Importing Sensor Values and Real-World Situational Information into a Virtual Environment

Using Mbed OS and MQTT

Bachelor's Thesis
Information and Communication Technology
Spring, 2025
Danial Mousavi

HAMK
Häme University
of Applied Sciences

Abstract

DP	Information and Communication Technology
Author	Danial Mousavi
Year	2025
Subject	Importing Sensor Values and Real-World Situational Information into a Virtual Environment
Supervisor	Timo Karppinen

This thesis presents a practical approach for importing sensor values and real-world situational information into a virtual environment. The primary objective is to demonstrate how physical sensor data can be collected, processed, and transformed into a format suitable for use in digital or virtual systems using Mbed Operating System and MQTT protocol. The work explores the interaction between hardware and software components necessary to achieve seamless data integration, emphasizing the need for standardized data formats and compatible infrastructure.

A modular software architecture was designed and implemented, supported by a demonstration system that visualizes live sensor data within a virtual environment using a web-based application. Data from physical sensors was transmitted via a public MQTT broker and visually rendered in a web-based application, illustrating the end-to-end data flow from physical systems to virtual representation. The successful demonstration confirmed the technical feasibility of the proposed approach.

While the implementation validates the proposed method, it is intended to serve as a reference model rather than a definitive solution. The approach can be extended or adapted for broader use cases involving real-time data integration into a virtual environment, and it provides a foundation for further research and development in similar contexts.

Keywords	MQTT, JavaScript Object Notation (JSON), Mbed
Pages	34 pages

Table of Contents

Figures

Tables

Equations

1 Introduction

The integration of sensor data and IoT devices into virtual environments is becoming increasingly central to modern applications such as digital twins, remote monitoring, and simulation systems. Translating physical data into virtual representations enables timely insights, contextual understanding, and interactive data exploration. This thesis demonstrates the process of gathering, processing sensor and situational data for use in a virtual digitalized environment, a process applicable across various domains, from web-based mapping to immersive simulation platforms.

The objective of this work is to showcase how data can be collected from sensors and morphed into a format that can be utilized in a virtual environment. A GPS module and a 6-axis Inertial Measurement Unit (IMU) are used for this purpose. The choice of the virtual environment is intentionally kept flexible. Whether it is a two-dimensional map, a simulation engine, or a game world, the core goal is to create a successful and reliable data acquisition and transmission pipeline.

The concept behind this thesis is closely tied to a broader project involving a remote-controlled test vehicle, shown in figure 1, equipped with additional sensors and communication equipment. The goal of that project was to control the remote-controlled test vehicle from a game engine based on data imported from sensors. Due to project restructuring, it was divided into two separate thesis projects. This work focuses exclusively on the software aspect of sensor data processing and transmission. As a result, the hardware and software choices were made with these broader goals in mind.

Figure 1. The remote-controlled test vehicle used at HAMK

This thesis presents a flexible and functional approach to software and hardware for the continuous collection and transmission of physical sensor data to a digital environment. For better understanding of software implementation shown in this work, the following requirements are needed:

- C and C++ programming languages,
- string manipulation functions in C and C++ programming languages,
- Mbed Studio and Mbed operating system as a whole,
- understanding of TCP/IP internet protocol suite and commonly used network protocols (DHCP, DNS),
- basic understanding of the wireless communication technologies,
- basic understanding of the communication protocols used in embedded devices (UART, I2C),
- familiarity with electronic circuits, sensors and inner workings of the microcontrollers.

2 Hardware

Hardware has a significant impact on what software can accomplish or what kind of software solutions are even possible for a given set of hardware. For instance, a sensor that outputs data once every hundred milliseconds is better suited for applications that require fast and up-to-date data acquisition than a slower sensor that updates once per second. Using multiple sensors with different output rates introduces complexity in the application. A 16-bit microcontroller may be cost-effective, but it might not be suitable for an application that requires high performance or support for multithreading capabilities. Therefore, it is essential to thoroughly study and analyse the available hardware before designing a software solution. This section presents a detailed overview of the hardware components selected for this development project.

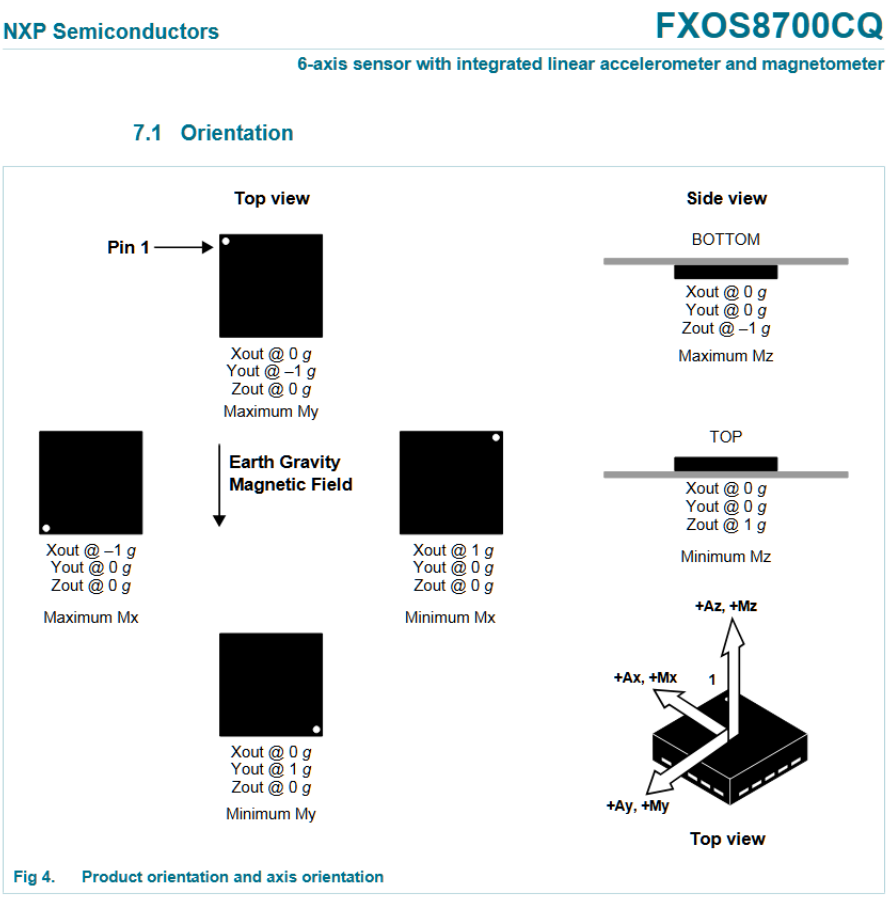
2.1 FDRM-K64F Development Board

FDRM-K64F Development Board is a low-cost platform powered by the Kinetis MK64FN1M0VLL12 microcontroller, featuring a 32-bit ARM Cortex M4 processor. The board is designed by NXP Semiconductors N.V., a global semiconductor company. It is equipped with an Ethernet interface that supports speeds of up to 100 Mbit/s. The board also includes an onboard FXO8700CQ 6-axis combo sensor. Figure 2 shows the development board, and its onboard sensor highlighted with red circle. The K64F microcontroller can be programmed using the latest versions of Mbed Operating System. (Mbed, n.d.-d)

Figure 2. FDRM-K64F development board

FXOS8700CQ is an Inertial measurement Unit (IMU) designed to be used in electronic compass (e-compass) applications. It is a 3-axis linear accelerometer and 3-axis magnetometer combined into a single Integrated Circuit (IC) package. According to the FXOS8700CQ documentation, when both sensors are enabled, the output data rate is 400Hz, which corresponds to a new data sample every 2.5 milliseconds. Figure 3, an image captured from the device's official documentation, shows the product orientation and axis orientation. (NXP Semiconductors N.V., 2017, p. 1)

Figure 3. Orientation diagrams and corresponding acceleration/magnetic field outputs for the 6-axis sensor (NXP Semiconductors N.V., 2017, p. 7)



The accelerometer measures acceleration forces applied along its axes. Earth's gravity applies a constant downward acceleration, which can be used as a reference to calculate pitch and roll angles of accelerometer. The 3-axis magnetometer operates on similar principles, but it detects only the magnetic field anomalies around it. The earth's magnetic fields can serve as a reference to calculate heading or yaw angle. (NXP Semiconductors N.V., 2017, p. 9)

The combined operation of accelerometer and magnetometer enables accurate measurements of gesture and orientation. Magnetometer's measured values are dependent on their orientation. Even slight changes in pitch or roll angle offset in measurements. Accelerometer can be used to compensate for these offsets, helping to correct the magnetometer data. (TronicsBench, n.d.)

