able to transmit data wirelessly to the internet.	high	√
5	Accessibility	Measurements should be available at a website.	high	√
6	Battery	The anemomter should be able to run on batteries and a solar panel.	med	×

[1]Structured Query Language

3 Design and implementation

3.1 System overview

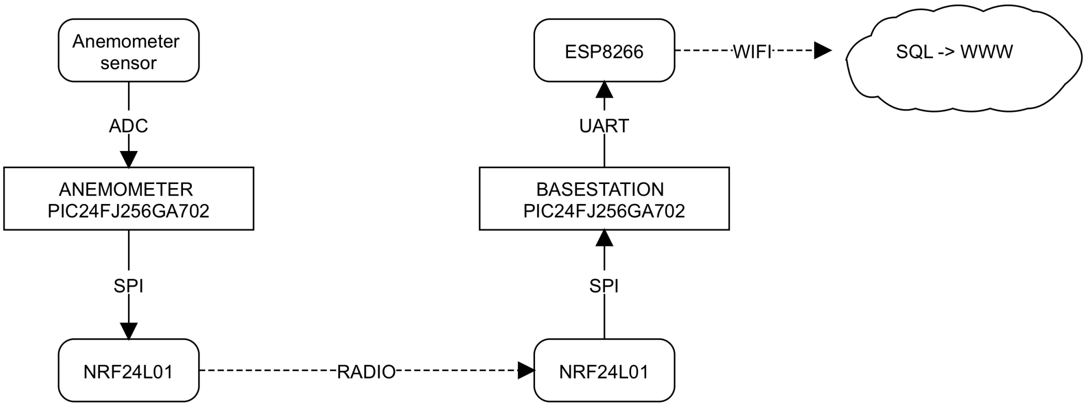


Figure 1: System overview

As seen on the above figure the anemometer sensor transmits a square wave signal representing the current windspeed via a wire to the anemometer base (*PIC24FJ256GA702*). The anemometer base processes that signal and transmits it via SPI[2] to a radio module (*NRF24L01*) which relays the transmission wirelessly to the other radio module. The other radio module then transmits the data to the basestation which relays the transmission to the wifi module (*ESP8266*) via UART. Then the wifi module which is connected to a router via wifi transmits the data via a website to a SQL server. At last the data is fetched from the SQL server by a website where the data is displayed weather.jorgengreve.dk.

Our main focus in this project has been to get data from the anemometer sensor to the basestation PIC microcontroller. Therefore the "path" from the basestation to the world wide web is not described in detail in this report but the code is attached in the appendix where *esp8266.ino* is the Arduino IDE code for the *ESP8266* module and *post-esp-data.php* is the code for the webpage the wifi module transmits its data to. The code for the webpage that fetches the data from the SQL server is not included because it is a wordpress[3] website which has primarily been built with a WYSIWYG[4] editor. But as mentioned earlier that part can be viewed live at jorgengreve.dk.

The *ESP8266* wifi module has been programmed in Arduino IDE with the help from Random Nerd Tutorials www.randomnerdtutorials.com this also applies to the webpages and the SQL server setup. A lot of the code from Random Nerd Tutorials has been altered to suit our needs.

3.2 Anemometer hardware design

Our goal was to engineer a system that would work in the real world which is why we chose to spend a lot of time in the beginning of the semester designing a fully working 3D printable anemometer in Autodesk Fusion 360. We are currently at version 4 which has proved itself to be an acceptable version. It has per 11th. of may 2021 been working flawlessly for a month. A youtube video of the anemometer at work can be viewed here https://youtu.be/4FcZ-f89uGU.

[2]Serial Peripheral Interface
[3]www.wordpress.org
[4]What You See Is What You Get

Figure 2: Anemometer @ work

We have created a small Fusion 360 anemometer explode view video that displays almost all the parts of the anemometer (The ball bearings are not included) the video can be viewed on https://youtu.be/TXc9oVmYras. The figure below shows the anemometer prior to assembly with all parts except the wire.

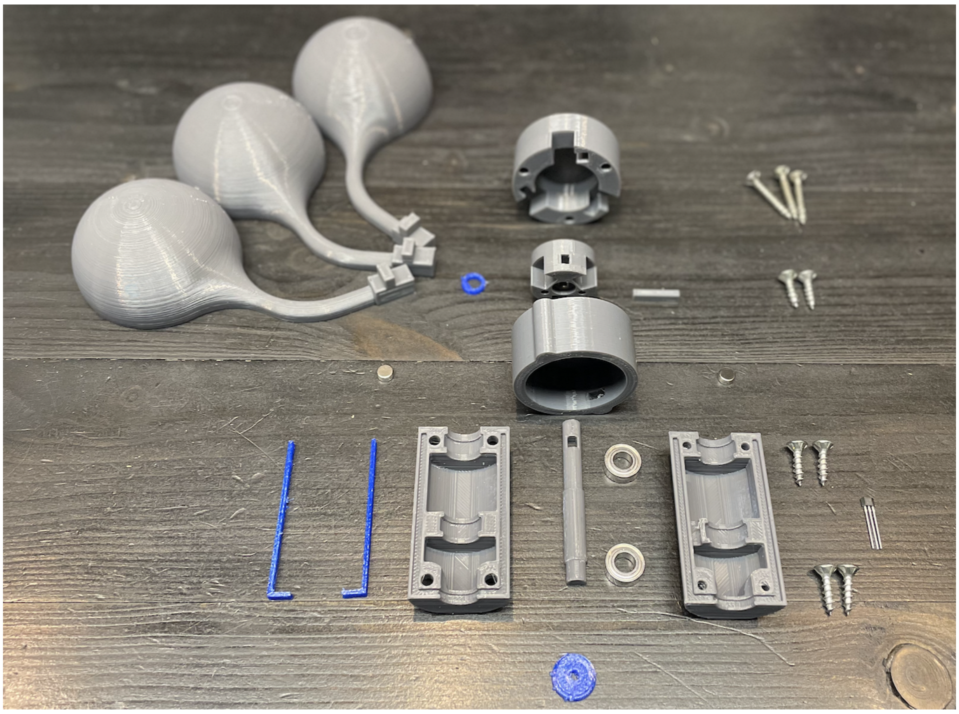


Figure 3: Anemometer disassembly

The anemometer works by having a neodymium magnet north pole and south pole passing by a hall sensor which acts like a switch. This creates a square wave signal we can use to interpret the windspeed from by timing for how long its high and low.

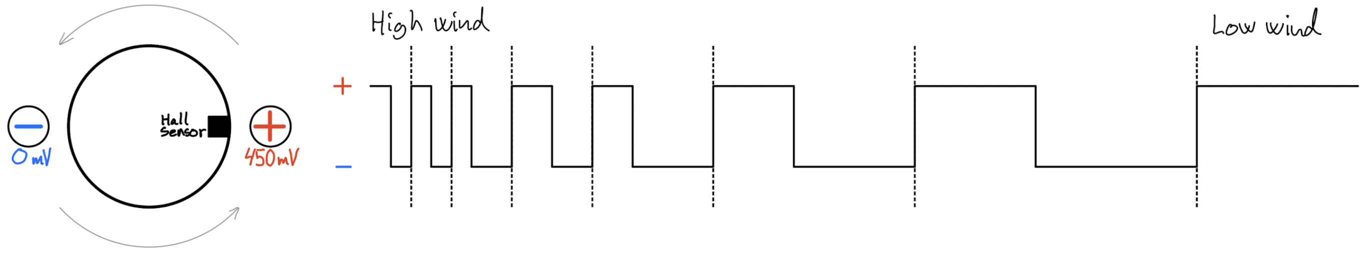


Figure 4: Anemometer concept drawing

We have chosen to do the timing with timer interrupt because we need to know the time between two counts to calculate the total time. We let a timer interrupt every $10\mu s$ and every time it interrupts we count a variable one up. When the north pole magnet passes the hall sensor and thereby creates a "high" ADC output we reset the counter variable and let it count until the south pole passes the hall sensor and the ADC output goes "low". Then we use the counter variable value to calculate how long the half turn took by multiplying the counter value with the interrupt time. This is then multiplied by 2 to get the time a whole turn would take and to get how many turns per second that is we divide 1 by the turn time.

$$trnTime = 2(intrrCnt \cdot intrrTime)$$
$$tps = \frac{1}{trnTime}$$

Now we have how many turns the anemometer has made in one second which is then multiplied with the distance one turn is to get the uncalibrated windspeed. At last we multiply the uncalibrated windspeed with a calibration factor to get the windspeed in meters per second.

$$wsUnc = trnDist \cdot tps$$
$$mps = wsUnc \cdot calibFact$$

3.2.1 Electrical hardware

The electrical hardware is still working out of a breadboard setup but the plan is to design PCB's so that the whole project can be mounted in plastic casings.

