# Lake Level Monitoring

Final Report for CS39440 Major Project

*Author*: Tim Stableford (tis4@aber.ac.uk)
*Supervisor*: Dr. Colin Suaze (cos@aber.ac.uk)

25th April 2015

Version 2.0 (Final)

This report is submitted as partial fulfilment of a BSc degree in Computer Science (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

**Declaration of originality**

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature ..... Tim Stableford

Date .....

**Consent to share this work**

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature ..... Tim Stableford

Date .....

Ethics research application number: 1103

# Abstract

Remotely monitoring of sensors is an old problem with many solutions, but integrating this into the IoT is a new problem. There is a building management system called EMonCMS which aims to aggregate data from many types of sensors and display them all in a web interface. The aim of this project is to create a low power network which can interface with EMonCMS through a gateway. The low power network should be able to run from low power sources such as solar power. It should be a generic system capable of relaying various types of sensor. The end output of this project is to implement an example sensor on this system which monitors the water level of the lake which supplies water for the Centre for Alternative Technology Wales.

# Contents

# 1. Background, Analysis & Process

## 1.1. Background

CAT (Centre for Alternative Technology Wales) is a company whose purpose is to educate and consult on sustainable solutions[3]. These include renewable power sources, recycling and energy monitoring. In conjunction with SAW (Software Alliance Wales) and Aberystwyth University multiple projects have been handed out for dissertation students. These projects focus on expanding the energy monitoring and management system. There are two projects which focus on expanding the sensor network. This is going to be accomplished by creating a low-power radio network and an interface between this system and EMonCMS. Creating the low-power network and an example sensor is the goal of this project.

This system provides an interesting mix of hardware and low level software development. It also builds upon existing work to produce a useful product which will actually be used at CAT's site for monitoring the level of water in the lake that provides the water supply for the site.

The type of project is an engineering project, this means that most of the research was researching hardware, libraries and similar projects which may have been undertaken. From a hardware choice perspective the goal was to maximize battery life and minimize cost without sacrificing functionality. From a software perspective the goal is to create a generic product compatible with any type of sensor and to make it fit on the micro-controller chosen.

Most of the previous work done by others has been creating low-power networks based upon many different chip-sets and many levels of abstraction on top, ranging from simple message sending without received acknowledgement to mesh networking with automatic re-transmission. There has also been lots of work done into minimizing power usage of the various components that were chosen for this project. Part of the hardware research was selecting which of these will be best suited to a low-power environment based upon others findings on actual power usage. It was also necessary to research some protocols such as Modbus to correctly interface with the pressure sensor supplied by the university.

## 1.2. Analysis

### 1.2.1. Document Analysis

The requirements for the project have been supplied in the form of stories as is often the case in Agile development. These stories supplied by CAT are as follows:

- I want something that can read the lake level.
- I want something that can send the lake level data back to the server in the specified format at a pre-defined interval.
- I want something that uses low-power radio to communicate with the server via a gateway.
- I want something developed on the Arduino platform.
- I want something that works off non-mains power.

- I want a self-contained communications stack/library included to allow encrypted communications from the specified inputs in the specified format.

The first requirement involves interfacing with the pressure sensor supplied by the university (PTM/N/RS485). This sensor communicates over half-duplex RS485[1] using the Modbus[2] protocol. This sensor requires converting the raw pressure reading to pressure in kPa which is the format required by CAT. To do this the sensor will have to be calibrated in water and then the base pressure level calibrated at the site to account for changes in air pressure from sea-level to the higher elevation at the CAT site.

The second requirement, to send the data back in the specified format at a pre-defined interval is mostly a loose requirement in this case because translation to the format they require is done by the gateway, implemented in another project. The specified format does however apply to the units that the lake level is measured in, in this case kPa as specified by CAT in an e-mail. Communications will be implemented in a format defined at the beginning of the project by myself and the developer of the gateway [4]. Pre-defined interval is elaborated on in the OEMan Communications Specification[5]. There are actually two types of nodes[1] that CAT want to be able to support, dumb nodes and smart nodes. Dumb nodes just post a value at set times, smart nodes are able to have requests made on them. Accomplishing posting data at a set time can have various different behaviours from the very simplistic to the more complex. At the simplest the node can post data whenever it is awake. At the more complex end of the spectrum a node can wake, send a post value and repeat until an acknowledgement is received until sleeping. All of these solutions require the addition of a real-time clock module to the hardware. For the interval to be pre-defined there must also be some way of storing the times to send data back to the server.

Another requirement is the use of low-power radios where low-power is defined as low power consumption. This step will involve choosing a suitable low-power radio that has enough range for the system to function on site. The data needs to go from the lake and over a hill before reaching the gateway. The total distance covered is approximately 1km. Under a best case scenario this is likely to require at least one repeater on the top of the hill. The requirement also states that this system must be able to communicate with the server through the gateway. To this end, as previously mentioned a specification has been developed for this project by the author and the gateway developer[4]. Depending on the radio chosen a network layer may have to be implemented. To meet CAT's needs most closely, a mesh or hybrid tree-mesh would need to be implemented.

CAT also wants something that runs on the Arduino platform. The Arduino platform includes a large range of products both official[6] and community based. The author chose to use official Arduino products rather than community based for the better support and smaller development time. The most commonly used chip on Arduino boards is the ATMega328P. This is used on the Uno, Duemilanove and Pro Mini. The Pro Mini is mostly a breakout board for the ATMega328P. It also has some support circuitry such as status LED's, a voltage regulator and an oscillator. The only more basic ATMega328P set-up uses a chip on breadboard with internal clock. Interestingly, a bare-bones set-up costs more than an Arduino Pro Mini. The Arduino Mega should

1 Nodes are a routing radio or sensor with radio connected to the low-power radio network.

also be noted for it's increased I/O and program space as a suitable candidate, however its power consumption and cost are higher. Some other ATMega AVR's are supported by the community such as the ATMega644P and the ATMega1284P. They are chips which can come in DIP formats to make them good for prototyping, unlike the Arduino Mega. They are also the chips which in terms of price and capability are between the ATMega328P based boards and the Arduino Mega.

Previously mentioned has been the need for the components such as the processor to be low-power. This is because one of the requirements are that the nodes run from non-mains power such as solar. To this end when components are selected, there power consumption must be kept in mind. The most important of these is the processor and real-time clock. The reason for that is that these are the devices which will be on the most. Other components such as the radio shouldn't use huge amounts of power but won't be on for long, so their power usage is not as important. Supporting circuitry should also take into account this low power use and voltage regulators such as for the radio should have low quiescent current. From a software perspective sleep modes should be implemented to conserve power, such as disabling the sensor and radio during periods of sleep. Further to this the ATMega chips support low-power sleep modes to conserve even more power.

CAT would also require that one of the software deliverables is a communications stack/library for the Arduino that supports encryption. This library will have to implement the specification previously mentioned[4] and provide an encryption layer between the networking layer and the message parsing layer. The encryption has to use little memory and as small code space as possible while being reasonably secure. Issues such as recording and repeating encrypted messages should be considered here.

### 1.2.2. Feature List

CAT have expressed interest in using an Agile methodology. The specific methodology has been left up to the author who has chosen to employ FDD (Feature Driven Development)[7]. FDD will be further described later in the document. Feature lists in FDD are grouped into feature sets which are groups of similar features. From the given stories the following ordered feature list was created:

- Read Lake level
  - Create hardware to interface with the sensor. (February 2015)
  - Convert sensor reading to pressure in kPa. (April 2015)
- Communications
  - Self-contained Communications Stack. (February 2015)
    - Register nodes to the gateway.
    - Register attributes to the gateway.
    - Post values at a specified interval to the gateway.
    - Respond to requests for values to the gateway.
  - Send data back to the server through the gateway. (March 2015)
    - Interface radio with the Arduino.
    - Build interface layer between radio and communications stack.
  - Encrypt communications. (April 2015)

- Power Management (April 2015)
  - Create hardware which runs from non-mains power.
    - Create circuits using low-power components.
    - Create code for going into low-power modes.
  - Create code for sleeping for pre-defined intervals.

### 1.2.3. Initial Choices

At this stage of development there is little room for anything outside of the scope of the features list. There are however a lot of different implementation options for each of the given features. This includes both hardware choice and software implementation. There are several key areas which need to be decided before going into development as they will have a greater impact across different features and to the second developer who is creating the gateway.

The first decision to be made was which Arduino to use, since using the Arduino platform is a requirement. Sticking to the official Arduinos a good choice would be the Arduino Fio, this is an Arduino Pro Mini with charge controller attached and connectors for a lithium battery. However, this board is not good value for money considering the price increase over the Arduino Pro Mini[2]. The Arduino Pro Mini can also have a clock speed of 16MHz compared to the 8MHz of the Arduino Fio which uses the ATMega328P internal clock. The other official Arduino to consider is the Arduino Mega, but this does not come in a low-power format without USB chips and voltage regulators, it's also more expensive than the ATMega328P. From this the ATMega328P is the best choice, the biggest downsides which will effect development being the lack of code space and memory.

The choice of radio is also important because depending on the level of implementation of libraries, these may have to be extended. An example of this would be the RFM12B[8]. This radio module has very little networking support except a small address range. It doesn't support meshing or routing and so that would have to be implemented in software. It's range is also limited to 300m, so it is not a suitable candidate for installation at the CAT site. Other modules such as the nRF24l01+LNA+PA[9] have a greater range, more features and better library support. A developer who goes by TMRh20 has enhanced existing Arduino and Raspberry Pi libraries for the nRF24l01 called RF24 and RF24Network[10]. This developer has also created another layer on top of the RF24Network layer called RF24Mesh. This adds limited meshing to the nRF24l01 radios in a tree-based topology with a master node coordinating. This radio also has a range of 1km in line of sight. This makes it ideal for use at the CAT site to meet the requirements they have stated.

At this stage there was also some research done into which encryption to use. The research at this stage was mainly into using Datagram Transport Layer Security (DTLS)[11]. This is because CAT had expressed an interest in using a version of IPv6 called 6LoWPAN[12]. This is a cut down version of IPv6 which is designed for low-power wireless networks. If this were to be used then using an encryption layer such as DTLS would fit well. It would also be reasonably radio agnostic and could perhaps even be placed below the IPv6 layer to provide better encryption. However, IPv6 was not used because CAT wanted a system which would “just work” with IPv6 devices and did not realise that a translation gateway would still have to be between the EMonCMS

2 The price is based upon eBay prices, April 2015.

server and the nodes. The networking layers provided by the RF24 libraries also provide similar functionality to 6LoWPAN but have existing implementations and are far more light weight protocols. Combining all of these makes 6LoWPAN a bad choice. The initial decision from here was to use a Diffie-Hellman key exchange and AES-128 on top of RF24Mesh.

### 1.2.4. Objectives

The objectives have been stated in the feature list but it does not state deliverables and there are a few stretch goals presented by CAT too. The features list implies the creation of hardware and software however, the intention is to deliver example hardware to CAT comprising of a sensor/radio board and the sensor itself with working software on. It's also a goal to deliver CAT enough information so that they can continue to develop the hardware and software. This includes circuit diagrams and well documented code.

CAT's stretch goals were all based upon expanding the system to include more nodes or using it as a data relay, these are as follows:

- Monitoring the flow of water into the lake which water level is being measured.
- Relaying weather data back from on-site wind turbines.
- Monitoring the water level of a second lake.

The biggest of these stretch goals is relaying data back from on-site wind turbines. This is because each wind turbine stores weather data in a different way, some have serial outputs and some write to SD card. Interfacing with these systems could be very complicated so this is an unlikely stretch goal. The one which would be accomplished first would be monitoring a second lake as that is just expanding the system.

## 1.3. Process

CAT requested that the projects done for them use an Agile methodology. Agile methodologies have the advantages of adaptive planning and continuous improvement. The agile methodology chosen was feature driven development (FDD)[7]. The key aspect of feature driven development compared to other Agile methodologies is that it has some upfront planning, this planning is then expanded upon and improved at the beginning of each feature. It also uses individual class ownership unlike other Agile methodologies such as Extreme Programming which use collective class ownership.

The reason FDD was chosen over other Agile methodologies is because it was unlikely that the stories would change significantly and some upfront planning can increase productivity in later stages. This is unlike in Extreme Programming where the design emerges from coding. This can cause problems where code has to be mostly rewritten to support a later story and can involve some wasted time. It does have benefits over FDD though such as a higher emphasis on testing and peer review which can create higher quality code on a low level even if the system when put together does not have a good design. Scrum is similar to Extreme Programming in that it has no upfront design and the major advantage it seems to have is that is scales better, such as the idea of a “Scrum of scrums”. It has much less of an emphasis on peer review and testing though, which makes it inferior when applied to single person projects such as this.

