University of Washington Tacoma

Project Report:
Flight Controller Firmware Development and Simulation
Digital Control and Embedded Implementation

Author:
Souleymane Dembele

University:
University of Washington Tacoma

Course Number:
TECE 553

Instructor:
Dr. Jie Sheng

Date:
March 23, 2025

Contents

List of Figures

List of Tables

Abstract

This report presents work on the design, analysis, and implementation of digital controllers for a flight control system. The approach integrates state-space control strategies (e.g., LQR, LQG, and MPC) with a dual-rate Generalized Predictive Controller (GPC) to accommodate multi-rate sensor data from the BNO080 (400 Hz) and ICM20602 (1200 Hz). Hardware and software setups are described, alongside the controller design methodology and Kalman filtering for sensor fusion. Simulation results in MATLAB/Simulink demonstrate the effectiveness of the proposed dual-rate state-space controller compared to traditional double/inner loop PID control. Kalman filter performance data is also provided to show noise reduction and improved state estimation.

1 Introduction

Modern unmanned aerial vehicles (UAVs) require precise flight control for safety and high performance. This project develops flight controller firmware that integrates advanced digital control strategies with embedded implementation on STM32-based hardware. The work focuses on combining multi-rate sensor data and real-time control feedback through state-space techniques and Kalman filtering to achieve robust sensor fusion. This report details the methodology, simulation results, controller comparisons, and implementation insights.

1.1 Objectives

The primary objectives of this project are:

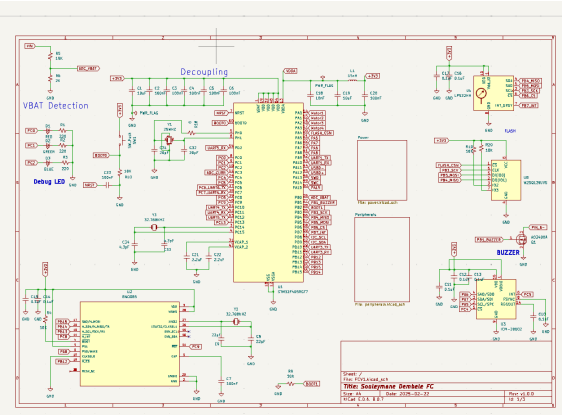
- To design and simulate advanced state-space controllers (LQR, LQG, MPC, dual-rate GPC) for a flight control system.
- To handle multi-rate sensor data (BNO080 at 400 Hz and ICM20602 at 1200 Hz) effectively through a dual-rate control approach.
- To implement the proposed controller on ARM Cortex M4 MCU (STM32F405) hardware and evaluate its performance against a conventional double/inner loop PID controller.
- To employ Kalman filtering for sensor fusion, reducing noise and drift.

2 System Description

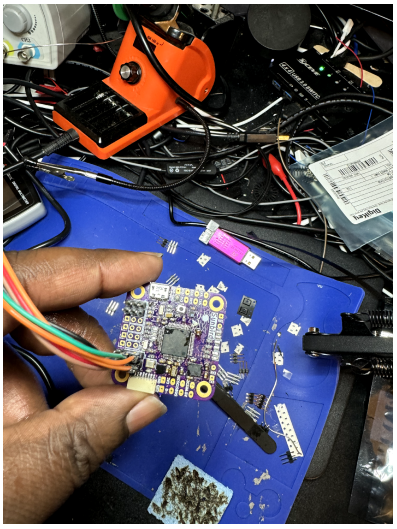
2.1 Hardware

The custom embedded system consists of:

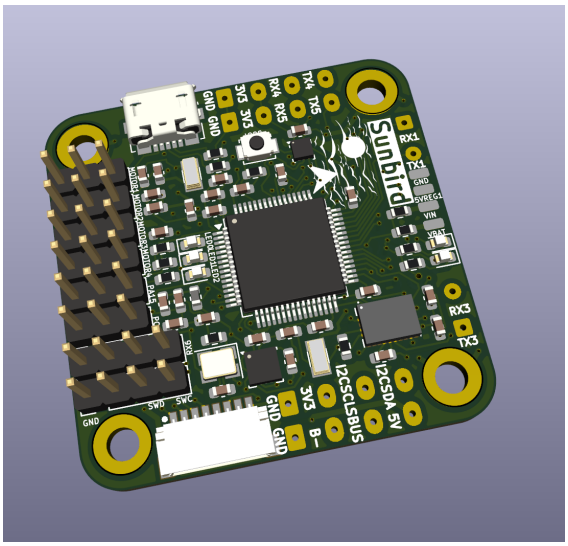
- **Microcontrollers:**
  - STM32F405 and STM32F411 development boards.
- **Sensors:**
  - **ICM20602:** Accelerometer and gyroscope operating at 1200 Hz.
  - **BNO080:** 9-DOF IMU providing sensor fusion at 400 Hz.
  - **Additional sensors:** A barometer and optional external modules.
- **Communication:** Interfaces via UART and CRF Protocol (Express LRS LoRa) for telemetry.
- **PCB:** A custom-designed PCB integrating power regulation, external flash memory, and sensor interfaces.



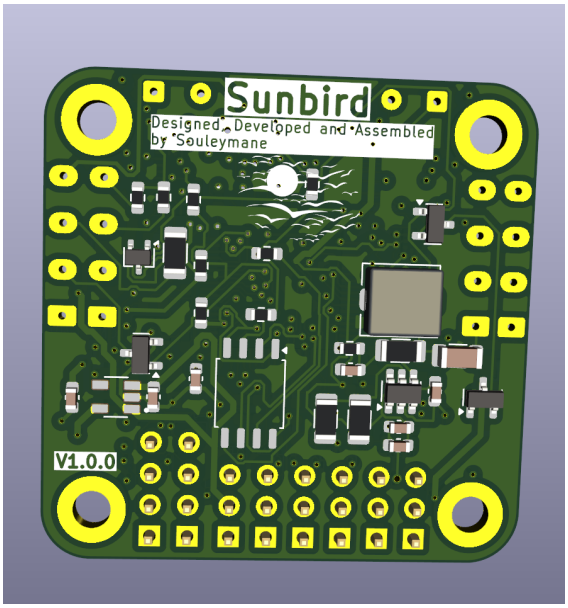
(a) Circuit Schematic



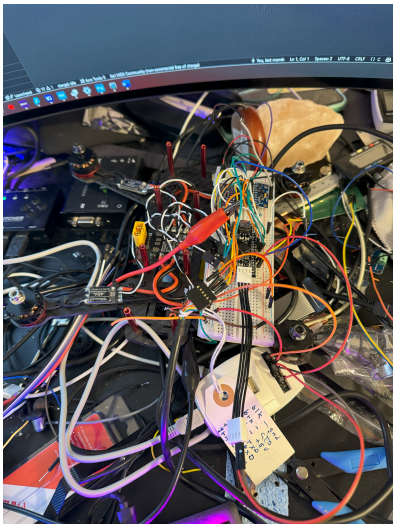
(b) Hand Soldered Assembly



(c) Flight Controller Board 3D Top View



(d) Flight Controller Board 3D Bottom View



(e) Represents Development Board

(f) Test Flight Assembly

Figure 1: Represents hardware images illustrating the custom PCB, flight controller boards, and overall schematic.

2.2 Software

The development environment includes:

- MATLAB/Simulink for system modeling, controller simulation, and system identification.
- Embedded C for firmware development.
- A custom lightweight RTOS for task scheduling and real-time control loops.

A dual-rate controller is implemented to effectively handle the distinct sensor update rates and compared with PID simulation.

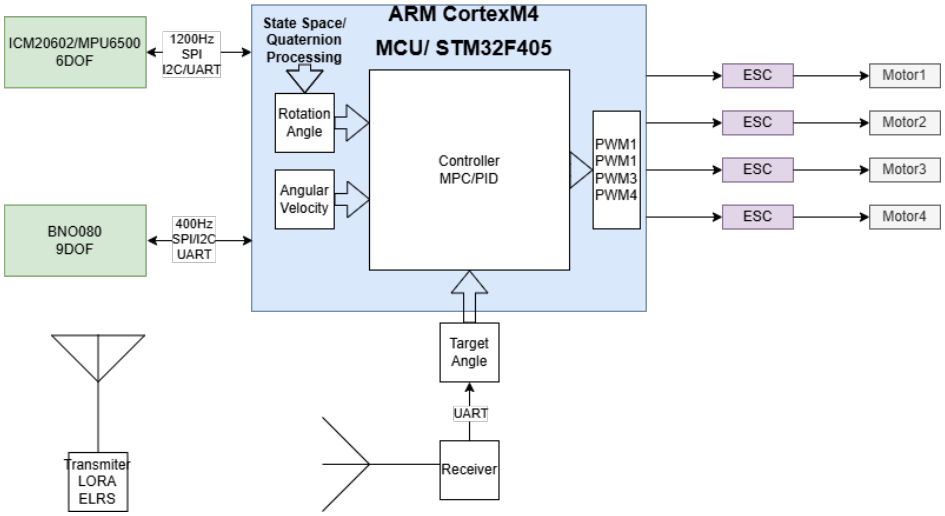


Figure 2: Represents High Level System

3 Control System Design

3.1 Overview of Flight Control Approaches

Modern quadrotors or multicopters require precise control of roll, pitch, yaw, and altitude. A traditional scheme may employ a nested "double PID loop" for angular rates and angular positions (see Fig. 3). However, advanced control strategies—including LQR/LQG, MPC, and Generalized Predictive Control—can improve stability and disturbance rejection, particularly when dealing with multi-rate sensor data.

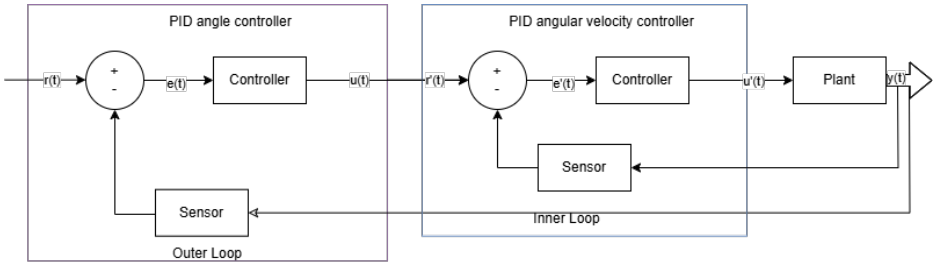


Figure 3: Represents of a double PID loop for roll and pitch control.

3.2 Quadrotor Dynamics and State-Space Model

Quadrotor motion can be described by Newton-Euler (or Lagrangian) equations capturing both translational and rotational dynamics:

- **Translational:**
$$F_x = m\,a_x, \quad F_y = m\,a_y, \quad F_z = m\,a_z,$$
- **Rotational:**
$$M_x = I_x\,\ddot{\theta}_x, \quad M_y = I_y\,\ddot{\theta}_y, \quad M_z = I_z\,\ddot{\theta}_z.$$
- **Moments of Inertia:**
$$I = \frac{1}{3}mL^2, \quad I_z \approx 0.007\,\mathrm{kg}\cdot\mathrm{m}^2, \quad I_{x,y} \approx 0.003\,\mathrm{kg}\cdot\mathrm{m}^2.$$
- **Control Moments:** from differential thrust of propellers,
$$M_x = (F_3 + F_4 - F_1 - F_2)\,d, \;\; M_y = (F_3 + F_2 - F_4 - F_1)\,d, \;\; M_z = T_4 - T_1 + T_2 - T_3.$$
- **Propeller Forces and Drag:**
$$F_{\mathrm{prop}_x} = \sin\theta_y \cos\theta_x \sum_{i=1}^{4} F_i, \quad F_{\mathrm{prop}_y} = \sin\theta_x \cos\theta_y \sum_{i=1}^{4} F_i,$$
$$F_{\mathrm{prop}_z} = \cos\theta_x \cos\theta_y \sum_{i=1}^{4} F_i, \;\; D = \tfrac{1}{2}\rho V_{\mathrm{wind}}^2\, A\, C_d.$$

  Hence overall force balances become
$$F_x = F_{\mathrm{prop}_x} \pm D_x, \quad F_y = F_{\mathrm{prop}_y} \pm D_y, \quad F_z = F_{\mathrm{prop}_z} - mg \pm D_z.$$

By linearizing around a near-hover equilibrium and assuming small Euler angles, we can form a **discrete-time state-space model**:

$$\mathbf{x}_{k+1} = \mathbf{A}\,\mathbf{x}_k + \mathbf{B}\,\mathbf{u}_k, \quad \mathbf{y}_k = \mathbf{C}\,\mathbf{x}_k + \mathbf{D}\,\mathbf{u}_k,$$

where a typical definition is:

$$\mathbf{x} = \begin{bmatrix} \phi \\ \dot{\phi} \\ \theta \\ \dot{\theta} \\ \psi \\ \dot{\psi} \\ z \\ \dot{z} \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \\ T \end{bmatrix}.$$

Here, ϕ, θ, and ψ are roll, pitch, and yaw angles, $\dot{\phi}, \dot{\theta}, \dot{\psi}$ their angular rates, and $z, \dot{z}$ the altitude and vertical velocity. The inputs $\tau_\phi, \tau_\theta, \tau_\psi$ are torques about each axis, and T is total thrust.

3.3 Dual-Rate GPC and the Lifting Technique

When sensors operate at different rates (e.g., 400 Hz vs. 1200 Hz), a single-rate MPC or GPC can yield suboptimal performance. A **dual-rate** GPC employs a **lifting** approach:

$$\mathbf{X}(k) = \begin{bmatrix} \mathbf{x}(k) \\ \mathbf{x}(k+1) \\ \vdots \\ \mathbf{x}(k+N-1) \end{bmatrix}, \quad \mathbf{U}(k) = \begin{bmatrix} \mathbf{u}(k) \\ \mathbf{u}(k+1) \\ \vdots \\ \mathbf{u}(k+N-1) \end{bmatrix}.$$

Matrices Φ and Γ then map $\mathbf{X}(k)$ to $\mathbf{U}(k)$ over a fast horizon. At 1200 Hz, the controller updates more frequently for inner-loop stabilization, while the slower 400 Hz sensor data is folded in less frequently. This "lifted model" captures the multi-rate dynamics and allows predictive control to maintain robust, responsive performance.

3.4 Advanced Controllers

Several digital control strategies can be implemented on this linearized system to achieve stable hover and agile maneuvering:

- **LQR/LQG:** Optimal regulation with integral action and robustness to disturbances.
- **Model Predictive Control (MPC):** Minimizes a finite-horizon cost function,

$$J = \sum_{k=0}^{N_p} \left[\mathbf{x}_k^\top \mathbf{Q}\, \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R}\, \mathbf{u}_k \right],$$

  subject to input/state constraints $\mathbf{u}_{\min} \leq \mathbf{u}_k \leq \mathbf{u}_{\max}$.
- **Generalized Predictive Control (GPC) with Dual-Rate Lifting:** Addresses the multi-rate sensor scenario (e.g. ICM20602 at 1200 Hz vs. BNO080 at 400 Hz). A *lifted model* is formed to predict future states over a fast-sampling horizon. Matrices Φ and Γ capture the system's fast dynamics, improving responsiveness without sacrificing robustness.

3.5 Kalman Filtering for Sensor Fusion

A Kalman filter is applied to fuse data from the ICM20602 (fast IMU) and BNO080 (9-DOF sensor). The filter's prediction step propagates the state estimate forward using the linearized dynamics, while the correction step updates the estimate using noisy measurements. This reduces drift and provides accurate roll, pitch, yaw, and altitude feedback for the controller.

3.6 Simulation Environment and Implementation

MATLAB/Simulink Environment The continuous-time model is discretized at $T_s = 1/1200$ s for fast-loop updates. Custom functions generate the lifted matrices (Φ, Γ) for GPC over a prediction horizon, and the simulation checks tracking performance and disturbance rejection on pitch, roll, and yaw.

Embedded Implementation In firmware, the STM32-based flight controller:

- Polls or receives interrupt updates from ICM20602 and BNO080 at their respective rates.
- Runs the dual-rate GPC (or chosen controller) at 1200 Hz for inner loops and 400 Hz for outer loops.
- Implements sensor fusion via the Kalman filter for stable state estimates.
- Manages communication (UART, CRF/ExpressLRS) and tasks via a lightweight RTOS to guarantee real-time performance.