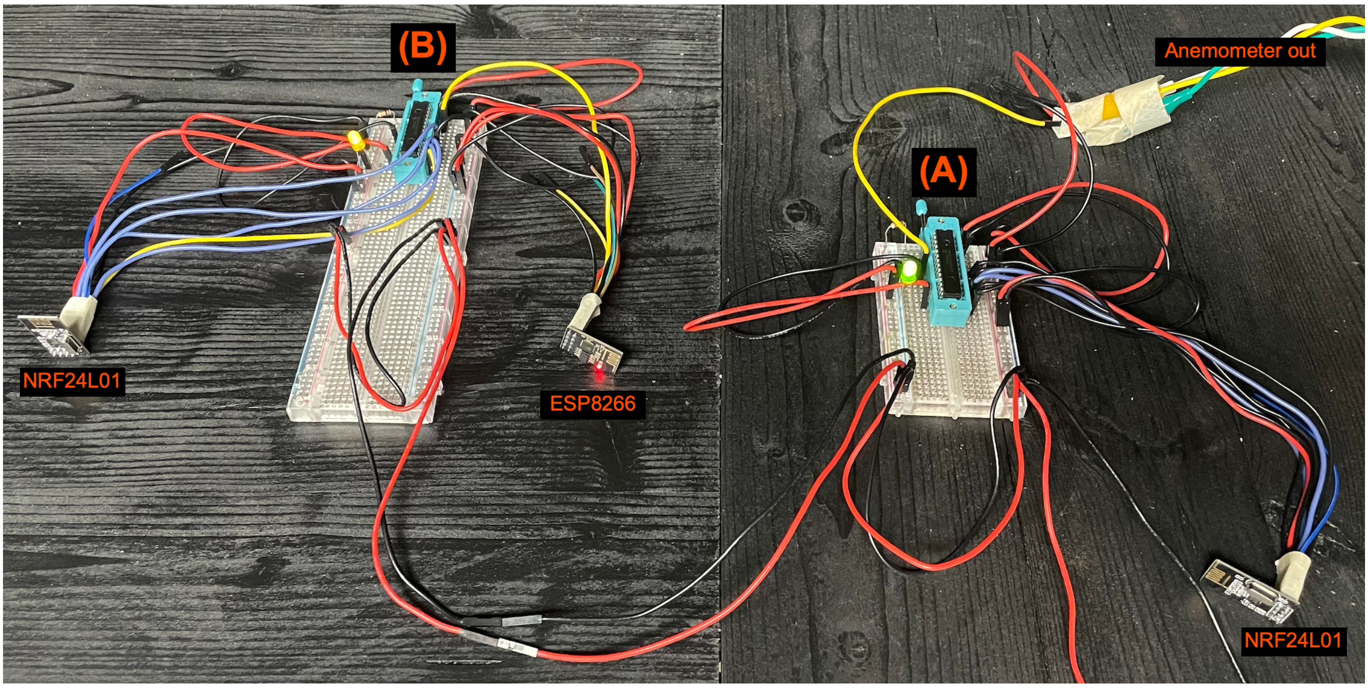


Figure 5: Hardware breadboard setup

As seen on the above figure (A) is the PIC24FJ256GA702 anemometer microcontroller and (B) is the PIC24FJ256GA702 base station microcontroller. The anemometer output is fed into (A) which processes the windspeed and then transmit the data in the format "WX.XXX" (or "WXX.XX" if its more windy) via the far right NRF24L01 chip to the far left NRF24L01 chip and into (B) which just relays the data to the ESP8266 module and then its finally transmitted to the server. Every time a NRF24L01 transmission happens the green and yellow LED toggles, this is just for testing purposes.

3.2.2 Schematics

Below are the schematics for the anemometer and the basestation. Decoupling capacitors are included in the schematics but not on the breadbords.

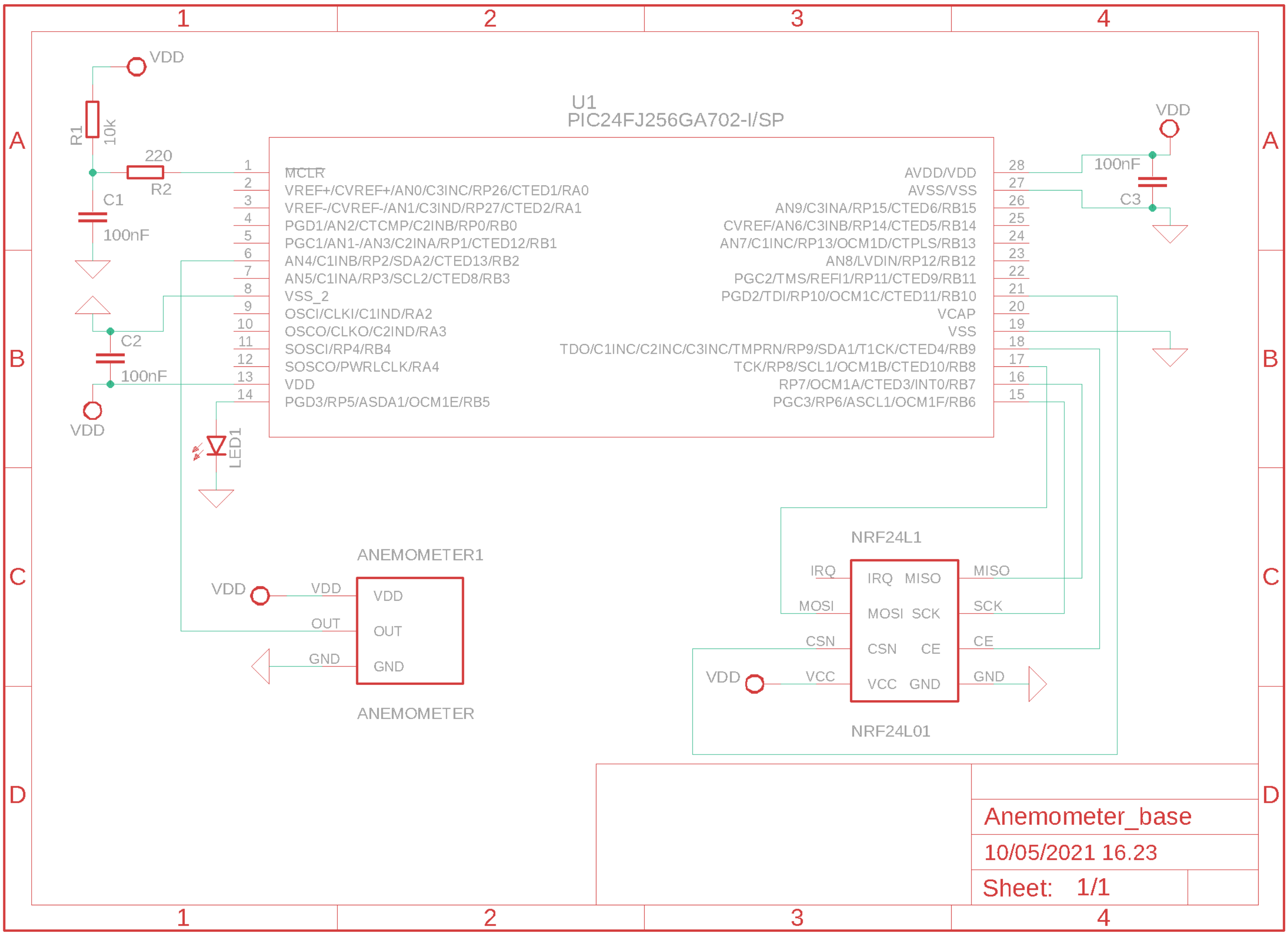


Figure 6: Anemometer base schematic

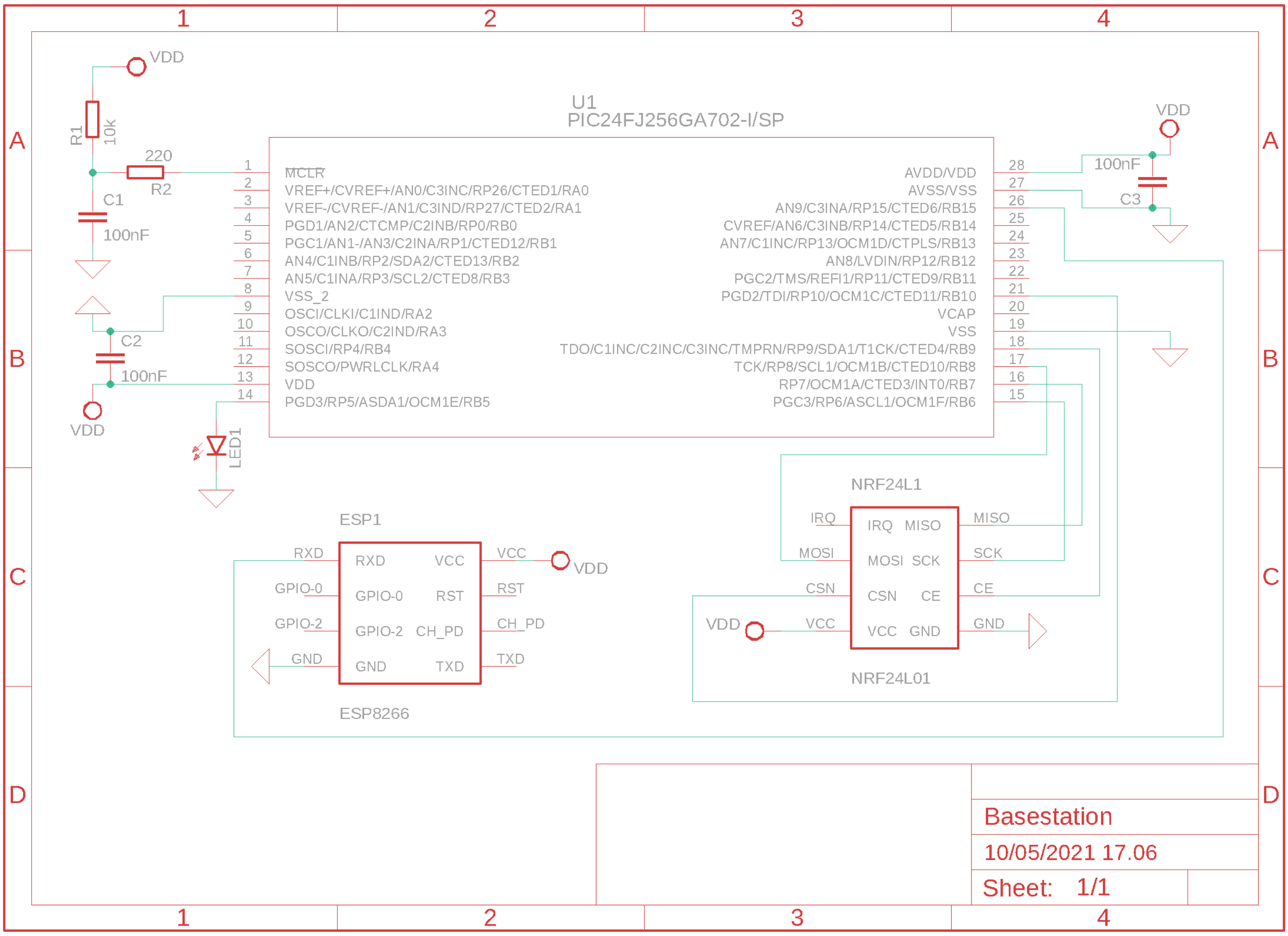


Figure 7: Basestation schematic

3.3 Anemometer software design

Both the anemometer PIC microcontroller and the basestation PIC microcontroller has been set up with MCC[5].

