# Description

The Drone IMU BT project is an extension of the Drone Project that adds Inertial Measurement Unit (IMU) and Bluetooth (BT) capabilities to the aircraft.

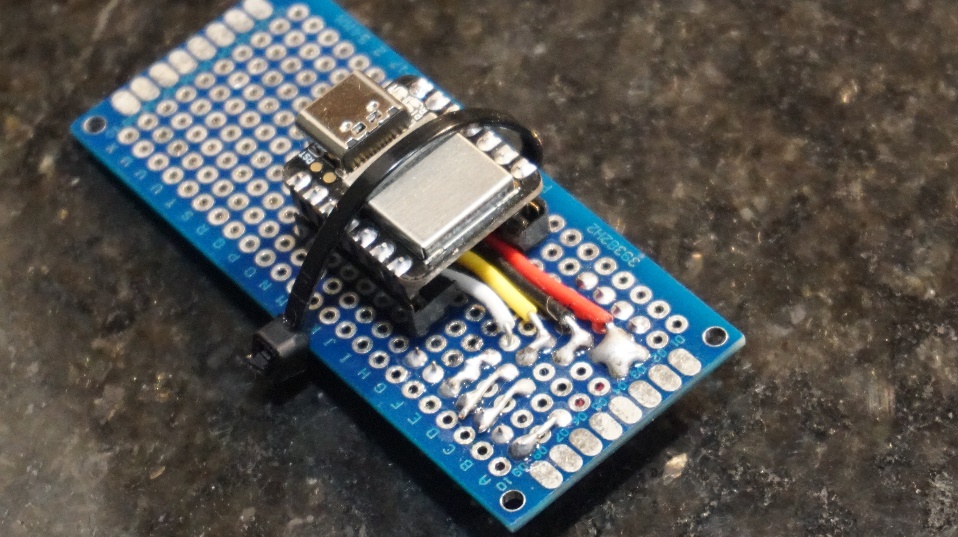


Figure – IMU Sensor Board

# Requirements

The following requirements describes the functionality set forth by the Drone IMU BT project. Note that most of the requirements were fulfilled by the Drone IMU (Project 8) requirements, the highlighted requirements below expresses the main difference:

* The MCU shall be implemented in a FreeRTOS environment.
* The MCU shall interface with a BNO055 Absolute Orientation Sensor.
* The MCU shall interface with the RPi via USB Serial.
* The MCU shall transmit Yaw, Pitch and Roll data to the RPi.
* The RPi shall read in the MCU Yaw, Pitch and Roll data and process the data for transmission to a Host.
* The RPi shall transmit to the Host via Bluetooth.
* The Host shall receive the transmitted data from the RPi.
* The Host shall display the received Yaw, Pitch and Roll transmitted data to the User.

# Design

The following sections describe the hardware, software, and configuration design decisions for the IMU Drone project.

## Hardware

The Seeed Studio XIAO SAMD21, an Arduino-like microcontroller was used as the MCU for this project. The Raspberry Pi Model 3B+ was used as the data processing computer. A second Raspberry Pi was used as the Host computer. In addition, the following materials were used:

* 1x BNO055 Absolute Orientation Sensor
* 1x 2’ USB-C cable
* 1x Perf Board
* An assortment of hookup cables.
* An assortment of pin headers.
* An assortment of chip sockets.
* Double sided mounting tape.

A picture containing text, electronics

Description automatically generated

Since the MCU is to interface with the BNO sensor onboard the Drone, it is highly recommended (if not, required) to solder electrical connections on a perf board. The MCU and BNO can be installed using chip sockets and securely zip-tied to the board. The wiring is simple, and a USB cable is used to connect the MCU to the RPi.