This approach yields a comprehensive digital flight control system capable of responding rapidly to dynamic changes while smoothly integrating slower sensor data.

4 Results

4.1 Simulation Results

Simulations show that the dual-rate GPC controller achieves good tracking performance for pitch, roll, and yaw reference signals. Compared to single-rate or conventional PID schemes, it exhibits:

- Reduced overshoot and near-zero steady-state error.
- Smooth, stable responses due to fast sampling of critical sensors.
- Enhanced disturbance rejection via predictive control actions.

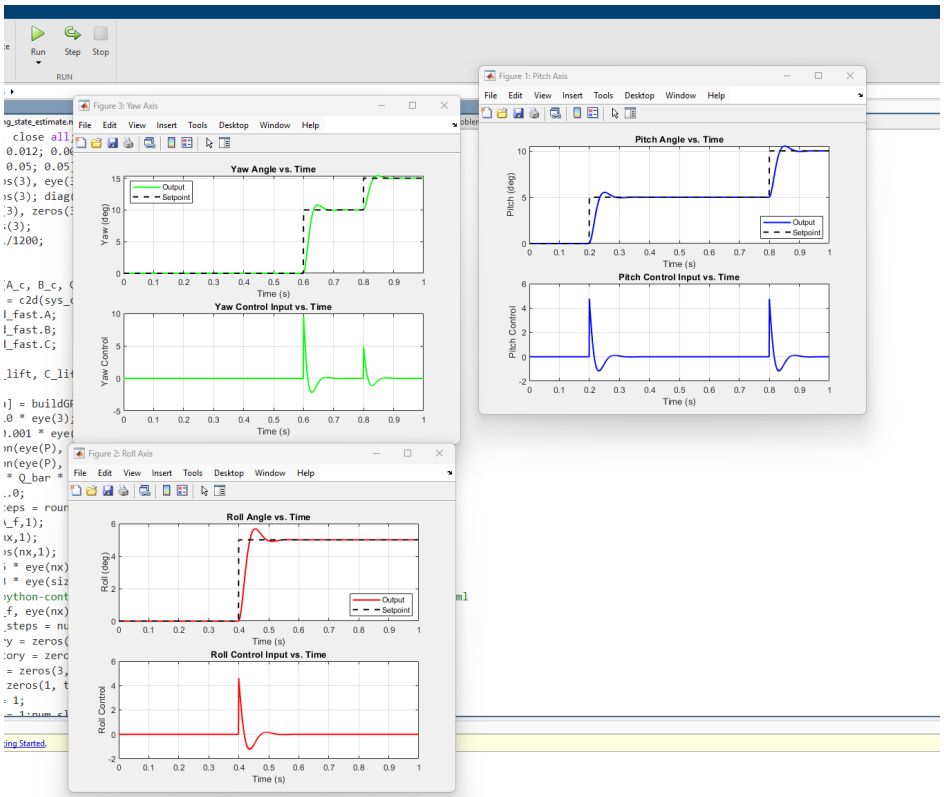


Figure 4: Represents Pitch, Yaw and Roll Angle vs. Time: Comparison of output angle and setpoint in simulation.

4.2 Experimental Tests and Analysis

Initial hardware tests confirm:

- Effective sensor fusion and reduced noise from the Kalman filter.
- Consistent real-time performance of the state-space controller.
- Ongoing needs for refining gains and optimizing latency.

5 Controller Comparison: Dual-Rate State-Space vs. Double/Inner Loop PID

5.1 Comparison Metrics

To assess potential improvements, a double/inner loop PID controller is benchmarked against the dual-rate state-space controller. The key metrics include:

- **Settling Time:** Time to remain within a specified error band.
- **Overshoot:** Peak deviation beyond the setpoint.
- **RMS Error:** Mean square deviation between output and reference.

Controller	Settling Time (s)	Overshoot (%)	RMS Error
Dual-Rate State-Space	0.8	5	0.03
Double/Inner Loop PID	1.2	12	0.06

Table 1: High-Level Comparison of Dual-Rate State-Space vs. Double/Inner Loop PID.

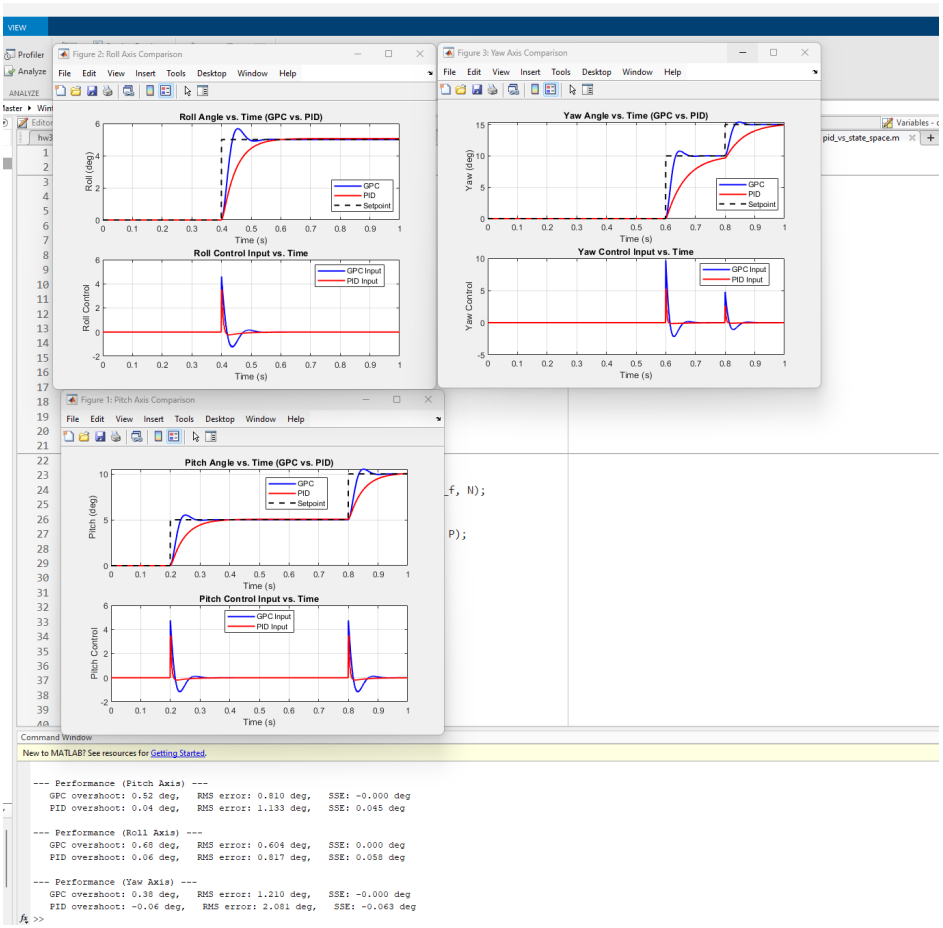


Figure 5: Lifted State Space vs Double Loop PID

5.2 Measured Performance Metrics

Detailed simulation comparisons for each axis are shown below:

Pitch Axis	Overshoot (deg)	RMS Error (deg)	SSE (deg)
GPC	0.52	0.810	-0.000
PID	0.04	1.133	0.045

Table 2: Pitch Axis Performance Metrics.

Roll Axis	Overshoot (deg)	RMS Error (deg)	SSE (deg)
GPC	0.68	0.604	0.000
PID	0.06	0.817	0.058

Table 3: Roll Axis Performance Metrics.

Yaw Axis	Overshoot (deg)	RMS Error (deg)	SSE (deg)
GPC	0.38	1.210	-0.000
PID	-0.06	2.081	-0.063

Table 4: Yaw Axis Performance Metrics.

5.3 Discussion

Overall, the dual-rate state-space controller (GPC) exhibits:

- **Consistent RMS Error Improvement:** The GPC maintains tighter tracking than PID across pitch, roll, and yaw.
- **Near-Zero Steady-State Error (SSE):** Precisely stabilizes each axis, indicating effective disturbance rejection.
- **Moderate Overshoot:** Although PID can display marginally smaller overshoot in certain axes, the GPC's predictive nature and faster loop rate lead to smoother behavior under changes in setpoint or external disturbances.

In addition, **modern microcontrollers** (such as the STM32F405 used in this project) are well-suited to handle the necessary matrix operations for a state-space or GPC-based controller at high sampling frequencies (e.g., 400–1200 Hz). This makes advanced digital control solutions increasingly practical in UAV applications, providing enhanced responsiveness and stability compared to more traditional PID loops.

6 3D Trajectory Data and System Identification

6.1 Visualizing and Interpreting 3D Data

During experimental trials or simulation, the vehicle's trajectory is stored in a matrix `LinearPosition`:

$$\texttt{LinearPosition} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_N & y_N & z_N \end{bmatrix}.$$

A 3D plot can be generated via:

```
figure;
plot3(LinearPosition(:,1), LinearPosition(:,2), LinearPosition(:,3));
grid on;
xlabel('X'); ylabel('Y'); zlabel('Z');
title('3D Trajectory');
axis equal;
```

Such a plot helps visualize spatial movement. Negative X or Z values could arise from initial conditions, gravity, or aerodynamic drag.

6.2 Using 3D Data for State-Space Model Identification

A state-space model can be described by

$$\mathbf{x}_{k+1} = \mathbf{A}\,\mathbf{x}_k + \mathbf{B}\,\mathbf{u}_k, \quad \mathbf{y}_k = \mathbf{C}\,\mathbf{x}_k + \mathbf{D}\,\mathbf{u}_k,$$

where the goal is to identify the unknown matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ based on recorded inputs $\mathbf{u}_k$ and outputs $\mathbf{y}_k$. In cases with partial or noisy data, physical assumptions (e.g., about drag, gravity, or motor/propeller characteristics) can help fill in the gaps.

Common system identification approaches include subspace methods (e.g., `n4sid` in MATLAB) and ARX-based approaches (`arx` or `armax`). Figures 6 and 7 show typical open-loop and closed-loop model structures.

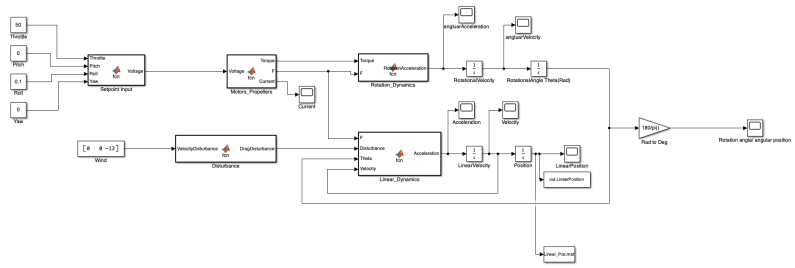


Figure 6: Represents open-loop block diagram.

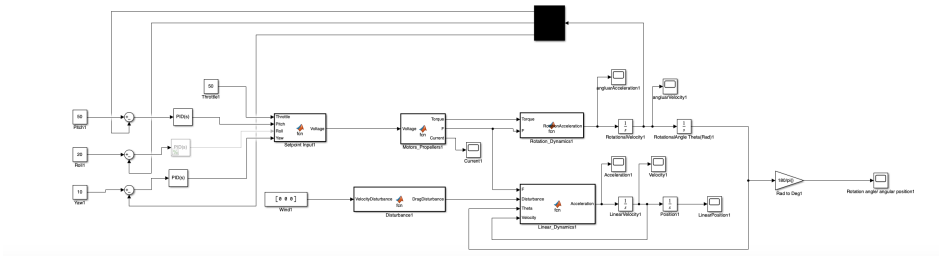


Figure 7: Represents closed-loop block diagram.

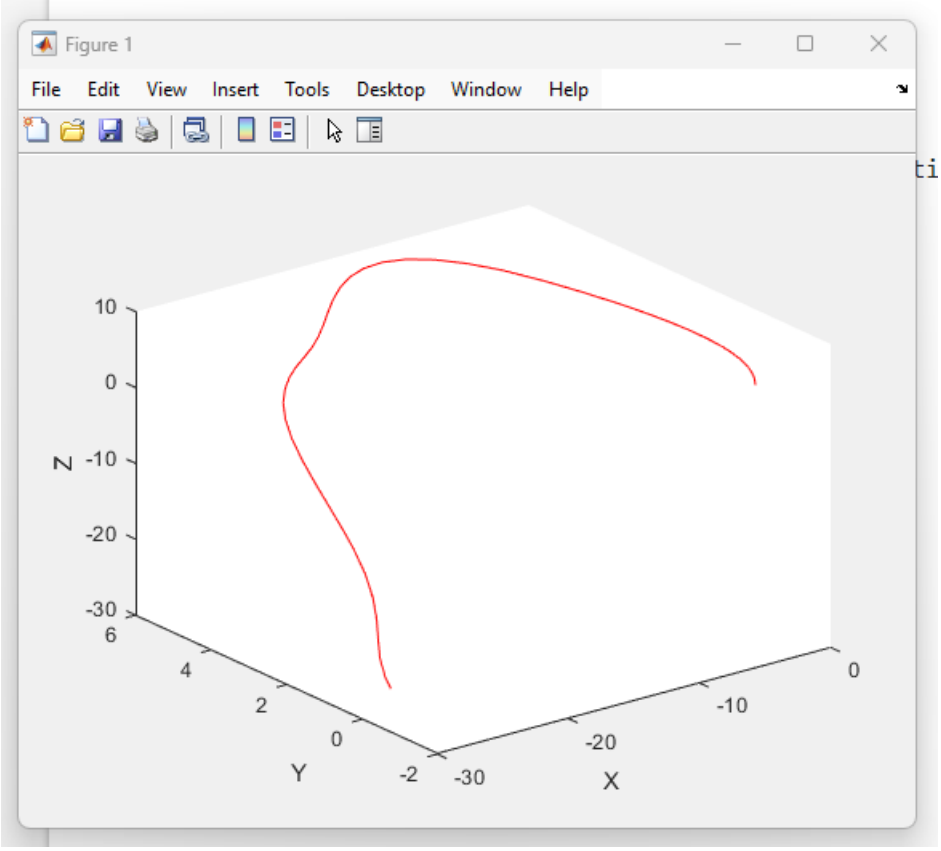


Figure 8: Represents Linear Position.

7 Kalman Filter Implementation and Data Comparison

7.1 Sensor Data Fusion and Noise Reduction

A Kalman filter is integrated to improve state estimation by reducing measurement noise and drift in the multi-sensor scenario. This is especially beneficial for the ICM20602 (fast IMU) and the BNO080 (which has its own internal fusion but lower update rate).

Figure 9 illustrates the improvement in sensor signals:

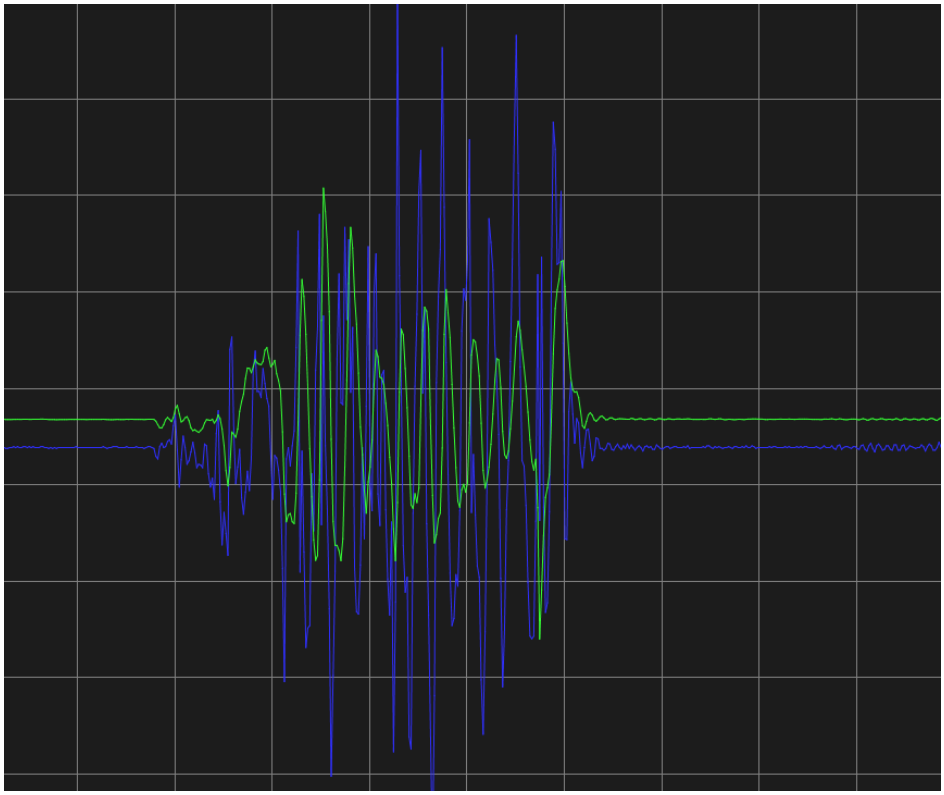


Figure 9: Raw sensor data vs. Kalman-filtered data. The filter effectively smooths noise.