3.3.1 void getWindspeed(void)

The getWindspeed() function is furthermore divided into the following sub categories.

3.3.1.1 Analog to Digital Conversion

The analog signal from the anemometer which is either 0V or 450 mV is being converted by the ADC to a value of either 0 or 3300. The digital value is then used to determine if the anemometer output is low or high as seen on the below ADC conversion flow chart.

[5] MplabX Code Configurator

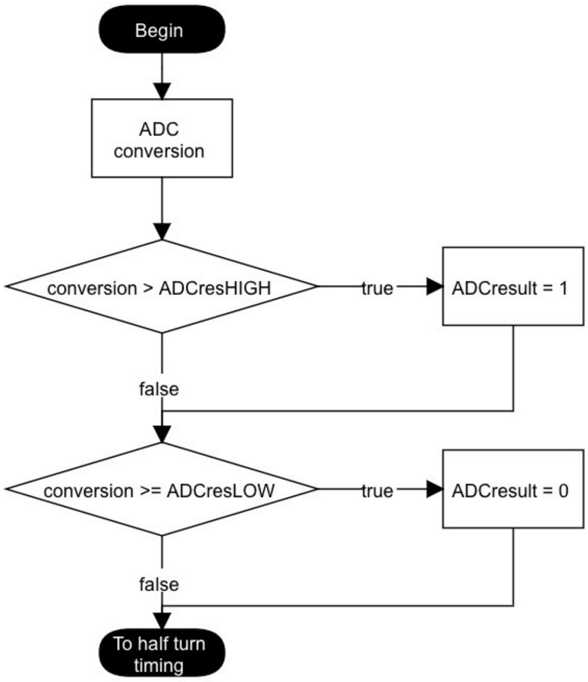


Figure 8: ADC conversion flow chart

3.3.1.2 Half turn timing

To time how long one turn on the anemometer takes we start a timer *tmr1intrrCnt1 = 0* or *tmr1intrrCnt2 = 0* (see timing under section 3.2) when either of the two magnetic poles pass by the hall sensor and then we stop the timer when the opposite pole of the one that started the timer pass the hall sensor. Every time a pole passes the sensor one timer starts, the other timer stops and an interrupt count is ready, *intrrCnt1ready = 1* or *intrrCnt2ready = 1*. The whole process can be seen in the below flowchart.

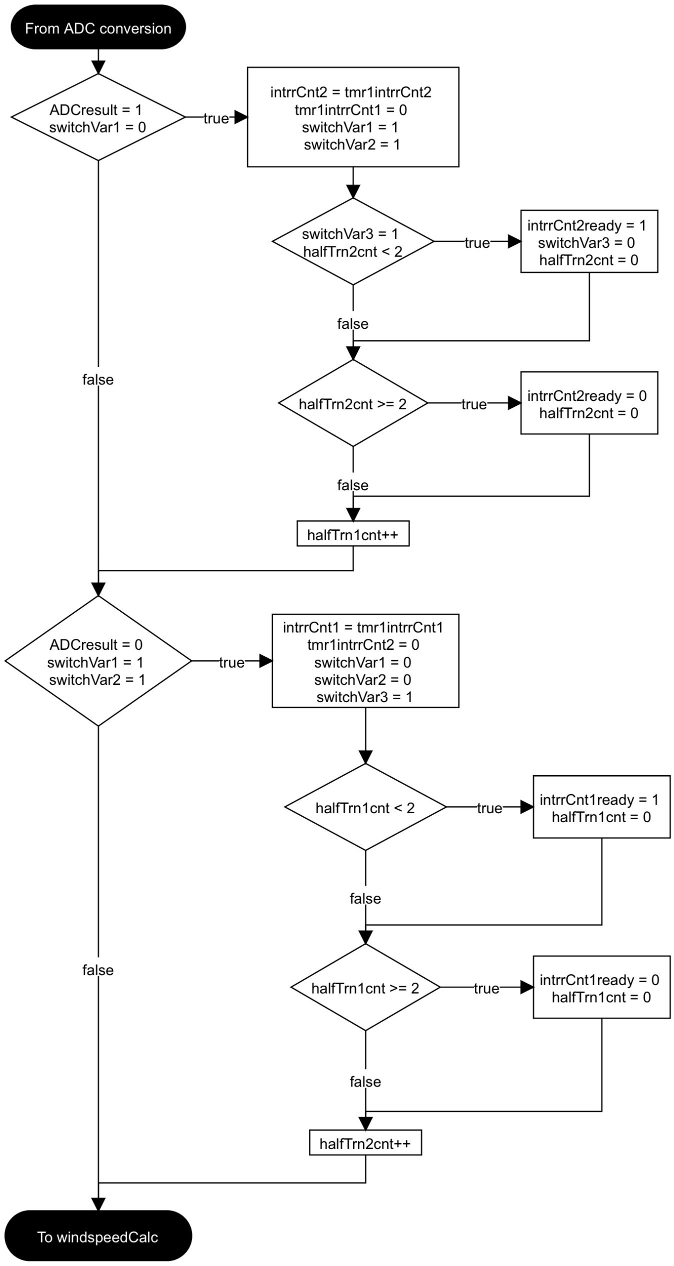


Figure 9: Half turn timing flowchart

In the below table column Init is the value that the variable is initialized with. Column 1. shows the initialization half turn on the anemometer. The half turn timing always start with a high output from the ADC when it starts up from being shut down and if the anemometer is at a standstill the output of the system will be 0.00 m/s until the anemometer starts turning. Columns 2. to 6. shows a repeated pattern of receiving a high or low *ADCresult* and then outputting *intrrCnt(1/2)ready* and the result *intrrCnt(1/2)*

Variable:	Init:	1.	2.	3.	4.	5.	6.
ADCresult	0	1	0	1	0	1	0
switchVar1	0	1	0	1	0	1	0
switchVar2	0	1	0	1	0	1	0
switchVar3	0	0	1	0	1	0	1
halfTrn1cnt	0	1	0	1	0	1	0
halfTrn2cnt	0	0	1	0	1	0	1
intrrCnt1ready	0	0	1	0	1	0	1
intrrCnt2ready	0	0	0	1	0	1	0
intrrCnt1 = tmr1intrrCnt1			X		X		X
intrrCnt2 = tmr1intrrCnt2		X		X		X	

Table 1: Half turn timing truth table

3.3.1.3 Windspeed calc

The windspeed is calculated only if *intrrCnt(1/2)ready* has been set in half turn timing because only then has the half turn been timed and the result is ready to be processed. This is done like it is described under section 3.2 and like in the flowchart below. The windspeed calc part of the *getWindspeed()* function outputs the calibrated windspeed *mps* and a flag (*mpsReady*) that tells the function *transmitMps()* that a wind measurement is ready to be sent.

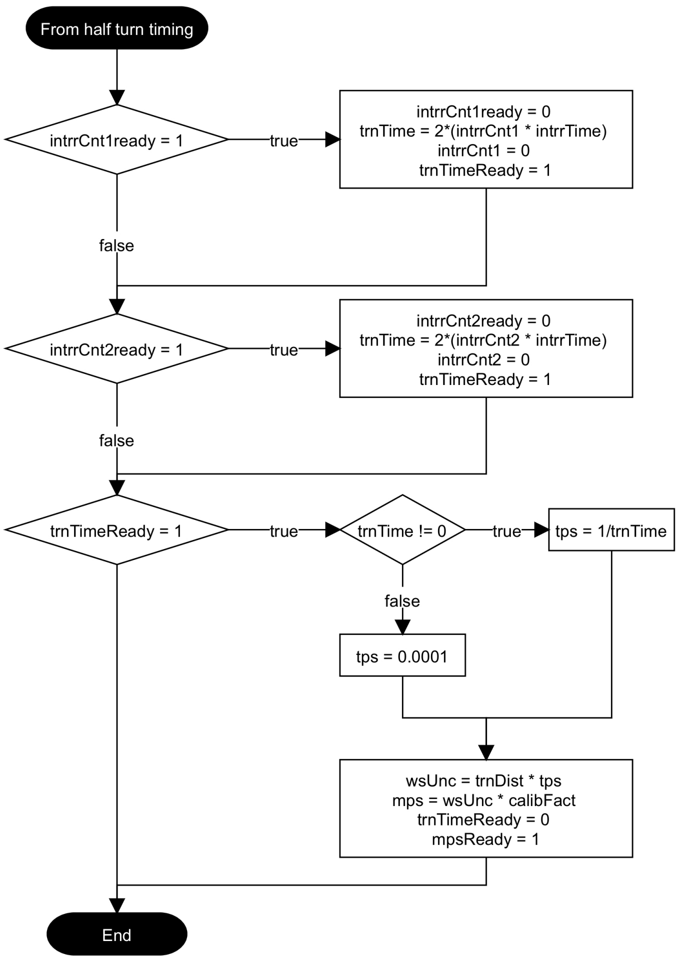


Figure 10: Windspeed calc flowchart

3.3.1.4 Calibration

The system can be calibrated by changing the calibration constant *calibFact*. The calibration constant is multiplied with the uncalibrated windspeed reading *wsUnc* so that it is possible to adjust the reading to match the exact windspeed. At the moment the anemometer has been loosely calibrated by checking the anemometer readout and adjusting *calibFact* until the readout matched current local wind data from dmi.dk. At the moment *calibFact* has been set to 6. The anemometer should be calibrated more thorough before relying solely on the readings. This could preferably be done in a windtunnel where you know the exact windspeed but it is also possible and much less expensive to buy a precalibrated anemometer and adjust ours to match or simply stick it out a car window at a certain speed an count the anemometer revolutions.

