Design of an Autonomous Wireless Weather Station

EE3L11- Bachelor Graduation Thesis

T.A. Brasser
D.J. Offerhaus
I.P. Tesselaar

Delft University of Technology

ABSTRACT

With increasingly changing climates, measuring the weather on remote, off-grid locations has become more important in order to accurately track emerging weather patterns. Cheap and easy access to meteorological information allows communities and individuals to better manage agricultural development, track localised extreme weather events, and expand knowledge of ground-true weather conditions for remote or hard to access locations. Nowadays, with the growth of modern technology and the rise of the Internet-of-Things, it has become possible to design a long range, low maintenance, affordable, continuously operating autonomous wireless weather station powered off-grid by renewable resources. This thesis explores the feasibility of such a self-supporting wireless station, and attempts to construct a prototype to support the findings. A multitude of solutions are discussed to tackle the different subsystems and their functions, culminating in a proposal for a weather station design. Low maintenance sensors are used to acquire weather data and LoRaWAN is used to send that data over long ranges with a low power draw. The station uses solar energy and li-ion batteries to sustain the system, which are scaled to prevent loss of power. A simple microcontroller unit manages the power and data flows of the system. Long term simulations of the design show the proposed system would be able to operate continuously throughout the year, while measuring every minute. The report also discusses the apparent strengths and shortcomings of the system, and suggests potential improvements that could help the system achieve a better long-term performance.

Contents

1

INTRODUCTION

Before the rise of modern telecommunication technologies, the only way to acquire data on outdoor activities was to physically monitor the events. The rise of low-power, long-range communication systems [1] and the "Internet of Things" (IoT) [2] has made it possible to collect data from autonomous sensors without human presence and without a wired power- and communication infrastructure. The goal of this project is to synthesise such a weather system.

1.1. STATE-OF-ART ANALYSIS

1.1.1. HISTORY AND APPLICATIONS

The history of autonomous wireless weather stations goes all the way back to 1939, when Brooks (1940) and Wood (1946) built an instrumental shelter on top of a cabin. The instruments were powered by a 1000 W, 115 V petrol generator. The size of the fuel tank enabled it to make 8 observations per day for 4 months, which were transmitted back via radio [3].

Modern day autonomous wireless weather stations or more commonly called Automatic Weather Stations (AWS) are present in two main areas of application. The first one is in the form of Personal Weather Stations (PWS), which are mainly directed at amateur meteorologists. The second is in the broader form of Wireless Sensor Nodes (WSN), most commonly used for research. These WSN's can usually capture one or several environmental parameters and are usually deployed in great numbers [4]. By having a lot of autonomous wireless nodes, researchers can gather data about an environment at a rate never before seen. With the rise of Internet-of-Things (IoT) applications and autonomous systems, it has become easier to extract data from the environment [5].

Data collected by wireless sensors or network of sensors have a very wide range of applications [6, 7], for example air pollution measurement in big cities, (soil) humidity measurements for farmers and aiding in prediction models for the weather. With the rise of global warming and the changing of the weather, conventional wisdom on the timing of crop rotation and periods of rain can no longer be relied upon. It is now more important than before to measure these weather parameters.

1.1.2. CURRENT STATUS

There is a large amount of slightly different weather stations available for almost any modern application. Two common examples of consumer oriented products are

1. short range (~ 30 m) models where a battery powered sensor node uses Wi-Fi or short-range RF to display measurements on a nearby LCD;
2. models using large rechargeable battery packs or grid power. A datalogger is used to store and/or send data over cellular, long-range, networks;

While the essential design does not differ greatly, the diversity of available sensor, communication and power supply combinations result in a large variety of types and models. These designs are oriented toward having

the weather station in relative close proximity to a power grid and network access. This implies people use it to have easily available data on weather parameters in a nearby area. Another application area is amateur meteorology, making use of data loggers, which don't necessarily communicate data as it becomes available but can store large quantities of data on the device. Mechanical rain sensors and wind gauges commonly show up in these models. Although these provide adequate measurements, but are maintenance intensive relative to alternative (solid state) sensor types and can break down more easily due to wear and tear.

The range of weather station solutions is even more diverse when looking at business oriented models, especially for agricultural businesses. Being able to keep track of precipitation, frost and soil data provides valuable information on crop growth and yield predictions. Modern agriculture in many countries also has grown in how much area a single farm manages. If an entrepreneur needs to keep track of a large amount of crops over an increasingly large area, a distributed sensor network can help in gathering and processing more information than a human observer will ever be able to. Weather stations in this category employ a wider range of measurements than those targeted at households. Models exist which, in addition to using just a battery, extend operational capabilities by including solar panels. Mechanical sensors are still common, but these are often more sturdy and failure proof than the consumer counterparts. Non-mechanical alternatives also exist. Although existing business models expand the application possibilities of weather stations [8], compared to consumer oriented models, these business oriented models are significantly more expensive, especially with the inclusion of better and more sensors, long range deployment, or self-sufficient power supply.

1.2. PROBLEM DEFINITION

So far technology of weather stations has greatly supported the field of meteorology by providing solutions to measure and predict weather patterns. As wireless communication improve and sensors become cheaper, it is now relatively easy and cheap to acquire a sensor station to measure relevant weather parameters on a location. This makes it possible for people to rely on quantified weather information instead of estimates and predictions for weather data. This is a great advantage to developing agriculture, observing climate, or tracking unpredictable localised weather. Current weather stations however have included design decisions that restrict the freedom of application. The location where such a station can be deployed is restricted considerably by including sensors that need to be maintained regularly, requiring the station to run on grid power, or using a telecommunications network that has limited coverage in rural areas. More professional solutions can be deployed at a larger range, but the additional costs associated can be significant, while coverage and maintenance can still be issues that reduce the effectiveness of a station in a remote location.

The benefits of these weather sensing solutions could be much greater to developing communities and other research opportunities. A solution is to provide simple but accurate weather measurement were designed specifically to run autonomously without power grid, capable of long range communication in remote areas and have as few possible ways of requiring regular maintenance unless failure occurs. In that case, a large section of locations that before were too ineffective or expensive for placing a weather station could then also be observed accurately for the benefits that weather stations can provide.

To this end, this project has been proposed to explore the design of a weather station specifically for long distance, off grid, maintenance low operation. If successful, this research could be expanded upon by others to make further improvements and boost progress toward having a cheap and robust solution for people and initiatives that so far have not been able to profit off the useful information such a station provides. The proposal was put forth by ir. R.M.A. van Puffelen and the Electronic Instrumentation department at the Delft University of Technology as part of the Bachelor Graduation Project. As part of the accepted proposal, the research will also be used to construct a functional prototype to confirm the viability of the design choices.

There are three main objectives to tackle in this project. First, the large design space for weather stations should be explored to find which component solutions improve on the current models' weaknesses as described here. Second, to design a new weather station that runs as autonomously as possible while minimising costs of the total system. Lastly, to develop a prototype that matches the final design as closely as possible in order to validate it. After discussing with the project coordinators, agreements have been made on the expected functions and conditions the design and prototype should take into account. The

design will feature different types of sensors, as the project will explore measuring the weather parameters of air temperature, humidity, atmospheric pressure, wind speed, wind direction and precipitation volume. To allow the department to use the research and prototype effectively, it was also decided that the system would be expected to run in the Netherlands. Finally, the system is to be designed to be able to gather a set of measurements every minute. The effectiveness of the final design can be assessed in several manners, but for any design to be successful it should obey the conditions set, look to support autonomy by maximising the mean time to failure of individual components and the system as a whole, and minimise its power consumption in order to operate off-grid effectively.

1.3. THESIS SYNOPSIS

The rise of the Internet of Things and increasingly affordable renewable energy make it possible to design an autonomous wireless weather station, which operates long range and on low power. Modern weather stations commonly operate connected to the power grid, use high maintenance sensors or use power intensive technologies such as Wi-Fi or Cellular communication. In this thesis, the feasibility of a self-supporting and wireless weather station is explored. The constraints and requirements of this design are charted. The design process consists of three stages and begins with the conceptual design. In it, all the possible, feasible solutions are compared, of which a conceptual design is chosen. The next step, the embodiment design, will finalise the choice of components, as well as the connections between the components and the power draw of the system. The detailed design will scale the energy harvesting with the power draw calculated, as well as feature software to manage its energy and any other details regarding the design. A final design is proposed, after which a prototype is built to test that design for continuous operation of the system.

2

PROGRAMME OF REQUIREMENTS

The list of requirements follows from the problem definition in 1.2. In short, the final version of the "Autonomous Wireless Weather Station" must be able to

- measure and collect data from weather parameters;
- transfer collected data without wires;
- collect and use energy without being connected to the power grid;
- efficiently distribute energy in order to operate continuously;
- require no regular maintenance when operating within specified weather conditions;
- have a high mean time to failure when operating within specified weather conditions.

From the first three items of the list, a list of mandatory requirements directly follows. For this specific design, the weather parameters that need to be measured are temperature, air pressure, humidity, amount of precipitation, wind speed and wind direction. Continuous operation is defined in the mandatory requirements. The latter two items of the list result in trade-off requirements.

2.1. MANDATORY REQUIREMENTS

2.1.1. FUNCTIONAL REQUIREMENTS

[1.1] The system must be able to collect temperature data from its environment.

[1.2] The system must be able to collect air pressure data from its environment.

[1.3] The system must be able to collect humidity data from its environment.

[1.4] The system must be able to collect precipitation data from its environment.

[1.5] The system must be able to collect wind speed data from its environment.

[1.6] The system must be able to collect wind direction data from its environment.

[1.7] The system must be able to transfer data to an external database without wires.

[1.8] The system must be able to operate without being connected to a power grid.

2.1.2. FUNCTIONAL REQUIREMENTS REGARDING WEATHER MEASUREMENTS

[2.1] The system must be able to collect environmental data regarding requirement [1.1] to [1.4] at least once per minute.

[2.2] The system must be able to collect wind data regarding requirement [1.5] and [1.6] five consecutive seconds per minute, averaging the collected data.

[2.3] The system must be able to collect temperature data from its environment within the range of −50 to 70 °C.

[2.4] The system must be able to collect temperature data from its environment, with a resolution of 0.04 °C.

[2.5] The system must be able to collect temperature data from its environment with an accuracy of at least 0.5 °C.

[2.6] The system must be able to collect air pressure data from its environment within the range of 900 to 1100 hPa.

[2.7] The system must be able to collect air pressure data from its environment with an accuracy of 4 hPa.

[2.8] The system must be able to collect air pressure data from its environment with an resolution of 1.5 Pa.

[2.9] The system must be able to collect humidity data from its environment with a range between 0 and 100 %RH.

[2.10] The system must be able to collect humidity data from its environment with an accuracy of 3.5 %RH.

[2.11] The system must be able to collect humidity data from its environment with a resolution of 0.03 %RH.

[2.12] The system must be able to collect the amount of precipitation data from its environment above a temperature of 0 °C.

[2.13] The system must be able to measure a maximum wind speed of 25 m/s.

[2.14] The system must be able to collect wind speed data with a relative accuracy of 1 %.

[2.15] The system must be able to collect wind direction data between 0° and 359°.

[2.16] The system must be able to collect wind direction data with an absolute accuracy of 1°.

2.1.3. COST REQUIREMENTS

[3.1] The maximum production cost of the system should not exceed €300,-[1].

[3.2] The maximum prototype cost of the system should not exceed €500,-[2].

2.1.4. SAFETY, ETHICAL AND ENVIRONMENTAL REQUIREMENTS

[4.1] The system should be able to operate in the Netherlands.

[4.2] The system should not weigh more than 10 kg, in order to be deployable by a single person.

[4.3] The packaged system should be able to fit in a box of 0.5 m by 0.33 m by 0.33 m, the average dimensions of a moving box.

[1] In order to remain economically competitive.
[2] As agreed upon with the project coordinators.

[4.4] The system should be able to installed by an individual with no previous experience, when supported with basic instructions.

[4.5] The system should not contaminate the environment while deployed by adhering to environmental law.

[4.6] The system must be removed from the environment when broken or disfunctional.

[4.7] The system must be disassembled into its constituent parts to allow separate processing for recycling and disposing.

2.2. TRADE-OFF REQUIREMENTS

2.2.1. AUTONOMY REQUIREMENTS

[5.1] The system should guarantee continuous functioning by minimising the chance the system fails due to not being able to autonomously supply enough power to operate.

[5.2] The system should require the least amount of maintenance possible within the specified lifetime.

[5.3] The system should have a lifetime of at least 3 years.

3

DESIGN PROCESS

The design process used follows the design procedure of [9], which describes a systematic approach to engineering design. The process can be split up in three phases: conceptual design, embodiment design and detailed design.

CHAPTER OVERVIEW

The conceptual design describes the basic functions of the system and divides the system into subsystems. Each solution of the subsystem is reviewed for the relevant criteria and a conceptual design is created. The embodiment design goes deeper into the design process. It takes into account estimations for the expected loads, as well as a preliminary choice of components and the layout between those. In the detailed design, the system is finalised. In that stage, software will regulate the data flows and the power flows after which the rest of the system is dimensioned. The chapter concludes with a look on the final design specifications.

3.1. CONCEPTUAL DESIGN

One approach to visualise and present implementations or 'solutions' is by the use of a morphological chart, in which the columns represent various problems or functions that are under consideration, which are filled with any possible implementation that would fulfil the individual function or solve the individual problem [9]. A combination of individual solutions is required for the complete design. As a result, combinations of the individual subsystem implementations that are incompatible are undesired and can be eliminated. Furthermore, similar subsystem implementations can be compared to each other. By weighing the positive and negative aspects of the implementations, a division can be made between implementations that are more consistent with the desired system properties formulated in chapter 2 or less consistent with those properties. The initial task is to formulate all conceivable solutions that could separately fulfil the functions of the subsystems.

3.1.1. SYSTEM ANALYSIS

The design process begins with identifying what the system has to do and what the relevant inputs and outputs are. In principle, the system has to collect weather data continuously and send that weather data without human interaction. Since the system has to collect data, it has to (temporary) store that data. In order for the system to work autonomously, it has to collect energy from its surrounding to support its functions. Furthermore, if the system has to operate continuously, the system has to store the energy and distribute that energy for when the source of energy is not available or unpredictable. A conceptual system overview of the system with the specified subsystems can be found in figure 3.1. In table 3.1, a morphological table has been constructed with a list of viable solutions for each of the subsystems.

In this design approach, the sensor platform, which gathers energy and sensor data, and transmit that data, can be viewed as separate from the sensors that could be attached to it. As long as the energy gathering and storage is sufficient for the load, the platform should be able to operate sensors for any number of measurements. Considering this, completing the weather station requirement is mainly determined by the properties of the sensors, instead of the platform. In order to not constrain the design of the platform,

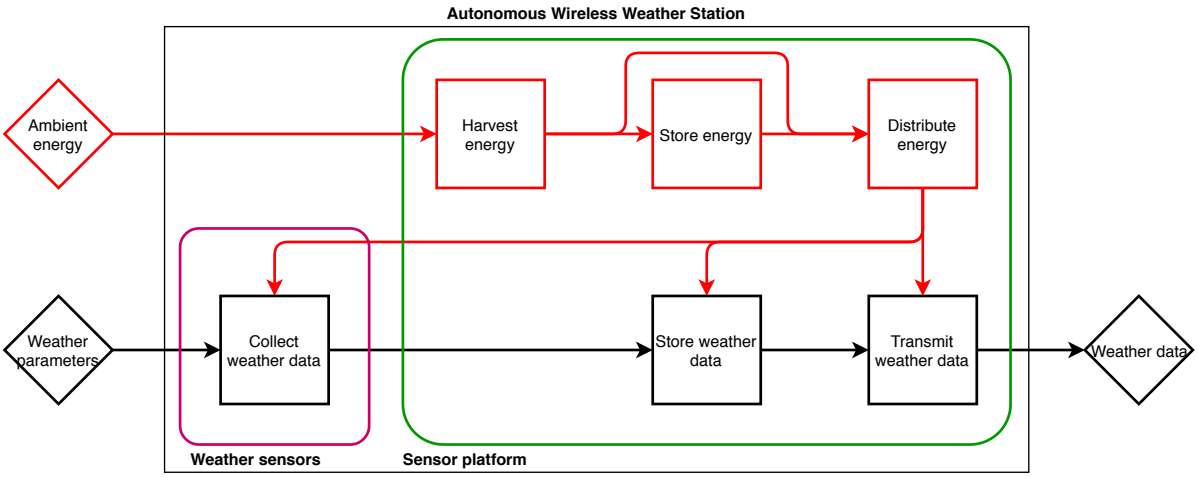


Figure 3.1: A diagram describing the conceptual system overview of the Autonomous Wireless Weather Station divided in a set of subsystems with the expected inputs and outputs. The system can be split up into two sets of subsystems: the sensors which measure the weather parameters and the platform which houses these sensors. The red lines represent power flows, the black lines represent data flows.

its subsystem solutions will be graded and selected separate from the sensors attached. In doing so the criteria for the platform and sensors can be more accurately tuned to what purpose they serve. Impactful choices are mostly the type of microcontroller and board. This leaves a very wide selection of options where a broad set of criteria and grades does not provide a conclusive answer. The selection and management of the microcontroller will be an important part in the later embodiment and detailed design and get discussed later in those chapters.

To summarise, in this section subsystem solutions will be graded using a weighted list of criteria in order to find one or more sets of options that are better suited to the design. Because of how the different subsystems interact and can be looked at as separate, the consideration is first made for the components of the sensor platform: the energy harvesting, storage, data storage and data transmission subsystems. For the sensor types a similar set of criteria will also be used to select a set of preferred sensors. Since the expected power draw of the rain sensor and the wind sensor are much higher than those of the temperature sensor, humidity sensor and air pressure sensor, the latter three are not considered here as major design choices, as explored in section 3.2.5.

Table 3.1: Morphological table of the possible solutions to the different subsystems.

Sensor Platform				Weather Sensors	
Energy		Data		Sensing	
Harvesting	Storage	Communication	Storage	Wind	Precipitation
Solar	Battery	Cellular	On chip	Mechanical	Tipping bucket
Wind	Super capacitor	LPWAN	External	Thermal Solid-State	Disdrometer
Hybrid	Hybrid	Wi-Fi		Sonic Solid-State	Optical
					Sonar

3.1.2. CONCEPTUAL DESIGN OF THE SENSOR PLATFORM

ENERGY HARVESTING AND STORAGE

The main function of the energy harvesting subsystem is to generate enough energy to let the system run by only using ambient energy. Possible forms of energy that can be harvested from the environment include wind, solar, vibrations, streaming water or RF. The considered systems are graded in table 3.2. These sources are ranked according to their expected predictability, maintenance, lifetime, and cost.

Table 3.2: Comparison of the energy harvesting options.

	Weight	Solar	Wind	Hybrid (Wind/Solar)	Vibration	Streaming water	RF
Availability	20 %	4	5	5	1	2	1
Predictability	30 %	4	1	5	1	4	2
Maintenance	25 %	4	3	2	4	1	5
Lifespan	20 %	5	3	3	4	1	4
Cost	5 %	3	3	2	4	1	4
Average Score		4.00	3.0	3.4	2.8	1.8	3.2
Weighted Score		4.15	2.8	3.7	2.5	2.1	3.05

As can be seen from the table, vibration and RF are rated with a low availability. RF will be more abundant in urban environments and less in rural ones. Moreover, the expected power that can be harvested from these sources is in the order of micro watts, which is expected to be too low for this application. Streaming water has a high potential for energy, but the source is highly location dependent and the application will likely need moving parts like turbines, which is not desired.

Solar energy harvesting with the use of a PV-cell has some advantages relative to a mechanical wind energy harvesting system. A PV-module has a typical lifetime of more than 10 yr [10] and has no moving parts, while a mechanical wind energy harvesting system has moving parts, making it more prone to wear. Solar energy is also more predictable than wind energy, making it easier to make prediction models for the energy distribution. Wind energy however is usually always present. A hybrid system is also considered, but complexity and cost of the system will rise when implementing two very different sources.

Storing energy is paramount when there is a desire to make the system operate continuously. The energy sources discussed are usually unpredictable, or predictable, but non-continuous, making the storage of energy a key feature. Available options of this subsystem include batteries, super capacitors or some mechanical solution. Fast charge and discharge are not necessary for this system, as the expected energy supply and draw are relatively low. The main concerns with storage are the cost and storage capacity, as well as lifetime and decay properties of the device. The comparison of different energy storing types is given in table 3.3.

Table 3.3: Comparison of the energy storage options.

	Weight	Battery	Supercap	Hybrid
Capacity	45 %	5	2	4
Lifespan	30 %	4	2	3
Cost	25 %	3	3	2
Average Score		4	2.33	3
Weighted Score		4.2	2.25	3.2

DATA TRANSFER AND STORAGE

The data gathered by the sensors will be transmitted to a database. However, before the data is transmitted it has to be stored (or buffered) somewhere. This can be done either on the chip or on an external storage medium. The fastest and least power-consuming way is to hold the data in RAM, but since RAM is volatile, a loss of power will result in data loss. Furthermore, while the amount of data buffered is not huge, the time

before data-transmission may prove to be too long to store all the sensor data. External (flash) storage can be used to overcome this limit, but every bit stored and read adds power losses that could be avoided by keeping data in volatile memory. Luckily, most SoC's (System on a Chip) come with on-board flash storage in addition to RAM, providing a cheap and readily available nonvolatile data buffer option. The choice of data storage is however not a significant choice for the sensor platform, since most available microprocessors and boards feature both memory systems. The power draw and costs associated with the memory are small compared to the other subsystems. The use of memory will be part of the chosen microprocessor and will therefore be discussed in section 3.3.

Collected data must be accessible by the end user without requiring them to be in the proximity of the device. Additionally, the device is required to be wireless, meaning a wireless communication module should transmit the data at which point it can be accessed through a conventional network (either via the internet or locally). Possible solutions include cellular communication in licensed bands and different Low-Power WAN (LPWAN) solutions operating mostly in unlicensed and sub-GHz bands. Properties of importance for these systems include power-draw and range. A relative comparison is tabulated in table 3.4. The use of existing (cellular) networks seems attractive since these provide ubiquitous and transparent coverage, however, these were not designed to handle the type of asymmetrical and low-power service that is optimal for Wide Sensor Area Networks (WSN) and Internet of Things (IoT) device deployment such as the weather station.[1]

Table 3.4: Comparison of communication systems.

	Cellular	LPWAN
Power consumption	2	5
Range	3	4
Average Score:	2.5	4.5

THE RESULTING SENSOR PLATFORM

The difficulty in any design is to make a choice concerning the many options. While better options may be available, the best option is chosen with the help of the morphological tables, while keeping in mind the complexity of the resulting system design. The resulting sensor platform is summarised in table 3.5.

For the energy harvesting, the solar panel is the best option according to table 3.2. This is the option which will be implemented. The solar panel os expected to provide a more reliable source of energy, as well as being more autonomous due to having no moving parts and a long expected lifetime.

The battery is the best option for the energy storage, according to table 3.3. The expected lifetime of a battery will be according to the requirements. A battery is better suited for long term energy storage than a supercapacitor, making it the better option for energy storage in combination with the solar panel. The battery will be implemented in the system.

The last design choice for the sensor platform is the communication system, which is graded in table 3.4. The desired choice is LPWAN. For this application, the power draw needs to be as low as possible and the required bit-rate will not have to be high. With the implementation of a base station, the range of a LPWAN-based system can be extended even further.

Table 3.5: The resulting sensor platform.

Subsystem	Design choice
Energy harvesting	Solar panel
Energy storage	Battery
Data transfer	LPWAN

3.1.3. CONCEPTUAL SENSOR CONFIGURATION

Sensors have a wide range of applications, data types and power draws, the specifics of which must be taken into further consideration for each relevant weather parameter. Considerations include the type of communication with the overall system, power draw when measuring and when not, start-up time, calibration requirements, size, average lifespan, accuracy of measurements and supply voltage. In the end, an optimal sensor would require the least amount of power per measurement when adding all possible additional draws combined with a minimum required accuracy of measurement. Sensors measuring temperature, humidity and air pressure have a relative low draw compared to wind speed and direction sensors and rainfall sensors, which can be a relative high draw on the system. The various implementation of the latter sensors have a significant effect on the criteria relevant to the system. When incorporating measurements for these parameters, comparing the various sensor types is essential for making a choice that does not put unnecessary strain on the system. The other sensors can be discussed during the embodiment design phase under the assumption that the design of the rainfall sensor and wind sensor both have a large impact on the final system. To that end, the weighted criteria can be applied specifically to the different types of precipitation and wind sensors in this conceptual stage, since deciding on the type of temperature, humidity and atmospheric pressure sensor has only marginal impact on the sensor loads as a whole.

PRECIPITATION SENSING

Starting with the precipitation sensing, following research on the current types of sensors used for rainfall measurements, 4 different types which could be viable in the application were identified. In order to select a sensor, the tipping bucket, optical solid state, acoustic disdrometer and radar solid state will be compared by using the criteria derived from chapter 2. In table 3.6, the four types of precipitation sensors are weighted according to which criteria is more important. Two sets of weighing factors are considered as the choice of prioritising the production cost or the fully autonomous properties is a decision with a large impact on the final design.

Table 3.6: Comparison of the precipitation sensors.

	Weight $\|_{\text{Cost}}$	Weight $\|_{\text{Autonomy}}$	Tipping bucket	Optical	Radar	Ac. Disdro
Power consumption	25 %	25 %	5	2	1	1
Cost	45 %	5 %	4	3	1	4
Maintenance	5 %	25 %	1	5	5	4
Lifespan	5 %	25 %	2	4	4	3
Accuracy	20 %	20 %	4	3	5	1
		Average Score	2.8	3.4	3	2.6
		Weighted Score $\|_{\text{Cost}}$	3.6	2.9	1.95	2.6
		Weighted Score $\|_{\text{Autonomy}}$	2.6	3.5	3.35	2.4

From the table it is apparent that due to the cost and power consumption, a radar based sensor is unlikely to be an optimal choice, as an optical sensor can have similar advantages but a lower cost and load. Early findings on the acoustic disdrometer indicated that these models have a high error between measured volume and actual volume of rainfall for smaller drops. While sensor stations with a disdrometer do exist, reference material for the sensor itself with respect its properties are difficult to find or unavailable. The consideration is then between a tipping bucket and an optical solid state sensor. The tipping bucket is a simple and cheap device, but its funnel collector for the rain means it needs to regularly be checked for clogging and a mechanical switch (reed switch for instance) is relatively prone to failure. Both sensors are only able to measure precipitation in normal modes above 0 °C. The sensors are able to measure below 0 °C if they have a heating function, but power draw can increase by 100 % to 500 % if required to operate for long times in low temperatures, depending on the model. When deploying the system in the Netherlands, it is important to consider that the low availability of sunlight in the colder winter months can force a choice between no precipitation measurements in winter or an oversized energy harvesting system. This location of deployment is also an indicator of how much impact the distribution of temperature and sunlight per month will have. In the end, it seems an optical solid state sensor should satisfy the requirements more often than the tipping bucket and radar alternatives, and this type can be used to formulate a final conceptual design.

WIND SENSING

The weather parameters relevant for wind sensing are the wind speed and wind direction. The considered options for measuring both are mechanical, sonic solid-state or thermal solid-state. The mechanical wind sensor consist of a weather vane for the direction and a mechanical anemometer for the wind speed. The main advantage of a mechanical set-up is its low power draw. The main disadvantage is that this type of sensor has moving parts, which have to be maintained in order to secure operation. Models do exist however which have a higher reliability with lower maintenance, but these are more expensive. The other options are both solid-state. While the initial costs are slightly higher, solid-state sensors don't typically require maintenance. They do however draw a higher amount of power than their mechanical counter-part, even when operating in 'low-power' mode. A comparative table concerning the three options can be found in table 3.7.

Table 3.7: Comparison of the wind sensors.

	Weight $\|_{\text{Cost}}$	Weight $\|_{\text{Autonomy}}$	Mechanical	Sonic S.S.	Thermal S.S.
Power consumption	30 %	30 %	5	3	2
Cost	50 %	10 %	3	2	2
Maintenance	10 %	30 %	4	5	5
Lifespan	10 %	30 %	3	5	5
		Average Score	3.75	3.75	3.5
		Weighted Score $\|_{\text{Cost}}$	3.8	2.9	2.6
		Weighted Score $\|_{\text{Autonomy}}$	3.9	4.1	3.8

3.1.4. RESULTING CONCEPTUAL DESIGN

In the previous sections different options were compared which result in multiple options for the system design, each of which would comply with the requirements, but differ slightly in prioritisation of trade-offs.

The main trade-off in this conceptual design is autonomy and a more expensive system versus maintenance and a less expensive system. In order to prevent the need for regular maintenance due to mechanical sensors, an all-solid-state overall system design is preferred and will be further fleshed out in the rest of the design phase. Even though a mechanical wind sensor would result in a more affordable weather station, the sonic solid-state wind sensor provides a sensible trade-off between maintenance and power draw.

Since autonomy is favoured above any maintenance, and since the costs of the system are expected to be within the set limits, the design with the least expected maintenance is chosen. The design choices are presented in table 3.8

Table 3.8: Resulting conceptual design choice.

Subsystem	Design choice
Energy Harvesting	PV
Energy Storage	Battery
Data Transmission	LPWAN
Wind Sensor	Sonic Solid-State
Precipitation Sensor	Optical Solid-State

3.2. EMBODIMENT DESIGN

In separating the platform from the sensors in the conceptual design, it is similarly possible to approach it separately in the embodiment design phase. In embodiment design, the structure of the components of the design are finalised. This structure indicates how the final product should function as a whole and how the components interact to perform that function. If the platform can be separated, the function it performs is itself also independent of the components attached to the platform.