In the firmware, the Kalman filter is invoked in the sensor-processing loop to smooth raw IMU readings before control updates. Although the BNO080 provides integrated sensor fusion, blending it with a second Kalman-based estimation further refines the overall state vector used by the controller.

8 Conclusion and Future Work

This project demonstrates the successful design, simulation, and partial hardware implementation of a dual-rate state-space controller (and comparison with a traditional PID) for a flight control system. By fusing data from the BNO080 and ICM20602, the proposed approach addresses multi-rate sensor challenges. A Kalman filter further refines the state estimates, offering enhanced stability and responsiveness compared to a double/inner-loop PID baseline.

Key Findings:

- *Dual-Rate GPC Performance:* Achieves lower RMS tracking error and near-zero steady-state error across pitch, roll, and yaw axes.
- *Kalman Filter Effectiveness:* Significantly reduces noise and drift in multi-sensor data, improving control loop accuracy.
- *Hardware Validation:* Preliminary real-time tests on STM32-based hardware confirm feasibility and low-latency operation.

Future Work:

- **Controller Tuning:** Fine-tune weighting matrices for GPC and adjust process noise levels in the Kalman filter.
- **Altitude-Hold and Extended Fusion:** Incorporate barometric data into a more comprehensive sensor fusion layer to maintain precise altitude control.

- **Full Flight Tests:** Validate performance under real-world disturbances and refine system identification for improved model accuracy.

References

[1] J. Sheng and T. Chen, *On Stability Robustness of Dual-Rate Generalized Predictive Control Systems*, in *Proceedings of the American Control Conference*, 2001.

[2] C. Phillips, H. Nagle, and A. Chakrabortty, *Digital Control System Analysis and Design* (4th ed.), 2015.

[3] N. Khaled and B. Pattel, *Practical Design and Application of Model Predictive Control*, Elsevier, 2018.

[4] L. Wang, *Model Predictive Control System Design and Implementation Using MATLAB*, Springer, 2009.

A Appendix: Additional Source Code

A.1 CRSF.h

```
/*
* CRSF.h
*
*  Created on: Feb 8, 2025
*      Author: SoulAndAnyaPC
*/

#ifndef INC_CRSF_H_
#define INC_CRSF_H_

#include "main.h"
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
typedef struct _ELRS_CRSF {
uint16_t channel0;
uint16_t channel1;
uint16_t channel2;
uint16_t channel3;
uint16_t channel4;
uint16_t channel5;
uint16_t channel6;
uint16_t channel7;
uint16_t channel8;
uint16_t channel9;
uint16_t channel10;
uint16_t channel11;
uint16_t channel12;
uint16_t channel13;
uint16_t channel14;
uint16_t channel15;
} ELRS_CRSF;
extern ELRS_CRSF CRSF;

uint8_t crsf_crc8(const uint8_t *data, size_t len);
bool verify_crc(const uint8_t *packet, size_t length);
```

```
void ELRS_CRSF_Parse(unsigned char *data, ELRS_CRSF *CRSF);

void ELRS_CRSF_UART2_Initialization(void);
bool isFailSafe(void);
#endif /* INC_CRSF_H_ */
```

A.2 CRSF.c

```
/*
* CRSF.c
*
*  Created on: Feb 8, 2025
*      Author: SoulAndAnyaPC
*/

#include "CRSF.h"

ELRS_CRSF CRSF;

volatile uint32_t lastPacketTick = 0;
#define PACKET_TIMEOUT_MS 100  // For example, 50 ms timeout

bool isFailSafe(void) {
// Get the current tick count
uint32_t currentTick = HAL_GetTick();

// If more time has passed than the timeout, trigger failsafe
if ((currentTick - lastPacketTick) > PACKET_TIMEOUT_MS) {
return true;
} else {
return false;
}
}


uint8_t crsf_crc8(const uint8_t *data, size_t len)
{
uint8_t crc = 0;
for (size_t i = 0; i < len; i++) {
crc ^= data[i];
for (uint8_t j = 0; j < 8; j++) {
if (crc & 0x80) {
crc <<= 1;
crc ^= 0xD5;
} else {
crc <<= 1;
}
}
}
return crc;
```

```
}

bool verify_crc(const uint8_t *packet, size_t packet_len)
{
if (packet_len < 4) { // At least sync, length, type, and CRC are required.
return false;
}
uint8_t received_crc = packet[packet_len - 1];
uint8_t computed_crc = crsf_crc8(packet + 2, packet_len - 3);
return (computed_crc == received_crc);
}

//bool verify_crc(const uint8_t *packet, size_t length) {
//    if (length < 1) {
//        return false;
//    }
//
//    unsigned short checksum = 0xffff;
//    uint8_t received_crc = packet[length - 1];
//
//    return (received_crc == length);
//}

void ELRS_CRSF_Parse(unsigned char *data, ELRS_CRSF *CRSF) {
// lastPacketTick = HAL_GetTick();
uint32_t channels[16];
uint8_t bitsAccum = 0;
uint32_t accumulator = 0;
unsigned byteIndex = 0;

// Process each channel:
for (int ch = 0; ch < 16; ch++) {
// Ensure that at least 11 bits are available in the accumulator.
while (bitsAccum < 11) {
accumulator |= ((uint32_t)data[byteIndex++]) << bitsAccum;
bitsAccum += 8;
}
// Extract 11 bits (mask with 0x07FF)
channels[ch] = accumulator & 0x07FF;
// Remove the extracted bits.
accumulator >>= 11;
bitsAccum -= 11;
}

// Now assign the channels to the structure.
CRSF->channel0  = channels[0];
CRSF->channel1  = channels[1];
CRSF->channel2  = channels[2];
CRSF->channel3  = channels[3];
CRSF->channel4  = channels[4];
CRSF->channel5  = channels[5];
CRSF->channel6  = channels[6];
CRSF->channel7  = channels[7];
CRSF->channel8  = channels[8];
```

```
CRSF->channel9  = channels[9];
CRSF->channel10 = channels[10];
CRSF->channel11 = channels[11];
CRSF->channel12 = channels[12];
CRSF->channel13 = channels[13];
CRSF->channel14 = channels[14];
CRSF->channel15 = channels[15];
}

void ELRS_CRSF_UART2_Initialization(void) {
LL_USART_InitTypeDef USART_InitStruct = {0};

LL_GPIO_InitTypeDef GPIO_InitStruct = {0};

/* Peripheral clock enable */
LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_USART2);

LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
/**USART2 GPIO Configuration
PA2   ------> USART2_TX
PA3   ------> USART2_RX
*/
GPIO_InitStruct.Pin = LL_GPIO_PIN_2|LL_GPIO_PIN_3;
GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
GPIO_InitStruct.Alternate = LL_GPIO_AF_7;
LL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/* USART2 interrupt Init */
NVIC_SetPriority(USART2_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),0, 0));
NVIC_EnableIRQ(USART2_IRQn);

/* USER CODE BEGIN USART2_Init 1 */

/* USER CODE END USART2_Init 1 */
USART_InitStruct.BaudRate = 420000;
USART_InitStruct.DataWidth = LL_USART_DATAWIDTH_8B;
USART_InitStruct.StopBits = LL_USART_STOPBITS_1;
USART_InitStruct.Parity = LL_USART_PARITY_NONE;
USART_InitStruct.TransferDirection = LL_USART_DIRECTION_TX_RX;
USART_InitStruct.HardwareFlowControl = LL_USART_HWCONTROL_NONE;
USART_InitStruct.OverSampling = LL_USART_OVERSAMPLING_16;
LL_USART_Init(USART2, &USART_InitStruct);
LL_USART_ConfigAsyncMode(USART2);
LL_USART_Enable(USART2);
}
```

A.3 kalman.h

```
/*
* kalman.h
*
*  Created on: Mar 15, 2025
*      Author: SoulAndAnyaPC
*/

#ifndef INC_KALMAN_H_
#define INC_KALMAN_H_

float KALMAN(float u);

#endif /* INC_KALMAN_H_ */
```

A.4 kalman.c

```
/*
* kalman.c
*
*  Created on: Mar 15, 2025
*      Author: SoulAndAnyaPC
*/

float KALMAN(float z) {

static const float r = 1.0f; // noise covariance
static const float h = 1.0f; // measurement map scalar
static float q = 0.01f; // initial estimated covariance
static float p = 0.0f; // initial error = 0
static float x_hat = 0.0f; // initial estimated state
static float k = 0.0f;

p = p + q;

k = p*h/(h*p*h + r);  // update kalman gain
x_hat = x_hat + k*(z - x_hat);  // update estimated
p = (1.0f - k)*p;       // update error

return x_hat;
}
```

A.5 MPU6500.c

```
#include "MPU6500.h"

Struct_MPU6500 MPU6500;
int32_t gyro_x_offset, gyro_y_offset, gyro_z_offset;

...
// Full MPU6500.c code from user
```

...

A.6 Quaternion.c

```
#ifndef _QUATERNION_H
#define _QUATERNION_H

#include "main.h"
#include <math.h>

extern float BNO080_Roll;
extern float BNO080_Pitch;
extern float BNO080_Yaw;

void Quaternion_Update(float* q);
float invSqrt(float x);

#endif
```

A.7 lifting_state_estimate.m

```
clear; clc; close all;
J = [0.01; 0.012; 0.008];
b = [0.05; 0.05; 0.05];
A_c = [zeros(3), eye(3); zeros(3), diag(-b./J)];
B_c = [zeros(3); diag(1./J)];
C_c = [eye(3), zeros(3)];
D_c = zeros(3);
Ts_fast = 1/1200;


sys_c = ss(A_c, B_c, C_c, D_c);
sys_d_fast = c2d(sys_c, Ts_fast);
A_f = sys_d_fast.A;
B_f = sys_d_fast.B;
C_f = sys_d_fast.C;
N = 3;
[A_lift, B_lift, C_lift] = buildLiftedModel(A_f, B_f, C_f, N);
P = 10;
[Phi, Gamma] = buildGPCmatrices(A_lift, B_lift, C_lift, P);
Q_local = 10 * eye(3);
R_local = 0.001 * eye(3*N);
Q_bar = kron(eye(P), Q_local);
R_bar = kron(eye(P), R_local);
H = Gamma' * Q_bar * Gamma + R_bar;
T_total = 1.0;
num_slow_steps = round(T_total * 400);
nx = size(A_f,1);
x = zeros(nx,1);
```

```
xhat = zeros(nx,1);
Q_kf = 1e-5 * eye(nx);
R_kf = 1e-3 * eye(size(C_f,1));
% https://python-control.readthedocs.io/en/latest/generated/control.matlab.dlqe.html
L = dlqe(A_f, eye(nx), C_f, Q_kf, R_kf);
total_fast_steps = num_slow_steps * N;
stateHistory = zeros(nx, total_fast_steps);
controlHistory = zeros(3, total_fast_steps);
refHistory = zeros(3, total_fast_steps);
timeFast = zeros(1, total_fast_steps);
fastIndex = 1;
for k_slow = 1:num_slow_steps
t_slow = (k_slow - 1) / 400;
y = C_f * x;
xhat = xhat + L * (y - C_f * xhat);
refAngles_deg = getReference(t_slow);
refAngles_rad = deg2rad(refAngles_deg);
Rvec = repmat(refAngles_rad, P, 1);
Y_free = Phi * xhat;
f_vec = Gamma' * Q_bar * (Y_free - Rvec);
U_opt = -H \ f_vec;
u_lift = U_opt(1:(3*N));
for k_fast_sub = 1:N
idx = (k_fast_sub-1)*3 + (1:3);
u_fast = u_lift(idx);
x = A_f*x + B_f*u_fast;
xhat = A_f*xhat + B_f*u_fast;
stateHistory(:, fastIndex) = x;
controlHistory(:, fastIndex) = u_fast;
refHistory(:, fastIndex) = refAngles_rad;
timeFast(fastIndex) = (fastIndex-1) * Ts_fast;
fastIndex = fastIndex + 1;
end
end
angles_deg = rad2deg(stateHistory(1:3,:));
ref_deg = rad2deg(refHistory);
figure('Name','Pitch Axis');
subplot(2,1,1);
plot(timeFast, angles_deg(1,:), 'b', 'LineWidth', 1.5); hold on; plot(timeFast, ref_deg(1,:), 'k--', 'Li
xlabel('Time (s)'); ylabel('Pitch (deg)'); title('Pitch Angle vs. Time'); grid on; legend('Output','Setp
subplot(2,1,2);
plot(timeFast, controlHistory(1,:), 'b', 'LineWidth', 1.5); xlabel('Time (s)'); ylabel('Pitch Control');
figure('Name','Roll Axis');
subplot(2,1,1);
plot(timeFast, angles_deg(2,:), 'r', 'LineWidth', 1.5); hold on; plot(timeFast, ref_deg(2,:), 'k--', 'Li
xlabel('Time (s)'); ylabel('Roll (deg)'); title('Roll Angle vs. Time'); grid on; legend('Output','Setpoi
subplot(2,1,2);
plot(timeFast, controlHistory(2,:), 'r', 'LineWidth', 1.5); xlabel('Time (s)'); ylabel('Roll Control');
figure('Name','Yaw Axis');
subplot(2,1,1);
plot(timeFast, angles_deg(3,:), 'g', 'LineWidth', 1.5); hold on; plot(timeFast, ref_deg(3,:), 'k--', 'Li
xlabel('Time (s)'); ylabel('Yaw (deg)'); title('Yaw Angle vs. Time'); grid on; legend('Output','Setpoint
subplot(2,1,2);
plot(timeFast, controlHistory(3,:), 'g', 'LineWidth', 1.5); xlabel('Time (s)'); ylabel('Yaw Control'); t
```

```
function ref_deg = getReference(t)
if t < 0.2
ref_deg = [0; 0; 0];
elseif t < 0.4
ref_deg = [5; 0; 0];
elseif t < 0.6
ref_deg = [5; 5; 0];
elseif t < 0.8
ref_deg = [5; 5; 10];
else
ref_deg = [10; 5; 15];
end
end
function [A_lift, B_lift, C_lift] = buildLiftedModel(Af, Bf, Cf, N)
nx = size(Af,1);
nu = size(Bf,2);
A_lift = Af^N;
B_lift = zeros(nx, nu*N);
for i = 1:N
B_lift(:, (i-1)*nu+1:i*nu) = Af^(N-i) * Bf;
end
C_lift = Cf;
end
function [Phi, Gamma] = buildGPCmatrices(A_lift, B_lift, C_lift, P)
nx = size(A_lift,1);
ny = size(C_lift,1);
nu_lift = size(B_lift,2);
Phi = zeros(P*ny, nx);
Gamma = zeros(P*ny, P*nu_lift);
for i = 1:P
Phi((i-1)*ny+1:i*ny, :) = C_lift * (A_lift^i);
for j = 1:i
Gamma((i-1)*ny+1:i*ny, (j-1)*nu_lift+1:j*nu_lift) = C_lift * (A_lift^(i-j)) * B_lift;
end
end
end
```