3.3.2 void transmitMps(void)

The "smart part" of the system lies in the *transmitMps()* function as the systems ability to adjust the rate of transmissions according to the current windspeed. This is done by the mathematical function:

$$sendSpeed = -383 \cdot mps + 25000$$

The function comes from a calculation done as below in Maple where the windspeed in m/s is found on the x-axis and the transmission delay is found on the y-axis.

$x1 := 0.0:$
$x2 := 60.0:$
$y1 := 25000:$
$y2 := 2000:$

$$a := \frac{y2 - y1}{x2 - x1} = -383.3333333000$$

$b := y1 = 25000$

$f(x) := a \cdot x + b = x \rightarrow a\,x + b$

$plot(f(x), x = 0\,..60, gridlines)$

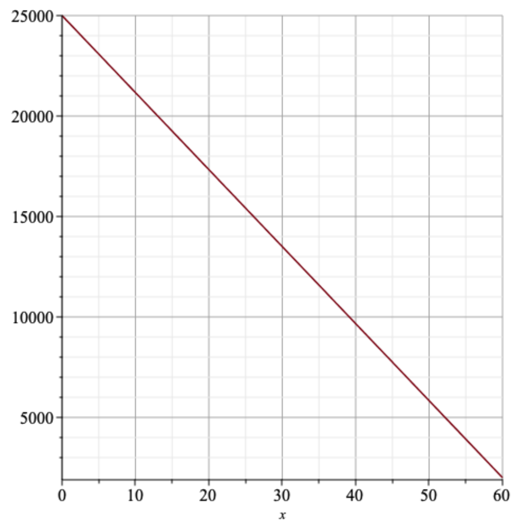


Figure 11: Transmit speed Maple calculation

The transmission delay is called *sendSpeed* in the flowchart below and is simply a variable value that is calculated from the mathematical function mentioned earlier and the current windspeed. This is then compared to a variable (*cnt1*) that is counted one up every time the *transmitMps()* function is run. When *cnt1* is larger than the current *sendSpeed* value and a windspeed reading is ready (*mpsReady = 1*) the windspeed reading is converted into a character array by the *sprintf* function so that we can insert a prefix 'W' into the windspeed reading. This is done to be able to transmit other data as well. At last the data is transmitted to the basestation via the *NRF24L01* mudule.

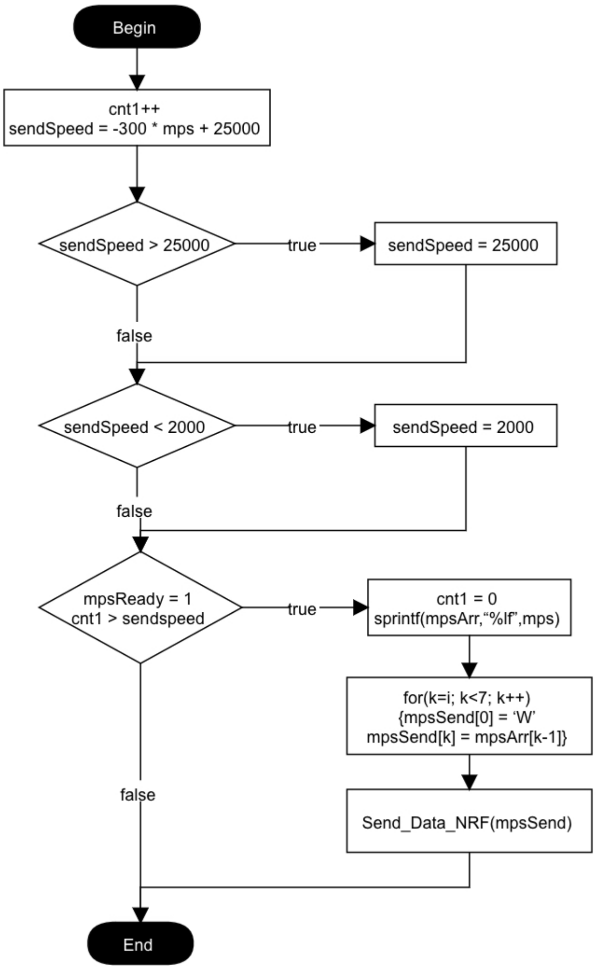


Figure 12: Transmit windspeed flowchart

3.4 Basestation

The basestation at the moment only relays data from the *NRF24L01* module to the *ESP8266* wifi module but later on the plan is to attach a screen to display the weather data directly and also to receive user input via an interface.

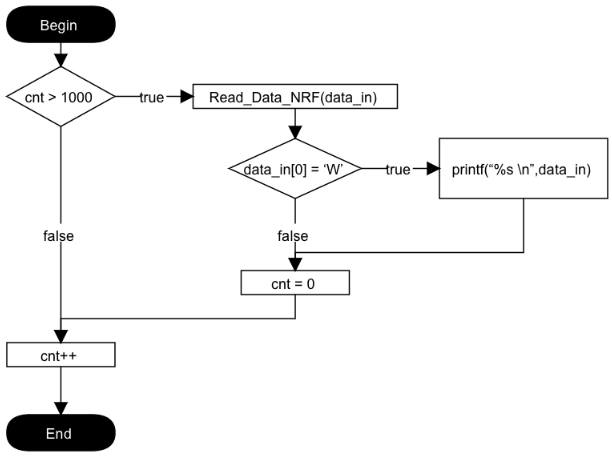


Figure 13: Basestation flowchart

3.5 Wireless communication

In this section the implementation of the two wireless technologies are discussed.

3.5.1 NRF24L01

As discussed in the analysis it was decided to implement the NRF24L01 RF module. We had no prior experience with this module, so inspiration was found in other projects on the internet, here one website helped a lot to understand how the communication between a PIC MCU and a NRF module worked as well as how to configure it the right way[6]. The wireless communication between the Anemometer and the Basestation was designed as one-way communication, where the anemometer sends the measured wind speed to the Basestation via two NRF24L01 modules. Below the basics of this module as well as the functions are explained:

Figure 14: NRF24L01 pin out diagram[*]
[*]Borrowed from https://circuitdigest.com/microcontroller-projects/interfacing-nrf24l01-rf-module-with-pic-microcontroller

As seen on the figure above the pinout of the NRF24L01 consist of the following pins:

- **GND** Ground.
- **VCC** Positive 1.9V - 3.3V.
- **CE** Chip Enable.
- **CSN** Chip Select NOT.
- **SCK** Serial clock.

[6]http://blog.diyembedded.com/2007/06/tutorials-1-3-for-pic-completed.html?m=1

- **MOSI** Master Out Slave In.
- **MISO** Master In Slave Out.
- **IRQ** Interrupt Request.

The CE pin is used to set the NRF to receive(CE=1) or transmit(CE=0). Further it is used in transmit mode. When a payload has been transferred to the TX_PAYLOAD buffer a high pulse on CE will transmit the payload from the NRF. The CSN are active low, so before a command is sent from the MCU to the NRF, CSN are pulled low. SCK, MOSI and MISO are used for the SPI connection. IRQ is an interrupt pin, in this project it is used for "listening" for received data from the Anemometer to the Basestation.
To operate the NRF24L01 the datasheet[7] reveal some commands that are used to eg. read or write a register:

Command name	Command word (binary)	Operation
R_REGISTER	000A AAAA	Read command followed by the register that should be read
W_REGISTER	001A AAAA	Write command followed by the register that should be written to
R_RX_PAYLOAD	0110 0001	Read the received payload
W_TX_PAYLOAD	1010 0000	Writes payload to the NRF
FLUSH_TX	1110 0001	Flush TX FIFO
FLUSH_RX	1110 0010	Flush RX FIFO
NOP	1111 1111	Dummy byte that have no effect, can be used when reading a register

Table 2: Table of commands

Furthermore the NRF24L01 have a number of registers that has to be written in order to configure the device[8]
In the NRF.c and NRF.h the functions used for communicating with the NRF are created. Every time an exchange of data, between the MCU and NRF, are excecuted the CSN pin are pulled low before and high again after the transaction.

3.5.1.1 void Write_NRF(char reg, char value)

This function first sends the write register command, table 2, followed by the register that should be written to.
Next it sends the value that should be written to the register and after that CSN are pulled high again.

Example:

Write_NRF(0x00,0x0A);

Writes the value of 0x0A to the config register(address 0x00).

3.5.1.2 char Read_NRF(char reg)

This function first sends the read register command, table 2, followed by the register that data should be read from.
Afterwards the data received from the NRF are stored in a variable (RB) and this value are returned from the function.

Example:

Read_NRF(0x00);

Reads the data stored in the config register(adress 0x00).

[7]nRF24L01_Product_Specification_v2_0-9199.pdf
[8]nRF24L01_Product_Specification_v2_09199.pdf, page 53-58

3.5.1.3 void Write_Buffer_NRF(char dest, char * buffer, char amount_bytes)

This function can send a buffer to the NRF module where "dest" is the register that receives the buffer, the pointer "buffer" are the buffer that should be transferred and "amount_bytes" are the amount of bytes the "buffer" holds. First a write command is OR'ed with the register are transferred to the NRF followed by a for loop that transfer every byte spaced with a small delay of $10\mu s$.

Example:

Write_Buffer_NRF(0xA0, example_buffer, 5);

Transmit the content of "example_buffer" (5 bytes) to the W_TX_PAYLOAD(0xA0), table 2.

3.5.1.4 void Read_Buffer_NRF(char dest, char * buffer, char amount_bytes)

This function reads the content of a buffer on the NRF. Where "dest" is the register from where the buffer is received from, the pointer "buffer" is the array on the MCU that recieves the bytes and "amount_bytes" are the amount of bytes to be received. First a write command is OR'ed with the register are transferred to the NRF followed by a for loop that receives the bytes one by one after the transfer is done the buffer is rounded off with a NULL character.