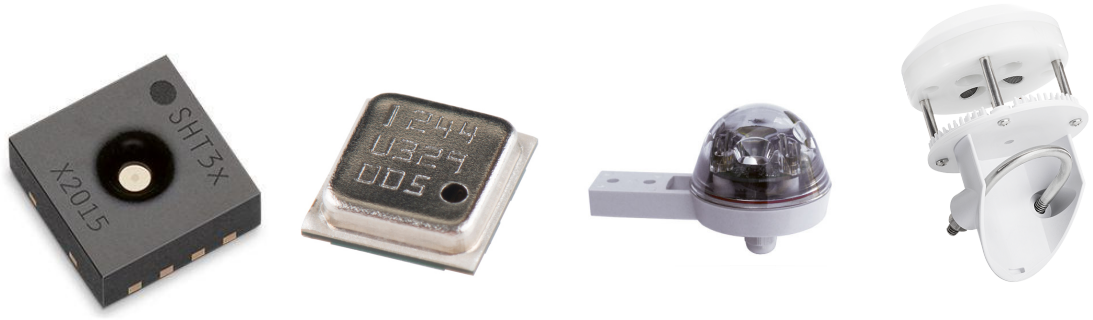
This section focuses on specific components, the layout between the subsystems and where and how much power draw is expected. In doing so, an overview of the entire system as a collection of interacting components is conceived. The sensor platform is divided into two subsections: data processing and energy processing. As a final step this embodiment design of the platform can then be combined with the selected sensors giving a broad indication of power flow in the system, which leads to estimates that can give clarity to the required size of the photovoltaic array and battery. The detailed design stage is used to pick specific implementations of the components that are still undefined at that point.To finalise the design in every detail, using the trade-offs properties found in the design to select a desired optimum that can be considered to be within the final requirements.

3.2.1. FINAL SENSOR CONFIGURATION

In sensor configuration, two of the most important aspects concerning embodiment design is the amount of energy used by each sensor and the amount of data each sensor delivers regarding the requirements of chapter 2. The total data size that results from these measurements are listed in table 3.10. The precipitation and wind sensors have been selected. The temperature sensor, atmospheric pressure sensor and humidity sensor have minimum requirements stated in chapter 2. The final sensor configuration can be found in table 3.9 and figures 3.2.

Table 3.9: Overview of the sensors

Weather Parameter	Brand	Name	Manual
Temperature & Relative Humidity	Sensirion	SHT30	[11]
Atmospheric Pressure	Bosch	BMP180	[12]
Precipitation Volume	Hydreon	RG-11	[13]
Wind Speed & Direction	Meter Group	ATMOS 22	[14]



(a) Sensirion SHT30 (b) Bosch BMP180 (c) Hydreon RG-11 (d) Meter Group ATMOS 22

Figure 3.2: Resulting sensors in embodiment design.

3.2.2. DETERMINING THE TRANSMISSION PAYLOAD AND PROTOCOL

DETERMINING THE TRANSMISSION PAYLOAD

Before determining the type of LPWAN communication system, it is important to know the payload of a transmission concerning the requirements. Requirements [2.1] to [2.16] are of concern. In table 3.10, the minimum requirements of the sensor measurements are given and the resulting minimum of bits needed. The exception is the precipitation sensor, which has no requirements concerning its resolution. This is because the precipitation is measured in volume and generally is measured with a (simulated) tipping bucket. The sensor would then trigger when a certain amount of precipitation is collected. In this calculation, the sensor is assumed to trigger after every 0.2 mm of precipitation. The maximum size in bits then is determined by the maximum amount of expected precipitation in a minute. The maximum precipitation is determined by a phenomenon called a 'cloudburst'. The Royal Netherlands Meteorological Institute (KNMI) defines a cloudburst as a minimum precipitation volume of 25 mm per hour or 10 mm per 5 minutes. Cloudbursts of 38.1 mm in 1 minute have been reported [15]. It is not expected that these amounts of precipitation volumes will be measured in the Netherlands, where the highest volume short burst rain was reported as 25-30mm in 5 minutes [16], but it is nevertheless good to dimension the system in order to make it possible to measure. In order to measure these kinds of volumes, 6 bits are needed per minute. This results in a maximum precipitation volume sensing of 12.6 mm per minute. This maximum is much higher than the average precipitation volume expected, but now the system has the ability to measure these events. This results in a total minimum payload of 56 bits of data collected per minute.

Table 3.10: Amount of bits needed per weather parameter with respect to the requirements given in chapter 2.

Weather Parameter	Range	Resolution	Distinct values	Bits needed
Temperature [°C]	−50 to 70	0.04	3001	12
Atmospheric Pressure [hPa]	900 to 1100	0.1	2001	11
Precipitation Volume [mm/ min]	0 to 12.6	0.2	64	6
Wind Direction [°]	0 to 359	1	360	9
Wind Speed [m/s]	0 to 25	0.1	251	8
Relative Humidity [%RH]	0 to 100	0.1	1001	10
			Total	56

CHOICE OF LOW POWER WAN COMMUNICATION SYSTEM

In section 3.1.2 of the conceptual design, a LPWAN communication system is considered. Multiple LPWAN solutions exist of which the most popular/relevant are SigFox, Long Range WAN (LoRaWAN) and Narrowband-IoT (NB-IoT)[1, 17–20]. The options with its properties are tabulated in table 3.11. In order to choose one over the other, the requirements given in section 2.1 are considered. Each of the measurement range and resolution requirements result in a minimal amount of bits of information that has to be send, as presented in table 3.10, which results in a data rate of 502 bytes per hour. This means that a SigFox communication link is unable to keep up with the stream of data leaving NB-IoT and LoRaWAN as the remaining options. NB-IoT has the best uplink characteristics. However the network is not easily extendable as it operates in the licensed LTE bands, this in contrast to LoRaWAN, which allows for the deployment of additional gateways and therefore the extension of coverage where none is available. LoRaWAN is the preferred choice for the application.

PROPERTIES OF LORAWAN

In order to move forward with the design process, the properties and limits of LoRaWAN need to be explored. LoRaWAN is the protocol for long range, low power data communications developed by the LoRa-Alliance. LoRa is the physical layer developed by Semtech which has datasheets on the subject [21] and documentation on the limits are described in [22].

[1] ISM stands for the Industrial, Scientific and Medical radio bands reserved internationally for applications other than licensed telecommunications.

[2] Long-Term Evolution (LTE) is a cellular standard for wireless communications.

Table 3.11: Comparison of leading LPWANs based on [18].

	Weight	SigFox		LoRaWAN		NB-IoT	
Frequency	30 %	Unlicensed ISM[1]	5	Unlicensed ISM	5	Licensed LTE[2]	1
Private network	50 %	Not allowed	3	Allowed	5	Not allowed	3
Uplink rate	10 %	70 B/h	1	765 – 18360 B/h	4	1600 B per TX	5
		Average Score	3		3.25		2.75
		Weighted Score	3.8		3		1.7

LoRa sends data by modulating the data signal with a chirp spread spectrum (CSS) with a center frequency of 868 MHz in the EU (EU 868). The data rate is highly dependent on the spreading factor of CSS, which for LoRa is between 7 and 12, with 7 being the best case scenario and 12 the worst. This Spreading Factor corresponds to the rate of the frequency sweep, a SF of 7 results in faster sweeps, meaning it takes a shorter amount of time to send the same information, a SF of 12 is a slow sweep, which makes it more resilient to interference and noise, but results in a slower data-rate. The effective data rate is also determined by the unlicensed EU 868 ISM band, on which the EU defined a maximum duty cycle of 1% time on air as to minimise packet loss. For the worst case scenario, i.e. spreading factor 12, the usable payload for a transmission is 51 bytes. LoRa data rates go from 27 kbit/s with spreading factor 7 to 0.3 kbit/s with a spreading factor of 12 [18, 19].

There are a couple of factors determining the parameters of concern when designing. The factors most important to know for the system are the time on air and the maximum payload in order to meet requirements. When the time on air is known, the duty cycle of the LoRa-module is known. The spreading factor causes the sent signal to have more energy, meaning it will travel more distance, but also requires more energy and time for the same information to be sent. This subsystem is treated at its worst condition in order to scale the power subsystems. The time on air is determined by the length of a packet sent. A typical packet is described in [21] and is determined by the number of symbols sent, which is determined by the overhead and the payload. The factors are: the bandwidth (BW) of the signal, which is 125 kHz in the EU, the code rate (CR), here assumed to be 1, the spreading factor (SF), which for the worst case scenario is 12, the header of the packet (H), the option for low data rate optimisation enabled (DE), which enables when $SF \geq 11$ and the payload which in the worst case is 51 bytes. The worst case scenario payload in symbols and preamble in symbols is then calculated by equations 3.1 [21].

$$payloadSymNb = 8 + max\left(ceil\left(\frac{8PL - 4SF + 28 + 16 - 20H}{4(SF - 2DE)}(CR + 4)\right), 0\right) \tag{3.1}$$

where $H = 0$ when the header is enabled and $DE = 1$ when data rate optimisation is enabled. This results in a symbol payload of 63 symbols. The symbol time is calculated by equation 3.2 [21].

$$T_{symb} = \frac{2^{SF}}{BW} \tag{3.2}$$

All packets commonly have a preamble ($npreamble$). For a standard LoRa packet, this is equal to 8 symbols. The time on air can then be calculated by equation 3.3 [21].

$$T_{onair} = (payloadSymNb + 4.25 \cdot npreamble) \cdot T_{symb} \tag{3.3}$$

which results in a worst case time on air of 2.4658 s. This means the minimum duty cycle the system has to adhere to has a period of at least 246.5 s. In this period, a total of 51 bytes can be sent. For simplicity of the design, a period of 300 s is taken. It was calculated in table 3.10 that the minimum payload to meet requirements is 56 bits per minute or 35 bytes per 5 minutes. This means that LoRaWAN can meet the requirements. Of the payload, 16 bytes per 5 minutes are unused which can be used for additional information, which is further elaborated on in subsection 5.2.1. Further details and special cases (some sensors require continuous sensing) will be discussed in section 3.3.

3.2.3. LAYOUT OF THE DATA STRUCTURE

In the embodiment design of the data structure of the system, the layout and the data flows will be explored. From that, the power consumptions of the data subsystems can be approximated, after which a better approximation can be made on the design of the energy subsystems. An overview of the data subsystems is given in figure 3.3. In this figure, the layout of the expected components and connections are given, styled after the conceptual design of figure 3.1. The two subsystems concerning data and the sensor platform are the microcontroller and the LoRa-module, the first of which will be discussed here.

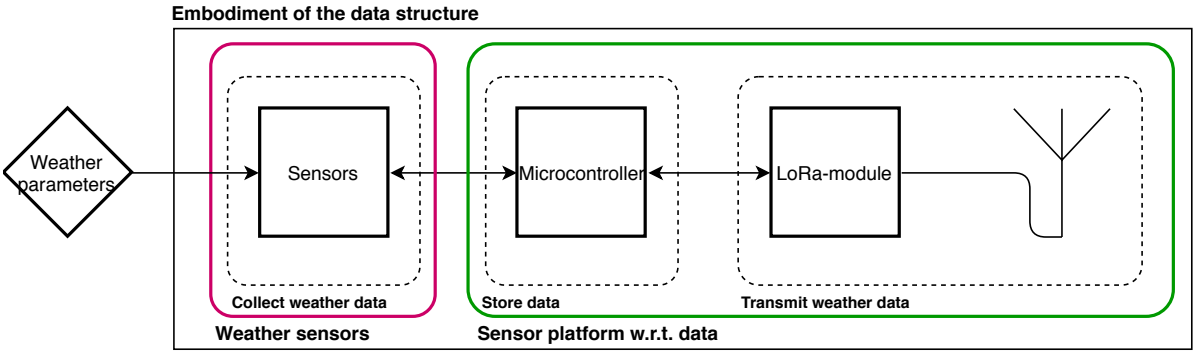


Figure 3.3: An overview of the layout of the subsystems concerning data flow. The exact type of interfaces are not known at this stage. Each component is connected to an energy source, to be discussed in subsection 3.2.4.

MICROCONTROLLER

In the completed system, strict control must be exercised over the sensors and communication systems to ensure few enough losses are incurred in the process of frequent measurements. The decisions of when and how long to measure, and handling incoming and outgoing data streams are made by a microcontroller. The microcontroller is responsible how much total power is being consumed by the load (board, microcontroller, sensors and communication) at any given time. This gives the opportunity to use this control to be able to respond to situations when the supplied power exceeds or falls below the values for nominal operation. The microcontroller and development board can therefore be considered by how they can improve on these two fronts. First, in minimising the total power the system would need over long periods of time, three points of focus mainly affect the result: determining the smallest amount of measurements required to provide enough data to satisfy the system requirements, reducing the energy used by the entire system when not actively measuring or communicating, and running the sensors and LoRa module in a duty-cycle mode. Second, the microcontroller is in control of what changes can be made to the process of measuring and communicating in the event that the average power supplied by the PV module over a long period is significantly larger or smaller than the consumed power. This leads to either an empty or full battery, in which case the microcontroller can reduce or increase the consumed power to better match the supply. Effectively this increases the time it takes for the system to fail if the power supply is insufficient during extended periods with low solar irradiation, as well as increasing the measurement capabilities beyond nominal when power is available and would otherwise be wasted. The exact implementation of the second type of control however should be a direct result of the trade-offs that are presented by the situations of insufficient or abundant supply power, as comparing the (dis)advantages of scaling measurement accuracy, frequency, or communication delay to the availability of power are considerations that must be compared and agreed upon later.

3.2.4. LAYOUT OF THE ENERGY SUBSYSTEMS

Continuing from the conceptual design of figure 3.1, the relevant subsystems that harvest and regulate the energy flows of the system are harvest energy, store energy and distribute energy. Distribution of the available energy is highly dependant on the load and demands and will be discussed in section 3.3. The focus of this piece of design is on the energy harvesting and the energy storage.

From section 3.1, the design choice for the energy harvesting is a solar panel and the choice for energy

storage is a battery. In this section, a choice will be made on the type of solar panel and battery. In order to connect a solar panel to the system, a DC to DC converter needs to be inserted in order to secure a steady voltage for the battery and the rest of the load. The connection between the solar panel and the battery has a maximum power point tracker (MPPT), which uses a feedforward loop in order to improve efficiency of the solar panel. Then, before the energy can be used for the load or stored in the battery, a battery protection circuit is inserted. The main function of the circuit is to prevent energy overflow in the battery, when the battery is fully charged and the solar panel provides energy. The DC/DC-converter, MPPT and battery protection circuit are readily available and together are commonly called a solar charge controller. A diagram describing the energy harvesting and energy storage in more detail with the relevant subsystems can be found in figure 3.4.

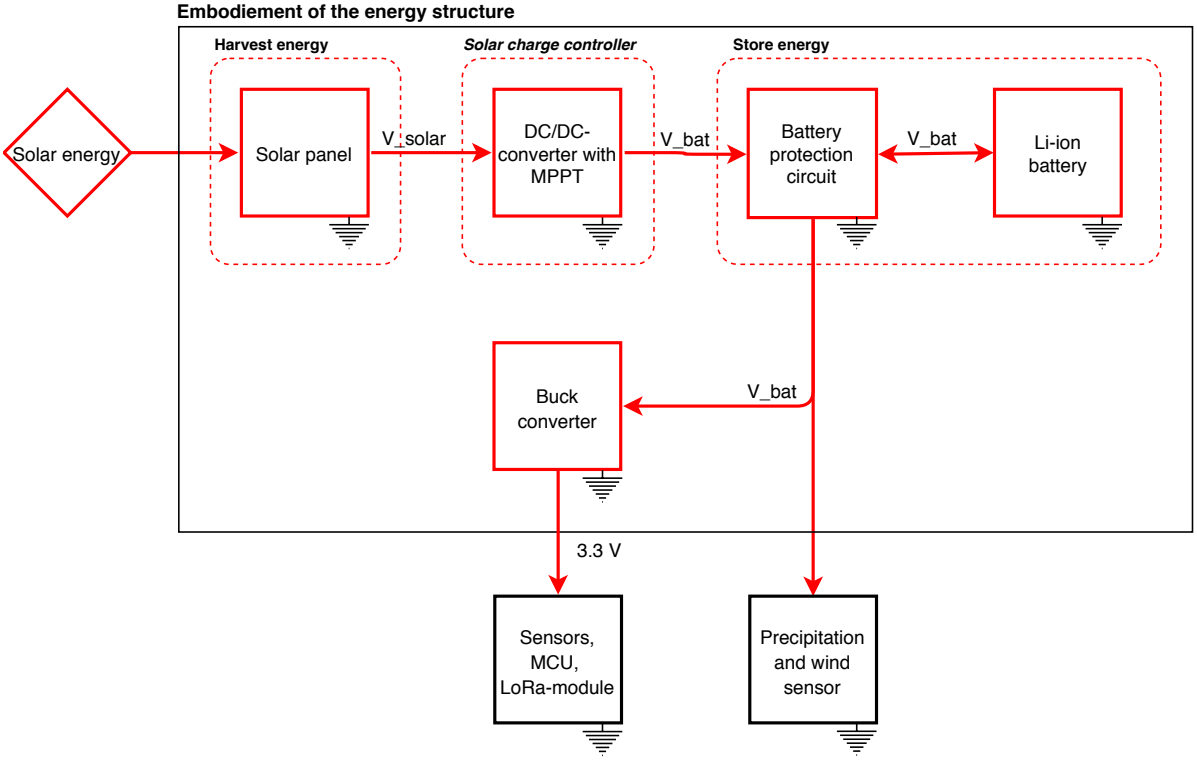


Figure 3.4: A diagram describing the energy harvesting and energy storage of figure 3.1 with its connections in more detail.

PHOTOVOLTAICS AND SOLAR CHARGE CONTROLLER

A few components are under consideration in the energy harvesting part, where a photovoltaic module is used to provide power to the system, as well as storing excess energy in a battery. A first consideration or comparison to make is on the preferred type of PV cell. Most small size commercially available PV modules fall under one of two classes; Monocrystalline, or Polycrystalline, based on the purity of silicon and its processing during production. Primarily it is important to find a solar module for which the rated power is closest to the required minimum supply, which can be determined in the detailed design as a function of the power draw of the connected load, but the difference in production process influences properties besides efficiency and peak power output. If there is the possibility to then choose between similar peak power modules of different types, table 3.12 compares this set of properties that have an impact on the weather station design results apart from the electrical characteristics of the power supply.

The clear support for a polycrystalline panel are in part due to the advantages of monocrystalline panels, such as spatial efficiency and aesthetics, are not as relevant to the requirements of this project as long as the peak power output is large enough. Contrary, the monocrystalline disadvantages of higher financial cost per Watt and the high quality silicon used leading to a more environmentally damaging production process, have

Table 3.12: Comparison of available PV module types.

	Monocrystalline	Polycrystalline
Resilience	5	4
Average cost per Wp	3	2
Environmental impact	3	1
Average score	3.7	2.3

a more direct correlation to the requirements and constraints of the system. To conclude, if comparable PV modules of the different types are suitable for the design, considering the costs and environmental impact of production, preference will be given to a polycrystalline module.

It is important to realise that simply connecting a PV cell to a battery does not constitute a proper setup to act as a reliable supply. A PV cell does not supply a fixed voltage or current, as these parameters also depend on the temperature and effective irradiance of the cell. To charge a battery or supply a load, the output voltage of the module should be fixed. Allowing the battery in question to act as a buffer for when PV power at this voltage is insufficient. This means a DC/DC power converter should be used to convert the varying output of the PV cell to one with fixed output voltage and varying power. For an application where the supply power should be carefully balanced to the load, any losses in this converter should be considered and minimised. The efficiency of the converter is a factor that increases the required daily average harvested power for the entire system. Both Koutrolis et al. [23], and Reverter and Gasulla [24] discuss inefficiencies in these converters and designing to minimise the losses for low-power PV systems. An important efficiency improvement in especially [23] is the concept of maximum power point tracking. The idea that with a variable source voltage and power depending on environmental conditions, an optimum switching duty-cycle can be obtained by using a feed-forward network or iterative approach to find the the operating point where $\frac{\delta P}{\delta V} = 0$. A DC/DC converter operating on this point of the curve then induces the smallest possible power losses for the conversion of a variable V_{solar}, P_{solar} to a fixed V_{bat}. Conclusively, what is considered as the PV module should be a PV cell connected to a DC/DC converter with maximum power point tracking and a high overall efficiency. Cursory research suggests efficiency upwards of 90% is obtainable for low-power applications, and commercially available MPPT converter circuits show similar efficiency in their respective specifications.

BATTERY

For the battery, the first concern is picking the type of battery. Common battery types include lithium-based batteries. For PV-powered systems, lead-acid batteries tend to be a dominant technology in small-scale domestic PV-systems. Main concerns when picking the type of battery are the costs per energy, lifetime or life cycles, efficiency and operating range, which all depend on each other. Comparisons are made in [25]. It suggests li-ion batteries are the best option for stationary applications, outperforming lead-acid batteries in life cycles, costs, charging capabilities and efficiency.

Charging below 0 °C is however not recommended, since irreversible damage can occur to the battery. To handle with this, the design should have all its components isolated to shield from the freezing temperatures. Energy dissipation of the components can be used for marginal heating. Discharging can occur even at temperatures as low as −20 °C. However, safety measures must be included, which will be discussed in chapter 5.

3.2.5. ANALYSIS ON THE POWER DRAW

In order for the system to operate continuously, it is essential that the power consumption over long periods of time does not exceed power generation over the same period. The difference in generation and consumption is buffered by the battery, and when the battery charge is 0 while more power is consumed than the PV system can generate, the system fails. If an operating period of a full day is considered, the expressions of equation

3.4 would need to hold.

$$P_{gen}(t) - P_{con}(t) = \frac{\mathrm{d}E_{bat}(t)}{\mathrm{d}t} \tag{3.4}$$

$$\int_0^T P_{gen}(t) - P_{con}(t)\mathrm{d}t = E_{bat}(T) \tag{3.5}$$

$$\int_{T-24h}^T P_{gen}(t) - P_{con}(t)\mathrm{d}t = E_{bat}(T) \geq 0 \tag{3.6}$$

where P_{gen} is the power generated by the energy harvesting, P_{con} is the power consumed by the system and E_{bat} is the energy stored in the battery. The critical analysis that must therefore be made is either to scale the power consumption in order to be smaller than the generation, or the generation to be larger than the consumption. In addition, if the time dependence of the Power flow can be made predictable by finding average power consumption over the 24h period instead of immediate power $P(t)$, the integral can be reduced to

$$24\,\mathrm{h} \cdot (P_{avg,gen} - P_{avg,con}) = E_{bat}(T) \geq 0 \tag{3.7}$$

where $P_{avg,gen}$ and $P_{avg,con}$ are the average values of the generated and consumed power, respectively. This is under the condition that the size of the battery is large enough to contain $(24h * P_{avg,gen})$ Wh of energy.

In the current stage of the design, because of the considerations in the conceptual design, a selection of sensors has already been made.With respect to available time and choice, scaling the PV module to equal the absolute maximum power consumption for a 24 h period is the preferred option. The first stage in balancing the power generation and consumption is to find the absolute maximum average power consumption expected from the selected sensors.

SENSORS

Before moving on to the detailed design phase and the beginning of the prototype, a rough estimation is made on how much power is consumed in the 1 minute measurement duty cycle and the 5 minute communications cycle. These estimations are the first step to finding an accurate representation of how the expected power is distributed and how much maximum power consumption is expected. Beginning with the sensors, one of six stands out in that its mode of operation is not a duty-cycled like the rest. The RG-11 optical rain gauge pulses an output signal each time a set volume of precipitation has been detected and therefore is only active during rain. The other sensors take a minimum of one measurement per minute, and are idle most of the time. The anemometer, because of the properties of wind, measures for a minimum of 5 seconds per minute and provides an average. Other sensors are capable of this for improving accuracy, but this is more a marginal trade-off of accuracy and power and not required for this estimate. Estimates are made by using the time it takes for 1 measurement and the active and passive currents for a measure/idle cycle, and an overview of the power draws are given in table 3.13. It is determined from this that the power usage of the Decagon ATMOS 22 anemometer outweighs the other duty-cycle sensors by a factor 10 to 100. Resulting from this analysis, for now it is assumed these sensors will consume on average 1.01 mW.

Table 3.13: Preliminary average power draw estimations of the four sensors which are bound by a duty cycle of 1%.

	I_s[µA]			P_{avg}[µW]
Sensor	Active	Idle	Average at 1/60 Hz	at rated V_{DD}
Sensyrion SHT30	800	0.2	0.23	0.8
Bosch BMP180	40	2	2.6	9
Meter Group ATMOS 22	500	150	200	1000

The rain sensor is different in that it has both a high rated voltage and power compared to the other sensors, and that it is only in active mode during rainfall events. The selected Hydreon RG-11 has three modes of operation when simulating a tipping bucket, one when no rain has detected and the sensor is sleeping, one where rain was recently detected and the device is sensing, and one where the tipping bucket triggers and the

Table 3.14: Hydreon RG-11 power draws

Weather Conditions	Mode	Current draw [mA]	Power draw [mW]
Dry period	Sleeping	1.5	18
Raining	Active	15	180
Raining	Output pulse (50 ms)	50	600

output is pulsed for 50 ms. When in active mode the device waits 20 minutes before switching to sleep if no rain was detected. The power draw of the Hydreon RG-11 for its different modes can be found in table 3.14. It is important to note that although the peak power draw is up to 600 mW, this only lasts for 50 ms each time 0.2 mm of rainfall is detected, which means that in the case of the highest precipitation intensity ever measured in the Netherlands, 79 mm in one hour, the peak power would still only add to the consumed energy, a total of

$$\frac{79\,\text{mm}}{0.2\,\text{mm}} \cdot 0.05\,\text{s} \cdot (600\,\text{mW} - 180\,\text{mW}) = 8.295\,\text{J} \tag{3.8}$$

which means it would only increase the total power consumption by 1.28 % during an event that has a chance of occurring once every 1000 years.

Clearly both the sleeping and active average consumption are significantly higher than any of the other sensors, even when no rain is being detected. As the RG-11 is rated between 10 V to 15 V, it will be connected directly to the battery at V_{bat}. As far as the sensors go, table 3.15 approaches the absolute maximum of what the sensors will require.

Table 3.15: Power estimation of the total sensor array during differing weather conditions.

Weather Conditions	Power draw [mW]
Dry period	19
Precipitation	181

This gives a clear idea of how to figure out how much power the entire sensor array will require. Because of the precipitation sensor outweighing the load of other sensors by a factor of 10 up to 100 based on rainfall, the approximation can be made with the following parameter: How frequent and long are precipitation events on average. The probability distributions of the occurrence of different types and volumes of rainfall can therefore be used in combination with these thwo power draws to determine the required supply power (and from that the size of the PV module and battery) in a realistic simulation.

LoRa

The power estimation of the LoRa Module is a lot simpler in practice than for the sensors. The module currently used in the proof-of-concept model has three levels of consumed power, for Sending, Idle, and Sleep mode, and will run with a fixed duty cycle of 1 % sending, 99 % idle/sleep as a result of the LoRa communication airtime rules. The power for the three modes can be found in table 3.16.

Table 3.16: LoRa power consumption

Status	Power draw [mW]
Sending	132
Idle	9.24
Sleep	0.0053

Deriving from this and the fixed duty cycle, the estimate for the LoRa-module is as follows in equation 3.9. It is expected the start up and shut down times to be less than an additional 1 % of the duty cycle (3 s total). This time-average power draw is comparable to the sum of the non-precipitation set of sensors, and although

not insignificant in a dry period does not contribute significantly to the power draw in the event of current or recent rainfall.

$$P_{LoRa} = 0.01 \cdot 132\,\text{mW} + 0.01 \cdot 9.24\,\text{mW} + 0.98 \cdot 0.0053\,\text{mW} = 1.42\,\text{mW} \tag{3.9}$$

MICROCONTROLLER UNIT

Because of the acquisition and use of the SODAQ ONE development board with LoRa-module for the proof-of-concept development in the early stages of the design, the same board is considered here as the expected microcontroller unit (MCU) guiding the sleep/active measurement cycle and data flows. By using a USB power monitor, two power states could be measured quickly, giving the estimates in table 3.17.

Table 3.17: Preliminary power draw measurement on SODAQ ONE.

Status	USB Power draw [mW]	Expected Power draw [mW]
Active	90	90
Idle	40	0.66

The Active measurement being obtained while having the processor performing a simple calculation on a loop, and the Idle by suspending processor operation until interrupt using the `__WFI()` command. An additional resource for more information was found on the sodaq developer forums, where multiple people had already shared resources in an attempt to reliably obtain an as low as possible power draw by entering the deep sleep mode and unpowering all unnecessary peripherals. A conclusion that can be drawn is that measuring the Idle (or Deep Sleep) power consumption can not be done properly over a USB bus, as several systems on the MCU will be engaged and consuming power as a part of not turning off the USB. Therefore, only the high power consumption measured can be used as an indication of reality, until current measurements can be done with the MCU running all necessary code on a separate power supply instead of USB. Suggestions indicate that the low power draw of the MCU in deep sleep can be expected to be anywhere between 50 mA to 200 mA. It is important that in developing and reporting the prototype, a clear view can be given of the real effective power consumption of the MCU in deep sleep mode, and how much the average consumption over one measurement cycle would be. For now, until conclusive measurements can be made, using the theoretical upper limit of 200 µA at 3.3 V supplied yields an expected power consumption of 660 µW in deep sleep.

3.3. DETAILED DESIGN

The detailed design continues were the embodiment design left off. The intermediate system overview is given in figure 3.5. The expected maximum power draw of the system is given in table 3.18. These estimations are necessary in order to get insight on the dimensioning of the energy supply and storage subsystems, since a system designed for these worst-case conditions will function nominally in real-world conditions.