A.8 BNO080.h

```
#ifndef _BNO080_H
#define _BNO080_H

#include "main.h"
////////////////////////////////////////////////////////////////////////

/**
* @brief Definition for connected to SPI2 (APB1 PCLK = 42MHz)
*/
#define BNO080_SPI_CHANNEL SPI2

#define BNO080_SPI_SCLK_PIN LL_GPIO_PIN_13
#define BNO080_SPI_SCLK_PORT GPIOB
```

```
#define BNO080_SPI_SCLK_CLK LL_AHB1_GRP1_PERIPH_GPIOB

#define BNO080_SPI_MISO_PIN LL_GPIO_PIN_14
#define BNO080_SPI_MISO_PORT GPIOB
#define BNO080_SPI_MISO_CLK LL_AHB1_GRP1_PERIPH_GPIOB

#define BNO080_SPI_MOSI_PIN LL_GPIO_PIN_15
#define BNO080_SPI_MOSI_PORT GPIOB
#define BNO080_SPI_MOSI_CLK LL_AHB1_GRP1_PERIPH_GPIOB

#define BNO080_SPI_CS_PIN LL_GPIO_PIN_12
#define BNO080_SPI_CS_PORT GPIOB
#define BNO080_SPI_CS_CLK LL_AHB1_GRP1_PERIPH_GPIOB

#define BNO080_PS0_WAKE_PIN LL_GPIO_PIN_8
#define BNO080_PS0_WAKE_PORT GPIOA
#define BNO080_PS0_WAKE_CLK LL_AHB1_GRP1_PERIPH_GPIOA

#define BNO080_RST_PIN LL_GPIO_PIN_1
#define BNO080_RST_PORT GPIOA
#define BNO080_RST_CLK LL_AHB1_GRP1_PERIPH_GPIOA

#define BNO080_INT_PIN LL_GPIO_PIN_12
#define BNO080_INT_PORT GPIOA
#define BNO080_INT_CLK LL_AHB1_GRP1_PERIPH_GPIOA

//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
#define CHIP_SELECT(BNO080) LL_GPIO_ResetOutputPin(BNO080_SPI_CS_PORT, BNO080_SPI_CS_PIN)
#define CHIP_DESELECT(BNO080) LL_GPIO_SetOutputPin(BNO080_SPI_CS_PORT, BNO080_SPI_CS_PIN)

#define WAKE_HIGH() LL_GPIO_SetOutputPin(BNO080_PS0_WAKE_PORT, BNO080_PS0_WAKE_PIN)
#define WAKE_LOW() LL_GPIO_ResetOutputPin(BNO080_PS0_WAKE_PORT, BNO080_PS0_WAKE_PIN)

#define RESET_HIGH() LL_GPIO_SetOutputPin(BNO080_RST_PORT, BNO080_RST_PIN)
#define RESET_LOW() LL_GPIO_ResetOutputPin(BNO080_RST_PORT, BNO080_RST_PIN)
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

//Registers
enum Registers
{
CHANNEL_COMMAND = 0,
CHANNEL_EXECUTABLE = 1,
CHANNEL_CONTROL = 2,
CHANNEL_REPORTS = 3,
CHANNEL_WAKE_REPORTS = 4,
CHANNEL_GYRO = 5
};

//All the ways we can configure or talk to the BNO080, figure 34, page 36 reference manual
//These are used for low level communication with the sensor, on channel 2
#define SHTP_REPORT_COMMAND_RESPONSE 0xF1
#define SHTP_REPORT_COMMAND_REQUEST 0xF2
```

```
#define SHTP_REPORT_FRS_READ_RESPONSE 0xF3
#define SHTP_REPORT_FRS_READ_REQUEST 0xF4
#define SHTP_REPORT_PRODUCT_ID_RESPONSE 0xF8
#define SHTP_REPORT_PRODUCT_ID_REQUEST 0xF9
#define SHTP_REPORT_BASE_TIMESTAMP 0xFB
#define SHTP_REPORT_SET_FEATURE_COMMAND 0xFD

//All the different sensors and features we can get reports from
//These are used when enabling a given sensor
#define SENSOR_REPORTID_ACCELEROMETER 0x01
#define SENSOR_REPORTID_GYROSCOPE 0x02
#define SENSOR_REPORTID_MAGNETIC_FIELD 0x03
#define SENSOR_REPORTID_LINEAR_ACCELERATION 0x04
#define SENSOR_REPORTID_ROTATION_VECTOR 0x05
#define SENSOR_REPORTID_GRAVITY 0x06
#define SENSOR_REPORTID_GAME_ROTATION_VECTOR 0x08
#define SENSOR_REPORTID_GEOMAGNETIC_ROTATION_VECTOR 0x09
#define SENSOR_REPORTID_TAP_DETECTOR 0x10
#define SENSOR_REPORTID_STEP_COUNTER 0x11
#define SENSOR_REPORTID_STABILITY_CLASSIFIER 0x13
#define SENSOR_REPORTID_PERSONAL_ACTIVITY_CLASSIFIER 0x1E

//Record IDs from figure 29, page 29 reference manual
//These are used to read the metadata for each sensor type
#define FRS_RECORDID_ACCELEROMETER 0xE302
#define FRS_RECORDID_GYROSCOPE_CALIBRATED 0xE306
#define FRS_RECORDID_MAGNETIC_FIELD_CALIBRATED 0xE309
#define FRS_RECORDID_ROTATION_VECTOR 0xE30B

//Command IDs from section 6.4, page 42
//These are used to calibrate, initialize, set orientation, tare etc the sensor
#define COMMAND_ERRORS 1
#define COMMAND_COUNTER 2
#define COMMAND_TARE 3
#define COMMAND_INITIALIZE 4
#define COMMAND_DCD 6
#define COMMAND_ME_CALIBRATE 7
#define COMMAND_DCD_PERIOD_SAVE 9
#define COMMAND_OSCILLATOR 10
#define COMMAND_CLEAR_DCD 11

#define CALIBRATE_ACCEL 0
#define CALIBRATE_GYRO 1
#define CALIBRATE_MAG 2
#define CALIBRATE_PLANAR_ACCEL 3
#define CALIBRATE_ACCEL_GYRO_MAG 4
#define CALIBRATE_STOP 5

#define MAX_PACKET_SIZE 128 //Packets can be up to 32k but we don't have that much RAM.
#define MAX_METADATA_SIZE 9 //This is in words. There can be many but we mostly only care about the firs

void BNO080_GPIO_SPI_Initialization(void);
int BNO080_Initialization(void);
unsigned char SPI2_SendByte(unsigned char data);
```

```
int BNO080_dataAvailable(void);
void BNO080_parseCommandReport(void);
void BNO080_parseInputReport(void);

float BNO080_getQuatI();
float BNO080_getQuatJ();
float BNO080_getQuatK();
float BNO080_getQuatReal();
float BNO080_getQuatRadianAccuracy();
uint8_t BNO080_getQuatAccuracy();
float BNO080_getAccelX();
float BNO080_getAccelY();
float BNO080_getAccelZ();
uint8_t BNO080_getAccelAccuracy();
float BNO080_getLinAccelX();
float BNO080_getLinAccelY();
float BNO080_getLinAccelZ();
uint8_t BNO080_getLinAccelAccuracy();
float BNO080_getGyroX();
float BNO080_getGyroY();
float BNO080_getGyroZ();
uint8_t BNO080_getGyroAccuracy();
float BNO080_getMagX();
float BNO080_getMagY();
float BNO080_getMagZ();
uint8_t BNO080_getMagAccuracy();
uint16_t BNO080_getStepCount();
uint8_t BNO080_getStabilityClassifier();
uint8_t BNO080_getActivityClassifier();
uint32_t BNO080_getTimeStamp();
int16_t BNO080_getQ1(uint16_t recordID);
int16_t BNO080_getQ2(uint16_t recordID);
int16_t BNO080_getQ3(uint16_t recordID);
float BNO080_getResolution(uint16_t recordID);
float BNO080_getRange(uint16_t recordID);

uint32_t BNO080_readFRSword(uint16_t recordID, uint8_t wordNumber);
void BNO080_frsReadRequest(uint16_t recordID, uint16_t readOffset, uint16_t blockSize);
int BNO080_readFRSdata(uint16_t recordID, uint8_t startLocation, uint8_t wordsToRead);
void BNO080_softReset(void);
uint8_t BNO080_resetReason();

float BNO080_qToFloat(int16_t fixedPointValue, uint8_t qPoint);

void BNO080_enableRotationVector(uint16_t timeBetweenReports);
void BNO080_enableGameRotationVector(uint16_t timeBetweenReports);
void BNO080_enableAccelerometer(uint16_t timeBetweenReports);
void BNO080_enableLinearAccelerometer(uint16_t timeBetweenReports);
void BNO080_enableGyro(uint16_t timeBetweenReports);
void BNO080_enableMagnetometer(uint16_t timeBetweenReports);
void BNO080_enableStepCounter(uint16_t timeBetweenReports);
void BNO080_enableStabilityClassifier(uint16_t timeBetweenReports);
```

```
void BNO080_calibrateAccelerometer();
void BNO080_calibrateGyro();
void BNO080_calibrateMagnetometer();
void BNO080_calibratePlanarAccelerometer();
void BNO080_calibrateAll();
void BNO080_endCalibration();
int BNO080_calibrationComplete();

void BNO080_setFeatureCommand(uint8_t reportID, uint32_t microsBetweenReports, uint32_t specificConfig);
void BNO080_sendCommand(uint8_t command);
void BNO080_sendCalibrateCommand(uint8_t thingToCalibrate);
void BNO080_requestCalibrationStatus();
void BNO080_saveCalibration();

int BNO080_waitForSPI(void);
int BNO080_receivePacket(void);
int BNO080_sendPacket(uint8_t channelNumber, uint8_t dataLength);

#endif
```

A.9 MPU6500.h

```
/**
* MPU6500.h
*
* Created on: Feb 7, 2025
* Author: SoulAndAnyaPC
*
* This header defines the register map, chip information, and function prototypes
* for the MPU-6500 MotionTracking device. All register addresses and related
* definitions are based on the MPU-6500 datasheet (Revision 1.3).
*/

#ifndef _MPU6500_H
#define _MPU6500_H
#include "main.h"
/*============================================================================
SPI & GPIO Pin Definitions (update these to match your board)
============================================================================*/
#define MPU6500_SPI_CHANNEL        SPI1

#define MPU6500_SPI_SCLK_PIN       LL_GPIO_PIN_5
#define MPU6500_SPI_SCLK_PORT      GPIOA
#define MPU6500_SPI_SCLK_CLK       LL_AHB1_GRP1_PERIPH_GPIOA

#define MPU6500_SPI_MISO_PIN       LL_GPIO_PIN_7
#define MPU6500_SPI_MISO_PORT      GPIOA
#define MPU6500_SPI_MISO_CLK       LL_AHB1_GRP1_PERIPH_GPIOA

#define MPU6500_SPI_MOSI_PIN       LL_GPIO_PIN_4
#define MPU6500_SPI_MOSI_PORT      GPIOB
#define MPU6500_SPI_MOSI_CLK       LL_AHB1_GRP1_PERIPH_GPIOB
```

```
#define MPU6500_SPI_CS_PIN         LL_GPIO_PIN_0
#define MPU6500_SPI_CS_PORT        GPIOB
#define MPU6500_SPI_CS_CLK         LL_AHB1_GRP1_PERIPH_GPIOB

#define MPU6500_INT_PIN            LL_GPIO_PIN_1
#define MPU6500_INT_PORT           GPIOB
#define MPU6500_INT_CLK            LL_AHB1_GRP1_PERIPH_GPIOB

/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
#define CHIP_SELECT(MPU6500) LL_GPIO_ResetOutputPin(MPU6500_SPI_CS_PORT, MPU6500_SPI_CS_PIN)
#define CHIP_DESELECT(MPU6500) LL_GPIO_SetOutputPin(MPU6500_SPI_CS_PORT, MPU6500_SPI_CS_PIN)
/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////

/**
* @brief MPU-6500 Register Map
*/

#define XG_OFFS_TC_H 0x04
#define XG_OFFS_TC_L 0x05
#define YG_OFFS_TC_H 0x07
#define YG_OFFS_TC_L 0x08
#define ZG_OFFS_TC_H 0x0A
#define ZG_OFFS_TC_L 0x0B
#define SELF_TEST_X_ACCEL 0x0D
#define SELF_TEST_Y_ACCEL 0x0E
#define SELF_TEST_Z_ACCEL 0x0F
#define XG_OFFS_USRH 0x13
#define XG_OFFS_USRL 0x14
#define YG_OFFS_USRH 0x15
#define YG_OFFS_USRL 0x16
#define ZG_OFFS_USRH 0x17
#define ZG_OFFS_USRL 0x18
#define SMPLRT_DIV 0x19
#define CONFIG 0x1A    //The default value of the register is 0x80.
#define GYRO_CONFIG 0x1B
#define ACCEL_CONFIG 0x1C
#define ACCEL_CONFIG2 0x1D
#define LP_MODE_CFG 0x1E
#define ACCEL_WOM_X_THR 0x20
#define ACCEL_WOM_Y_THR 0x21
#define ACCEL_WOM_Z_THR 0x22
#define FIFO_EN 0x23
#define FSYNC_INT 0x36
#define INT_PIN_CFG 0x37
#define INT_ENABLE 0x38
#define FIFO_WM_INT_STATUS 0x39

#define INT_STATUS 0x3A
#define ACCEL_XOUT_H 0x3B
#define ACCEL_XOUT_L 0x3C
#define ACCEL_YOUT_H 0x3D
```

```
#define ACCEL_YOUT_L 0x3E
#define ACCEL_ZOUT_H 0x3F
#define ACCEL_ZOUT_L 0x40
#define TEMP_OUT_H 0x41
#define TEMP_OUT_L 0x42
#define GYRO_XOUT_H 0x43
#define GYRO_XOUT_L 0x44
#define GYRO_YOUT_H 0x45
#define GYRO_YOUT_L 0x46
#define GYRO_ZOUT_H 0x47
#define GYRO_ZOUT_L 0x48
#define SELF_TEST_X_GYRO 0x50
#define SELF_TEST_Y_GYRO 0x51
#define SELF_TEST_Z_GYRO 0x52
#define FIFO_WM_TH1 0x60
#define FIFO_WM_TH2 0x61
#define SIGNAL_PATH_RESET 0x68
#define ACCEL_INTEL_CTRL 0x69
#define USER_CTRL 0x6A
#define PWR_MGMT_1 0x6B //The default value of the register is 0x41.
#define PWR_MGMT_2 0x6C
#define I2C_IF 0x70
#define FIFO_COUNTH 0x72
#define FIFO_COUNTL 0x73
#define FIFO_R_W 0x74
#define WHO_AM_I 0x75 //The default value of the register is 0x12.
#define WHO_AM_I_MPU6500_ANS 0x70
#define XA_OFFSET_H 0x77
#define XA_OFFSET_L 0x78
#define YA_OFFSET_H 0x7A
#define YA_OFFSET_L 0x7B
#define ZA_OFFSET_H 0x7D
#define ZA_OFFSET_L 0x7E


/**
* @brief MPU6500 structure definition.
*/

typedef struct _MPU6500{
short acc_x_raw;
short acc_y_raw;
short acc_z_raw;
short temperature_raw;
short gyro_x_raw;
short gyro_y_raw;
short gyro_z_raw;

float acc_x;
float acc_y;
float acc_z;
float gyro_x;
float gyro_y;
float gyro_z;
```

```
}Struct_MPU6500;

/**
* @brief MPU6500 structure definition.
*/

extern Struct_MPU6500 MPU6500;
extern int32_t gyro_x_offset, gyro_y_offset, gyro_z_offset;

/**
* @brief MPU6500 function prototype definition.
*/

void MPU6500_GPIO_SPI_Initialization(void);
int MPU6500_Initialization(void);
void MPU6500_Get6AxisRawData(short* accel, short* gyro);
void MPU6500_Get3AxisGyroRawData(short* gyro);
void MPU6500_Get3AxisAccRawData(short* accel);
int MPU6500_DataReady(void);

#endif
```