### 1.3.1. Roles

FDD sets out many different roles and is designed to be used in a team. The primary roles set out are[14]:

- Project Manager. This is the administrative head of the project whose work may involve reporting projects and managing resources.
- Chief Architect. Primarily responsible for the overall design of the system. Also provides support when overcoming technical hurdles.
- Development Manager. Leads day to day development activities. Also resolves everyday conflicts between chief programmers.
- Chief Programmers. In charge of groups of between three and six developers. This role ideally is filled by developers who have done the entire development cycle several times.
- Class Owner. Developers who do the bulk of designing, testing and coding under the guidance of a chief programmer.
- Domain Experts. This can be a range of people including sponsors or business analysts.

These roles can be filled by one or more people. In the case of this project they will all be filled by a single person. This could make several roles redundant, such as the difference between chief programmers and class owners being insignificant. The domain expert will not be on the team but will be a contact at CAT available through email.

### 1.3.2. Overall Model

The first iteration, or initial planning stage involves creating a high-level scope. This includes a system overview, feature list and can also include sequence diagrams and class diagrams. Subsequent iterations then focus on developing each feature and during the development of each feature there is a planning/design stage, involving documentation such as class diagrams, then that is followed by building it and improving the initial iterations documentation.

The first step of the initial iteration is often to form a modelling team. The modelling team is a core group of engineers which have domain specific knowledge. This is a project for one person so the author will be the modelling team. The next step is that the modelling team, the author, gives a domain walk-through, that is the background section of this document. The modelling team then analyses source documents such as user guides or in this case, stories. With these documents the modelling team then splits into small groups each creating a basic model of the system. Before that step occurs the chief architect may elect to create a straw man model, which is just a core design. The designs produced by each of the modelling teams is then compiled into one design by the modelling teams and the chief architect. This may be repeated until a satisfactory design is produced. Notes are also added to the design to supplement the main documents where necessary. The documents produced by this development stage are then reviewed. This will be modified to be done by one person by creating a straw man model initially and adding some detail to it including notes, the review will be done informally as it is unlikely the author will have any modifications. A lot of detail though will be added during the iterations. The primary output of this stage is class diagrams focusing on what classes are within the domain, how they are connected and what the restraints are between the classes[13]. Secondary outputs can be design justification and sequence diagrams.

### 1.3.3. Feature List

The next step in FDD is to create a feature list. The author elected to do this before creating the design because it seemed a more logical step to place first since it is more abstracted from the implementation than class and sequence diagrams. The first action in creating the feature list is to create the feature list team. The team should consist of the chief programmers from the previous stage. The feature list is then created by splitting the system into domain groups and decomposing the design into areas that comprise business activities. A feature is presented in the form "<action> <result> <object>". An example of this is "Create hardware to interface with the sensor". Features also must take no longer than two weeks to complete, if they do then they must be split into sub-features. It isn't expected that features go so low as to define getters and setters. The feature list is then assessed by the feature list team and the customer. The output of this step is a feature list split into subject areas.

### 1.3.4. Plan By Feature

The third step in FDD is to plan by feature. This is where the order of tasks in the feature list is decided and completion dates assigned to each feature. As with the other steps the first sub-step in planning by feature is to form the planning team. This team consists of the development manager and the chief engineers. The first sub-step is to determine the sequence and attach completion dates to each feature. These dates should only be as accurate as a month. There are a number of factors for deciding the order in which to complete features. These are:

- Dependencies between features.
- Balancing load across class owners. This will not be considered in a single person project.
- Risk management. Bringing forward high risk or complex features.
- Consideration of external milestones. In terms of this project this will not be considered as there is only a single delivery date.

There are also other sub-tasks defined for this task, one of which is assigning business activities to the chief programmers and assigning classes to developers. These tasks are irrelevant for a single person project. A self-assessment should be carried out on the output of this step, the feature list with dates. This will be an informal check that there won't be dependency issues and that time scales will work.

### 1.3.5. Per Feature Activities

After the initial planning stages there is then a set of tasks which is repeated for each feature. Sometimes it may be the case that multiple features are completed in a set as they use the same classes, this depends on what the chief programmer chooses to do with the features assigned to their team.

#### *Design By Feature*

As with other steps the first thing done is to form a team, a feature team in this case, consisting of the relevant class owners as decided by the chief programmer. The domain expert may then give an overview of the domain area to the feature team, this is an optional step. Another optional task is then to study external documents, for some features such as simple ones this is not necessary. The planning team should then create sequence diagrams for the features to be designed and any alternatives or notes should be included. The chief programmer then uses this to enhance the original object model. This can include updating of any initial documentation such as class diagrams and

sequence diagrams. The feature team then adds method and class prologues for their relevant classes. Lastly a design inspection is carried out by either the feature team or with other project members depending on what the chief programmer decides. The output from this task is the improved documentation.

### *Build By Feature*

Each member of the feature team now implements the classes necessary to meet the design proposed in the previous step. The code is then reviewed by any or all of the feature team and the chief engineer. The code is then tested using unit tests and the chief engineer decides if any feature tests are required.

# 2. Design

## 2.1. Overall Architecture

The design for this project is both hardware and software. The initial design started with some prototyping which will be detailed in implementation. And there was also the addition of a second Arduino on the sensor board, this is shown in the following block diagram of the final system.

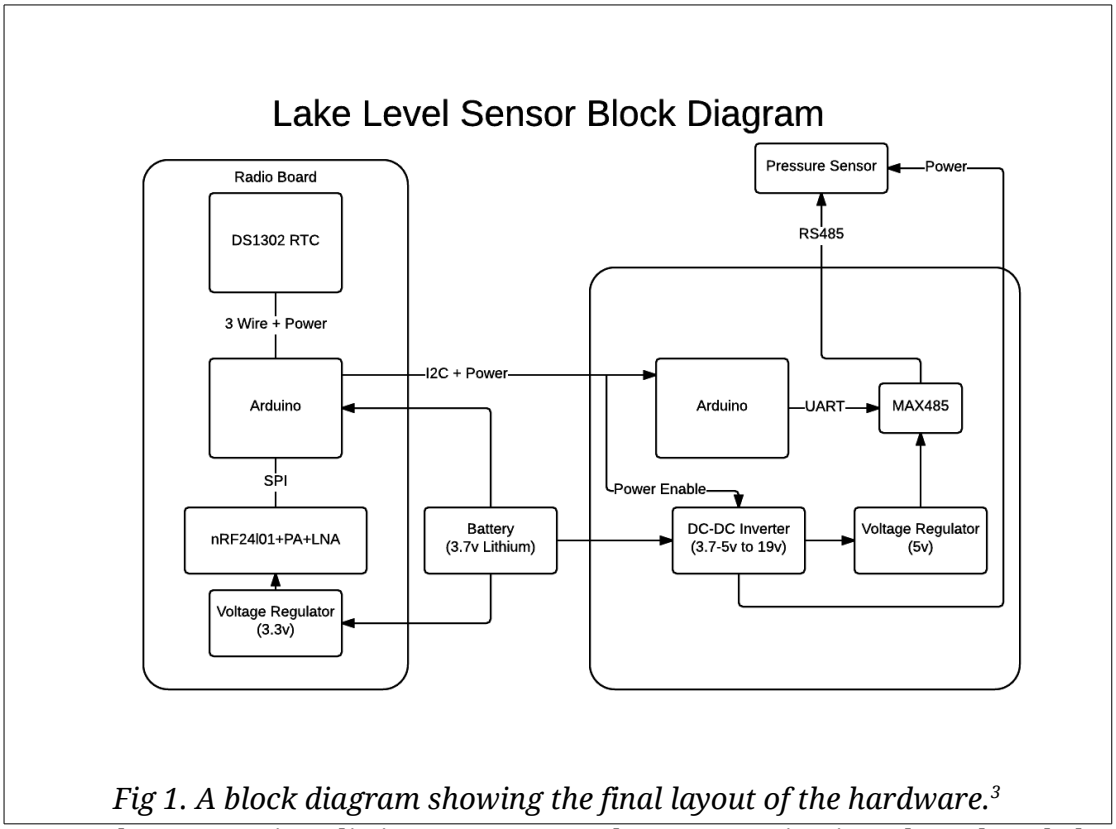



*Fig 1. A block diagram showing the final layout of the hardware.*[3]

The system is split into two parts, the communications board and the sensor interface board. The communications board is designed to operate as a router when no sensor interface boards are connected to it. Simple sensors such as a 1-wire temperature sensor could be directly connected to the router. Timing sensitive communications require a separate Arduino though as the interrupts are taken by the SPI communications with the nRF24l01. The serial interface on the radio Arduino is used for debugging and configuration so connecting sensors to it makes the system far more difficult to debug. This is the other reason that a second Arduino was chosen for interfacing with the pressure sensor as Modbus is very timing sensitive. Circuit diagrams will be included as an appendix to this document. The block diagram in Fig 1. gives enough of a hardware summary that further detail will not be required for a good understanding of the system.

The radio board portion is where most of the code will be stored. This section will handle incoming packets from the gateway, manage the mesh and respond to requests. It will also be in charge of power management, disabling the sensor board and sleeping for set times. From a software perspective the code is split into libraries for sleeping, parsing EMonCMS packets and configuring the node. There is an overarching sketch which interfaces with the radio, retrieves the value from the sensor and converts it to pressure in kPa.

3 3 Wire is a custom protocol for this RTC that can be done any any IO ports at low speed[32].

This sketch also controls whether the device should be in configuration mode which is decided by a switch on the board. This board also makes use of several Arduino core libraries SPI, EEPROM and Wire[16]. It also makes use of RF24, RF24Network and RF24Mesh by TMRh20[10]. For sleeping the Time[19] and DS1302RTC[18] libraries. For encryption it makes use of the DES library[20]. Several other encryption mechanisms and libraries were considered, primarily AES-128. This library was chosen over others because it's been implemented on both Arduino and Raspberry Pi, the respective hardware for nodes and gateway.

The sensor board will poll the pressure sensor when it is enabled and will respond to requests over I2C from the primary board for this raw sensor reading. The board will also do voltage shifting as necessary to power the 5V MAX485 and the 9-30V pressure sensor[15]. The software on this board uses the Arduino Wire library[16] for interfacing over I2C with the radio Arduino. The Modbus implementation is very basic and only has the command implemented to read the sensor value. The Modbus libraries for Arduino are not particularly well written. The most well maintained one contains a timing bug where it does not wait after setting transmit enable to high before beginning transmission[17]. The original intention was also to have Modbus done on the radio board, and because of the limited code space a more simple implementation was chosen to save space.

The third piece of work done is a configuration user interface and debugging tool. This is a simple Java GUI that interfaces with the radio board over serial when it's in configuration mode. This piece of software was not initially planned because the original intention was to have as much as possible configure dynamically over the mesh. This fell through for several reasons though, primarily that there is a library limitation in RF24Mesh that means it has to have a static address, this is configured at start-up and is unique per node and it cannot communicate without it. From this the GUI was developed as a debug tool to set this address, synchronise the RTC and fetch the available amount of RAM. This tool further developed as it became necessary to calibrate the pressure sensor and use pre-shared keys for encryption. As such, this program can synchronise and read the RTC, get free RAM, set and get EEPROM values, load a binary file to EEPROM, calibrate the pressure sensor and retrieve the depth in meters.

## 2.2. Detailed Software Design

As mentioned in Overall Architecture the software is split into three programs. The following sub-sections will be detailing what each of these programs do and their structure. They will also detail the tools used to create each program. The design detailed here is the final design from FDD. Most design decisions will be documented under Implementation, some will be covered in here.

### 2.2.1. Sensor Board

The simplest program is the software running on the sensor board. It consists of a single class and an Arduino sketch which handles I2C, initialises the PTMNRS485 class and polls the sensor for the current reading through the PTMNRS485 class. The sketch also outputs the read value to the serial port. Arguably the RS485/Modbus portion of this code could have been done in the main sketch file too. However, putting it in a separate class allows it to be made into a library, and as previously mentioned in this report the original intention was to have the PTMNRS485 class on the radio board. This wasn't possible because of timing issues though. There are also other iterations of this

class that use hardware serial instead of software serial in an attempt to get this to run on the radio board, this was unsuccessful.

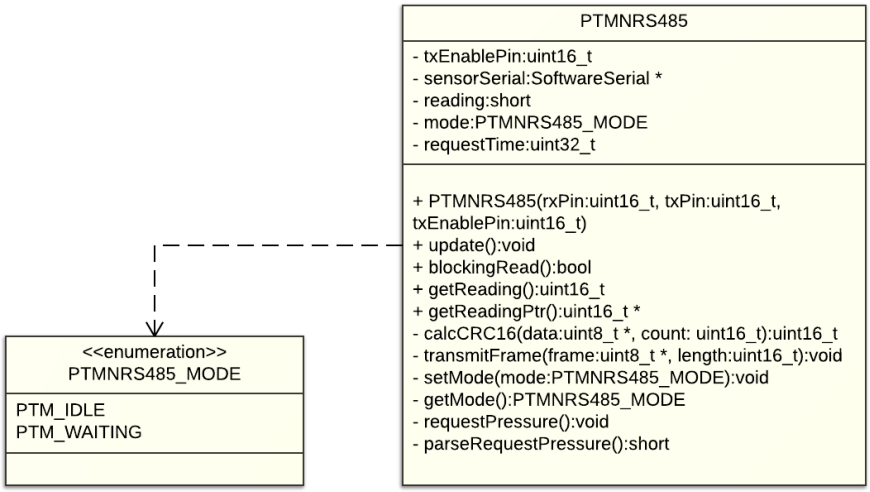



Fig 2. Class Diagram for the sensor board software.