Example:

Read_Buffer_NRF(Read_RX_PLD, example_buffer, 5);

Receives the content, 5 bytes, of R_RX_PAYLOAD(0x61) to the "example_buffer".

3.5.1.5 void Send_Data_NRF(char * buffer)

This function is used for sending data from one NRF to another NRF. A buffer is transferred to the function, this buffer is transferred to the W_TX_PAYLOAD using the Write_Buffer_NRF function a high pulse on CE execute the NRF to transmit the buffer.

Example:

Send_Data_NRF(example_buffer);

Transfer the bytes from example_buffer to the NRF and transmit it to another NRF module.

3.5.1.6 void NRF_FLUSH(void)

This function reset the Data_Ready, Data_Sent and Max_Retransmits interrupt flags. Afterwards the TX FIFO and RX FIFO registers are flushed. This function makes sure that the interrupt flags and the registers are cleared after receiving data.

Example:

NRF_FLUSH();

Clear interrupt flags as well as flush RX and TX FIFO registers.

3.5.1.7 void Read_Data_NRF(char *buffer)

This function transfers received data from the NRF. This is used when data are received on the NRF. After the bytes have been received the Data Ready RX interrupt flag are cleared and the NRF_FLUSH function flush out the TX and RX FIFO registers.

Example:

Read_Data_NRF(example_buffer);

Reads the data that are ready on the NRF and transfer it to the example_buffer.

3.5.1.8 char Data_Ready_NRF(void)

This function only works if the RX_DR interrupt mask are activated in the CONFIG register. This function are built up on polling this function in a while loop, when the STATUS register & 0x40 = 0x40 the function return a zero and breaks the while loop. Otherwise a one is returned at the while loop carry on.

Example:

```
while(Data_Ready_NRF())
{

}
```

Read_Data_NRF(example_buffer);

Polling on the Data_Ready_NRF(). When the RX data ready flag are raised the while loop breaks and the received data are transferred to example_buffer with the Read_Data_NRF(example_buffer).

3.5.1.9 void startup_NRF_delay(void)

This function simply inflict a small delay before powering up and configuring the NRF. This is created due to issues with the SPI initializing were not up and running fast enough, hence this delay gives a little time for the SPI to stabilize before the MCU begin to communicate with the NRF module.

3.5.1.10 void pwr_up_NRF(void)

This function power up the NRF module by setting the power up bit high in the CONFIG register.

3.5.1.11 void config_NRF(void)

This function is the one that setup the NRF module to the required specifications. In the bullet points below the configuration is explained[8]:

- **CONFIG** Set to RX mode, PWR_UP kept high, CRC disabled, RX_DR kept low to enable interrupt mask(the other interrupt masks are also used, but in this project the IRQ pin is only used on the receiving MCU.
- **EN_AA** Auto Acknowledgment disabled.
- **EN_RXADDR** Data pipe 0 enabled.
- **SETUP_AW** Address width set to 5 bytes.
- **SETUP_RETR** Not used so set to register is just set to 0x00 which marks a wait for $250\mu s$.
- **RF_CH** Set the channel to $F0 = 2400MHz + RF\_CH = 2400MHz + 50Mhz = 2450MHz$.

- **RF_SETUP** LNA gain off, RF output PWR = 0dBm, Air Data Rate set to 2Mbps.
- **RX_ADDR_P0** Receive address P0 set to 0xe1,0xe1,0xe1,0xe1,0xe1.
- **TX_ADDR** Transmit address set to 0xe1,0xe1,0xe1,0xe1,0xe1.
- **RX_PW_P0** Number of bytes in RX payload set to 5 bytes.

After these registers are configured the NRF device are either set to RX or TX operation. If set to RX CE=1 and the CONFIG register bit 0 are set high. If set to TX CE=0 and the CONFIG register bit 0 are set low.

3.5.2 ESP8266

The *ESP8266* wifi module is programmed in Arduino IDE (code is in the appendix) and is relatively simple to operate. It receives data via UART and then it transmits the data via wifi to a webpage which relays the data to a SQL server.

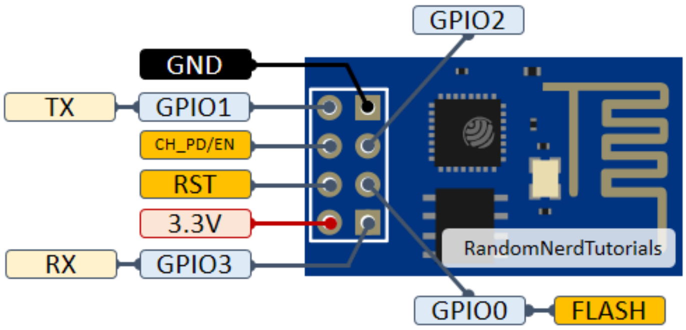


*Figure 15: ESP8266 pin out diagram**
*Borrowed from https://randomnerdtutorials.com/esp8266-pinout-reference-gpios/

To be a bit more specific the *ESP8266* connects to the wifi with the specified network credentials. When it is connected it "listens" for data on the UART RX pin. When it receives data it removes the 'W' we have inserted in the anemometer base, then it creates a connection to the SQL server via the *post-esp-data.php* (code is in appendix) webpage and post the data to the server before it closes the connection again and continue to listen for data on the UART RX.

4 Test

4.1 Requirement 1 - Windspeed

4.1.1 Test script

To test the windspeed requirement we send out a square wave signal from a signal generator into the ADC on the PIC microcontroller. To test if the system can measure windspeeds as low as 0.5 m/s we need a square wave signal with a frequency of 114 mHz[9] and to test if the system can measure windspeeds up to 60 m/s we need a square wave signal with a frequency of 13.7 Hz[10]. To see what the system outputs we use a logic analyzer to read UART output from the PIC microcontroller.

4.1.2 Test performance

As seen on the below figure we feed the ADC with a 0V/500mV square wave signal with a frequency of 114 mHz and the UART outputs W0.5035 where the W is the prefix for windspeed and 0.5035 is the windspeed in m/s which corresponds nicely with our requirement of 0.5 m/s.

Figure 16: Windspeed minimum test (0.5 m/s)

As for the 60 m/s test we do the same as above but with a frequency of 13.7 Hz instead. On the figure below we can see that the UART outputs W61.156 where the W is the prefix for windspeed and 61.156 is the windspeed in m/s which is a slight bit more than our requirement of 60 m/s. This could be due to the fact that we only time one half turn on the anemometer which multiply by 2 to get the time for a whole turn and then multiply by the calibration factor and since it is only an issue at very high windspeeds where it is of less concern if it is 1 m/s higher or lower we find it acceptable.

[9] $mps = 0.5m/s$, $trnDist = 0.73m$, $calibFact = 6$, $wsUnc = mps/calibFact = 0.08333$, $tps = wsUnc/trnDist = 0.114Hz$

[10] $mps = 60m/s$, $trnDist = 0.73m$, $calibFact = 6$, $wsUnc = mps/calibFact = 10$, $tps = wsUnc/trnDist = 13.7Hz$

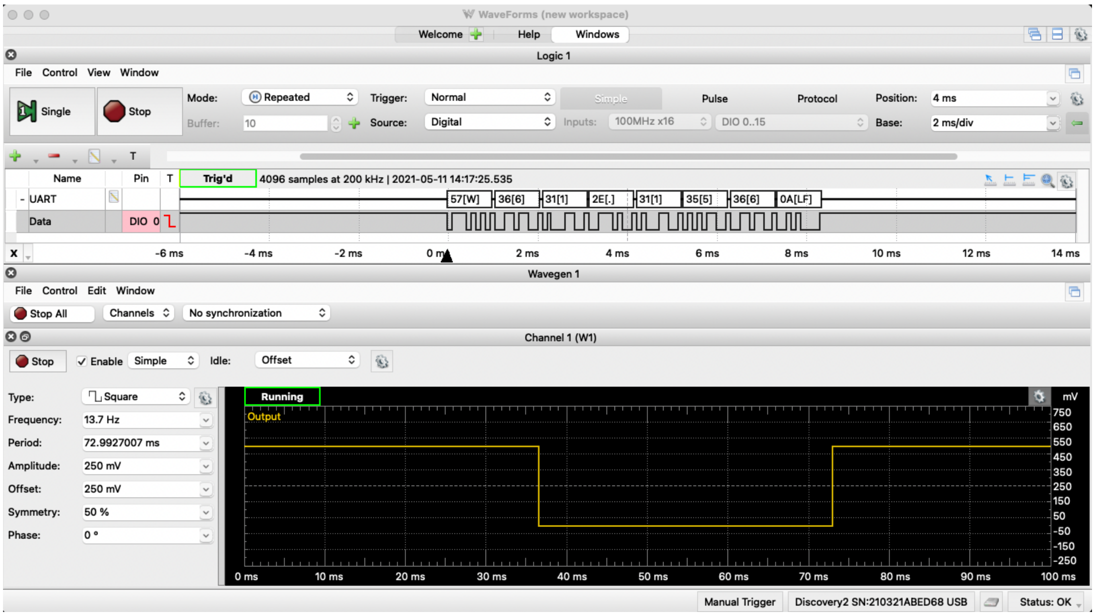


Figure 17: Windspeed maximum test (60 m/s)

4.2 Requirement 2 - Transmission rate

4.2.1 Test script

The transmission rate can be measured by toggling a LED every time a transmission happens and then use an oscilloscope to measure the time between.

4.2.2 Test performance

As seen in the figure below the transmission rate is about 17 seconds at low windspeeds (114 mHz = 0.5 m/s).

Figure 18: Low transmission rate test

As seen in the figure below the transmission rate is about 2 seconds at high windspeeds (13.7 Hz = 60 m/s).

Figure 19: High transmission rate test

4.3 Requirement 3 - Transmission type

The transmission type is 2.4GHz wireless transmission via two NRF24L01 modules which we have not tested as such because we get the expected output every time which we are certain about because of the prefix 'W' we put in front of every transmission and if we get the W in the beginning of our output the following digits must also be the same as the input.