A.10 Quaternion.h

```
#ifndef _QUATERNION_H
#define _QUATERNION_H

#include "main.h"
#include <math.h>

extern float BNO080_Roll;
extern float BNO080_Pitch;
extern float BNO080_Yaw;

void Quaternion_Update(float* q);
float invSqrt(float x);

#endif
```

A.11 spi.h

```
/**
******************************************************************************
* @file    spi.h
* @brief   This file contains all the function prototypes for
*          the spi.c file
******************************************************************************
*/

#ifndef __SPI_H__
```

```
#define __SPI_H__

#ifdef __cplusplus
extern "C" {
#endif

#include "main.h"

void MX_SPI1_Init(void);
void MX_SPI2_Init(void);

#ifdef __cplusplus
}
#endif

#endif /* __SPI_H__ */
```

A.12 BNO080.c (Implementation)

```
#include "BNO080.h"
#include <math.h>


//Global Variables
uint8_t shtpHeader[4]; //Each packet has a header of 4 bytes
uint8_t shtpData[MAX_PACKET_SIZE];
uint8_t sequenceNumber[6] = {0, 0, 0, 0, 0, 0}; //There are 6 com channels. Each channel has its own seq
uint8_t commandSequenceNumber = 0; //Commands have a seqNum as well. These are inside command packet, th
uint32_t metaData[MAX_METADATA_SIZE]; //There is more than 10 words in a metadata record but we'll stop

//These are the raw sensor values pulled from the user requested Input Report
uint16_t rawAccelX, rawAccelY, rawAccelZ, accelAccuracy;
uint16_t rawLinAccelX, rawLinAccelY, rawLinAccelZ, accelLinAccuracy;
uint16_t rawGyroX, rawGyroY, rawGyroZ, gyroAccuracy;
uint16_t rawMagX, rawMagY, rawMagZ, magAccuracy;
uint16_t rawQuatI, rawQuatJ, rawQuatK, rawQuatReal, rawQuatRadianAccuracy, quatAccuracy;
uint16_t stepCount;
uint32_t timeStamp;
uint8_t stabilityClassifier;
uint8_t activityClassifier;
uint8_t *_activityConfidences; //Array that store the confidences of the 9 possible activities
uint8_t calibrationStatus;  //Byte R0 of ME Calibration Response

//These Q values are defined in the datasheet but can also be obtained by querying the meta data records
//See the read metadata example for more info
int16_t rotationVector_Q1 = 14;
int16_t accelerometer_Q1 = 8;
int16_t linear_accelerometer_Q1 = 8;
int16_t gyro_Q1 = 9;
int16_t magnetometer_Q1 = 4;


void BNO080_GPIO_SPI_Initialization(void)
```

```
{
LL_SPI_InitTypeDef SPI_InitStruct = {0};

LL_GPIO_InitTypeDef GPIO_InitStruct = {0};
/* Peripheral clock enable */
LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_SPI2);

LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);
LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOC);
LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
/**SPI2 GPIO Configuration
PB13   ------> SPI2_SCK
PB14   ------> SPI2_MISO
PB15   ------> SPI2_MOSI
*/
GPIO_InitStruct.Pin = LL_GPIO_PIN_13|LL_GPIO_PIN_14|LL_GPIO_PIN_15;
GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
GPIO_InitStruct.Alternate = LL_GPIO_AF_5;
LL_GPIO_Init(GPIOB, &GPIO_InitStruct);

SPI_InitStruct.TransferDirection = LL_SPI_FULL_DUPLEX;
SPI_InitStruct.Mode = LL_SPI_MODE_MASTER;
SPI_InitStruct.DataWidth = LL_SPI_DATAWIDTH_8BIT;
SPI_InitStruct.ClockPolarity = LL_SPI_POLARITY_HIGH;
SPI_InitStruct.ClockPhase = LL_SPI_PHASE_2EDGE;
SPI_InitStruct.NSS = LL_SPI_NSS_SOFT;
SPI_InitStruct.BaudRate = LL_SPI_BAUDRATEPRESCALER_DIV16;
SPI_InitStruct.BitOrder = LL_SPI_MSB_FIRST;
SPI_InitStruct.CRCCalculation = LL_SPI_CRCCALCULATION_DISABLE;
SPI_InitStruct.CRCPoly = 10;
LL_SPI_Init(BNO080_SPI_CHANNEL, &SPI_InitStruct);
LL_SPI_SetStandard(BNO080_SPI_CHANNEL, LL_SPI_PROTOCOL_MOTOROLA);

/**BNO080 GPIO Control Configuration
* PB12 ------> BNO080_CS (output)
* PA8  ------> BNO080_PS0/WAKE (output)
* PC9  ------> BNO080_RST (output)
* PC8  ------> BNO080_INT (input)
*/
/**/
LL_GPIO_ResetOutputPin(BNO080_RST_PORT, BNO080_RST_PIN);
LL_GPIO_ResetOutputPin(BNO080_SPI_CS_PORT, BNO080_SPI_CS_PIN);
LL_GPIO_ResetOutputPin(BNO080_PS0_WAKE_PORT, BNO080_PS0_WAKE_PIN);

/**/
GPIO_InitStruct.Pin = BNO080_SPI_CS_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
LL_GPIO_Init(BNO080_SPI_CS_PORT, &GPIO_InitStruct);
```

```
/**/
GPIO_InitStruct.Pin = BNO080_RST_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
LL_GPIO_Init(BNO080_RST_PORT, &GPIO_InitStruct);

/**/
GPIO_InitStruct.Pin = BNO080_PS0_WAKE_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
LL_GPIO_Init(BNO080_PS0_WAKE_PORT, &GPIO_InitStruct);

/**/
GPIO_InitStruct.Pin = BNO080_INT_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = LL_GPIO_PULL_UP;
LL_GPIO_Init(BNO080_INT_PORT, &GPIO_InitStruct);

LL_SPI_Enable(BNO080_SPI_CHANNEL);

CHIP_DESELECT(BNO080);
WAKE_HIGH();
RESET_HIGH();
}

int BNO080_Initialization(void)
{
BNO080_GPIO_SPI_Initialization();

printf("Checking BNO080...");

CHIP_DESELECT(BNO080);

//Configure the BNO080 for SPI communication
WAKE_HIGH(); //Before boot up the PS0/WAK pin must be high to enter SPI mode
RESET_LOW(); //Reset BNO080
HAL_Delay(200); //Min length not specified in datasheet?
RESET_HIGH(); //Bring out of reset

BNO080_waitForSPI(); //Wait until INT pin goes low.

//At system startup, the hub must send its full advertisement message (see 5.2 and 5.3) to the
//host. It must not send any other data until this step is complete.
//When BNO080 first boots it broadcasts big startup packet
//Read it and dump it
BNO080_waitForSPI(); //Wait for assertion of INT before reading advert message.
BNO080_receivePacket();

//The BNO080 will then transmit an unsolicited Initialize Response (see 6.4.5.2)
```

```
//Read it and dump it
BNO080_waitForSPI();  //Wait for assertion of INT before reading Init response
BNO080_receivePacket();

//Check communication with device
shtpData[0] = SHTP_REPORT_PRODUCT_ID_REQUEST; //Request the product ID and reset info
shtpData[1] = 0;  //Reserved

//Transmit packet on channel 2, 2 bytes
BNO080_sendPacket(CHANNEL_CONTROL, 2);

//Now we wait for response
BNO080_waitForSPI();
if (BNO080_receivePacket() == 1)
{
printf("header: %d %d %d %d\n", shtpHeader[0], shtpHeader[1], shtpHeader[2], shtpHeader[3]);
if (shtpData[0] == SHTP_REPORT_PRODUCT_ID_RESPONSE)
{
printf("BNO080 who_am_i = 0x%02x...ok\n\n", shtpData[0]);
return (0);
}// Sensor OK
}

printf("BNO080 Not OK: 0x%02x Should be 0x%02x\n", shtpData[0], SHTP_REPORT_PRODUCT_ID_RESPONSE);
return (1); //Something went wrong
}

unsigned char SPI2_SendByte(unsigned char data)
{
while(LL_SPI_IsActiveFlag_TXE(BNO080_SPI_CHANNEL)==RESET);
LL_SPI_TransmitData8(BNO080_SPI_CHANNEL, data);

while(LL_SPI_IsActiveFlag_RXNE(BNO080_SPI_CHANNEL)==RESET);
return LL_SPI_ReceiveData8(BNO080_SPI_CHANNEL);
}


//////////////////////////////////////////////////////////////////////////
//init
//////////////////////////////////////////////////////////////////////////

//Updates the latest variables if possible
//Returns false if new readings are not available
int BNO080_dataAvailable(void)
{
//If we have an interrupt pin connection available, check if data is available.
//If int pin is NULL, then we'll rely on BNO080_receivePacket() to timeout
//See issue 13: https://github.com/sparkfun/SparkFun_BNO080_Arduino_Library/issues/13
if (LL_GPIO_IsInputPinSet(BNO080_INT_PORT, BNO080_INT_PIN) == 1)
return (0);

if (BNO080_receivePacket() == 1)
{
//Check to see if this packet is a sensor reporting its data to us
```

```
if (shtpHeader[2] == CHANNEL_REPORTS && shtpData[0] == SHTP_REPORT_BASE_TIMESTAMP)
{
BNO080_parseInputReport(); //This will update the rawAccelX, etc variables depending on which feature re
return (1);
}
else if (shtpHeader[2] == CHANNEL_CONTROL)
{
BNO080_parseCommandReport(); //This will update responses to commands, calibrationStatus, etc.
return (1);
}
}
return (0);
}

//This function pulls the data from the command response report

//Unit responds with packet that contains the following:
//shtpHeader[0:3]: First, a 4 byte header
//shtpData[0]: The Report ID
//shtpData[1]: Sequence number (See 6.5.18.2)
//shtpData[2]: Command
//shtpData[3]: Command Sequence Number
//shtpData[4]: Response Sequence Number
//shtpData[5 + 0]: R0
//shtpData[5 + 1]: R1
//shtpData[5 + 2]: R2
//shtpData[5 + 3]: R3
//shtpData[5 + 4]: R4
//shtpData[5 + 5]: R5
//shtpData[5 + 6]: R6
//shtpData[5 + 7]: R7
//shtpData[5 + 8]: R8
void BNO080_parseCommandReport(void)
{
if (shtpData[0] == SHTP_REPORT_COMMAND_RESPONSE)
{
//The BNO080 responds with this report to command requests. It's up to use to remember which command we
uint8_t command = shtpData[2]; //This is the Command byte of the response

if (command == COMMAND_ME_CALIBRATE)
{
calibrationStatus = shtpData[5]; //R0 - Status (0 = success, non-zero = fail)
}
}
else
{
//This sensor report ID is unhandled.
//See reference manual to add additional feature reports as needed
}

//TODO additional feature reports may be strung together. Parse them all.
}

//This function pulls the data from the input report
```

```
//The input reports vary in length so this function stores the various 16-bit values as globals

//Unit responds with packet that contains the following:
//shtpHeader[0:3]: First, a 4 byte header
//shtpData[0:4]: Then a 5 byte timestamp of microsecond clicks since reading was taken
//shtpData[5 + 0]: Then a feature report ID (0x01 for Accel, 0x05 for Rotation Vector)
//shtpData[5 + 1]: Sequence number (See 6.5.18.2)
//shtpData[5 + 2]: Status
//shtpData[3]: Delay
//shtpData[4:5]: i/accel x/gyro x/etc
//shtpData[6:7]: j/accel y/gyro y/etc
//shtpData[8:9]: k/accel z/gyro z/etc
//shtpData[10:11]: real/gyro temp/etc
//shtpData[12:13]: Accuracy estimate
void BNO080_parseInputReport(void)
{
//Calculate the number of data bytes in this packet
int16_t dataLength = ((uint16_t)shtpHeader[1] << 8 | shtpHeader[0]);
dataLength &= ~(1 << 15); //Clear the MSbit. This bit indicates if this package is a continuation of the
//Ignore it for now. TODO catch this as an error and exit

dataLength -= 4; //Remove the header bytes from the data count

timeStamp = ((uint32_t)shtpData[4] << (8 * 3)) | (shtpData[3] << (8 * 2)) | (shtpData[2] << (8 * 1)) | (

uint8_t status = shtpData[7] & 0x03; //Get status bits
uint16_t data1 = (uint16_t)shtpData[10] << 8 | shtpData[9];
uint16_t data2 = (uint16_t)shtpData[12] << 8 | shtpData[11];
uint16_t data3 = (uint16_t)shtpData[14] << 8 | shtpData[13];
uint16_t data4 = 0;
uint16_t data5 = 0;

if (dataLength > 14)
{
data4 = (uint16_t)shtpData[16] << 8 | shtpData[15];
}
if (dataLength > 16)
{
data5 = (uint16_t)shtpData[18] << 8 | shtpData[17];
}

//Store these generic values to their proper global variable
switch(shtpData[5])
{
case SENSOR_REPORTID_ACCELEROMETER:
{
accelAccuracy = status;
rawAccelX = data1;
rawAccelY = data2;
rawAccelZ = data3;
break;
}
case SENSOR_REPORTID_LINEAR_ACCELERATION:
{
```

```
accelLinAccuracy = status;
rawLinAccelX = data1;
rawLinAccelY = data2;
rawLinAccelZ = data3;
break;
}
case SENSOR_REPORTID_GYROSCOPE:
{
gyroAccuracy = status;
rawGyroX = data1;
rawGyroY = data2;
rawGyroZ = data3;
break;
}
case SENSOR_REPORTID_MAGNETIC_FIELD:
{
magAccuracy = status;
rawMagX = data1;
rawMagY = data2;
rawMagZ = data3;
break;
}
case SENSOR_REPORTID_ROTATION_VECTOR:
case SENSOR_REPORTID_GAME_ROTATION_VECTOR:
{
quatAccuracy = status;
rawQuatI = data1;
rawQuatJ = data2;
rawQuatK = data3;
rawQuatReal = data4;
rawQuatRadianAccuracy = data5; //Only available on rotation vector, not game rot vector
break;
}
case SENSOR_REPORTID_STEP_COUNTER:
{
stepCount = data3; //Bytes 8/9
break;
}
case SENSOR_REPORTID_STABILITY_CLASSIFIER:
{
stabilityClassifier = shtpData[5 + 4]; //Byte 4 only
break;
}
case SENSOR_REPORTID_PERSONAL_ACTIVITY_CLASSIFIER:
{
activityClassifier = shtpData[5 + 5]; //Most likely state

//Load activity classification confidences into the array
for (uint8_t x = 0; x < 9; x++)    //Hardcoded to max of 9. TODO - bring in array size
_activityConfidences[x] = shtpData[11 + x]; //5 bytes of timestamp, byte 6 is first confidence byte
break;
}
case SHTP_REPORT_COMMAND_RESPONSE:
{
```

```
//printf("!");
//The BNO080 responds with this report to command requests. It's up to use to remember which command we
uint8_t command = shtpData[5 + 2]; //This is the Command byte of the response

if (command == COMMAND_ME_CALIBRATE)
{
//printf("ME Cal report found!");
calibrationStatus = shtpData[5 + 5]; //R0 - Status (0 = success, non-zero = fail)
}
break;
}
default:
{
//This sensor report ID is unhandled.
//See reference manual to add additional feature reports as needed
}
}

//TODO additional feature reports may be strung together. Parse them all.
}

//Return the rotation vector quaternion I
float BNO080_getQuatI()
{
return BNO080_qToFloat(rawQuatI, rotationVector_Q1);
}

//Return the rotation vector quaternion J
float BNO080_getQuatJ()
{
return BNO080_qToFloat(rawQuatJ, rotationVector_Q1);
}

//Return the rotation vector quaternion K
float BNO080_getQuatK()
{
return BNO080_qToFloat(rawQuatK, rotationVector_Q1);
}

//Return the rotation vector quaternion Real
float BNO080_getQuatReal()
{
return BNO080_qToFloat(rawQuatReal, rotationVector_Q1);
}

//Return the rotation vector accuracy
float BNO080_getQuatRadianAccuracy()
{
return BNO080_qToFloat(rawQuatRadianAccuracy, rotationVector_Q1);
}

//Return the acceleration component
uint8_t BNO080_getQuatAccuracy()
{
```

```
return (quatAccuracy);
}

//Return the acceleration component
float BNO080_getAccelX()
{
return BNO080_qToFloat(rawAccelX, accelerometer_Q1);
}

//Return the acceleration component
float BNO080_getAccelY()
{
return BNO080_qToFloat(rawAccelY, accelerometer_Q1);
}

//Return the acceleration component
float BNO080_getAccelZ()
{
return BNO080_qToFloat(rawAccelZ, accelerometer_Q1);
}

//Return the acceleration component
uint8_t BNO080_getAccelAccuracy()
{
return (accelAccuracy);
}

// linear acceleration, i.e. minus gravity

//Return the acceleration component
float BNO080_getLinAccelX()
{
return BNO080_qToFloat(rawLinAccelX, linear_accelerometer_Q1);
}

//Return the acceleration component
float BNO080_getLinAccelY()
{
return BNO080_qToFloat(rawLinAccelY, linear_accelerometer_Q1);
}

//Return the acceleration component
float BNO080_getLinAccelZ()
{
return BNO080_qToFloat(rawLinAccelZ, linear_accelerometer_Q1);
}

//Return the acceleration component
uint8_t BNO080_getLinAccelAccuracy()
{
return (accelLinAccuracy);
}

//Return the gyro component
```

```
float BNO080_getGyroX()
{
return BNO080_qToFloat(rawGyroX, gyro_Q1);
}

//Return the gyro component
float BNO080_getGyroY()
{
return BNO080_qToFloat(rawGyroY, gyro_Q1);
}

//Return the gyro component
float BNO080_getGyroZ()
{
return BNO080_qToFloat(rawGyroZ, gyro_Q1);
}

//Return the gyro component
uint8_t BNO080_getGyroAccuracy()
{
return (gyroAccuracy);
}

//Return the magnetometer component
float BNO080_getMagX()
{
return BNO080_qToFloat(rawMagX, magnetometer_Q1);
}

//Return the magnetometer component
float BNO080_getMagY()
{
return BNO080_qToFloat(rawMagY, magnetometer_Q1);
}

//Return the magnetometer component
float BNO080_getMagZ()
{
return BNO080_qToFloat(rawMagZ, magnetometer_Q1);
}

//Return the mag component
uint8_t BNO080_getMagAccuracy()
{
return (magAccuracy);
}

//Return the step count
uint16_t BNO080_getStepCount()
{
return (stepCount);
}

//Return the stability classifier
```

```
uint8_t BNO080_getStabilityClassifier()
{
return (stabilityClassifier);
}

//Return the activity classifier
uint8_t BNO080_getActivityClassifier()
{
return (activityClassifier);
}

//Return the time stamp
uint32_t BNO080_getTimeStamp()
{
return (timeStamp);
}

//Given a record ID, read the Q1 value from the metaData record in the FRS (ya, it's complicated)
//Q1 is used for all sensor data calculations
int16_t BNO080_getQ1(uint16_t recordID)
{
//Q1 is always the lower 16 bits of word 7
return BNO080_readFRSword(recordID, 7) & 0xFFFF; //Get word 7, lower 16 bits
}

//Given a record ID, read the Q2 value from the metaData record in the FRS
//Q2 is used in sensor bias
int16_t BNO080_getQ2(uint16_t recordID)
{
//Q2 is always the upper 16 bits of word 7