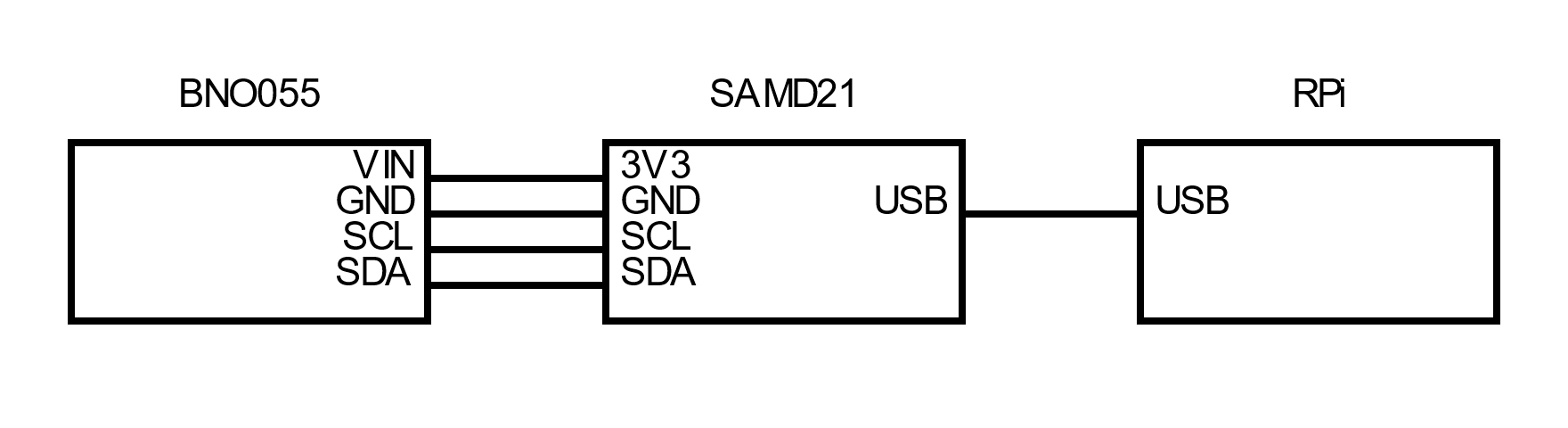


Figure – Schematic

The board was mounted at the front of the Drone, centered with the longitudinal axis of chassis.

## Software

The section is divided into the MCU (XIAO SAMD21), RPi and Host sections. Each component acts independently of software. Protocols are defined for communication between each component.

### MCU

The MCU software uses the Seeed\_Arduino\_FreeRTOS library for implementation of the RTOS. This library was specifically built to support the Seeed series microcontroller. Two task threads are defined in the software:

* SensorThread: polls the BNO sensor repeatedly and updates the global buffer.
* SerialThread: transmits the global buffer via serial as defined by the TX\_RATE.

The SerialThread is given higher priority than the SensorThread. This is to ensure that TX\_RATE is fulfilled and the SensorThread is given plenty of sampling events to occur. When loading to the global buffer occurs, the assignment of fields is surrounded by CRITICAL\_SECTION calls to prevent the SerialThread from interrupting mid-operation.

The payload that is transmitted from the MCU is defined as follows:

Diagram, table

Description automatically generated with medium confidence

A 32-bit sync word is used to ensure that the receiver can synchronize with the serial stream. If a failure occurs during the MCU operations, either no data is transmitted or garbage data is sent (that does not contain the sync word).

### RPi

The RPi (Raspberry Pi) software interfaces with the MCU via Serial USB. The MCU serial device appears as “/dev/ttyAMC1”. To ensure that the RPi receives alignment of data, a 32-bit sync word that is prefacing in the payload is scanned and the remaining payload is loaded.

The software then processes the data received. Note that both the MCU and RPi are little endian systems, when reading data no endian conversions are performed. At the time of writing, the processing performed is just a straightforward pass-thru with no calculations performed. Depending on the performance of this software, additional processing may be considered in the future (e.g. filtering, averaging, acceleration calculations, etc.…)

The RPi interfaces with another Raspberry Pi computer serving as ground control. This is through Bluetooth Peer to Peer communication. The bluetoothcfg utility was used for pairing the two devices. A specialized Bluetooth library, bluedot was used in Python 3 to handle transmission. Note that another Raspberry Pi SoC was used due to difficulty using other available devices such as an Apple iPhone.

The RPi transmits the formatted output of the payload to the Host (Raspberry Pi Ground Control).

### Host

The Host software executes on another Raspberry Pi computer. The Host software is written in Python 3 and uses the bluedot library. The Host starts a Bluetooth server for the Client RPi to connect. Once connection has been established, the content of the payload is printed to standard output, no formatting or conversions are performed.

A picture containing projector

Description automatically generated

Figure – Second Raspberry Pi (Ground Control)

# Implementation

The source code of the Drone IMU is divided into the MCU, RPi and Host sections.