4.4 Requirement 4 - IOT

4.4.1 Test script

This requirement has been tested by setting up the signal generator to provide a steady square wave on the ADC and see if we get the expected windspeed reading on the website.

4.4.2 Test performance

We set up the signal generator with a 1.142 Hz square wave which corresponds to 5 m/s.

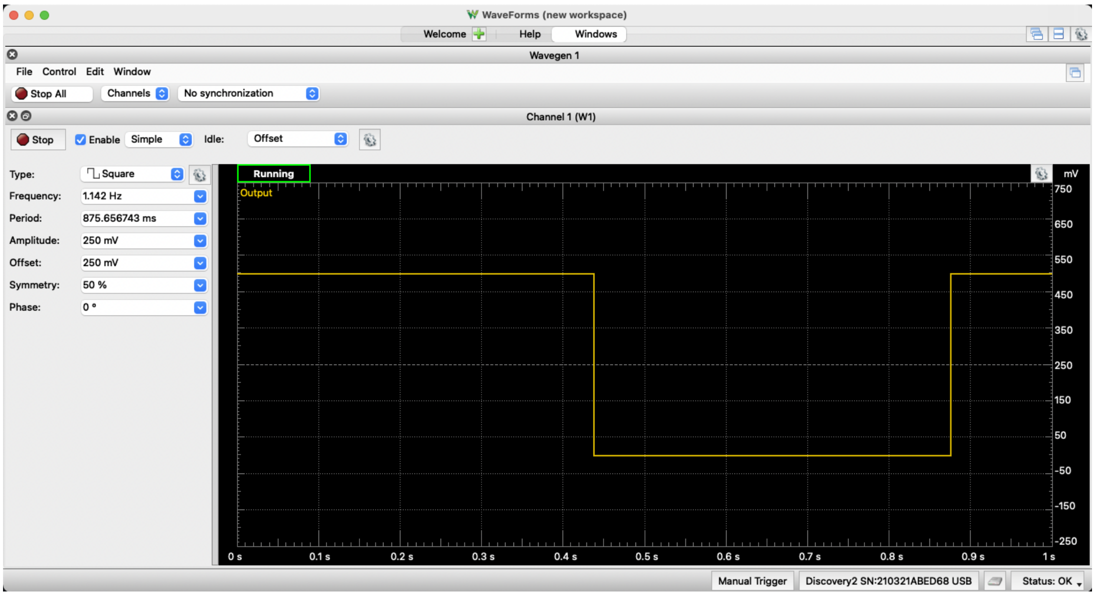


Figure 20: IOT test ADC input

And on the website weather.jorgengreve.dk we got a steady 5.02 m/s which is a tiny bit off but within acceptable limits. (The other data points on the graph except for the zero readings are earlier windspeed readings)

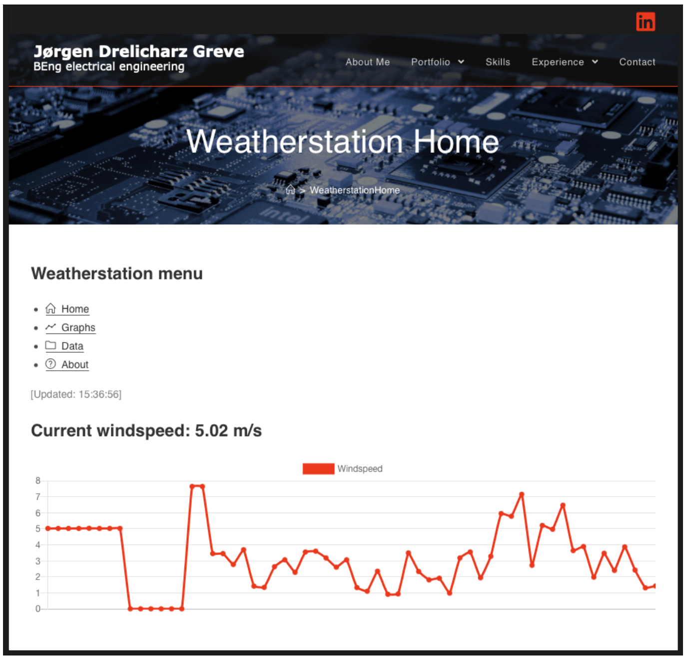


Figure 21: IOT test output - weather.jorgengreve.dk

4.5 Requirement 5 - Accessibility

As shown above the windspeed data can be viewed at the website weather.jorgengreve.dk[11] where it is also possible to see at the graph of the past six hours windspeed and data from the last 24 hours.

[11] The system is not always up and running because it is still running of of breadboards.

4.6 Requirement 6 - Battery

Unfortunately we did not find the time to implement batteries and solar panel before our deadline but it is going to happen in the future.

5 Conclusion

As stated in the problem statement our main problem that we wanted to solve was to find a way to tell how much the wind is blowing using our knowledge of electronics and microcontrollers. At that we have succeeded by designing and realising an IOT windspeed measuring device.

We wanted to design and build an anemometer which could send a signal carrying windspeed information to a PIC microcontroller. That we did, it took some time and some attempts but at the end we had created a fully functional weatherproof windspeed sensor that can provide windspeed data as a square wave signal to the PIC MCU.

The issue of communicating over longer distance we solved with radio communication in the form of *NRF24L01* radio modules. They are cheap and functional, but definitely not easy to work with when you have to teach yourself how they work. This alone has taken a lot of hours to solve and we almost dropped the implementation of it before we had a breakthrough. Fortunately it works perfectly now.

The upload to and display of data on the world wide web succeeded thanks to the *ESP8266* wifi module. It seamlessly connects our project to the internet and thereby qualifies it as an IOT system.

The only problem we wanted to solve and did not succeed with was the anemometer base power issue. But we knew from the beginning that this was one of the things that we probably did not have enough time to work out. Fortunately we have a long summer coming where this is definitely being implemented together with some more sensors such as temperature and maybe pressure or other interesting parameters we could measure. We also want to move the project from the breadboards and onto PCB's and design boxes that can house the electronics. At last the basestation needs a screen to display weather data and an interface where you can set certain parameters such as the rate of data transmissions or change the calibration constants.

If we have to clarify our individual main contributions to this project, Kim has spent a lot of time and energy in getting the *NRF24L01* radio modules up and running which includes creating headerfiles from scratch (*NRF.h* and *NRF.c*) and Jørgen's main contribution is the anemometer code (*anemometer.h* and *anemometer.c*).

We headed into this project without any prior knowledge of how to design an anemometer or to set up any form of wireless communication. But now we have learned a lot about it and will therefore be able to implement both radio communication and wifi into our future projects which is a huge benefit for us as we both would love to work with microcontroller systems in our future career as engineers.

At last, as also encouraged in the report, feel free to explore our webpage and youtube videos:

(*The anemometer can be down and false values can be posted to the website due to work on the system*)

Our webpage with windspeed data: **jorgengreve.dk**
Anemometer @ work video: **youtu.be/4FcZ-f89uGU**
Anemometer explode view video: **youtu.be/TXc9oVmYras**