### 2.2.2. Radio Board

The radio board consists of the main sketch and the following sub-elements:

- EMonCMSLib. This is the code for parsing incoming radio packets, creating packets and responding to packets. It also contains the code for registering the node and its attributes.
- SerialEventHandler. This code is only used in configuration mode and is used to do initial configuration of a node and some debugging.
- Sleep. This code loads sleep times from the EEPROM and gets called regularly, shutting down and waking the node at appropriate times.
- Debug header. This header file contains the definitions for enabling and disabling debug messages on both Linux and Arduino.
- Definitions header. This contains global constants such as the EEPROM map and pin map.

Each of these elements is then pieced together by the main sketch. The main sketch also does initial hardware set-up such as setting pin modes, turning hardware on and initialising external libraries. Encryption is also handled in the main sketch. Each of these classes will be explained separately and presented in individual class diagrams and there are no interdependencies between the classes.

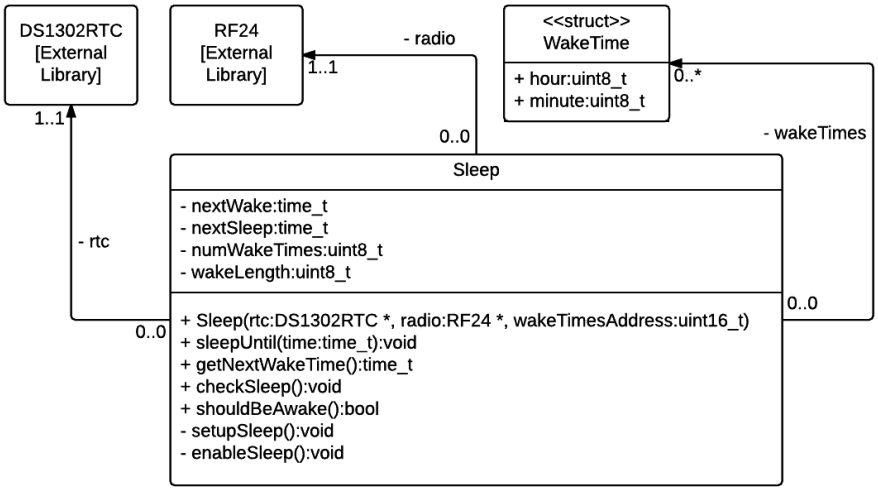



Fig 3. Class diagram of sleep class.

The sleep class is responsible for enabling and disabling all peripherals and powering down the Arduino on the radio board. The sleep times are stored in the Arduinos EEPROM and set through the configuration program. If there are no alarms then the device is assumed to be constantly awake as this may be a feature required of nodes which only operate as routers. Alarms are stored in the EEPROM as an array of the WakeTime structure. The hour field of this structure can be 0-23 and also 255 for the case that an Alarm may want to operate hourly. If the Sleep class detects it is time to sleep then it shuts off sensor boards, radio and RTC. It then puts the Arduino into a sleep mode with wake activated by the watchdog. The maximum time for this watchdog is 8 seconds, so every 8 seconds the class checks whether it should be awake and then sleeps again or enables all the peripherals and returns from the checkSleep method.

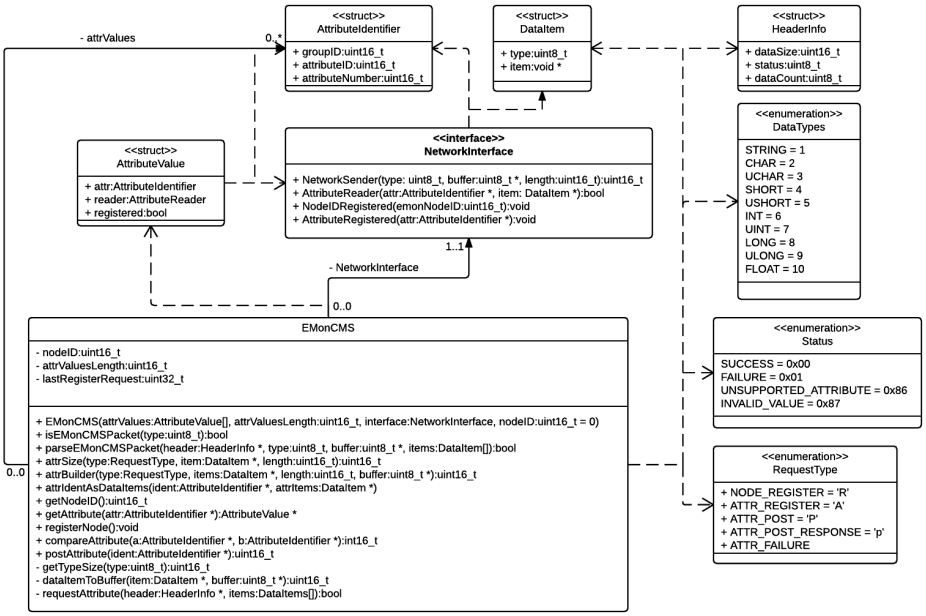



Fig 4. EMonCMS library class diagram.

Fig 4. Shows the class diagram for the EMonCMSLib portion of the radio board program. This program parses incoming requests for attributes and requests values from registered callbacks. The callbacks shown in the diagram are a C++ interface, this isn't true, as for compatibility with Arduino sketches it uses C function pointers for callbacks in the actual implementation. The library is designed to be radio agnostic which is why there are no references to the RF24 radio. DataItem and the values in DataTypes are defined in the Low-Power Radio Communications Specification[4]. Attribute reader callbacks are registered when the library is initialised and saved into internal storage. The class is capable of sending all required types of message including node ID requests, attribute registration requests, attribute posting and responding to attribute requests. There are wrapper functions around these basic requests such as registerNode, which requests a node ID and then when this is successful registers the attributes. Another example is the postAttribute method which calls the attribute reader for a given attribute and then builds a post request and sends it to the gateway. There are callbacks for when a node ID request has succeeded and an attribute has been successfully registered so that the registration status can be saved to EEPROM and not repeated on each Arduino reset.

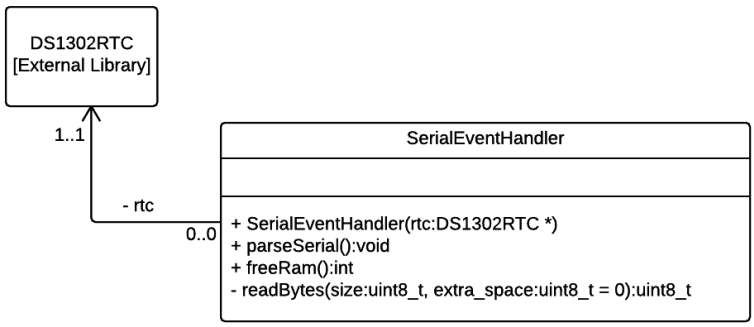



Fig 5. SerialEventHandler class diagram.

Fig 5. Shows the SerialEventHandler class. This is a simple class that reads serial data and parses it into various requests in a switch statement. This is the Arduino half of the configuration system. It supports synchronising and reading the clock, reading and writing EEPROM, getting free RAM, getting the raw pressure reading, setting calibration values for the sensor and getting the depth in meters. The method for getting freeRam is accessible publicly to allow it to be logged in other formats.

### 2.2.3. Configuration Software

The configuration software is a program written in Java which interfaces with the Arduino over serial. It uses the JSSC library for cross-platform serial communications[21]. It also uses the SimpleRegression class for the Apache Commons Math library[22] for calibrating the depth sensor. The calibration works by requesting raw sensor readings from the depth sensor. The user then enters the depth in meters in the GUI, this is then fed into the SimpleRegression class which calculates a line of best fit. After several repeats the SimpleRegression class outputs a gradient and y-intercept value. These values are then copied as floats over to the radio board. The depth sensors are sensitive enough that they pick up changes in atmospheric pressure based on elevation, so there is also a feature to set the base pressure. Fig 6. Shows the class diagram for the configuration software. An alternative approach to configuration could have been to have it Arduino side and use a serial console. This has the advantage of being more platform independent as it can be used from any system with a serial console, however it also means using up limited program space and having something which is less user friendly. It also would have been difficult to upload shared keys, they would have had to be entered manually in hexadecimal.

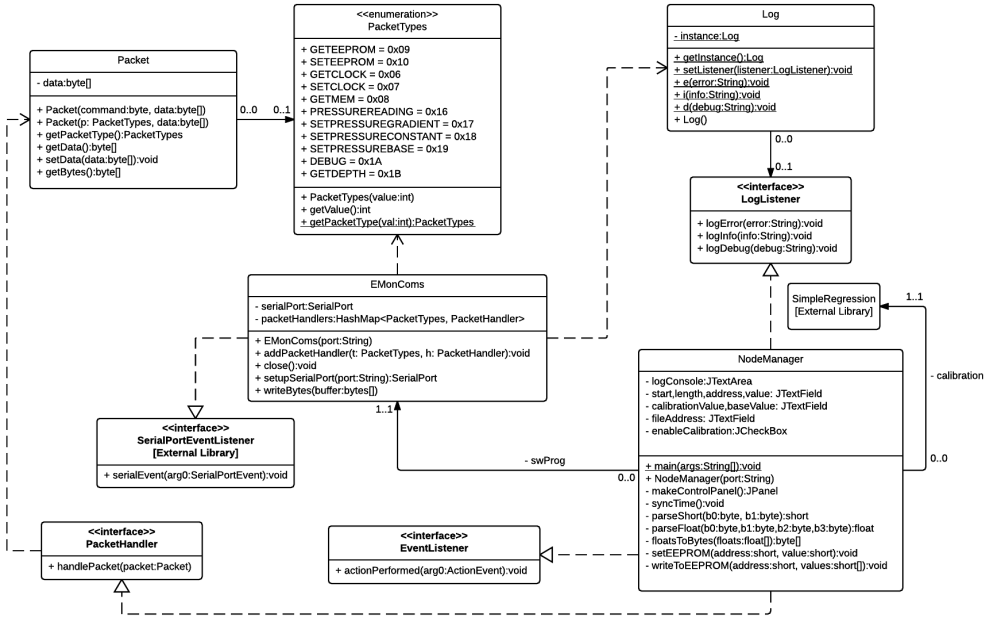



Fig6. Class diagram for the configuration program.

## 2.3. Tools Used

The operating system used to develop on was Ubuntu 14.04 Linux, the tools used in this project however are platform agnostic. Linux was chosen for the comfort of the author using command line tools such as Git.

The tool used most throughout the project is the Git version control system. This is a highly configurable version control system capable of distributed and centralised repositories. It also has very good branching support compared to other version control systems such as SVN. In this project Git[24] has been used in a centralised manner with repositories being pushed to a single server and then periodically backed up to various computers using the Bittorrent Sync tool[23].

For the configuration program portion of the project, the Eclipse IDE[25] was used for development in the Java language. Other IDEs could have been used for this but the author was most comfortable with Eclipse. Java was chosen as it's a very quick language to develop in and has very good debugging support and is hypothetically write once run anywhere. In reality the library I've chosen for serial communication only has Mac, Linux, and Windows support. The configuration program also used Git for source control.

The programs on Arduino were developed using UECIDE[26]. UECIDE is a fork of the default Arduino IDE offering a much easier user experience for code browsing, highlighting compiler errors and code editing. Git was also used for source management for the two Arduino programs. In addition to these the EMonCMS Library was designed to be compiled on Linux for unit tests and makes use of GCC, make and the Geany text editor for code editing.

# 3. Implementation

At this point of the project there was no design other than the features list. For each selected feature or feature set there was a prototype phase where hardware was breadboarded or software hacked together. This was then put to the side and a design consisting of class diagrams created and then the software implemented. The testing is sporadic at best because most of the code is on the Arduino. Some is tested on Linux, most testing however is functional rather than unit driven, unlike is often the case in FDD. The configuration program is the exception to this, prototype, design, build, test cycle as this was unplanned and created as a tool to aid development. This was developed and expanded throughout the project as necessary and then a class diagram created. One thing noticed early on in development was the allocating and de-allocating memory can easily cause heap fragmentation on an Arduino, which with its limited memory can quickly lead to a crash. This is why a lot of the code does things in unusual ways to keep memory allocation stack based. Using function pointers instead of C++ interfaces are to aid compatibility with default sketch files, something also realised during the following implementation stages.