Table 3.18: Preliminary power draw estimations of the different subsystems.

Subsystem	Maximum estimated power draw [mW]
Sensors (during rain)	181
MCU	90
LoRa-module	1.5
Total estimated maximum power draw:	272.5 mW
Estimated 24h power draw:	6.6 Wh/day

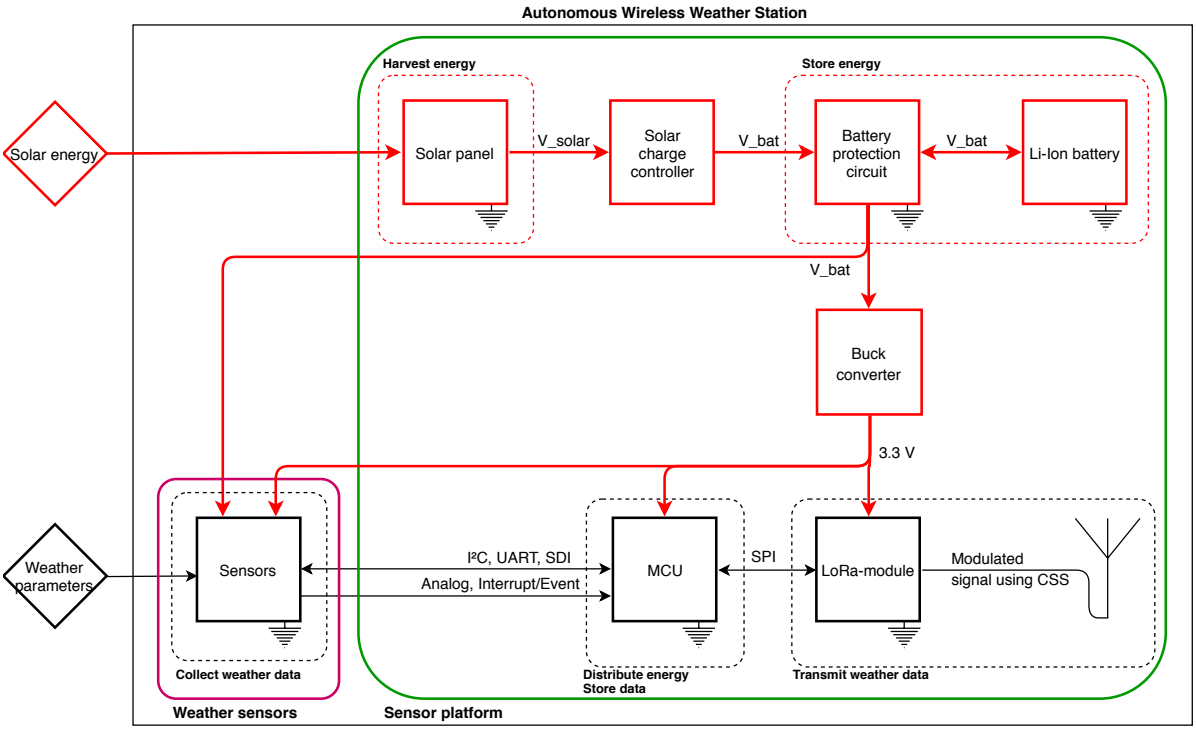


Figure 3.5: A diagram describing the system of the Autonomous Wireless Weather Station. The diagram is modelled after figure 3.1 in order to show the design process. The red lines represent power flows with their most important parameters, the black arrows represent data flows with their type of possible connections.

The detailed design will feature the rest of the design process. The software concerning the energy management and the data flows will be elaborated on, as well as the expected power draws of the rain sensor and the communication system. After that, the exact design of the dimensions of the power supply and storage are calculated, taking in consideration all the losses from the converters necessary in the supply. Finally, a complete design is presented.

3.3.1. SOFTWARE ON THE DATA AND POWER MANAGEMENT

The detailed design is the first design phase in which the software will be discussed. The software components of the design are required for management of when the system is active or in deep sleep and management on the data flows of the system. The software is designed with the requirements of chapter 2 and the limitations of LoRa of chapter 3.2.2 in mind. Requirement [2.1] states that for all sensor data except precipitation, the system should be able to measure every minute. Chapter 3.2.2 states that in the worst case scenario for transmission, the system can send out data at a rate of 1 transmission per 5 minutes in which all the measurement data can be fitted. Furthermore, the precipitation sensor works by sending an 50 ms pulse when 0.2 mm of precipitation has fallen. When that happens, a counter should increment as to collect the precipitation data. This means there are 3 reasons the system activates. Considering minimising the energy consumption of the platform is a part of fulfilling the requirements, designing the software in order to run with minimal power in sleep mode and minimal time spent in active mode is a priority. An overview of the functionality of the software is given in figure 3.6. The figure displays the three functions the MCU should perform, and the modes in which these can performed. Every minute a measurement is done during active mode, followed by entering sleep mode as soon as measurements finish. Every 5 measurements the result is sent using the LoRa-module during active mode. Finally, in either active or sleep mode, if a pulse arrives from the rain gauge, the MCU increments a counter. If this happens during active mode, it should not intervene with the other measurements, and if it happens during sleep mode the system should not wake up unnecessarily. During the next active mode, the counter should be read, saved, and reset as part of the measurements.

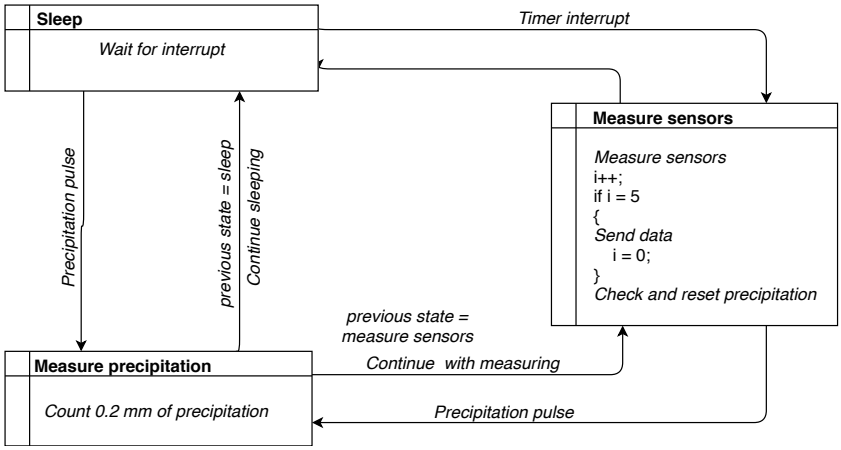


Figure 3.6: An overview of the software systems in play.

The first function of the software structured here is to command and receive measurements from the 3 sensors which are run on a duty cycle, which is set up as follows. A measurement cycle of 60 s is set up on an internal timer named the Real Time Counter (RTC) which can trigger an interrupt after the minute is over. Following this, using I^2C a single shot measurement command is issued to the SHT30 and BMP180 sensors which will start taking temperature, humidity, pressure measurements. After the sensors start a measurement, there is a delay of a few milliseconds until the data is ready. The ATMOS-22 wind sensor can return an average of its most recent measurements which are taken each 10 s, which is very preferable to a single measurement, as the wind speed and direction are much more varied with time and a single data point provides significantly less insight than an average of 6 measurements over a minute. The sensor communicates using a different standard, SDI-12, and consists of sending a request, and after 15 ms the sensor will return all available data back. Because the communication with the ATMOS must be active during this period, and the 15 ms reply delay exceeds the measurement delay of the SHT30 and BMP180 sensor the data acquisition part of the MCU cycle can be structured sequentially as follows in figure 3.7.

The LoRa-module is also activated during the measurement cycle, and sends one packet consisting of 5 data sets every 5 minutes, since the maximum expected time between transmissions is 4 minutes and 7 seconds. As the packet consists of the last 5 measurements, and the LoRa-module uses the same active cycle as the data acquisition part, these two should be combined to run concurrently where possible to decrease time

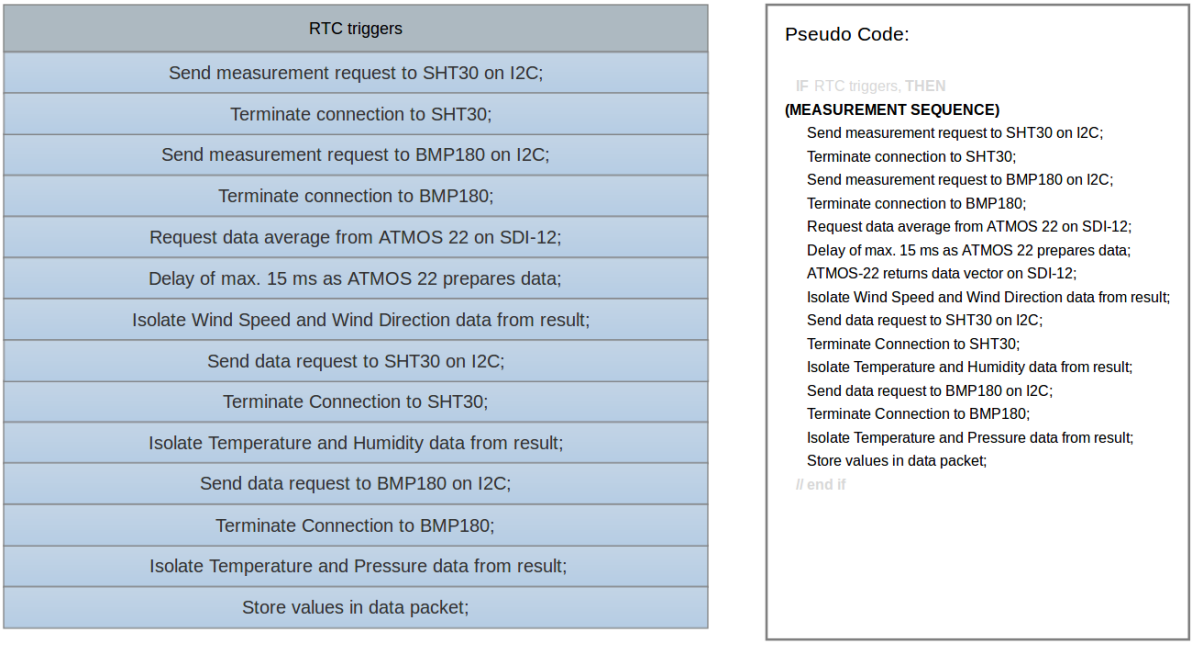


Figure 3.7: Pseudocode on the measurement sequence.

spent in the active cycle. The LoRa-module is luckily able to operate independent of the MCU once the data packet is presented. It is also important to take into account that the LoRa-module takes time to start up and shut down form deep sleep mode, where it should be powered down to minimise losses during the sleep cycle. The sequence for running the LoRa-module (as well as measuring) then can be expected to be structured as such in figure 3.8.

The last function to perform is a measurement of the cumulative volume of rainfall in the same 60 s cycle of the other sensors. The RG-11 optical sensor in the system provides data in the same way a typical 0.2 mm rain gauge would. As it senses drops on its surface and the immediate area, it triggers a 50 ms long pulse that occurs for every 0.2 mm of measured rain. Because of this principle, the rain gauge is not cycled like the other sensors but must be checked constantly for output in order to catch all pulses in the event of rain. If this were to be done by using the interrupt controller it would unfortunately be able to disrupt the processor during communication with other sensors or interrupt an outgoing data packet. Additionally, it is possible that interrupts would need to be blocked during certain sensor communications, which could then result in rain triggers going unnoticed . A more useful solution is to run a general counter as a background process to count rising or falling edges on an external pin, and use the register to keep track of the cumulative amount of rainfall at a given time. To count a rising or falling edge, a clock must be supplied to the counter. Additional clocks running increase power consumption, but the 1 kHz clock of the RTC module can be used for the counter simultaneously. Finally, by running the counter to keep track of incoming pulses but not trigger an interrupt, the MCU only needs to read from and set to 0 the counter value registers at the end of a the measurement cycle and the software should now be able to perform all of its main functions.

3.3.2. EXPECTATION OF THE POWER DRAW

Continuing from the embodiment design phase, it is possible to conclude that the immediate power draw of the system at any given time is determined mostly by the rate and volume of precipitation over the half hour preceding it. In order to get a complete view of the power draw of the system it is also important to get a clear view of the average power draw over a period of T as a result of this immediate power draw. In the design of a stand-alone photovoltaic powered load, a period which is very important for consideration is $T = 24\,\text{h}$. To calculate this, first the predictable sources of power consumption are determined in detail, these are the sensors which run on a set measurement cycle and the MCU. Second, a closer look is taken at the probability distribution of the length of precipitation events by frequency of occurrence, to find an expected average load of the precipitation sensor normalised for $T = 24\,\text{h}$. By combining these two results, a better representation

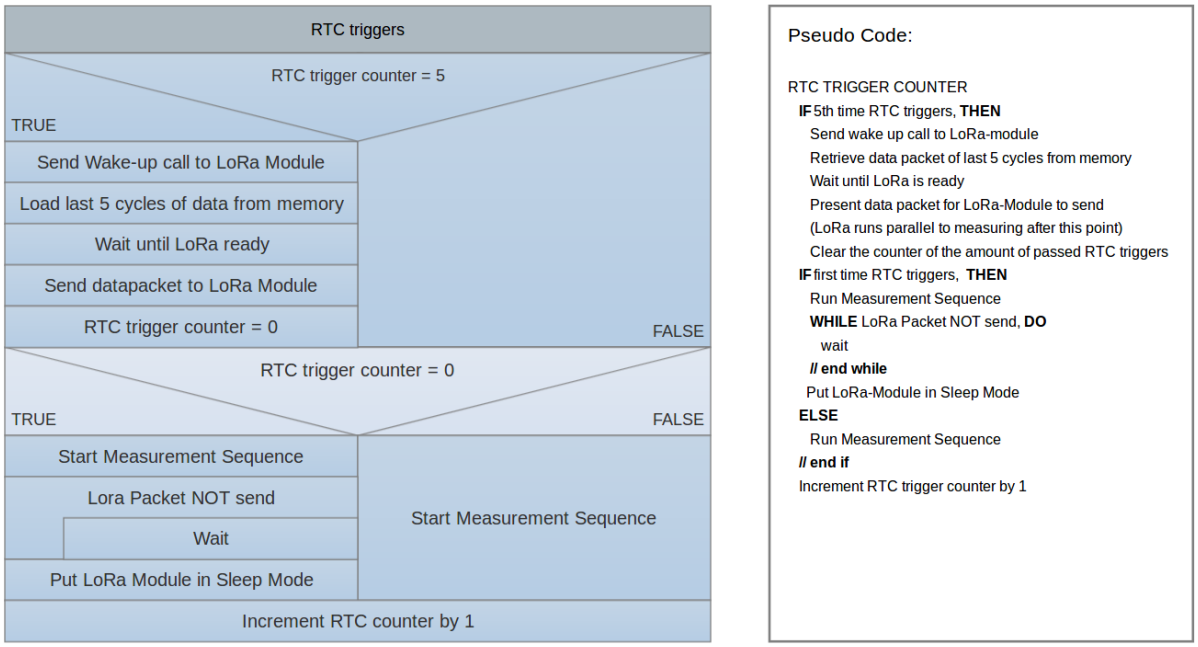


Figure 3.8: Pseudocode on the behaviour of the LoRa-module while sending data.

of the expected average daily system power can be achieved. In estimating the predictable sources, most loads already have disclosed the active and passive power draws in the respective data sheets, including all 3 sensors and the LoRa-module. The MCU's load is dependent on the size of the duty cycle and the power consumption in active and deep sleep mode and is as a result dependent on the software it runs. In order to determine it, the MCU was connected to a power supply and run firstly in a simulated active mode, secondly in a deep sleep mode with minimal processes running, and finally in a duty cycle between both to account for losses in entering and exiting the active mode. To determine an estimate of the duty cycle the length of the SDI-12 communication is required, which can be determined from its integrators guide[26]. The data rate of the communication is 1200 baud, and a reply from the sensor has a length of 8 data sets of 4 characters and 4 extra separation characters. A reply comes at most 15 ms after requesting a measurement. This results in a rough estimate of 30 ms for a wind measurement to complete. The I^2C request and receive communications that occur before and after the ATMOS request are run at 100 kHz and contain at most 40 bytes of data for an additional 3 ms. Even when start up and shut down timings for the sensors are observed, it is likely that the active part of the duty cycle will take 50 ms on average, meaning the MCU would run on a duty cycle of 0.083 % if measurements are taken every minute. This results in the estimation of table 3.19 for the MCU to be combined with the rest of the predictable loads.

Table 3.19: Expected power supply and draw of the different subsystems.

Subsystem	Active power [mW]	Sleep power [mW]	Average Power [mW]
Sensors excl. Rain	2.5	0.970	1
MCU	80	0.660	0.73
LoRa	132	0.053	1.5
		Cumulative predicted load:	3.23 mW
		Total load for T_{24h}:	0.077 Wh/day
		T_{24h} Load incl. Rain sensor:	4.4 Wh/day

This highlights the stark difference between the predictable loads and the rain sensor, as the load of the rain sensor in passive mode is more than 5 times as large as all other loads combined. When it is raining, the rain sensor is up to 60 times more power intensive than the rest of the system combined. Finally, this amount falls short of the initially expected 6.6 Wh/day because the new calculation takes into account the duty cycle of

the MCU with an active and sleeping state, whereas the 6.6Wh prediction was made with a constant 90mW power draw from the MCU.

For this reason, an estimate on average power consumption must also contain an assessment to the average occurrence of rainfall. Fortunately, the Dutch national meteorological institution (KNMI) has been actively observing the weather in the Netherlands for decades and has a vast amount of data on the volume, intensity and frequency of rainfall in their database. Delving through all of this information would of course be far to time consuming, but the analysis of this data has already been done by national and local government agencies, yielding the following estimations. As can be seen in figure 3.9, days with rainfall exceeding 1 mm occur between 130 and 145 times a year, which results in a 35 % to 40 % chance of having a day with rain. This does not account for most smaller rain events where more than 1mm can fall within an hour. If the yearly average total amount of rainfall is taken and divided by the yearly average intensity of rainfall per hour, the estimate is around 7 % to 10 %, so it rains 10 % of the time. As weather data is available logged by hour[1] , including the condition 'did it rain during this hour', the most accurate approach is to analyse this data by simply expressing the percent of hours with rain over the course of decades. This yields the expected chance of any hour having rainfall, therefore triggering the precipitation sensor and multiplying the system power consumption by 10. It takes 20 minutes for the rain sensor to return to sleep mode after detecting rain, so rainfall events shorter than 20 minutes will cause this estimate to be somewhat higher than the real value. The expected average power draw of the precipitation sensor is the average value of a Bernoulli event, which is to say:

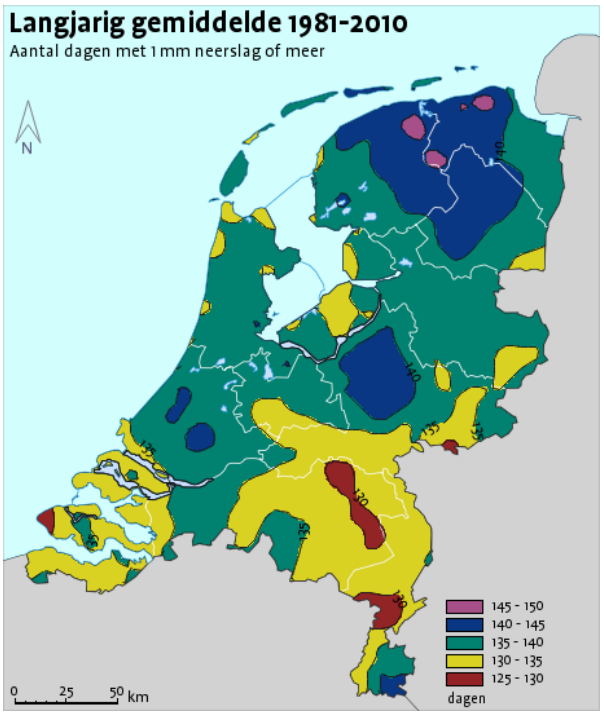


Figure 3.9: A map plotting the yearly average amount of days with rain events

$$\begin{aligned} P_{RG,avg} &= \Pr(rain) \cdot P_{RG,active} + \Pr(dry) \cdot P_{RG,sleep} \\ &= \Pr(rain) \cdot P_{RG,active} + (1 - \Pr(rain)) \cdot P_{RG,sleep} \end{aligned} \tag{3.10}$$

Where $\Pr(X)$ is the chance of event X occurring and $P_{RG,Y}$ is the power consumed by RG during state Y. The hour by hour analysis of the rain occurrence at De Bilt for the period Jan 1st, 2000 to Dec 31st, 2017, is 33417 hours with rain over a total of 6574 days, meaning $\Pr(rain) = 21\%$. This is more than considering the total the percentage of time it rains, but with the delayed sleep mode of the sensor, this estimate should give an accurate absolute upper limit of how much the system is expected to be active versus asleep. The final expected average daily power consumption over long periods of time is thus:

$$\begin{aligned} E_{system,avg} &= (P_{cycle} + P_{RG,avg}) \cdot 24\,\text{h} \\ &= E_{cycle} + \Pr(rain) \cdot E_{RG,active} + (1 - \Pr(rain)) \cdot E_{RG,sleep} \\ &= 0.077\,\text{Wh} + E_{RG,avg} = 0.077\,\text{Wh} + 0.21 \cdot 4.32\,\text{Wh} + (1 - 0.21) \cdot 0.43\,\text{Wh} \\ &= 0.077\,\text{Wh} + 1.248\,\text{Wh} = 1.325\,\text{Wh} \\ \Rightarrow P_{system,avg} &= 1.325\,\text{Wh/day} \end{aligned} \tag{3.11}$$

[1] https://projects.knmi.nl/klimatologie/uurgegevens/selectie.cgi

3.3.3. DIMENSIONING OF THE ENERGY SUBSYSTEMS

From the expected daily and instantaneous power draw the process of dimensioning the various energy system components can begin. In order to realise continuous operation, the photovoltaic subsystem should be able to charge the battery with a days worth of energy on a day with the least expected effective solar hours and with continuous rainfall.

In order to minimise losses, a design with as few converters and regulators as possible is preferred. From the components discussed in chapter 3 it follows that a supply of 3.3 V is needed for the MCU, humidity, pressure and temperature sensors, and 10 V to 15 V for the wind and precipitation sensors. An overview is given in figure 3.10.

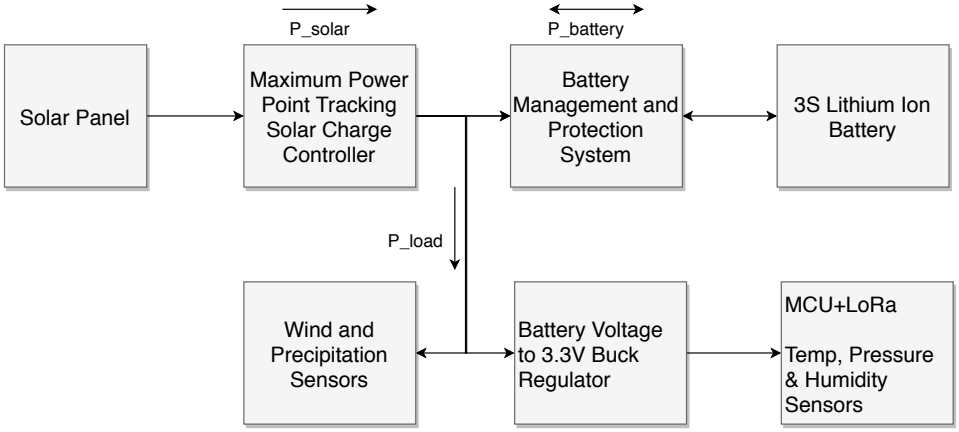


Figure 3.10: An overview of the power flows.

BATTERY

Lithium Ion battery cells have a nominal voltage of 3.7 V, which ranges between 3.4 V to 4.2 V depending on the state of charge. This means that a 3-series (3S) combination results in a battery voltage that lies within the range needed for nominal operation of the system.

With an expected daily energy consumption of 6.6 Wh (table 3.18), a battery of that capacity is needed for each day of storage. In order to maximise the lifetime of the battery, a depth of discharge (DoD) below 10 % and a state of charge (SoC) above 90 % should be avoided, resulting in a battery the size of:

$$\begin{aligned} E_{bat} &= SoCFactor \cdot E_{daily} \cdot T_{days} \\ E_{bat} &= \frac{100\,\%}{90\,\% - 10\,\%} \cdot 6.6\,\text{Wh/day} \cdot T_{days} \\ &= 1.25 \cdot 6.6\,\text{Wh/day} \cdot T_{days} \\ &= 8.25\,\text{Wh/day} \cdot T_{days} \end{aligned} \tag{3.12}$$

Because unprotected Li-Ion cells can behave catastrophically [27], protection is needed to prevent overcharging, depletion, and other situations that result in unusable batteries or even runaway thermal events. This can easily be achieved by the numerous protection circuits available on the market and for series-cascaded cells these circuits also take care of balancing the relative voltages in the battery pack.

SOLAR PANEL AND CHARGER

In order to be able to provide the (maximally) required daily energy supply of 6.6 Wh, the solar panel (and solar charge circuit) is dimensioned to be able to pull this of on the worst day that can be expected solar wise. The Dutch Royal Meteorological Institute (KNMI) defines this as a day in December with only 30 minutes of effective solar hours. To provide an upper limit for the peak power the solar panel must be able to provide, this hypothetical day with 30 minutes of 1000 $W m^{-2}$ of solar irradiance and 23.5 hours of total darkness is used. In order to generate the needed amount of watt hours on such a day, a solar panel with a watt-peak of $W_p = \frac{6.6\,Wh}{0.5\,h} = 13.2\,W$ is needed.

In order to extract as much energy as possible from the solar panel, a Maximum Power Point Tracker is required. An IC that combines both an MPPT and a Charge Controller is the LT3652 by Analog Devices. With a 13.2 W peak solar panel, the theoretical maximum charge current (when the conversion would be 100 % efficient, and the battery is almost depleted at $V_{bat} \approx 10\,V$) is $13.2\,W/10\,V = 1.32\,A$, which both the LT3652 and any Li-Ion battery larger than 1320 mA h can handle. In figure 3.10, the LT3652 and its surrounding circuitry are shown as the 'Maximum Power Point Tracking Solar Charge Controller' block.

3.3.4. FINAL DESIGN SPECIFICATIONS

Specifications of the final design with respect to measurement parameters are given in table 3.20.

Table 3.20: Specification of the weather sensors on the final design.

Parameter	Min	Max	Typ. Resolution	Typ. Accuracy	Unit	Ref
Temperature	−40	125	0.01	±0.2	°C	[11]
Humidity	0	100	0.01	±2	%RH	[11]
Air pressure	300	1100	0.01	±1	hPa	[12]
Precipitation	0	12.6	0.2	N.A.	mm/min	[13]
Wind speed	0	30	0.01	max(±0.03, ±3 %)	m/s	[14]
Wind direction	0	359	1	±5	°	[14]
Operating Temperature Range	**Min**	**Max**				
With Rain detection	0 °C	45 °C				
Without Rain detection	−20 °C	45 °C				

4

PROTOTYPE

The system build up in the design phase needs to make it to the real world. In this chapter, actual hardware, corresponding with the detailed design, is used to construct a prototype. While an eventual product, ready for mass production, should be fully optimized for cost and as integrated as possible, the prototype serves as a testing and validation platform. It is build with time, cost and debugging in mind and therefore makes use of developer and consumer grade hardware.

CHAPTER OVERVIEW

This chapter covers the prototyping of the system, as well as an analysis on that prototype and an early prototype version. Section 4.1 consists of an early prototype in order to test the communications. Section 4.2 describes the implementation of the system based on the detailed design of chapter 3. Section 4.3 describes the validation of the prototype.

4.1. PROOF OF CONCEPT

In the embodiment design phase, different options for various system components were compared which converged to a single global system design. Before starting on the detailed design in which the subsystems and later the whole system are dimensioned, a LoRa development board is used to do some real-life testing on different aspects of the preliminary system design to try to prevent setbacks when implementing the detailed design into a prototype.

The proof of concept consists of a development board made by SODAQ, based on an ATSAMD21G18, 32-Bit ARM Cortex M0+ MCU and features a 1Wp solar panel, 1200mAh single cell battery, solar charge controller, LoRa-module, and different sensors including GPS, Accelerometer, Board Temperature and Battery Voltage.

4.1.1. THE VIABILITY OF A LORAWAN COMMUNICATION STACK

In order to test the viability of LoRaWAN, the PoC was connected to "The Things Network", which is a publicly available, worldwide, LoRaWAN to which users can add their own gateways to extend coverage. Testing the connectivity in Delft, Gilze and Jabbeke (Belgium) showed that relying on existing TTN Gateways is not always possible, meaning that in order to guarantee a connection, the weatherstation would need to come with a gateway as well. However, commercial LoRaWAN networks are being built all around the world and in The Netherlands, KPN provides a LoRaWAN network with nation-wide coverage. Connecting the PoC to KPN's network worked flawlessly and to that extend, the choice for LoRa communication is validated for use in the detailed design.

4.2. IMPLEMENTATION

In this section, the prototype is built using the final detailed design of 3. Any differences in the design and prototype are explained here. The list of prototype parts is given in table 4.1.