return BNO080_readFRSword(recordID, 7) >> 16; //Get word 7, upper 16 bits
}

//Given a record ID, read the Q3 value from the metaData record in the FRS
//Q3 is used in sensor change sensitivity
int16_t BNO080_getQ3(uint16_t recordID)
{
//Q3 is always the upper 16 bits of word 8
return BNO080_readFRSword(recordID, 8) >> 16; //Get word 8, upper 16 bits
}

//Given a record ID, read the resolution value from the metaData record in the FRS for a given sensor
float BNO080_getResolution(uint16_t recordID)
{
//The resolution Q value are 'the same as those used in the sensor's input report'
//This should be Q1.
int16_t Q = BNO080_getQ1(recordID);

//Resolution is always word 2
uint32_t value = BNO080_readFRSword(recordID, 2); //Get word 2

return BNO080_qToFloat(value, Q);
}
```

```
//Given a record ID, read the range value from the metaData record in the FRS for a given sensor
float BNO080_getRange(uint16_t recordID)
{
//The resolution Q value are 'the same as those used in the sensor's input report'
//This should be Q1.
int16_t Q = BNO080_getQ1(recordID);

//Range is always word 1
uint32_t value = BNO080_readFRSword(recordID, 1); //Get word 1

return BNO080_qToFloat(value, Q);
}

//Given a record ID and a word number, look up the word data
//Helpful for pulling out a Q value, range, etc.
//Use readFRSdata for pulling out multi-word objects for a sensor (Vendor data for example)
uint32_t BNO080_readFRSword(uint16_t recordID, uint8_t wordNumber)
{
if (BNO080_readFRSdata(recordID, wordNumber, 1) == 1) //Get word number, just one word in length from FR
return (metaData[0]);   //Return this one word

return (0); //Error
}

//Ask the sensor for data from the Flash Record System
//See 6.3.6 page 40, FRS Read Request
void BNO080_frsReadRequest(uint16_t recordID, uint16_t readOffset, uint16_t blockSize)
{
shtpData[0] = SHTP_REPORT_FRS_READ_REQUEST; //FRS Read Request
shtpData[1] = 0; //Reserved
shtpData[2] = (readOffset >> 0) & 0xFF; //Read Offset LSB
shtpData[3] = (readOffset >> 8) & 0xFF; //Read Offset MSB
shtpData[4] = (recordID >> 0) & 0xFF; //FRS Type LSB
shtpData[5] = (recordID >> 8) & 0xFF; //FRS Type MSB
shtpData[6] = (blockSize >> 0) & 0xFF; //Block size LSB
shtpData[7] = (blockSize >> 8) & 0xFF; //Block size MSB

//Transmit packet on channel 2, 8 bytes
BNO080_sendPacket(CHANNEL_CONTROL, 8);
}

//Given a sensor or record ID, and a given start/stop bytes, read the data from the Flash Record System
//Returns true if metaData array is loaded successfully
//Returns false if failure
int BNO080_readFRSdata(uint16_t recordID, uint8_t startLocation, uint8_t wordsToRead)
{
uint8_t spot = 0;

//First we send a Flash Record System (FRS) request
BNO080_frsReadRequest(recordID, startLocation, wordsToRead); //From startLocation of record, read a # of

//Read bytes until FRS reports that the read is complete
while (1)
{
```

```
//Now we wait for response
while (1)
{
uint8_t counter = 0;
while (BNO080_receivePacket() == 0)
{
if (counter++ > 100)
return (0); //Give up
HAL_Delay(1);
}

//We have the packet, inspect it for the right contents
//See page 40. Report ID should be 0xF3 and the FRS types should match the thing we requested
if (shtpData[0] == SHTP_REPORT_FRS_READ_RESPONSE)
if (((uint16_t)shtpData[13] << 8 | shtpData[12]) == recordID)
break; //This packet is one we are looking for
}

uint8_t dataLength = shtpData[1] >> 4;
uint8_t frsStatus = shtpData[1] & 0x0F;

uint32_t data0 = (uint32_t)shtpData[7] << 24 | (uint32_t)shtpData[6] << 16 | (uint32_t)shtpData[5] << 8
uint32_t data1 = (uint32_t)shtpData[11] << 24 | (uint32_t)shtpData[10] << 16 | (uint32_t)shtpData[9] <<

//Record these words to the metaData array
if (dataLength > 0)
{
metaData[spot++] = data0;
}
if (dataLength > 1)
{
metaData[spot++] = data1;
}

if (spot >= MAX_METADATA_SIZE)
{
printf("metaData array over run. Returning.");
return (1); //We have run out of space in our array. Bail.
}

if (frsStatus == 3 || frsStatus == 6 || frsStatus == 7)
{
return (1); //FRS status is read completed! We're done!
}
}
}

//Send command to reset IC
//Read all advertisement packets from sensor
//The sensor has been seen to reset twice if we attempt too much too quickly.
//This seems to work reliably.
void BNO080_softReset(void)
{
shtpData[0] = 1; //Reset
```

```
//Attempt to start communication with sensor
BNO080_sendPacket(CHANNEL_EXECUTABLE, 1); //Transmit packet on channel 1, 1 byte

//Read all incoming data and flush it
HAL_Delay(50);
while (BNO080_receivePacket() == 1);
HAL_Delay(50);
while (BNO080_receivePacket() == 1);
}

//Get the reason for the last reset
//1 = POR, 2 = Internal reset, 3 = Watchdog, 4 = External reset, 5 = Other
uint8_t BNO080_resetReason()
{
shtpData[0] = SHTP_REPORT_PRODUCT_ID_REQUEST; //Request the product ID and reset info
shtpData[1] = 0;   //Reserved

//Transmit packet on channel 2, 2 bytes
BNO080_sendPacket(CHANNEL_CONTROL, 2);

//Now we wait for response
if (BNO080_receivePacket() == 1)
{
if (shtpData[0] == SHTP_REPORT_PRODUCT_ID_RESPONSE)
{
return (shtpData[1]);
}
}

return (0);
}

//Given a register value and a Q point, convert to float
//See https://en.wikipedia.org/wiki/Q_(number_format)
float BNO080_qToFloat(int16_t fixedPointValue, uint8_t qPoint)
{
return fixedPointValue * powf(2, qPoint * -1);
}

//Sends the packet to enable the rotation vector
void BNO080_enableRotationVector(uint16_t timeBetweenReports)
{
BNO080_setFeatureCommand(SENSOR_REPORTID_ROTATION_VECTOR, timeBetweenReports, 0);
}

//Sends the packet to enable the rotation vector
void BNO080_enableGameRotationVector(uint16_t timeBetweenReports)
{
BNO080_setFeatureCommand(SENSOR_REPORTID_GAME_ROTATION_VECTOR, timeBetweenReports, 0);
}

//Sends the packet to enable the accelerometer
void BNO080_enableAccelerometer(uint16_t timeBetweenReports)
```

```
{
BNO080_setFeatureCommand(SENSOR_REPORTID_ACCELEROMETER, timeBetweenReports, 0);
}

//Sends the packet to enable the accelerometer
void BNO080_enableLinearAccelerometer(uint16_t timeBetweenReports)
{
BNO080_setFeatureCommand(SENSOR_REPORTID_LINEAR_ACCELERATION, timeBetweenReports, 0);
}

//Sends the packet to enable the gyro
void BNO080_enableGyro(uint16_t timeBetweenReports)
{
BNO080_setFeatureCommand(SENSOR_REPORTID_GYROSCOPE, timeBetweenReports, 0);
}

//Sends the packet to enable the magnetometer
void BNO080_enableMagnetometer(uint16_t timeBetweenReports)
{
BNO080_setFeatureCommand(SENSOR_REPORTID_MAGNETIC_FIELD, timeBetweenReports, 0);
}

//Sends the packet to enable the step counter
void BNO080_enableStepCounter(uint16_t timeBetweenReports)
{
BNO080_setFeatureCommand(SENSOR_REPORTID_STEP_COUNTER, timeBetweenReports, 0);
}

//Sends the packet to enable the Stability Classifier
void BNO080_enableStabilityClassifier(uint16_t timeBetweenReports)
{
BNO080_setFeatureCommand(SENSOR_REPORTID_STABILITY_CLASSIFIER, timeBetweenReports, 0);
}

//Sends the commands to begin calibration of the accelerometer
void BNO080_calibrateAccelerometer()
{
BNO080_sendCalibrateCommand(CALIBRATE_ACCEL);
}

//Sends the commands to begin calibration of the gyro
void BNO080_calibrateGyro()
{
BNO080_sendCalibrateCommand(CALIBRATE_GYRO);
}

//Sends the commands to begin calibration of the magnetometer
void BNO080_calibrateMagnetometer()
{
BNO080_sendCalibrateCommand(CALIBRATE_MAG);
}

//Sends the commands to begin calibration of the planar accelerometer
void BNO080_calibratePlanarAccelerometer()
```

```
{
BNO080_sendCalibrateCommand(CALIBRATE_PLANAR_ACCEL);
}

//See 2.2 of the Calibration Procedure document 1000-4044
void BNO080_calibrateAll()
{
BNO080_sendCalibrateCommand(CALIBRATE_ACCEL_GYRO_MAG);
}

void BNO080_endCalibration()
{
BNO080_sendCalibrateCommand(CALIBRATE_STOP); //Disables all calibrations
}

//See page 51 of reference manual - ME Calibration Response
//Byte 5 is parsed during the readPacket and stored in calibrationStatus
int BNO080_calibrationComplete()
{
if (calibrationStatus == 0)
return (1);
return (0);
}

//Given a sensor's report ID, this tells the BNO080 to begin reporting the values
//Also sets the specific config word. Useful for personal activity classifier
void BNO080_setFeatureCommand(uint8_t reportID, uint32_t microsBetweenReports, uint32_t specificConfig)
{
shtpData[0] = SHTP_REPORT_SET_FEATURE_COMMAND;  //Set feature command. Reference page 55
shtpData[1] = reportID;  //Feature Report ID. 0x01 = Accelerometer, 0x05 = Rotation vector
shtpData[2] = 0;  //Feature flags
shtpData[3] = 0;  //Change sensitivity (LSB)
shtpData[4] = 0;  //Change sensitivity (MSB)
shtpData[5] = (microsBetweenReports >> 0) & 0xFF;  //Report interval (LSB) in microseconds. 0x7A120 = 50
shtpData[6] = (microsBetweenReports >> 8) & 0xFF;  //Report interval
shtpData[7] = (microsBetweenReports >> 16) & 0xFF; //Report interval
shtpData[8] = (microsBetweenReports >> 24) & 0xFF; //Report interval (MSB)
shtpData[9] = 0;  //Batch Interval (LSB)
shtpData[10] = 0;  //Batch Interval
shtpData[11] = 0;  //Batch Interval
shtpData[12] = 0;  //Batch Interval (MSB)
shtpData[13] = (specificConfig >> 0) & 0xFF;      //Sensor-specific config (LSB)
shtpData[14] = (specificConfig >> 8) & 0xFF;      //Sensor-specific config
shtpData[15] = (specificConfig >> 16) & 0xFF;  //Sensor-specific config
shtpData[16] = (specificConfig >> 24) & 0xFF;  //Sensor-specific config (MSB)

//Transmit packet on channel 2, 17 bytes
BNO080_sendPacket(CHANNEL_CONTROL, 17);
}

//Tell the sensor to do a command
//See 6.3.8 page 41, Command request
//The caller is expected to set P0 through P8 prior to calling
void BNO080_sendCommand(uint8_t command)
```

```
{
shtpData[0] = SHTP_REPORT_COMMAND_REQUEST; //Command Request
shtpData[1] = commandSequenceNumber++;  //Increments automatically each function call
shtpData[2] = command;    //Command

//Caller must set these
/*shtpData[3] = 0; //P0
shtpData[4] = 0; //P1
shtpData[5] = 0; //P2
shtpData[6] = 0;
shtpData[7] = 0;
shtpData[8] = 0;
shtpData[9] = 0;
shtpData[10] = 0;
shtpData[11] = 0;*/

//Transmit packet on channel 2, 12 bytes
BNO080_sendPacket(CHANNEL_CONTROL, 12);
}

//This tells the BNO080 to begin calibrating
//See page 50 of reference manual and the 1000-4044 calibration doc
void BNO080_sendCalibrateCommand(uint8_t thingToCalibrate)
{
/*shtpData[3] = 0; //P0 - Accel Cal Enable
shtpData[4] = 0; //P1 - Gyro Cal Enable
shtpData[5] = 0; //P2 - Mag Cal Enable
shtpData[6] = 0; //P3 - Subcommand 0x00
shtpData[7] = 0; //P4 - Planar Accel Cal Enable
shtpData[8] = 0; //P5 - Reserved
shtpData[9] = 0; //P6 - Reserved
shtpData[10] = 0; //P7 - Reserved
shtpData[11] = 0; //P8 - Reserved*/

for (uint8_t x = 3; x < 12; x++) //Clear this section of the shtpData array
shtpData[x] = 0;

if (thingToCalibrate == CALIBRATE_ACCEL)
shtpData[3] = 1;
else if (thingToCalibrate == CALIBRATE_GYRO)
shtpData[4] = 1;
else if (thingToCalibrate == CALIBRATE_MAG)
shtpData[5] = 1;
else if (thingToCalibrate == CALIBRATE_PLANAR_ACCEL)
shtpData[7] = 1;
else if (thingToCalibrate == CALIBRATE_ACCEL_GYRO_MAG)
{
shtpData[3] = 1;
shtpData[4] = 1;
shtpData[5] = 1;
}
else if (thingToCalibrate == CALIBRATE_STOP)
; //Do nothing, bytes are set to zero
```

```
//Make the internal calStatus variable non-zero (operation failed) so that user can test while we wait
calibrationStatus = 1;

//Using this shtpData packet, send a command
BNO080_sendCommand(COMMAND_ME_CALIBRATE);
}

//Request ME Calibration Status from BNO080
//See page 51 of reference manual
void BNO080_requestCalibrationStatus()
{
/*shtpData[3] = 0; //P0 - Reserved
shtpData[4] = 0; //P1 - Reserved
shtpData[5] = 0; //P2 - Reserved
shtpData[6] = 0; //P3 - 0x01 - Subcommand: Get ME Calibration
shtpData[7] = 0; //P4 - Reserved
shtpData[8] = 0; //P5 - Reserved
shtpData[9] = 0; //P6 - Reserved
shtpData[10] = 0; //P7 - Reserved
shtpData[11] = 0; //P8 - Reserved*/

for (uint8_t x = 3; x < 12; x++) //Clear this section of the shtpData array
shtpData[x] = 0;

shtpData[6] = 0x01; //P3 - 0x01 - Subcommand: Get ME Calibration

//Using this shtpData packet, send a command
BNO080_sendCommand(COMMAND_ME_CALIBRATE);
}

//This tells the BNO080 to save the Dynamic Calibration Data (DCD) to flash
//See page 49 of reference manual and the 1000-4044 calibration doc
void BNO080_saveCalibration()
{
/*shtpData[3] = 0; //P0 - Reserved
shtpData[4] = 0; //P1 - Reserved
shtpData[5] = 0; //P2 - Reserved
shtpData[6] = 0; //P3 - Reserved
shtpData[7] = 0; //P4 - Reserved
shtpData[8] = 0; //P5 - Reserved
shtpData[9] = 0; //P6 - Reserved
shtpData[10] = 0; //P7 - Reserved
shtpData[11] = 0; //P8 - Reserved*/

for (uint8_t x = 3; x < 12; x++) //Clear this section of the shtpData array
shtpData[x] = 0;

//Using this shtpData packet, send a command
BNO080_sendCommand(COMMAND_DCD); //Save DCD command
}

//Blocking wait for BNO080 to assert (pull low) the INT pin
//indicating it's ready for comm. Can take more than 104ms
//after a hardware reset
```

```
int BNO080_waitForSPI(void)
{
for (uint32_t counter = 0; counter < 0xffffffff; counter++) //Don't got more than 255
{
if (LL_GPIO_IsInputPinSet(BNO080_INT_PORT, BNO080_INT_PIN) == 0)
{
//printf("\nData available\n");
return (1);
}
//printf("SPI Wait %d\n", counter);
}
printf("\nData not available\n");
return (0);
}


//Check to see if there is any new data available
//Read the contents of the incoming packet into the shtpData array
int BNO080_receivePacket(void)
{
uint8_t incoming;

if (LL_GPIO_IsInputPinSet(BNO080_INT_PORT, BNO080_INT_PIN) == 1)
return (0); //Data is not available

//Old way: if (BNO080_waitForSPI() == 0) return (0); //Something went wrong

//Get first four bytes to find out how much data we need to read

CHIP_SELECT(BNO080);

//Get the first four bytes, aka the packet header
uint8_t packetLSB = SPI2_SendByte(0);
uint8_t packetMSB = SPI2_SendByte(0);
uint8_t channelNumber = SPI2_SendByte(0);
uint8_t sequenceNumber = SPI2_SendByte(0); //Not sure if we need to store this or not

//Store the header info
shtpHeader[0] = packetLSB;
shtpHeader[1] = packetMSB;
shtpHeader[2] = channelNumber;
shtpHeader[3] = sequenceNumber;

//Calculate the number of data bytes in this packet
int16_t dataLength = ((uint16_t)packetMSB << 8 | packetLSB);
dataLength &= 0x7fff; //Clear the MSbit.
//This bit indicates if this package is a continuation of the last. Ignore it for now.
//TODO catch this as an error and exit
if (dataLength == 0)
{
//Packet is empty
return (0); //All done
}
dataLength -= 4; //Remove the header bytes from the data count
```

```
//printf("length: %d\n", dataLength);

//Read incoming data into the shtpData array
for (uint16_t dataSpot = 0; dataSpot < dataLength; dataSpot++)
{
incoming = SPI2_SendByte(0xFF);
//printf("%d ", incoming);
if (dataSpot < MAX_PACKET_SIZE) //BNO080 can respond with upto 270 bytes, avoid overflow
shtpData[dataSpot] = incoming; //Store data into the shtpData array
}
//printf("\n");

CHIP_DESELECT(BNO080); //Release BNO080
return (1); //We're done!
}


//Given the data packet, send the header then the data
//Returns false if sensor does not ACK
//TODO - Arduino has a max 32 byte send. Break sending into multi packets if needed.
int BNO080_sendPacket(uint8_t channelNumber, uint8_t dataLength)
{
uint8_t packetLength = dataLength + 4; //Add four bytes for the header

//Wait for BNO080 to indicate it is available for communication
if (BNO080_waitForSPI() == 0)
return (0); //Data is not available

//BNO080 has max CLK of 3MHz, MSB first,
//The BNO080 uses CPOL = 1 and CPHA = 1. This is mode3
CHIP_SELECT(BNO080);

//Send the 4 byte packet header
SPI2_SendByte(packetLength & 0xFF); //Packet length LSB
SPI2_SendByte(packetLength >> 8); //Packet length MSB
SPI2_SendByte(channelNumber); //Channel number
SPI2_SendByte(sequenceNumber[channelNumber]++);  //Send the sequence number, increments with each packet

//Send the user's data packet
for (uint8_t i = 0; i < dataLength; i++)
{
SPI2_SendByte(shtpData[i]);
}

CHIP_DESELECT(BNO080);

return (1);
}
//////////////////////////////////////////////////////////////////////////
```