A Appendix

A.1 anemometer.c

```c
/*
 *
 * Author(s):   Kim Holmberg Christensen
 *              Jørgen Drelicharz Greve
 *
 * Filename:    anemometer.c
 * Version:     1.0
 * Date:        16.05.2021
 *
 */


#include <stdio.h>
#include "anemometer.h"
#include "mcc_generated_files/system.h"
#include "mcc_generated_files/adc1.h"
#include "mcc_generated_files/spi1.h"
#include "globalVariables.h"
#include "NRF.h"


/********** DEFINITIONS FOR getWindspeed() ************************************/
#define channel 0               // ADC channel used
#define ADCresHIGH 300          // 450mV = 3351
#define ADCresLOW 200           // 15mV  = 112
#define intrrTime 0.000010      // Timer 1 interrupt time
#define trnDist 0.73            // One scoup turn length in meters
#define calibFact 6             // Calibration constant


/********** DEFINITIONS FOR getWindspeed() ************************************/
#define testMode 0              // 1 = UART and LED, 2 = UART, 3 = LED
                                // The system will transmit more frequent when
                                // testMode is 1


/********** VARIABLES FOR getWindspeed() **************************************/
int i = 0;                      // for loop counting variable
int switchVar1 = 0;             // The switchVar 1, 2 and 3 variables are used -
int switchVar2 = 0;             // to control the anemometer half turn timing -
int switchVar3 = 0;             // calculation.
int intrrCnt1ready = 0;         // First half turn interrupt count ready
int intrrCnt2ready = 0;         // Second half turn interrupt count ready
int halfTrn1cnt = 0;            // Counts number of half turns
int halfTrn2cnt = 0;            // Counts number of half turns
int trnTimeReady = 0;           // One turn time is calculated and ready
int conversion = 0;             // ADC conversion result
int ADCresult = 0;              // ADC result (1 or 0)
long intrrCnt1 = 0;             // Holds the latest number of half turn -
long intrrCnt2 = 0;             // interrupts.
long tmr1intrrCnt1 = 0;         // Extern variable counted one up in the timer -
long tmr1intrrCnt2 = 0;         // 1 interrupt callback function.
double trnTime = 0;             // Time of one turn in seconds
double tps = 0;                 // Number of turns per second
double wsUnc = 0;               // Uncalibrated windspeed
volatile int mpsReady = 0;      // 1 if m/s result is ready, 0 if not
volatile double mps = 0;        // Windspeed in meters per second


/********** VARIABLES FOR transmitMps() ***************************************/
```

```
char dataCheck[6] = {NULL};     // Used to sort out any extreme values
char dataBuff[6] = {NULL};      // Used to store the five latest mps values
char dataSum = 0;               // The sum of the latest five mps values
char mpsArr[6] = {NULL};        // Holds double to char array conv. mps result
char mpsSend[6] = {NULL};       // Used for inserting a W into the result
int j = 0;                      // for loop counting variable
int k = 0;                      // for loop counting variable
int sendSpeed = 0;              // How often to transmit data
float dataAvg = 0;              // Windspeed avg. for extreme value sorting
long cnt1 = 0;                  // Used to control the transmission speed




/////////// GET CURRENT WINDSPEED FROM ANEMOMETER //////////////////////////////
void getWindspeed(void)
    {

    /////////// GET ADC CONVERSION RESULT ///////////////////////////////////////
    // MCC generated ADC initialization routine

    ADC1_Enable();
    ADC1_ChannelSelect(channel);
    ADC1_SoftwareTriggerEnable();
    for(i=0;i <1000;i++)            // Provide a small delay
    {
    }
    ADC1_SoftwareTriggerDisable();
    while(!ADC1_IsConversionComplete(channel));
    conversion = ADC1_ConversionResultGet(channel);
    ADC1_Disable();

    /////////// ADC RESULT = HIGH/LOW ///////////////////////////////////////////
    // Decide if the ADC conversion should be interpreted as 1 or 0
    if(conversion > ADCresHIGH)
        {
        ADCresult = 1;
        }

    if(conversion <= ADCresLOW)
        {
        ADCresult = 0;
        }


    /////////// HALF TURN TIMING ////////////////////////////////////////////////
    // Using timer interrupt to time a anemometer half turn
    if(ADCresult == 1 && switchVar1 == 0)   // Hall sensor = HIGH
        {
        intrrCnt2 = tmr1intrrCnt2;  // 2nd half turn interrupt count update
        tmr1intrrCnt1 = 0;          // Begin 1st half turn interrupt count
        switchVar1 = 1;
        switchVar2 = 1;

        if(switchVar3 == 1 && halfTrn2cnt < 2) // Make sure only ONE half turn
            {                                  // has passed
            intrrCnt2ready = 1;     // 2nd half turn interrupt count ready
            switchVar3 = 0;
            halfTrn2cnt = 0;
            }
```

```
        if(halfTrn2cnt >= 2)
            {
            intrrCnt2ready = 0;     // 2nd half turn interrupt count NOT ready
            halfTrn2cnt = 0;        // Restart half turn count
            }
        halfTrn1cnt++;
        }

    if(ADCresult == 0 && switchVar1 == 1 && switchVar2 == 1)// Hall sensor = LOW
        {
        intrrCnt1 = tmr1intrrCnt1;  // 1st half turn interrupt count update
        tmr1intrrCnt2 = 0;          // Begin 2nd half turn interrupt count
        switchVar1 = 0;
        switchVar2 = 0;
        switchVar3 = 1;

        if(halfTrn1cnt < 2)         // Make sure only ONE half turn has passed
            {
            intrrCnt1ready = 1;     // 1st half turn interrupt count ready
            halfTrn1cnt = 0;
            }
        if(halfTrn1cnt >= 2)
            {
            intrrCnt1ready = 0;     // 1st half turn interrupt count NOT ready
            halfTrn1cnt = 0;        // Restart half turn count
            }
        halfTrn2cnt++;
        }


    /////////// WINDSPEED CALC ///////////////////////////////////////////////////
    if(intrrCnt1ready == 1)             // Calculate turn time if count is ready
        {
        intrrCnt1ready = 0;
        trnTime = 2*(intrrCnt1 * intrrTime);    // 2x because of half turn
        intrrCnt1 = 0;
        trnTimeReady = 1;               // Turn time is ready
        }

    if(intrrCnt2ready == 1)             // Calculate turn time if count is ready
        {
        intrrCnt2ready = 0;
        trnTime = 2*(intrrCnt2 * intrrTime);    // 2x because of half turn
        intrrCnt2 = 0;
        trnTimeReady = 1;               // Turn time is ready
        }

    if(trnTimeReady == 1)
        {
        if(trnTime != 0)                // Avoid division by zero
            {
            tps = 1/trnTime;            // Turns per second
            }
        else
            {
            tps = 0.0001;
            }
        wsUnc = trnDist * tps;          // Uncalibrated Windspeed
        mps = wsUnc * calibFact;        // Calibrated windspeed in m/s
        trnTimeReady = 0;
```

```
        mpsReady = 1;                   // Windspeed is ready for transmission
        }
    }



/////////// TRANSMIT CURRENT WINDSPEED /////////////////////////////////////////
void transmitMps(void)
    {
    cnt1++;

    /////////// DETERMINE HOW OFTEN TO TRANSMIT DATA //////////////////////////
    // cnt1 is used as a simple timer that counts one up every time to avoid
    // using timer interrupt resources on something that does not need it.
    sendSpeed = -383 * mps + 25000;    // Determine transmission interval from
                                       // current windspeed (mps))
    if(sendSpeed > 25000)              // Limit thelowest transmission speed
        {                              // to about every 15 seconds if windspeed
        sendSpeed = 25000;             // is 0 m/s
        }
    if(sendSpeed < 2000)               // Limit the highest transmission speed
        {                              // to about every 1 second if windspeed
        sendSpeed = 2000;              // is 60 m/s
        }


    if(testMode == 1)                  // If the system is in test mode 1 it
        {                              // will transmit more frequently
        sendSpeed = 1000;
        }


    /////////// TRANSMIT DATA ////////////////////////////////////////////////
    // Transmit about every 10 seconds at low windspeeds and about every second
    // at high windspeeds with a fluent transition between low and high
    // windspeeds.
    if(mpsReady == 1 && cnt1 > sendSpeed)
        {
        cnt1 = 0;
        sprintf(mpsArr,"%lf",mps);      // Convert double to char array (string)
        mps = 0;

        /////////// PREPARE AND TRANSMIT DATA ////////////////////////////////
        // Inserting a W in front of the windspeed data to sort good data from
        // bad and to be able to send and process other types of data later on
        for(k=1;k<7;k++)
            {
            mpsSend[0] = 'W';
            mpsSend[k] = mpsArr[k-1];
            }

        Send_Data_NRF(mpsSend);         // SPI to NRF module

        for(k=0; k<=sizeof(mpsSend); k++)
            {
            mpsSend[i] = '0';           // Reset the array
            }

        if(testMode == 1)               // Used for testing the system
            {
```

```
            printf("%s\n",mpsSend);             // UART for test
            LATBbits.LATB5 = !LATBbits.LATB5;   // Control LED
            }
        if(testMode == 2)               // Used for testing the system
            {
            printf("%s\n",mpsSend);             // UART for test
            }
        if(testMode == 3)               // Used for testing the system
            {
            LATBbits.LATB5 = !LATBbits.LATB5;   // Control LED
            }
        }
    }
```

A.2 anemometer.h

```c
/*
 *
 * Author(s):   Kim Holmberg Christensen
 *              Jørgen Drelicharz Greve
 *
 * Filename:    anemometer.h
 * Version:     1.0
 * Date:        16.05.2021
 *
 */

#ifndef ANEMOMETER_H
#defineANEMOMETER_H

#ifdef__cplusplus
extern "C" {
#endif

void getWindspeed(void);    // Calculate windspeed from ADC data
void transmitMps(void);     // Transmit windspeed

#ifdef__cplusplus
}
#endif

#endif/* ANEMOMETER_H */
```

A.3 NRF.c

```c
/*
 *
 * Author(s):   Kim Holmberg Christensen
 *              Jørgen Drelicharz Greve
 *
 * Filename:    NRF.c
 * Version:     1.0
 * Date:        16.05.2021
 *
 */

#define    FCY    (_XTAL_FREQ/2)

#include <xc.h>
#include <libpic30.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdint.h>
#include <stdbool.h>

#include "mcc_generated_files/system.h"
#include "mcc_generated_files/pin_manager.h"
#include "mcc_generated_files/spi1.h"
#include "NRF.h"



void Write_NRF(char reg, char value)
    {

    CSN_lo
    SPI1_Exchange8bit(W_reg | reg);
    SPI1_Exchange8bit(value);
    CSN_hi

    }

char Read_NRF(char reg)
    {
    char RB;
    CSN_lo
    SPI1_Exchange8bit(R_reg | reg);
    RB = SPI1_Exchange8bit(0x00);
    CSN_hi

    return RB;

    }

void Write_Buffer_NRF(char dest, char * buffer, char amount_bytes)
    {
    char i;

    CSN_lo
    SPI1_Exchange8bit(W_reg | dest);
    for (i = 0; i < amount_bytes; i++)
    {
```

```
        SPI1_Exchange8bit(*buffer);
        buffer++;
        __delay_us(10);
    }
    CSN_hi

    }

void Read_Buffer_NRF(char dest, char * buffer, char amount_bytes)
    {
    char i;

    CSN_lo
    SPI1_Exchange8bit(R_reg | dest);
    for (i = 0; i < amount_bytes; i++)
    {
        *buffer = SPI1_Exchange8bit(0xFF);
        buffer++;
    }
    *buffer = (char)NULL;
    CSN_hi

    }

void Send_Data_NRF(char * buffer)
    {

    Write_Buffer_NRF(Write_TX_PLD, buffer, 5);
    CE_hi;
    __delay_us(200);
    CE_lo;

    }

void NRF_FLUSH(void)
{
    CSN_lo
    Write_NRF(STATUS, 0x70);
    __delay_ms(10);
    CSN_hi

    CSN_lo
    SPI1_Exchange8bit(0xE1); // Flush TX
    __delay_ms(10);

    CSN_hi

    CSN_lo
    SPI1_Exchange8bit(0xE2); // Flush RX
    __delay_ms(10);
    CSN_hi
}

void Read_Data_NRF(char *buffer)
    {

    Read_Buffer_NRF(Read_RX_PLD, buffer, 5);
    Write_NRF(STATUS, 0x70);
    NRF_FLUSH();