Table 4.1: Prototype part list

Type	Brand	Model	Cost
Solar Panel	Enjoy Solar	Eco Line ES10P36	€ 21.95
Battery	Panasonic	3S combination of NCR18650B 3350 mAh	€ 15.00
BMS	N.A.	HX-3S-FL25A-A	€ 6.90
Wind Sensor	Meter Group	ATMOS 22	€ 540.00[1]
Precipitation Sensor	Hydreon	RG-11	€ 50.77
Antenna	N.A.	Center-Fed half-wave dipole 868 MHz	€ 7.40
Solar Charge Controller	SparkFun	Sunny Buddy (LT3652)	€ 27.95
DC Buck Regulator	N.A.	LM2596	€ 4.50
Humidity (& Temp) Sensor	WEMOS	SHT30 Shield	€ 4.00
Pressure (& Temp) Sensor	Bosch	GY-68 BMP180	€ 4.50
Level Converter	SparkFun	Bi-Directional Logic Level Converter	€ 2.75
MCU & LoRa Module	SODAQ	ONE (Cortex M0+ and Microchip RN2483)	€ 95.00
		Prototype Cost with Wind sensor:	€ 780.72[1]
		Prototype Cost without Wind sensor:	€240.72

4.2.1. MCU & LORA MODULE

The prototype uses the SODAQ One board which includes a SAMD21 ARM Cortex M0+ and the RN2483 LoRa module, which was obtained to support the use of LoRa through the Proof of Concept developed after the conceptual design. This board has also been included as the MCU and LoRa combination in the final design, but this was done mostly out of convenience more than the result of a comparative search for an optimal board for the proposed system. Simply put, the design requires an MCU with the capacity to run continuously with very low power power consumption and the capacity to have I2C communication, SPI and SDI-12 serial communication and use external pins as an interrupt, which means a very large selection of suitable replacements can be used to make the design not rely on this board, but another preferred MCU instead. Similarly, a LoRa module can be selected separately to further optimise for cost or performance, but the convenience of SODAQ having all necessities in an affordable package, complete with LoRa module and online support for using the board as a low-power measurement node, means it was an obvious pick to use in these circumstances. The consistency between design and prototype also supports using the prototype to validate as much of the detailed design as possible. The SODAQ ONE is shown in figure 4.1.

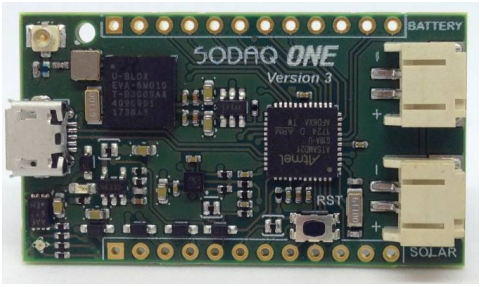


Figure 4.1: SODAQ ONE LoRa Development board

[1] Discussion on the wind sensor and its cost in chapter 5.

4.2.2. SOLAR PANEL

Other than a peak power output of 13.2 W, section 3.3.3 provides no further requirements for the photovoltaic panel. The physical size relates to the peak power output with the efficiency of the panel. Although a high-efficiency, small-size panel would result in a more compact overall system, the cost increases greatly with efficiency. Commercially, solar module with conversion efficiency above 15 % were found to be priced significantly higher than lower efficiency models.

In order to capture 13.2 W when the solar irradiance is $1000\,\mathrm{W\,m^{-2}}$ with a solar panel that is 15 % efficient, it needs to have an area of $\frac{13.2\,\mathrm{W}}{0.15\cdot 1000\,\mathrm{W\,m^{-2}}} = 0.088\,\mathrm{m^2}$. The specifications of the used solar panel can be found in table 4.2.

Table 4.2: Prototype solar panel specifications

Brand	Enjoy Solar
Model	Eco Line ES10P36
Rated max power	10 W
Voltage at max power	18.1 V
Current at max power	0.56 A

As can be seen from table 4.2, the chosen panel is a 10 W peak power panel, which is adequate for use in the prototype, since the 13.2 W peak is dimensioned on the basis that it rains for 24 hours, and only 30 minutes of effective solar hours are available, which in reality could occur, but will not pose an immediate problem when these conditions occur for a period shorter than the amount of days the battery can buffer for. The panel chosen is the model Eco Line ES10P36 and is shown in figure 4.2. Its specifications can be found in appendix A.

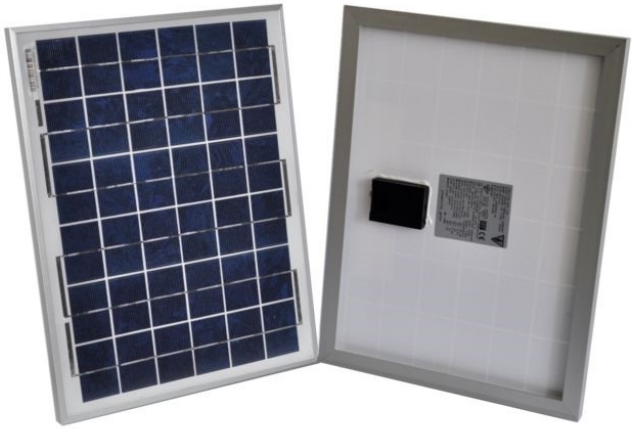


Figure 4.2: Enjoy Solar Eco Line ES10P36

4.2.3. SOLAR CHARGE CONTROLLER

As discussed at the end of the detailed design phase, in order to maximise the power from the solar panel, a solar charge controller with maximum power point tracking is to be implemented by using the LT3652 charge controller. The Prototype uses the same charge controller in a package provided by Sparkfun, called the Sunny Buddy. The Sunny Buddy has a simple layout, but was designed to convert the the output of the solar panel down to 3 V to 4 V, so it could be used without alterations in tandem with the battery and microcontroller systems Sparkfun designs for solar applications [28]. The LT3652 fortunately is capable of managing much higher rated voltages both on the input and on the output of the charge controller, and by replacing the voltage divider circuit that determined the effective battery float voltage, it has been modified to supply the 12.15 V on its output for the Battery Management system. This does mean the efficiency of the system is

reduced, as the circuit around the converter is optimised to have the highest efficiency for lower voltages, but in the final design the LT3652 can be placed in a circuit that is most efficient around the operating point of the power supply to minimise leakage and obtain in theory 90 % conversion efficiency. The Sunny Buddy is shown in figure 4.3.

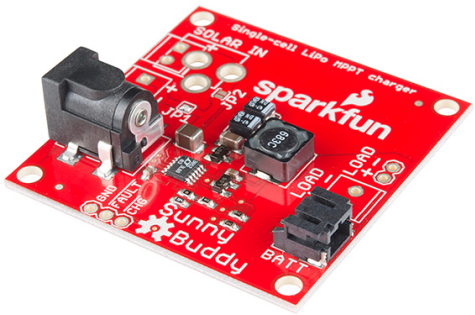


Figure 4.3: Sparkfun Sunny Buddy MPPT SCC.

4.2.4. BATTERY & BATTERY MANAGEMENT SYSTEM

In the detailed design phase, a 3S (3 in Series) configuration of Lithium-Ion cells was determined, in order to provide a voltage V_{bat} between 10 V to 12.15 V, corresponding with a state of charge from 10 % to 90 % respectively. While numerous Li-Ion cells are available on the market, the most popular, the 18650 package is used in the prototype [29]. The main difference between the various 18650 cells are supported maximum charge and discharge rate & cell capacity. The expected maximum charge and discharge rate of the batteries when employed in prototype are so low, that any of the available cells would suffice, the choice then remained between different capacities and different brands, resulting in the purchase of Panasonic 3350 mA h cells. The combined capacity is thus $3 \cdot 3.7\,\text{V} \cdot 3.35\,\text{A h} = 37.185\,\text{W h}$, resulting in a buffer of 37.185 W h/ 8.25 W h/day ≈ 4.5 days.

While the discharge and charge currents aren't expected to surpass the recommended maxima for these cells, additional protection in the form of a Battery Management System (BMS) is recommended, as described in section 3.3.3. To this end, the BMS HX-3S-25A is used to disconnect the batteries when a short would occur in the system or when the SoC drops below minimum level. The resulting combination of 3 Li-Ion cells in series with the BMS connected is shown in figure 4.4.

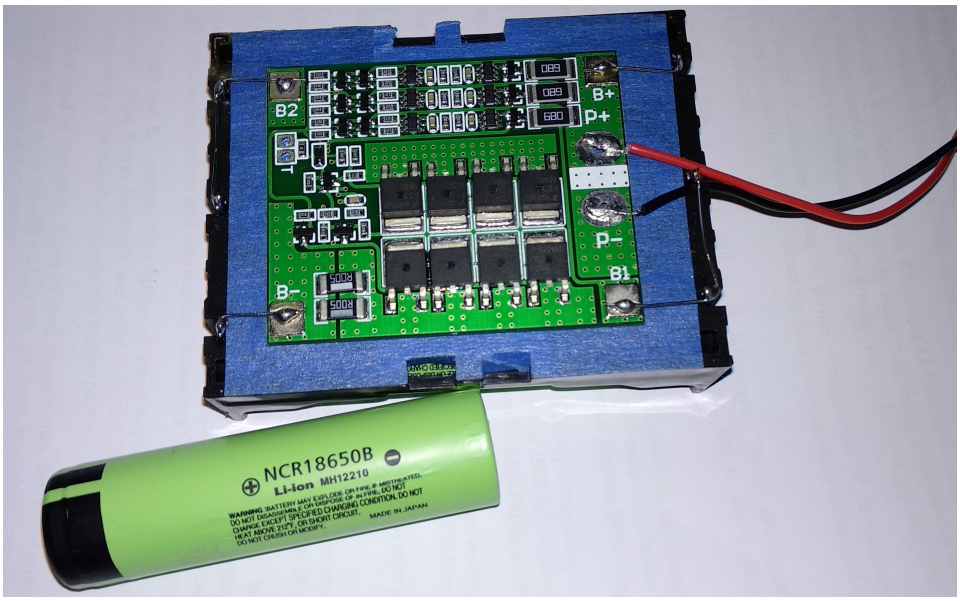


Figure 4.4: HX-3S-25A connected to a 3S configuration of Panasonic 18650 Cells.

4.2.5. BACK END

In order to receive data from the weather station, a LoRa Application server has to receive the data. The application server then has to store the received data in a database, which a graphing application will then query to present graphs to the end user. An overview of the (free and open-source) software used to fulfil these functions is presented in table 4.3.

Table 4.3: Overview of software used to receive, store and present data.

Name	Type	Function
Node-RED	Flow based programming tool	LoRa Application Server, Payload Decoder
InfluxDB	Time-Series Database	
Grafana	time series analytics and visualisation	

In order to act as a LoRa Application Server, Node-RED listens for payloads send by the weather station via the KPN LoRa network at `https://nodered.weatherstation.link/`. At arrival, the payload is then decoded and its contents send as measurements to the InfluxDB database. Grafana, which is accessible at `https://weatherstation.link/`, then queries the database in order to present the measurements in user-friendly graphs on a dashboard, of which a working concept can be seen in figure 4.5, which utilizes test data in order to provide a preview. A live version of this Dashboard is available at `https://tinyurl.com/weatherstationdashboard`.

Figure 4.5: Live Grafana Prototype Dashboard with test data.

DECODING THE LORA PAYLOAD

The payload has a size of 51 bytes, in which the 5 measurement points are laid out sequentially. Node-RED decomposes this payload into its corresponding keys and values and stores them in the InfluxDB database. An example of such a decoding operation, the one which is used in the Proof of Concept, is shown in listing 4.1.

```
// Application server payload decoder PoC
var bytes = new Buffer(msg.payload.DevEUI_uplink.payload_hex, 'hex');
var epoch = (bytes[3] << 24) | (bytes[2] << 16) | (bytes[1] << 8) | bytes[0];
var batt = (3000+10*bytes[4])/1000;
var temp = bytes[5];
var lat = (bytes[9] << 24) | (bytes[8] << 16) | (bytes[7] << 8) | bytes[6];
var lon = (bytes[13] << 24) | (bytes[12] << 16) | (bytes[11] << 8) | bytes[10];
var alt = (bytes[15] << 8) | bytes[14];
var speed = (bytes[17] << 8) | bytes[16];
var course = bytes[18];
var sats = bytes[19];
var ttf = bytes[20];
msg.payload = [
    {
    course: course,
    satellites: sats,
    time_to_fix: ttf,
    battery: batt,
    speed: speed,
    temperature: temp},
    {
    latitude: lat,
    longitude: lon,
    epoch: epoch}
];
return msg
```

Listing 4.1: PoC Payload Decoder

KPN DEVELOPER PORTAL LIMITATIONS

The prototype is connected to the KPN LoRaWAN via their developer portal, this is the same network that commercial LoRa devices connect to, but makes use of a free trial period in order to test and debug during the development process. There are a few limitations that result from the use of this developer portal, the first is that OTAA (Over The Air Activation) is disabled, such that authentication keys have to be entered manually in the LoRa Node. The second has a bit more impact, because the amount of uplink messages are capped at 6 per hour. Which means that instead of the designed measurement resolution of 1 minute, and a payload every 5, the prototype will take measurements every 2 minutes, and send them every 10. For power measurements the default 1 min / 5 min cycle is used, resulting in data being received by the LoRa Application server for only the first 30 minutes of every hour. However for demonstration purposes the internal counter of the MCU will be doubled to demonstrate 'continuous' operation.

4.3. VALIDATION

One of the main purposes of the prototype is to validate the viability of the detailed design. In order to consider the design successful, it has to be shown that the system can be assembled to work as described, and shown to behave as expected. Unfortunately, a large part of the problem definition and a key focus of this project is the continued operation of the device without grid-supplied power or regular maintenance. Even if the prototype was ready to be deployed, sufficient data to provide a confirmation of the autonomy of the system would take longer to gather than the project allows for. The reality is that with respect to the mean time to failure of the system and the extent to which maintenance needed, no data can credibly be obtained outside of lifespan guarantees and maintenance instructions of the selected components as presented by their data sheets. A validation of immediate power draw of the system is something that can be obtained however. Using the real active and sleep power of the system in the calculations from embodiment and detailed design the real average power consumption is approached. If the prototype does not exceed the estimated values of average power consumption, the power estimates used to calculate the required battery and solar panel capacity can be considered valid as well.

4.3.1. MEASURED POWER CONSUMPTION

One subsystem which was difficult to analyse in terms of power consumption is the MCU. Its power estimation has been reliant on how much of the chip can be powered down in deep sleep mode, how much it consumed while measuring, and how long waking up and going into sleep mode took. Unfortunately, early in the embodiment design phase, measurements were taken when the device was powered over USB bus, which prevented powering down the USB bus itself. The measurements were therefore a bad indication especially of how much the MCU consumed in deep sleep mode, which it would be 99.9 % of the time. In the detailed design, the theoretical lowest current draw, which was provided by the developer platform was used, but this amount of 200 µA, or 660 µW power, was expected to be much lower than the real consumption as the prototype runs at least 6 extra clocks in deep sleep to support timing the measurement cycle and detecting rain pulses. By connecting the device with a power supply at 3.3 V instead of a 5 V USB bus, an accurate measurement of the real consumption is obtained and worked into the expected power consumption to approach the real consumption.

Table 4.4: Power draw measurement on SODAQ ONE.

Status	Current draw [mA]	Power consumption [mW]
Active	12.4	42.1
Deep Sleep	0.6	1.98
Average per measurement cycle:		2.01 mW

Table 4.5: Measured consumption of the different subsystems.

Subsystem	Active power [mW]	Sleep power [µW]	Average power [mW]
Sensors	2.5	970	1
MCU	41.2	1980	2
LoRa	132	53	1.5
		Cumulative predicted load:	4.5 mW
		Total load for T_{24h}:	0.11 Wh/day

This shows that with the current prototype the system will consume 0.05 Wh more per day than was calculated in the original design 3.19. This might seem significant, as its duty cycled loads turn out to consume almost twice the expected amount, but an important factor to consider is that the system and its power supply were scaled not to supply the 0.077 Wh/day duty cycle loads, but to ensure operation when the 6.6 Wh/day rain sensor is active for the full 24 hours. This means the difference between expected and measured loads has only meant a marginal increase on the consumption side of balancing the supply and load of less than 1 %. Furthermore, this increase is mostly due to a high power consumption in the MCU during sleep mode, which can still be reduced further by disabling unnecessary clocks and systems inside the microcontroller. This was however too time-consuming an optimisation to be done in time for the Thesis. In conclusion the measured real power consumption of the system exceeds estimations, but the difference of magnitude between the rain sensor and the rest of the system with respect to power consumption ensures the power supply can support the system regardless of this.

4.3.2. LONG TERM STATE OF CHARGE SIMULATION OF THE BATTERY

The validation of the system can be aided by proper simulation as well. In the case of long-term off-grid solar applications, if the peak power output, available battery capacity and expected average or immediate load are known, it is possible to simulate the effective solar irradiation on the panel at a given location. By iteratively calculating the incoming and outgoing power in the system, the battery charge at a given month, week or day only depends on the correlation in irradiation between any set of successive days. The higher the chance an overcast day will be followed by more overcast days, the more a PV system is unable to supply a full day load cycle and the more battery is drained between charging. The Loss of Power probability is the result, where this data combined gives a probability distribution of the entire system failing due to continued lack of solar power [30]. This is thereby also the percentage of time the system is inoperable. To consider the

design valid, continuous operation by support of solar power was a primary aspect, and therefore simulation should indicate whether or not this design can continue operating in winter without fully draining its battery. To do so a simulation suite provided by the European Commission to aid solar power development turned out very suited to provide all required data, models and expertise [31]. The results of the simulation can be found in appendix B, and can be summarised with table 4.6.

Table 4.6: Loss of Power Probability / Percentage of time with empty battery

Size of PV [W_p]	Battery [Wh]	Estimated Load [Wh]	Expected LPP [%]
10	30	6.6	1.25
10	50	6.6	0.05
10	30	2	0

As shown the prototype is already almost never unable to keep the battery charged while the load is estimated at its maximum draw for 24 hours/day. When looking at the possibility to effectively reduce the loss of load probability to 0 %, doubling the available battery in size should definitely ensure operation under any reasonable conditions for the occurrence of rain and the availability of sun in the Netherlands. This itself would increase system complexity and costs only marginally, and having more capacity than required can help abate the degradation Li-Ion and Li-Po experience when charged or discharged fully. Additionally, when the more realistic rainfall probability weighted power consumption of $2\,\mathrm{Wh\,day^{-1}}$ is used as a load estimate, the 30 Wh effective battery capacity of the prototype is abundantly sufficient in sustaining the system, as 5 Wh would be just enough. Of course since the system should work in the least favourable of possibilities when concerning the weather, scaling the battery to have 0 % chance of draining is the better choice. This then should be a good validation of the capability of the system as designed and prototyped in this project to perform according to the power balance requirements.

5

Conclusions, Discussion and future work

5.1. Conclusions

In this project the goal has been to scientifically explore the possibility of, design, and prototype a weather station that would function as autonomously as possible, including sustainable off-grid power supply and long-range communication, with minimal costs to the system. The design and prototype more specifically are required to

- measure air temperature, atmospheric pressure, wind speed, wind direction, and volume of precipitation each minute
- be able to function in the Netherlands with minimal chance of losing power
- minimise the required regular maintenance to ensure proper functioning
- prioritise low production costs of the final system in addition to other requirements

In this chapter the results of the conceptual, embodiment, and detailed design process, and the success of the realised prototype will be summarised and discussed. The original program of requirements gives clear targets to reach for the design and prototype to be considered successful.

5.1.1. Results of the Conceptual design

The conceptual design focuses on creating considerations to assess the wide variety of components and subsystem solutions that can perform the required functions of collecting and storing energy, sensing weather parameters, directing operations, and communicating data back to the end user. These considerations were then used to comparatively select the best combination of solutions in order to as well as possible comply with the requirements. The most important choices made at this stage were to supply power reliably using a photovoltaic module instead of a wind source generator, and to similarly prioritise sensors with a solid state operating principle over mechanically functioning sensors. These choices were made in order to comply best with the autonomy requirements set in the programme. The solid state preference for both the sensors and supply is in order to minimise regular maintenance requirements, for a higher mean time to failure if regular maintenance would be lacking. For the supply, the high predictability of available daily solar irradiation for power compared to the more stochastic availability of wind outweighed the high seasonal correlation meaning few effective sun hours during winter in the Netherlands. The knowledge of how much solar irradiation is present during winter was found much easier to calculate than determining what a reasonable probability and duration would be for a long time lack of wind, these being worst case scenarios under which the system would need to guarantee operation.

5.1.2. Embodiment design

The Embodiment design phase was used in order to accomplish a clear overview of the technological limitations and properties that resulted from the choices made in the conceptual design in order to define

a clear boundary on the capabilities of the system. This was necessary in order to verify the choices of conceptual design before moving on to implementing the choices in the prototype and describing detailed design. To ensure achieving the requirements in the final design, this design phase first had to take into account the relevant requirements for sensors in order to select a set of sensor models, and similarly a communications module and microcontroller, before using the electrical characteristics of the selected devices to estimate the viability of the design with respect to power balance and communications framework. This allowed moving to the detailed design phase with an indication of the largest amount of supplied power these subsystems would need while under normal operating circumstances, which helped set the resulting requirements for the electrical properties of the PV module to be selected for the detailed design and prototyping. It also provided a timing framework that allowed the system measure to all parameters each minute in accordance with the system requirements, and transmit this data under the constraints of the selected LoRaWAN communications protocol.

5.1.3. DETAILED DESIGN

All components and subsystems choices which have not been made specifically in the earlier design phases are to be decided upon in the detailed design phase. This means that phase includes determining the size of the PV module and battery capacity based on the power calculations from embodiment design, which are refined as well in the detailed design. Another part of this stage is defining the code structure used to sequence and time measurements, process data and properly transmit the information from the system to the data servers for online processing and visual presentation. Where in the early design phases choices were made that supported the trade-off requirements for autonomy through low maintenance sensors that complied with the functional requirements for these sensors, during the detailed design the other side of this trade-off had to be upheld by the requirement of continuous operation. The selected solid-state sensors and photovoltaic module could only be used successfully in the system if it can be proven the design holds to the requirement of continuously functioning under the worst reasonable environmental conditions. This has been accomplished by calculating the worst case power draw of each of the subsystems combined, and determining what size, efficiency, and rated peak power is required for the PV module to produce that much power during the winter months. This ensures that in any winter the PV module on average supplies as much or more energy than the other subsystems dissipate. The battery can act as a buffer to ensure that any daily available energy is an average of an amount of preceding days proportional to battery size. The trade off requirement of reducing the chance of supply failure can thus be met by increasing battery size of the system.

5.1.4. PROTOTYPE AND VALIDATION

In this project one of the primary tasks has been to develop a prototype of the final design, both to test and validate the choices made in the design stages and to present a working prototype for the contractor to use at the end of the project. As soon as the embodiment en detailed design phase provided a detailed view of which components and sensors would be in the weather station, a prototype was constructed to match the design choices as much as possible. This allowed limited testing of the final design, but unfortunately due to the nature of the problem analysis, both the long term power balance of the system and low maintenance requirements cannot be assessed in realistic operating conditions properly. Fortunately developing the prototype has shown the functionality and helped to clarify the subsystem layout of the final design, and provided conclusive measurements that supported the power calculations from earlier design steps. In addition to this, the Photovoltaic Geographical Information Sysytem research project, a research group EU Commision Joint Research Centre, was vital in providing a complete set of simulations of the final system design which proved invaluable in assessing the long term energy balance of the system. Finally, in developing the prototype along the design stages, the financial costs of different components can be more accurately determined to give a preliminary figure to prototyping costs. The final design and prototype were not within the initially required costs, but the single ATMOS 22 wind sensor included was over half of the total expenses, and was included only after this decision was approved by the contractor.

5.2. FUTURE WORK

A significant portion of this project and thesis has been working towards the design and validation of a weather station that used current technological advancements to expand the application possibilities of the system by supporting long range, low maintenance autonomy. The resulting design and prototype were able to confirm the viability of such a system operating in the Netherlands, but over the course of research and development much more has been discovered with respect to additional technologies that can further expand the functionality of the device, or improve its current effectiveness. In this chapter, before fully discussing the complete design, process and results of the project, first a selection of system additions or improvements is discussed that may provide immediate gains to the performance of the system. Immediate improvements that are described here consist of increasing the data efficiency of the LoRa module, adding to the selection of available sensors, and dynamically observing and adapting power consumption.

5.2.1. UTILISING LORA'S FULL POTENTIAL

In section 3.2.2 the basic properties and limitations of the LoRaWAN-protocol were explained along with the data transferred. As stated, 7 bytes/minute were needed in order to meet requirements, meaning that an additional 16 bytes per 5 minutes are unused and can be used for other purposes. This empty space can be used in numerous ways, a few of which are discussed here.

REQUESTING MISSING DATA

While LoRaWAN is able to provide 2-way communication, the limitations on downlink messages are very severe. The reason for this is that during the transmission of a downlink message by the LoRa Gateway, receiving uplink LoRa messages from end-devices is blocked. In order to maximise the network's QoS (Quality of Service), as few downlink messages as possible should be sent. Acknowledgements also count as downlink messages, which makes acknowledging every uplink impossible. This results in the loss of messages in case of collisions, since the end-device can't know whether the message it has send is received successfully.
In order to improve the reliability without the use of acknowledgements, a "missing data request" scheme can be implemented. The server knows which payloads are missing, since they should contribute to a measurement resolution of 1 minute, allowing the server to compile a list of time-stamps corresponding with the missing payloads which it can downlink to the weather station. The weather station can then use the available 16 bytes per payload to send a missing measurement (including a timestamp) and in this way can provide the server with up to 1 missing payload every 25 minutes. (1 missing measurement per 5 minutes.)

PROVIDING SYSTEM STATUS

When there are no missing payloads, this empty space can be used to provide additional, non-meteorological, data to the end-user / server. This can include, but is not limited to, battery state-of-charge, brown-out detection (whether the system has experienced a loss of power), sensor status and potential errors.

DYNAMIC TRANSMISSION

While a minimum duty-cycle of 4 minutes and 7 seconds is used in the Design phase to meet LoRa specifications under all circumstances, which is rounded up to 5 minutes for elegant division of measurements per transmission. This delay between transmissions is only needed when a spreading factor of 12 is required to obtain a connection with the gateway. Instead of hard-coding a spreading factor, ADR (Adaptive Data Rate) can be used to find the optimum (lowest) spreading factor. A lower spreading factor results in a shorter transmission duration (time-on-air), which in turn results in the ability to do more uplink transmissions without exceeding the LoRaWAN specifications. These 'extra' transmissions could be filled by measuring more often (depending on available system power), and by providing missing payloads or system status information in separate transmissions instead of in the discussed unused bits.

In conclusion, a combination of these measures results in a system that:

- Can provide missing data and other information by utilising the available payload space with a SF of 12.
- Can measure more often and provide missing data and other information by transmitting more often when the SF falls below 12.

5.2.2. ADAPTIVE POWER MANAGEMENT

SOLAR TRACKING

Another optimization that could have merit is physically tracking the solar angle with the solar panel. The trade-off that comes into play is between the increase in available power on the one hand, which could result in a cheaper (lower rated) panel, and increased complexity due to the tracking hardware needed on the other, which result in a shorter time to failure, increased cost and increased power dissipation. Expected gains relative to a fixed solar panel vary between 15 % to 35 % [32, 33], meaning that in order to benefit the system, a $\approx 35\,\%$ larger solar panel should at least be more expensive than the total cost of dual-axis solar tracking. With the trend of ever decreasing prices of photovoltaic panels, it is unlikely that this results in a trade-off favouring solar tracking over fixed panels for low-power and low-maintenance systems. However, novel developments in solar tracking hardware could change this perspective making it an interesting prospect for future work.

DYNAMIC MEASUREMENT DUTY-CYCLES

When discussing possible 'extra' transmissions in section 5.2.1, it is noted that whether or not such extra transmissions are possible not only relies on limitations of the communication stack, but also on the available amount of energy. Two methods of dynamically handling changing weather conditions, and with that, a change in the amount of energy that can be harvested, can be thought of. The first applies algorithms to data acquired by the system, past and present, in order to predict and manage the power flows, described in [34–36]. Another that tries to estimate the amount of effective sun hours in advance by using the weather forecast, which are described in [37–39]. In that case, the system must be able to receive transmissions to calculate with a weather forecast. Of course, these methods are not mutually exclusive, and an implementation of 1 or both would allow the system to rise above its worst-case conditions performance.

BATTERY CHARGING IN SUB ZERO AMBIENT TEMPERATURES

As stated in 3.2.4, li-ion batteries should not normally be charged when the batteries are below a temperature of 0 °C. This can cause irreversible lithium build-up on the anode of the battery. There are several solutions to combat this, which can be used simultaneously. The first is to huddle all the electrical components of the sensor platform into a single casing, insulating that casing from the ambient temperatures. The dissipated energy of those components could heat the components marginally. It is also expected that at most sub zero temperatures, the battery would not charge, since most cases of sub zero temperatures occur in the winter during the night. Depending on the power balance of the system in winter, a low power heater could act as a safety measure should insulation not suffice. The first solution might not be sufficient, since in the Netherlands there is a nonzero number of winter days where temperature does not rise above 0 °C. The second solution uses the solar charge controller, which can be connected to a 10 kΩ thermistor, which cuts off the charge current when the battery temperature reaches a temperature below 0 °C. Although this is effective in protecting the battery, preventing all charging during a sub zero day disrupts the power balance of the system significantly. It is thus recommended to primarily explore possible heating or insulation methods to prevent this issue with a reasonable degree of certainty, and use the charging shutoff backstop to ensure safety parallel to this. Long term testing on the internal temperature of the sensor platform is necessary in order to verify these solutions.