## MCU

### imu\_def.hpp

//-----------------------------------------------------------------------------

/// @file imu\_def.hpp

/// @author Nate Lao (nlao1@jh.edu)

/// @brief Defines the IMU serial payload

//-----------------------------------------------------------------------------

#ifndef \_\_IMU\_DEF\_HPP\_\_

#define \_\_IMU\_DEF\_HPP\_\_

#if RPI

#include <netinet/in.h>

#define float\_t float

#endif

namespace IMU

{

    //-----------------------------------------------------------------------------

    /// @brief Defines the serial payload buffer.

    /// @details The buffer contents are defined as follows:

    //

    /// Label            | Byte Index | Type     | Valid Values/Range  | Failure Values

    /// ---------------- | ---------- | -------- | ------------------- | ------------------

    /// Sync Word        | 0..3       | U32      | 0x444F5045          | 0x43524150

    /// Yaw (X-Rot)      | 4..7       | F32      | 0..360 (CW -> +)    | 0xFFFFFFFF

    /// Pitch (Y-Rot)    | 8..11      | F32      | -90..90 (UP -> +)   | 0xFFFFFFFF

    /// Roll (Z-Rot)     | 12..15     | F32      | -180..180 (R -> -)  | 0xFFFFFFFF

    //

    //-----------------------------------------------------------------------------

    typedef struct

    {

        uint32\_t sync\_word;

        float\_t yaw;

        float\_t pitch;

        float\_t roll;

    } PAYLOAD;

    const uint32\_t SYNC\_WORD = 0x444F5045;

    const uint32\_t FAIL\_WORD = 0x43524150;

    const PAYLOAD FAILURE\_PAYLOAD = {FAIL\_WORD, (float\_t)0xFFFFFFFF, (float\_t)0xFFFFFFFF, (float\_t)0xFFFFFFFF};

}

#endif

### imu\_sensor.hpp

//-----------------------------------------------------------------------------

/// @file imu\_sensor.hpp

/// @author Nate Lao (nlao1@jh.edu)

/// @brief Defines the IMU sensor class.

//-----------------------------------------------------------------------------

#ifndef \_\_IMU\_SENSOR\_HPP\_\_

#define \_\_IMU\_SENSOR\_HPP\_\_

#include <Wire.h>

#include <Adafruit\_Sensor.h>

#include <Adafruit\_BNO055.h>

#include <utility/imumaths.h>

#include "imu\_def.hpp"

namespace IMU

{

    class Sensor

    {

    public:

        Sensor(int32\_t sensorID, IMU::PAYLOAD\* bufferAddr);

        bool begin();

        bool frame();

    private:

        IMU::PAYLOAD\* buffer;

        Adafruit\_BNO055 sensor;

        sensors\_event\_t event;

    };

}

#endif

### imu\_sensor.cpp

//-----------------------------------------------------------------------------

/// @file imu\_sensor.cpp

/// @author Nate Lao (nlao1@jh.edu)

/// @brief Defines the IMU sensor initialization and reading operations.

//-----------------------------------------------------------------------------

#include <Seeed\_Arduino\_FreeRTOS.h>

#include <Wire.h>

#include <Adafruit\_Sensor.h>

#include <Adafruit\_BNO055.h>

#include <utility/imumaths.h>

#include "include/imu\_sensor.hpp"

//-----------------------------------------------------------------------------

/// @brief IMU Sensor

//-----------------------------------------------------------------------------

IMU::Sensor::Sensor(int32\_t sensorID, IMU::PAYLOAD \*bufferAddr)

{

    buffer = bufferAddr;

    sensor = Adafruit\_BNO055(sensorID);

}

//-----------------------------------------------------------------------------

/// @brief  Initializes the IMU Sensor. Must be called before operations

/// @return true if IMU Sensor intialization was successful.

//-----------------------------------------------------------------------------

bool IMU::Sensor::begin()

{

    bool retval = sensor.begin();

    if (retval)

        sensor.setExtCrystalUse(true);

    return retval;

}

//-----------------------------------------------------------------------------

/// @brief  Loads the io read buffer with sensor data.