```

```
    }

char Data_Ready_NRF(void)
    {
    if((Read_NRF(STATUS) & 0x40) == 0x40)
    {
        return 0;

    }
    return 1;
    }


void startup_NRF_delay(void)
    {
    __delay_ms(500);
    }

void pwr_up_NRF(void)
    {

     Write_NRF(CONFIG, PWR_UP);
    //SPI1_Exchange8bit(0x20 | 0x00);
    //SPI1_Exchange8bit(0x02);
    __delay_ms(2);

    }

void config_NRF(void)
    {
    char Adress[5] = {0xe1,0xe1,0xe1,0xe1,0xe1};
    Write_NRF(CONFIG, 0x0B); // Setup Config register
    Write_NRF(EN_AA, 0x00); // Enable auto ack data pipe0
    Write_NRF(EN_RXADDR, 0x01); // Enable data pipe0
    Write_NRF(SETUP_AW, 0x03); // Adress width 3 bytes
    Write_NRF(SETUP_RETR, 0x00);
    Write_NRF(RF_CH, 0x32); //Set RF channel F0 = 2400+RF_CH MHz F0 = 2450 Mhz
    Write_NRF(RF_SETUP, 0x0f);
    Write_Buffer_NRF(W_reg | RX_ADDR_P0, Adress, 5);
    Write_Buffer_NRF(W_reg | TX_ADDR, Adress, 5);
    Write_NRF(RX_PW_P0, 0x05);

    /********************************Set as RX******************************************/
    //__delay_us(10);
    //Write_NRF(CONFIG, 0x0B);
    //CE_hi;
    //__delay_us(10);
    /***********************************************************************************/
    /********************************Set as TX******************************************/
    __delay_us(10);
    Write_NRF(CONFIG, 0x0A);
    CE_lo;
    __delay_us(10);

    /***********************************************************************************/
    }
```

A.4 NRF.h

```c
/*
 *
 * Author(s):   Kim Holmberg Christensen
 *              Jørgen Drelicharz Greve
 *
 * Filename:    NRF.h
 * Version:     1.0
 * Date:        16.05.2021
 *
 */

#include "mcc_generated_files/system.h"
#include <libpic30.h>

#define CSN_lo LATBbits.LATB10 = 0;
#define CSN_hi LATBbits.LATB10 = 1;
#define CE_lo LATBbits.LATB9 = 0;
#define CE_hi LATBbits.LATB9 = 1;
#define PWR_UP          0x02
#define TX_mode         0x0A
#define RX_mode         0x0B
#define W_reg           0x20
#define R_reg           0x00
#define Write_TX_PLD    0xA0
#define Read_RX_PLD     0x61

//Register adresses NRF
#define CONFIG          0x00
#define EN_AA           0x01
#define EN_RXADDR       0x02
#define SETUP_AW        0x03
#define SETUP_RETR      0x04
#define RF_CH           0x05
#define RF_SETUP        0x06
#define STATUS          0x07
#define OBSERVE_TX      0x08
#define CD              0x09
#define RX_ADDR_P0      0x0A
#define TX_ADDR         0x10
#define RX_PW_P0        0x11

void Write_NRF(char reg, char value);

char Read_NRF(char reg);

void Write_Buffer_NRF(char dest, char * buffer, char amount_bytes);

void Read_Buffer_NRF(char dest, char * buffer, char amount_bytes);

void Send_Data_NRF(char * buffer);

void NRF_FLUSH(void);

void Read_Data_NRF(char *buffer);

char Data_Ready_NRF(void);

void startup_NRF_delay(void);

void pwr_up_NRF(void);
```

```
void config_NRF(void);
```

A.5 anemometer main.c

```c
/*
 *
 * Author(s):   Kim Holmberg Christensen
 *              Jørgen Drelicharz Greve
 *
 * Filename:    main.c
 * Version:     1.0
 * Date:        16.05.2021
 *
 */

#include "mcc_generated_files/system.h"
#include "mcc_generated_files/adc1.h"
#include "NRF.h"
#include "anemometer.h"


int main(void)
{
    SYSTEM_Initialize();        // MCC initialize
    startup_NRF_delay();        // SPI startup delay
    pwr_up_NRF();               // Start the NRF module
    config_NRF();               // Configure the NRF module
    ADC1_Initialize();          // Initialize ADC1

    while (1)
    {
        getWindspeed();         // Calculate windspeed from ADC data
        transmitMps();          // Transmit windspeed
    }

    return 1;
}
```

A.6 basestation main.c

```c
/*
 *
 * Author(s):   Kim Holmberg Christensen
 *              Jørgen Drelicharz Greve
 *
 * Filename:    main.c
 * Version:     1.0
 * Date:        16.05.2021
 *
 */


#include "mcc_generated_files/system.h" // MCC files
#include "NRF.h"
#include <stdio.h>


int main(void)
{
    SYSTEM_Initialize();            // MCC initialize
    startup_NRF_delay();            // To let SPI get ready for TX
    pwr_up_NRF();                   // Power up NRF
    config_NRF();                   // Configure NRF

    char data_in[5] = {NULL};       // NRF received data is stored here
    long cnt = 0;                   // For small delay
    int dataRead = 0;               // For LED toggle on data receive

    while (1)
    {

        if(cnt > 1000)  // Adds a small delay for stability
            {

            /////////// RECEIVE DATA FROM NRF //////////////////////////////////
            while(Data_Ready_NRF())         // Stay while NRF data ready
                {
                dataRead = 1;
                }

            Read_Data_NRF(data_in);         // Read NRF data to data_in

            if(dataRead == 1)               // LED toggle when data is received
                {
                dataRead = 0;
                LATBbits.LATB5 = !LATBbits.LATB5;
                }

            /////////// WIND DATA PROCESSING ///////////////////////////////////
            if(data_in[0] == 'W')           // Select only wind data
                {
                printf("%s \n",data_in);    // Send data to ESP8266 wifi
                }

            cnt = 0;
            }

        cnt++;
    }

```

```
    return 1;
}
```

A.7 esp8266.ino

```
/*
  Rui Santos
  Complete project details at https://RandomNerdTutorials.
com/esp32-esp8266-mysql-database-php/

  Permission is hereby granted, free of charge, to any person obtaining a copy
  of this software and associated documentation files.

  The above copyright notice and this permission notice shall be included in
all
  copies or substantial portions of the Software.

*/



#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#include <WiFiClient.h>
#include "secrets.h"

// Replace with your network credentials
const char* ssid     = SECRET_SSID;   // SSID (see secrets.h)
const char* password = SECRET_PASS;   // Pswrd (see secrets.h)

// REPLACE with your Domain name and URL path or IP address with path
const char* serverName =
"http://jorgengreve.dk/projects/weatherstation/post-esp-data.php";

// Keep this API Key value to be compatible with the PHP code provided in the
project page.
// If you change the apiKeyValue value, the PHP file /post-esp-data.php also
needs to have the same key
String apiKeyValue = "tPmAT5Ab3j7F9";

String sensorName = "Windspeed";
String dataIn;
String data;

int httpResponseCode;

unsigned int timeNow = 0;
unsigned int lastTime = 0;
unsigned int delayed = 5000;
unsigned int dataReady = 0;

void setup() {
  Serial.begin(115200);
```

```
  WiFi.begin(ssid, password);
  Serial.println("Connecting");

  while(WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.print("Connected to WiFi network with IP Address: ");
  Serial.println(WiFi.localIP());
}


void loop() {

  while(Serial.available()) {
    data = Serial.readString(); // read the incoming data as string
    dataReady = 1;
  }


  //Check WiFi connection status
  if(WiFi.status()== WL_CONNECTED)
  {
    if(dataReady == 1)
    {
      data.remove(0,1);   // Removes the 'W' from the windspeed data

      HTTPClient http;

      // Your Domain name with URL path or IP address with path
      http.begin(serverName);

      // Specify content-type header
      http.addHeader("Content-Type", "application/x-www-form-urlencoded");

      // Prepare your HTTP POST request data
      String httpRequestData = "api_key=" + apiKeyValue + "&sensor=" +
sensorName + "&value1=" + String(data) + "";

      Serial.println(httpRequestData);

      // Send HTTP POST request
      httpResponseCode = http.POST(httpRequestData);

      if (httpResponseCode>0) {
        Serial.print("HTTP Response code: ");
```

```
        Serial.println(httpResponseCode);
      }
      else {
        Serial.print("Error code: ");
        Serial.println(httpResponseCode);
      }

      // Free up resources
      http.end();

      dataReady = 0;
    }
  }
  else {
    Serial.println("WiFi Disconnected");
  }
}
```

A.8 post-esp-data.php

```php
<?php
/*
  Rui Santos
  Complete project details at https://RandomNerdTutorials.com/esp32-esp8266-mysql-database-php/

  Permission is hereby granted, free of charge, to any person obtaining a copy
  of this software and associated documentation files.

  The above copyright notice and this permission notice shall be included in all
  copies or substantial portions of the Software.
*/

$servername = "localhost";

// REPLACE with your Database name
$dbname = "jorgengreve_dkesp8266_data";
// REPLACE with Database user
$username = "jorgengreve_dkesp8266_data";
// REPLACE with Database user password
$password = "X#qe#rf=Tz3n)7Aw]D#.:";

// Keep this API Key value to be compatible with the ESP32 code provided in the project page.
// If you change this value, the ESP32 sketch needs to match
$api_key_value = "tPmAT5Ab3j7F9";

$api_key= $sensor = $value1 = "";

if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $api_key = test_input($_POST["api_key"]);
    if($api_key == $api_key_value) {
        $sensor = test_input($_POST["sensor"]);
        $value1 = test_input($_POST["value1"]);

        // Create connection
        $conn = new mysqli($servername, $username, $password, $dbname);
        // Check connection
        if ($conn->connect_error) {
            die("Connection failed: " . $conn->connect_error);
        }
        $sql = "INSERT INTO SensorData (sensor, value1)
        VALUES ('" . $sensor . "', '" . $value1 . "')";

        if ($conn->query($sql) === TRUE) {
            echo "New record created successfully";
        }
        else {
            echo "Error: " . $sql . "<br>" . $conn->error;
        }
        $conn->close();
    }
    else {
        echo "Wrong API Key provided.";
    }
}
else {
    echo "No data posted with HTTP POST.";
}

function test_input($data) {
    $data = trim($data);
    $data = stripslashes($data);
    $data = htmlspecialchars($data);
    return $data;
}
```