5.2.3. THE SENSOR MODULARITY ASPECT

This project mainly focused on the design and prototyping of an autonomous wireless weather station. This design was taken in to parts: the sensors, and the sensor platform which supports those sensors. One benefit of the sensor platform is that its design can be taken separately from the sensors. The sensor platform can then be taken in order to support any sensor in order to fit the demands of varying customers.

In the current design, the sensors communicate by SDI, I^2C and by reading a digital pin. The microcontroller can also support sensors which use UART and analog communication. There are currently two ways in order to make the system modular to fit any sensor a consumer desires. The modularity will of course result in trade offs, which will be discussed. The first is by use of I^2C only. In this model, the designers of the system will select a number of I^2C sensors which the consumer can choose from. These sensors can then be used on the system in a plug-and-play fashion. The system then knows which sensor is being connected by storing the I^2C slave addresses of the selected sensors and sweeping the sensors for those addresses. The second option for modularity is when the consumer selects a sensor with a different type of communication. The

designer can then modify the platform in order to fit the consumer demands.

Any addition of sensors will of course result in a number of trade offs. While it will not always be the case, LoRa has a worst case scenario of 51 bytes per 5 minutes, limiting the amount data which can be sent per time. More sensors implies a higher power draw, which can be compensated for by scaling the energy subsystems accordingly. When using more sensors, a careful consideration must be made between amount of sensors, size of the energy systems, the rate at which data is acquired and the rate at which data is sent.

5.2.4. ENVIRONMENTAL CONCERNS AND DISPOSAL

Because the system is designed for remote weather sensing, it is important to note that the system should not interfere with its environment and vice versa. The systems should be cased in such a way that its components are shielded from hazards to prevent damage and possible pollution to the environment. When the system has reached the end of its lifetime, the system should removed from its environment. The system should be taken apart into comparable sets of similar disposable components and recycled when possible.

5.3. DISCUSSION ON THE AUTONOMOUS WIRELESS WEATHER STATION

From the conclusions, the viability of an autonomous and wireless weather station was proven to be not only theoretically possible, but has shown to be realistic in practice. There is no way to bypass the correlation between increased autonomy and financial cost, but due to the ever evolving technologically advancement and decreasing cost of solar power due to mass production, the reasons for weather stations to be connected by wire to the power grid or the internet are becoming less significant. Advancements especially in LoRaWAN have in the project allowed the use of this technology effectively to support both the long range and low power requirements, whereas previous technologies such as cellular networks would be less effective. An important feature of the research and design is also the separation between platform and sensors, which allows the design process to be applied similarly to a selection of different or improved sensors. As long as the payload conditions of the LoRa-module are met, the current design in theory supports any combination of sensors with a combined load of 6 Wh/day. This also means that flaws that can be found with the current sensors can be addressed when alternatives are available. The two sensors that stand out from the start are the wind and precipitation sensor, and both are sources of considerable added costs to the system. Especially the selection of a precipitation sensor seemed to have no right answer, and caused the resulting choice to outscale all other sensors with respect to power draw. This was decided on after other choices proved even less compatible with the program of requirements or did not have enough information on commercially available models to compare with clarity. A direct example was the acoustic disdrometer rain sensor, for which commercially available sensor packages could not be considered with the depth of other sensor types as no datasheet or quantitive information on the specifications for this model sensor could be found. The selected rain sensor as a result was the lesser of 3 bad options, as the mechanical gauge would run opposed to requirement [5.2], and the radar based sensor had an even larger power draw than the optical gauge. It is therefore a recommendation of this report that the first and foremost priority for improving the performance of the system is to find or develop a solid state rain sensor that requires less power, either by improving on the design of the RG-11 sensor, which was not made with this specific low power application in mind, or by exploring the possibility of replacing it with a more efficient disdrometer.

In a similar, but less forced trade off the selected model acoustic wind sensor, the ATMOS 22, was initially considered too expensive to incorporate in the design or prototype, but after clarification by the contractor was added as it was agreed the model should be included regardless of the cost requirements. However, if costs are a more mandatory requirements a replacement model can be used in the system instead. Aside from replacing the model with a cheaper sonic anemometer, cheaper high-end mechanical wind sensors like the Davis 6410 anemometer have been also found which according to specifications can operate without maintenance in accordance with requirement [5.2] and [5.3].

Despite the fact that clear improvements can still be made to the specific sensors attached to the platform, the possibilities of the platform as a separate product and as a unified design that can be scaled and adapted to the needs of the customer are numerous. The project challenge has revolved around both balancing the power and data flowing through the system, and that balance, both of power draw and supply, and gathered data and transmission payload size, can be maintained even with larger or smaller sets of sensors. The room

left in the LoRa payload can support more, or more accurate, sensors, and as long as the power remains balanced, the design principles demonstrated in this project should hold. Balancing the power flow does not strictly have to mean attaching a larger panel, and optimising the scale of the panel and battery has multiple equally valid solutions for the same Loss of Load Probability [40]. This also means the increased financial costs of expanding power supply can be adaptive to market forces and available models, as more than one combination of battery/pv sizes are equally valid solutions.

A

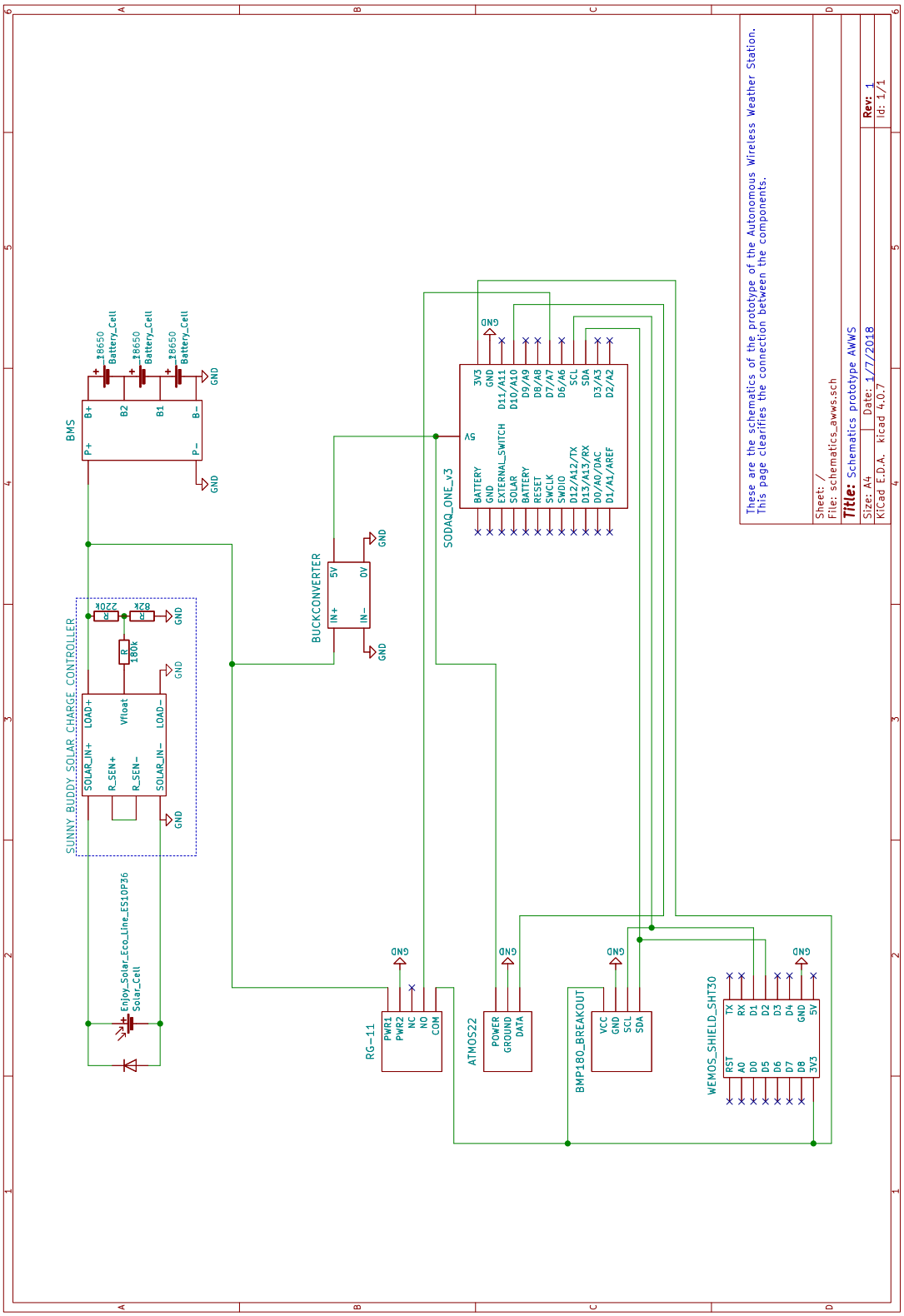
ADDITIONAL SPECIFICATIONS AND SCHEMATICS

SPECIFICATIONS ECO LINE ES10P36 SOLAR PANEL

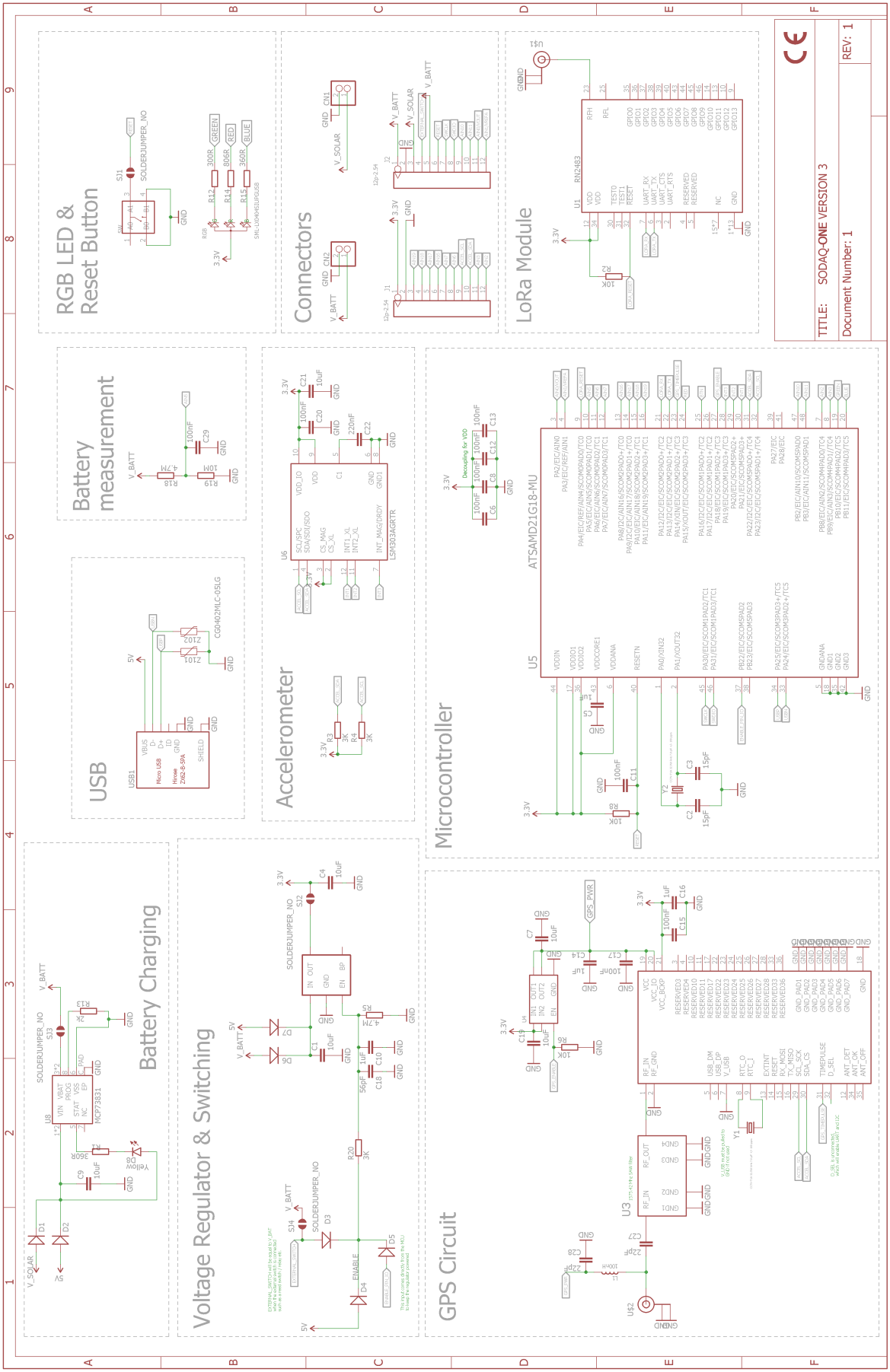
Table A.1: Specifications for the Eco Line ES10P36 solar panel.

Parameter	Symbol	Unit	Value
Rated Max Power	P_{max}	[W]	10
Power Tolerance Range		[%]	0 to 3
Voltage at P_{max}	V_{mp}	[V]	18.1
Current at P_{max}	I_{mp}	[A]	0.56
Open-circuit Voltage	V_{oc}	[V]	22.3
Short-circuit Current	I_{sc}	[A]	0.60
Normal Operating Cell Temp	$NOTC$	[°C]	50
Maximum System Voltage	V_{DC}	[V]	1000
Dimension		[mm]	350x260x25
Cell quantity and array			36 (4x9)

SCHEMATICS PROTOTYPE



SCHEMATICS SODAQ ONE VERSION 3



B

SIMULATIONS

European Commission

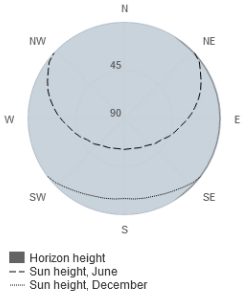
Performance of off-grid PV systems

PVGIS-5 estimates of solar electricity generation

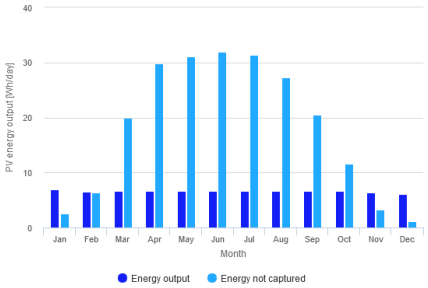
Provided inputs

Latitude/Longitude:	51.999, 4.374	Slope angle:	35 °
Horizon:	Calculated	Azimuth angle	0 °
Database used:	PVGIS-CMSAF	**Simulation outputs**	
PV installed:	10 Wp	Percentage days with full battery:	78.53 %
Battery capacity:	30 Wh	Percentage days with empty battery:	1.26 %
Cutoff limit:	0 %	Average energy not captured:	23.08 Wh
Consumption per day:	6.6 Wh	Average energy missing:	2.12 Wh

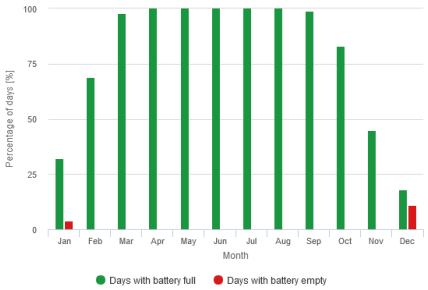
Outline of horizon at chosen location:



Power production estimate for off-grid PV:



Battery performance for off-grid PV system:

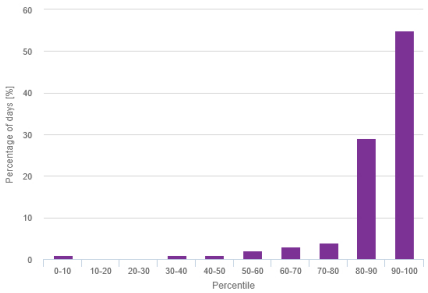


Monthly average performance

Month	Ed	El	Ff	Fe
January	6.86	2.6	32	4
February	6.55	6.4	69	0
March	6.68	20	98	0
April	6.61	29.8	100	0
May	6.6	31.1	100	0
June	6.6	31.9	100	0
July	6.6	31.4	100	0
August	6.59	27.3	100	0
September	6.59	20.5	99	0
October	6.58	11.6	83	0
November	6.4	3.3	45	0
December	6.14	1.1	18	11

Ed: Average energy production per day [Wh/day].
El: Average energy not captured per day [Wh/day].
Ff: percentage of days when battery became full [%].
Fe: percentage of days when battery became empty [%].

Probability of battery charge state at the end of the day:



Cs	Cb
0-10	1
10-20	0
20-30	0
30-40	1
40-50	1
50-60	2
60-70	3
70-80	4
80-90	29
90-100	55

Cs: Charge state at the end of each day [%].
Cb: percentage of days with this charge state [%].

Joint Research Centre

Report generated on 2018/06/12

European Commission

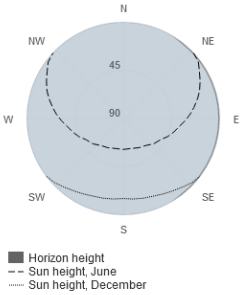
Performance of off-grid PV systems

PVGIS-5 estimates of solar electricity generation

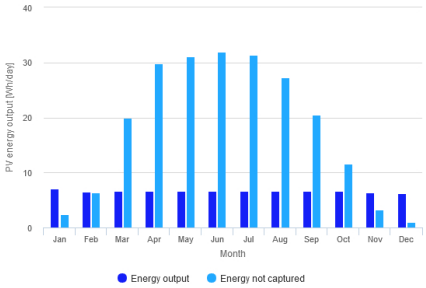
Provided inputs

Latitude/Longitude:	51.999, 4.374	Slope angle:	35 °
Horizon:	Calculated	Azimuth angle	0 °
Database used:	PVGIS-CMSAF	**Simulation outputs**	
PV installed:	10 Wp	Percentage days with full battery:	78.2 %
Battery capacity:	50 Wh	Percentage days with empty battery:	0.05 %
Cutoff limit:	0 %	Average energy not captured:	23.14 Wh
Consumption per day:	6.6 Wh	Average energy missing:	0.18 Wh

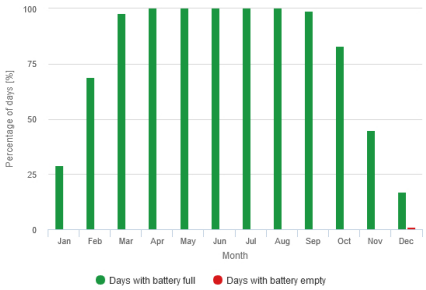
Outline of horizon at chosen location:



Power production estimate for off-grid PV:



Battery performance for off-grid PV system:

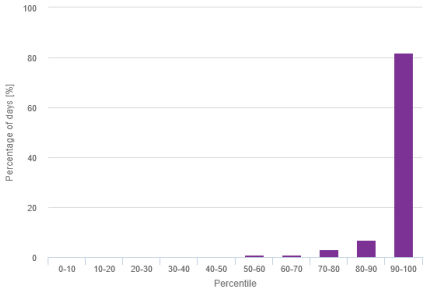


Monthly average performance

Month	Ed	El	Ff	Fe
January	7.11	2.4	29	0
February	6.55	6.4	69	0
March	6.68	20	98	0
April	6.61	29.8	100	0
May	6.6	31.1	100	0
June	6.6	31.9	100	0
July	6.6	31.4	100	0
August	6.59	27.3	100	0
September	6.59	20.5	99	0
October	6.58	11.6	83	0
November	6.4	3.3	45	0
December	6.2	1	17	1

Ed: Average energy production per day [Wh/day].
El: Average energy not captured per day [Wh/day].
Ff: percentage of days when battery became full [%].
Fe: percentage of days when battery became empty [%].

Probability of battery charge state at the end of the day:



Cs	Cb
0-10	0
10-20	0
20-30	0
30-40	0
40-50	0
50-60	1
60-70	1
70-80	3
80-90	7
90-100	82

Cs: Charge state at the end of each day [%].
Cb: percentage of days with this charge state [%].

Report generated on 2018/06/12

Joint Research Centre

European Commission

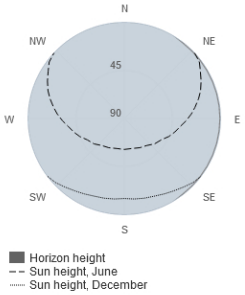
Performance of off-grid PV systems

PVGIS-5 estimates of solar electricity generation

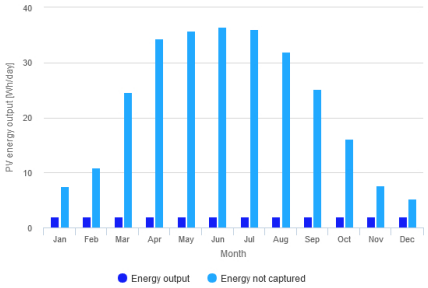
Provided inputs

Latitude/Longitude:	51.999, 4.374	Slope angle:	35 °
Horizon:	Calculated	Azimuth angle	0 °
Database used:	PVGIS-CMSAF	**Simulation outputs**	
PV installed:	10 Wp	Percentage days with full battery:	96.44 %
Battery capacity:	30 Wh	Percentage days with empty battery:	0 %
Cutoff limit:	0 %	Average energy not captured:	23.53 Wh
Consumption per day:	2 Wh	Average energy missing:	0 Wh

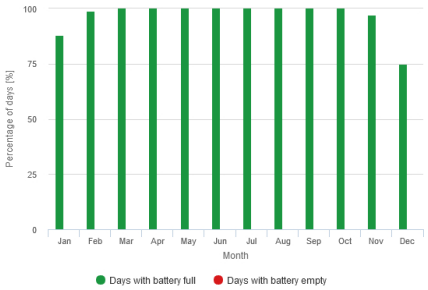
Outline of horizon at chosen location:



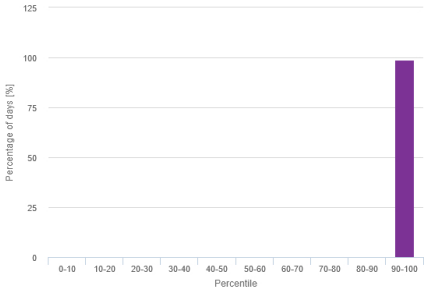
Power production estimate for off-grid PV:



Battery performance for off-grid PV system:



Probability of battery charge state at the end of the day:



Monthly average performance

Month	Ed	El	Ff	Fe
January	2	7.5	88	0
February	2	10.9	99	0
March	2	24.6	100	0
April	2	34.4	100	0
May	2	35.7	100	0
June	2	36.5	100	0
July	2	36	100	0
August	2	31.9	100	0
September	2	25.1	100	0
October	2	16.1	100	0
November	1.99	7.7	97	0
December	2	5.2	75	0

Ed: Average energy production per day [Wh/day].
El: Average energy not captured per day [Wh/day].
Ff: percentage of days when battery became full [%].
Fe: percentage of days when battery became empty [%].

Cs	Cb
0-10	0
10-20	0
20-30	0
30-40	0
40-50	0
50-60	0
60-70	0
70-80	0
80-90	0
90-100	99

Cs: Charge state at the end of each day [%].
Cb: percentage of days with this charge state [%].

Joint Research Centre

Report generated on 2018/06/12

C

Source Code

MAIN.CPP

```
#include <Arduino.h>                    // Include Arduino Library
#include <Wire.h>                       // Include I2C Library
#include <Sodaq_RN2483.h>               // Include LoRa Module Library
#include <Sodaq_wdt.h>                  // Include SODAQ WatchDogTimer Library

#include "LedColor.h"                   // for USBstreamless debugging
#include "BMP180.h"                  // imported Library for communicating with the pressure sensor
#include "LoRaFunc.h"                   //separated lora functions to clean up main.cpp
#include "regSetup.h"                   //separated register operations to initialise hardware functions
#include "SDI12func.h"                  //SDI-12 functions and definitions. imports and makes use of the enviroDIY SDI12 library
#include "SHT30.h"                   // imported library for communicating with the humidity sensor
#include "dataBuffer.h"              // our functions for managing the data and formatting it correctly for transmission

//Most headers include these definitions themselves for debugging, but these must exist in main if no previous similar definitions exist
#ifndef DEBUG_STREAM
#define DEBUG_STREAM SerialUSB          // Define SerialUSB as debug stream
#define DEBUG_BAUD 9600                 // Define Serial BAUD
#endif
#ifndef LORA_STREAM
#define LORA_STREAM Serial1             // Define Serial1 as LoRa stream
#endif

#ifndef SDI_DATA_PIN
#define SDI_DATA_PIN 10          // The pin of the SDI-12 data bus
#endif

//these are all the individual class initialisations for communicating with the ATMOS22,SHT30,and BMP180
extern SDI12 mySDI12;
ClosedCube_SHT31D sht3xd;
BMP180 bmp;

SHT31D result; //used to collect and use the result from SHT30 as its class defines a special struct for returning information

dataBuffer databuf; //initialise the databuffer object to encode, store and concactenate all measurement data

char atmosAddress; //address at which the SDI12 initialisation has detected the ATMOS22 sensor

float temperature,hPa,windSpeed,windDir,humidity,rainVolume; //the relevant variables for the weather parameters being measured

long pressure,t2; //temporary variable since the BMP180 library returns long instead of float, whereas we process measurements as floating point values
int bmperror;     //used to check if BMP measurement was performed and communicated correctly
```

```
42 int measureflag = 0; //polling flag that gets checks continuously during loop() to indicate start of
       measurement. set by RTC, reset by measurement
43 int iteration = 0; //keeps track of the current iteration, to correctly synchronise the measurement and
       transmission cycles
44
45
46 void setup()
47 {
48   #if defined(LORA_RESET)     // Enable LoRa module by Hard reset the RN module
49   pinMode(LORA_RESET, OUTPUT);
50   digitalWrite(LORA_RESET, LOW);
51   delay(100);
52   digitalWrite(LORA_RESET, HIGH);
53   delay(100);
54   #endif
55
56   /*
57   bugs were detected when initialising the Wire library after data streams and/or SDI communication
58   only initialise Wire once, check that libraries do not also use begin()
59   the Wire object is external and can be used by all functions across files that include the library
60   */
61   Wire.begin();
62
63   while ((!DEBUG_STREAM) && (millis() < 10000)){}              // Wait for SerialUSB or start after 10
       seconds
64   //DO NOT REMOVE. This function makes sure that under any conditions the board can be reset to
       bootloader mode (for flashing new code) in the first 10 second
65
66   DEBUG_STREAM.begin(DEBUG_BAUD);
67   LORA_STREAM.begin(LoRaBee.getDefaultBaudRate());
68   //  DEBUG_STREAM.println("setting up LoRa");
69   setupLoRaABP();
70
71   LoRaBee.setSpreadingFactor(12);                              // Set initial Spreading Factor SF to 12
72
73   //initiate the I2C connections with their respective libraries
74   SHT31D_ErrorCode err = sht3xd.begin(0x45); // I2C address: 0x44 or 0x45, depends the sht package
75   if(err){DEBUG_STREAM.print("Error on SHT30 connect");}
76
77   int errcode =  bmp.begin();
78   if(errcode){DEBUG_STREAM.print("Error on BMP180 connect");}
79
80   //setup the Real Time Counter
81   rtcSetup();
82
83   //setup the external event counting funciontality for counting rain gauge pulses
84   evsSetup();
85   eicSetup();
86   tcSetup();
87
88   //setup SDI communication, returns ' ' on unsuccessful initiation, first detected responsive address
       otherwise
89   atmosAddress = sdiSetup();
90   setLedColor(NONE);
91   if(atmosAddress == ' '){DEBUG_STREAM.println("Error on ATMOS22 connect");}
92
93
94   DEBUG_STREAM.println("setup complete");
95   RTC->MODE0.CTRL.bit.ENABLE = 1;  //start the Real Time Counter at the end of setup
96
97 }
98
99 void RTC_Handler(void)                  // Function to clear flag, then make sensor measurement
100 {
101   RTC->MODE0.INTFLAG.bit.CMP0 = 1;    // if flag high and IRQ handler is called, write a 1 to remove flag
102   measureflag = 1;                    // this flag indicates a measurement should be taken
103 }
104
105
106 //Returns and resets to 0 the current value of the event counting Timer/Counter register, to find the
       precipitation volume
```

```
107 //
108 uint16_t tcRead(void)
109 {
110   while (TC3->COUNT16.STATUS.reg & GCLK_STATUS_SYNCBUSY);
111   uint16_t count = TC3->COUNT16.COUNT.reg;
112   TC3->COUNT16.COUNT.reg = 0;
113   while (TC3->COUNT16.STATUS.reg & GCLK_STATUS_SYNCBUSY);
114   return count;
115 }
116
117 void loop()
118 {
119
120   if(measureflag == 1) // we only start a measurement after the RTC has reached a minute, setting the '
        take measurement' flag
121   {
122    //take a combined temperature and humidity measurement with the SHT3X library
123    result = sht3xd.readTempAndHumidity(SHT3XD_REPEATABILITY_HIGH, SHT3XD_MODE_POLLING, 15);
124    if (result.error == SHT3XD_NO_ERROR)
125    {
126    /*
127       insert debugging functions to convey correct operation, or print results over the USB stream
128    */
129    }
130    else
131    {
132    /*
133       indicate error in the SHT30 measuremet
134    */
135    }
136
137    //take a combined temperature/pressure measurement with the BMP180 library
138    bmperror = bmp.readTP(&t2, &pressure, OVERSAMPLING_HIGH_RESOLUTION);
139    humidity = result.rh;
140
141    if (bmperror == 0)
142    {
143    /*
144       insert debugging functions to convey correct operation, or print results over the USB stream
145    */
146    }
147    else
148    {
149    /*
150       indicate error in the BMP180 measurement
151    */
152    }
153
154    // read out the counter keeping track of how many rain pulses have been detected
155    rainVolume = tcRead()*0.2;
156
157    //currently the only indication of a bad SDI connection is a faulty setup, so we check for that before
         measuring wind speed/direction
158    if(atmosAddress != ' ') sdiMeasure(&windSpeed,&windDir,atmosAddress);
159    //!! If a measurement is taken without a proper SDI connection, this can freeze the system into waiting
         for a reply!!
160
161    //after all measurements have been taken, construct a new dataset out of it
162    databuf.newDataset(&temperature,&hPa,&windSpeed,&windDir,&humidity,&rainVolume,0);
163    //and place it in the correct position of the full 5 measurement buffer for transmission later
164    databuf.fillBuffer(iteration);
165
166    /*
167    this sequencing of increments means our first measurement is placed into the buffer with position 0,
168    and successive measurements are placed correctly, while also ensuring that after 5 measurements
169    iteration = 5 in order to correctly trigger a transmission and reset
170    */
171    iteration++;
172
173    if(iteration >= 5)   //at 5 measurements the system is ready for a data transmission, resetting the
         cycle
```

```
174     {
175
176       iteration = 0;
177       int error = LoRaBee.send(1,databuf.head,40); //transmit the databuffer by pointing to head and
          sending the 40 bytes inside
178
179       if(error){DEBUG_STREAM.print("A LoRa error has occurred");DEBUG_STREAM.println(error);}
180       else {DEBUG_STREAM.print("LoRa Tx successful");}
181     }
182   measureflag = 0;   //after a full measurement cycle, the flag is reset and the system waits idly for the
        Real Time Counter to set it again
183   }
184 }
```

