IFN701 Project 1

IoT Framework for GNSS Devices

A development project

N10124853 Mu Wei Kuo

Supervisor: Charles Wang

Queensland University of Technology

EXECUTIVE SUMMARY

Modern integrated IoT solutions are usually based on the Raspberry Pi single board computer for its simplicity and functionality. However, although the Raspberry Pi is already far more compact than a normal personal computer, it is still significantly larger and more power consuming than a microcontroller. The purpose of this student project is to develop a wireless internet ready integrated IoT framework for GNSS devices which enables remote monitoring and manipulation. The choice of computing hardware is a microcontroller, more specifically, a MicroPython enabled Pycom FiPy module. The GNSS module selected for this project is the ublox NEO-M8L chip. Because there is no operating system overhead on the FiPy board as it simply executes the Python scripts that were developed and uploaded to the FiPy by the developer, it requires significantly less energy to operate than a regular Raspberry Pi implementation under similar conditions. In addition, the FiPy board comes with build in support for advanced long-range wireless connectivity options such as LTE-M.

To achieve the goal of the project, the developer will implement the deliverables using software development practices managed with Agile project management. The software development is conducted with a Visual Studio Code IDE and Pymakr plug-in, and code written in python which is compatible with the MicroPython environment. The codes are always uploaded to the FiPy module for testing. The development cycle will include programming, testing, debugging and evaluating. There are three main deliverables in this project, namely a GNSS raw message parser, a local web server and a remote MQTT client, all written in python. The project successfully delivered all planned deliverables listed in the sprint product backlog, with all major functionality validated and demonstrated in the report.

The outcome of the project is a working software suite that allows local Wi-Fi connection to the IoT framework on FiPy, where the user can view real-time GNSS readings and have the option to switch the operation mode to a remote MQTT configuration, all performed wirelessly eliminating the need for a USB cable. In summary, the project delivered a working software package for the integrated IoT framework for GNSS devices, which will serve as an important foundation for future related development projects as well as further improvements on the existing implementation.

TABLE OF CONTENTS

INTRODUCTION

The Internet of Things refers to the extension of the internet to many non-conventional computing devices. Those devices, which are considered slow or limited in computing power, can be found in many everyday applications such as house appliances, on-vehicle systems, traffic sensors installed on the high way and many more. Most Internet of Things uses low-power processor to perform basic and hardware related tasks. A microcontroller is a lightweight, purpose-built single board computer that does not have an operating system overhead(Wilmshurst, 2001). It is designed to run specific tasks only. Consequently, it has very low power consumption, high computational efficiency, and reliability when compared to a standard computer. With these attributes combined, it is suitable for use with IoT equipment. One notable example for the microcontroller is the Raspberry Pi. The Raspberry Pi was first released in 2012 as an ultra-low-cost, low power single board computer that became very popular with IoT development. However, since the Raspberry Pi supports a full Linux environment, it consumes a considerable amount of energy even during standby. Other popular low-cost hardware such as Arduino and ESP32 remove the operating system overhead from the system, therefore further reduce its power consumption.

The nature of IoT includes constant internet connection, mostly through wireless connections. Current solutions such as Raspberry Pi(Maksimović, Vujović, Davidović, Milošević, & Perišić, 2014), Arduino and ESP32(Maier, Sharp, & Vagapov, 2017) only have built-in support for Bluetooth or Wi-Fi, which limits the application of these microcontrollers. In order to expend the wireless connectivity range beyond the standard Wi-Fi, addition expansions are needed to add other long-range communication options to the package, which can be costly and requires significant work to implement. This low-energy, long-range wireless protocols such as LoRa, Sigfox, and LTE M are becoming more and more important in the deployment of advanced IoT applications as they significantly extend the range of signal reception which enables the IoT devices to operate in the remote area as well as in city center where Wi-Fi signal is suboptimal.

Global Navigation Satellite System (GNSS) consists of several publicly accessible satellite networks, including the U.S.' Global Positioning System, Russia's GLONASS as well as other regional systems(Gebre-Egziabher & Gleason, 2009). Modern civilian grade GNSS devices have mediocre accuracy, which is around ±5m. High precision GNSS receiver that utilizes advanced correction algorithms like Real-Time Kinematics (RTK) has a significantly higher accuracy of up to ±1cm(Gerke & Przybilla, 2016). Autonomous vehicles are one of the applications that require high precision positioning to avoid incorrect navigation, which can lead to traffic accidents. The rise of the Internet of Things (IoT) brings many opportunities to the GNSS. For one thing, the use of always-on IoT devices means that it is possible to keep track of the GNSS system in real-time. Additionally, the retail price of popular high precision GNSS receiver has dropped significantly(GPS WORLD, February 5, 2013).

As a result, the combination of always-on IoT devices and high precision GNSS receivers becomes the foundation for advanced applications such as autonomous vehicles and drones, which enables many new possibilities that used to be limited to academic and industrial applications(Stempfhuber & Buchholz, 2011). The GNSS receiver that originally planned in the project is a ZED-F9P module, due to its small form factor, multi-GNSS, as well as internal hardware accelerated Real-Time Kinematic (RTK) engine(ublox, 2019). However, this was swapped with a NEO-M8L module due to an external factor. Although the advanced ZED-F9P GNSS module was not implemented in the current project, it can, however, be added to the current scope in the follow-up development. As a result, the project act as a foundation for further development of next-gen high accuracy GNSS IoT frameworks.

To enable RTK calibration, the receiver requires correlation data from a nearby base station, which can be delivered via a wireless network(Al-Shaery, Zhang, & Rizos, 2013). We decided to implement the software on a Pycom FiPy board because it is using the latest ESP32 system on chip

Fig 1. The ZED-F9P GNSS module.

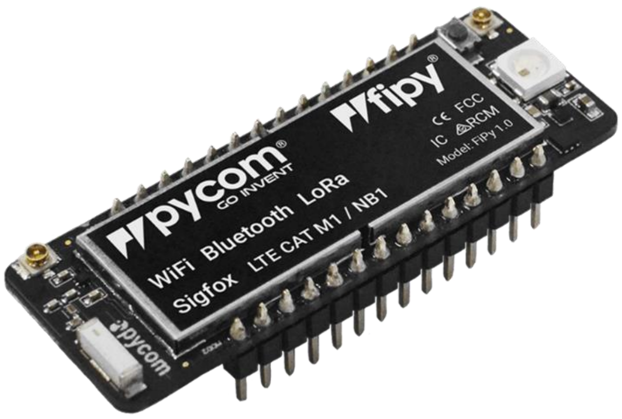


Fig 2. Pycom FiPy microcontroller board.

microcontroller, which is very popular for IoT applications(Singh & Kapoor, 2017). Additionally, the FiPy board supports 5 wireless network protocols: WiFi, Bluetooth, cellular LTE-CAT M1/NB1, LoRa and Sigfox(Pycom, 2019). As a result, the FiPy microcontroller is ideal for the purpose of this project as an internet-enabled gateway.

Although the use of the FiPy microcontroller solved the lack of advanced wireless connectivity issue that is present on previous microcontrollers such as Arduino and ESP32, it does have its own drawbacks. The FiPy board uses MicroPython as its programming language, which is relatively new and therefore suffers from the lack of support and resources(Bell, 2017). Currently, most similar implementations were developed on raspberry pi, which provides a fully functional Linux operating system. However, to further reduce the power consumption and package size of the hardware, it is essential to explore the possibilities of new hardware such as FiPy. As a result, the project will act as a foundation for future implementation on similar MicroPython enabled hardware and explore the potential opportunity and limitation on the new hardware-software ecosystem.

The project plans to implement the WIFI connectivity to enable internet access for the GNSS framework. Software-wise, a microserver will be developed and operate on the FiPy, which handles remote access to the system and controls the hardware(Hartwig, Stromann, & Resch, 2002). Furthermore, a MQTT client will be implemented on the FiPy which should enable remote reading monitoring of the IoT framework. The said MQTT implementation can also be used to remotely control the behavior of the FiPy board. Like other software development projects, the methods include background analysis, software development, and implementation and testing/evaluations, which will be explained further later in the report.