## 3.1. Iteration 1

Several features from the feature list were often worked on at a time and although done in a mostly linear fashion there is some overlap between the prototyping phase of the next feature as one feature comes to a close. The first feature selected to be worked on was creating hardware to interface with the sensor. This feature actually stretched over most of the project as refinements were made upon the initial prototype. The first implementation of the sensor interface consisted of a MAX485 on a breadboard connected to an Arduino Mega and a 12V power supply connected to the sensor.

The PTM/N/RS485 series of sensors can communicate either over Modbus/RS485 or 4-20mA. 4-20mA forms a current loop between the sensor and the receiver, this value is then read locally. This has the benefit that from a software perspective it's easier to implement however it cannot access other features of the pressure sensor and if the hardware is faulty for measuring current it may not work. RS485 is a specification for digital data transfer over long distances, it uses a differential pair and transceivers are available in half or full duplex configurations. Modbus is an addressable transport layer on top of half or full duplex RS485 that has several built in functions such as reading remote registers and setting an output state. This makes Modbus far more flexible than 4-20mA which is why Modbus was chosen to interface with the pressure sensor.

The first step in software development in this section was deciding which library to use, if any. The first library considered was the Arduino Modbus Master library[27] which as an object-orientated library with a simple API. Upon beginning to create the sketch for this library I realised the publicly accessible documents for the sensor were inadequate, for instance it doesn't contain the address of the device or details on the operations for reading the sensor value. This task was then put on hold while a request for more documentation was made to the company, during this time the self-contained communications stack was started. When the documentation arrived which does not allow redistribution the author attempted to use the Arduino Modbus Master library. Inexplicably this library did not work and so another library was used, the SimpleModbusMaster library[28]. After filling in the details, this library didn't work either. The next step in the debugging process was to make

the most basic code possible. Some of the documentation supplied by the manufacturer contained an example packet for requesting pressure data, with CRC included, using this packet and Modbus specification documents[29] a basic Arduino sketch was created with the correct timings. This sketch successfully retrieved the raw pressure reading from the sensor. With a working example and an increased knowledge of Modbus from the previous debugging it was then possible to find a bug in the SimpleModbusMaster library where there was no delay between enabling transmit and beginning transmission. Since the feature set required for reading pressure was only a small sub-set of that implemented by SimpleModbusMaster, the author elected to develop the sketch into a library which could use software based serial. From this the design in Fig 2. was created, implemented, and its functionality checked.

As previously mentioned there was a pause during the development of this first feature where the self-contained communications stack was started. The self-contained communications stack is an implementation of a binary protocol[4] based on a JSON based protocol[5] set out by CAT. The specification was developed primarily by the author with contributions from Jonathan Newman on sections relevant to encryption and key exchange. The code for this library was initially prototyped on Linux and basic functionality tested using unit tests until it was ready to be used on an Arduino. The first methods implemented involved building node ID requests. The type of a message is passed into the network sending functions of the RF24 libraries and the header for the communications stack contains the size of the incoming data, the number of items (where an item is any data type of any length [see specification[4]]) and the status according to the status codes set out by CAT. Not including the message type in the communications stack header was perhaps a poor decision because if this specification were to be applied to a radio which didn't have this type byte in the header then another communication layer would have to be in between to fill this role. The next function implemented was handling responses from the gateway, this was initially tested by a mock response to a node registration and later expanded with responding to attribute requests, posting attributes was also completed at this stage. The finished product of this is represented in Fig 4.

## 3.2. Iteration 2

The second iteration focuses around continuing the hardware prototypes and integrating the software created in the first iteration into this hardware. The feature set this refers to is “Send data back to the server through the gateway”. For the part of the system implemented in this project this involves speaking to the software created by Jonathan Newman, the gateway software which runs on Raspberry Pi.

In addition to the sensor connected to the radio boards software serial port the nRF24l01+LNA+PA is connected to the SPI port which power regulators in place to supply enough current. At this stage the DC-DC converter for the sensor had also arrived so this was added to make the breadboarded system portable. The hardware was then tested using the RF24 ping sketches and the sketch developed in the first iteration to get the pressure sensor reading. Then a sketch was created which used the communications stack to request a node ID, register an attribute and accept requests for this attribute. The attribute wasn't the pressure value to start with, it was simply the time in milliseconds since the Arduino had last reset, it was done this way so the developer doing the gateway would have something to test against. This sketch

worked with the gateway and so in addition posting data was added to the sketch and this also functioned correctly with the gateway.

The next step was to get the sketch to read the raw pressure value from the sensor, register it as an attribute and post it to the gateway. This proved more complicated than expected. After putting in the code to read from the sensor the sensor kept returning bad data, this led to trying to use the hardware serial port with the code previously written and with the SimpleModbusMaster library. None of this worked and so code was gradually commented out until it worked again and it seemed to be an incompatibility between the serial port and SPI, this is likely due to interrupts from the SPI interfering with the timing of the serial write which Modbus is very sensitive to. This led to using a second Arduino to do the work of interfacing over Modbus. This was connected to the radio portion over I2C using the Arduino Wire library. The reasoning behind using I2C over connecting over serial is for future expandability, this way it would be possible to connect multiple sensor boards. This set-up worked and the sketch running on the Arduino connected to the sensor used the software serial cut-down version earlier developed, this was used over SimpleModbusMaster to leave the hardware serial port free for debugging.

At the end of this iteration raw sensor values were being successfully read, registered with the gateway and posted to the gateway.

## 3.3. Iteration 3

The third iteration is mainly focused on developing hardware into a more stable form and creating the sleep code, this is the feature group “Power Management”. Near the end of this iteration it also became clear that a configuration tool would be necessary.

The first thing done in this iteration was to draw some circuit diagrams, sketched out on paper. Some things were changed during the actual building of the board and some simple elements such as the resistive voltage divider for measuring battery level weren't sketched out but are still on the board. The first thing made was the radio board.

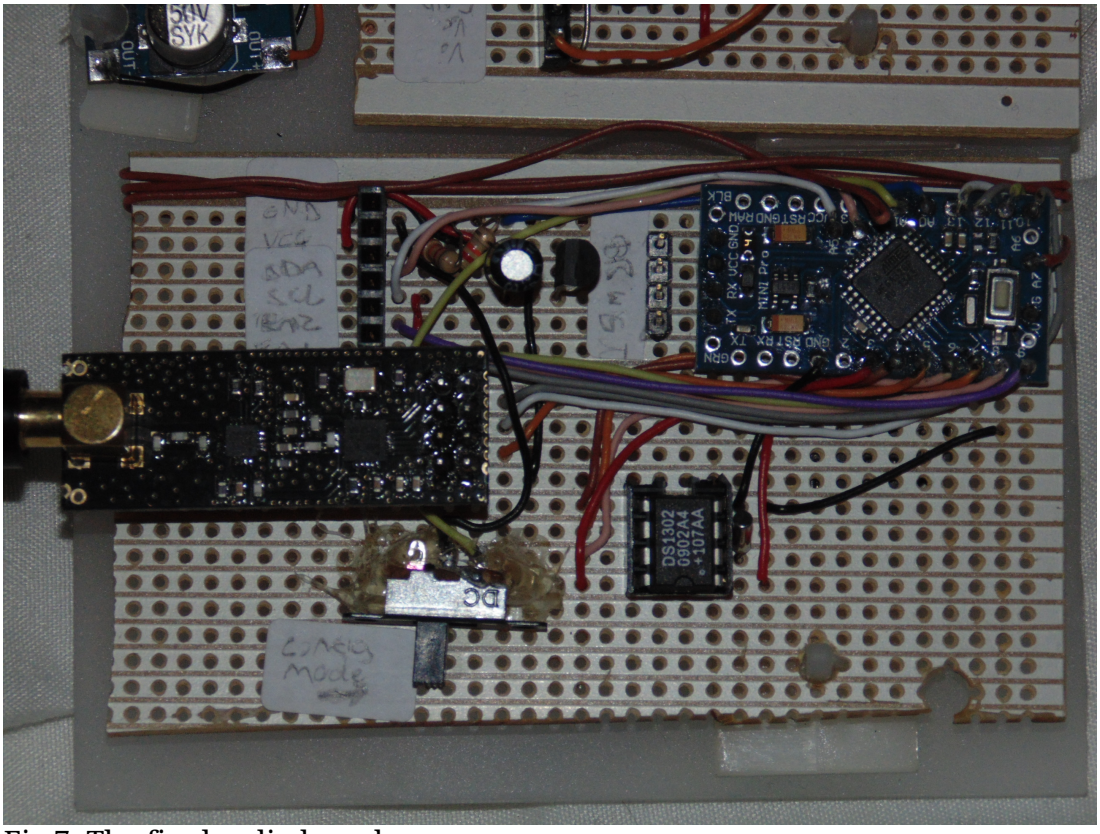



Fig 7. The final radio board.

Fig 7. Shows the final radio board, there were several improvements made to this board during development. Initially the header in the top left was going to be serial with a transmit enable pin. This was revised to be I2C when the bugs previously mentioned came to light. Just to the right of that is a resistive voltage divider, this takes the input voltage, a maximum of 5V and scales it down to 1.1V, the internal reference voltage for the ATMega328P. To the right of that is a 3.3V voltage regulator, this is there because the nRF24l01 requires a 3.3V power supply but has 5V tolerant IO pins. The reason the whole system doesn't run at 3.3V is because the ATMega328P can't run at 16MHz at that voltage, whereas anything above 3.7V is safe[30][Figure 29-1].

To the right of the voltage regulator are pin headers for programming, to the right of that the Arduino Pro Mini, this has been modified to have the power and pin 13 LEDs removed to save power, power also does not go through the on-board regulator, in testing this lowers power consumption in sleep mode to 0.6mA. The board on the centre left is the radio, below that is a switch which puts the board into programming mode, where the serial port can be used for configuration. Originally the RTC was attached by pin headers but the DS1302 breakout board was very fragile and broke off so the author elected to attach the chip, its clock and the backup battery to the board. The battery for the clock is surface mount so is attached to the other side of the board.

Alternately to implementing this on a breadboard it could have been made into a printed circuit board (PCB), but as most of the hardware would be connected to it through pin headers it adds little benefit to the time involved to develop a PCB. The brown wire attached in a coil on the side of the board is connected to an input pin on one end and nothing on the other, the purpose of this was to generate entropy for the Diffie-Hellman key exchange intended to be used.

With the RTC attached it became apparent that this would need testing. This is where the configuration GUI comes in. The configuration GUI was created by the author for debugging a different project that used similar hardware such as the DS1302 RTC.

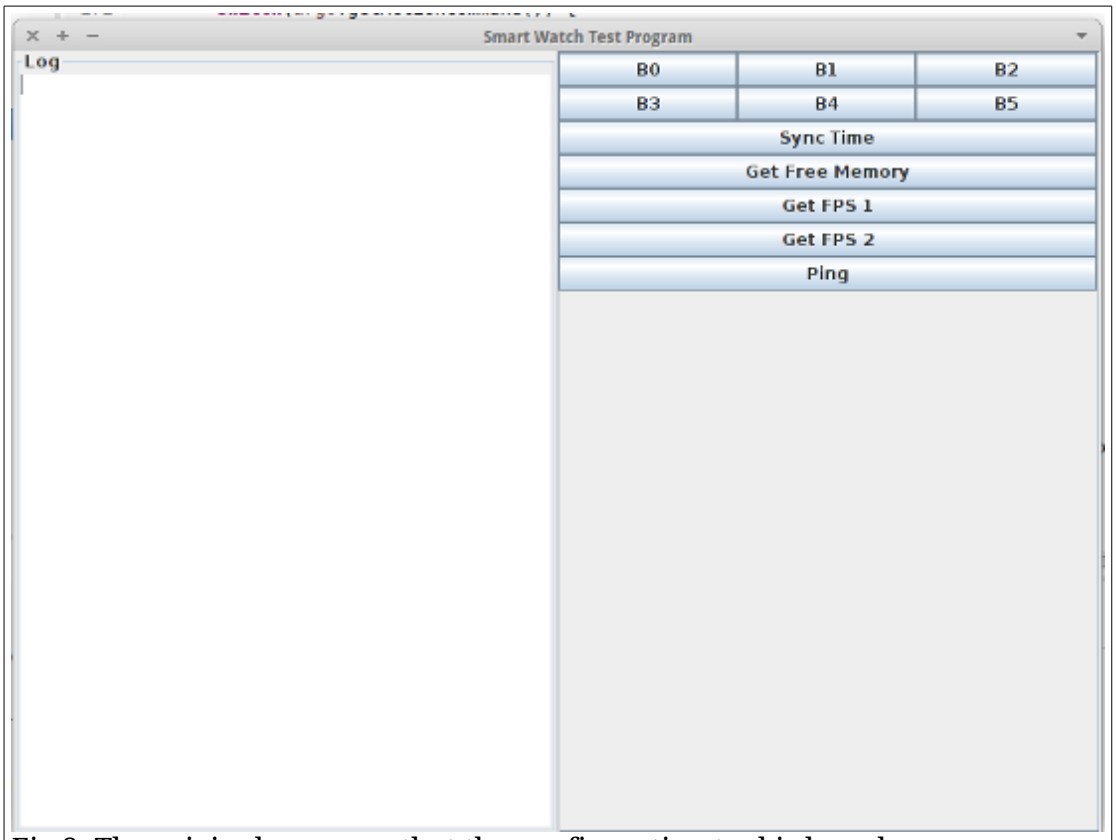



Fig 8. The original program that the configuration tool is based on.