pages/sourcecode/main.cpp

REGSETUP.H AND .CPP

```
1  #ifndef REGSETUP_h
2  #define REGSETUP_h
3  //Real Time Counter controls duty cycle, initiates clocks to trigger measurement at 1/60Hz
4  void rtcSetup(void);       //sets up registers for Real Time Counter of 1 minute
5  void rtcSetup(int timer);  //sets up registers for a counter of *timer* ms
6  /*
7  In order to count external pulses from raingauge, external pin is used to generate an event instead of an
       interrupt,
8  which can be passed without waking the CPU and does not interfere with other external interrupts, for
       instance
9  the SDI-12 communication (which blocks all other external interrupts while expecting to receive a message)
10 */
11
12 void eicSetup(void);       //sets up register for external pin event generation, linking a pin to an
       interrupt and then generating an event instead
13 void evsSetup(void);       //sets up registers for event handler system, which passes the generated event
       along a channel to a counter
14 void tcSetup(void);        //sets up registers for the event counter, which increments when EIC generates
       an event which is passed on by EVSYS
15
16 #endif
```

pages/sourcecode/regSetup.h

```
1  #include <Arduino.h>                    // Include Arduino Library
2  #include <Sodaq_RN2483.h>               // Include LoRa Module Library
3  #include <Sodaq_wdt.h>                  // Include SODAQ WatchDogTimer Library
4  #include "regSetup.h"                   // Include function declarations
5
6  void rtcSetup(void)                     // Function to Setup the RTC & Timers
7  {
8    //Set RTC source clock to always be running, even in standby, enable it and wait for register changes to
         sync
9    SYSCTRL->OSC32K.bit.ONDEMAND = 0;
10   SYSCTRL->OSC32K.bit.RUNSTDBY = 1;
11   SYSCTRL->OSC32K.bit.EN32K = 1;
12   SYSCTRL->OSC32K.bit.ENABLE = 1;
13   while((SYSCTRL->PCLKSR.bit.OSC32KRDY) == 0){}
14
15   //Use the source clock to generate a Generic Clock signal of 1kHz (Divide by 32). Uses the default RTC
       ID number to select which GCLK. wait for sync
16   GCLK->GENDIV.reg = ( GCLK_GENDIV_DIV(32) | GCLK_GENDIV_ID(RTC_GCLK_ID));
17   GCLK->GENCTRL.reg = (GCLK_GENCTRL_GENEN | GCLK_GENCTRL_SRC_OSC32K | GCLK_GENCTRL_ID(RTC_GCLK_ID));
18   while (GCLK->STATUS.reg & GCLK_STATUS_SYNCBUSY);
19
20   //enable the GCLK signal and connect the RTC to the GCLK with its ID, enable clock. Wait for sync
21   GCLK->CLKCTRL.reg = (GCLK_CLKCTRL_ID(GCLK_CLKCTRL_ID_RTC) | GCLK_CLKCTRL_GEN(RTC_GCLK_ID) |
       GCLK_CLKCTRL_CLKEN);
22   while (GCLK->STATUS.reg & GCLK_STATUS_SYNCBUSY);
23
24   RTC->MODE0.COUNT.reg = 0;          //reset counter bc a software reset no longer clears the value (due to
       sleep mode running)
```

```
25   RTC->MODE0.CTRL.reg = 0x0080;      //select Mode0, no clock divider for 1kHz, Clear on match
26   RTC->MODE0.INTENSET.reg = 0x01;    //compare interrupt enabled
27   RTC->MODE0.COMP[0].reg = 118000;  //timer period in ms (small bias to offset unknown f deviation from
       1000 Hz)
28   NVIC_EnableIRQ(RTC_IRQn);
29 }
30
31 void rtcSetup(int timer)                      // Function to Setup the RTC & Timers
32 {
33
34   SYSCTRL->OSC32K.bit.ONDEMAND = 0;
35   SYSCTRL->OSC32K.bit.RUNSTDBY = 1;
36   SYSCTRL->OSC32K.bit.EN32K = 1;
37   SYSCTRL->OSC32K.bit.ENABLE = 1;
38   while((SYSCTRL->PCLKSR.bit.OSC32KRDY) == 0){}
39
40
41   GCLK->GENDIV.reg = ( GCLK_GENDIV_DIV(32) | GCLK_GENDIV_ID(RTC_GCLK_ID));
42   GCLK->GENCTRL.reg = (GCLK_GENCTRL_GENEN | GCLK_GENCTRL_SRC_OSC32K | GCLK_GENCTRL_ID(RTC_GCLK_ID));
43   while (GCLK->STATUS.reg & GCLK_STATUS_SYNCBUSY);
44
45   GCLK->CLKCTRL.reg = (GCLK_CLKCTRL_ID(GCLK_CLKCTRL_ID_RTC) | GCLK_CLKCTRL_GEN(RTC_GCLK_ID) |
       GCLK_CLKCTRL_CLKEN);
46   while (GCLK->STATUS.reg & GCLK_STATUS_SYNCBUSY);
47
48   RTC->MODE0.COUNT.reg = 0;          //reset counter bc a software reset no longer clears the value
49   RTC->MODE0.CTRL.reg = 0x0080;      //select Mode0, no clock divider for 1kHz, Clear on match
50   RTC->MODE0.INTENSET.reg = 0x01;    //compare interrupt enabled
51   RTC->MODE0.COMP[0].reg = timer;    //timer period in ms
52   NVIC_EnableIRQ(RTC_IRQn);
53 }
54
55 void evsSetup(void)
56 {
57   PM->APBCMASK.reg |= (PM_APBCMASK_EVSYS);
58
59   GCLK->CLKCTRL.reg = (GCLK_CLKCTRL_ID(GCLK_CLKCTRL_ID_EVSYS_1) | GCLK_CLKCTRL_GEN(RTC_GCLK_ID) |
       GCLK_CLKCTRL_CLKEN);
60   while (GCLK->STATUS.reg & GCLK_STATUS_SYNCBUSY);
61
62   EVSYS->USER.reg |= (uint16_t) ( EVSYS_USER_CHANNEL(2) | EVSYS_USER_USER(0x12) ); //connect User TC3 to
       event with Channel 1 (selecting channel n-1)
63   EVSYS->CHANNEL.reg |= (uint32_t) ( EVSYS_CHANNEL_EDGSEL_RISING_EDGE| EVSYS_CHANNEL_PATH_SYNCHRONOUS |
       EVSYS_CHANNEL_EVGEN(0x13) | EVSYS_CHANNEL_CHANNEL(1) );
64 }
65
66 void tcSetup(void)
67 {
68   PM->APBCMASK.reg |= (PM_APBCMASK_TC3);     //enable TC3 in clock power manager
69   //attach clock to TC3
70   GCLK->CLKCTRL.reg =  ( GCLK_CLKCTRL_ID_TCC2_TC3 | GCLK_CLKCTRL_GEN(RTC_GCLK_ID) | GCLK_CLKCTRL_CLKEN);
       //Use the same Generic Clock as RTC
71   while ( GCLK->STATUS.reg & GCLK_STATUS_SYNCBUSY );
72
73   TC3->COUNT16.CTRLA.reg |= ( TC_CTRLA_RUNSTDBY | TC_CTRLA_MODE_COUNT16 );
74   TC3->COUNT16.EVCTRL.reg |= ( TC_EVCTRL_TCEI | TC_EVCTRL_EVACT_COUNT );
75   TC3->COUNT16.READREQ.reg |= (TC_READREQ_RCONT | TC_READREQ_ADDR(0x10));
76   while (TC3->COUNT16.STATUS.reg & GCLK_STATUS_SYNCBUSY);
77
78   TC3->COUNT16.CTRLA.bit.ENABLE =1;      //dont forget to turn on the counter dummy
79 }
80
81 void eicSetup(void)
82 {
83   PM->APBAMASK.reg |= (PM_APBAMASK_EIC);     //enable EIC APB clock in the respective power manager (A)
84   //Use the same Generic Clock as RTC (since this saves setting up more clocks) to run the interrupt
       controller
85   GCLK->CLKCTRL.reg =  ( GCLK_CLKCTRL_ID_EIC | GCLK_CLKCTRL_GEN(RTC_GCLK_ID) | GCLK_CLKCTRL_CLKEN);
86   while ( GCLK->STATUS.reg & GCLK_STATUS_SYNCBUSY );
87   // Configure External Interrupt Controller
88   EIC->EVCTRL.reg |= EIC_EVCTRL_EXTINTEO7;     //Enable Event trigger when detecting something on External
```

```
    Interrupt port 7
  EIC->CONFIG[0].reg = 0x11111111;              //Detect rising edge

  //EIC must be disabled when changing settings, finish by enabling and waiting for sync to clear
  EIC->CTRL.bit.ENABLE = 1;
  while(EIC->STATUS.bit.SYNCBUSY);

  // Use the PORT rounting registers to configure Pin A7 as a digitial input (pull down) and assign to
    External Interrupt port 7
  PORT->Group[0].DIRCLR.reg |= (1 << 7);
  PORT->Group[0].OUTCLR.reg |= (1 << 7);
  PORT->Group[0].PINCFG[7].reg |= (PORT_PINCFG_PULLEN | PORT_PINCFG_INEN | PORT_PINCFG_PMUXEN ); //Input
    Enable, MUX on, Pull Resistor
  //Pin MUX forwarding is asynchronous and has no delay, as opposed to synchronised forwarding
  PORT->Group[0].PMUX[3].reg |= (PORT_PMUX_PMUXO_A); //PinMux to EXTINT7
}
```

pages/sourcecode/regSetup.cpp

DATABUFFER.H AND .CPP

```
#ifndef DATABUFFER_h
#define DATABUFFER_h

#include <Arduino.h>

#ifndef DEBUG_STREAM
#define DEBUG_STREAM SerialUSB          // Define SerialUSB as debug stream
#define DEBUG_BAUD 9600                 // Define Serial BAUD
#endif

// the following struct is use to store the encoded values of the weather measurements each minute

struct weatherData {
    unsigned int temperature : 12;
    unsigned int pressure : 11;
    unsigned int windSpeed : 9;
    unsigned int windDirection : 8;
    unsigned int humidity : 10;
    unsigned int rainVolume : 6;
    unsigned int status : 8;
};

class dataBuffer {
    public:

    //constructor
    dataBuffer();

    //destructor
    ~dataBuffer();

    /*this function gets called each minute and processes the weather parameters to fit in the 8 byte
     dataSet struct
    tmp = temperature
    bap = atmospheric pressure

    */
    void newDataset(float* tmp,float* bap,float* spd, float* dir, float* hum, float* vol, int errcode);

    void fillBuffer(int position);

    String createLoRaPayload();

    uint8_t* head;

    private:
    weatherData set;
    uint8_t fullBuffer[40];
};
```

```

#endif
```

pages/sourcecode/dataBuffer.h

```
#include "dataBuffer.h"

dataBuffer::dataBuffer()
{
    dataBuffer::head = &dataBuffer::fullBuffer[0];
}

dataBuffer::~dataBuffer(){}

void dataBuffer::newDataset(float* tmp,float* hpa,float* spd, float* dir, float* hum, float* vol, int
     errcode)
{
    uint16_t tmpu = 0;
    float tmpf = 0.0;

    dataBuffer::set.status = 0;

    if(errcode)
    {
        dataBuffer::set.status |= (errcode & 0x03); //or some 2 bits of information
    }


    tmpf = (*tmp);
    if(tmpf <= -50.0){tmpf = -50.0; dataBuffer::set.status |= (1 << 7);}
    if(tmpf >= 70.0){tmpf = 70.0;dataBuffer::set.status |= (1 << 7);}
    tmpu =(unsigned int) ((tmpf + 50.0)*((1<<12)-1) / 120);
    dataBuffer::set.temperature = tmpu;
//    DEBUG_STREAM.print("temperature encoded to:");
//    DEBUG_STREAM.println(tmpu,BIN);

    tmpf = *hpa;
    if(tmpf <= 900.0){tmpf = 900;dataBuffer::set.status |= (1 << 6);}
    if(tmpf >= 1100.0){tmpf = 1100;dataBuffer::set.status |= (1 << 6);}
    tmpu =(unsigned int) ((tmpf - 900)*((1<<11)-1) / 200);
    dataBuffer::set.pressure = tmpu;
//    DEBUG_STREAM.print("pressure encoded to:");
//    DEBUG_STREAM.println(tmpu,BIN);

    tmpf = *spd;
    if(tmpf <= 0.0){tmpf = 0.0;dataBuffer::set.status |= (1 << 5);}
    if(tmpf >= 25.0){tmpf = 25.0;dataBuffer::set.status |= (1 << 5);}
    tmpu =(unsigned int) ((tmpf)*((1<<9)-1) / 25.0);
    dataBuffer::set.windSpeed = tmpu;
//    DEBUG_STREAM.print("wind spd encoded to:");
//    DEBUG_STREAM.println(tmpu,BIN);

    tmpf = *dir;
    if(tmpf <= 0.0){tmpf = 0.0;dataBuffer::set.status |= (1 << 4);}
    if(tmpf >= 360.0){tmpf = 360.0;dataBuffer::set.status |= (1 << 4);}
    tmpu =(unsigned int) ((tmpf)*((1<<8)-1) / 360);
    dataBuffer::set.windDirection = tmpu;
//    DEBUG_STREAM.print("wind dir encoded to:");
//    DEBUG_STREAM.println(tmpu,BIN);

    tmpf = *hum;
    if(tmpf <= 0.0){tmpf = 0.0;dataBuffer::set.status |= (1 << 3);}
    if(tmpf >= 100.0){tmpf = 100.0;dataBuffer::set.status |= (1 << 3);}
    tmpu =(unsigned int) ((tmpf)*((1<<10)-1) / 100.0);
    dataBuffer::set.humidity = tmpu;
//    DEBUG_STREAM.print("humidity encoded to:");
//    DEBUG_STREAM.println(tmpu,BIN);

    tmpf = *vol;
    if(tmpf <= 0.0){tmpf = 0.0;dataBuffer::set.status |= (1 << 2);}
```

```
65      if(tmpf >= 12.6){tmpf = 12.6;dataBuffer::set.status |= (1 << 2);}
66      tmpu =(unsigned int) ((tmpf)*((1<<6)-1) / 12.6);
67      dataBuffer::set.rainVolume = tmpu;
68  //    DEBUG_STREAM.print("rain volume encoded to:");
69  //    DEBUG_STREAM.println(tmpu,BIN);
70
71  }
72
73  void dataBuffer::fillBuffer(int position)
74  {
75      int offset;
76      offset = 8*position;
77
78      dataBuffer::fullBuffer[offset] = ((dataBuffer::set.temperature & 0xFF0) >> 4);
79      dataBuffer::fullBuffer[offset+1] = (((dataBuffer::set.temperature & 0x00F) << 4) | ((dataBuffer::set.
        pressure & 0x780) >> 7));
80      dataBuffer::fullBuffer[offset+2] = (((dataBuffer::set.pressure & 0x07F) << 1 ) | ((dataBuffer::set.
        windSpeed & 0x100) >> 8 ));
81      dataBuffer::fullBuffer[offset+3] = (dataBuffer::set.windSpeed & 0x0FF);
82      dataBuffer::fullBuffer[offset+4] = (dataBuffer::set.windDirection & 0xFF);
83      dataBuffer::fullBuffer[offset+5] = ((dataBuffer::set.humidity & 0x3FC) >> 2);
84      dataBuffer::fullBuffer[offset+6] = (((dataBuffer::set.humidity & 0x003) << 6) | (dataBuffer::set.
        rainVolume & 0x3F));
85      dataBuffer::fullBuffer[offset+7] = (dataBuffer::set.status);
86  }
87
88  String dataBuffer::createLoRaPayload()
89  {
90      String payload;
91      payload = (char*) dataBuffer::fullBuffer;
92      DEBUG_STREAM.println("Lora Payload created");
93      for(unsigned int i = 0;i < sizeof(fullBuffer); i++)
94      {
95          if(i == 8 || i == 16 || i == 24 || i == 32) DEBUG_STREAM.println("");
96          DEBUG_STREAM.print(fullBuffer[i],HEX);
97      }
98      DEBUG_STREAM.println("");
99      return payload;
100 }
```

pages/sourcecode/dataBuffer.cpp

LORAFUNC.H AND .CPP

```
1  #ifndef LORAFUNC_h
2  #define LORAFUNC_h
3  #include <Arduino.h>                    // Include Arduino Library
4
5  #define DEBUG_STREAM SerialUSB          // Define SerialUSB as debug stream
6  #define LORA_STREAM Serial1             // Define Serial1 as LoRa stream
7  #define DEBUG_BAUD 9600                 // Define Serial BAUD
8
9  const uint8_t devAddr[4]  = { 0x14, 0x20, 0x39, 0x1E }; // Define LoRa ABP Adresses and Keys
10 const uint8_t appSKey[16] = { 0xb3, 0xb3, 0xc1, 0xc0, 0xa7, 0xbd, 0x12, 0x83, 0x1d, 0xe2, 0xe7, 0x16, 0x17
       , 0x69, 0xea, 0xbe };
11 const uint8_t nwkSKey[16] = { 0xfd, 0x3e, 0xf5, 0x8e, 0x3c, 0x44, 0xb7, 0x5d, 0x46, 0xbf, 0xa1, 0x18, 0x2a
       , 0xda, 0x5c, 0x4c };
12
13 const uint8_t DevEUI[8]  = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}; // Define LoRa OTAA Adresses
       and Keys
14 const uint8_t AppEUI[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
15 const uint8_t AppKey2[16] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
       , 0x00, 0x00, 0x00 };
16
17
18 void setupLoRaABP(void);
19 void sendPacket(String packet);
20 #endif
```

pages/sourcecode/LoRaFunc.h

```
#include <Sodaq_RN2483.h>                // Include LoRa Module Library
#include "LoRaFunc.h"
#include "LedColor.h"


void setupLoRaABP(void)                        // Function to connect to LoRaWAN (KPN) via ABP
{
  //DEBUG_STREAM.println("Setting up communication with LoRa Module");
  if (LoRaBee.initABP(LORA_STREAM, devAddr, appSKey, nwkSKey, false))
  {
    //DEBUG_STREAM.println("Communication with LoRa Module successful.");
    delay(500);
    setLedColor(GREEN);
    delay(500);
    setLedColor(NONE);
  }
  else
  {
    //DEBUG_STREAM.println("Communication with LoRa Module failed!");
    delay(500);
    setLedColor(RED);
    delay(500);
    setLedColor(NONE);
  }
}

void sendPacket(String packet)                                  // Function to send LoRa packet
{
  switch (LoRaBee.send(1, (uint8_t*)packet.c_str(), packet.length()))
  {
    case NoError:
      DEBUG_STREAM.println("Successful transmission.");
      break;
    case NoResponse:
      DEBUG_STREAM.println("There was no response from the device.");
      setupLoRaABP();
      break;
    case Timeout:
      DEBUG_STREAM.println("Connection timed-out. Check your serial connection to the device! Sleeping for
      20sec.");
      delay(20000);
      break;
    case PayloadSizeError:
      DEBUG_STREAM.println("The size of the payload is greater than allowed. Transmission failed!");
      break;
    case InternalError:
      DEBUG_STREAM.println("Oh No! This shouldn't happen. Something is really wrong! Try restarting the
     device!\r\nThe network connection will reset.");
      setupLoRaABP();
      break;
    case Busy:
      DEBUG_STREAM.println("The device is busy. Sleeping for 10 extra seconds.");
      delay(10000);
      break;
    case NetworkFatalError:
      DEBUG_STREAM.println("There is a non-recoverable error with the network connection. You should re-
     connect.\r\nThe network connection will reset.");
      setupLoRaABP();
      break;
    case NotConnected:
      DEBUG_STREAM.println("The device is not connected to the network. Please connect to the network
     before attempting to send data.\r\nThe network connection will reset.");
      setupLoRaABP();
      break;
    case NoAcknowledgment:
      DEBUG_STREAM.println("There was no acknowledgment sent back!");
      break;
    default:
      break;
  }
```

```
}
```

pages/sourcecode/LoRaFunc.cpp

BMP180.H AND .CPP

```
#ifndef BMP180_H
#define BMP180_H

/* BMP180 Digital Pressure Sensor Class for use with Mbed LPC1768 and other platforms
*  BMP180 from Bosch Sensortec
*  Copyright (c) 2013 Philip King Smith
*
* Permission is hereby granted, free of charge, to any person obtaining a copy
* of this software and associated documentation files (the "Software"), to deal
* in the Software without restriction, including without limitation the rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included in
* all copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHEITHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
* THE SOFTWARE.
*
* Bosch data sheet at: http://ae-bst.resource.bosch.com/media/products/dokumente/bmp180/BST-BMP180-DS000-09.pdf
* Some parts of the calculations used are infact from Rom Clement published Mbed code here: https://mbed.org/users/Rom/code/Barometer_bmp085/
*   I only used snippets of the Rom's code because i was making this into a class and so this is structured totaly different then his code example is.
* I also used the Bosch data sheet showing the calculations and adjusted everything accordingly!
*/

#include <Arduino.h>
#include <Wire.h>
#include <Sodaq_wdt.h>

#define EEprom 22             // The EEPROM has 176bits of calibration data (176/8 = 22 Bytes)
#define BMP180ADDR 0x77       // I2C address of BMP180 device
#define BMP180FREQ 1000000    // Data sheet says 3.4 MHz is max but not sure what mbed can do here!
#define CMD_READ_VALUE 0xF6
#define CMD_READ_CALIBRATION 0xAA
#define CMD_RESET_VALUE 0xB6
#define CMD_RESET_REG 0xE0
#define OVERSAMPLING_ULTRA_LOW_POWER 0  // these are the constants used in the oversample variable in the below code!
#define OVERSAMPLING_STANDARD 1
#define OVERSAMPLING_HIGH_RESOLUTION 2
#define OVERSAMPLING_ULTRA_HIGH_RESOLUTION 3

#ifndef DEBUG_STREAM
#define DEBUG_STREAM SerialUSB          // Define SerialUSB as debug stream
#define DEBUG_BAUD 9600                 // Define Serial BAUD
#endif

/** BMP180 Digital Pressure Sensor class using mbed's i2c class
 *
 * Example:
 * @code
 * // show how the BMP180 class works
 * #include "mbed.h"
 * #include "BMP180.h"
 *
```

```
61  * int main()
62  * {
63  *
64  *     long temp ;
65  *     long pressure;
66  *     int error ;
67  *     BMP180 mybmp180(p9,p10);
68  *     while(1) {
69  *         error = mybmp180.readTP(&temp,&pressure,OVERSAMPLING_ULTRA_HIGH_RESOLUTION);
70  *         printf("Temp is %ld\r\n",temp);
71  *         printf("Pressure is %ld\r\n",pressure);
72  *         printf("Error is %d\r\n\r\n",error);
73  *         wait(2);
74  *     }
75  * }
76  * @endcode
77  */
78
79  class BMP180
80  {
81  public:
82      /** Create object connected to BMP180 pins ( remember both pins need pull up resisters)
83          *
84          * Ensure the pull up resistors are used on these pins.  Also note there is no checking on
85          *  if you use these pins p9, p10, p27, p28 so ensure you only use these ones on the LPC1768 device
86          *
87          * @param sda pin that BMP180 connected to (p9 or p28 as defined on LPC1768)
88          * @param slc pin that BMP180 connected to (p10 or p27 ad defined on LPC1768)
89          */
90      BMP180();   // Constructor
91
92      ~BMP180();  // Destructor
93
94      int begin(void);    //Initialise communications and receive calibration data from EEPROM
95
96      /** Read Temperature and Pressure at the same time
97      *
98      * This function will only return when it has readings.  This means that the time will vary depending
        on oversample setting!
99      *   Note if your code can not wait for these readings to be taken and calculated you should use the
        functions below.
100     *      These other functions are designed to allow your code to do other things then get the final
        readings.
101     *      This function is only designed as a one shot give me the answer function.
102     *
103     * @param t the temperature fully compensated value is returned in this variable. Degrees celsius with
        one decimal so 253 is 25.3 C.
104     * @param p the barometric pressure fully compensated value is returned in this variable. Pressure is
        in Pa so 88007 is 88.007 kPa.
105     * @param oversample is the method method for reading sensor.  OVERSAMPLING_ULTRA_HIGH_RESOLUTION is
        used if an incorrect value is passed to this function.
106     * @param returns 0 for no errors during i2c communication.  Any other number is just a i2c
        communication failure of some kind!
107     */
108     uint16_t readTP(long *t, long *p, int oversample);     // get both temperature and pressure fully
        compensated values! Note this only returns when measurements are complete
109
110     /** Start the temperature reading process but return after the commands are issued to BMP180
111     *
112     * This function is ment to start the temperature reading process but will return to allow other code
        to run then a reading could be made at a later time.
113     *   Note the maximum time needed for this measurment is 4.5 ms.
114     *
115     * @param returns 0 for no errors during i2c communication.  Any other number is just a i2c
        communication failure of some kind!
116     */
117     uint8_t startTemperature();             // Start temperature measurement
118
119     /** Reads the last temperature reading that was started with startTemperature() function
120     *
```

```
121     * This function will return the fully compensated value of the temperature in Degrees celsius with one
          decimal so 253 is 25.3 C.
122     *    Note this function should normaly follow the startTemperature() function and should also preceed
         the startPressure() and readPressure() commands!
123     *    Note this function should follow startTemperature() after 4.5 ms minimum has elapsed or reading
         will be incorrect.
124     *
125     * @param t the temperature fully compensated value is returned in this variable.
126     * @param returns 0 for no errors during i2c communication.  Any other number is just a i2c
         communication failure of some kind!
127     */
128     uint8_t readTemperature(long *t);        // Get the temperature reading that was taken in
         startTemperature() but ensure 4.5 ms time has elapsed
129
130     /** Start the pressure reading process but return after the commands are issued to BMP180
131     *
132     * This function is ment to start the pressure reading process but will return to allow other code to
         run then a reading could be made at a later time.
133     *    Note the time needed for this reading pressure process will depend on oversample setting.  The
         maximum time is 25.5 ms and minimum time is 4.5 ms.
134     *
135     * @param oversample is the method for reading sensor.  OVERSAMPLING_ULTRA_HIGH_RESOLUTION is used if
         an incorrect value is passed to this function.
136     * @param returns 0 for no errors during i2c communication.  Any other number is just a i2c
         communication failure of some kind!
137     */
138     uint8_t startPressure(int oversample);  // Start pressure measurement!  Note oversample will vary the
         time to complete this measurement. See defines above for oversampling constants to use!
139
140     /** Reads the last barometric pressure reading that was started with startPressure() function
141     *
142     * This function will return the fully compensated value of the barometric pressure in Pa.
143     *    Note this function should follow startPressure() after the time needed to read the pressure.  This
          time will vary but maximum time is 25.5 ms and minimum time is 4.5 ms.
144     *    Note that this reading is dependent on temperature so the startTemperature() and readTemperature()
          functions should proceed this function or the pressure value will be incorrect!
145     *
146     * @param p the barometric pressure fully compensated value is returned in this variable. Pressure is
         in Pa so 88007 is 88.007 kPa.
147     * @param returns 0 for no errors during i2c communication.  Any other number is just a i2c
         communication failure of some kind!
148     */
149     uint8_t readPressure(long *p);          // Get the pressure reading that was taken in startPressure()
         but ensure time for the measurement to complete
150
151     uint8_t softReset();
152
153 protected:
154     long x1;
155     long x2;
156     long x3;
157     short ac1;
158     short ac2;
159     short ac3;
160     unsigned short ac4;
161     unsigned short ac5;
162     unsigned short ac6;
163     short b1;
164     short b2;
165     long b3;
166     unsigned long b4;
167     long b5;
168     long b6;
169     unsigned long b7;
170     short mb;
171     short mc;
172     short md;
173     int oversampling_setting;
174     char rReg[3];
175     char wReg[2];
176     char cmd;
```

```
177     char data[EEprom];
178     char w[2];
179 
180     int oversampleCheck(int oversample);
181 };
182 
183 #endif
```