//-----------------------------------------------------------------------------

bool IMU::Sensor::frame()

{

    bool retval = sensor.getEvent(&event);

    /// @todo investigate low-pass filtering

    /// @todo the retval for BNO sensor may always be true, investigate

    /// invalid values as failure operations may never occur (even in

    /// the event of an actual failure)

    if (retval)

    {

        /// @note Mismatched rotation measurements can be

        /// tolerated. The higher priority task should

        /// not be able to interrupt during singular assignments

        /// (though they might be atomic by nature...)

        // Sync Word

        taskENTER\_CRITICAL();

        buffer->sync\_word = IMU::SYNC\_WORD;

        taskEXIT\_CRITICAL();

        // Yaw Measurements

        taskENTER\_CRITICAL();

        buffer->yaw = event.orientation.x;

        taskEXIT\_CRITICAL();

        // Pitch Measurements

        taskENTER\_CRITICAL();

        buffer->pitch = event.orientation.y;

        taskEXIT\_CRITICAL();

        // Roll Measurements

        taskENTER\_CRITICAL();

        buffer->roll = event.orientation.z;

        taskEXIT\_CRITICAL();

    }

    else

    {

        taskENTER\_CRITICAL();

        memcpy(buffer, &IMU::FAILURE\_PAYLOAD, sizeof(IMU::PAYLOAD));

        taskEXIT\_CRITICAL();

    }

    return retval;

}

### imu.ino

//-----------------------------------------------------------------------------

/// @file imu.ino

/// @author Nate Lao (nlao1@jh.edu)

/// @brief Main RTOS driver for the IMU Sensor.

/// @details Designed for Seeed Studio Xiao microcontroller.

//-----------------------------------------------------------------------------

// ----- RTOS LIBRARY ----- //

#include <Seeed\_Arduino\_FreeRTOS.h>

// ----- LOCAL LIBRARIES ----- //

#include "include/imu\_def.hpp"

#include "include/imu\_sensor.hpp"

const int32\_t BNO\_SENSOR\_ID = 55; /\*\* @note Sensor ID, do not change \*/

const unsigned long TX\_RATE = 50; /\*\* @note Delay per transmission (smaller -> faster) \*/

IMU::PAYLOAD buffer;

IMU::Sensor \*sensor\_driver = 0;

// ----- RTOS THREADS ----- //

/// @note Prority Ranking (larger -> higher priority)

UBaseType\_t PRIORITY\_SENSOR\_TASK = tskIDLE\_PRIORITY + 1;

TaskHandle\_t Handle\_SensorTask;

static void SensorThread(void \*pvParameters)

{

  sensor\_driver = new IMU::Sensor(BNO\_SENSOR\_ID, &buffer);

  // Attempt to initialize the sensor

  // If failed, re-attempt indefinitely

  while (!sensor\_driver->begin())

  {

    delay(100);

  }

  // Sensor Intialization Success - Normal Operation

  while (1)

  {

    sensor\_driver->frame();

  }

}

/// @note Prority Ranking (larger -> higher priority)

UBaseType\_t PRIORITY\_SERIAL\_TASK = tskIDLE\_PRIORITY + 2;

TaskHandle\_t Handle\_SerialTask;

static void SerialThread(void \*pvParameters)

{

  while (1)

  {

    Serial.write((const char \*)&buffer, sizeof(buffer));

    delay(TX\_RATE);

  }

}

// ----- MAIN ----- //

void setup(void)

{

  // Intialize the output payload to FAILURE on startup

  memcpy(&buffer, &IMU::FAILURE\_PAYLOAD, sizeof(buffer));

  // Setup Serial

  Serial.begin(9600);

  vNopDelayMS(1000); // prevents USB driver crash on startup, do not omit this

  while (!Serial)

  {

  } // Wait for Serial terminal to open port before starting program

  // Spin up the threads

  xTaskCreate(SensorThread, "IMU Sensor Sampling", 256, NULL, PRIORITY\_SENSOR\_TASK, &Handle\_SensorTask);

  xTaskCreate(SerialThread, "Serial Transmission", 256, NULL, PRIORITY\_SERIAL\_TASK, &Handle\_SerialTask);

  // Start the RTOS, this function will never return and will schedule the tasks.

  vTaskStartScheduler();

}

void loop(void)

{

  // NOTHING

}

## RPi

### flypi-bt.py

'''

@file flypi-bt.py

@author Nate Lao (nlao1@jh.edu)

@brief Simple Bluetooth Server processing for IMU/Bluetooth client interface.