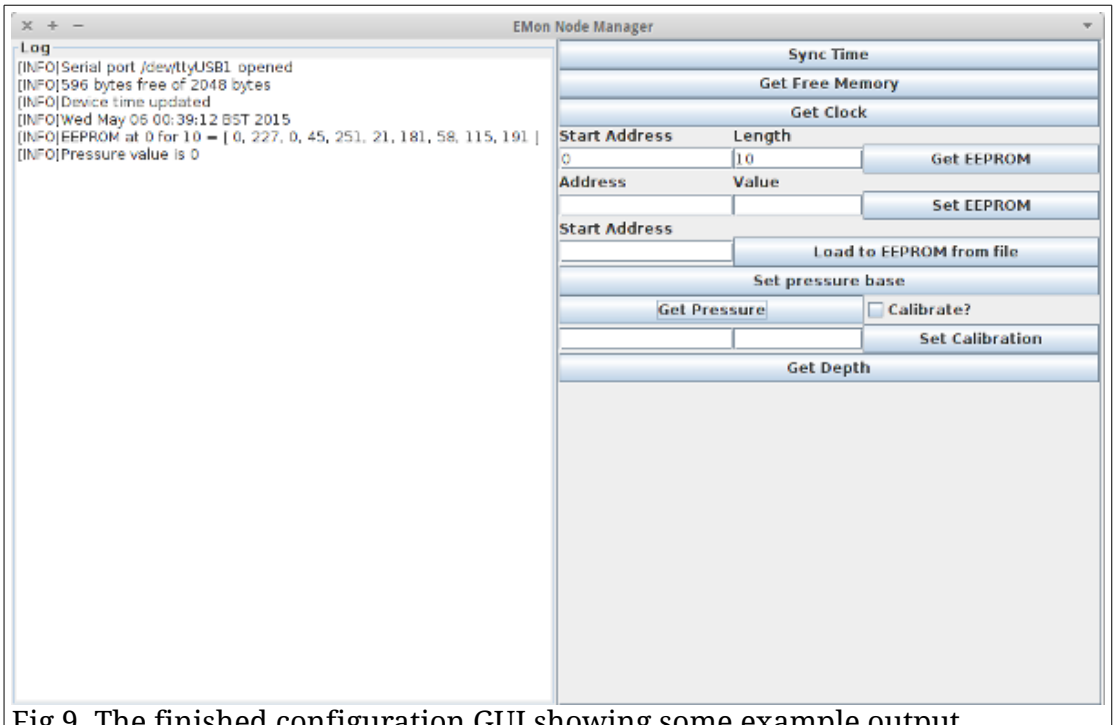



Fig 9. The finished configuration GUI showing some example output.

The configuration program sends very simple binary messages which have a header consisting of 2 bytes, the first is the message type, the second is

the data size. Following that is the data, which may consist of nothing or, for example the current time. Pressure can be calibrated by checking the “Calibrate?” button and pressing “Get Pressure” while alternating the depths and entering the depth in meters. The calibration values appear in the boxes to the left of “Set Calibration” so they can be copied in externally. The button at the botton will return the depth in meters of the sensor. The user interaction on this is not at all intuitive which can be attributed to its origins as a development tool. Given more work this GUI could be vastly improved to be an actual configuration tool. The feature that is perhaps needed most is the ability to interact with an EEPROM map and manipulate data in there in a more intuitive manor than setting single values or dumping a file to EEPROM. Various configuration values are stored in the EEPROM such as whether the node is registered, its node ID, its radio node ID and sleep times.

The final feature implemented in this iteration was the sleep modes. This was prototyped in two halves. The first half was putting the device into minimum sleep mode. This means disabling any potential external boards so setting EN1 and EN2 to low on the sensor board header, powering down the radio, switching off the RTC and then putting it into watchdog sleep. This went reasonably painlessly and was put into the Sleep class. A problem that did occur however was when resuming hardware was not re-enabling. The initial testing of the sleep code was done while the device was in configuration mode, the radio board would start, sleep and then enter programming mode. The getting and setting time features no longer worked because the RTC was still switched off, this was a quick fix though.

The second half of the prototype was to create the alarms. The first step in this was figuring out how to store the alarms, initially considered was to have alarms that repeat daily at a set hour and minute, this however is a poor use of EEPROM when the device is likely to be waking multiple times every hour. The solution to this was to have a special value for hour, if this value is set then it is considered as every hour. The class is designed to be used by setting it up on device start-up, loading the alarm times from EEPROM and then calling a blocking method once per loop, if it is meant to be asleep this function disables all peripherals and sleeps until the next wake time and then returns from the method. The SerialEventHandler code is also based upon code from the same project but modified to add support for the new features.

After implementing sleeping the next thing to do was to create the final version of the sensor board, with the I2C rather than serial interface.

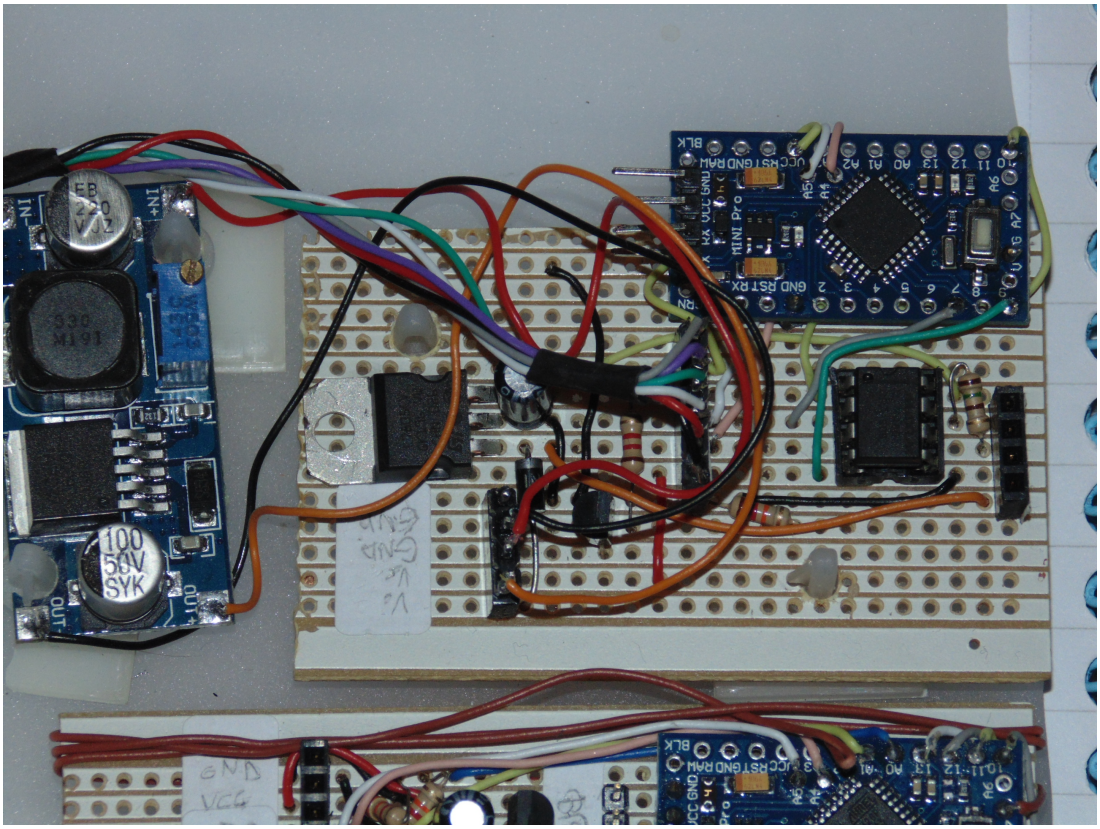



Fig 10. The final hardware for the sensor board.

Fig 10. Shows the final hardware for the serial interface with the extra Arduino, power controls, regulators and converters. Power goes first into the Arduino and a transistor. From the transistor it goes into the DC-DC converter, from there to the sensor and the 5V voltage regulator and that feeds the MAX485 chip. This hardware was far more problematic than the radio board for minimising power consumption. The end result was 8mA after several improvements. Originally the Arduino on the sensor board was connected to the 5V voltage regulator but then it still drew power when disabled which is why it's fed directly from the other Arduino.

There's also the addition of a diode before the DC-DC converter this also dramatically lowered power consumption by about 10mA. Somewhere in the circuit there is still a circuit forming and drawing power despite the transistor set to an off state. However, even if the device was drawing 9mA constantly this would provide 222 hours of runtime with a 2000mAh lithium battery, this means it's still capable of being solar powered and meeting the requirements. The radio board is far better for power consumption, this does mean the routers will be very power efficient. A solution considered for the power leakage to the sensor board was to use relays, this does have the problem of a lot more circuitry to avoid a bug rather than solve it and relays use power while on.

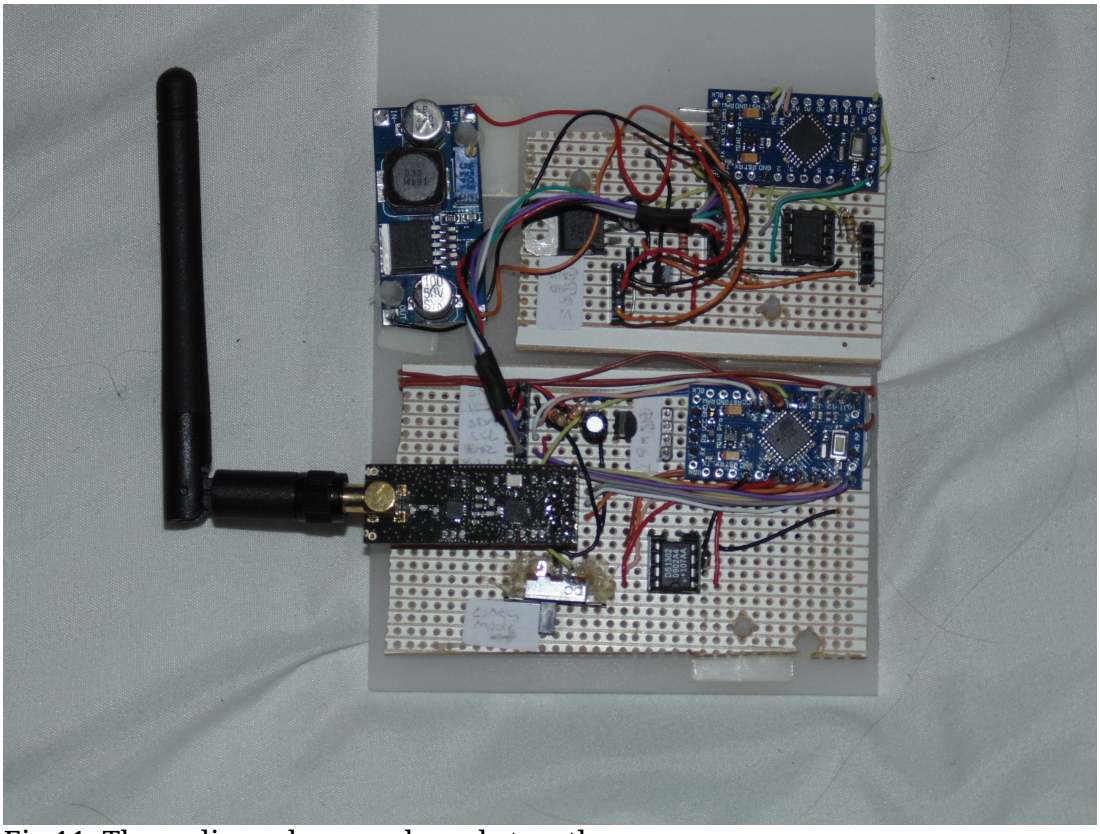


Fig 11. The radio and sensor boards together.

The system was then connected together and tested against the gateway for successful registration and posting for pressure values. Connecting the two boards together caused no issues. An alternative approach to the hardware could have been to have the boards combined into one but that doesn't allow for a modular approach where hardware can be re-used as simply.

## 3.4. Iteration 4

The fourth iteration is mainly finishing touches, adding encryption and adding pressure sensor calibration. The pressure sensor calibration was the first thing to be implemented, followed by encryption, to an extent.

Initial attempts at sensor calibration used just two or three points into a set formula taking the average gradients between the points, this works roughly and produces reasonable calibration data but it leaves a lot of space to be improved. The Apache Commons Math library includes a class called SimpleRegression this uses the least squares method to calculate a line of best fit much more accurately than the authors original model. This calibration is done in the configuration program to keep minimize code usage on the Arduino. The biggest problem with calibration is that if it is calibrated at sea level and then carried up 100m or so the calibration becomes invalid and it needs to be recalibrated. The attempt to fix this was adding a displacement to the read sensor value based on a base pressure and its difference between the x-intercept produced by the calibration data. In testing this has proved unpredictable and can produce negative numbers. This isn't a huge issue though as it can be calibrated at the installation site, such as CAT's lake. This depth in meters is then converted to kPa using a constant supplied by CAT.

Encryption was planned to use Diffie-Hellman key exchange and AES-128, this completely fell through though. The first cause of this is Diffie-Hellman, doing modulus or arbitrary number arithmetic with the limited memory of and Arduino to the degree Diffie-Hellman asks for is difficult to

impossible to do in a meaningfully secure way, at least the author was unable to find a work around to this problem. The author decided to use pre-shared keys instead, installed on a per node basis during the initial necessary configuration of each node, this is why there is an option to load files to EEPROM, to load key files. From here development proceeded to use the pre-shared key and AESLib[31].

The system at this point was fully working with AESLib although not fully tested, upon attempting integration with the new version of the gateway supporting encryption packets were decrypted jumbled. The first thought was that perhaps one of us was using CBC rather than ECB, CBC is cyclic block chaining and is more secure when sending multiple blocks, ECB is much better for single blocks. It did turn out we were using the wrong ones so both switched to ECB which is simpler because it doesn't require an initialisation vector, the messages were still being incorrectly decrypted. Debugging messages were then added to ensure that the messages being sent were the same as the messages being received, which also turned out to be the case, so the problem still existed.

The next step which was attempted was to use a different encryption library since the gateway and the node were using different AES implementations. The library which became obvious, and is almost as secure as AES-128 is an implementation of Triple-DES called ArduinoDES[20]. This library uses 112 bit keys rather than 128 bit, making it only marginally less secure. This library runs on both Arduino and Raspberry Pi, which is why it was chosen over other libraries. Upon testing this library though in place of the AES library the encryption still did not function, and as of the end of the project still does not.

## 3.5. Review

To summarise all features, except encryption have been implemented, there is still room for improvement on power management though. Following is a copy of the feature list with comments added reviewing each implementation.

- Read Lake level
  - Create hardware to interface with the sensor. A final implementation was done on strip-board after overcoming the timing difficulties with Modbus.
  - Convert sensor reading to pressure in kPa. This was done in two stages, converting to depth and then applying a constant to convert to pressure.
- Communications
  - Self-contained Communications Stack. This was done in the early stages and tested on Linux.
    - Register nodes to the gateway.
    - Register attributes to the gateway.
    - Post values at a specified interval to the gateway.
    - Respond to requests for values to the gateway.
  - Send data back to the server through the gateway. Successful communication between the gateway and the radio was established and data sent to the gateway.

        - Interface radio with the Arduino.
        - Build interface layer between radio and communications stack.
    - Encrypt communications. This was attempted but was unsuccessful, the hypothetical implementation is still included in delivered code, it is unknown whether the issue is with the gateway or node.
- Power Management.
    - Create hardware which runs from non-mains power. This is done and the only two flaws are that the battery level monitoring cannot be relayed over the communications specification given by CAT and the sensor board draws unnecessary power when disabled.
        - Create circuits using low-power components. A parts list and circuit diagrams are included as an appendix, for example showing the choice of voltage regulators to have a low quiescent current.
        - Create code for going into low-power modes. This works really well on the radio board, significantly lowering power consumption.
    - Create code for sleeping for pre-defined intervals. This is successfully done and has undergone basic functional testing but should have more extensive testing done.

# 4. Testing

## 4.1. Overall Approach to Testing

The approach to testing during this project is primarily functional as unit tests on an Arduino system are not the easiest to implement. While they can definitely be done by creating sketches which act as a test harness, running them would require uploading each test suite to the Arduino. With this system most components of it are quite interdependent, meaning that if any part does not work then the whole system will show the result. During development of individual modules there has been informal testing where the behaviour of each module and its debug output is monitored to see if it functions are expected. The communications stack also has a limited number of unit tests which test basic functionality, the purpose of these though was more to aid in development by mimicking the radio. The bulk of testing is functional, these are tests which cover system behaviour on a higher level.

## 4.2. Unit Tests

The communications stack uses some unit testing to test a few features through imitating the radio, no other part of the system uses unit tests. The configuration program was developed as a debug tool originally, which is the main reason it has no tests, it's also a very simple program that wraps the serial library it uses and provides a UI. The tests for the communication library don't use any testing suite because the tests are very simple. These tests are:

- Testing node register request generation.
- Testing generation of attribute register requests.
- Testing posting an attribute value.
- Testing generation of attribute posts.
- Testing receiving a EMon node ID.
- Testing responding to an attribute post request.

## 4.3. Functional Testing

Most testing as states is functional, these are detailed in the following test tables. The results in this test table represent the final delivered system.

### 4.3.1. Sensor Board.

The sensor board is quite easy to test, the software on it will output either the sensor reading or a failure message over the serial port. The set-up for this test involves disabling sleep (see EEPROM map in appendix), connecting the power and ground to the main board and the TX/RX cables of the serial cable to the sensor board Arduino. The sensor board and radio board must be connected together and the sensor connected to the sensor board.

| Test Number | Action | Expected Result | Pass / Fail |
|---|---|---|---|
| 1 | View the serial output with the sensor connected while lowering the sensor | The value printed to the serial port should increase. | Pass |

| Test Number | Action | Expected Result | Pass / Fail |
|---|---|---|---|
| | into water. | | |
| 2 | Disconnect the sensor and view the serial output. | The serial output should read “Sensor read failed” every couple of seconds. | Pass |

#### 4.3.2. Configuration Mode

The tests in this section refer actions carried out by the configuration program and their results on the radio board. The set-up requires the radio board, sensor board and sensor connected. The radio board should also be connected to the PC through a serial cable. The gateway does not have to be online. The configuration tool does not test for failure conditions such as out of range values, these will not be tested.

| Test Number | Action | Expected Result | Pass / Fail |
|---|---|---|---|
| 1 | Launch the configuration program. | The configuration program prompts the user to select a serial port, the port for the Arduino is listed. | Pass |
| 2 | Select the Arduino serial port and click “Ok”. | The program launches into its main screen. | Pass |
| 3 | Click the button “Get Free Memory”. | The log should show the number of bytes remaining of the total memory. | Pass |
| 4 | Click the button “Sync Time”. | The log reports “Device time updated” | Pass |
| 5 | Click the button “Get Clock” | The time and date of the host machine should be output in the log window. | Pass |
| 6 | Click the button “Get EEPROM”. | The log window should show the EEPROM values from the address for the length specified. | Pass |
| 7 | In the “Address” and “Value” fields to the left of “Set EEPROM” enter the respective values of 300 and 123. In the “Address” and “Length” fields next to “Get EEPROM” enter 300 and 1. Click “Get EEPROM”. | The log outputs that the value at EEPROM address 300 is 123. | Pass |
| 8 | Create a binary file with the bytes 0x00, 0x01, 0x02, 0x03, 0x04. For the “Address” field | The values output match the test data entered in the binary file. | Pass |

| Test Number | Action | Expected Result | Pass / Fail |
|---|---|---|---|
| | next to “Load file to EEPROM” enter 100, then click load file. Navigate to the file and open it. Get the EEPROM at address 100 for length 5. | | |
| 9 | Click the “Get Pressure” button. | A non-zero value related to pressure of the sensor should be output to the log. | Pass |
| 10 | Check the “Calibrate?” check-box and click “Get Pressure” at various depths entering the depth into the GUI when prompted. | The two boxes to the left of “Set Calibration” show gradient and y-intercept calibration values. | Pass |
| 11 | Click the “Set Calibration” button with the previously calculated calibration values. | The log outputs that the gradient and y-intercept are being set. | Pass |
| 12 | Remove the probe from the water and click “Set Pressure Base”. While the probe is still out of the water click “Get Depth”. | The log should show the depth in meters and it should be close to or zero. | Pass |
| 13 | Put the pressure sensor into the water at measurable depths and click “Get Depth”. | The depths reported by the sensor and GUI match that of the actual depth. | Pass |

### 4.3.3. Communications

The tests in this section focus on the overall operation of the system. The set-up for this test requires a radio board, not in configuration mode, connected to a sensor with sensor connected. A serial cable should be connected to the radio board serial and this connected to a terminal. The set-up also requires a Raspberry Pi with radio running the sensor gateway program by Jonathan Newman without encryption enabled. Tests in this section are difficult to break down and it boils down to checking for certain output in the program outputs.

| Test Number | Action | Expected Result | Pass / Fail |
|---|---|---|---|
| 1 | Using the debug program set the reset EEPROM flag and then | The gateway receives and responds to a “RegisterMessage”. It then | Pass |

| Test Number | Action | Expected Result | Pass / Fail |
|---|---|---|---|
| | restart the Arduino to non-programming mode. Then start the sensor gateway. Monitor the output of the sensor gateway program. | receives and responds to at least one “RegisterAttrMessage”. Then every couple of seconds it should print “PostMessage received” and the messages data. | |
| 2 | Using the configuration program, enable encryption using the same shared key for the gateway and node. Repeat test 1 with this enabled. | The same expected result as 1 with the addition that the gateway reports the incoming encrypted message and the correctly decrypted message. | Fail |

#### 4.3.4. Sleep

This test section covers sleeping. The set-up should be the same as the communications functional tests.

| Test Number | Action | Expected Result | Pass / Fail |
|---|---|---|---|
| 1 | Using the debug program set the reset EEPROM flag. Configure the sleep as documented in the EEPROM map to wake for 5 minutes several minutes in the future. Then reset the Arduino into non-programming mode. Then start the sensor gateway. Monitor the output of the sensor gateway program. | At the time specified for waking the gateway receives and responds to a “RegisterMessage”. It then receives and responds to at least one “RegisterAttrMessage”. Then every couple of seconds it should print “PostMessage received” and the messages data until the 5 minutes has elapsed and then it should stop receiving. | Pass |

## 4.4. Acceptance Testing

Following the completion of the project and the final demonstration it is intended that the projects created for CAT are tested and demonstrated on-site to the staff of CAT. This will involve setting up the system at the lake and either carrying the gateway away and demonstrating it's still receiving values or setting up a makeshift repeater, or possibly more than one and demonstrating that lake values are read correctly.

# 5. Critical Evaluation

This final section will look back at how well the development process was followed, identify whether the stories were correctly interpreted into features, how well these were implemented. It will also look into whether it meets CAT's needs.

## 5.1. Process

The development process chosen for this project was FDD, it wasn't followed well but it was a basis for a process that emerged during development that seemed to take on aspects of other Agile methodologies such as the refactoring of Extreme Programming and the daily stand-ups of Scrum which would be me talking to a house-mate about my progress.

The initial iteration of FDD is where it was followed most closely, where source documents were analysed and a feature list created. This though was not exactly FDD as FDD specifies that before creating a feature list, sequence diagrams are created, these are emphasised in various documents describing FDD. I find that sequence diagrams do little to help my understanding of a system when entering into someone else's code, this is why I don't think they are worth the time invested into creating them. Without these though there was very little up-front design and I don't think that stuck very closely to FDD, other than the block diagram created during the early stages of the project. The feature list should have been sent to the client too during early stages, something which I neglected to do and should have because this could have potentially helped to avoid any ambiguities in CAT's original stories.

During the iterations FDD was followed less and the process leaned more towards the concept of emergent design with prototyping an area of the system to a limited or fully functional if messy state, creating a design from this mess and then refactoring the (or re-designing circuits) based upon this new design. This seemed to work well with the idea of a feature list, where parts of the system were broken down and created separately before being integrated together, this meant the design of each of these components never became too complex. Design outputs from these iterations was in the form of class diagrams or hand drawn circuit diagrams with attached notes, this still is not quite the level asked for by FDD but I feel it is adequate in explaining the system.

## 5.2. Requirements & Aims Retrospective

I don't think the stories for this project were particularly well written, there are ambiguities such as "the specified format" which was still being developed when this feature list was released. If the feature list was based solely on this I think there would have been a lot of things missing. Early during development though some extra documents were supplied which defined these formats and gave a much deeper level of understanding of the system. I think the feature list did work well and was reasonably accurate at least at the lowest level of functions, maybe the groups needed to be worked on a little though because some of them had some overlap in functionality. I also think my choices of which features to work on during which iterations could have been better.

The big feature which didn't make it to completion was the encryption, this isn't a problem for a functioning system though, it is just less secure. Also time synchronisation ended up not being implemented, despite being defined.

The RTC's have a habit of drifting by a few seconds a week which can cumulatively cause problems for the synchronised sleep, and for a long-term installation this would need to be implemented.

## 5.3. Tools

I'm very happy with the tools I chose to develop this project, the main tool I used, UECIDE has been excellent throughout development and not something I had used before this project. A particularly notable feature was its ability to internalise libraries to a project. This means I can give the end result to CAT as a well organised folder containing all the libraries they need in a format that can be quickly opened. The code browser is also amazing compared to the original Arduino IDE, listing functions on the browser along with grouping headers, sketches, and source files. I did consider for a time using Eclipse with plug-ins to allow AVR development because even if UECIDE is a good IDE, it is certainly not as good as Eclipse. I decided against this as I had previously attempted to set this up and it was unsuccessful and took quite a lot of time, which would make it very annoying for CAT compared to following a simple installation.

There were other tools used during this project, such as Eclipse which is an excellent choice for Java development and then various smaller Linux utilities such as nano, Git, and other terminal utilities. I think Git was a very good choice for source control, although this project did not fully stretch it to its full potential and except for a few branches SVN could have been substituted in easily.

## 5.4. Design Decisions

I'm mostly happy with how the design turned out, especially the implementation of the communications stack because it compiles on Arduino and Linux and is completely independent of the radio making it very reusable, possibly to the extent that it could be used in a more advanced sensor node running Linux. The aspect of the design I'm least comfortable with is the configuration program since this was more hacked together than anything and did not undergo the same process as the rest of the code, but while not neat it does fulfil its function and can be easily expanded in the future, I would definitely wish to re-write this program though.

The most questionable choice with the largest impact was choosing the radios because the mesh networking layer has significant limitations, mainly due to Arduino memory size. The first of these is that each node/router can only have a maximum of 4 directly connected child nodes, each of these child nodes can then each have 4 child nodes though. The second limitation is the maximum number of devices per-network which is limited to 255 plus gateway. This can be overcome though by having multiple mesh networks running on different radio channels. The biggest other choice was to use Xbee's, these have better mesh networking but finding good documentation about using the addressing rather than broadcasting on these networks was difficult to come across. Generally Xbee's came across as not being as well supported as the nRF24l01 radios which I think were a good choice.

Encryption was a complete mess. It wasn't initially included in the communications specification and when it was it wasn't adequately researched and so the limits of the Arduino were not taken into account. An example of this was the choice to use Diffie-Hellman key exchange and not realising the memory required to do the modulus arithmetic required by the algorithm. I think it would have been best to pick the simplest solution from the beginning,

which is what was chosen in the end. Using a pre-shared key and an encryption library which runs both on Arduino and Linux.

During the course of this project the Arduino CryptoShield[33] was released which contains various security which support RSA and AES encryption. With this it would have been possible to do a key exchange using public/private key encryption. Along similar lines, if a chip with more RAM had been used even if it only had 8Kb it would have been possible to do the Diffie-Hellman key exchange as originally planned, it may have even been the reason the AES library used by the gateway could not run on the sensor node.

I also would have liked to have had the gateway coordinate network sleeping and a way to update other EEPROM values over the network. The problem with the current implementation is if the sleep times of the network were to be reconfigured it would have to be done individually to each node in the network. This would have been rather difficult to implement in the gateway which is why it was not done.

Another limitation of the system caused by limited RAM and a bug in the RF24 libraries is that there is a limited maximum packet size and it is not dynamically allocated. When attempting to read just a few bytes from the incoming data buffer the RF24 libraries just return everything, this means that reading the communications library header to see how big the message is does not work. The packet size of 128 was chosen because that is the size implemented by the gateway, the limited RAM factor of the ATMega328P means this could only be doubled with what remaining RAM there is.

## 5.5. Customer Acceptance

Near the end of the project when implementation was done to a good standard by all people doing CAT projects we had a meeting with them and demonstrated what had been implemented. They seemed pleased with what had been produced although somewhat disappointed that encryption wasn't working, overall though they seemed happy with what has been done and it seems to have met their expectations. CAT have asked the students who did projects for them to to a demonstration similar to that meeting for the CAT staff along with perhaps a demonstration to the public.

## 5.6. Conclusions

I'm happy with how the project has turned out, there are certain things I would like to go back and improve. The two big things for me are the choice of processor and the configuration GUI. In retrospect I think I should have used a more powerful processor such as the ATMega1284P because on the ATMega328P the code uses 31Kb of 32Kb code space and most of the RAM, this doesn't leave much space for future improvements. Power consumption could have been improved by building around a bare-bones chip, making this project more suitable to renewable energy. The other big thing, like I said previously, would be to redo the configuration program to include a user interface for manipulating the EEPROM and adding an overlay of what each address is assigned to do, along with conversions between data types and bytes. I've enjoyed working on this project and think given some adjustments it has a lot of potential for future expandability.

# 6. Appendices

## 6.1. Documents From CAT

### 6.1.1. E-Mails

***Depth units (from Adam Tyler)***
Hi Tim,

If you could stick to the specification please and send the data in kPa (10kPa = 1.0197442889 meters of water column @ 4degc). Bit late now but, the system could send water temp as well to correctly convert, but as water at the bottom of a lake will be a fairly constant 4degc, it's probably not worth worrying about.

On the basic attributes, I am half way through writing them, but they're not going to be done in time for you. The draft is available at this link https://docs.google.com/document/d/1uMhxR_-R7Prg2ErURpzX6xldsHasf_OAEdlefPVkBkA/edit?usp=docslist_api. I don't feel I can ask you to implement such an early draft specification, but if you do feel like some extra work, the Basic Device Information section is mostly finished; if you do implement it, please comment that it is in draft form and subject to change. If you don't feel like implementing the Basic Device Information section, as long as the ability to add in new attributes in the future is available, and how to do it is documented, that will be fine.

Adam

***Problem with maximum node ID being 32 (from Adam Tyler)***
Indeed 32 is far to low, and I would like to effectively have no limit going forward. We need to get to the bottom of why there is a limit of 32, but assuming it is just an oversight, it is something that will be changed. For this initial stage, hopefully if the limit remains in the short term, it shouldn't have a significant effect.

We will keep you updated.

***System overview (from Adam Tyler)***
I am in the middle of drafting another section for the OEMan coms specification as I realised there isn't much in there explaining how it will all work. Hopefully it will be finished soon, but I thought I would send round a quick summary a) so you have a better understanding to be working with, and b) if you have any comments before I fully write it up.

So we have three main components, the Emoncms server, dumb nodes (sensors and output devices that just talk), and smart nodes (sensors and output devices that listen and talk back).

When a new node appears on the network it registers itself with the server, and gets issued a node id (effectively the nodes to address as there may be more than one node on a single ip address). It then must declare what type of device it is to the server, so it cycles through all of its attributes. In the specification there are a list of groups of attributes, and a node can have as many groups as you want, as long as it declares all the mandatory attributes

for the groups that it wants to use; all nodes will have to use the basic group (yet to be defined in the specification, but it will be basic things like name, firmware version, node type).

for dumb nodes in normal operation, when you want to send data to the server, you will send off the input request as shown in the specification, with the group id, attribute id and attribute number in the first section of the json packet, so that the server knows where the data is coming from, and the value of that data in the second part of the packet in the format specified in the specification. It is intended that all the Emoncms input format will be supported, e.g. you will be able to send in not only a single value, but a batch of values if for example the sensor takes regular readings, but only powers up the radio once in a while.

for smart nodes, the idea is that the sensor will store the inputs as feeds within the node. When the server requests data from a smart node, it requests a feed from the node, by sending the group id, attribute id, attribute number and node id in a json packet. The node then responds with either the latest value, the value for a specific point in time, or a batch of values depending on what is requested.

Smart nodes may need to request data from other nodes, e.g. a thermostat on node A requesting an external temperature from node B. This is something that needs a bit more thought, as in theory node A can directly contact node B, without going through the server; except that only the server will hold the ip addresses of all the nodes. So unless node A is pre programmed with node B's ip, or a way of sending node A node B's ip automatically is developed (which it will be eventually), for now node A will request a node B feed from the server, the server will then request the feed from node B and then pass it onto node A.

Hopefully that all makes some sense to you.

#### 6.1.2. Other Documents

Other documents supplied by CAT have been referenced. The big one is the OEMan Communications Specification, around 100 or so pages. See [5].

## 6.2. EEPROM Map

A lot of configuration is done in the EEPROM, this is detailed here. Values are a single byte unless specified otherwise in square brackets, which will be the byte count.

#### 6.2.1. EEPROM Reset (Address 0)

If address 0 is set to a value other than 0 then the EEPROM is reset with 0's.

#### 6.2.2. RF24 Node ID (Address 1)

This is the node ID used by the RF24 library. If it is 0 a random number is chosen between 220 and 245, the random number generator is seeded by the RTC.

#### 6.2.3. EMon Node ID (Address 2 [2])

The node ID used by EMonCMS is stored as an unsigned short in these two bytes. The first address contains the high order byte and the second address the low order byte.

#### 6.2.4. EMon Calibration Gradient (Address 4[4])
The gradient for the linear calibration is stored as a float in this address as an IEEE754 value.

#### 6.2.5. EMon Calibration Y-Intercept (Address 8[4])
The y-intercept for the linear calibration is stored as a float in this address as an IEEE754 value.

#### 6.2.6. EMon Calibration Base (Address 12[2])
The base, 0 value for pressure is stored here as an unsigned short. The first address contains the high order byte and the second address the low order byte.

#### 6.2.7. Encrypt Enable (Address 14)
If this is set to anything other than 0 it is assumed an encryption key is set. This enabled encryption.

#### 6.2.8. Encryption Key (Address 15 [24])
The Triple DES encryption key.

#### 6.2.9. Alarm Storage (Address 40 [Variable, minimum 2])
The first byte is the wake length in minutes as an unsigned byte. The second byte is the number of alarms following as an unsigned byte. If the number of alarms is 0 then the device is always awake.

Alarms are specified after this as two unsigned byte values. The first byte is the hour between 0 and 23, the second byte is minutes between 0 and 59. If the hour byte is 255 then it means every hour.

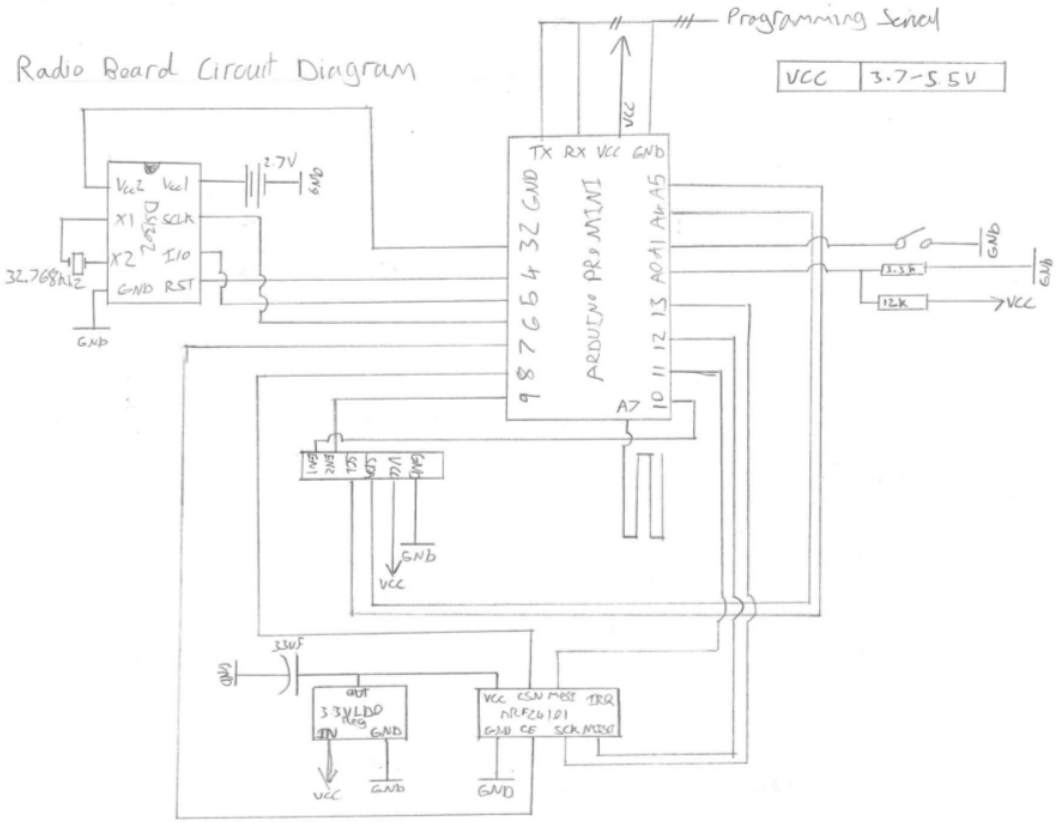
An example with one alarm that wakes for 5 minutes at 10 past every hour.

| Wake Time | Number of Alarms | Hour | Minute |
|---|---|---|---|
| 0x05 | 0x01 | 0xFF | 0x0A |

#### 6.2.10. Attribute Registered (Address 400[Variable, currently 2])
A byte is set to a non-zero value for each successfully registered attribute so it is not repeatedly registered. The current implementation registers time since starting and the pressure attribute, which is why it is two bytes.