pages/sourcecode/BMP180.h

```
1  #include "BMP180.h"
2  
3  BMP180::BMP180()
4  {
5  }
6  
7  BMP180::~BMP180()
8  {
9  }
10 
11 int BMP180::begin()
12 {
13 // Wire.begin();
14     oversampling_setting = OVERSAMPLING_HIGH_RESOLUTION;
15     rReg[0] = 0;
16     rReg[1] = 0;
17     rReg[2] = 0;
18     wReg[0] = 0;
19     wReg[1] = 0;
20     w[0] = 0xF4;
21     w[1] = 0xF4;
22 
23     cmd = CMD_READ_CALIBRATION; // EEPROM calibration command
24         Wire.beginTransmission(BMP180ADDR);       //check device = ready
25         Wire.write(0xD0);
26         Wire.endTransmission();
27         Wire.beginTransmission(BMP180ADDR);
28         Wire.requestFrom(BMP180ADDR, 1);
29         rReg[0] = Wire.read();
30         Wire.endTransmission();
31     if(rReg[0] != 0x55)  return 1;
32     rReg[0] = 0;
33 
34     for (int i = 0; i < EEprom; i++) // read the 22 registers of the EEPROM
35     {
36         Wire.beginTransmission(BMP180ADDR);       // start transmission to device
37         Wire.write(cmd);                              // sends register address to read from
38         Wire.endTransmission();                       // end transmission
39 
40         Wire.beginTransmission(BMP180ADDR);       // start transmission to device
41         Wire.requestFrom(BMP180ADDR, 1);          // send data n-bytes read
42         data[i] = Wire.read();
43         Wire.endTransmission();                       // end transmission
44         cmd += 1;
45     }
46 
47     // parameters AC1-AC6
48     //The calibration is partioned in 11 words of 16 bits, each of them representing a coefficient
49     ac1 =  (data[0] <<8) | data[1]; // AC1(0xAA, 0xAB)... and so on
50     ac2 =  (data[2] <<8) | data[3];
51     ac3 =  (data[4] <<8) | data[5];
52     ac4 =  (data[6] <<8) | data[7];
53     ac5 =  (data[8] <<8) | data[9];
54     ac6 = (data[10] <<8) | data[11];
55     // parameters B1,B2
56     b1 =  (data[12] <<8) | data[13];
57     b2 =  (data[14] <<8) | data[15];
58     // parameters MB,MC,MD
59     mb =  (data[16] <<8) | data[17];
60     mc =  (data[18] <<8) | data[19];
```

```
61      md =  (data[20] <<8) | data[21];
62
63      return 0;
64  }
65
66  uint8_t BMP180::startTemperature()              // Start temperature measurement
67  {
68      uint8_t error = 0;
69      wReg[0] = 0xF4;
70      wReg[1] = 0x2E;
71      Wire.beginTransmission(BMP180ADDR);       // start transmission to device
72      Wire.write(wReg, 2);                      // transmit to conversion control (0xF4) the instruction to
         start T measurement (0x2E)
73      error |= Wire.endTransmission();          // end transmission
74      return error;
75  }
76
77
78  uint8_t BMP180::readTemperature(long *t)        // Get the temperature reading that was taken in
         startTemperature() but ensure 4.5 ms time has elapsed
79  {
80      uint8_t error = 0;
81      rReg[0] = 0;
82      rReg[1] = 0;
83      cmd = CMD_READ_VALUE;
84      Wire.beginTransmission(BMP180ADDR);       // start transmission to device
85      Wire.write(cmd);                          // set pointer on 0xF6 before reading it
86      error |= Wire.endTransmission();
87      error <<= 2;
88
89      Wire.beginTransmission(BMP180ADDR);        // start transmission to device
90      Wire.requestFrom(BMP180ADDR, 2);           // read 0xF6 (MSB) and 0xF7 (LSB)
91      rReg[0] = Wire.read();
92      rReg[1] = Wire.read();
93      error |= Wire.endTransmission();            // end transmission
94
95          *t = (rReg[0] << 8) | rReg[1];                  // UT = MSB << 8 + LSB
96
97      x1 = (((long) *t - (long) ac6) * (long) ac5) >> 15; // aka (ut-ac6) * ac5/pow(2,15)
98      x2 = ((long) mc << 11) / (x1 + md);                 // aka mc * pow(2, 11) / (x1 + md)
99      b5 = x1 + x2;
100     *t = ((b5 + 8) >> 4);                               // (b5+8)/pow(2, 4) => Temperature in 0.1C
101     return error;
102 }
103
104 uint8_t BMP180::startPressure(int oversample)  // Start pressure measurement!  Note oversample will vary
     the time to complete this measurement. See defines above for oversampling constants to use!
105 {
106     uint8_t error = 0;
107     int uncomp_pressure_cmd;
108     oversampling_setting = BMP180::oversampleCheck(oversample);
109     uncomp_pressure_cmd = 0x34 + (oversampling_setting<<6);
110     wReg[0] = 0xF4;
111     wReg[1] = uncomp_pressure_cmd;
112     Wire.beginTransmission(BMP180ADDR);       // start transmission to device
113     Wire.write(wReg, 2);                      // transmit to conversion control (0xF4) the instruction to
         start P measurement w oversampling
114     error |= Wire.endTransmission();          // end transmission
115     return error;
116 }
117
118 uint8_t BMP180::readPressure(long *p)   // Get the pressure reading that was taken in startPressure() but
     ensure time for the measurement to complete
119 {
120     uint8_t error = 0;
121     rReg[0] = 0;
122     rReg[1] = 0;
123     cmd = CMD_READ_VALUE;
124     Wire.beginTransmission(BMP180ADDR);       // start transmission to device
125     Wire.write(cmd);                          // set pointer on 0xF6 before reading it
126     error |= Wire.endTransmission();
```

```
127     error <<= 2;
128
129     Wire.beginTransmission(BMP180ADDR);        // start transmission to device
130     Wire.requestFrom(BMP180ADDR, 2);           // read 0xF6 (MSB) to 0xF8 (LSB)
131     rReg[0] = Wire.read();
132     rReg[1] = Wire.read();
133     rReg[2] = Wire.read();
134     error |= Wire.endTransmission();           // end transmission
135
136     *p = ((rReg[0] << 16) | (rReg[1] << 8) | rReg[2]) >> (8 - oversampling_setting);
137
138     b6 = b5 - 4000;             // realise b5 is set in readTemperature() function so that needs to be
        done first before this function!
139     x1 = (b6*b6) >> 12;         // full formula(b2*(b6*b6)/pow(2,12))/pow(2,11)
140     x1 *= b2;
141     x1 >>= 11;
142
143     x2 = (ac2*b6);
144     x2 >>= 11;
145
146     x3 = x1 + x2;
147
148     b3 = (((((long)ac1 )*4 + x3) <<oversampling_setting) + 2) >> 2;
149
150     x1 = (ac3* b6) >> 13;
151     x2 = (b1 * ((b6*b6) >> 12) ) >> 16;
152     x3 = ((x1 + x2) + 2) >> 2;
153     b4 = (ac4 * (unsigned long) (x3 + 32768)) >> 15;
154
155     b7 = ((unsigned long) *p - b3) * (50000>>oversampling_setting);
156     if (b7 < 0x80000000) {
157         *p = (b7 << 1) / b4;
158     } else {
159         *p = (b7 / b4) << 1;
160     }
161
162     x1 = *p >> 8;
163     x1 *= x1;                                  // pressure/pow(2,8) * pressure/pow(2, 8)
164     x1 = (x1 * 3038) >> 16;
165     x2 = ( *p * -7357) >> 16;
166     *p += (x1 + x2 + 3791) >> 4;               // pressure in Pa
167     return error;
168 }
169
170
171 uint16_t BMP180::readTP(long *t, long *p, int oversample)  // get both temperature and pressure
        calculations that are compensated
172 {
173     uint16_t errors = 0;
174     errors |= BMP180::startTemperature();
175     sodaq_wdt_safe_delay(4.5);
176     errors <<= 4;
177     errors |= BMP180::readTemperature(t);
178     errors <<= 4;
179     errors |= BMP180::startPressure(oversample);
180     errors <<= 4;
181     switch (oversample) {
182         case OVERSAMPLING_ULTRA_LOW_POWER:
183             sodaq_wdt_safe_delay(5);
184             break;
185         case OVERSAMPLING_STANDARD:
186             sodaq_wdt_safe_delay(85);
187             break;
188         case OVERSAMPLING_HIGH_RESOLUTION:
189             sodaq_wdt_safe_delay(14);
190             break;
191         case OVERSAMPLING_ULTRA_HIGH_RESOLUTION:
192             sodaq_wdt_safe_delay(26);
193             break;
194     }
195     errors |= BMP180::readPressure(p);
```

```
196     errors <<= 4;
197     return(errors);
198 }
199
200 int BMP180::oversampleCheck(int oversample)
201 {
202     switch(oversample) {
203         case OVERSAMPLING_ULTRA_LOW_POWER:
204             break;
205         case OVERSAMPLING_STANDARD:
206             break;
207         case OVERSAMPLING_HIGH_RESOLUTION:
208             break;
209         case OVERSAMPLING_ULTRA_HIGH_RESOLUTION:
210             break;
211         default:
212             oversample = OVERSAMPLING_ULTRA_HIGH_RESOLUTION;
213             break;
214     }
215     return(oversample);
216 }
217
218 uint8_t BMP180::softReset()
219 {
220     uint8_t error = 0;
221     wReg[0] = CMD_RESET_REG;
222     wReg[1] = CMD_RESET_VALUE;
223     Wire.beginTransmission(BMP180ADDR);        // start transmission to device
224     Wire.write(wReg,2);                        // set pointer on 0xE0, write 0xB6 to initiate a soft Reset
     to fix repeat errors
225     error |= Wire.endTransmission();
226     return error;
227 }
```

pages/sourcecode/BMP180.cpp

SHT30.H AND .CPP

```
1  /*
2
3  Arduino Library for Sensirion SHT3X-DIS Digital Humidity & Temperature Sensors
4  Written by AA
5  ---
6
7  The MIT License (MIT)
8
9  Copyright (c) 2015-2017 ClosedCube Limited
10
11 Permission is hereby granted, free of charge, to any person obtaining a copy
12 of this software and associated documentation files (the "Software"), to deal
13 in the Software without restriction, including without limitation the rights
14 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
15 copies of the Software, and to permit persons to whom the Software is
16 furnished to do so, subject to the following conditions:
17
18 The above copyright notice and this permission notice shall be included in
19 all copies or substantial portions of the Software.
20
21 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
22 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
23 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
24 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
25 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
26 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
27 THE SOFTWARE.
28
29 */
30
31 #ifndef CLOSEDCUBE_SHT31D
32 #define CLOSEDCUBE_SHT31D
```

```
33
34 /* SHT30 Humidity and Temperature Sensor Class
35 *
36 * for use with Atmel SAM D MCU's such as the SODAQ ONE
37 * SHT30 from Sensirion
38 * Library built from Philip King Smith's BMP280 Library
39 */
40
41 #include <Arduino.h>
42
43 typedef enum {
44     SHT3XD_CMD_READ_SERIAL_NUMBER = 0x3780,
45
46     SHT3XD_CMD_READ_STATUS = 0xF32D,
47     SHT3XD_CMD_CLEAR_STATUS = 0x3041,
48
49     SHT3XD_CMD_HEATER_ENABLE = 0x306D,
50     SHT3XD_CMD_HEATER_DISABLE = 0x3066,
51
52     SHT3XD_CMD_SOFT_RESET = 0x30A2,
53
54     SHT3XD_CMD_CLOCK_STRETCH_H = 0x2C06,
55     SHT3XD_CMD_CLOCK_STRETCH_M = 0x2C0D,
56     SHT3XD_CMD_CLOCK_STRETCH_L = 0x2C10,
57
58     SHT3XD_CMD_POLLING_H = 0x2400,
59     SHT3XD_CMD_POLLING_M = 0x240B,
60     SHT3XD_CMD_POLLING_L = 0x2416,
61
62     SHT3XD_CMD_ART = 0x2B32,
63
64     SHT3XD_CMD_PERIODIC_HALF_H = 0x2032,
65     SHT3XD_CMD_PERIODIC_HALF_M = 0x2024,
66     SHT3XD_CMD_PERIODIC_HALF_L = 0x202F,
67     SHT3XD_CMD_PERIODIC_1_H = 0x2130,
68     SHT3XD_CMD_PERIODIC_1_M = 0x2126,
69     SHT3XD_CMD_PERIODIC_1_L = 0x212D,
70     SHT3XD_CMD_PERIODIC_2_H = 0x2236,
71     SHT3XD_CMD_PERIODIC_2_M = 0x2220,
72     SHT3XD_CMD_PERIODIC_2_L = 0x222B,
73     SHT3XD_CMD_PERIODIC_4_H = 0x2334,
74     SHT3XD_CMD_PERIODIC_4_M = 0x2322,
75     SHT3XD_CMD_PERIODIC_4_L = 0x2329,
76     SHT3XD_CMD_PERIODIC_10_H = 0x2737,
77     SHT3XD_CMD_PERIODIC_10_M = 0x2721,
78     SHT3XD_CMD_PERIODIC_10_L = 0x272A,
79
80     SHT3XD_CMD_FETCH_DATA = 0xE000,
81     SHT3XD_CMD_STOP_PERIODIC = 0x3093,
82
83     SHT3XD_CMD_READ_ALR_LIMIT_LS = 0xE102,
84     SHT3XD_CMD_READ_ALR_LIMIT_LC = 0xE109,
85     SHT3XD_CMD_READ_ALR_LIMIT_HS = 0xE11F,
86     SHT3XD_CMD_READ_ALR_LIMIT_HC = 0xE114,
87
88     SHT3XD_CMD_WRITE_ALR_LIMIT_HS = 0x611D,
89     SHT3XD_CMD_WRITE_ALR_LIMIT_HC = 0x6116,
90     SHT3XD_CMD_WRITE_ALR_LIMIT_LC = 0x610B,
91     SHT3XD_CMD_WRITE_ALR_LIMIT_LS = 0x6100,
92
93     SHT3XD_CMD_NO_SLEEP = 0x303E,
94 } SHT31D_Commands;
95
96
97 typedef enum {
98     SHT3XD_REPEATABILITY_HIGH,
99     SHT3XD_REPEATABILITY_MEDIUM,
100    SHT3XD_REPEATABILITY_LOW,
101 } SHT31D_Repeatability;
102
103 typedef enum {
```

```
104     SHT3XD_MODE_CLOCK_STRETCH,
105     SHT3XD_MODE_POLLING,
106 } SHT31D_Mode;
107
108 typedef enum {
109     SHT3XD_FREQUENCY_HZ5,
110     SHT3XD_FREQUENCY_1HZ,
111     SHT3XD_FREQUENCY_2HZ,
112     SHT3XD_FREQUENCY_4HZ,
113     SHT3XD_FREQUENCY_10HZ
114 } SHT31D_Frequency;
115
116 typedef enum {
117     SHT3XD_NO_ERROR = 0,
118
119     SHT3XD_CRC_ERROR = -101,
120     SHT3XD_TIMEOUT_ERROR = -102,
121
122     SHT3XD_PARAM_WRONG_MODE = -501,
123     SHT3XD_PARAM_WRONG_REPEATABILITY = -502,
124     SHT3XD_PARAM_WRONG_FREQUENCY = -503,
125     SHT3XD_PARAM_WRONG_ALERT = -504,
126
127     // Wire I2C translated error codes
128     SHT3XD_WIRE_I2C_DATA_TOO_LOG = -10,
129     SHT3XD_WIRE_I2C_RECEIVED_NACK_ON_ADDRESS = -20,
130     SHT3XD_WIRE_I2C_RECEIVED_NACK_ON_DATA = -30,
131     SHT3XD_WIRE_I2C_UNKNOW_ERROR = -40
132 } SHT31D_ErrorCode;
133
134 typedef union {
135     uint16_t rawData;
136     struct {
137         uint8_t WriteDataChecksumStatus : 1;
138         uint8_t CommandStatus : 1;
139         uint8_t Reserved0 : 2;
140         uint8_t SystemResetDetected : 1;
141         uint8_t Reserved1 : 5;
142         uint8_t T_TrackingAlert : 1;
143         uint8_t RH_TrackingAlert : 1;
144         uint8_t Reserved2 : 1;
145         uint8_t HeaterStatus : 1;
146         uint8_t Reserved3 : 1;
147         uint8_t AlertPending : 1;
148     };
149 } SHT31D_RegisterStatus;
150
151 struct SHT31D {
152     float t;
153     float rh;
154     SHT31D_ErrorCode error;
155 };
156
157 class ClosedCube_SHT31D {
158 public:
159     ClosedCube_SHT31D();
160
161     SHT31D_ErrorCode begin(uint8_t address);
162     SHT31D_ErrorCode clearAll();
163     SHT31D_RegisterStatus readStatusRegister();
164
165     SHT31D_ErrorCode heaterEnable();
166     SHT31D_ErrorCode heaterDisable();
167
168     SHT31D_ErrorCode softReset();
169     SHT31D_ErrorCode reset(); // same as softReset
170
171     SHT31D_ErrorCode generalCallReset();
172
173     SHT31D_ErrorCode artEnable();
174
```

```
175     uint32_t readSerialNumber();
176
177     SHT31D readTempAndHumidity(SHT31D_Repeatability repeatability, SHT31D_Mode mode, uint8_t timeout);
178     SHT31D readTempAndHumidityClockStretch(SHT31D_Repeatability repeatability);
179     SHT31D readTempAndHumidityPolling(SHT31D_Repeatability repeatability, uint8_t timeout);
180
181     SHT31D_ErrorCode periodicStart(SHT31D_Repeatability repeatability, SHT31D_Frequency frequency);
182     SHT31D periodicFetchData();
183     SHT31D_ErrorCode periodicStop();
184
185     SHT31D_ErrorCode writeAlertHigh(float temperatureSet, float temperatureClear, float humiditySet, float
          humidityClear);
186     SHT31D readAlertHighSet();
187     SHT31D readAlertHighClear();
188
189     SHT31D_ErrorCode writeAlertLow(float temperatureClear, float temperatureSet, float humidityClear, float
          humiditySet);
190     SHT31D readAlertLowSet();
191     SHT31D readAlertLowClear();
192
193
194 private:
195     uint8_t _address;
196     SHT31D_RegisterStatus _status;
197
198     SHT31D_ErrorCode writeCommand(SHT31D_Commands command);
199     SHT31D_ErrorCode writeAlertData(SHT31D_Commands command, float temperature, float humidity);
200
201     uint8_t checkCrc(uint8_t data[], uint8_t checksum);
202     uint8_t calculateCrc(uint8_t data[]);
203
204     float calculateHumidity(uint16_t rawValue);
205     float calculateTemperature(uint16_t rawValue);
206
207     uint16_t calculateRawHumidity(float value);
208     uint16_t calculateRawTemperature(float value);
209
210     SHT31D readTemperatureAndHumidity();
211     SHT31D readAlertData(SHT31D_Commands command);
212     SHT31D_ErrorCode read(uint16_t* data, uint8_t numOfPair);
213
214     SHT31D returnError(SHT31D_ErrorCode command);
215 };
216
217
218 #endif
```

pages/sourcecode/SHT30.h

```
1  /*
2
3  Arduino Library for Sensirion SHT3X-DIS Digital Humidity & Temperature Sensors
4  Written by AA
5  ---
6
7  The MIT License (MIT)
8
9  Copyright (c) 2015-2017 ClosedCube Limited
10
11 Permission is hereby granted, free of charge, to any person obtaining a copy
12 of this software and associated documentation files (the "Software"), to deal
13 in the Software without restriction, including without limitation the rights
14 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
15 copies of the Software, and to permit persons to whom the Software is
16 furnished to do so, subject to the following conditions:
17
18 The above copyright notice and this permission notice shall be included in
19 all copies or substantial portions of the Software.
20
21 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
```

```
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.

*/
#include <Wire.h>

#include "SHT30.h"

ClosedCube_SHT31D::ClosedCube_SHT31D()
{
}

SHT31D_ErrorCode ClosedCube_SHT31D::begin(uint8_t address) {
    SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
    _address = address;
    return error;
}

SHT31D_ErrorCode ClosedCube_SHT31D::reset()
{
    return  softReset();
}

SHT31D ClosedCube_SHT31D::periodicFetchData()
{
    SHT31D_ErrorCode error = writeCommand(SHT3XD_CMD_FETCH_DATA);
    if (error == SHT3XD_NO_ERROR)
        return readTemperatureAndHumidity();
    else
        returnError(error);
}

SHT31D_ErrorCode ClosedCube_SHT31D::periodicStop() {
    return writeCommand(SHT3XD_CMD_STOP_PERIODIC);
}

SHT31D_ErrorCode ClosedCube_SHT31D::periodicStart(SHT31D_Repeatability repeatability, SHT31D_Frequency
    frequency)
{
    SHT31D_ErrorCode error;

    switch (repeatability)
    {
    case SHT3XD_REPEATABILITY_LOW:
        switch (frequency)
        {
        case SHT3XD_FREQUENCY_HZ5:
            error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_L);
            break;
        case SHT3XD_FREQUENCY_1HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_1_L);
            break;
        case SHT3XD_FREQUENCY_2HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_2_L);
            break;
        case SHT3XD_FREQUENCY_4HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_4_L);
            break;
        case SHT3XD_FREQUENCY_10HZ:
            error = writeCommand(SHT3XD_CMD_PERIODIC_10_L);
            break;
        default:
            error = SHT3XD_PARAM_WRONG_FREQUENCY;
            break;
        }
        break;
    case SHT3XD_REPEATABILITY_MEDIUM:
```

```
92          switch (frequency)
93          {
94          case SHT3XD_FREQUENCY_HZ5:
95              error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_M);
96              break;
97          case SHT3XD_FREQUENCY_1HZ:
98              error = writeCommand(SHT3XD_CMD_PERIODIC_1_M);
99              break;
100         case SHT3XD_FREQUENCY_2HZ:
101             error = writeCommand(SHT3XD_CMD_PERIODIC_2_M);
102             break;
103         case SHT3XD_FREQUENCY_4HZ:
104             error = writeCommand(SHT3XD_CMD_PERIODIC_4_M);
105             break;
106         case SHT3XD_FREQUENCY_10HZ:
107             error = writeCommand(SHT3XD_CMD_PERIODIC_10_M);
108             break;
109         default:
110             error = SHT3XD_PARAM_WRONG_FREQUENCY;
111             break;
112         }
113         break;
114
115     case SHT3XD_REPEATABILITY_HIGH:
116         switch (frequency)
117         {
118         case SHT3XD_FREQUENCY_HZ5:
119             error = writeCommand(SHT3XD_CMD_PERIODIC_HALF_H);
120             break;
121         case SHT3XD_FREQUENCY_1HZ:
122             error = writeCommand(SHT3XD_CMD_PERIODIC_1_H);
123             break;
124         case SHT3XD_FREQUENCY_2HZ:
125             error = writeCommand(SHT3XD_CMD_PERIODIC_2_H);
126             break;
127         case SHT3XD_FREQUENCY_4HZ:
128             error = writeCommand(SHT3XD_CMD_PERIODIC_4_H);
129             break;
130         case SHT3XD_FREQUENCY_10HZ:
131             error = writeCommand(SHT3XD_CMD_PERIODIC_10_H);
132             break;
133         default:
134             error = SHT3XD_PARAM_WRONG_FREQUENCY;
135             break;
136         }
137         break;
138     default:
139         error = SHT3XD_PARAM_WRONG_REPEATABILITY;
140         break;
141     }
142
143     delay(100);
144
145     return error;
146 }
147
148 SHT31D ClosedCube_SHT31D::readTempAndHumidity(SHT31D_Repeatability repeatability, SHT31D_Mode mode,
        uint8_t timeout)
149 {
150     SHT31D result;
151
152     switch (mode) {
153     case SHT3XD_MODE_CLOCK_STRETCH:
154         result = readTempAndHumidityClockStretch(repeatability);
155         break;
156     case SHT3XD_MODE_POLLING:
157         result = readTempAndHumidityPolling(repeatability, timeout);
158         break;
159     default:
160         result = returnError(SHT3XD_PARAM_WRONG_MODE);
161         break;
```

```
162     }
163
164     return result;
165 }
166
167
168 SHT31D ClosedCube_SHT31D::readTempAndHumidityClockStretch(SHT31D_Repeatability repeatability)
169 {
170     SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
171     SHT31D_Commands command;
172
173     switch (repeatability)
174     {
175     case SHT3XD_REPEATABILITY_LOW:
176         error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_L);
177         break;
178     case SHT3XD_REPEATABILITY_MEDIUM:
179         error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_M);
180         break;
181     case SHT3XD_REPEATABILITY_HIGH:
182         error = writeCommand(SHT3XD_CMD_CLOCK_STRETCH_H);
183         break;
184     default:
185         error = SHT3XD_PARAM_WRONG_REPEATABILITY;
186         break;
187     }
188
189     delay(50);
190
191     if (error == SHT3XD_NO_ERROR) {
192         return readTemperatureAndHumidity();
193     } else {
194         return returnError(error);
195     }
196
197 }
198
199
200 SHT31D ClosedCube_SHT31D::readTempAndHumidityPolling(SHT31D_Repeatability repeatability, uint8_t timeout)
201 {
202     SHT31D_ErrorCode error = SHT3XD_NO_ERROR;
203     SHT31D_Commands command;
204
205     switch (repeatability)
206     {
207     case SHT3XD_REPEATABILITY_LOW:
208         error = writeCommand(SHT3XD_CMD_POLLING_L);
209         break;
210     case SHT3XD_REPEATABILITY_MEDIUM:
211         error = writeCommand(SHT3XD_CMD_POLLING_M);
212         break;
213     case SHT3XD_REPEATABILITY_HIGH:
214         error = writeCommand(SHT3XD_CMD_POLLING_H);
215         break;
216     default:
217         error = SHT3XD_PARAM_WRONG_REPEATABILITY;
218         break;
219     }
220
221     delay(timeout);
222
223     if (error == SHT3XD_NO_ERROR) {
224         return readTemperatureAndHumidity();
225     } else {
226         return returnError(error);
227     }
228
229 }
230
231 SHT31D ClosedCube_SHT31D::readAlertHighSet() {
232     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_HS);
```

```
233 }
234
235 SHT31D ClosedCube_SHT31D::readAlertHighClear() {
236     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_HC);
237 }
238
239 SHT31D ClosedCube_SHT31D::readAlertLowSet() {
240     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_LS);
241 }
242
243 SHT31D ClosedCube_SHT31D::readAlertLowClear() {
244     return readAlertData(SHT3XD_CMD_READ_ALR_LIMIT_LC);
245 }
246
247
248 SHT31D_ErrorCode ClosedCube_SHT31D::writeAlertHigh(float temperatureSet, float temperatureClear, float
        humiditySet, float humidityClear) {
249     SHT31D_ErrorCode error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_HS, temperatureSet, humiditySet);
250     if (error == SHT3XD_NO_ERROR)
251         error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_HC, temperatureClear, humidityClear);
252
253     return error;
254 }
255
256 SHT31D_ErrorCode ClosedCube_SHT31D::writeAlertLow(float temperatureClear, float temperatureSet, float
        humidityClear, float humiditySet) {
257     SHT31D_ErrorCode error = writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_LS, temperatureSet, humiditySet);
258     if (error == SHT3XD_NO_ERROR)
259         writeAlertData(SHT3XD_CMD_WRITE_ALR_LIMIT_LC, temperatureClear, humidityClear);
260
261     return error;
262 }
263
264 SHT31D_ErrorCode ClosedCube_SHT31D::writeAlertData(SHT31D_Commands command, float temperature, float
        humidity)
265 {
266     SHT31D_ErrorCode  error;
267
268     if ((humidity < 0.0) || (humidity > 100.0) || (temperature < -40.0) || (temperature > 125.0))
269     {
270         error = SHT3XD_PARAM_WRONG_ALERT;
271     } else
272     {
273         uint16_t rawTemperature = calculateRawTemperature(temperature);
274         uint16_t rawHumidity = calculateRawHumidity(humidity);
275         uint16_t data = (rawHumidity & 0xFE00) | ((rawTemperature >> 7) & 0x001FF);
276
277         uint8_t  buf[2];
278         buf[0] = data >> 8;
279         buf[1] = data & 0xFF;
280
281         uint8_t checksum = calculateCrc(buf);
282
283         Wire.beginTransmission(_address);
284         Wire.write(command >> 8);
285         Wire.write(command & 0xFF);
286         Wire.write(buf[0]);
287         Wire.write(buf[1]);
288         Wire.write(checksum);
289         return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
290     }
291
292     return error;
293 }
294
295
296 SHT31D_ErrorCode ClosedCube_SHT31D::writeCommand(SHT31D_Commands command)
297 {
298     Wire.beginTransmission(_address);
299     Wire.write(command >> 8);
300     Wire.write(command & 0xFF);
```

```
301     return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
302 }
303
304 SHT31D_ErrorCode ClosedCube_SHT31D::softReset() {
305     return writeCommand(SHT3XD_CMD_SOFT_RESET);
306 }
307
308 SHT31D_ErrorCode ClosedCube_SHT31D::generalCallReset() {
309     Wire.beginTransmission(0x0);
310     Wire.write(0x06);
311     return (SHT31D_ErrorCode)(-10 * Wire.endTransmission());
312 }
313
314 SHT31D_ErrorCode ClosedCube_SHT31D::heaterEnable() {
315     return writeCommand(SHT3XD_CMD_HEATER_ENABLE);
316 }
317
318 SHT31D_ErrorCode ClosedCube_SHT31D::heaterDisable() {
319     return writeCommand(SHT3XD_CMD_HEATER_DISABLE);
320 }
321
322 SHT31D_ErrorCode ClosedCube_SHT31D::artEnable() {
323     return writeCommand(SHT3XD_CMD_ART);
324 }
325
326
327 uint32_t ClosedCube_SHT31D::readSerialNumber()
328 {
329     uint32_t result = SHT3XD_NO_ERROR;
330     uint16_t buf[2];
331
332     if (writeCommand(SHT3XD_CMD_READ_SERIAL_NUMBER) == SHT3XD_NO_ERROR) {
333         if (read(buf, 2) == SHT3XD_NO_ERROR) {
334             result = (buf[0] << 16) | buf[1];
335         }
336     }
337
338     return result;
339 }
340
341 SHT31D_RegisterStatus ClosedCube_SHT31D::readStatusRegister()
342 {
343     SHT31D_RegisterStatus result;
344
345     SHT31D_ErrorCode error = writeCommand(SHT3XD_CMD_READ_STATUS);
346     if (error == SHT3XD_NO_ERROR)
347         error = read(&result.rawData, 1);
348
349     return result;
350 }
351
352 SHT31D_ErrorCode ClosedCube_SHT31D::clearAll() {
353     return writeCommand(SHT3XD_CMD_CLEAR_STATUS);
354 }
355
356
357 SHT31D ClosedCube_SHT31D::readTemperatureAndHumidity()
358 {
359     SHT31D result;
360
361     result.t = 0;
362     result.rh = 0;
363
364     SHT31D_ErrorCode error;
365     uint16_t buf[2];
366
367     if (error == SHT3XD_NO_ERROR)
368         error = read(buf, 2);
369
370     if (error == SHT3XD_NO_ERROR) {
371         result.t = calculateTemperature(buf[0]);
```

```
372         result.rh = calculateHumidity(buf[1]);
373      }
374      result.error = error;
375
376      return result;
377   }
378
379   SHT31D ClosedCube_SHT31D::readAlertData(SHT31D_Commands command)
380   {
381      SHT31D result;
382
383      result.t = 0;
384      result.rh = 0;
385
386      SHT31D_ErrorCode error;
387
388      uint16_t buf;
389
390      error = writeCommand(command);
391
392      if (error == SHT3XD_NO_ERROR)
393         error = read(&buf, 1);
394
395      if (error == SHT3XD_NO_ERROR) {
396         result.rh = calculateHumidity(buf & 0xFE00);
397         result.t = calculateTemperature(buf << 7);
398      }
399
400      result.error = error;
401
402      return result;
403   }
404
405   SHT31D_ErrorCode ClosedCube_SHT31D::read(uint16_t* data, uint8_t numOfPair)
406   {
407      uint8_t  buf[2];
408      uint8_t checksum;
409
410      const uint8_t numOfBytes = numOfPair * 3;
411      Wire.requestFrom(_address, numOfBytes);
412
413      int counter = 0;
414
415      for (counter = 0; counter < numOfPair; counter++) {
416         Wire.readBytes(buf, (uint8_t)2);
417         checksum = Wire.read();
418
419         if (checkCrc(buf, checksum) != 0)
420            return SHT3XD_CRC_ERROR;
421
422         data[counter] = (buf[0] << 8) | buf[1];
423      }
424
425      return SHT3XD_NO_ERROR;
426   }
427
428
429   uint8_t ClosedCube_SHT31D::checkCrc(uint8_t data[], uint8_t checksum)
430   {
431      return calculateCrc(data) != checksum;
432   }
433
434   float ClosedCube_SHT31D::calculateTemperature(uint16_t rawValue)
435   {
436      return 175.0f * (float)rawValue / 65535.0f - 45.0f;
437   }
438
439
440   float ClosedCube_SHT31D::calculateHumidity(uint16_t rawValue)
441   {
442      return 100.0f * rawValue / 65535.0f;
```

```
443 }
444
445 uint16_t ClosedCube_SHT31D::calculateRawTemperature(float value)
446 {
447    return (value + 45.0f) / 175.0f * 65535.0f;
448 }
449
450 uint16_t ClosedCube_SHT31D::calculateRawHumidity(float value)
451 {
452    return value / 100.0f * 65535.0f;
453 }
454
455 uint8_t ClosedCube_SHT31D::calculateCrc(uint8_t data[])
456 {
457    uint8_t bit;
458    uint8_t crc = 0xFF;
459    uint8_t dataCounter = 0;
460
461    for (; dataCounter < 2; dataCounter++)
462    {
463       crc ^= (data[dataCounter]);
464       for (bit = 8; bit > 0; --bit)
465       {
466          if (crc & 0x80)
467             crc = (crc << 1) ^ 0x131;
468          else
469             crc = (crc << 1);
470       }
471    }
472
473    return crc;
474 }
475
476 SHT31D ClosedCube_SHT31D::returnError(SHT31D_ErrorCode error) {
477    SHT31D result;
478    result.t = 0;
479    result.rh = 0;
480    result.error = error;
481    return result;
482 }
```