'''

import serial

import struct

import bluedot.btcomm

# Must match the device to send to - raspberrypi refers to the ground station host

BT\_SERVER = "raspberrypi"

# setup bluetooth interface

bt\_socket = bluedot.btcomm.BluetoothClient("raspberrypi", None)

with serial.Serial('/dev/ttyACM1', 9600) as ser:

        while True:

                input = ser.read(16) # read in serial data from IMU

                \_, yaw, pitch, roll = struct.unpack('Lfff',input) # unpack payload

                output = "YAW={:.3f}\tPITCH={:.3f}\tROLL={:.3f}".format(yaw, pitch, roll)

                bt\_socket.send(output) # transmit formatted data

## Host (Raspberry Pi Ground Station

### groundctl-bt.py

'''

@file groundctl-bt.py

@author Nate Lao (nlao1@jh.edu)

@brief Simple Bluetooth Server processing for IMU/Bluetooth server interface.

'''

from bluedot.btcomm import BluetoothServer

from signal import pause

def data\_received(data):

    # Printout the data received, no processing (as is)

    print("recv - {}".format(data))

    server.send(data)

def client\_connected():

    print("client connected")

def client\_disconnected():

    print("client disconnected")

print("init")

# Setup bluetooth server, wait for connection

server = BluetoothServer(

    data\_received,

    auto\_start = False,

    when\_client\_connects = client\_connected,

    when\_client\_disconnects = client\_disconnected)

print("starting")

server.start()

print(server.server\_address)

print("waiting for connection")

try:

    pause()

except KeyboardInterrupt as e:

    print("cancelled by user")

finally:

    print("stopping")

    server.stop()

print("stopped")

# Maintainance Squawks

The following list are known squawks encountered during Drone development. It should be noted that some issues have been resolved during development while others are still present and monitored.

* Bad Battery (RESOLVED):
  + ~~The battery delivered was apparently INOP. An attempt was made to charge the battery with the provided charger. The charger did not indicate a charge being performed. After letting charge for approximately 2 hours, the main output terminals indicated only approximately 3 volts DC, rather than the expected ~11 volts.~~
  + Extra batteries were ordered via Amazon. Specs are similar to the original battery (voltage is matches, but the milliampere hours (mAh) and charge constants (C) are allowed to vary).
* Yaw Potentiometer Broken (RESOLVED – WORKAROUND):
  + The Yaw Potentiometer in the KK Controller seems to broken. Rotating the pot does not have a stopping point. Further investigation is needed.
  + No noticeable impact from this issue. This is a non-grounding squawk.
* Inconsistent Motor (RESOLVED)
  + ~~The aft motors seem to be slow to start. If under no load, the aft motors spin at the same time as the front, however, having a propeller load impedes the motor start.~~
  + Extra motors were ordered via Amazon. Investigation over the impact of swapping motors is to be conducted in the future.
  + ~~There is a possibility that calibration is lacking or improperly configured. Future investigation will be conducted.~~
  + Replacing the bad motor allows for an actual takeoff. More practice is needed for proficiency.
* Motion Application Issues
  + Annoyingly, support for motion startup at boot (daemon mode) does not work.
  + The framerate of the Motion RPi library is 1 FPS on browser. Need a more robust method and investigation
* Poor Camera Mounting
  + The camera will shift or break off at hard landings. Need to find a more secure way to lock position.
* No lat/long
  + The xgps GUI did not feature lat/long coordinates. 3D fix was confirmed and zulu time was determined. Possible bug?

# Lessons Learned

* XIAO SAMD21 Bricked
  + The microcontroller can be un-bricked by loading a good application on the recovery bootloader (two quick reset on pads). Note that the COM interface will be different on recovery partition. The main partition can be restored and can be reprogrammed.
* UDP Multicast
  + There is a specific range of IPs that support UDP multicast. Reference <http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml>

# FAA Registration

The Drone was registered under FAA Part 107. The Drone is designed under Tail Number (Nickname) of N42069 (pronounced “November-four-twenty-sixty-nine”) and Registration Number of FA3NL7PRFE.

Graphical user interface, text

Description automatically generated

# DEMO

A demonstration of the Drone IMU can be found here:

<https://youtu.be/VUEKnWNBtWw>