2.2 TEL0132 module

TEL0132 is a positioning module designed and manufactured by DFRobot, a company that produces open-source hardware and educational electronics. This module is powered by AT6558 GNSS chip. The chip supports multiple Global Navigation Satellite Systems (GNSS) constellations, including but not limited to GPS, Galileo and BeiDou. Users can retrieve geolocation data in the form of standardized NMEA 0183 strings via the module's UART interface, with an update rate of 1Hz. (DFRobot, n.d.)

NMEA 0183 is a voluntary industry standard that defines an electrical interface and data protocol for communication between marine instrumentation. It is designed and maintained by the National Marine Electronics Association (NMEA), a non-profit organization. The NMEA 0183 is a standard data protocol adopted by GPS manufacturers. TEL0132 outputs data in compliance with this standard. It transmits information as structured ASCII-encoded sentences. Each sentence starts with "$" sign and ends with a cartridge return (CR) and a line feed (LF). (Raymond, n.d.)

The TEL0132 module can output a wide variety of NMEA sentences that provide wide range of information. This work focuses primarily on retrieving positioning data. Therefore, only two specific set of NMEA sentences will be used for this development work:

- GNRMC – Recommended Minimum Navigation Information, from GNSS system.
- GPRMC – Recommended Minimum Navigation Information, from GPS system.

These sentences contain essential positioning and timing information such as latitude, longitude, date and time, making them particularly relevant for the objectives of this work. The GPRMC sentence provides minimum recommended GPS-specific navigation data and remains available even in challenging conditions where multi-constellation data might be unavailable. GNRMC presents combined data from multiple GNSS constellations, offering potentially more accurate information (DFRobot, n.d.). A complete GNRMC sentence read from TLU0132 module is written in the following line in form of a string:

"$GNRMC,060229.000,A,6015.13435,N,02500.35679,E,0.00,343.69,110525,,,A,V*07\r\n"

The breakdown of GNRMC message is shown in table 1 with description for each field. The green coloured fields are geolocation information that will be used for this work.

Table 1. Decoded GNRMC Message retrieved from TLU0132 module

Field	Value	Description
$	$	Indicates the Start of NMEA message
ID	GNRMC	Type of NMEA message
1	060229.000	UTC time (HHMMSS.SSS)
2	A	Status
3	6015.13435	Latitude (DDMM.MMMMM)
4	N	Latitude Direction
5	02500.35679	Longitude (DDDMM.MMMMM)
6	E	Longitude Direction
7	0.00	Speed over Ground
8	343.69	Course over Ground
9	110525	UTC Date
10	-	Magnetic Variation
11	-	Magnetic Variation Direction
12	A	Mode Indicator
13	V	Navigation Status
14	*07	Checksum

15	CRFL (\r\n)	Indicates the end of NMEA message

The GNRMC and GPRMC sentences adhere to the same standardized structure defined for Recommended Minimum Navigation Information (RMC) within the NMEA 0183 protocol. The distinction between them lies solely in the talker ID. (Raymond, n.d.)

2.3 ZYXEL NBG-418N v2 Router

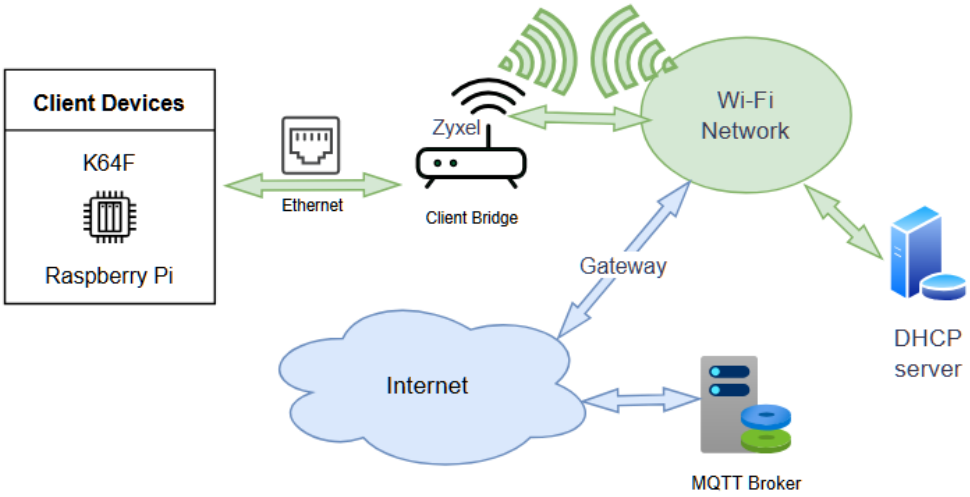
One of the requirements for this work was to investigate how the remote-controlled test vehicle can be equipped with a device that enables it to connect to the existing Wi-Fi network provided by the Riihimäki campus of Häme University of Applied Sciences. This device should function as a client bridge, connecting client devices, sensors, and systems installed on the vehicle to the campus gateway network and, ultimately, to the internet.

The NBG-418N v2 is a compact and affordable home router developed by Zyxel, a company specializing in the manufacturing of networking equipment. The router operates on just 5 volts. During testing and hardware evaluation, it draws a maximum current of 500 mA under heavy load, making it suitable for prototyping with electronic devices. Its compact small size makes it easy to install on the remote-controlled test vehicle.

There was other hardware choices also considered such as using ESP8266 or ESP32 with specialized firmware installed to function as Client Bridge. The ESP series of chips require extensive setup and do not include an Ethernet interface, making it challenging to connect other devices present on the test vehicle. NBG-418N provides a simple web-based configuration that can be accessed through a web browser. In addition, the router provides additional modes of operations which can be set up with a few clicks of a mouse.

The Zyxel router has many modes of operation, including the Client Bridge mode. In this mode, the router needs to be assigned with a static IP address on the same domain as the Wi-Fi network. All connected client devices are also to have IP addresses of same domain. Figure 4 illustrates the role of the Client Bridge device in a network. (Zyxel, 2019)

Figure 4. A diagram of a network presenting how Client Bridge mode works



To the client devices connected to the router running in Client Bridge mode, the Zyxel router will be transparent, meaning that they will assume that they are connected to the actual bridged Wi-Fi network. The client devices can also acquire an IP address via the DHCP protocol, provided that a DHCP server is available. The Zyxel router will only need to be configured once. The router will stay in this mode of operation as long as the device is not reset. (Zyxel, 2019)

3 Software

With the hardware parameters established, the focus shifts to software design and implementation. The setup involves two sensors with distinct output characteristics each providing data in a different format and at different rates. The objective is to transform the raw data from these sensors into a standardized format suitable for import into virtual environments. Achieving this requires a series of structured steps, tailored to the specific characteristics of each sensor device. The process is outlined as follows:

1. Connect, initialize, and configure the hardware.
2. Read the output of the connected device.
3. Evaluate and process data.
4. Convert data to a recognizable format for external use.
5. Send data to its destination.

The raw data provided by these sensors is not immediately usable. It must first be processed and transformed into a structured format that can be easily interpreted and consumed by other systems. Since our target consumer systems are virtual environments, the chosen format must be both available and readily usable within such contexts.

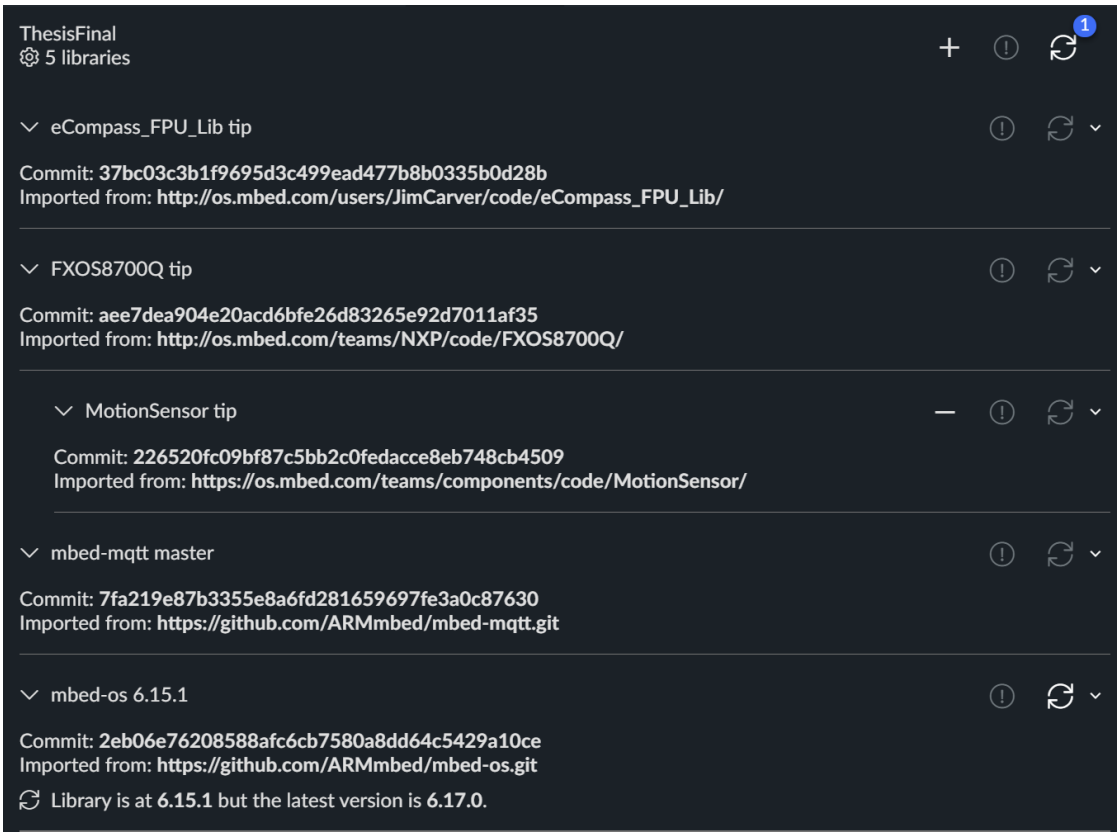
3.1 Mbed OS

Mbed Operating System (Mbed OS) is an open-source operating system designed for IoT devices, developed by Arm Ltd, a semiconductor and software design company headquartered in Cambridge, United Kingdom. ARM Ltd is best known for designing the Arm architecture, a family of energy-efficient processor architectures widely used in embedded systems. Mbed OS is a real time operating system that is built specifically for embedded systems that are powered by Arm based microcontrollers. (Mbed, n.d.-e)

Mbed Studio is the official desktop Integrated Development Environment (IDE) for developing with Mbed OS. Mbed Studio provides a simple interface, built-in compiler, library management, and tools to make it easier to write, build, and debug program for supported microcontrollers. This is the tool that was used to create the microcontroller software for this development work. (Mbed, n.d.-f)

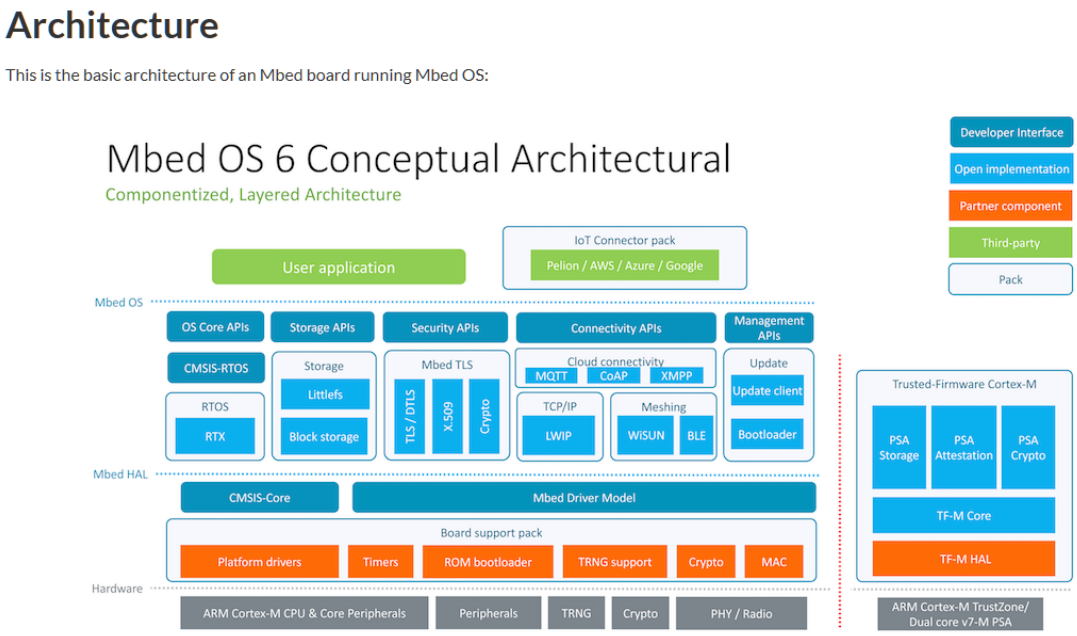
The task of reading, managing and evaluating the sensors' data was assigned to the microcontroller on FDRM-K64F development board. Mbed OS was used to manage and program the microcontroller. Mbed OS includes a set of built-in libraries that can be readily used in created projects. In addition to these, community-driven external libraries are also available, but they need to be manually imported into the Mbed Studio workspace. Figure 5 presents all the libraries imported in the Mbed Studio into the project used in this development work.

Figure 5. A picture showing all imported libraries to the Mbed project



Mbed OS provides a hardware abstraction layer (HAL) that standardizes access to essential microcontroller features, like timers and communication interfaces. This abstraction allows developers to write applications using consistent Application Programming Interfaces (APIs), regardless of the specific hardware in use. When a program is compiled, the appropriate libraries and drivers for common microcontroller components such as I2C, UART, or Ethernet interfaces are automatically included. These hardware features are represented in software through class-based objects, allowing developers to interact with physical components as if they were regular software constructs. Figure 6 provides an overview of Mbed system architecture. (Mbed, n.d.-a)

Figure 6. A screenshot taken from the official documentation of Mbed Operating System, providing an overview of the system's architecture of Mbed OS (Mbed, n.d.-a)



3.1.1 Data extraction and structuring

Jim Carver, a member of Mbed community has published an external library code named "eCompass_lib", that turns FXOS8700CQ sensor into a e-compass. The NXP has also implemented an external library code that provides all necessary functions to setup, initialize and interface with the FXO8700CQ. Later in this work, these external libraries will be referred to as the e-compass and the NXP libraries.

Through the use of the external libraries, much of the low-level work associated with the FXOS8700CQ sensor including hardware initialization, hardware configuration, calibration and handling the raw data from the sensor is abstracted away, allowing the development time to be focused on higher-level application logic. (Carver, n.d.)

The external libraries for the FXOS8700CQ sensor are written for much older version of Mbed OS, specifically Mbed 2. This work requires the use of Mbed 6. This however does not cause an issue. The issue can be resolved by replacing all the used deprecated functions in the libraries with the equivalent ones provided by the Mbed 6.

The FXOS8700CQ sensor is physically connected to I2C1 interface of the microcontroller via PCB traces on the FDRM-K64F development board. The I2C1 interface uses pins PTE25 and PTE24 of the microcontroller. (Mbed, n.d.-d)

An instance of I2C using the same pins must be defined and passed to the initialization functions provided by the NXP library. Figure 6 provides a snippet of code demonstrating how FXOS8700CQ hardware initialization can be achieved using NXP library.

Figure 7. Initializing FXOS8700CQ 6-axis sensor using the NXP library

```
#include "FXOS8700Q.h"


I2C i2c1(PTE25, PTE24);
FXOS8700QAccelerometer acc(i2c1, FXOS8700CQ_SLAVE_ADDR1);
FXOS8700QMagnetometer mag(i2c1, FXOS8700CQ_SLAVE_ADDR1);


int main(){
    acc.enable();
    mag.enable();

    acc.getAxis( acc_raw);
    mag.getAxis( mag_raw);
}
```

The e-compass library provides the means to turn the raw input of FXOS8700CQ sensor into useful usable data. The purpose of e-compass is to show the heading of magnetic north pole or the yaw (y) axis of the sensor. The output of the e-compass library is given in degrees. It also provides the means to calibrate the sensor. Figure 8 shows how to implement e-compass library in code.

Figure 8. Turning raw FXOS8700CQ sensor data into e-compass

```
#include "FXOS8700Q.h"
#include "eCompass_Lib.h"

eCompass compass;
motion_data_counts_t mag_raw;
motion_data_counts_t acc_raw;

void read_imu_sensor() {
    compass.init();
    while (true) {
        // get raw data from the sensors
        acc.getAxis(acc_raw);
        mag.getAxis(mag_raw);
        compass.run(acc_raw, mag_raw); // calculate the eCompass
        compass.calibrate();
        ThisThread::sleep_for(500ms);
    }
}
```

The TEL0132 module does not require hardware initialization or configuration. The device's only requirement is to be powered by a 3.3- or 5-volt source. As soon is its powered, it will start outputting stream of text consisting of NMEA sentences. The TEL0132 has a red LED that will blink when geolocation data is complete, and it is in a usable state. If geolocation data is not complete or unavailable, the LED will be continuously on and GNRMC fields containing positioning data will be either empty or incomplete. (DFRobot, n.d.)

The TEL0132 requires a Universal Asynchronous Receiver / Transmitter (UART) interface for transmission of NMEA sentences to the microcontroller. An UART interface with appropriate settings must be defined. Figure 9 demonstrates this implemented in code.

Figure 9. Defining an UART interface for the use with TEL0132 module

```
#include "mbed.h"

//UART 3, TX: PTC17, RX: PTC16
static BufferedSerial GpsSerial(PTC17, PTC16);

int main(){

    GpsSerial.set_baud(9600);
    GpsSerial.set_format(
        /* bits */ 8,
        /* parity */ BufferedSerial::None,
        /* stop bit */ 1
    );
    ThisThread::sleep_for(1s);
}
```

The extraction of data from GNRMC or GPRMC sentences is a straightforward process due to the well-defined structure of these NMEA sentence types. By employing basic string comparison techniques, it is possible to identify and extract the necessary data fields.

The procedure involves reading the complete output stream from the TEL0132 module and scanning for the beginning and end of the relevant sentence. Given that NMEA sentences may not be received at consistent time intervals, the implementation must account for such variability. As a result, it is essential to validate both the start and end markers of each sentence to ensure data integrity during parsing. Figure 10 provides a code example, where NMEA stream is read from the UART interface. The example also validates the NMEA sentences.

Figure 10. Example code for finding and reading GNRMC or GPRMC sentence

```
void Read_gps(){

    if (GpsSerial.readable()) {
        num = GpsSerial.read(tmpBuf, 1);
        gpsRxLength = gpsRxLength +  num;
        if (gpsRxLength == gpsRxBufferLength)RST_GpsRxBuffer();
        snprintf(gpsRxBuffer, gpsRxLength, "%s%s", gpsRxBuffer, tmpBuf);

    }
    char *GPS_DATAHead;
    char *GPS_DATATail;

    if ((GPS_DATAHead = strstr(gpsRxBuffer, "$GPRMC,")) != NULL ||
        (GPS_DATAHead = strstr(gpsRxBuffer, "$GNRMC,")) != NULL)
        {

            if (((GPS_DATATail = strstr(GPS_DATAHead, "\r\n")) != NULL) &&
                (GPS_DATATail > GPS_DATAHead))
            {
                //Now to be parsed by parse_gps()
                memcpy(GpsData.GPS_DATA, GPS_DATAHead, GPS_DATATail - GPS_DATAHead);
                GpsData.GetData_Flag = true;

                RST_GpsRxBuffer();
            }
        }
}
```

Once a complete and validated GNRMC or GPRMC sentence has been identified, the relevant fields can be extracted using basic string parsing techniques. A dedicated data structure is defined to store the extracted information for subsequent use. This structure may also include status flags to represent the parser's state and indicate the validity or relevance of the extracted data. An implementation example of this process is presented in Figure 11.

Figure 11. Example code on how to extract data from GNRMC or GPRMC sentence and storing it into a data structure for later use

```
struct
{
  char GPS_DATA[80];
  bool GetData_Flag;       //Get GPS data flag bit
  bool ParseData_Flag;     //Parse completed flag bit
  char UTCTime[11];        //UTC time
  char latitude[11];       //Latitude
  char N_S[2];             //N/S
  char longitude[12];      //Longitude
  char E_W[2];             //E/W
  bool Usefull_Flag;       //If the position information is valid flag bit
} GpsData;

void parse_gps(){
    // printf("Entered parse_GpsDATA()\n");
    char *subString;
    char *subStringNext;
    if (GpsData.GetData_Flag)
    {
    GpsData.GetData_Flag = false;
    for (int i = 0 ; i <= 6 ; i++)
    {
      if (i == 0)
      {
        if ((subString = strstr(GpsData.GPS_DATA, ",")) == NULL)
          Error_Flag(1);    //Analysis error
      }
      else
      {
        subString++;
        if ((subStringNext = strstr(subString, ",")) != NULL)
        {
          char usefullBuffer[2];
          switch(i)
          {
            case 1:memcpy(GpsData.UTCTime, subString, subStringNext - subString);break;    //Get UTC time
            case 2:memcpy(usefullBuffer, subString, subStringNext - subString);break;        //Get position status
            case 3:memcpy(GpsData.latitude, subString, subStringNext - subString);break;   //Get latitude information
            case 4:memcpy(GpsData.N_S, subString, subStringNext - subString);break;        //Get N/S
            case 5:memcpy(GpsData.longitude, subString, subStringNext - subString);break;  //Get longitude information
            case 6:memcpy(GpsData.E_W, subString, subStringNext - subString);break;        //Get E/W

            default:break;

          }
          subString = subStringNext;
          GpsData.ParseData_Flag = true;
          if(usefullBuffer[0] == 'A')
            GpsData.Usefull_Flag = true;
          else if(usefullBuffer[0] == 'V')
            GpsData.Usefull_Flag = false;
        }

        else
        {
          Error_Flag(2);    //Analysis error
        }
      }
    }
  }
}
```

The extracted and validated data from GPRMC or GNRMC sentences can be formatted into different representations for further use depending on the end user requirements.

3.1.2 Degree Decimal Minute to Decimal Degree conversion

NMEA sentences represent geographic coordinates using the Degrees and Decimal Minutes (DDMM.MMMM or DDDMM.MMMM) format for both latitude and longitude (Trimble, 2004, p. 7). Common mapping services, such as Google Maps, a web-based Geographic Information System (GIS) platform developed by Google LLC use a different format to represent geospatial and geolocation data. Mapping services typically use decimal degrees (DD) to express latitude and longitude coordinates.

To make the sensor data usable for GIS services, the geolocation data can be converted to Decimal Degrees. The official documentation for TEL0132 modules provides an example of such conversion (DFRobot, n.d.). Based on the example, an equation can be formed that is illustrated in equation 1.

Equation 1. Equation used for Degrees and Decimal Minutes to Decimal Degrees conversion

$$Decimal\ Degrees\ (DD) = \ Degrees + \frac{Minutes}{60}$$

Based on the equation, a function can be written to convert raw coordinate data extracted from an NMEA sentence into decimal degrees. The function must separate the degree and minute components from each value, then apply a simple arithmetic operation illustrated in the equation 1. The function can also account for hemisphere direction by assigning negative values to coordinates located in the southern or western hemispheres. Figure 12 provides a simple implementation of this process.