The main deliverable will be a parser generator which reads and parse raw data streaming from the GNSS receiver, which is passed onto the web server. The web server prepares and serves a self-updating webpage, which will constantly feed real-time GNSS readings and display it on a simple webpage. The FiPy module operates in the wireless access point mode, which the user can connect to. From there, the user is presented with a concise explanation and a button, which when clicked will lead to another webpage read.html. This page displays the aforementioned self-updating GNSS readings, which is the main outcome of the project.

Previously, software development on IoT/microcontroller relies heavily on low-level programming languages such as assembly and C. With the introduction of MicroPython, it is now possible to write codes that are easier to understand and therefore makes junior development project like this possible.

The implementation of a barebone web application will allow users to interact with the GNSS framework through a GUI. Consequently, the user can handle most operations without any programming knowledge. This facilitates user access to the system.

Secondly, long-range wireless connection to the device enables remote control and monitoring of the GNSS framework. For instance, it is possible to pack the GNSS framework, Wi-Fi antenna and a battery in a sealed, waterproof package and place it on a building to monitor land subsidence or other anomalies, thanks to the industrial-grade accuracy of the NEO-M8L module.

The software artifacts and deliverables are applicable for future developments. For example, it can be merged as a foundation for development project like an autonomous vehicle’s GNSS module to improve its positioning capability.

Lastly, building our solution on microcontroller platforms such as FiPy instead of the popular Raspberry Pi reduces the power consumption significantly, which allows a much smaller footprint of the entire GNSS IoT package.

ENVIRONMENTAL SCAN AND REVIEW OF PRIOR RELATED WORK

ENVIRONMENTAL SCAN

- Software for Development – MicroPython, MQTT, and UART

  MicroPython is a software implementation of the Python 3 programming language written in C, which is optimized to run on a microcontroller, which is a complete Python compiler and runtime that runs on the microcontroller hardware. Present an interactive prompt (REPL) to the user to immediately execute the supported commands. Includes a range of core Python libraries; MicroPython includes modules that allow programmers to access low-level hardware. Although the original Kickstart event released MicroPython using the pyboard microcontroller, MicroPython supports many ARM-based architectures. Conventionally, the programming language for IoT was dominated by C/C++, which usually takes a considerable amount of time in terms of writing and debugging the code, which could cost up to almost 25% of the time for a large-scale software development project(Du, 2009). As a result, it is important to choose a programming language that is not only more user-friendly but also easier to debug, as the software developer who is responsible for this project is relatively inexperienced. The MicroPython shines especially in this regard, as its time required for debugging and coding is around 5 times shorter than traditional tools such as C++(Kodali & Mahesh, 2016). However, choosing MicroPython does has its disadvantages, as like python it is an interpreting language which can be slower during similar execution conditions. Fortunately, due to the relatively small amount of code and the restricted scale of the project, the impact is practically negligible. Thanks to its ease of implementation and efficient operation, MicroPython is becoming an uprising programming language especially in the sector of IoT and embedded devices.

  As the project is built around a webserver, it is important to choose a good and lean webserver framework/library which is optimized to operate on a resource confined MicroPython environment. With proper online research and web crawling, I found a MicroWebSrv implementation which was written by J-Christophe Bos(Bos, 2019). According to the author, “MicroWebSrv is a micro HTTP Web server that supports WebSocket, HTML/python language templating and routing handlers, for MicroPython (principally used on ESP32 and Pycom modules)”. In the project, we only imported his “microWebSrv.py” to our development board, which serves as the backend/server of our implementation. With the help of the MicroWebSrv module, it is possible to create websites and host them directly on the FiPy board. In addition, it can also handle popular HTTP requests such as GET and POST to interact with the user.

  Aside from the main MicroPython programming language, two important built-in modules are applied in our project. The first one is UART, which stands for Universal Asynchronous Receiver-Transmitter, is, in fact, hardware that is part of the microcontroller which is responsible for asynchronous serial communication between the microcontroller and its peripherals. In MicroPython, there is a system module/class responsible for the configuration and establishment of such protocol. Another protocol used in the project is the MQTT, which stands for Message Queuing Telemetry Transport. It is a very light-weight messaging protocol that is specifically designed for devices that have strict internet bandwidth or computational resource. It has a publish-subscribe model and relies on a “MQTT broker” to handle communication between its clients. The client, like the FiPy module, can publish its GNSS readings to the broker, which will push the information to the subscribers like a mobile phone or computer.

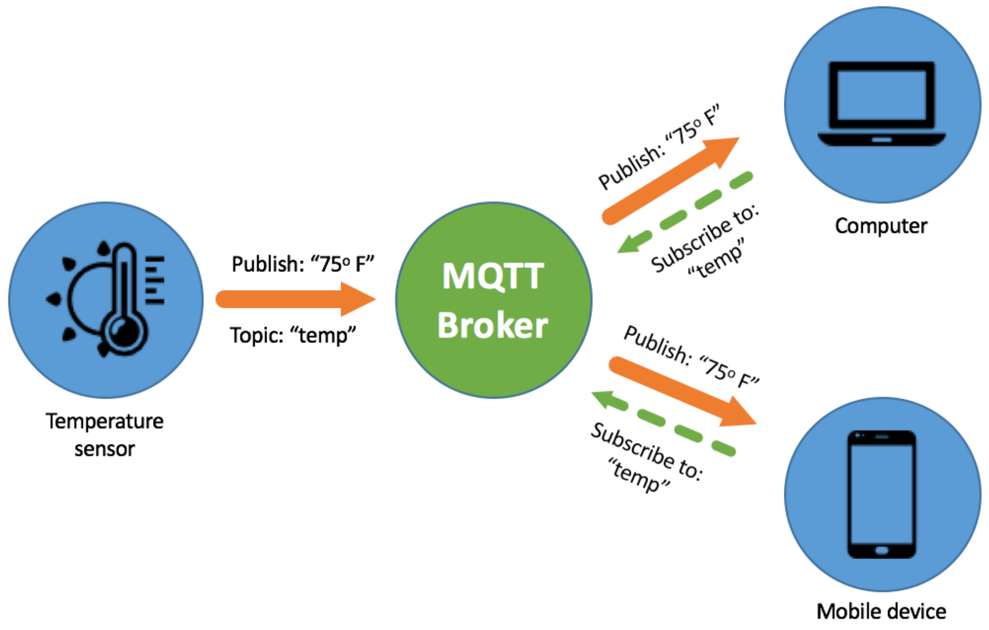


Fig x. MQTT use case illustration. Graph adapted from http://www.innorobix.com/mqtt/message-queuing-telemetry-transport-mqtt/

- Computing Hardware – FiPy

Previously, most of the research/academic IoT devices were built on the popular raspberry pi single board computer, which proves to be a hugely successful computational platform thanks to its versatility and ease of use. The Raspberry Pi is a Linux-based single-chip computer. It was initially developed by the British Raspberry Pi Foundation to encourage basic computer science education in schools with low-cost hardware and free to distribute software. The Raspberry Pi is currently produced by two companies with open production licenses (production is made possible through 2 companies with production licenses: Element 14 and Premier Farnell and RS Components. Both companies sell their Raspberry Pi devices through the online store.) Raspberry Pi is equipped with an ARM architecture SoC from Broadcom, installed 256MB memory (type B has been increased to 512MB memory), using SD card as its storage medium, and has one Ethernet and two USB interfaces, which allows easy access to the system. The system also has HDMI (support sound output) and RCA terminal output support. The Raspberry Pi area is only a credit card size and is about the size of a matchbox. The operating system uses the open-source Linux system to meet the basic needs of web browsing, word processing, and computer learning. Available in A and B models, the price is $25 for the A and $35 for the B. The Raspberry Pi Foundation began accepting Type B orders since on February 29, 2012. The Raspberry Pi Foundation provides distributions for its opensource Linux operating system based on the ARM architecture for mass downloads and plans to provide support for Python as the main programming language, supporting BBC BASIC (Copy via RISC OS image or Linux "Brandy Basic"), C language and Perl. The Raspberry Pi Foundation released the Raspberry Pi 3 in February 2016. Compared to the previous generation Raspberry Pi 2, the Raspberry Pi 3 processor was updated to the 64-bit Broadcom BCM2837, adding a Wi-Fi wireless network. And Bluetooth and the price is still $35. However, as of now, the raspberry SBC does not support wireless connectivity other than Bluetooth and Wi-Fi, which inevitably limits its application spectrum, especially in scenarios like limited space or restricted power supply. In addition, although the raspberry pi SBC is already significantly smaller than a regular computer, it is still larger than MicroPython enabled Pycom devices like FiPy.

As for the microcontroller selected for our project, it is the latest offering from Pycom, the FiPy MicroPython enabled development module. Currently available for 54€, the FiPy module is based on the popular ESP32 microcontroller, along with several added wireless connectivity options. It has a powerful Xtensa® dual-core 32-bit LX6 microprocessor capable of hardware floating point acceleration as well as support for multithreading in Python. A total of 520KB + 4MB RAM is available on the FiPy which enables multithreading. Additionally, the module comes with 8MB of onboard flash memory, which is used to store Python scripts as well as other related files. As for wireless connections, it is 802.11b/g/n 16mbps Wi-Fi enabled, which is the one that is implemented in this project. Additionally, the FiPy board supports 4 other additional wireless network protocols: Bluetooth, cellular LTE-CAT M1/NB1, LoRa and Sigfox(Pycom, 2019). Although not currently within the project scope, this advanced connectivity can be useful in future projects. In addition to the advanced wireless connectivity, the FiPy board has numerous general-purpose IO pins that can be used to connect to a wide variety of peripherals, which boast its versatility dramatically. For example, one can connect a DHT22 humidity and temperature sensor to a user designated GPIO pin on the FiPy board and program the MicroPython code to decode and parse the raw data stream coming from the input pins.

Aside from a vastly superior wireless connectivity capability, the FiPy also excels in terms of power draw. The popular model from its competitor, Raspberry Pi 2 Model B+, can consume up to 820 mA when subject to maximum load and 220 mA when idle(He, Segee, & Weaver, 2016). By comparison, the FiPy module consumes only up to 126 mA when the Wi-Fi module is under load and as low as 62.7 mA when idle(Pycom, 2019). As we can see, the FiPy board consumes merely a fraction of the energy when compared with other popular solutions like raspberry pi, which is critical as not only does the lower power consumption allows for longer run time, smaller package size for the power supply, it also enables fewer maintenance visits and the bundled battery can last significantly longer. Additionally, the FiPy module comes with built-in deep-sleep mode, which when enabled will reduce the idle power usage even further, making it ideal for asynchronous reporting IoT device. The programming language available for FiPy is the MicroPython, which as stated before has multiple advantages for the purpose of IoT development.

In conclusion, FiPy has advantages over other similar competitors in terms of size, power consumption, connectivity while still providing the core functionality of a python implementation. Altogether, these attributes make it an ideal choice for the implementation of internet-enabled IoT devices.

REVIEW OF PRIOR RELATED WORK

- Low-Cost Ambient Monitoring using ESP8266 (Kodali & Mahesh, 2016)
  The ESP8266 is essentially the predecessor of the more powerful ESP32, which in the term is what the FiPy SBC based on. As a result, analyzing why and how the author went through their project can help to assist our own project, due to the nature of the development style project and the similarity between. The basic idea of their project is to develop a software solution that is capable of running on the low-cost, low performance, resource-constrained ESP8266 board, which can read and parse raw readings from a DHT22 temperature and humidity sensor connected to the ESP8266 board through I^2C connection. The parsed data are then displayed on a small OLED display panel, which is also connected to the ESP8266 board through another I^2C connection. As a result, it forms a simple, single directional data flow from the DHT22 sensor to the ESP8266 and finally the OLED display. However, as we can see there's no wireless connection to be made, nor is any web server implementation. Consequently, this publication provides the most barebone and simple, low-level demonstration of how a local sensor-reader-output loop can be achieved.
- Design and Implementation of a Low-Cost Web Server Using ESP32 for Real-Time Photovoltaic System Monitoring (Allafi & Iqbal, 2017)
  Although this project was developed using Arduino IDE, which writes code in C/C++, therefore, is different from our MicroPython implementation, it is otherwise conceptually very similar to our own project. Their project uses the ESP32 microcontroller to read input monitoring data from connected photovoltaic sensors. The data is parsed and processed. A web server operates on the ESP32 board, and WIFI connection is established. The web server listens to incoming HTTP requests and serves web contents accordingly. The ESP32 will continuously read and store the data on a micro SD card, which can be downloaded by the client through the said web service hosted by the web server. Although their project does not involve any dynamic webpage that displays self-refreshing contents of sensor readings, the overall structure still centers around the web server and the microcontroller.

PROJECT METHODOLOGY

Overview of the methodology

The main purpose of this project is to establish a MicroServer that acts as a middleman between the IoT hardware and the user. Unlike conventional microcontrollers which are frequently implemented in C or even assembly language, the Pycom FiPy board is driven by MicroPython. MicroPython is a relatively new programming language which came into existence since 2014, which is based on the popular Python 3 with C as its backend. Consequently, it is a streamlined implementation which focuses on efficiency and is created specifically for applications such as microcontroller and IoT. As a result, we will be implementing the software solution using the MicroPython programming language in this project as it is supported by the hardware (FiPy) used in the project.

The project intends to achieve the objectives by:

1. Analyze available information from both academic databases, manufacturer's resource and online repository like GitHub.
2. Develop a MicroServer that runs on the FiPy board using integrated developing environment such as Visual Studio Code with the pymakr plugin. A simple web application will also be constructed to enable user-friendly interaction.
3. Implement a parser-generator to read and parse raw data from the NEO-M8L module through UART.
4. Configure and establish AP/Client mode Wi-Fi connectivity to the microcontroller.
5. Establish an MQTT client on the FiPy and connect to a remote MQTT broker/server
6. Test the above software solution and evaluate its effectiveness and reliability.

Tools and IDE

The software development section of the project was carried out using Microsoft Visual Studio Code, which is a very popular source-code editor. As the name suggests, it is developed and maintained by Microsoft. However, unlike other software made by the same company, Microsoft Visual Studio Code is an open-source software published under the permissive MIT License, therefore making it popular among software developers and also making it easy to discover and resolve bugs. The Microsoft Visual Studio Code software is written in JavaScript and CSS and is highly expandable with the support of plug-ins. As our microcontroller hardware is a highly customized version of the original ESP32 board, the manufacturer (Pycom) provides its own Microsoft Visual Studio Code plugin called Pymakr, which when activated adds a read–eval–print loop (REPL) terminal to the VS Code interface. Through the REPL and other functionalities provided by the Pymakr, the developer can easily interact with the FiPy microcontroller. The web server is based on the MicroWebSrv library as described in the background scan section, which is used by the developer to implement our own web server. The reason why the developer selected this package is because of its simplicity and ease of use, which boosts the speed and reliability of the development.

Background Analysis

In order to understand the structure of a typical IoT framework, a certain amount of literature review is needed, mostly through querying on QUT library as well as on Google scholar websites. In addition, exploring manuals and online resources for the FiPy microcontroller is a key necessary step to successfully develop software solutions in a MicroPython environment. Throughout the software development process, various internet knowledge bases will be utilized, such as the Pycom forum, Stack Overflow, and GitHub for the purpose of searching existing solutions and troubleshooting. The key outcomes from this session are explained in the background scanning section.

Development and Implementation

The development of the MicroPython software follows the iterative design, meaning it is going to be a continuous and staged implementation. The first piece and most important deliverable is the microserver. It will respond to web requests on a local Wi-Fi network. The next target for implementation is the MicroPython code to correctly read and parse raw data streaming from the NEO-M8L GNSS receiver via the UART, as well as present the parsed data such as the precise longitude and latitude on the web application hosted by the microserver. Lastly, an attempt will be made to establish a connection to the cloud MQTT broker and publish GNSS readings on it. Throughout the process of software development, the developer will go through a series of the concept-implement-test-debug cycle in order to ensure the quality of code is standardized.

Analysis and Evaluation

At this stage, field trials will be conducted to verify the reliability and effectiveness of the deliverables. The remote wireless connection will be tested along with the designed user interaction and use case simulation. Both the local web server and remote MQTT connectivity will be examined and verified. All relevant data and recording will be analyzed and summarized in the result and discussion section of this report.

Communication Plan

As the project consists of only 2 stakeholders- the student and the supervisor, the principle of communication is mostly on-demand and on a regular basis. There will be a weekly meeting with the supervisor to update the development progress, clarify the next step as well as summarizing issues encountered and potential solution.
In addition, both instant messenger and email will act as on-demand communication medium so that it is possible to resolve difficulties that may impede the project. At the end of each Agile sprint, a journal file containing the steps and progress of the development will be submitted to the blackboard for record keeping.

Project Management and Sprints

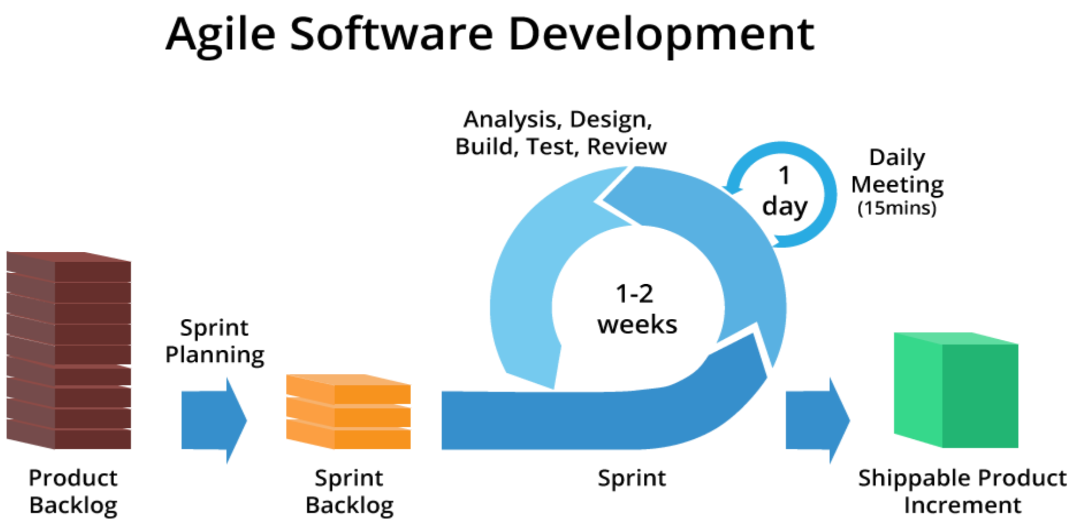

Agile Software Development
Analysis, Design, Build, Test, Review
1 day
Daily Meeting (15mins)
1-2 weeks
Sprint Planning
Product Backlog
Sprint Backlog
Sprint
Shippable Product Increment

Fig 4. Agile software development structure outline

As this is a development type project, the Agile software development principle will be observed. All target deliverables are categorized according to the MoSCoW prioritization method. A typical Agile sprint lasts two weeks in this project. Depending on the progress made during the previous week, low priority backlog items may or may not be implemented in order to adapt to the iterative development requirements and keep the project on track. The following is the product backlog and the associated sprint timeframe, with each sprint share the same background color. Since the start of the project, there have been some delays caused by technical difficulties and hardware availability issues, which resulted in the modification of the project deliverables and scope. The following is the revised version of the original sprint backlog.

ID	Sprint Backlog Item	Week	MoSCoW
1	Pre-project documentation, student agreement	1	MUST
2	Academic literature review and background analysis	2	MUST
3	Week 3 oral presentation, a project pitch	3	MUST
4	Project plan documentation	4	MUST
5	Prepare development environment, hardware set up	5	MUST
6	Implement MicroServer	6	MUST
7	Web app development #1 (Landing page)	7	MUST
8	Configure NEO-M8L to the FiPy	8	MUST
9	Parse NEO-M8L raw data	8	MUST
10	Web app development #2 (Sensor reading page)	9	SHOULD
11	WIFI configuration, network protocol and IP set up for remote access	10	MUST
12	The WIFI mode switch and Web app development #3 (Switch page)	11	COULD

13	Web app development #3 and MQTT client implementation	12	COULD
14	Field trial and evaluation of deliverables	13	MUST
15	Final project report and presentation	14	MUST

OUTCOME OR RESULTS

Recap of the deliverables

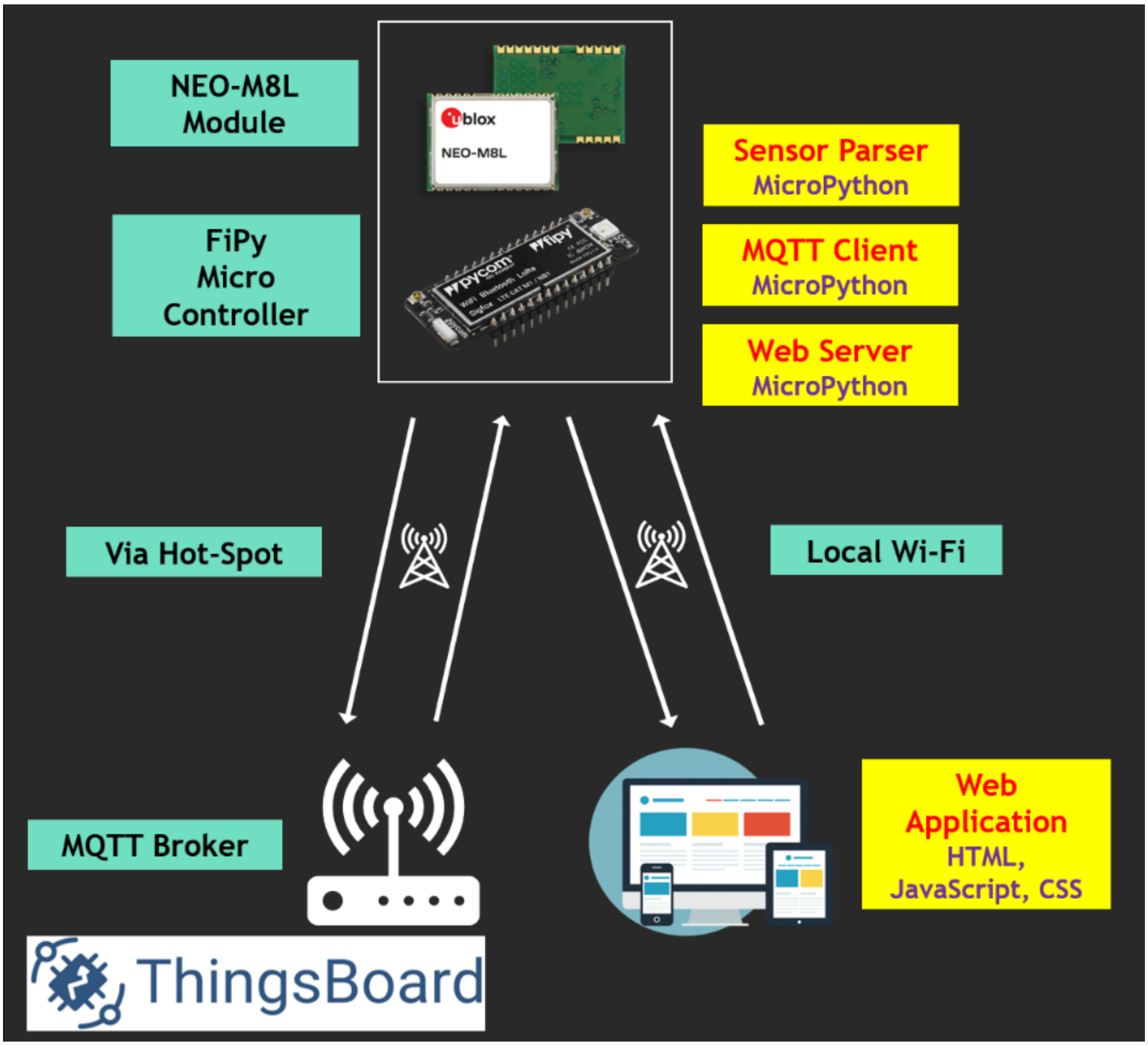


Fig 3. Graphical representation of the GNSS framework layout. The yellow boxes represent software implementations (artifacts, deliverables) while mint boxes are hardware or services.

The target deliverables of this project are:

1. A functional MicroServer that runs on the Pycom FiPy microcontroller.
2. A compact NEO-M8L raw reading parser
3. A web application accessible through a web browser for user-friendly interaction with the GNSS framework.
4. Functional wireless Wi-Fi connection for the GNSS framework.
5. MQTT MicroPython implementation.
6. A comprehensive end of project report summarizing the outcome of the development.

Artifacts implemented

1. Sensor Parser (parserGen.py)

The sensor parser part of the MicroPython code implementation is responsible for receiving raw reading string that was received from the UART python object. The raw readings generated by NEO-M8L follows the NMEA (National Marine Electronics Association) naming convention. The NEO-M8L module generates 6 lines of raw sensor readings in each polling cycle: RMC, VTG, GGA, GSA, GSA, and GSV.

Each type of NMEA message is a string of characters divided by comma. Each divided data field has a meaning specific to its NMEA message tag.

For example, the RMC NMEA string looks like this:

RMC: Position, velocity, and time

`$GPRMC,123519,A,4807.038,N,01131.000,E,022.4,084.4,230394,003.1,W*6A`

The first section of the message is the Message-ID, followed by the UTC time of fix, the satellite status, latitude reading in DMS format, north or south hemisphere, longitude in DMS format, west or east hemisphere, ground speed, track angle, date, magnetic variation and lastly checksum data.

The parser takes the raw reading string as its input data. The data goes through a series of type conversion, reformatting and other processing. For example, in order to convert the dd-mm-ss formatted latitude into a common decimal one, the developer coded the following function in the project MicroPython script:

```python
def parseLat(temp, temp2):
        try:
            raw = list(str(int(float(temp)*10000000)))
            degree = float(raw[0] + raw[1])
            minutes = float(raw[2] + raw[3])/60
            seconds = float(raw[4] + raw[5] + raw[6] + raw[7] + raw[8] +
                      raw[9] + raw[10])/100000/(60*60)
            dms = str(degree + minutes + seconds)
            dirr = temp2
            Lat = str(dirr + dms)
            return Lat
        except:
            return 'N/A'
```

The `raw = list(str(int(float(temp)*10000000)))` transform the original reading to a list object, which allows individual digit selection. The next three lines pick positional digits from the list object and recombine/process them accordingly. Finally, the degree, minutes, seconds are assembled back together and is now in decimal format. The final decimal value is then paired with its hemisphere indicator and the combined value is returned. An exception will be raised when there are no data to be parsed, which happens while the NEO-M8L is trying to get a valid fix. In this case, an 'N/A' message is returned, indicating its current situation. Other types of GNSS readings are parsed accordingly in a similar fashion. For the purpose of this project, the developer selected RMC, GGA and GSA messages to be processed and generated relevant GNSS user-friendly information like latitude and longitude in decimal format, velocity in knots, height above sea level and others.

However, because the return syntax will terminate a loop condition, the developer needs to implement an alternative approach to continuously generate new parsed readings whenever new raw data comes in. By using the yield syntax instead of return, the developer was able to create a generator object which can constantly receive and yield parsed results. However, during the initial attempt, the developer observed strange raw readings being returned by the uart object, which is responsible for acquiring strings from the connected NEO-M8L device. The string appears to be incomplete and randomly distribute across the test case. After multiple bugfix attempts, the developer eventually discovered that the root cause is likely originated from buffer overload. The NEO-M8L GNSS module has a small onboard buffer memory where it temporarily stores the raw NMEA strings. If no read command was issued for a while, the buffer will run out of space and messages stored inside can become corrupted. In order to fix this error, the developer implemented a line of code that will be executed before each raw data reading loops. The `flushCache = UT.readall()` as its name suggests, reads as much information as possible from the NEO-M8L on-board buffer. This action effectively clears out any potentially corrupted messages that may be present within the buffer, thereby eliminate the corrupt NMEA message which can prevent the parser from parsing the message correctly. The relevant python code is as follows with clear in-line comments:

```python
def output(UT):
    while True:
        pycom.rgbled(0xff6600) # yellow
        counter = 0
        tempStore = [] # store up to 5 reads, reset at each round
        flushCache = UT.readall() #flush cache on m8l to prevent erroneous readings
        while counter <= 4: # read the first five lines of raw data
            try:
                # keep reading data until the first stream of raw data is received
                data = UT.readline()
                if data:
                    decoded = data.decode("utf-8")  #the raw reading from
                                                     uart needs to be decoded
                    tempStore.append(decoded)
                    counter += 1
                    time.sleep(0.03) # avoid sensor overload
            except:
                pycom.rgbled(0xff0000) # red
                print("Error")
        pycom.rgbled(0x00ff00)
        LatLon, velocity, gpsQ, height, GMT, PDOP, HDOP, VDOP, uniqsatNum =
        dataParse(tempStore) # parse decoded raw readings
        yield LatLon, velocity, gpsQ, height, GMT, PDOP, HDOP, VDOP, uniqsatNum
             # generate readings
```

The output function takes one argument which is the instantiated uart object, and the outer while loop is set to true so it can process raw readings indefinitely. Each while loop cycles through a flush-read-parse-yield process and generates useful GNSS data as its output. This sensor parser MicroPython implementation is applied throughout the software framework and is responsible for acquiring GNSS readings for the rest of the software components.

2. Web Server (part of the allInOne.py) and the Web Application

As stated in the background scan section, the web server component in this project is based on the great work done by J-Christophe Bos, who crafted a compact yet powerful web server MicroPython library and made it opensource on GitHub. Although his library comes with several python modules, each empowers functionality such as web socket and pyhtml templating, for the purpose of this project the developer aims to minimize the total footprint and file size the implementation, only the very core of the `microWebSrv.py` was imported and incorporated into this project. This single python module contains the essence of a functional and compact backend for the project's web implementation part.

```
routeHandlers = [ ( '/read', 'GET',  _httpHandlerNEOGet1s ),
                ( "/test",  "GET",  _httpHandlerTestGet ),
                ( "/test",  "POST", _httpHandlerTestPost )]
srv = MicroWebSrv(routeHandlers=routeHandlers, webPath = '/flash/www/')
time.sleep(0.25)
srv.Start(threaded = True) # start server
```

Three route handlers were defined for the web server to respond to the respective web requests. The first route handler is designed to allow the implementation of a self-updating sensor reading web page. Here, the developer utilized an HTML5 standard called server-sent events, which enables one-directional data update from the server to the client. Once the webpage is loaded, the server will send the server sent events formatted strings to the client, which when supported by the browser will be updated and displayed on the webpage via the associated JavaScript code. Part of the server-side python script is as follows:

```
def _httpHandlerNEOGet1s(httpClient, httpResponse):
    try:
        latlon, velocity, gpsQ, height, GMT, PDOP, HDOP, VDOP, uniqsatNum =
        parsedReadings.__next__()
    except:
        latlon = velocity = gpsQ = height = GMT = PDOP = HDOP = VDOP = uniqsatNum =
        'Error occurred, please restart FiPy...'
    second = 1000
    httpResponse.WriteResponseOk(
        headers = ({'Cache-Control': 'no-cache'}),
        contentType = 'text/event-stream',
        contentCharset = 'UTF-8',
        content =   'retry: {0}\n'.format(second)+
                    'data: {0}<br/>\n'.format('Latitude and Longitude:') +
                    'data: {0}<br/><br/>\n'.format(latlon)+
```

The server-side script corresponds to the JavaScript code embedded in the read.html file. The JavaScript code related to the server sent event is displayed below:

```
<script>
```

```
if(typeof(EventSource) !== 'undefined') {
    const source = new EventSource('http://' + window.location.hostname + '/read');
    source.onmessage = function(event) {
        document.getElementById("result").innerHTML = event.data;
    };
}
else {
    document.getElementById("result").innerHTML =
    "It appears that your browser does not support server-sent events...";
}
</script>
```

The other two route handler feed a very barebone webpage which simply ask the user to select the next step of action to be executed. The webpage prompts the user to confirm switching to reote MQTT mode by typing 'mqtt' in the input text box. The web form reads and send the content to the FiPy microserver.

```
def _httpHandlerTestPost(httpClient, httpResponse) :
    formData  = httpClient.ReadRequestPostedFormData()
    status = formData["status"]
    content   = """\
    <!DOCTYPE html>
    <html lang=en>
        <head>
            <meta charset="UTF-8" />
            <title>MODE SWITCH</title>
        </head>
        <body style="background-color: lightgrey; text-align: center;">
            <h1><b>===MODE SWITCH===</b></h1>
            Choice = %s<br />
        </body>
    </html>
    """ % ( MicroWebSrv.HTMLEscape(status))
    httpResponse.WriteResponseOk( headers         = None,
                                  contentType     = "text/html",
                                  contentCharset = "UTF-8",
                                  content         = content )
    if status == "mqtt":
        print(">>MQTT mode selected<<")
        mf = open("status.txt", "w")
        mf.write("yes")
        mf.close()
```

As demonstrated above, should the returned posted form data contain the 'mqtt' keyword, a confirmation text file is created on the server, which is used to determine whether the system should be reconfigured to the MQTT mode.

```
while True:
    time.sleep(1)
    condf = open("status.txt", "r")
    cond = condf.read()
    condf.close()
    if cond == "yes": # use to determine mqtt switch
        print(">>stopping server<<")
        pycom.rgbled(0xff0000) #red
        time.sleep(1)
        srv.Stop() #stop server
        print(">>local server stopped<<")
        time.sleep(1)
        break
    else:
        pass
```

The above while true loop is also running alongside the web server script, which checks whether the text file contains the 'yes' keyword at a fixed interval. Should the keyword be found, it will stop the web server and initiate the mode switch procedure.

3. MQTT Client (also part of the allInOne.py)

Within the `allInOne.py` script, a function named `switchToParallelMQTT` is defined, which will be executed once the server receives a mode switch request and is successfully connected to the bundled Wi-Fi access point, which is a Telstra USB device. Two MQTT clients are set up and run concurrently, each publishes GNSS readings using the fire-and-forget method, meaning that the MQTT client only sends the message through MQTT protocol once, and does not check whether the MQTT broker receives the message or not. This is useful for one-directional sensor polling as the sensor-IoT device is only required to constantly publish its current reading to the broker. Additionally, this mode of MQTT allows resource saving as verification is omitted.

```
client = MQTTClient(AIO_CLIENT_ID, AIO_SERVER, AIO_PORT, AIO_USER, AIO_KEY)
client.connect()
client2 = MQTTClient(client_id="test", server=THINGSBOARD_HOST, port=1883,
          user=QUT01_ACCESS_TOKEN, password=QUT01_ACCESS_TOKEN, keepalive=60)
client2.connect()
```

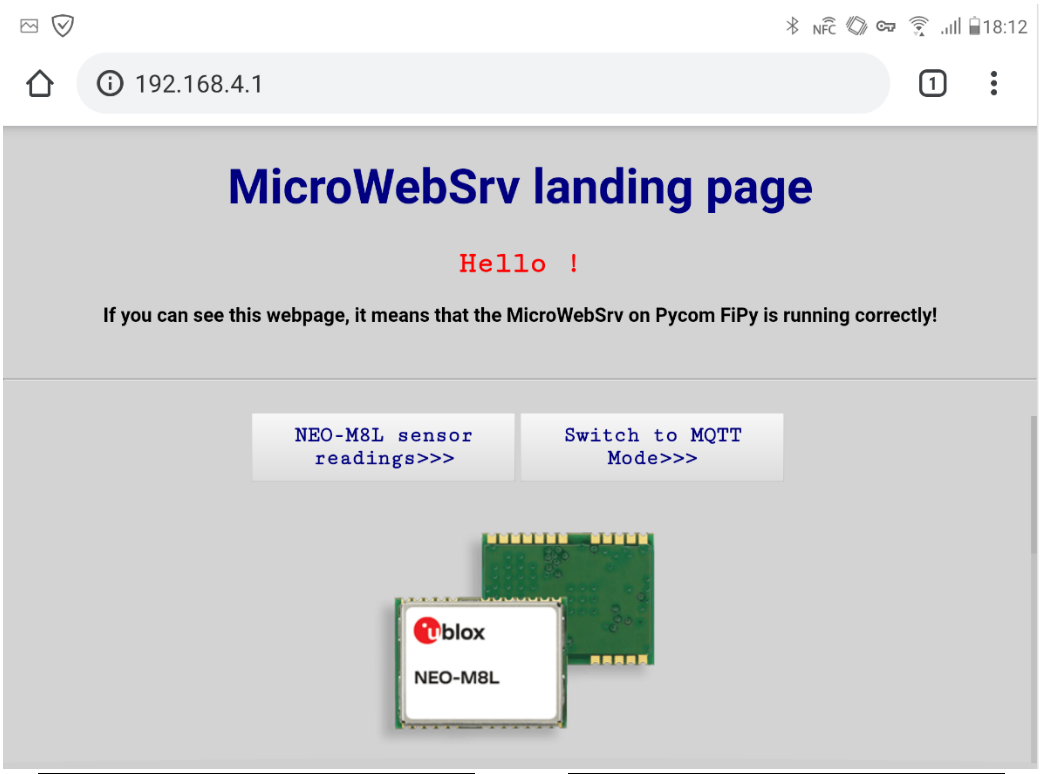
And the following code to publish readings to the remote MQTT broker.

```
client.publish(topic=AIO_RANDOMS_FEED, msg=str(gnssReadings))
```

Most of the materials related to the MQTT client was adapted from the previous work done by the supervisor, which was designed to operate on a Raspberry Pi single board computer. As a result, there exist some differences in the system MQTT library and the developer must make a few adjustments to correctly adapt the original python code to the MicroPython environment.

Demonstrations

The following are screenshots of the solution package working as intended. Additionally, an online video demonstrating its dynamic polling is embedded in this section.

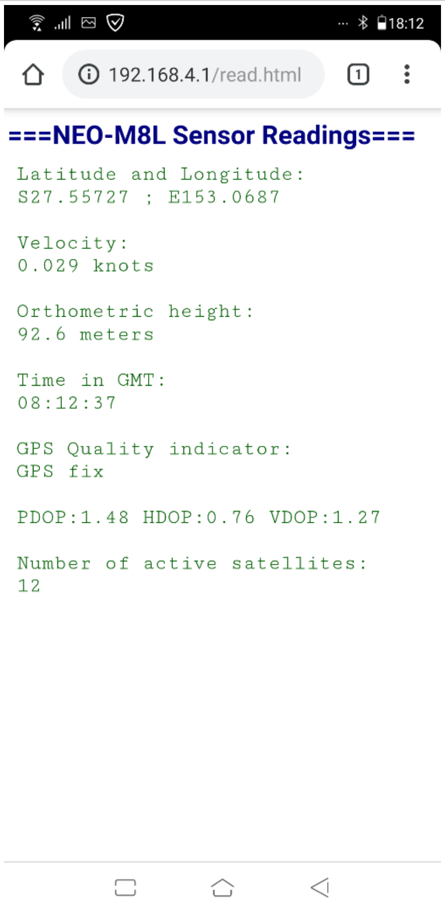


192.168.4.1

MicroWebSrv landing page

Hello !

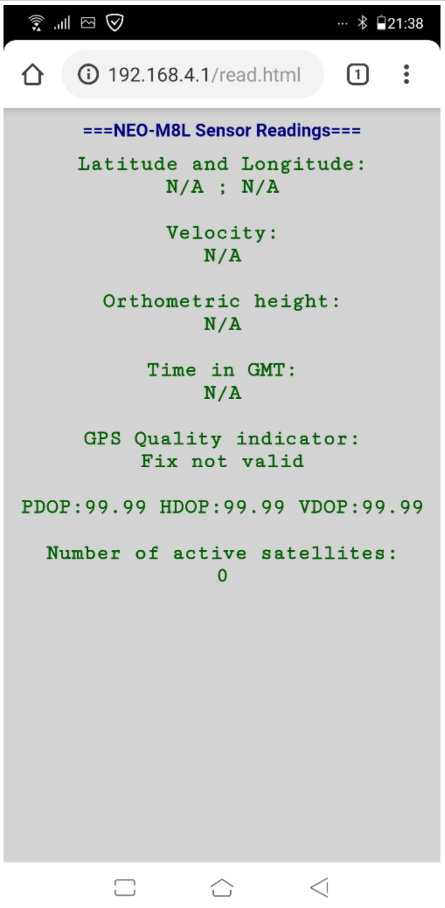
If you can see this webpage, it means that the MicroWebSrv on Pycom FiPy is running correctly!



192.168.4.1/read.html

===NEO-M8L Sensor Readings===

```
Latitude and Longitude:
S27.55727 ; E153.0687

Velocity:
0.029 knots

Orthometric height:
92.6 meters

Time in GMT:
08:12:37

GPS Quality indicator:
GPS fix

PDOP:1.48 HDOP:0.76 VDOP:1.27

Number of active satellites:
12
```

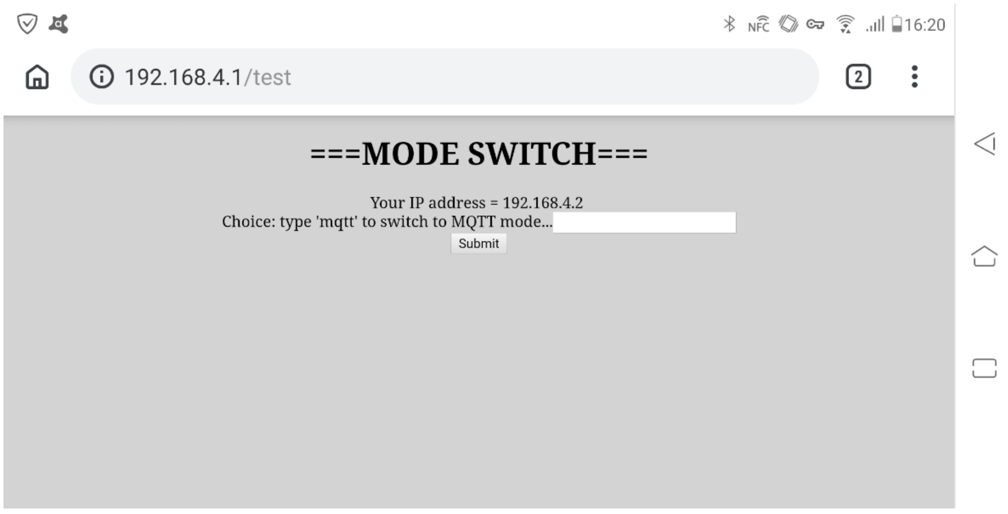


192.168.4.1/read.html

===NEO-M8L Sensor Readings===

```
    Latitude and Longitude:
          N/A ; N/A

          Velocity:
             N/A

     Orthometric height:
             N/A

         Time in GMT:
             N/A

    GPS Quality indicator:
        Fix not valid

PDOP:99.99 HDOP:99.99 VDOP:99.99

  Number of active satellites:
              0
```

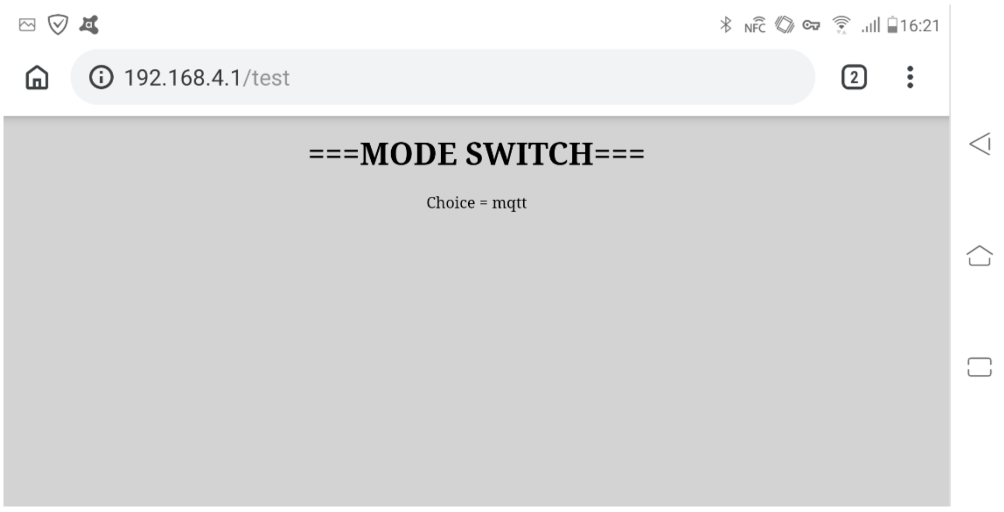
Fig x. Screenshots of the front-end deliverables. Top: A clean landing page, with two action buttons to select by the user. Bottom left: real-time NEO-M8L readings, refreshes every 2 seconds. Bottom right: re-styled sensor reading page.



===MODE SWITCH===

Your IP address = 192.168.4.2
Choice: type 'mqtt' to switch to MQTT mode...

Fig x. Mode switch confirmation page. When the user clicks the ‘switch to MQTT mode’ button on the landing page, they will be redirected to this page. Bottom: confirmation message returned by the server, indicating system changing to MQTT mode.



===MODE SWITCH===

Choice = mqtt

The following YouTube links display the live action of the implemented IoT framework.

Local web server, sensor reading page **Remote MQTT with ThingsBoard.io map integration**

Remote Adafruit MQTT integration

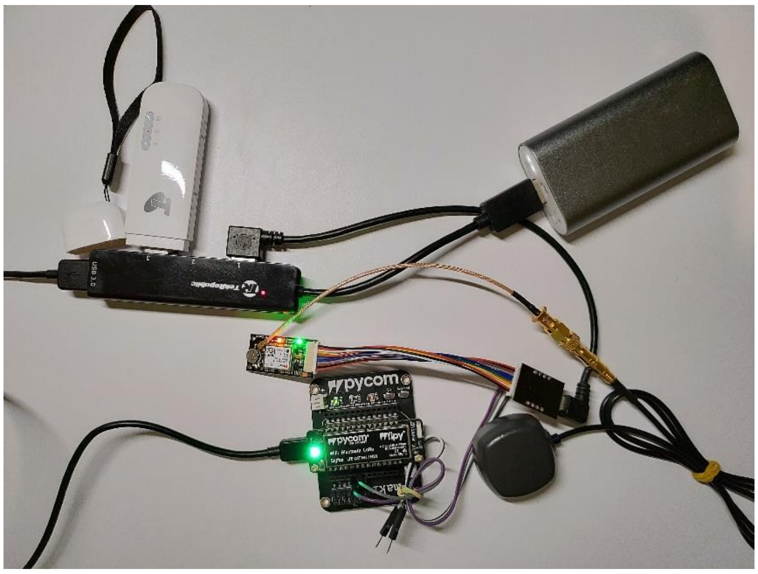


Fig x. Hardware set-up of the IoT framework and the NEO-M8L GNSS sensor

DISCUSSION

It is clear that the project developer successfully delivered all promised deliverables mentioned in the modified product backlog. The completion of the project presents a more energy and space efficient implementation of the original Raspberry Pi based solution, therefore indicating the feasibility of migrating existing Raspberry Pi based python solution to the new MicroPython framework. As described in the background scan section of this report, the Pycom FiPy module only costs a fraction of the power consumption produced by a Raspberry Pi 3B. The software implementation provides easy wireless access to the hardware, eliminating the need of connecting to a computer and access through REPL. Instead, the user only needs to connect to the Wi-Fi from FiPy. After verifying the status of the connected GNSS sensor through the local readings web page, the user can alter the software configuration running on the FiPy wirelessly through the web application, instruction the framework to connect and publish to remote MQTT broker, therefore enabling long-distance monitoring of the sensor polling data. At its current stage, the system relies on the use of a standalone USB powered Wi-Fi hot spot. However, the FiPy module does come with built-in support for several long-range wireless options including LoRa, Sigfox, and LTE-M. these wireless communication protocols are supported by FiPy and have built-in system MicroPython library, so future projects can explore advanced low-power onboard wireless modules should key hardware such as LTE-M SIM card become available to the project. Successful implementation of LoRa, Sigfox or LTE-M should further reduce not only the package power consumption but also its package size, making the solution more suitable for applications where limited space and power supply are mandatory conditions. For example, a current Raspberry Pi based implementation will require at least four pieces of hardware components to accomplish the integrated IoT platform for our GNSS device. In addition, the overall power consumption for the whole Raspberry Pi package will be significantly higher than any of the microcontroller-based implementation. Such extra power draw will require a larger capacity battery, which further increases the overall package weight as well as space required by it.

Currently, the software implementation would only work for the following hardware composition: NEO-M8L GNSS device connect to a specific pin combination (P3 + P4) on the Pycom FiPy board. This severely limits the potential application scenarios. Also, in the current version of software implementation, many of the variables are hard-coded within the python code, which makes maintaining and modifying the project difficult. For example, if the bundled Wi-Fi hotspot is modified, the user would need to access the FiPy board through USB and VS Code IDE in order to modify the Wi-Fi Hotspot SSID and password. Such practice is inconvenient, to say the least. The same can be applied to the login credentials of the remote MQTT broker, which is also hardcoded in the MicroPython code. In the follow-up project, these configurations should be made flexible by implementing user input within the web application, which will enable the user to type relevant data on the web app and submit these data to the FiPy web server.

Overall, the completion of the project establishes a solid foundation for advanced microcontroller based GNSS IoT framework as well as its future researches and developments.

CONCLUSION

Overall, the main purpose of the project was to create a MicroPython based integrated software package that serves as a framework for IoT enabled GNSS devices, which will enable applications such as remote location tracking and real-time position reporting through the MQTT protocol. In addition, the software implementation will enable the user to modify the settings/configurations wirelessly on the fly. The main target user of the designed artifact of the project deliverable is tech admins, IoT aware users who have a tech-savvy background.

Although there are other similar software implementations which achieves similar outcomes when compared to this project, there are significant differences that differentiate the project from the rest of the other solutions. Firstly, our solution is purely written in MicroPython, which is essential for speedy development and deployment as it is an interpreting language which does not require any compilation. This attribute will significantly reduce the amount of time required for software development, which boosts productivity. Secondly, our choice of hardware makes a small footprint solution package possible. As the Pycom FiPy integrates all key functionality except the GNSS module into a small single board package while drawing less than a fraction of the power, it is possible to build a significantly more compact GNSS tracking device with long battery runtime. Such improvement will enable advanced embedded GNSS solutions.

In summary, the current implementation of the software package is based on the regular NEO-M8L GNSS module which does not support the advanced high precision RTK positioning system. However, with the MicroPython powered IoT framework delivered by the project, it is suggested as the next step to implement the ZED-F9P RTK GNSS module based on the outcome of this project. Likewise, the MQTT protocol supports bidirectional transmission, which can be modified to deploy remote commands to the FiPy hardware, therefore modifying its configuration remotely. Additional work needs to be done to fully exploit the potential of this internet-enabled MicroPython powered IoT framework for GNSS devices package. The project successfully delivered all promised artifacts listed on the second version of the product backlog, which will serve as a solid foundation for advanced microcontroller based GNSS IoT framework as well as its future researches and developments as mentioned above.

REFERENCES

Al-Shaery, A., Zhang, S., & Rizos, C. (2013). An enhanced calibration method of GLONASS inter-channel bias for GNSS RTK. *GPS solutions, 17*(2), 165-173.

Allafi, I., & Iqbal, T. (2017). *Design and implementation of a low cost web server using ESP32 for real-time photovoltaic system monitoring.* Paper presented at the 2017 IEEE Electrical Power and Energy Conference (EPEC).

Bell, C. (2017). Introducing MicroPython. In *MicroPython for the Internet of Things* (pp. 27-57): Springer.

Bos, J.-C. (2019). MicroWebSrv.

Du, C. (2009). *Empirical study on college students' debugging abilities in computer programming.* Paper presented at the 2009 First International Conference on Information Science and Engineering.

Gebre-Egziabher, D., & Gleason, S. (2009). *GNSS applications and methods*: Artech House.

Gerke, M., & Przybilla, H.-J. (2016). Accuracy analysis of photogrammetric UAV image blocks: Influence of onboard RTK-GNSS and cross flight patterns. *Photogrammetrie-Fernerkundung-Geoinformation, 2016*(1), 17-30.

GPS WORLD. (February 5, 2013). Why the Price of Precision Receivers Will Drop. Retrieved from https://www.gpsworld.com/why-the-price-of-precision-receivers-drop/

Hartwig, S., Stromann, J.-P., & Resch, P. (2002). Wireless microservers. *IEEE Pervasive Computing, 1*(2), 58-66.

He, Q., Segee, B., & Weaver, V. (2016). *Raspberry Pi 2 B+ GPU Power, Performance, and Energy Implications.* Paper presented at the 2016 International Conference on Computational Science and Computational Intelligence (CSCI).

Kodali, R. K., & Mahesh, K. S. (2016). *Low cost ambient monitoring using ESP8266.* Paper presented at the 2016 2nd International Conference on Contemporary Computing and Informatics (IC3I).

Maier, A., Sharp, A., & Vagapov, Y. (2017). *Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things.* Paper presented at the 2017 Internet Technologies and Applications (ITA).

Maksimović, M., Vujović, V., Davidović, N., Milošević, V., & Perišić, B. (2014). Raspberry Pi as Internet of things hardware: performances and constraints. *design issues, 3*(8).

Pycom. (2019). FiPy.

Singh, K. J., & Kapoor, D. S. (2017). Create Your Own Internet of Things: A survey of IoT platforms. *IEEE Consumer Electronics Magazine, 6*(2), 57-68.

Stempfhuber, W., & Buchholz, M. (2011). A precise, low-cost RTK GNSS system for UAV applications. *Proc. of Unmanned Aerial Vehicle in Geomatics, ISPRS*.

ublox. (2019). ZED-F9P module. Retrieved from https://www.u-blox.com/en/product/zed-f9p-module

Wilmshurst, T. (2001). *An introduction to the design of small-scale embedded systems*: Palgrave.

APPENDIX

Reflections on your learning

- **What were the things/activities you thought you did best in this project?**
  Independence. As I am the only developer working on the project, there are no teammates to fall back on aside from my supervisor. As a result, I forced my self to learn individually and tried my best to solve issues that were encountered during the software development process.
- **What were the things/activities you thought you did least well in this project?**
  Report writing. I'm the type of person who can do and complete the task but struggles to describe it appropriately. I find myself lost during the final report writing from time to time, making the progress slow and inefficient. There's definitely room for improvements in terms of my writing skills.
- **Were there any specific problems or challenges you encountered? How did you handle them?**
  Mostly compatibility issues regarding the MicroPython implementation as it does not support all regular Python modules. As a result, workarounds are often needed to circumvent the issues. Aside from that, I also find time management challenging as I need to balance the number of efforts equally between the project and the rest of the other courses.
- **What did you find as the hardest part of this project?**
  Personally, it will be the 8000 words report writing. I have never written so many words individually in my life and I found the progress a lot more difficult than what I initially imagined. In addition, since it is a software development style project which means I'll be spending most of my time working on coding and testing, leaving insufficient time for report writing.
- **What was the most important thing you learned doing this project?**
  The software development project is more practical than a regular academic assignment as I require initial planning, software development, bug resolving, testing and finally evaluating the outcomes. All of the steps executed during the project are segments contribute to a working software implementation that achieves a significant objective. I think the aforementioned experience is the most important thing that I acquired after completing this project.
- **What kind of opportunities or next steps you see based on your learnings doing this project?**
  The outcome of the project can be extensively expanded with the replacement of a higher-end GNSS module which has RTK capability. The internet enabled IoT framework will be able to support the real-time network RTK correction which can improve the accuracy and precision of the GNSS module up to ± 1cm.