pages/sourcecode/SHT30.cpp

SDI12FUNC.H AND .CPP

```
1  #ifndef SDI12FUNC_H
2  #define SDI12FUNC_H
3  #include "SDI12.h"
4  #include "LedColor.h"
5
6  #ifndef DEBUG_STREAM
7  #define DEBUG_STREAM SerialUSB          // Define SerialUSB as debug stream
8  #define DEBUG_BAUD 9600                 // Define Serial BAUD
9  #endif
10 #define SDI_DATA_PIN 10         // The pin of the SDI-12 data bus
11
12
13 byte charToDec(char i);
14 char decToChar(byte i);
15 void printBufferToScreen();
16 void printInfo(char i);
17 void takeMeasurement(char i,String* value);
18 boolean checkActive(char i);
19 boolean isTaken(byte i);
20 boolean setTaken(byte i);
21 boolean setVacant(byte i);
22 char sdiSetup(void);
23 void sdiMeasure(char address);
24 void sdiMeasure(float* spd,float* dir, char address);
25 #endif
```

pages/sourcecode/SDI12func.h

```
#include <Arduino.h>                    // Include Arduino Library
#include <Wire.h>                       // Include I2C Library

#include "SDI12func.h"                  // Include Header

byte addressRegister[8] = {             // Initialise "Empty" SDI12 Adress Register
  0B00000000,
  0B00000000,
  0B00000000,
  0B00000000,
  0B00000000,
  0B00000000,
  0B00000000,
  0B00000000
};

uint8_t numSensors = 0;                 // Initialise the number of sensors to 0

SDI12 mySDI12(SDI_DATA_PIN);

// keeps track of active addresses
// each bit represents an address:
// 1 is active (taken), 0 is inactive (available)
// setTaken('A') will set the proper bit for sensor 'A'

// converts allowable address characters '0'-'9', 'a'-'z', 'A'-'Z',
// to a decimal number between 0 and 61 (inclusive) to cover the 62 possible addresses
byte charToDec(char i){
  if((i >= '0') && (i <= '9')) return i - '0';
  if((i >= 'a') && (i <= 'z')) return i - 'a' + 10;
  if((i >= 'A') && (i <= 'Z')) return i - 'A' + 37;
  else return i;
}

// THIS METHOD IS UNUSED IN THIS EXAMPLE, BUT IT MAY BE HELPFUL.
// maps a decimal number between 0 and 61 (inclusive) to
// allowable address characters '0'-'9', 'a'-'z', 'A'-'Z',
char decToChar(byte i){
  if((i >= 0) && (i <= 9)) return i + '0';
  if((i >= 10) && (i <= 36)) return i + 'a' - 10;
  if((i >= 37) && (i <= 62)) return i + 'A' - 37;
  else return i;
}

void printBufferToScreen(){
  String buffer = "";
  mySDI12.read(); // consume address
  while(mySDI12.available()){
    char c = mySDI12.read();
    if(c == '+'){
      buffer += ',';
    }
    else if ((c != '\n') && (c != '\r')) {
      buffer += c;
    }
    while(!(mySDI12.available() > 0));
  }
 DEBUG_STREAM.print(buffer);
}

// gets identification information from a sensor, and prints it to the serial port
// expects a character between '0'-'9', 'a'-'z', or 'A'-'Z'.
void printInfo(char i){
  String command = "";
  command += (char) i;
  command += "I!";
  mySDI12.sendCommand(command);
```

```
68    delay(50);
69
70    printBufferToScreen();
71  }
72
73  void takeMeasurement(char i,String* value)
74  {
75    String command = "";
76    command += i;
77    command += "R0!"; // SDI-12 measurement command format  [address]['R0'][!]
78
79    mySDI12.sendCommand(command);
80
81    mySDI12.clearBuffer();
82
83    // in this example we will only take R0 measurement no. 1 and 2 (avg speed, avg direction)
84    String sdiResponse = "";
85    String *p;
86    p = value;
87    int k=0;
88
89    while(!(mySDI12.available() > 1)); // wait for acknowlegement
90    mySDI12.read();
91    while(!(mySDI12.available() > 1));
92    mySDI12.read();
93    while (mySDI12.available())  // build reply string
94    {
95      char c = mySDI12.read();
96      DEBUG_STREAM.print(c);
97      DEBUG_STREAM.print(' ');
98      DEBUG_STREAM.println(" ");
99      if(c == '+')
100     {
101       k ++;
102       if(k >=2)
103       {
104         break;
105       }
106       sdiResponse += ' ';
107       value += ' ';
108       while(!(mySDI12.available() > 0));
109     }
110     else if ((c != '\n') && (c != '\r'))
111     {
112
113       sdiResponse += c;
114       while(!(mySDI12.available() > 0));
115     }
116   }
117
118   DEBUG_STREAM.println("finished message");
119   //DEBUG_STREAM.println(sdiResponse);
120   mySDI12.clearBuffer();
121   *p = sdiResponse;
122
123 }
124
125 // this checks for activity at a particular address
126 // expects a char, '0'-'9', 'a'-'z', or 'A'-'Z'
127 boolean checkActive(char i){
128
129   String myCommand = "";
130   myCommand = "";
131   myCommand += (char) i;                 // sends basic 'acknowledge' command [address][!]
132   myCommand += "!";
133
134   for(int j = 0; j < 3; j++){            // goes through three rapid contact attempts
135     mySDI12.sendCommand(myCommand);
136     unsigned long timerStart = millis();
137     while((millis() - timerStart) < (30))
138     {
```

```
      if(mySDI12.available())
      {  // If we here anything, assume we have an active sensor
        printBufferToScreen();
        mySDI12.clearBuffer();
        return true;
      }
    }
  }
  mySDI12.clearBuffer();
  return false;
}

// this quickly checks if the address has already been taken by an active sensor
boolean isTaken(byte i){
  i = charToDec(i); // e.g. convert '0' to 0, 'a' to 10, 'Z' to 61.
  byte j = i / 8;   // byte #
  byte k = i % 8;   // bit #
  return addressRegister[j] & (1<<k); // return bit status
}

// this sets the bit in the proper location within the addressRegister
// to record that the sensor is active and the address is taken.
boolean setTaken(byte i){
  boolean initStatus = isTaken(i);
  i = charToDec(i); // e.g. convert '0' to 0, 'a' to 10, 'Z' to 61.
  byte j = i / 8;   // byte #
  byte k = i % 8;   // bit #
  addressRegister[j] |= (1 << k);
  return !initStatus; // return false if already taken
}

// THIS METHOD IS UNUSED IN THIS EXAMPLE, BUT IT MAY BE HELPFUL.
// this unsets the bit in the proper location within the addressRegister
// to record that the sensor is active and the address is taken.
boolean setVacant(byte i){
  boolean initStatus = isTaken(i);
  i = charToDec(i); // e.g. convert '0' to 0, 'a' to 10, 'Z' to 61.
  byte j = i / 8;   // byte #
  byte k = i % 8;   // bit #
  addressRegister[j] &= ~(1 << k);
  return initStatus; // return false if already vacant
}


char sdiSetup(void){
  mySDI12.begin();
  setLedColor(BLUE);
  delay(800); // allow things to settle

  boolean found = false;

  if(checkActive('A')){numSensors++; setTaken('A');found = true;}


  if(!found) {
    setLedColor(YELLOW);
    for(byte i = '0'; i <= '9'; i++) if(checkActive(i)) {numSensors++; setTaken(i);}    // scan address
     space 0-9

    for(byte i = 'a'; i <= 'z'; i++) if(checkActive(i)) {numSensors++; setTaken(i);}    // scan address
     space a-z

    for(byte i = 'A'; i <= 'Z'; i++) if(checkActive(i)) {numSensors++; setTaken(i);}    // scan address
     space A-Z

  /*
      See if there are any active sensors.
   */
    found = false;

    for(byte i = 0; i < 62; i++)
```

```
207     {
208       if(isTaken(i))
209       {
210         found = true;
211         DEBUG_STREAM.print("First address found:  ");
212         DEBUG_STREAM.println(decToChar(i));
213         DEBUG_STREAM.print("Total number of sensors found:  ");
214         DEBUG_STREAM.println(numSensors);
215         setLedColor(GREEN);
216         return decToChar(i);
217       }
218     }
219     DEBUG_STREAM.println();
220     setLedColor(RED);
221     return ' ';
222   }
223   else{
224     DEBUG_STREAM.println("ATMOS22 found at 'A'");
225     setLedColor(GREEN);
226     return 'A';
227   } // stop here
228
229 }
230
231
232
233 void sdiMeasure(float* spd,float* dir, char address)
234 {
235   float spd2,dir2;
236   String spdS,dirS;
237   String value = "";
238   takeMeasurement(address,&value);
239
240   for(uint i = 0;i < value.length(); i++)
241   {
242     if(value.charAt(i) == ' ')
243     {
244       spdS = value.substring(0,i-1);
245       dirS = value.substring(i);
246     }
247   }
248
249   spd2 = spdS.toFloat();
250   dir2 = dirS.toFloat();
251   *spd = spd2;
252   *dir = dir2;
253
254 }
255
256 void sdiMeasure(char address)
257 {
258   float spd,dir;
259   String spdS,dirS;
260   String value = "";
261   takeMeasurement(address,&value);
262   for(uint i = 0;i < value.length(); i++)
263   {
264     if(value.charAt(i) == ' ')
265     {
266       spdS = value.substring(0,i-1);
267       dirS = value.substring(i);
268     }
269   }
270   spd = spdS.toFloat();
271   dir = dirS.toFloat();
272   DEBUG_STREAM.println(spd);
273   DEBUG_STREAM.println(dir);
274 }
```

pages/sourcecode/SDI12func.cpp

SDI12.H AND .CPP

The code functions described in SDI12func make use of the EnviroDIY SDI12 library for Arduino. Since the library is used without any changes, and the .cpp accounts for over 1000 lines of code, it has been decided to just include the header file which contains licence and developer's credits, as well as function descriptions.

```
/* ======================== Arduino SDI-12 ================================

Arduino library for SDI-12 communications to a wide variety of environmental
sensors. This library provides a general software solution, without requiring
any additional hardware.

======================== Attribution & License ============================

Copyright (C) 2013  Stroud Water Research Center
Available at https://github.com/EnviroDIY/Arduino-SDI-12

Authored initially in August 2013 by:

        Kevin M. Smith (http://ethosengineering.org)
        Inquiries: SDI12@ethosengineering.org

Modified 2017 by Manuel Jimenez Buendia to work with ARM based processors
(Arduino Zero)

Maintenance and merging 2017 by Sara Damiano

based on the SoftwareSerial library (formerly NewSoftSerial), authored by:
        ladyada (http://ladyada.net)
        Mikal Hart (http://www.arduiniana.org)
        Paul Stoffregen (http://www.pjrc.com)
        Garrett Mace (http://www.macetech.com)
        Brett Hagman (http://www.roguerobotics.com/)

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
*/

#ifndef SDI12_h
#define SDI12_h

    //  Import Required Libraries
#include <inttypes.h>           // integer types library
#include <Arduino.h>            // Arduino core library
#include <Stream.h>             // Arduino Stream library

typedef const __FlashStringHelper *FlashString;

#define NO_IGNORE_CHAR '\x01' // a char not found in a valid ASCII numeric field
#define SDI12_BUFFER_SIZE 64   // max Rx buffer size

class SDI12 : public Stream
{
protected:
  // hides the version from the stream to allow custom timeout value
  int peekNextDigit(LookaheadMode lookahead, bool detectDecimal);

private:

  // For the various SDI12 states
```

```
66   typedef enum SDI12_STATES
67   {
68     DISABLED = 0,
69     ENABLED = 1,
70     HOLDING = 2,
71     TRANSMITTING = 3,
72     LISTENING = 4
73   } SDI12_STATES;
74
75   static SDI12 *_activeObject;    // static pointer to active SDI12 instance
76
77   void setPinInterrupts(bool enable);  // Turns pin interrupts on or off
78   void setState(SDI12_STATES state);   // sets the state of the SDI12 objects
79   void wakeSensors();              // used to wake up the SDI12 bus
80   void writeChar(uint8_t out);     // used to send a char out on the data line
81   void startChar();                // creates a blank slate for a new incoming character
82   void receiveISR();               // the actual function responding to changes in rx line state
83   void charToBuffer(uint8_t c);    // puts a finished character into the SDI12 buffer
84
85   #ifndef __AVR__
86     static uint8_t parity_even_bit(uint8_t v);
87   #endif
88
89   uint8_t _dataPin;                // reference to the data pin
90
91   static uint8_t _rxBuffer[SDI12_BUFFER_SIZE];  // A single buffer for ALL SDI-12 objects
92   static volatile uint8_t _rxBufferTail;
93   static volatile uint8_t _rxBufferHead;
94   bool _bufferOverflow;            // buffer overflow status
95
96 public:
97   SDI12();                           // constructor - without argument, for better library integration
98   SDI12(uint8_t dataPin);            // constructor
99   ~SDI12();                          // destructor
100  void begin();                      // enable SDI-12 object
101  void begin(uint8_t dataPin);       // enable SDI-12 object - if you use the empty constuctor, USE THIS
102  void end();                        // disable SDI-12 object
103  int TIMEOUT;                       // value to return if a parse times out
104  void setTimeoutValue(int value);   // sets the value to return if a parse int or parse float times out
105  uint8_t getDataPin();              // returns the data pin for the current instace
106
107  void forceHold();                      // sets line state to HOLDING
108  void forceListen();                    // sets line state to LISTENING
109  void sendCommand(String &cmd);         // sends the String cmd out on the data line
110  void sendCommand(const char *cmd);     // sends the String cmd out on the data line
111  void sendCommand(FlashString cmd);     // sends the String cmd out on the data line
112  void sendResponse(String &resp);       // sends the String resp out on the data line (for slave use)
113  void sendResponse(const char *resp);   // sends the String resp out on the data line (for slave use)
114  void sendResponse(FlashString resp);   // sends the String resp out on the data line (for slave use)
115
116  int available();             // returns the number of bytes available in buffer
117  int peek();                  // reveals next byte in buffer without consuming
118  int read();                  // returns next byte in the buffer (consumes)
119  void clearBuffer();          // clears the buffer
120  void flush(){};              // Waits for sending to finish - because no TX buffering, does nothing
121  virtual size_t write(uint8_t byte);  // standard stream function
122
123  // hide the Stream equivalents to allow custom value to be returned on timeout
124  long parseInt(LookaheadMode lookahead = SKIP_ALL, char ignore = NO_IGNORE_CHAR);
125  float parseFloat(LookaheadMode lookahead = SKIP_ALL, char ignore = NO_IGNORE_CHAR);
126
127  bool setActive();          // set this instance as the active SDI-12 instance
128  bool isActive();           // check if this instance is active
129
130  static void handleInterrupt();  // intermediary used by the ISR
131
132  // #define SDI12_EXTERNAL_PCINT  // uncomment to use your own PCINT ISRs
133
134 };
135
136 #endif  // SDI12_h
```

pages/sourcecode/SDI12.h

D

LoRa

This appendix is included to provide additional information about LoRa and LoRaWAN.
LoRa is a wireless data communications IoT technology. The technology was developed by Cycleo of Grenoble, which was later acquired by Semtech. With the use of LoRa, data can be communicated over long ranges with little power consumption. The technology makes use of license free sub-Gigahertz RF bands or the so called ISM-bands (Industrial, Scientific and Medical bands), which in Europe includes 868 MHz.

D.1. Link Budget

A link budget is the amount of power that can be attenuated before losing the ability to communicate. This means that it's the difference between the transmitters TX power and the receivers RX sensitivity.
In the case of LoRa, the TX power is maximally 14 dB m, while the receiver sensitivity is at −137 dB m, resulting in a link budget of 151 dB m! Meaning LoRa is able to operate below the noise floor. This is really high when compared with other Wireless Communication Networks as tabulated in table D.1

Table D.1: Comparison of Link Budgets of different technologies.

	TX Power [dB m]	RX Sensitivity [dB m]	Link Budget [dB m]
Wi-Fi	20	−75	95
Sub-GHz 6LoWPAN	11	−110	121
LoRa	14	−137	151

D.2. LoRa, the Physical Layer

LoRa is an abbreviation of Long Range. To achieve this Long Range (Longer than the widely used FSK modulation (Frequency Shift Keying)) LoRa makes use, among onther techniques, of a modulation scheme called Chrip Spread Spectrum (CSS). To receive or transmit LoRa signals, hardware that supports this modulation scheme is needed and is only made by Semtech, since they have the patent on LoRa.
Due to this fact, rigorous documentation about the inner workings of this physical layer are not available, however, reverse-engineering attempts have been made and via this way some more insight can be acquired on the workings of this physical layer.[41, 42]
In addition to the reverse engineering work, the LoRa-Alliance itself provides an explaining document.[43]

BIBLIOGRAPHY

[1] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, "Long-range communications in unlicensed bands: the rising stars in the iot and smart city scenarios," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 60–67, October 2016.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, Fourthquarter 2015.

[3] J. K. Hart and K. Martinez, "Environmental sensor networks: A revolution in the earth system science?" *Earth-Science Reviews*, vol. 78, no. 3, pp. 177 – 191, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0012825206000511

[4] A. Förster, *Designing and Deploying WSN Applications*. Wiley-IEEE Press, 2016, pp. 186–. [Online]. Available: https://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7906230

[5] M. T. Lazarescu, "Design of a wsn platform for long-term environmental monitoring for iot applications," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 1, pp. 45–54, March 2013.

[6] J. Devaraju, K. Suhas, H. Mohana, and V. A. Patil, "Wireless portable microcontroller based weather monitoring station," *Measurement*, vol. 76, pp. 189 – 200, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0263224115004406

[7] S. C. Padwal, M. Kumar, P. Balaramudu, and C. K. Jha, "Analysis of environment changes using wsn for iot applications," in *2017 2nd International Conference for Convergence in Technology (I2CT)*, April 2017, pp. 27–32.

[8] Holfuy Meteorology. *About Holfuy*. Accessed: [11-6-2018]. [Online]. Available: https://holfuy.com/en/about

[9] G. Pahl, W. Beitz, J. Feldhusen, and K. Grote, *Engineering Design: A Systematic Approach*, ser. Solid mechanics and its applications. Springer London, 2007. [Online]. Available: https://books.google.nl/books?id=57aWTCE3gE0C

[10] J. H. Wohlgemuth, D. W. Cunningham, P. Monus, J. Miller, and A. Nguyen, "Long term reliability of photovoltaic modules," in *2006 IEEE 4th World Conference on Photovoltaic Energy Conference*, vol. 2, May 2006, pp. 2050–2053.

[11] Sensirion, *Datasheet SHT3x-DIS*, May 2018, rev 5. [Online]. Available: https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/0_Datasheets/Humidity/Sensirion_Humidity_Sensors_SHT3x_Datasheet_digital.pdf

[12] Bosch Sensortech, *BMP180 Digital pressure sensor*, April 5th 2013, rev 2.5. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf

[13] Hydreon Corporation, *MODEL RG-11 OPTICAL RAIN GAUGE*, rev 016. [Online]. Available: http://hydreon.com/wp-content/uploads/sites/3/2015/documents/rg-11_instructions.pdf

[14] METER Group, *ATMOS 22*, 2018. [Online]. Available: http://library.metergroup.com/Manuals/20419_ATMOS22_Manual_Web.pdf

[15] Encyclopædia Britannica, "Cloudburst," October 25, 2016, accessed 17-6-2018. [Online]. Available: https://www.britannica.com/science/cloudburst

[16] A. Buishand and J. Wijngaard, "Statistiek van extreme neerslag voor korte neerslagduren," Koninklijk Nederlands Meteorologisch Instituut, Tech. Rep. TR-295, January 2007.

[17] P. S. Cheong, J. Bergs, C. Hawinkel, and J. Famaey, "Comparison of lorawan classes and their power consumption," in *2017 IEEE Symposium on Communications and Vehicular Technology (SCVT)*, Nov 2017, pp. 1–6.

[18] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer, "A comparative study of lpwan technologies for large-scale iot deployment," *ICT Express*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405959517302953

[19] J. P. Bardyn, T. Melly, O. Seller, and N. Sornin, "Iot: The era of lpwan is starting now," in *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, Sept 2016, pp. 25–30.

[20] H. Mroue, A. Nasser, S. Hamrioui, B. Parrein, E. Motta-Cruz, and G. Rouyer, "Mac layer-based evaluation of iot technologies: Lora, sigfox and nb-iot," in *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*, April 2018, pp. 1–5.

[21] Semtech, *SX1272/3/6/7/8: LoRa Modem Designer's Guide AN1200.13*, July 2013. [Online]. Available: https://www.semtech.com/uploads/documents/LoraDesignGuide_STD.pdf

[22] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, "Understanding the limits of lorawan," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 34–40, 2017.

[23] E. Koutroulis, K. Kalaitzakis, and N. C. Voulgaris, "Development of a microcontroller-based, photovoltaic maximum power point tracking control system," *IEEE Transactions on Power Electronics*, vol. 16, no. 1, pp. 46–54, Jan 2001.

[24] F. Reverter and M. Gasulla, "Optimal inductor current in boost dc/dc converters regulating the input voltage applied to low-power photovoltaic modules," *IEEE Transactions on Power Electronics*, vol. 32, no. 8, pp. 6188–6196, Aug 2017.

[25] H. Keshan, J. Thornburg, and T. S. Ustun, "Comparison of lead-acid and lithium ion batteries for stationary storage in off-grid energy systems," in *4th IET Clean Energy and Technology Conference (CEAT 2016)*, Nov 2016, pp. 1–7.

[26] METER Group, *ATMOS 22 Integrators Guide*, 2018. [Online]. Available: http://library.metergroup.com/Integration\Guides/18195\ATMOS\22\Integrators\Guide.pdf

[27] J. Zhang, L. Zhang, F. Sun, and Z. Wang, "An overview on thermal safety issues of lithium-ion batteries for electric vehicle application," *IEEE Access*, vol. 6, pp. 23 848–23 863, 2018.

[28] Linear Technology Corporation, "Lt3652- power tracking 2a battery charger for solar power," 2010, rev D. [Online]. Available: https://cdn.sparkfun.com/datasheets/Prototyping/LT3652.pdf

[29] Panasonic Corporation, *NCR18650B*, 2012, rev 13.11. [Online]. Available: https://www.batteryspace.com/prod-specs/NCR18650B.pdf

[30] S. Klein and W. Beckman, "Loss-of-load probabilities for stand-alone photovoltaic systems," *Solar Energy*, vol. 39, no. 6, pp. 499 – 512, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0038092X87900570

[31] Photovoltaic Geographical Information System. "performance of off-grid pv systems". Accessed: [12-6-2018]. [Online]. Available: http://re.jrc.ec.europa.eu/pvg_tools/en/tools.html#SA

[32] N. Khanduri, A. Kukreti, and N. Shah, "Implementation of solar time based sun tracking systems for mobile platforms and smart cities," in *2017 IEEE Region 10 Symposium (TENSYMP)*, July 2017, pp. 1–5.

[33] J. S. Reddy, A. Chakraborti, and B. Das, "Implementation and practical evaluation of an automatic solar tracking system for different weather conditions," in *2016 IEEE 7th Power India International Conference (PIICON)*, Nov 2016, pp. 1–6.

[34] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power management in energy harvesting sensor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 4, Sep. 2007. [Online]. Available: http://doi.acm.org/10.1145/1274858.1274870

[35] C. Moser, L. Thiele, D. Brunelli, and L. Benini, “Adaptive power management in energy harvesting systems,” in *2007 Design, Automation Test in Europe Conference Exhibition*, April 2007, pp. 1–6.

[36] J. R. Piorno, C. Bergonzini, D. Atienza, and T. S. Rosing, “Prediction and management in energy harvested wireless sensor nodes,” in *2009 1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology*, May 2009, pp. 6–10.

[37] N. Sharma, J. Gummeson, D. Irwin, and P. Shenoy, “Cloudy computing: Leveraging weather forecasts in energy harvesting sensor systems,” in *2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, June 2010, pp. 1–9.

[38] S. Janković and L. Saranovac, “Improving energy usage in energy harvesting wireless sensor nodes using weather forecast,” in *2017 25th Telecommunication Forum (TELFOR)*, Nov 2017, pp. 1–4.

[39] Y. Li, Z. Jia, and X. Li, “Task scheduling based on weather forecast in energy harvesting sensor systems,” *IEEE Sensors Journal*, vol. 14, no. 11, pp. 3763–3765, Nov 2014.

[40] R. Ayop, N. M. Isa, and C. W. Tan, “Components sizing of photovoltaic stand-alone system based on loss of power supply probability,” *Renewable and Sustainable Energy Reviews*, vol. 81, pp. 2731 – 2743, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364032117310201

[41] Rev Space. Decodinglora. Accessed: [14-6-2018]. [Online]. Available: https://revspace.nl/DecodingLora

[42] M. Knight and B. Seeber. Decoding lora: Realizing a modern lpwan with sdr". Accessed: [14-6-2018]. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/download/8/7

[43] LoRa Alliance. "lorawan™: What is it? a technical overview of lora® and lorawan™". Accessed: [14-6-2018]. [Online]. Available: https://www.lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf