# Schafalarm



**Write your documentation here!**

## First Draft Ideas

- RF regulations! ETSI (EU), FCC (USA), be compliant!
- Allowed frequencies (ISM), 433 MHz, 868 MHz, 2.4 GHz, (e.g. LoRa: 868 Mhz (EU), 915 MHz (USA, AU))
- BLE(?): No roaming, data size in beacon mode very limited
- ~~Wifi~~: Limited range, too much power
- Research: Zigbee (IoT multihop full meshed network with concentrators, routers, end devices)??? Why isn't this feasible?
- 433 MHz, FSK (QFSK enconding):
- IMU
- UWB Indoor Navigation for outdoor
- RTK GPS, low power?
- Animal cameras with AI based pose estimation

## Links

- Seeed XIAO BLE nRF52840 Sense Microcontroller with IMU
- https://wiki.seeedstudio.com/XIAO_BLE/
- https://www.seeedstudio.com/Seeed-XIAO-BLE-Sense-nRF52840-p-5253.html

- Wolf Early Warning: https://wiki.eolab.de/doku.php?id=project_ideas:wolf_early_warning:start
- RTK GPS: https://www.youtube.com/watch?v=Oc1LBFDj2MA

# 1. Introduction

Currently, in North Rhine-Westphalia and Rhineland-Palatinate there has been an increase in the number of attacks from wolves on flocks of sheep. This rise in attacks has led to political discussions about how to address the issue and what actions should be taken regarding the wolves. This project focuses on protecting sheep by detecting when they are being attacked by wolves. The goal is to identify signs of an attack by tracking the movements and behavior of the sheep. By monitoring their motion it's possible to send out an alert when the sheep exhibit signs of distress or unusual activity that may indicate a threat. The challenge lies in effectively tracking the motion of a flock of sheep. Using advanced motion sensors and communication technology we can continuously monitor the activity levels of the sheep. These sensors can detect abrupt movements or patterns that differ from normal behavior, signaling a potential attack. By alerting farmers or authorities, appropriate measures can be taken to protect the sheep and mitigate the damage caused by wolf attacks.

# 2. Materials and Methods

## 2.1 Materials

To develop the sheep movement detection system, various hardware and software tools were used. The materials employed in the project are the following:

- **ESP32-S3 dev kit microcontroller**, a powerfull and enregy efficient microcontroller with advanced features like the deep sleep mode to minimize power consumption and a high computational power.
- **Xiao BLE sense nRF52840**, a very small module with a 6-axis IMU onboard and Bluetooth low energy connectivity. It has a great usability since it can be connected to a phone application from Nordic Semiconductors (nRF connect) and it has a relatively high data transmission rate.
- **GY-521 Accelerometer and Gyroscope module (6-axis)**, this module uses the MPU6050 sensor, which captures the motion and orientation of the sheep. It measures the acceleration of the sheep, providing crucial data.
- **RFM69HCW radio module**, this communication module enables the transmission of data between the sensor and a receiving module. It operates at a frequency of 433/434 MHz, and it was chosen due to its long-range communication.
- **Powering components**, a battery is necessary in order to supply power to the board and modules. A lithium-ion battery was chosen due to its rechargeability and high energy density.
- **Connecting wires (jumper cables)**, were used to connect all of the pins.
- **Software libraries**, such as Adafruit_MPU6050 and RH_RF69 for interfacing with the MPU6050 sensor and RFM69 radio module.

## 2.2 Methods

Two prototypes were built, one was ruled out so another one had to be developed.

### First Prototype (Xiao BLE Sense)

#### Hardware setup

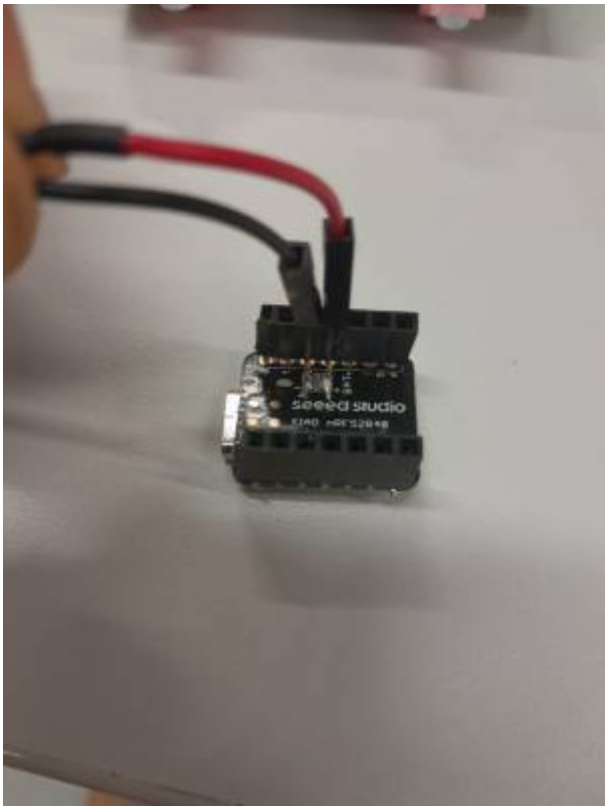



Fig. 1: Xiao BLE Sense connected to a LIB For this prototype, since the microcontroller has the IMU onboard and BLE, only a supplementary LIB was used to power the module.

**Software setup**

The Xiao BLE Sense was programmed to read accelerometer and gyroscope data from its built-in IMU. The data was then formatted and sent via BLE. The software also included features to handle data transmission intervals.
Code explanation:

1º The relevant libraries were includes and the IMU (LSM6DS3, uses i2c) and BLE were initialized.

```
#include <ArduinoBLE.h>
#include "LSM6DS3.h"
#include "Wire.h"

// Create an instance of class LSM6DS3
LSM6DS3 myIMU(I2C_MODE, 0x6A); // I2C device address 0x6A

// BLE Service
BLEService imuService("5d2af01a-439d-45ca-9dc7-edc49cf2a539"); // Custom
UUID for the service

// BLE Characteristic with custom UUID
BLECharacteristic imuChar("ea1404e2-5466-4a03-a921-3786d1642e5d", BLERead |
BLENotify, 400); // Custom UUID for IMU data
```

2º A function which converts float to string was created (floatToString) so that the IMU values can be read as a string in the nRF app.

```
// Custom function to convert float to string
void floatToString(char* buffer, float value, int places) {
    // Handle negative numbers
    if (value < 0) {
        *buffer++ = '-';
        value = -value;
    }

    // Round value to the specified number of decimal places
    float rounding = 0.5;
    for (int i = 0; i < places; ++i) {
        rounding /= 10.0;
    }
    value += rounding;

    // Extract integer part
    unsigned long intPart = (unsigned long)value;
    float remainder = value - (float)intPart;

    // Convert integer part to string
    itoa(intPart, buffer, 10);
    while (*buffer != '\0') {
        buffer++;
    }

    // Add decimal point
    *buffer++ = '.';

    // Extract fractional part
    while (places-- > 0) {
        remainder *= 10.0;
        int toPrint = int(remainder);
        *buffer++ = toPrint + '0';
        remainder -= toPrint;
    }

    // Null-terminate the string
    *buffer = '\0';
}
```

3º Setup function, here the serial communication is initialized and also the IMU. Regarding the BLE, the device settings are customized and it begins advertising to find other devices nearby.

```
void setup() {
    // Initialize serial communication
    Serial.begin(115200);

    // Initialize the IMU
    if (myIMU.begin() != 0) {
        Serial.println("Device error");
```

```
    } else {
        Serial.println("Device OK!");
    }

    // Initialize BLE
    if (!BLE.begin()) {
        Serial.println("starting BLE failed!");
        while (1);
    }

    // Set device name and local name
    BLE.setDeviceName("XIAO BLE Sense");
    BLE.setLocalName("XIAO BLE Sense");
    BLE.setAdvertisedService(imuService);

    // Add characteristic to the service
    imuService.addCharacteristic(imuChar);

    // Add service
    BLE.addService(imuService);

    // Start advertising
    BLE.advertise();

    Serial.println("Bluetooth device active, waiting for connections...");
}
```

4º Loop Function, the BLE listens for connections from central devices (this is configured as a periphreal). When a connexion is established, the IMU sensor readings for the accelerometer and gyroscope are periodically read, converted to strings, combined into a single message and then sent to the central device. This loop continues to read and send data every 0.1 seconds (10Hz) for as long as the central device is connected to it. When it gets disconnected, it starts again to listen for new connections. Lastly all the action like connexions, disconnexions, the data strings sent and everything gets printed on the serial monitor, just for test and debugging purposes.

```
void loop() {
    // Listen for BLE connections
    BLEDevice central = BLE.central();

    // If a central is connected to the peripheral:
    if (central) {
        Serial.print("Connected to central: ");
        Serial.println(central.address());

        // Check the IMU and send data periodically
        while (central.connected()) {
            // Read accelerometer and gyroscope values
            float accelX = myIMU.readFloatAccelX();
            float accelY = myIMU.readFloatAccelY();
            float accelZ = myIMU.readFloatAccelZ();
            float gyroX = myIMU.readFloatGyroX();
```

```
            float gyroY = myIMU.readFloatGyroY();
            float gyroZ = myIMU.readFloatGyroZ();

            // Create a message string for the package
            char message[400] = {0};
            char valueStr[20];

            // Add readings to the message
            floatToString(valueStr, accelX, 4);
            snprintf(message + strlen(message), sizeof(message) -
strlen(message), "%s, ", valueStr);
            floatToString(valueStr, accelY, 4);
            snprintf(message + strlen(message), sizeof(message) -
strlen(message), "%s, ", valueStr);
            floatToString(valueStr, accelZ, 4);
            snprintf(message + strlen(message), sizeof(message) -
strlen(message), "%s, ", valueStr);
            floatToString(valueStr, gyroX, 4);
            snprintf(message + strlen(message), sizeof(message) -
strlen(message), "%s, ", valueStr);
            floatToString(valueStr, gyroY, 4);
            snprintf(message + strlen(message), sizeof(message) -
strlen(message), "%s, ", valueStr);
            floatToString(valueStr, gyroZ, 4);
            snprintf(message + strlen(message), sizeof(message) -
strlen(message), "%s ", valueStr);


            // Update BLE characteristic with message string
            imuChar.writeValue(message);

            // Print the message to the Serial Monitor
            Serial.println(message);

            // Delay for 0.1 second to send data at 10 Hz frequency
            delay(100);
        }

        // When the central disconnects:
        Serial.print("Disconnected from central: ");
        Serial.println(central.address());
    }
}
```

## Second Prototype (ESP32-S3 with RFm69hcw and gy-521)

### Hardware setup

The rfm69hcw module needs a breakout for better mounting on a breadboard. Instead of using the

SparkFun Breakout, a PCB Adapter ESP 07 / 12 – Board is used. The Youtube guide by Mobilefish.com was followed. After that, an antenna, cut to the length of 164mm was soldered to the antenna pin. The pin connections can be observed on Figure 1.

MICROCONTROLLER TO MPU6050: the ESP32-S3 microcontroller is connected to the GY-521 module using I2C communication. The connexions are:
VCC –> 3.3V
GND –> GND
GPIO9 (SCL) –> GY-521 SCL
GPIO8 (SDA) –> GY-521 SDA
GPIO1 (RTC GPIO) –> INT

MICROCONTROLLER TO RFM69HCW: the RFM69HCW module is connected to the ESP32-S3 using SPI communication. The connections are:

3.3V –> 3.3V
GND –> GND
MISO –> GPIO13
MOSI –> GPIO11
SCK –> GPIO12
NSS –> GPIO10
RST –> GPIO38
DIO0 –> GPIO4

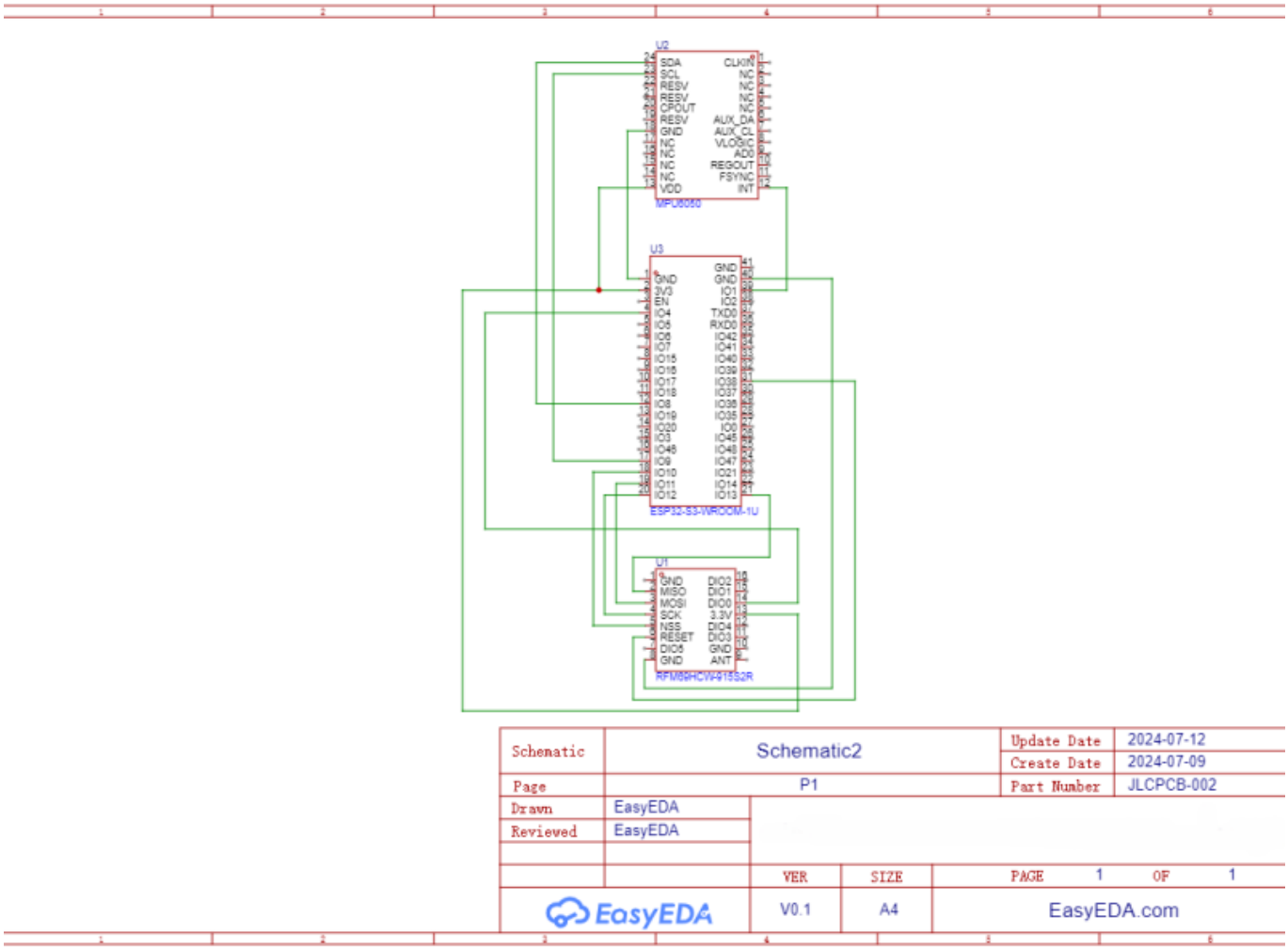



Fig. 2: Circuit Prototype 2 schematic

This is a circuit schematic produced with EasyEDA which illustrates the connexions previously described.

### Software setup

The software integrates the hardware components to collect, process, and transmit motion data. The code includes libraries for the MPU6050 sensor and RFM69 radio module, as well as functions to read sensor data, detect significant movements, and manage power consumption. It covers the sensor initialization, data collection, wireless communication and power management implementation.
The code is the following:

**Important**
To use esp32 with RFM69HCW, first download the RadioHead library, and the RH_ASK.cpp library has to be modified. Otherwise the code wouldn't work.

1º Including libraries and initializing the MPU6050, the RF69 module, and the esp32 deep sleep (setting up the constants for sleep and wake time). Also the state machine was defined for detecting motion and sleep modes, as well as the GPIO pins.

```
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
#include <SPI.h>
#include <RH_RF69.h>
#include "esp_sleep.h"

// MPU6050 Setup
Adafruit_MPU6050 mpu;

// RFM69 Radio Setup
#define RF69_FREQ 434.0

// ESP32 S3 pins
#define RFM69_CS   10
#define RFM69_INT  4
#define RFM69_RST  38

RH_RF69 rf69(RFM69_CS, RFM69_INT);

#define uS_TO_S_FACTOR 1000000ULL /* Conversion factor for micro seconds to
seconds */
#define TIME_TO_SLEEP  30          /* Time ESP32 will go to sleep (in
seconds) */
#define WAKE_TIME      5          /* Time ESP32 will stay awake (in seconds)
*/

// GPIO pin connected to the MPU6050 interrupt pin
#define MPU_INT_PIN 1

RTC_DATA_ATTR int bootCount = 0;
```

```
// State machine states
enum State {
  MOVE,
  STOP,
  TIMER
};

State currentState = STOP;
unsigned long lastMotionTime = 0;
const unsigned long motionTimeout = 3000;  // 3 seconds
```

2º MPU6050 Setup, here the sensor is initialized and the features are configured.

```
void setupMPU6050() {
  // Initialize MPU6050
  if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1) {
      delay(10);
    }
  }
  Serial.println("MPU6050 Found!");

  // Setup motion detection
  mpu.setHighPassFilter(MPU6050_HIGHPASS_0_63_HZ);
  mpu.setMotionDetectionThreshold(1);
  mpu.setMotionDetectionDuration(20);
  mpu.setInterruptPinLatch(true);  // Keep it latched. Will turn off when
reinitialized.
  mpu.setInterruptPinPolarity(true);  // Set to low for interrupt
  mpu.setMotionInterrupt(true);
}
```

3º Prints the wake up reason for the esp32 from the deep sleep, there are different causes like timers, movement

```
void print_wakeup_reason() {
  esp_sleep_wakeup_cause_t wakeup_reason = esp_sleep_get_wakeup_cause();
  switch (wakeup_reason) {
    case ESP_SLEEP_WAKEUP_EXT0:
      Serial.println("Wakeup caused by movement");
      break;
    case ESP_SLEEP_WAKEUP_EXT1:
      Serial.println("Wakeup caused by external signal using RTC_CNTL");
      break;
    case ESP_SLEEP_WAKEUP_TIMER:
      Serial.println("Wakeup caused by timer");
      break;
    default:
      Serial.printf("Wakeup was not caused by deep sleep: %d\n",
wakeup_reason);
```

```
      break;
  }
}
```

4º “Read and transmit IMU data”, this function reads the IMU data and creates a message with the accelerometer and gyroscope data and transmits it using the RF69 module. This data is also printed on the serial monitor.

```
void readAndTransmitIMUData(unsigned long duration) {
  unsigned long startMillis = millis();
  unsigned long currentMillis = startMillis;

  while (currentMillis - startMillis <= duration) {  // Run for the
specified duration
    // Get new sensor events with the readings
    sensors_event_t a, g, temp;
    mpu.getEvent(&a, &g, &temp);

    // Create a message with the IMU data
    char radiopacket[100];
    snprintf(radiopacket, sizeof(radiopacket), "Accel: %.2f, %.2f, %.2f
Gyro: %.2f, %.2f, %.2f",
             a.acceleration.x, a.acceleration.y, a.acceleration.z,
             g.gyro.x, g.gyro.y, g.gyro.z);

    // Send the message
    rf69.send((uint8_t *)radiopacket, strlen(radiopacket));
    rf69.waitPacketSent();

    // Print out the values
    Serial.println(radiopacket);

    currentMillis = millis();
    delay(100);  // Read every 0.1 second
  }
}
```

5º The setup function initializes the serial communication, prints the wake up reaon and sets up the MPU6050 and RF69 module (by configuring its frequency, transmission power and encryption key). The wake up reason is also used to set the current state of the device (of the state machine) and the esp32 is confihured to wake up on a timer or on MPU6050 motion detection.

```
void setup() {
  Serial.begin(115200);
  delay(1000);  // Take some time to open up the Serial Monitor

  // Increment boot number and print it every reboot
  //++bootCount;
  //Serial.println("Boot number: " + String(bootCount));
```

```
  // Print the wakeup reason for ESP32
  print_wakeup_reason();

  // MPU6050 Initialization
  setupMPU6050();

  // RFM69 Initialization
  pinMode(RFM69_RST, OUTPUT);
  digitalWrite(RFM69_RST, LOW);

  // Manual reset
  digitalWrite(RFM69_RST, HIGH);
  delay(10);
  digitalWrite(RFM69_RST, LOW);
  delay(10);

  if (!rf69.init()) {
    Serial.println("RFM69 radio init failed");
    while (1);
  }
  Serial.println("RFM69 radio init OK!");

  if (!rf69.setFrequency(RF69_FREQ)) {
    Serial.println("setFrequency failed");
  }

  rf69.setTxPower(20, true);  // range from 14-20 for power, 2nd arg must be
true for 69HCW

  uint8_t key[] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};
  rf69.setEncryptionKey(key);

  Serial.print("RFM69 radio @");  Serial.print((int)RF69_FREQ);
Serial.println(" MHz");

  // Handle wakeup
  esp_sleep_wakeup_cause_t wakeup_reason = esp_sleep_get_wakeup_cause();
  if (wakeup_reason == ESP_SLEEP_WAKEUP_EXT0) {
    currentState = MOVE;
    lastMotionTime = millis();
  } else if (wakeup_reason == ESP_SLEEP_WAKEUP_TIMER) {
    currentState = TIMER;
  } else {
    currentState = STOP;
  }

  // Setup ESP32 to wake up on timer and MPU6050 motion interrupt
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_1, 0); // 0 = Low level to wake up
  Serial.println("Setup ESP32 to wake up on timer and MPU6050 motion
```

```
interrupt");
}
```

6º The loop function makes the state machine handle different states:
MOVE STATE:
Occurs when the system wakes up due to a motion interrupt from the MPU6050 sensor. In the void loop, read the IMU data, transfer to a radio packet and send at 10Hz. Check if motion is still detected using mpu.getMotionInterruptStatus(). This function returns “motion_interrupt” if the interrupt is activated, meaning that motion is detected and lastMotionTime will update to the current time. The system remains in the MOVE state as long as new motion is detected within 3 seconds. Otherwise, transit to the STOP state.

TIMER STATE:
Occurs when the system wakes up due to a built-in timer interrupt. The predefined amount of time to sleep and wake up are 30 seconds and 5 seconds respectively. In this state, call out the function readAndTransmitIMUData. IMU sensor reading, radio packet sending will run in a defined duration (in this case 5s). After that transit to STOP state.

STOP STATE:
Esp32 goes into a deep sleep.

This specific setup makes the esp32 wake up on motion detection or on a timer, then it transmits the IMU data and goes back to sleep to save power.

```
void loop() {
  unsigned long currentMillis = millis();

  switch (currentState) {
    case MOVE:
      // Get new sensor events with the readings
      sensors_event_t a, g, temp;
      mpu.getEvent(&a, &g, &temp);

      // Create a message with the IMU data
      char radiopacket[60];
      snprintf(radiopacket, sizeof(radiopacket), "Accel: %.2f, %.2f, %.2f
Gyro: %.2f, %.2f, %.2f",
               a.acceleration.x, a.acceleration.y, a.acceleration.z,
               g.gyro.x, g.gyro.y, g.gyro.z);

      // Send the message
      rf69.send((uint8_t *)radiopacket, strlen(radiopacket));
      rf69.waitPacketSent();

      // Print out the values
      Serial.println(radiopacket);

      // Check if motion is still detected using MPU6050 interrupt status
      if (mpu.getMotionInterruptStatus()) {
        // Motion is detected, update last motion time
        lastMotionTime = currentMillis;
```

```
      } else {
        // No motion detected, check timeout
        if (currentMillis - lastMotionTime > motionTimeout) {
          currentState = STOP;
        }
      }

      delay(100);  // Read every 0.1 second
      break;

    case TIMER:
      Serial.println("Handling timer wakeup");
      readAndTransmitIMUData(WAKE_TIME * 1000);  // Handle timer wakeup for
specified wake time
      currentState = STOP;
      break;

    case STOP:
      // Go to sleep now
      Serial.println("No motion detected, going to sleep now");
      Serial.flush();
      esp_deep_sleep_start();
      break;
  }
}
```

# 3. Results

This section presents the findings from testing two prototypes for monitoring sheep movement. Each prototype was tested for range in outdoors and indoors conditions.

### First Prototype (Xiao BLE Sense)

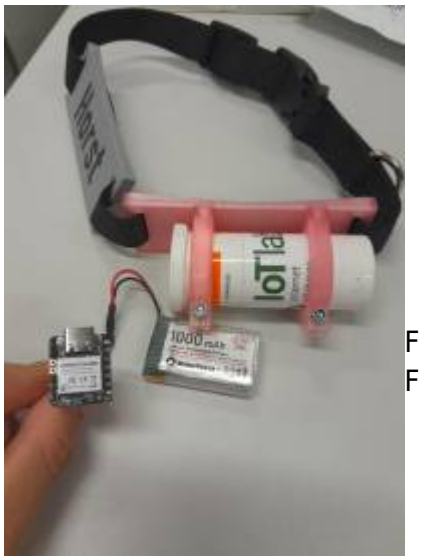


Fig. 3: Collar containing a Xiao BLE Sense module



Fig. 4: Testing the prototype with an alpaca

In this prototype the microcontroller was flashed with the code and connected to a 1000mAh lithium-ion battery and inserted on a casing attached to a collar. Some measurements and tests were taken with some alpacas that were available.

The module receiving the data sent by the prototype was another Xiao BLE Sense flashed with some receiver code, since it was not possible to receive it on the desktop application of Nordic Semiconductors (However, on the mobile app enabling and data download was possible, allowing to read the accelerometer values from the alpacas with a smartphone).

**RANGE TESTS**



Fig. 5: Indoor range test



Fig. 6: Outdoor range test

The maximum range acheived both indoors and outdoors in normal conditions was of 42m, +-2m, on the onther side, when tested with the nRF app on the smartphone values of 60m, +-2m, both indoors and outdoors were obtained as well.

However, it is important to highlight that this values were obtained without any obstacle in between and at average height of 1.4m from the ground. **OUTOORS** it was observed that if you the board was placed in the ground or close to it, the conexion would be lost, and recoverd as soon as you raised it. Also in the test spot there was a tiny 2m hill and the modules would loose their BLE conexion if they were placed in between the hill. **INDOORS** The range was tested also in a straight line and in a concrete corridor. When testing the nrf to module range modules would disconnect after taking a turn in a corner at 40m from the transmitter even though it has a maximum range in a straight line of 60m.

The Bluefruit library increases the BLE range by adjusting the transmit power and modifying the connection parameters to maximize the signal strength and stability over longer distances, in this case between two Xiao BLE Sense modules. However, each of these adjustments has trade-offs in terms of power consumption and latency.

## Second Prototype (ESP32-S3 with RFM69HCW and GY-521)

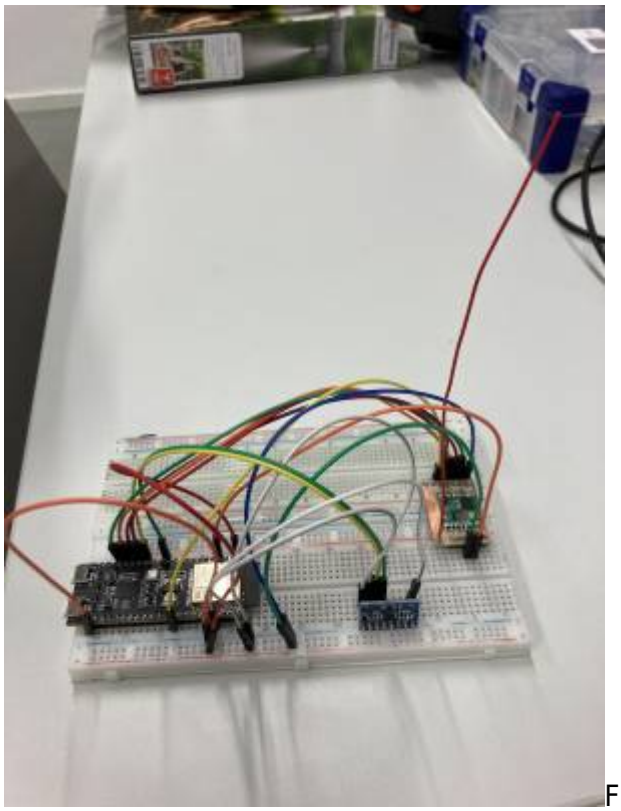


Fig. 7: Second prototype IMU data Transmitter

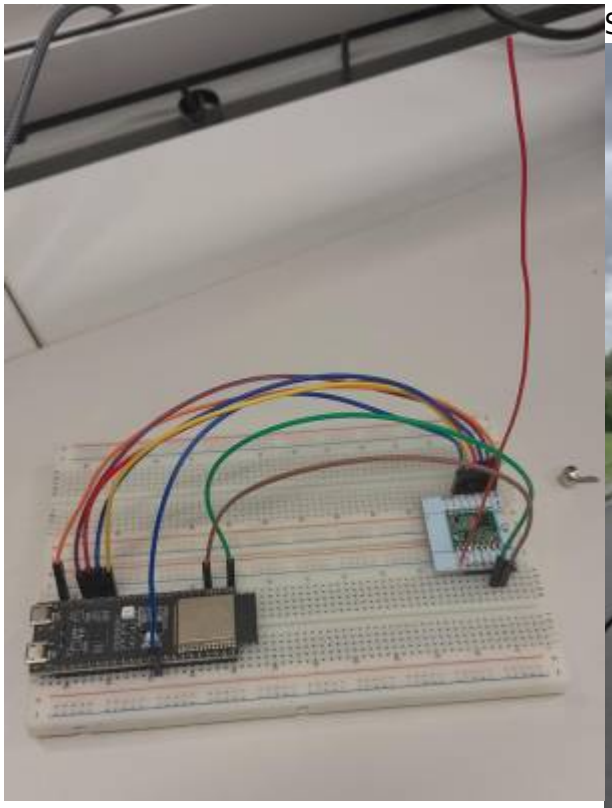


Fig. 8: Second prototype data Receiver



Fig. 9: Field test with second prototype

One module consisted of an ESP32-S3 board connected to a GY-521 6-axis accelerometer, transmitting the values at a frequency of 433/434 MHz using an RFM69HCW transceiver module. To receive this data, another ESP32-S3 board was connected to a second RFM69HCW breakout. This receiver board was set up to capture the radio signal transmitted by the first module, allowing to

monitor and analyze the accelerometer data in real time for any significant movements or irregularities. This setup provided a wireless link between the accelerometer and the receiver module, ensuring data was transmitted effectively over a considerable distance.

### RANGE TESTS

The maximum range we managed to achieve was 140 meters, but the connection was not very stable. We did not receive the values every 0.1 seconds as specified in the code; instead, the data came in roughly every 1 to 2 seconds. At a distance of 100 meters, the connection was moderate, with values being received every 0.3 to 0.5 seconds. Finally, a strong and stable connection was achieved at around 58 meters. The tests were performed in the Zechenpark, Kamp-Lintfort, since it provides a cleared and outdoor space where tests could be performed without bothering anyone.




Fig. 10: Max range acheived



Fig. 11: Moderate connexion acheived

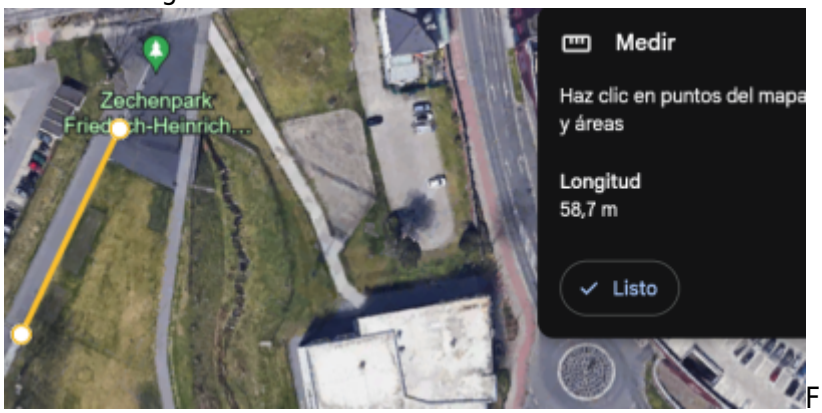



Fig. 12: Stable connexion acheived

### POWER CONSUMPTION TESTS

In order to optimize the power efficiency of our ESP32-based project, a series of power consumption tests under different operational scenarios were conducted. Specifically, we measured the power usage when the ESP32 was in deep sleep mode and when it wasnt, during periods of motion detection, and when no motion was detected. The results for the average current consumptions over a period of 10s on the different cases are displayed in the following table:

| | **Motion is detected** | **No motion is detected** |
|---|---|---|
| **ESP32 on deep sleep** | 48.52 mA | 7.47 mA |
| **ESP32 without deep sleep** | 64.38 mA | 61.11 mA |

All current consumption were measured on a time interval of 10s, it can be observed that with the esp32 on deep sleep and when no motion is being detected the power consumption is severely

reduced. This would help to reduce the power consumption therefore extending the life of the battery.

**with deep sleep**



Fig. 13: When motion is detected

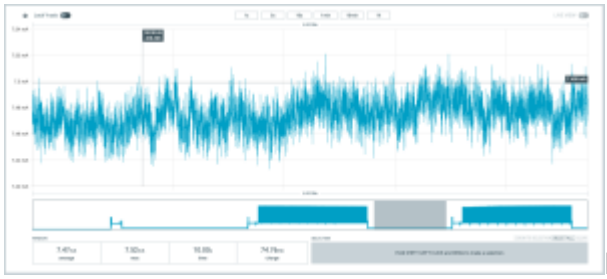


Fig. 14: No motion detected (deep sleep)

**without deep sleep**



Fig. 15: When motion is detected

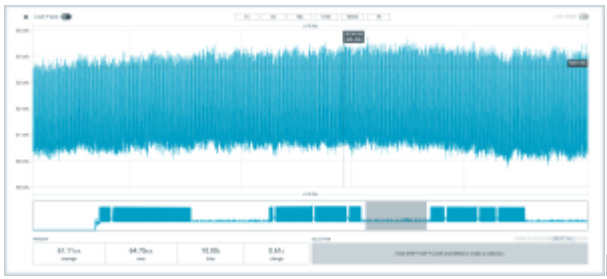


Fig. 16: No motion detected

# 4. Discussion

The initial use of the Xiao BLE Sense showed that while it was capable of detecting motion and transmitting data via BLE, the range limitations and lack of roaming capability made it unsuitable for our needs. The range limitations might be able to be overcomed with the Bluefruit library, but it was not possible to properly configure it and test it, so it remains unknown to which extent would have the range increased.

The switch to the ESP32-S3 microcontroller with the RFM69HCW radio module and the GY-521 accelerometer overcame these issues. The RF69 module provided a much larger range, and the ESP32-S3's deep sleep mode helped in reducing power consumption, making the system more viable for real-world use.

The project aimed to detect the movement of sheep and alert when they are being attacked by wolves. Despite the progress made with both prototypes, the project is not yet complete. One thing left to do is to test the system in the field with actual sheep, so real-world performance and reliability can be assesed. Additionally, to fully track the position of sheep in the field, an Ultra-Wideband (UWB) system would need to be introduced. This would allow precise location tracking (through triangulation) alongside the movement detection provided by the accelerometers.

Another challenge that remains is finding a way to receive accelerometer signals from a large flock of sheep continuously. This requires a robust and scalable communication system that can handle multiple data streams without interference or significant data loss. The current RF69 prototype showed promise in terms of range, but its ability to manage multiple connections simultaneously needs further investigation and testing.

Power consumption is also a critical factor for this project. Since the sensors and communication modules will be attached to sheep, they need to operate efficiently on battery power. One of the solutions to further reduce the power consumption would be cutting of the power supply to the RF69 module when the esp32 in on deep sleep so more power can be saved.

# 5. Conclusion

In conclusion, while the project has made significant progress in developing a system to detect sheep movement and potential wolf attacks, several critical steps remain. Field testing is necessary to check the system's performance in real-world conditions. Incorporating Ultra-Wideband technology would allow to track sheep positions accurately. Additionally, the challenge of continuously receiving accelerometer signals from multiple sheep needs to be addressed to ensure the system's scalability. Finally, optimizing power consumption is crutial for the practical application of the system, ensuring that it can operate efficiently over extended periods. These next steps are crucial for moving the project from prototype to a fully functional solution.

# 6. References

SparkFun Breakout
PCB Adapter ESP 07 / 12 – Board
Mobilefish.com
RadioHead library
Using ESP32 with rfm69hcw
nRF Connect for mobile
UUID Generator
Bluefruit library
using-the-rfm69-radio

RFM69HCW Hookup Guide by **Sparkun**
Antenna Section of RFM69HCW Hookup Guide by **Sparkun**
Github Sparkfun RFM69HCW Breakout

# Seeed XIAO BLE Sense nRF52840

- Schematic:
- IMU: https://files.seeedstudio.com/wiki/XIAO-BLE/ST_LSM6DS3TR_Datasheet.pdf

Result: Excellent technical design, easy to use, high data rate, **but range too short**.

## Prototype

The first prototype is done by connecting the Seeed XIAO BLE Sense nRF52840 (XIAO BLE Sense) to a 1000 mAh Lithium Battery and insert in a Vitamin tube with desiccant cap which will be hold on an animal collar and finally tested with two alpacas.





## Reading Data with mobile app

The following Github link is direct to the arduino code used in this prototype. Run the code in the file 'xiao_imu_10Hz.ino'. Once it connects with the client device, it reads the IMU data and starts BLE advertising.

Before running the code, make changes to the BLE service and BLE characteristic UUID, the UUID has to be unique within the area of your connection. Simply generate random UUID with UUID Generator and replace it in the code.

```
#include <ArduinoBLE.h>
#include "LSM6DS3.h"
#include "Wire.h"

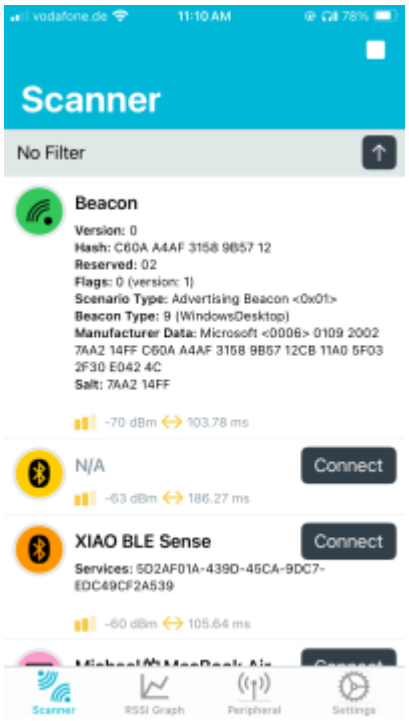
// Create an instance of class LSM6DS3
LSM6DS3 myIMU(I2C_MODE, 0x6A); // I2C device address 0x6A

// BLE Service
BLEService imuService("5d2af01a-439d-45ca-9dc7-edc49cf2a539"); // Custom UUID for the service

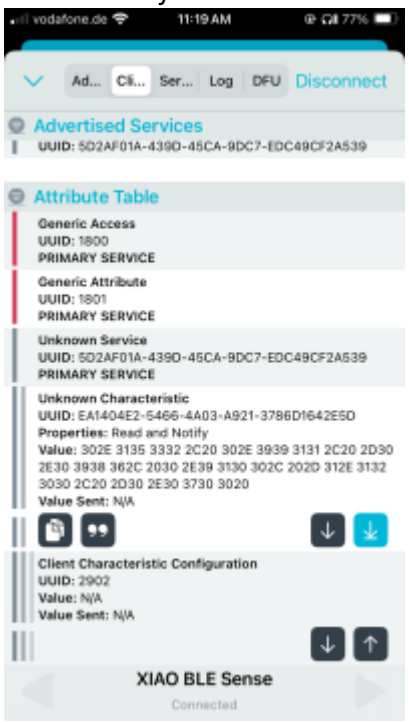
// BLE Characteristic with custom UUID
BLECharacteristic accelXChar("ea1404e2-5466-4a03-a921-3786d1642e5d", BLERead | BLENotify, 400); // Custom UUID for Accelerometer X
```

To read the data easily, use the nRF Connect for mobile, the app allows you to scan BLE devices and read their characteristics. The steps are as follow:

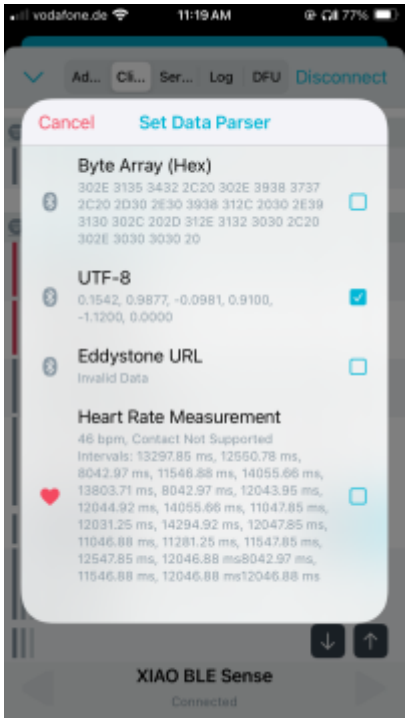
1. Scan the BLE devices and search for the name of the device, in this case it is ‘XIAO BLE Sense’. Then connect.



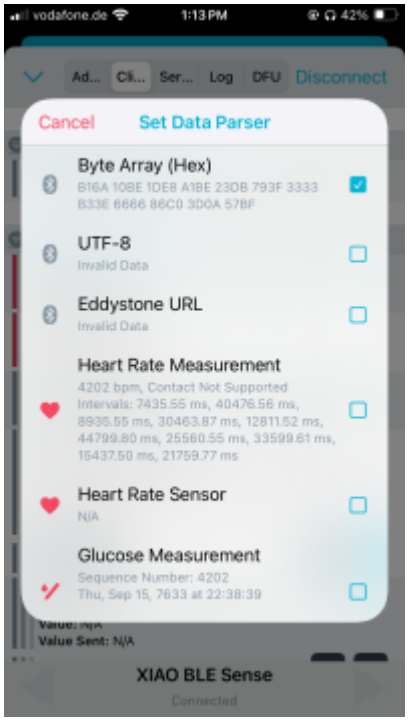

2. Press the downward arrow on the right, it allows you to download the data and update continuously.




3. Use the  button, Convert the Hexadecimal to a readable format.




## *Without changing float to string




If the data type remains as float and sends it, it will not be readable in the app.

## Reading Data with another XIAO BLE Sense

Upload the code from the github file ‘xiao_imu_receive.ino’ and run it on another XIAO BLE Sense. Set the UUIDs the same as the one you want to communicate with.

This step is not necessary but allows you to confirm you are connected to a correct device by verifying the device name.

```
24    void loop() {
25      // Check if a peripheral has been discovered
26      BLEDevice peripheral = BLE.available();
27
28      if (peripheral) {
29        // Check if the discovered peripheral is the right one
30        if (peripheral.localName() == "XIAO BLE Sense") {
31          Serial.print("Connecting to peripheral: ");
32          Serial.println(peripheral.localName());
```

Then the IMU data should print on the serial monitor.

## Reading Data with Real time plot in Python

To visualize the data after running the receiver code, real time plotting in python is an option. The python library PyQtGraph which can handle high update rates is used. The jupyter notebook needs to run locally and running the code needs few packages. Open the terminal and write the following command:

1. create a conda environment

```
conda create -n ble -c conda-forge python=3.9
```

2. activate the conda environment

```
conda activate ble
```

3. install the necessary packages in the environment

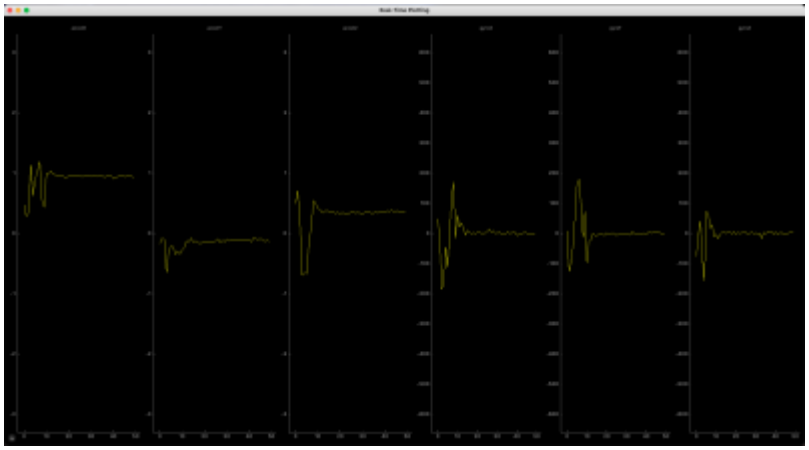
```
conda install -c conda-forge jupyterlab numpy pyqtgraph pyqt pyserial
```

The idea is to open the serial monitor in the jupyter notebook instead of the arduino IDE. Then split the sensor data with commas and store them separately in the empty lists, the lists can store up to 50 data points which you can adjust from the code. The lists is then plotted in an extra window and updated every 0.1 second.
From github, download the jupyter notebook 'imu_realtimeplot_pyqtgraph.ipynb' and open it locally. Change the port name that matches the name in the arduino ide.




```
# Maximum number of data points to store
MAX_DATA_POINTS = 50

# Establish Serial connection
ser = serial.Serial("/dev/cu.usbmodem11101", 115200)  # Adjust the port name and baud rate as needed
time.sleep(2)  # Time delay for Arduino Serial initialization
```



## Range/ RSSI test of BLE

- Indoor
- Outdoor
- In mining site
- Between two Xiao sensor

It is important to highlight that this values were obtained without any obstacle in between and at average height of 1.4m from the ground. **OUTOORS** it was observed that if you placed the board in the ground or close to it, the conexion would be lost for and recoverd as soon as you raised it. Also in our test spot there was a tiny 2m hill and the modules would loose their BLE conexion if they were placed in between the hill. **INDOORS** The range was tested also in a straight line and in a concrete corridor. When testing the nrf to module range modules would disconnect after taking a turn in a corner at 40m from the transmitter even though it has a maximum range in a straight line of 60m.

### Indoor



The indoor range reaches 60 meters, +-2m.

### Outdoor



After testing the conexion between the board and the nrf smartphone app, the range of the BLE was 60 meters, +-2m.

**Xiao to Xiao connection**

This was tested with one XIAO module reading the IMU values and transmitting them via BLE and another module acting as a receiver, after testing the ranges were:

42 meters, +-2m, both indoors and outdoors.

**It was observed that indoors and outdoors, XIAO to XIAO and XIAO to nrf, almost there was no difference between the environments.**

**In mining site**



This was test with XIAO to nrf. The range in the mining tunnel reaches 90 meters. However, in the condition of the curve line(white crosses), it reaches only 30 meters approximately.

## Range improvement with Bluefruit.h

The Bluefruit library increases the BLE range by adjusting the transmit power and modifying the connection parameters to maximize the signal strength and stability over longer distances.

By using the Bluefruit library to adjust transmit power, connection parameters, advertising intervals, and features like Coded PHY (only in Bluetooth 5.0), you can significantly extend the BLE range

between two Xiao BLE Sense modules. While each of these adjustments has trade-offs in terms of power consumption and latency, they help maintain a robust connection over longer distances.

## Power consumption

The supply voltage is set to 3.7V. The first image is the result when XIAO BLE Sense disconnected with the mobile app and was waiting for a connection.
While the second image is the result when XIAO BLE Sense is connected and sent a 6-axis IMU at 10Hz. (10 sets of data in 1 second)




## Reading Data with Linux laptop

The blog post in macchina.io gave a tutorial of how to communicate with BLE devices on Linux. One of the methods is using bluepy, a python interface to BLE on Linux. Follow the step **Install Dependencies→ Get and Build bluepy-helper**. If Git is not installed yet, install Git:

```
$ sudo apt install git
```

However the command line:

```
$ sudo hcitool lescan
```

doesn’t run in the terminal. Alternatively I used:

```
sudo btmgmt find
```

for scanning.

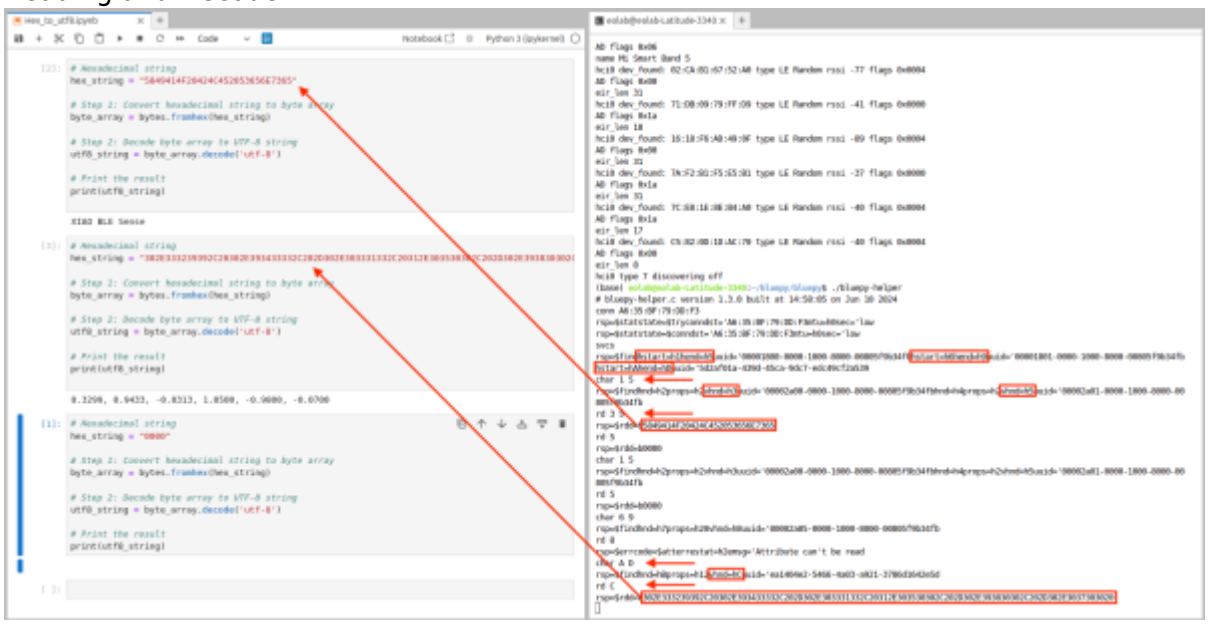
The scanning results are as follows:

```
eolab@eolab-Latitude-3340: ×
(base) eolab@eolab-Latitude-3340:~$ cd bluepy/bluepy
(base) eolab@eolab-Latitude-3340:~/bluepy/bluepy$ sudo btmgmt find
[sudo] password for eolab:
Discovery started
hci0 type 7 discovering on
hci0 dev_found: F8:5D:2A:CA:2E:D9 type LE Random rssi -37 flags 0x0004
AD flags 0x00
eir_len 8
hci0 dev_found: 38:06:23:31:4F:FA type LE Random rssi -78 flags 0x0004
AD flags 0x00
eir_len 31
hci0 dev_found: 67:70:97:46:6F:81 type LE Random rssi -36 flags 0x0000
AD flags 0x1a
eir_len 31
hci0 dev_found: 51:06:B5:A9:10:B1 type LE Random rssi -46 flags 0x0000
AD flags 0x1a
eir_len 18
hci0 dev_found: 57:05:E7:54:82:88 type LE Random rssi -92 flags 0x0004
AD flags 0x00
eir_len 31
hci0 dev_found: 17:DC:07:11:F7:AC type LE Random rssi -69 flags 0x0004
AD flags 0x00
eir_len 31
hci0 dev_found: 39:29:9C:87:CA:AC type LE Random rssi -62 flags 0x0004
AD flags 0x00
eir_len 31
hci0 dev_found: A6:35:BF:79:DD:F3 type LE Public rssi -54 flags 0x0000
AD flags 0x06
name XIAO BLE Sense
hci0 dev_found: 73:0C:7F:B1:49:94 type LE Random rssi -38 flags 0x0000
AD flags 0x1a
eir_len 31
hci0 dev_found: 2A:13:E4:A5:23:A5 type LE Random rssi -94 flags 0x0004
AD flags 0x00
eir_len 31
hci0 dev_found: 5A:62:8A:0C:FC:5B type LE Random rssi -36 flags 0x0004
AD flags 0x1a
eir_len 27
hci0 dev_found: E7:F1:5C:21:FC:1D type LE Public rssi -87 flags 0x0000
AD flags 0x06
name Mi Smart Band 5
```

After that, follow the steps **Start bluepy-helper→Connect to a Device→Discover Services Offered by Device**:

```
(base) eolab@eolab-Latitude-3340:~/bluepy/bluepy$ ./bluepy-helper
# bluepy-helper.c version 1.3.0 built at 14:58:05 on Jun 10 2024
conn A6:35:BF:79:DD:F3
rsp=$ststate=$tryconndst='A6:35:BF:79:DD:F3mtu=h0sec='low
rsp=$ststate=$conndst='A6:35:BF:79:DD:F3mtu=h0sec='low
svcs
rsp=$findhstart=h1hend=h5uuid='00001800-0000-1000-8000-00805f9b34fbhstart=h6hend=h9uuid='00001801-0000-1000-8000-00805f9b34fbhstart=hAhend=hDuuid='5d2af01a-439d-45ca-9dc7-edc49cf2a539
```

Reading and Decode



# HopeRF RFM69HCW 433/434 MHz high power module

- RFM69HCW Hookup Guide by **Sparkun**
- Antenna Section of RFM69HCW Hookup Guide by **Sparkun**
- Sparkfun RFM69HCW Breakout Github
- Datasheet RFM69HCW-V1.1.pdf

## Test

- PCB Adapter ESP 07/12 with rfm69hcw by Mobilefish.com
- Wireing for ESP8266
- using-the-rfm69-radio
- esp8266 with GY-521
- RadioHead library

## First Try

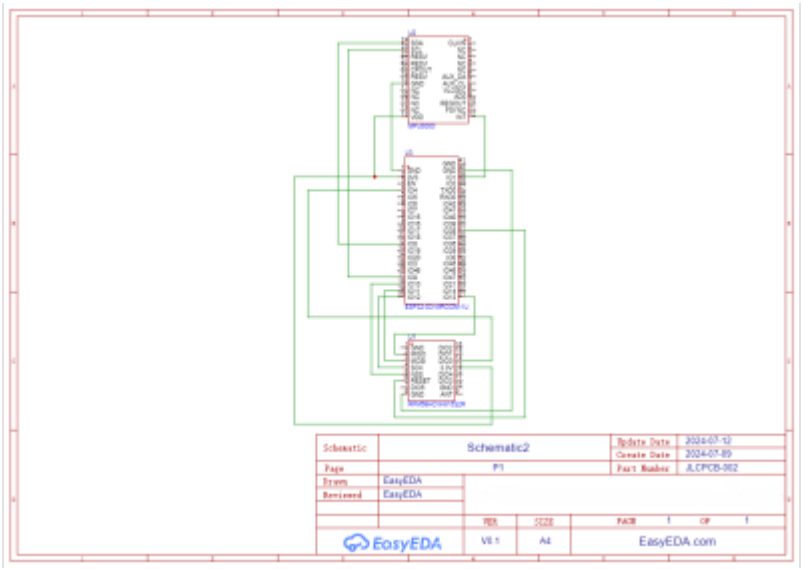
The first test with the rfm69hcw was with the esp8266 and mpu6050, the radio module needs a breakout for better mounting on a breadboard. Instead of using the SparkFun Breakout, a PCB Adapter ESP 07 / 12 – Board is used. We follow the Youtube guide by Mobilefish.com.

## Range Test of RFM69HCW breakout (with esp8266, mpu6050)

There was one module constituted by one ESP8266 board connected to the GY-521 6 axis accelerometer and transmiting the values at a frequency of 433/434 MHz with a RFM69HCW transciver module.

Receiving this data there was another ESP8266 board connected to a second RFM69HCW breakout,

but this time receiving the radio signal.  This was the module sending the data from the acelerometer.

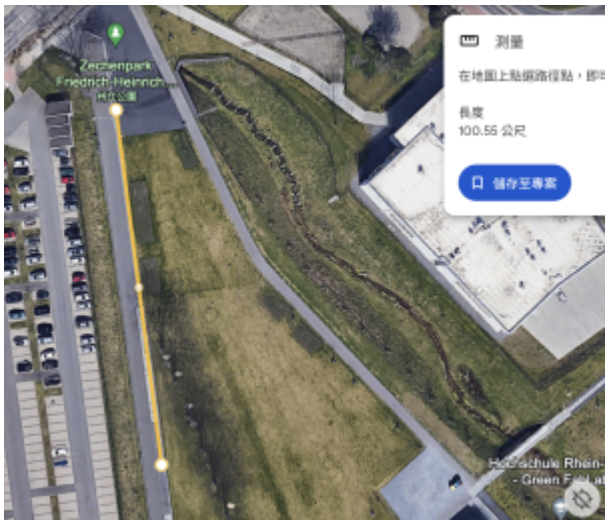



This is the schematic of the transceiver module.

The max range we managed to acheive was 140m, but the conexion was not very stable. We were not getting the values every 0.1s as stated in the code, but rather every 1s/2s. At 100m the conexion was moderate, values were obtained every 0.5s-0.3s and lastly a really good and stable conexion was reached at arround 58m.



max range



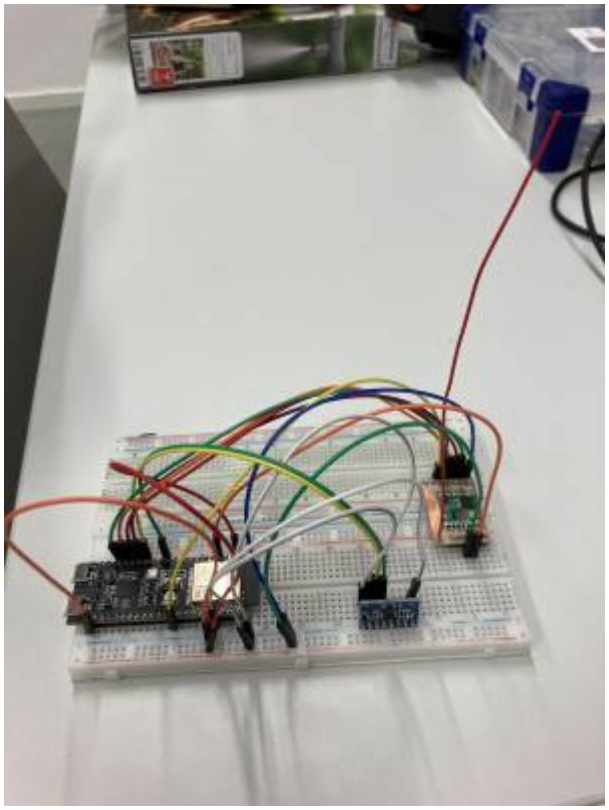
moderate conexion



very stable conexion

**The reason for not continuing with esp8266 is the unstable power supply. Esp8266 is not powerful enough to power both rfm69hcw and mpu6050. It will cause an extra complication of using more than one power source.**

## Range Test of RFM69HCW breakout (with esp32s3 dev kit, mpu6050)



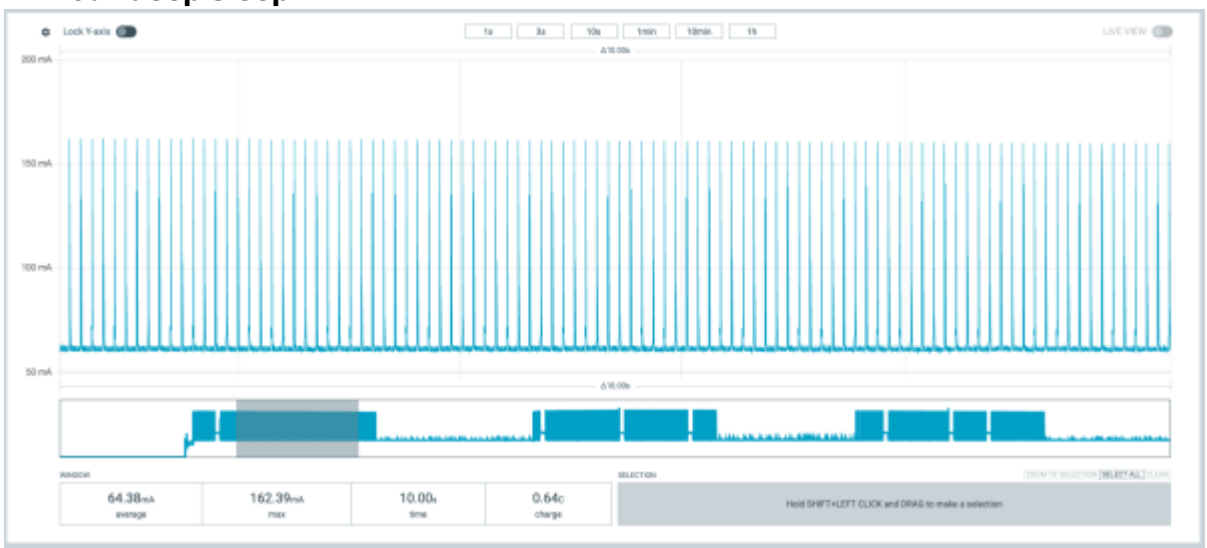
The test is then continued with esp32. To use esp32 with RFM69HCW, first download the RadioHead library, and the RH_ASK.cpp library has to be modified.

## Code

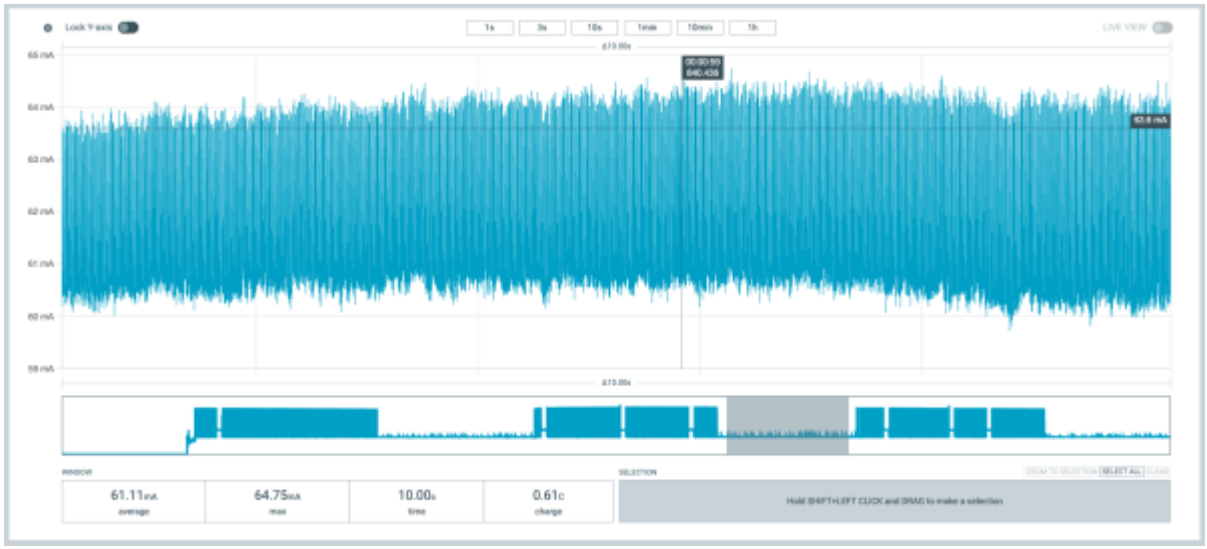
Github

## Power Consumption

### without deep sleep




When motion is detected

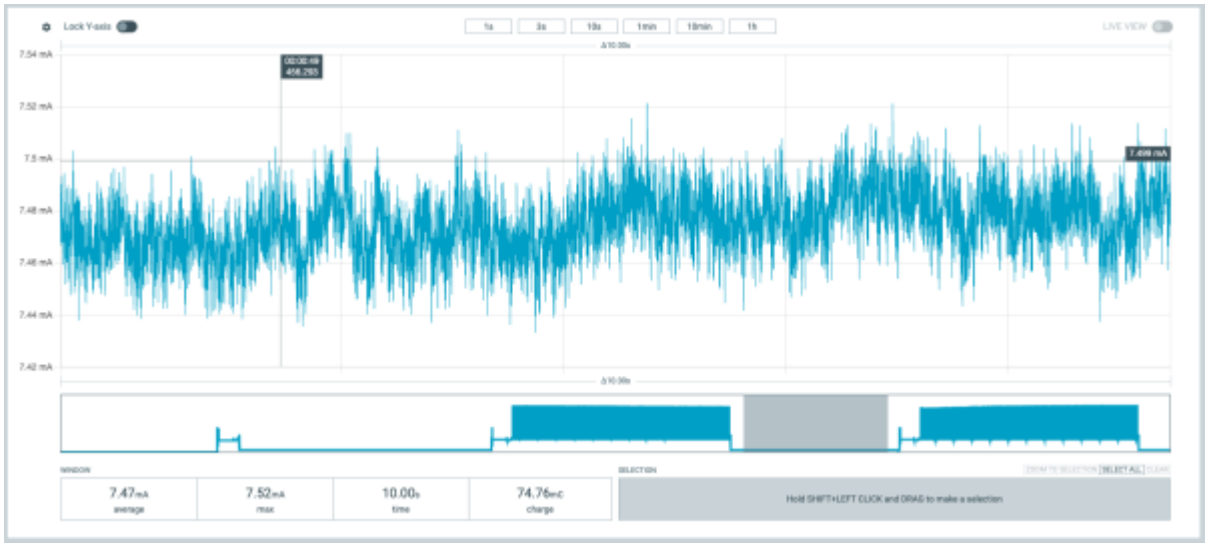



When no motion is detected

**with deep sleep**




When motion is detected




deep sleeping, 7.47mA avarage

From:
https://student-wiki.eolab.de/ - **HSRW EOLab Students Wiki**

Permanent link:
**https://student-wiki.eolab.de/doku.php?id=amc:ss2024:schafalarm:start&rev=1722370009**

Last update: **2024/07/30 22:06**