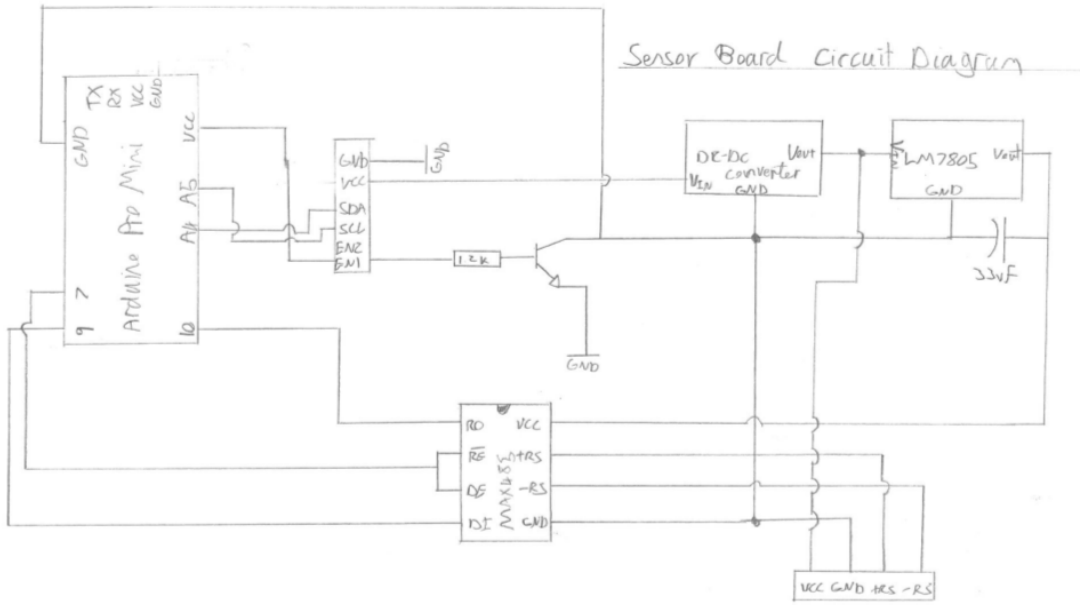
### 6.3. Circuit Diagrams
Following are circuit diagrams for the project. The two boards connect together over the header that is common on both diagrams. Higher resolution images can be found in the digital copy of this report.

### 6.3.1. Radio Board




This is the circuit diagram for the radio board as implemented. The wire on A7 is not necessary and is used to generate entropy if random libraries are used, which this project currently does not.

### 6.3.2. Sensor Board




The sensor board circuit diagram does not completely portray the actual implementation. The implementation has ground leakage where the input ground is connected to the output from the transistor which switches the project. A diode was added to minimize this loss, this circuit though I believe should fix the problem.

## 6.4. Parts List

### 6.4.1. Radio Board

| Component | Count | Unit Price | Total |
|---|---|---|---|
| Arduino Pro Mini (5V, 16Mhz) | 1 | £1.88 | £1.88 |
| nRF24l01+PA+LNA | 1 | £3.25 | £3.25 |
| DS1302 RTC | 1 | £1.92 | £1.92 |
| Micro Switch (SPST) | 1 | | |
| 33uF Electrolytic Capacitor | 1 | | |
| 2000mAh Li-Po Battery 3.7V | 1 | £2.65 | £2.65 |
| Strip-board | 1 | £0.50 | £0.50 |
| MCP1700-3302E/TO Microchip V Reg, LDO +3.3V 250Ma, To-92-3 | 1 | £1.62 | £1.62 |
| 12KOhm Resistor | 1 | | |
| 3.3KOhm Resistor | 1 | | |
| Miscellaneous Cables | | | |
| Miscellaneous Pin Headers | | | |
| | | | **£11.82** |

### 6.4.2. Sensor Board

| Component | Count | Unit Price | Total |
|---|---|---|---|
| Arduino Pro Mini (5V, 16Mhz) | 1 | £1.88 | £1.88 |
| XL6009 DC-DC Voltage Step Up Boost Converter (Configured for 15V) | 1 | £2.57 | £2.57 |
| LM7805 | 1 | £1.98 | £1.98 |
| 33uF Electrolytic Capacitor | 1 | | |
| Diode | 1 | | |
| BC337 Transistor | 1 | £0.20 | £0.20 |
| MAX485 | 1 | £1.50 | £1.50 |
| 100Ohm Resistor | 1 | | |
| 11KOhm Resistor | 1 | | |
| 1.2KOhm Resistor | 1 | | |
| Strip-board | 1 | £0.50 | £0.50 |
| Miscellaneous Cables | | | |
| Miscellaneous Pin Headers | | | |
| | | | **£8.63** |

### 6.4.3. Raspberry Pi Interface

| Component | Count | Unit Price | Total |
|---|---|---|---|
| nRF24l01+PA+LNA | 1 | £3.25 | £3.25 |
| 33uF Electrolytic Capacitor | 1 | | |
| MCP1700-3302E/TO Microchip V Reg, LDO +3.3V 250Ma, To-92-3 | 1 | £1.62 | £1.62 |
| Miscellaneous Cables | | | |
| Miscellaneous Pin Headers | | | |
| | | | **£4.87** |

#### 6.4.4. Total Cost

The total cost of parts is **£25.32**.

# 7. Annotated Bibliography

This final section should list all relevant resources that you have consulted in researching your project. Each reference should also include a brief annotation.

[1] Manny Soltero, Jing Zhang, and Chris Cockril. RS-422 and RS-485 Standards Overview and System Configurations. http://www.ti.com/lit/an/slla070d/slla070d.pdf, 2010. Accessed April 2015.

The specification for RS485.

[2] The Modbus Organization. MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf, 2006. Accessed April 2015.

The specification for the Modbus communication protocol.

[3] Centre for Alternative Technology Wales. About Centre for Alternative Technology. http://content.cat.org.uk/index.php/about-cat-what-do-we-do, 2012. Accessed April 2015.

Information about the Centre for Alternative Technology Wales.

[4] Tim Stableford and Jonty Newman. Low-Power Radio Communications Specification for EMonCMS. https://docs.google.com/document/d/1w8eNexTgmPpNE55wStmc8pmFvy_8MDga7StbSGITetw/edit, 2015. Accessed April 2015.

Radio communications specification implemented and created for this project.

[5] Centre for Alternative Technology Wales. OEMan Communications Specification Version 1.1. https://docs.google.com/document/d/1uMhxR_-R7Prg2ErURpzX6xldsHasf_OAEdlefPVkBkA/edit#heading=h.b11peulovkzy, 2015. Accessed April 2015.

Document describing the specification for the OEMan communications. These are based upon the EMonCMS specification.

[6] Various. Arduino. http://www.arduino.cc/en/Main/Products, 2015. Accessed April 2015.

Arduino product page.

[7] Various. Feature Driven Development. http://www.agilemodeling.com/essays/fdd.htm, 2014. Accessed April 2015.

A description of the process of Feature Driven Development.

[8] Open Energy Monitor. RFM12B. http://openenergymonitor.org/emon/buildingblocks/rfm12b-wireless, 2015. Accessed May 2015.

A brief description of the RFM12B radio module as used by the Open Energy Monitor group.

[9] Various. 2.4G Wireless nRF24L01p with PA and LNA. http://www.elecfreaks.com/wiki/index.php?title=2.4G_Wireless_nRF24L01p_with_PA_and_LNA, 2015. Accessed May 2015.

Description to the nRF24l01 module with power amplifier and low noise amplifier.

[10] TMRh20. Arduino: Using the full potential of NRF24L01 radio modules. http://tmrh20.blogspot.co.uk/2014/03/high-speed-data-transfers-and-wireless.html, 2015. Accessed May 2015.

A description, download and other documentation for enhanced RF24, RF24Network and RF24Mesh libraries.

[11] N. Modadugu and E. Rescorla, "Datagram Transport Layer Security," Internet Requests for Comment, RFC Editor, Fremont, CA, USA, Tech. Rep. 4347, Apr. 2006. Available: http://www.rfc-editor.org/rfc/rfc4347.txt. Accessed May 2015.

The relevant RFC for DTLS.

[12] N. Kushalnagar, G. Montenegro, D. E. Culler, and J. W. Hui, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," Internet Requests for Comment, RFC Editor, Fremont, CA, USA, Tech. Rep. 4944, Sept. 2007. Available: http://www.rfc-editor.org/rfc/rfc4944.txt. Accessed May 2015.

The relevant RFC for 6LoWPAN.

[13] Nebulon. FDD Process. http://www.nebulon.com/articles/fdd/download/fddprocessesA4.pdf. Accessed May 2015.

A document which describes the process of feature driven development.

[14] Sadhna Goyal. Agile Techniques for Project Management and Software Engineering. http://csis.pace.edu/~marchese/CS616/Agile/FDD/fdd.pdf, 2007. Accessed May 2015.

Document on FDD which details what each of the primary roles in FDD are amongst other FDD details.

[15] STS. Rangeable RS485/4-20 mA Depth/Level Transmitter PTM/N/RS485. http://www.pmc1.com/Customer-Content/www/Products/Files/PTM_N_RS485_A.pdf. Accessed May 2015.

Product sheet for the PTM/N/RS485 range of sensors.

[16] Various. Arduino Source Code. https://github.com/arduino/Arduino. Accessed May 2015.

Source code for the Arduino core. This includes all default Arduino libraries used in this project such as Wire.

[17] Various. SimpleModbusMaster library. https://github.com/angeloc/simplemodbusng/blob/master/SimpleModbusMaster/SimpleModbusMaster.cpp#L430. Accessed May 2015.

The SimpleModbusMaster library which may have been used as an alternative. The highlighted line is where there should be a delay.

[18] Timur Maksimov, Jack Christensen. Arduino DS1302RTC library. http://playground.arduino.cc/Main/DS1302RTC, 2014. Accessed May 2015.

RTC library for communicating with DS1302.

[19] Paul Stoffregen. Arduino Time Library. https://github.com/PaulStoffregen/Time, 2015. Accessed May 2015.

Arduino Time Library source code.

[20] Georgios Spanos. ArduinoDES library. http://spaniakos.github.io/ArduinoDES/. Accessed May 2015.

DES and Triple DES library for Arduino and Raspberry Pi.

[21] Various. JSSC Library Source. https://github.com/scream3r/java-simple-serial-connector, 2014. Accessed May 2015.

Java-simple-serial-connector source code.

[22] Various. Apache Commons Math Library. http://commons.apache.org/proper/commons-math/. Accessed May 2015.

Apache Commons Math library homepage, containing links to source.

[23] Various. BitTorrent Sync Homepage. https://www.getsync.com/. Accessed May 2015.

Homepage for the BitTorrent Sync tool used for backing up this project.

[24] Various. GIT SCM Homepage. http://git-scm.com/. Accessed May 2015.

Homepage for the GIT source control management software used for version management in this project.

[25] Various. Eclipse IDE Homepage. https://eclipse.org/. Accessed May 2015.

Homepage for the Eclipse IDE used for Java development.

[26] Various. UECIDE Homepage. http://uecide.org/. Accessed May 2015.

Homepage for the IDE used for Arduino development.

[27] 4-20ma. Modbus Master Library. https://github.com/4-20ma/ModbusMaster/. Accessed May 2015.

Source for Arduino Modbus master library.

[28] Various. SimpleModbusMaster library source. https://github.com/angeloc/simplemodbusng. Accessed May 2015.

SimpleModbusMaster library Github page.

[29] Various. MODBUS over Serial Line Specification and Implementation Guide. http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf. Accessed May 2015.

Specification for using Modbus over a serial line.

[30] Atmel. ATMega328P datasheet. http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf. Accessed May 2015.

Datasheet for the ATMega328P.

[31] Davy Landman. AESLib Github page. https://github.com/DavyLandman/AESLib/. Accessed May 2015.

Source code for Arduino AESLib.

[32] Dallas Semiconductor. DS1302 Trickle Charging Time Keeping Chip. http://web.media.mit.edu/~msung/SAK2/DS1302Z.pdf. Accessed May 2015.

The data sheet for the DS1302 RTC, specifying the 3 wire interface.

[33] Sparkfun. CryptoShield. https://www.sparkfun.com/products/13183. Accessed May 2015.

Product page for the Sparkfun CryproShield.