A.13 main.c

```
/* USER CODE BEGIN Header */
/**
******************************************************************************
* @file           : main.c
* @brief          : Main program body
******************************************************************************
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "spi.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include "BNO080.h"
#include "Quaternion.h"
#include "MPU6500.h"
#include "CRSF.h"
#include "kalman.h"
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
int _write(int file, char* p, int len) {
for(int i = 0; i< len; i++) {
LL_USART_TransmitData8(USART1, *(p+i));
HAL_Delay(1);
}
return len;
}
/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */
extern uint8_t uart1_rx_flag;
extern uint8_t uart1_rx_data;

extern uint8_t crsf_rx_buf[32];
```

```
extern uint8_t crsf_rx_cplt_flag;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
/* USER CODE BEGIN PFP */
int Is_CRSF_Throttle_Min(void);
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
* @brief  The application entry point.
* @retval int
*/
int main(void)
{
/* USER CODE BEGIN 1 */
int count = 0;
float f = 1.234;
float q[4];
float quatRadianAccuracy;
uint32_t tim4_ccr4 = 12000;
/* USER CODE END 1 */

/* MCU Configuration--------------------------------------------------------*/

/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_TIM3_Init();
MX_USART1_UART_Init();
MX_SPI2_Init();
MX_SPI1_Init();
MX_USART2_UART_Init();
MX_TIM4_Init();
/* USER CODE BEGIN 2 */
//  HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
```

```
//  HAL_Delay(100);
LL_TIM_EnableCounter(TIM3);


LL_USART_EnableIT_RXNE(USART1);
LL_USART_EnableIT_RXNE(USART2);

//  printf("Checking BNO080...");

BNO080_Initialization();
BNO080_enableRotationVector(2500);

MPU6500_Initialization();

LL_TIM_EnableCounter(TIM4);
LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH1);
LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH2);
LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH3);
LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH4);

//  TIM4->CCR1 = 24000;
//  TIM4->CCR2 = 24000;
//  TIM4->CCR3 = 24000;
//  TIM4->CCR4 = 24000;
//  HAL_Delay(7000);
//
//  TIM4->CCR1 = 12000;
//  TIM4->CCR2 = 12000;
//  TIM4->CCR3 = 12000;
//  TIM4->CCR4 = 12000;
//  HAL_Delay(8000);

// while(Is_CRSF_Throttle_Min() == 0);

LL_TIM_CC_EnableChannel(TIM3, 1);

TIM3->PSC = 2000;
HAL_Delay(100);
TIM3->PSC = 1500;
HAL_Delay(100);
TIM3->PSC = 1000;
HAL_Delay(100);
LL_TIM_CC_DisableChannel(TIM3, 1);

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
//   HAL_Delay(1000);
```

```
//   if(uart1_rx_flag == 1) {
//   uart1_rx_flag = 0;
//   LL_USART_TransmitData8(USART1, uart1_rx_data);
//   switch(uart1_rx_data) {
//   case '0': LL_GPIO_TogglePin(GPIOC, GPIO_PIN_13); break;
//   case '1': LL_TIM_CC_EnableChannel(TIM3, 1); break;
//   case '2': LL_TIM_CC_DisableChannel(TIM3, 1); break;
//   }
//   }

//printf("%d %f\r\n", count+1, f+=0.001);
//HAL_Delay(1000);
//  LL_USART_TransmitData8(USART1, 'A');
//   printf("Checking BNO080...\r\n");
//   if(BNO080_dataAvailable() == 1)
//   {
//   LL_GPIO_TogglePin(GPIOC, LL_GPIO_PIN_13);
//
//   q[0] = BNO080_getQuatI();
//   q[1] = BNO080_getQuatJ();
//   q[2] = BNO080_getQuatK();
//   q[3] = BNO080_getQuatReal();
//   quatRadianAccuracy = BNO080_getQuatRadianAccuracy();
//
//   Quaternion_Update(&q[0]);
//
//   printf("%d,%d,%d\n", (int)(BNO080_Roll*100), (int)(BNO080_Pitch*100), (int)(BNO080_Yaw*100));
//   }
if(MPU6500_DataReady() == 1) {
LL_GPIO_TogglePin(GPIOC, LL_GPIO_PIN_13);
// LL_GPIO_TogglePin(GPIOC, LL_GPIO_PIN_14);

MPU6500_Get3AxisGyroRawData(&MPU6500.gyro_x_raw);

MPU6500.gyro_x = MPU6500.gyro_x_raw * 2000.f / 32768.f;
MPU6500.gyro_y = MPU6500.gyro_y_raw * 2000.f / 32768.f;
MPU6500.gyro_z = MPU6500.gyro_z_raw * 2000.f / 32768.f;

float filtered_gyro_x = KALMAN(MPU6500.gyro_x_raw*100);
float filtered_gyro_y = KALMAN(MPU6500.gyro_y_raw*100);
float filtered_gyro_z = KALMAN(MPU6500.gyro_z_raw*100);
//   printf("%d,%d,%d,\n", (int)MPU6500.gyro_x_raw*100, (int)MPU6500.gyro_y_raw*100, (int)MPU6500.gyro_z
//   printf("%d,%d,%d,%d,%d,%d,\n", (int)MPU6500.gyro_x_raw*100, (int)filtered_gyro_x,
//   (int)MPU6500.gyro_y_raw*100,
//   (int)filtered_gyro_y,
//   (int)MPU6500.gyro_z_raw*100,
//   (int)filtered_gyro_z
//   );
printf("%d,%d\n", (int)MPU6500.gyro_x_raw*100, (int)filtered_gyro_x);
}
if(crsf_rx_cplt_flag == 1) {
crsf_rx_cplt_flag = 0;
uint8_t packet_length = crsf_rx_buf[1] + 2;
if(verify_crc(crsf_rx_buf, packet_length)) {
```

```
LL_GPIO_TogglePin(GPIOC, LL_GPIO_PIN_13);
ELRS_CRSF_Parse(&crsf_rx_buf[3], &CRSF);
// uint16_t *channels = (uint16_t *)&CRSF;
//
// for (int i = 0; i < 16; i++) {
//  printf("%d\t ", channels[i]);
// }
// printf("\r\n");
//   HAL_Delay(30);

}
//   if(isFailSafe()) {
//    LL_TIM_CC_EnableChannel(TIM3, 1);
//    } else {
//    LL_TIM_CC_DisableChannel(TIM3, 1);
//    }
}
int32_t ch2Val = (CRSF.channel2 - 191)*12;
//printf("%d\t ", ch2Val);
TIM4->CCR1 = 12000 + ch2Val;
TIM4->CCR2 = 12000 + ch2Val;
TIM4->CCR3 = 12000 + ch2Val;
TIM4->CCR4 = 12000 + ch2Val;
//   tim4_ccr4 += 10;
//   if(tim4_ccr4 >= 24000) {
//   tim4_ccr4 = 12000;
//   }
//   TIM4->CCR4 = tim4_ccr4;
//   HAL_Delay(1);
}
/* USER CODE END 3 */
}

/**
* @brief System Clock Configuration
* @retval None
*/
void SystemClock_Config(void)
{
RCC_OscInitTypeDef RCC_OscInitStruct = {0};
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

/** Configure the main internal regulator output voltage
*/
__HAL_RCC_PWR_CLK_ENABLE();
__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

/** Initializes the RCC Oscillators according to the specified parameters
* in the RCC_OscInitTypeDef structure.
*/
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
RCC_OscInitStruct.HSEState = RCC_HSE_ON;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
```

```
RCC_OscInitStruct.PLL.PLLM = 25;
RCC_OscInitStruct.PLL.PLLN = 192;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
RCC_OscInitStruct.PLL.PLLQ = 4;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
Error_Handler();
}

/** Initializes the CPU, AHB and APB buses clocks
*/
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
|RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_3) != HAL_OK)
{
Error_Handler();
}
}

/* USER CODE BEGIN 4 */

int Is_CRSF_Throttle_Min(void) {
if(crsf_rx_cplt_flag == 1) {
crsf_rx_cplt_flag = 0;
uint8_t packet_length = crsf_rx_buf[1] + 2;
if(verify_crc(crsf_rx_buf, packet_length)) {
ELRS_CRSF_Parse(&crsf_rx_buf[3], &CRSF);
if(CRSF.channel2 < 200) return 1;
}
}
return 0;
}

/* USER CODE END 4 */

/**
* @brief  This function is executed in case of error occurrence.
* @retval None
*/
void Error_Handler(void)
{
/* USER CODE BEGIN Error_Handler_Debug */
/* User can add his own implementation to report the HAL error return state */
__disable_irq();
while (1)
{
}
/* USER CODE END Error_Handler_Debug */
}
```

```
#ifdef  USE_FULL_ASSERT
/**
* @brief  Reports the name of the source file and the source line number
*         where the assert_param error has occurred.
* @param  file: pointer to the source file name
* @param  line: assert_param error line source number
* @retval None
*/
void assert_failed(uint8_t *file, uint32_t line)
{
/* USER CODE BEGIN 6 */
/* User can add his own implementation to report the file name and line number,
ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
/* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

A.14 MPU6500.c

```
/**
* @brief MPU6500 structure definition.
*/

#include "MPU6500.h"

Struct_MPU6500 MPU6500;
int32_t gyro_x_offset, gyro_y_offset, gyro_z_offset; // To remove offset

void MPU6500_GPIO_SPI_Initialization(void) {
LL_SPI_InitTypeDef SPI_InitStruct = {0};

LL_GPIO_InitTypeDef GPIO_InitStruct = {0};
/* Peripheral clock enable */
LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SPI1);

LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);
/**SPI1 GPIO Configuration
PA5   ------> SPI1_SCK
PA6   ------> SPI1_MISO
PA7   ------> SPI1_MOSI
*/
GPIO_InitStruct.Pin = LL_GPIO_PIN_5 | LL_GPIO_PIN_6 | LL_GPIO_PIN_7;
GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
GPIO_InitStruct.Alternate = LL_GPIO_AF_5;
LL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

```
SPI_InitStruct.TransferDirection = LL_SPI_FULL_DUPLEX;
SPI_InitStruct.Mode = LL_SPI_MODE_MASTER;
SPI_InitStruct.DataWidth = LL_SPI_DATAWIDTH_8BIT;
SPI_InitStruct.ClockPolarity = LL_SPI_POLARITY_HIGH;
SPI_InitStruct.ClockPhase = LL_SPI_PHASE_2EDGE;
SPI_InitStruct.NSS = LL_SPI_NSS_SOFT;
SPI_InitStruct.BaudRate =
LL_SPI_BAUDRATEPRESCALER_DIV8; // MPU6500 MAX SPI CLK is 10MHz. But
// DIV2(42MHz) is available.
SPI_InitStruct.BitOrder = LL_SPI_MSB_FIRST;
SPI_InitStruct.CRCCalculation = LL_SPI_CRCCALCULATION_DISABLE;
SPI_InitStruct.CRCPoly = 10;
LL_SPI_Init(MPU6500_SPI_CHANNEL, &SPI_InitStruct);
LL_SPI_SetStandard(MPU6500_SPI_CHANNEL, LL_SPI_PROTOCOL_MOTOROLA);

/**MPU6500 GPIO Control Configuration
* PC4  ------> MPU6500_SPI_CS_PIN (output)
* PC5  ------> MPU6500_INT_PIN (input)
*/
/**/
LL_GPIO_ResetOutputPin(MPU6500_SPI_CS_PORT, MPU6500_SPI_CS_PIN);

/**/
GPIO_InitStruct.Pin = MPU6500_SPI_CS_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
LL_GPIO_Init(MPU6500_SPI_CS_PORT, &GPIO_InitStruct);

/**/
GPIO_InitStruct.Pin = MPU6500_INT_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = LL_GPIO_PULL_UP;
LL_GPIO_Init(MPU6500_INT_PORT, &GPIO_InitStruct);

LL_SPI_Enable(MPU6500_SPI_CHANNEL);

CHIP_DESELECT(MPU6500);
}

unsigned char SPI1_SendByte(unsigned char data) {
while (LL_SPI_IsActiveFlag_TXE(MPU6500_SPI_CHANNEL) == RESET)
;
LL_SPI_TransmitData8(MPU6500_SPI_CHANNEL, data);

while (LL_SPI_IsActiveFlag_RXNE(MPU6500_SPI_CHANNEL) == RESET)
;
return LL_SPI_ReceiveData8(MPU6500_SPI_CHANNEL);
}

////////////////////////////////////////////////////////////

uint8_t MPU6500_Readbyte(uint8_t reg_addr) {
```

```
uint8_t val;

CHIP_SELECT(MPU6500);
SPI1_SendByte(reg_addr | 0x80); // Register. MSB 1 is read instruction.
val = SPI1_SendByte(0x00);      // Send DUMMY to read data
CHIP_DESELECT(MPU6500);

return val;
}

void MPU6500_Readbytes(unsigned char reg_addr, unsigned char len,
unsigned char *data) {
unsigned int i = 0;

CHIP_SELECT(MPU6500);
SPI1_SendByte(reg_addr | 0x80); // Register. MSB 1 is read instruction.
while (i < len) {
data[i++] = SPI1_SendByte(0x00); // Send DUMMY to read data
}
CHIP_DESELECT(MPU6500);
}

void MPU6500_Writebyte(uint8_t reg_addr, uint8_t val) {
CHIP_SELECT(MPU6500);
SPI1_SendByte(reg_addr & 0x7F); // Register. MSB 0 is write instruction.
SPI1_SendByte(val);             // Send Data to write
CHIP_DESELECT(MPU6500);
}

void MPU6500_Writebytes(unsigned char reg_addr, unsigned char len,
unsigned char *data) {
unsigned int i = 0;
CHIP_SELECT(MPU6500);
SPI1_SendByte(reg_addr & 0x7F); // Register. MSB 0 is write instruction.
while (i < len) {
SPI1_SendByte(data[i++]); // Send Data to write
}
CHIP_DESELECT(MPU6500);
}

int MPU6500_Initialization(void) {

uint8_t who_am_i = 0;
int16_t accel_raw_data[3] = {0}; // To remove offset
int16_t gyro_raw_data[3] = {0};  // To remove offset

MPU6500_GPIO_SPI_Initialization();

printf("Checking MPU6500...");

// check WHO_AM_I (0x75)
who_am_i = MPU6500_Readbyte(WHO_AM_I);

// who am i = WHO_AM_I_MPU6500_ANS
```

```
if (who_am_i == WHO_AM_I_MPU6500_ANS) {
printf("\nMPU6500 who_am_i = 0x%02x...OK\n\n", who_am_i);
}
// recheck
else if (who_am_i != WHO_AM_I_MPU6500_ANS) {
who_am_i = MPU6500_Readbyte(WHO_AM_I); // check again WHO_AM_I (0x75)

if (who_am_i != WHO_AM_I_MPU6500_ANS) {
printf("MPU6500 Not OK: 0x%02x Should be 0x%02x\n", who_am_i,
WHO_AM_I_MPU6500_ANS);
return 1; // ERROR
}
}

MPU6500_Writebyte(PWR_MGMT_1, 0x80);
HAL_Delay(50);

MPU6500_Writebyte(
PWR_MGMT_1,
0x01);
HAL_Delay(50);

// PWR_MGMT_2 0x6C
MPU6500_Writebyte(PWR_MGMT_2,
0x38);

HAL_Delay(50);

MPU6500_Writebyte(SMPLRT_DIV, 0x00);
HAL_Delay(50);

MPU6500_Writebyte(CONFIG,
0x05);
HAL_Delay(50);

MPU6500_Writebyte(
GYRO_CONFIG,
0x18);

HAL_Delay(50);

MPU6500_Writebyte(ACCEL_CONFIG, 0x18);
HAL_Delay(50);

MPU6500_Writebyte(
ACCEL_CONFIG2,
0x03);
HAL_Delay(50);

MPU6500_Writebyte(INT_ENABLE, 0x01); // Enable DRDY Interrupt
HAL_Delay(50);

// printf("gyro bias: %d %d %d\n", gyro_x_offset, gyro_y_offset,
// gyro_z_offset);
```

```
// Remove Gyro X offset
// MPU6500_Writebyte( XG_OFFS_USRH, offset_x>>8 ); // gyro x offset high
// byte  MPU6500_Writebyte( XG_OFFS_USRL, offset_x ); // gyro x offset
// low byte
//
// // Remove Gyro Y offset
// MPU6500_Writebyte( YG_OFFS_USRH, offset_y>>8 ); // gyro y offset high
// byte  MPU6500_Writebyte( YG_OFFS_USRL, offset_y ); // gyro y offset
// low byte
//
// // Remove Gyro Z offset
// MPU6500_Writebyte( ZG_OFFS_USRH, offset_z>>8 ); // gyro z offset high
// byte  MPU6500_Writebyte( ZG_OFFS_USRL, offset_z ); // gyro z offset
// low byte

return 0; // OK
}

void MPU6500_Get6AxisRawData(short *accel, short *gyro) {
unsigned char data[14];
MPU6500_Readbytes(ACCEL_XOUT_H, 14, data);

accel[0] = (data[0] << 8) | data[1];
accel[1] = (data[2] << 8) | data[3];
accel[2] = (data[4] << 8) | data[5];

gyro[0] = ((data[8] << 8) | data[9]);
gyro[1] = ((data[10] << 8) | data[11]);
gyro[2] = ((data[12] << 8) | data[13]);
}

void MPU6500_Get3AxisGyroRawData(short *gyro) {
unsigned char data[6];
MPU6500_Readbytes(GYRO_XOUT_H, 6, data);

gyro[0] = ((data[0] << 8) | data[1]);
gyro[1] = ((data[2] << 8) | data[3]);
gyro[2] = ((data[4] << 8) | data[5]);
}

void MPU6500_Get3AxisAccRawData(short *accel) {
unsigned char data[6];
MPU6500_Readbytes(ACCEL_XOUT_H, 6, data);

accel[0] = ((data[0] << 8) | data[1]);
accel[1] = ((data[2] << 8) | data[3]);
accel[2] = ((data[4] << 8) | data[5]);
}

int MPU6500_DataReady(void) {
return LL_GPIO_IsInputPinSet(MPU6500_INT_PORT, MPU6500_INT_PIN);
}
```

A.15 stm32f4xx_it.c

```
/* USER CODE BEGIN Header */
/**
******************************************************************************
* @file    stm32f4xx_it.c
* @brief   Interrupt Service Routines.
******************************************************************************
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "stm32f4xx_it.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */


/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN TD */

/* USER CODE END TD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */
#define CRSF_SYNC_BYTE             0xC8  // CRSF sync; note: OpenTX/EdgeTX might send 0xEE
#define CRSF_FRAME_SIZE    0x18
#define CRSF_PACKET_RC_CHANNELS    0x16  // RC channels packed packet type

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */

uint8_t uart1_rx_flag = 0;
uint8_t uart1_rx_data = 0;

uint8_t uart2_rx_flag = 0;
uint8_t uart2_rx_data = 0;

uint8_t crsf_rx_buf[32];
uint8_t crsf_rx_cplt_flag = 0;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
```

```
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/* External variables --------------------------------------------------------*/

/* USER CODE BEGIN EV */

/* USER CODE END EV */

/******************************************************************************/
/*           Cortex-M4 Processor Interruption and Exception Handlers          */
/******************************************************************************/
/**
* @brief This function handles Non maskable interrupt.
*/
void NMI_Handler(void)
{
/* USER CODE BEGIN NonMaskableInt_IRQn 0 */

/* USER CODE END NonMaskableInt_IRQn 0 */
/* USER CODE BEGIN NonMaskableInt_IRQn 1 */
while (1)
{
}
/* USER CODE END NonMaskableInt_IRQn 1 */
}

/**
* @brief This function handles Hard fault interrupt.
*/
void HardFault_Handler(void)
{
/* USER CODE BEGIN HardFault_IRQn 0 */

/* USER CODE END HardFault_IRQn 0 */
while (1)
{
/* USER CODE BEGIN W1_HardFault_IRQn 0 */
/* USER CODE END W1_HardFault_IRQn 0 */
}
}

/**
* @brief This function handles Memory management fault.
*/
void MemManage_Handler(void)
{
/* USER CODE BEGIN MemoryManagement_IRQn 0 */
```

```
/* USER CODE END MemoryManagement_IRQn 0 */
while (1)
{
/* USER CODE BEGIN W1_MemoryManagement_IRQn 0 */
/* USER CODE END W1_MemoryManagement_IRQn 0 */
}
}

/**
* @brief This function handles Pre-fetch fault, memory access fault.
*/
void BusFault_Handler(void)
{
/* USER CODE BEGIN BusFault_IRQn 0 */

/* USER CODE END BusFault_IRQn 0 */
while (1)
{
/* USER CODE BEGIN W1_BusFault_IRQn 0 */
/* USER CODE END W1_BusFault_IRQn 0 */
}
}

/**
* @brief This function handles Undefined instruction or illegal state.
*/
void UsageFault_Handler(void)
{
/* USER CODE BEGIN UsageFault_IRQn 0 */

/* USER CODE END UsageFault_IRQn 0 */
while (1)
{
/* USER CODE BEGIN W1_UsageFault_IRQn 0 */
/* USER CODE END W1_UsageFault_IRQn 0 */
}
}

/**
* @brief This function handles System service call via SWI instruction.
*/
void SVC_Handler(void)
{
/* USER CODE BEGIN SVCall_IRQn 0 */

/* USER CODE END SVCall_IRQn 0 */
/* USER CODE BEGIN SVCall_IRQn 1 */

/* USER CODE END SVCall_IRQn 1 */
}

/**
* @brief This function handles Debug monitor.
```

```
*/
void DebugMon_Handler(void)
{
/* USER CODE BEGIN DebugMonitor_IRQn 0 */

/* USER CODE END DebugMonitor_IRQn 0 */
/* USER CODE BEGIN DebugMonitor_IRQn 1 */

/* USER CODE END DebugMonitor_IRQn 1 */
}

/**
* @brief This function handles Pendable request for system service.
*/
void PendSV_Handler(void)
{
/* USER CODE BEGIN PendSV_IRQn 0 */

/* USER CODE END PendSV_IRQn 0 */
/* USER CODE BEGIN PendSV_IRQn 1 */

/* USER CODE END PendSV_IRQn 1 */
}

/**
* @brief This function handles System tick timer.
*/
void SysTick_Handler(void)
{
/* USER CODE BEGIN SysTick_IRQn 0 */

/* USER CODE END SysTick_IRQn 0 */
HAL_IncTick();
/* USER CODE BEGIN SysTick_IRQn 1 */

/* USER CODE END SysTick_IRQn 1 */
}

/******************************************************************************/
/* STM32F4xx Peripheral Interrupt Handlers                                    */
/* Add here the Interrupt Handlers for the used peripherals.                  */
/* For the available peripheral interrupt handler names,                      */
/* please refer to the startup file (startup_stm32f4xx.s).                    */
/******************************************************************************/

/**
* @brief This function handles USART1 global interrupt.
*/
void USART1_IRQHandler(void)
{
/* USER CODE BEGIN USART1_IRQn 0 */
if(LL_USART_IsActiveFlag_RXNE(USART1)) {
LL_USART_ClearFlag_RXNE(USART1);
uart1_rx_data = LL_USART_ReceiveData8(USART1);
```

```
uart1_rx_flag = 1;
}
/* USER CODE END USART1_IRQn 0 */
/* USER CODE BEGIN USART1_IRQn 1 */

/* USER CODE END USART1_IRQn 1 */
}

/**
* @brief This function handles USART2 global interrupt.
*/
void USART2_IRQHandler(void)
{
/* USER CODE BEGIN USART2_IRQn 0 */
static unsigned char cnt = 0;
static uint8_t expected_length = 0;

if(LL_USART_IsActiveFlag_RXNE(USART2)) {
LL_USART_ClearFlag_RXNE(USART2);
uart2_rx_data = LL_USART_ReceiveData8(USART2);
uart2_rx_flag = 1;
// while(!LL_USART_IsActiveFlag_TXE(USART1));
// LL_USART_TransmitData8(USART1, uart2_rx_data);
// while(!LL_USART_IsActiveFlag_TXE(USART1));
// LL_USART_TransmitData8(USART1, uart2_rx_data);
switch(cnt) {
case 0:
if(uart2_rx_data == CRSF_SYNC_BYTE) {
crsf_rx_buf[cnt] = uart2_rx_data;
cnt++;
}
break;
case 1:
if(uart2_rx_data == CRSF_FRAME_SIZE) {
crsf_rx_buf[cnt] = uart2_rx_data;
expected_length = uart2_rx_data + 2;
cnt++;
} else {
cnt = 0; // invalid
}
break;
case 2:
if(uart2_rx_data == CRSF_PACKET_RC_CHANNELS) {
crsf_rx_buf[cnt] = uart2_rx_data;
cnt++;
} else {
cnt = 0; // invalid
}
break;
default:
if((expected_length != 0) && (cnt == expected_length)) {
// crsf_rx_buf[cnt] = uart2_rx_data;
cnt++;
//if(verify_crc(crsf_rx_buf, cnt)) {
```

```
crsf_rx_cplt_flag = 1;
//   }
cnt = 0;
expected_length = 0;
} else {
crsf_rx_buf[cnt] = uart2_rx_data;
cnt++;
}
break;
}

//        if(cnt >= sizeof(crsf_rx_buf)) {
//            cnt = 0;
//            expected_length = 0;
//        }



// while(!LL_USART_IsActiveFlag_TXE(USART1));
// LL_USART_TransmitData8(USART1, uart2_rx_data);
}
/* USER CODE END USART2_IRQn 0 */
/* USER CODE BEGIN USART2_IRQn 1 */

/* USER CODE END USART2_IRQn 1 */
}

/* USER CODE BEGIN 1 */

/* USER CODE END 1 */
```