Figure 12. Example code on how to convert from Degrees and Decimal Minutes to Decimal Degrees format

```
struct
{
  char GPS_DATA[80];
  bool GetData_Flag;      //Get GPS data flag bit
  bool ParseData_Flag;    //Parse completed flag bit
  char UTCTime[11];       //UTC time
  char latitude[11];      //Latitude
  char N_S[2];            //N/S
  char longitude[12];     //Longitude
  char E_W[2];            //E/W
  bool Usefull_Flag;      //If the position information is valid flag bit
} GpsData;


//Converts from Degrees and Decimal Minutes (DDMM.MMMM) to Decimal Degrees (DD.dddd) format
void gpsFormatter(char* rawLat, char* latDir, char* rawLong, char* longDir) {
    char buf[12] = {0};
    int intLat, intLong;
    float fLat, fLong;

    // Latitude degrees
    strncpy(buf, rawLat, 2);
    buf[2] = '\0';
    intLat = atoi(buf);

    // Latitude minutes
    strcpy(buf, rawLat + 2);
    fLat = atof(buf);

    // Longitude degrees
    strncpy(buf, rawLong, 3);
    buf[3] = '\0';
    intLong = atoi(buf);

    // Longitude minutes
    strcpy(buf, rawLong + 3);
    fLong = atof(buf);

    // Convert to decimal degrees
    fLat = intLat + (fLat / 60.0f);
    fLong = intLong + (fLong / 60.0f);

    // Apply hemisphere
    if (latDir[0] == 'S') fLat = -fLat;
    if (longDir[0] == 'W') fLong = -fLong;

    // Store result
    snprintf(GpsData.latitude, sizeof(GpsData.latitude), "%.6f", fLat);
    snprintf(GpsData.longitude, sizeof(GpsData.longitude), "%.6f", fLong);
}
```

Such conversions result in a data format that is easier for end users to integrate into mapping applications and geographic calculation. Alternatively, the conversion can be deferred to the end user or the target system, depending on design requirements. It is important to note that each data transformation or format conversion introduces computational overhead, which can be significant in resource-constrained embedded environments. Therefore, unless required, it may be more efficient to delegate this task to the end user.

3.1.3 JSON

JSON (JavaScript Object Notation) is a lightweight data format commonly used for exchanging information between systems. It plays a crucial role in various fields, including game development and web development. Often, structured data needs to be transmitted to external systems for further processing or integration. By converting sensor data into a recognizable JSON format, it can be easily shared and understood across different platforms and environments. This flexibility makes JSON an essential tool for enabling seamless communication between systems, such as a game engine and its connected components. (D'mello & Sriparasa, 2018)

Figure 13 illustrates how sensor data stored in a C program can be represented in equivalent key-value pair JSON format, making it readable and usable across different platforms and applications.

Figure 13. Sensor data structures being represented in key-value pairs using JSON format

In C programming language, JSON data can be constructed simply by using strings. String manipulation and formatting functions provided by the C and C++ libraries allow us to insert sensor values into a predefined JSON structure using format specifiers. This approach provides a lightweight method for generating JSON without the need for external libraries. Figure 14 illustrates an example of how to implement a JSON representation of data using a string formatting function in the C programming language.

Figure 14. Example on how to format predefined JSON structure string using string format specifiers in C programming language

```
char jsonStrFormIMU[] =
    R"({"imu":{"pitch":"%d", "roll":"%d", "yaw":"%d", "time": "%d"}})";


sprintf(msg, jsonStrFormIMU, axis6.pitch, axis6.roll, axis6.yaw, axis6.timestamp);
```

3.1.4 Threads

Threads are units of execution within a program that allow multiple tasks to run at the same time. They provide a way to achieve concurrency, meaning different parts of a program can operate independently and simultaneously. Each thread in Mbed OS is scheduled and managed by kernel's scheduler, which handles timing, context switching, and task prioritization. Mbed OS provides a set of Application Programming Interfaces (APIs) to create, execute and manage threads. (Mbed, n.d.-c)

In a program that interacts with hardware and communicates over a network, one thread can be dedicated to collecting sensor data, while another handles communication with a server or broker. This separation keeps tasks organized and responsive. Using threads allows for development of modular and responsive applications where different parts of the system operate independently but cooperatively.

Since multiple threads may access the same shared variables concurrently, it is essential to implement a mechanism to prevent simultaneous access of shared variables, which could lead to data corruption or undefined behaviour. Mbed OS provides various synchronization primitives to handle such scenarios. One such mechanism is a mutex (mutual exclusion), which ensures that only one thread can access a critical section of code or shared resource at a time (Mbed, n.d.-h). Figure 15 illustrates an example in which a thread is created in Mbed OS, and access to the global variables is safely managed using a mutex.

Figure 15. Example on how to create and use a thread and a mutex in a Mbed application

```
#include "mbed.h"
#include "FXOS8700Q.h"
#include "eCompass_Lib.h"

eCompass compass;
motion_data_counts_t mag_raw;
motion_data_counts_t acc_raw;
Mutex imu_mutex;
Thread sensorThread;

void read_imu_sensor() {
    compass.init();
    while (true) {
        // get raw data from the sensors
        acc.getAxis(acc_raw);
        mag.getAxis(mag_raw);
        compass.run(acc_raw, mag_raw); // calculate the eCompass
        compass.calibrate();
        ThisThread::sleep_for(500ms);
    }
}

int main(){
    acc.enable();
    mag.enable();
    sensorThread.start(read_imu_sensor);
    while (true) {
        // main loop
        imu_mutex.lock();
        printf("X: %d, Y: %d, Z: %d\n", acc_raw.x, acc_raw.y, acc_raw.z);
        printf("Mag X: %d, Y: %d, Z: %d\n", mag_raw.x, mag_raw.y, mag_raw.z);
        imu_mutex.unlock();
        ThisThread::sleep_for(1s);

    }
}
```

3.1.5 Network Socket

In network communication, a socket serves as one endpoint of a bidirectional communication channel between two systems. It is an abstraction that allows programs to send and receive data over a network using standardized interfaces, such as Ethernet.

Each socket is uniquely identified by a combination of an IP address and a port number, enabling reliable communication between networked devices. (University of Glasgow, n.d.)

To use a socket in an Mbed program, an instance of a network interface must be created. The created instance can then be passed to the created sockets for further use. Socket access to the underlying networking hardware is managed by the kernel of the Mbed operating system (Mbed, n.d.-i). When an instance of communication hardware is created, Mbed operating system's kernel does many initializations under the hood. Dynamic Host Configuration Protocol (DHCP) and Domain Name System (DNS) are enabled by default for instantiated network interfaces during the initialization (Mbed, n.d.-b). Figure 16 provides a simple example that illustrates how the Ethernet interface can be initialized and passed to a created TCP socket.

Figure 16. Example on how to initialize Ethernet interface for the use by TCP sockets

```
#include "mbed.h"
#include "MQTTClientMbedOs.h"
#include "EthernetInterface.h"

#define MBED_CONF_APP_MQTT_BROKER_HOSTNAME          "test.mosquitto.org"
#define MBED_CONF_APP_MQTT_BROKER_PORT              1883
//network, socket and mqtt
EthernetInterface eth;
TCPSocket socket;

// ip addresses
SocketAddress MQTTBroker;       //socket struct to store retrieved ip address from DNS
SocketAddress DeviceAddress;    //socket struct to store device ip address via DHCP
SocketAddress BuffAddr;         //socket struct to store netmask and gateway address

MQTTClient client(&socket);
MQTTPacket_connectData mqttData;


void eth_connection_init(){
    // Bring up the ethernet interface and connect to network, HDCP and DNS enabled by default.
    eth.connect();

    //gather network address, netmask and gateway address and show them
    eth.get_ip_address(&DeviceAddress);
    ThisThread::sleep_for(1s);

    printf("IP address: %s\n", DeviceAddress.get_ip_address() ? DeviceAddress.get_ip_address() : "None");

    eth.get_gateway(&BuffAddr);
    printf("Gateway address: %s\n", BuffAddr.get_ip_address() ?BuffAddr.get_ip_address() : "None");

    eth.get_netmask(&BuffAddr);
    printf("Subnetmask:: %s\n",  BuffAddr.get_ip_address() ? BuffAddr.get_ip_address() : "None");
    ThisThread::sleep_for(1s);

    // Tell the TCP socket to use the instantiated Ehternet Interface
    socket.open(&eth);
    ThisThread::sleep_for(2s);

}

int main() {
    // Initialize the ethernet connection
    eth_connection_init();
    eth.gethostbyname(MBED_CONF_APP_MQTT_BROKER_HOSTNAME, &MQTTBroker, NSAPI_IPv4, "Ethernet"); // use DNS to get the IP V4 address of the MQTT broker

    int ret = socket.connect(MQTTBroker, MBED_CONF_APP_MQTT_BROKER_PORT);   // connect to the MQTT broker
    while (ret != 0) {
        printf("Connection failed, retrying...\n");
        socket.close();
        socket.open(&eth);
        ret = socket.connect(addr, 80);
        ThisThread::sleep_for(1s);
    }
}
```

In multithreaded applications, it is common for each thread to maintain its own socket. This separation allows threads to independently manage communication without contention or interference. By assigning a dedicated socket to each thread, data transfer to distinct destinations becomes more organized and efficient. Importantly, from the perspective of the socket, each instance functions as a client, acting as the communication endpoint that initiates or maintains a connection with a remote server.

3.2 MQTT

MQTT protocol is a lightweight, publish-subscribe messaging protocol designed for constrained devices and low-bandwidth, high-latency, or unreliable networks. It operates on top of the Transmission Control Protocol (TCP), leveraging TCP's reliable, connection-oriented data stream to ensure ordered and complete delivery of messages. The use of TCP as the transport layer enables MQTT to provide dependable communication between networked systems, particularly in Internet of Things (IoT) applications. (OASIS Open, 2019)

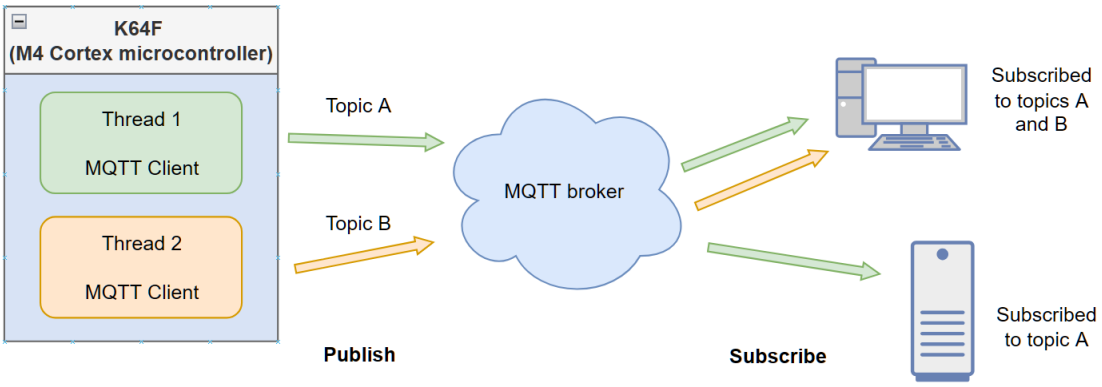
MQTT employs a client–broker architecture. Clients are uniquely identifiable devices or applications that publish messages to specific topics or subscribe to topics to receive relevant messages. The broker, serving as an intermediary, manages message distribution and ensures that published messages are forwarded to all clients subscribed to the corresponding topics. This decoupled design enables asynchronous and scalable communication across multiple clients. (OASIS Open, 2019)

A topic is structured using hierarchical levels separated by forward slashes ("/"). Each topic must consist of at least one level and must not contain empty segments. Topics are used to categorize and organize data generated by sensor devices (HiveMQ, n.d.). For example, topic structures relevant to sensor data in the context of this work may include:

- hamk/iot/devices/truck/sensors/gps, for the TEL0132 module,
- hamk/iot/devices/truck/sensors/imu, for the FXOS8700CQ sensor.

The publish-subscribe model in MQTT contrasts with traditional client–server interactions. Instead of sending data directly from sender to receiver, publishers send data to the broker, which then routes the data to subscribers. This architecture is particularly advantageous in IoT systems, where devices may intermittently connect and disconnect from the network. Figure 17 visually illustrates the MQTT publish-subscribe communication architecture.

Figure 17. MQTT publish-subscribe communication architecture



In the context of program development in the Mbed OS, MQTT support is provided through a C++ MQTT client library. This library interfaces with Mbed operating system's networking stack, using TCP sockets to establish and manage connections between MQTT clients and the broker (Mbed, n.d.-g). Each MQTT client within an application is typically bound to a dedicated TCP socket, which represents an endpoint for bidirectional communication. Since sockets are abstracted representations of network connections, multiple MQTT clients can be instantiated within the same or different threads, with each socket-client pair operating independently. This design facilitates flexible and concurrent message publishing or subscription operations within a single execution context.

To create a MQTT client, an instance of MQTT client must be created. An instance of a socket is passed to the created client instance. MQTT client library uses serialization and deserialization for transmission and reception of data that is abstracted away by the library. The MQTT client library provides data structures that is used for client connections and messages. MQTT client library provides all the necessary functions to connect to a broker and for topic subscription or publication. Figure 18 illustrates example code where multiple instances of MQTT clients are created and are used for publishing data to a broker.

Figure 18. Example code demonstrating the use of MQTT Client library in Mbed OS

```
#include "mbed.h"
#include "MQTTClientMbedOs.h"
#include "EthernetInterface.h"

EthernetInterface eth;
TCPSocket socket_forGPS;
TCPSocket socket_forIMU;
MQTTClient gpsClient(&socket_forGPS);
MQTTClient imuClient(&socket_forIMU);
MQTTPacket_connectData mqttDataForGpsClient;
MQTTPacket_connectData mqttDataForImuClient;

mqttDataForGpsClient.clientID.cstring = "UNIQUE_CLIENT_ID_GPS";
mqttDataForImuClient.clientID.cstring = "UNIQUE_CLIENT_ID_IMU";

void PublishGpsMsg(char* topic, char* buf){
    if (gpsClient.isConnected() == false) {
        printf("Connection to Broker lost: %s\n", __PRETTY_FUNCTION__);
        gpsClient.disconnect();
        reconnect_socket(3, &socket_forGPS, &eth);
        gpsClient.connect(mqttDataForGpsClient);
    }

    MQTT::Message msg;
    msg.qos = MQTT::QOS0;
    msg.retained = false;
    msg.dup = false;
    msg.payload = (void*)buf;
    msg.payloadlen = strlen(buf);
    gpsClient.publish(topic, msg);
    gpsClient.yield(1000);
}

void PublishImuMsg(char* topic, char* buf){
    if (imuClient.isConnected() == false) {
        printf("Connection to Broker lost: %s\n", __PRETTY_FUNCTION__);
        imuClient.disconnect();
        reconnect_socket(3, &socket_forIMU, &eth);
        imuClient.connect(mqttDataForImuClient);
    }
    MQTT::Message msg;
    msg.qos = MQTT::QOS0;
    msg.retained = false;
    msg.dup = false;
    msg.payload = (void*)buf;
    msg.payloadlen = strlen(buf);
    imuClient.publish(topic, msg);
    imuClient.yield(300);
}
```

The MQTT broker plays a pivotal role in ensuring reliable and structured communication between clients in an MQTT network. As the central node in the publish-subscribe architecture, the broker is responsible for receiving all published messages, filtering them by topic, and distributing them to clients subscribed to those topics. This decouples the sender and receiver, enabling asynchronous and scalable message exchange.

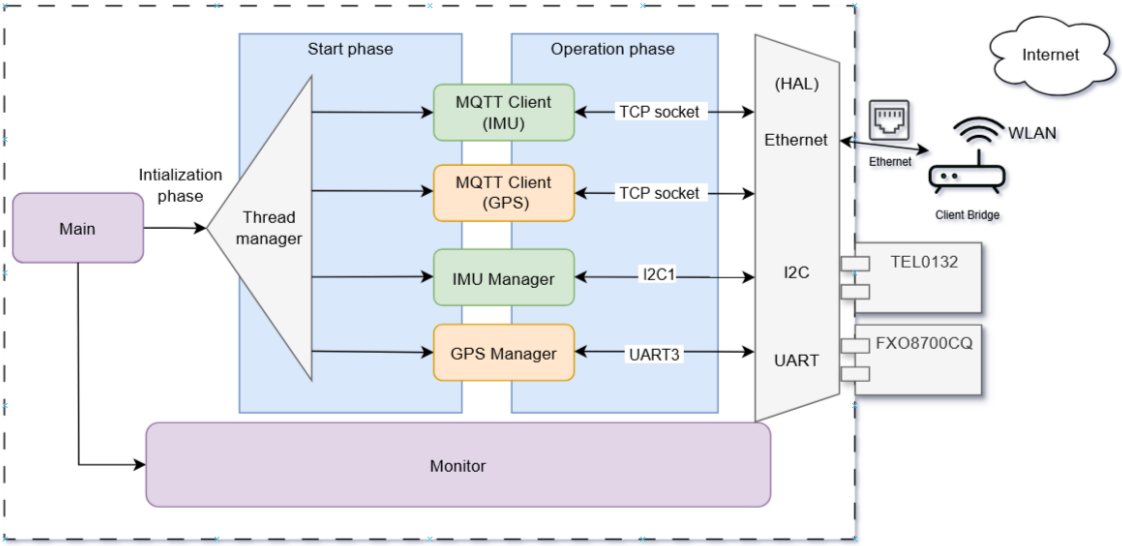
The broker's performance directly affects the efficiency and latency of data transfer, as well as the system's ability to handle large numbers of concurrent connections and topic hierarchies. Moreover, the broker enforces topic-based access and routing rules, which determine how granular or broad the message distribution can be thereby influencing the design and behaviour of the entire messaging system.

For demonstration and prototyping purposes of this development work, a publicly accessible MQTT broker instance provided by the Eclipse Mosquitto project will be utilized. This service supports multiple MQTT connection types, including standard TCP, secure TLS, and Web Sockets. While it is intended for testing and development and does not offer service guarantees or persistent storage, it enables rapid experimentation with MQTT-based applications in a live environment without requiring local broker setup. Users can publish topics of their choice without limitation. (Eclipse Foundation, n.d.)

3.3 Software Architecture

With a comprehensive understanding of both hardware and software components, a robust software solution can be designed. Each software element corresponding to a specific sensor can be encapsulated within its own dedicated thread, enabling modularity and concurrent processing. This approach forms the basis of the software architecture underpinning our solution for this development work. Figure 19 provides a visualized map of the software architecture design for this development work. It provides a visual representation of how all software modules and hardware components are integrated, highlighting their interactions and data flow within the system.

Figure 19. The software architecture developed for this project



The software architecture is designed to be both flexible and scalable, allowing for the integration of additional hardware components if required. In the accompanying diagram (Figure 19), threads sharing the same colour represent the use of common resources. Appropriate synchronization mechanisms, such as the use of a mutex, are essential to ensure safe and concurrent access. To preserve the operational integrity of each software component and their communication with external systems, a range of monitoring and management mechanisms can be implemented by using common and well stablished software development designs, such as a finite state machine.

A simplified software implementation of the proposed architecture was developed for testing, incorporating smaller subset of the software components. The test software was utilized for the demonstration presented as part of this development work.

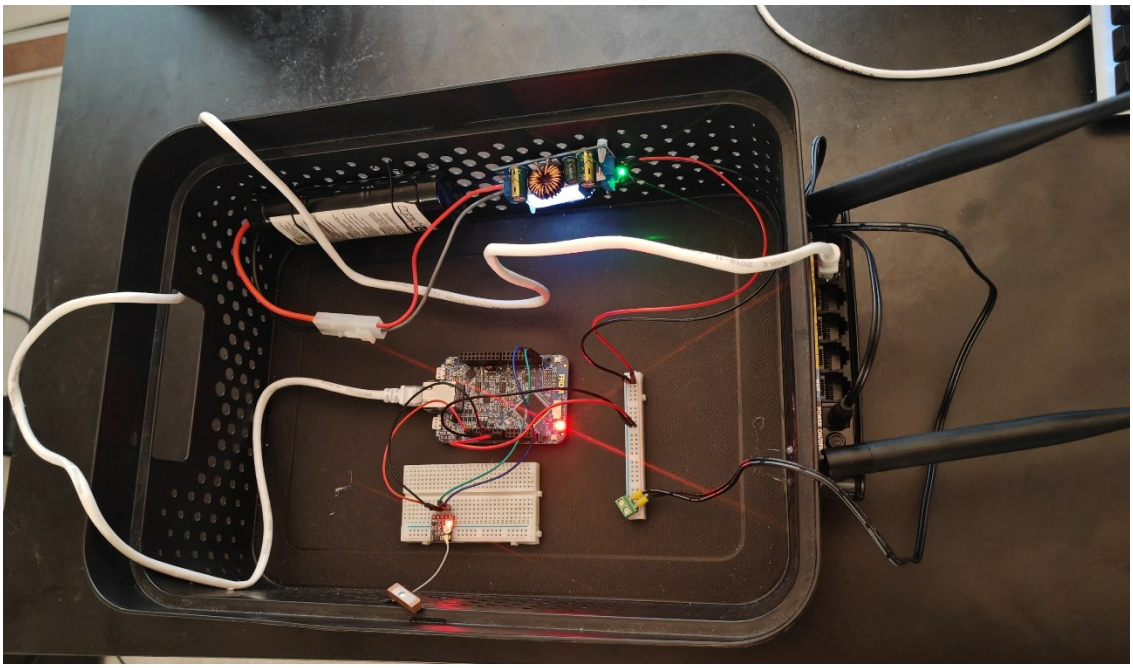
4 Demonstration

To validate the practical applicability of the proposed software architecture, a demonstration was planned. The objective was to transmit sensor data to the Mosquitto public broker over a Wi-Fi network. The integrity and usability of the transmitted data could then be verified using a client software capable of retrieving the data from the broker.

The demonstration required only a minimal hardware setup. A 7.2V battery pack was used to power all the hardware components used in this work. To meet the voltage requirements of the components, a DC-to-DC buck converter was used to step the voltage down to 5V, the operational voltage level required by the Zyxel router, the FDRM-K64F development board, and the TEL0132 module.

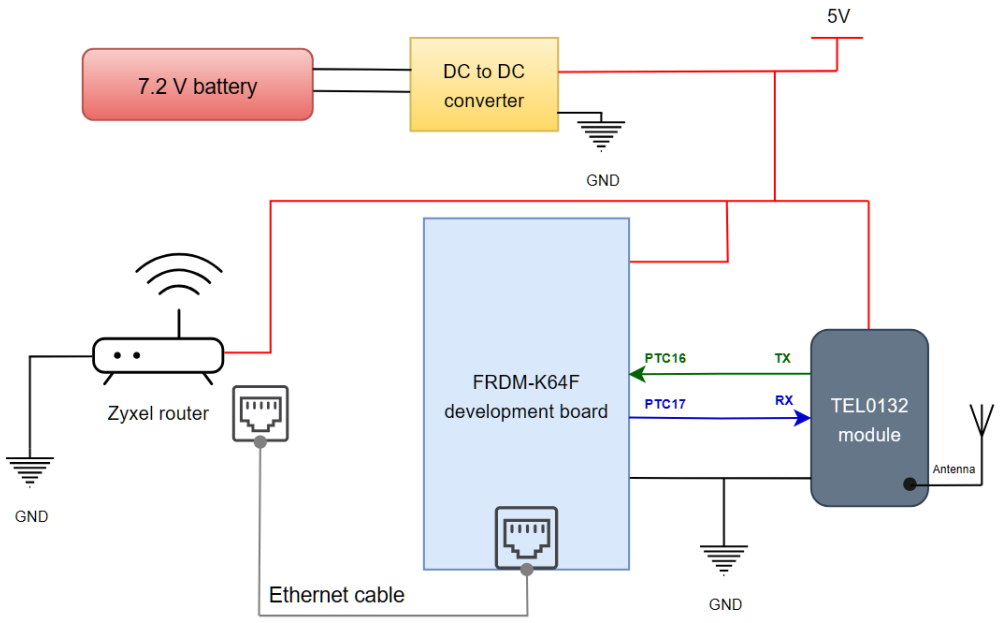
The FDRM-F64F development board was connected to the Zyxel router, already configured to operate in Client Bridge mode, via an Ethernet cable. The test hardware was installed on a plastic box, shown in figure 20, for better maintenance and handling of the hardware.

Figure 20. A picture illustrating the hardware configuration used for the demonstration.



The TEL0132 module was connected to the UART3 interface of the microcontroller through the pins PTC17, PTC16. Figure 21 outlines the corresponding pin connections between the TEL0132 module and the FDRM-K64F development board as well as the circuitry between all devices used for this development work.

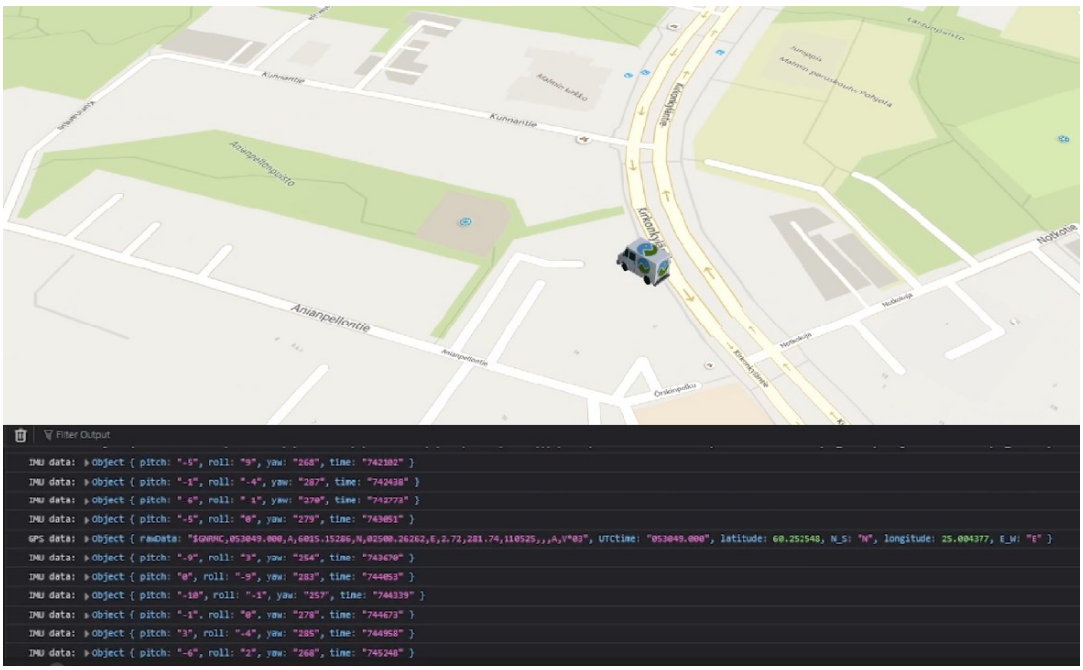
Figure 21. Circuitry used to integrate all hardware components involved for this development work



A web application was developed using fundamental web technologies to subscribe to designated MQTT topics and receive live data streams from the hardware used in this development work. This web application functioned as the client and the receiving endpoint, importing data from the public MQTT broker and, by extension, from the physical devices. CesiumJS, an open-source JavaScript library for creating 3D globes and maps in a web browser was used to facilitate this process.

The incoming data streams to the web application were then processed and visualized within CesiumJS by mapping sensor data onto a two-dimensional map using a three-dimensional vehicle model. A screenshot of this process, shown in figure 22, was taken during software development phase of this work.

Figure 22. A picture illustrating the web application developed for this work. The application visualizes sensor data on a map using a 3-Dimensional model



The core logic responsible for building the web application was implemented using standard web development technologies and publicly available libraries. As the goal of this development work is to illustrate how sensor data can be collected, transmitted, and represented for a virtual environment, the detailed implementation of the web application is not included. Instead, the focus remains on presenting the data flow from physical sensors, through the broker, and into a digital representation highlighting the feasibility of the approach taken.

5 Conclusion

To enable the import of sensor values and real-world situational information into a virtual or digital environment, it is essential that the sensor data be analysed, structured, and formatted according to standardized protocols. Achieving this integration requires the combined use of appropriate software and hardware components. While software solutions can be developed to address a wide range of tasks and objectives, their functionality is inherently dependent on the capabilities of the selected hardware. Therefore, careful analysis and evaluation of the hardware is a necessary step in the development of effective and compatible software systems.

The outcome of this development work was the design of a software architecture with a demonstration to test the feasibility of such software design. Although the implementation of this design was successfully demonstrated, it should not be regarded as the sole approach for achieving sensor data portability for virtual or digitized environments. Different environments may impose varying software requirements. In applications where real-time performance is essential, the software architecture must be designed to meet strict timing constraints and ensure that data is delivered to the end user within defined deadlines. Nonetheless, the practical approach of this work serves as a valuable reference for future applications and further research in similar contexts.

References

Carver, J. (n.d.). *This is an example of a tilt compensated eCompass*. Retrieved May 14, 2025, from https://os.mbed.com/users/JimCarver/code/K64F_eCompass/

DFRobot. (n.d.). *GPS + BDS BeiDou Dual Module Flight Control Wiki*. Retrieved May 12, 2025, from https://wiki.dfrobot.com/GPS_%2B_BDS_BeiDou_Dual_Module_SKU_TEL0132#target_5

D'mello, B. J., & Sriparasa, S. (2018). *JavaScript and JSON Essentials*. Packt Publishing.

Eclipse Foundation. (n.d.). *Eclipse Mosquitto*. Retrieved May 18, 2025, from https://mosquitto.org/

HiveMQ. (n.d.). *MQTT Topics, Wildcards, & Best Practices – MQTT Essentials: Part 5*. Retrieved May 12, 2025, from https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/

Mbed. (n.d.-a). *Architecture*. Retrieved May 18, 2025, from https://os.mbed.com/docs/mbed-os/v6.16/introduction/architecture.html

Mbed. (n.d.-b). *EthInterface Class Reference abstract*. Retrieved May 17, 2025, from https://os.mbed.com/docs/mbed-os/v6.16/mbed-os-api-doxy/class_eth_interface.html

Mbed. (n.d.-c). *Execution*. Retrieved May 21, 2025, from https://os.mbed.com/docs/mbed-os/v6.16/program-setup/concepts.html

Mbed. (n.d.-d). *FRDM-K64F*. Retrieved May 12, 2025, from https://os.mbed.com/platforms/FRDM-K64F/

Mbed. (n.d.-e). *Mbed OS*. Retrieved May 21, 2025, from https://os.mbed.com/mbed-os/

Mbed. (n.d.-f). *Mbed Studio*. Retrieved May 21, 2025, from https://os.mbed.com/studio/

Mbed. (n.d.-g). *mbed-mqtt*. Retrieved May 21, 2025, from https://github.com/ARMmbed/mbed-mqtt

Mbed. (n.d.-h). *Mutex*. Retrieved May 21, 2025, from https://os.mbed.com/docs/mbed-os/v6.16/apis/mutex.html

Mbed. (n.d.-i). *Network socket*. Retrieved May 17, 2025, from https://os.mbed.com/docs/mbed-os/v6.16/apis/network-socket.html

NXP Semiconductors N.V. (2017). *FXOS8700CQ, 6-axis sensor with integrated linear accelerometer and magnetometer*. https://www.nxp.com/docs/en/data-sheet/FXOS8700CQ.pdf

OASIS Open. (2019, March 7). *MQTT Version 5.0*. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

Raymond, E. S. (n.d.). *NMEA Revealed*. Retrieved May 12, 2025, from https://gpsd.gitlab.io/gpsd/NMEA.html

Trimble. (2004). *NMEA-0183 Messages Guide for AgGPS Receivers*. https://www2.etown.edu/wunderbotvi/Downloads/PDF%27s/GPS/NMEA_Messages_RevA_Guide_ENG.pdf

TronicsBench. (n.d.). *Magnetometer tilt compensation*. Retrieved May 12, 2025, from https://www.best-microcontroller-projects.com/magnetometer-tilt-compensation.html

University of Glasgow. (n.d.). *The Berkeley Sockets API Networked Systems Architecture 3 Lecture 4*. https://csperkins.org/teaching/2007-2008/networked-systems/lecture04.pdf