UNIVERSITY OF MESSINA

DEPARTMENT OF ENGINEERING **Master's Degree in Engineering and Computer Science**

EMBEDDED SYSTEMS PROJECT

ASSIGNMENT:

Acquisition of sensors data from X-NUCLEO IKS4A1 expansion board with STM32H723

STUDENT:
Longo Andrea
mat. 563714

ACADEMIC YEAR 2024-2025

Contents

Chapter 1

Introduction

The goal of this project is to acquire and process data from three types of inertial sensors: an accelerometer, a gyroscope, and a magnetometer. Each sensor provides distinct but complementary information: linear acceleration, angular velocity, and magnetic field direction, respectively. These raw sensor readings are collected, synchronized, and then passed through a sensor fusion algorithm. This algorithm integrates the data to produce an accurate and stable estimate of the system's orientation in three-dimensional space, represented as yaw, pitch, and roll. The fusion process enhances robustness against noise and drift in individual sensors, enabling reliable real-time orientation tracking.

1.1 IMU and Fusion

Inertial Measurement Units (IMUs) are electronic devices that combine accelerometer, gyroscope and magnetometer data to estimate orientation in 3D space. They are commonly used in embedded systems for tasks such as motion tracking, orientation estimation, and navigation. Each individual sensor has its own limitations:

- **Accelerometer:** Measures linear acceleration along the three axes: x, y and z. It is useful for detecting changes in velocity relative to gravity but can produce noisy readings.
- **Gyroscope:** Measures angular velocity around the three

axes. It helps in tracking rotational motion and estimating orientation. The gyroscope is affected by bias and bias instability, which, when integrated over time, lead to drift, a cumulative orientation error that grows even in the absence of real motion.

- **Magnetometer:** Measures the surrounding magnetic field and acts as a compass to determine orientation relative to Earth's magnetic north. However, it can be affected by magnetic interference from nearby electronic components.

1.2 Library and Tools

For developing this project a STM32 microcontroller was used, specifically the STM32H7 series. The sensor data is collected using the LSM6DSO16IS (which integrates an accelerometer and gyroscope) and the LIS2MDL magnetometer, both mounted on the X-NUCLEO-IKS4A1 expansion board. Sensor fusion is performed using the MotionFX library provided by STMicroelectronics, which supports 9-axis data fusion. The project is developed and debugged using STM32CubeIDE, with data visualization and debugging facilitated through UART communication.

Chapter 2

Hardware

2.1 Nucleo - H723ZG

The **Nucleo - H723ZG** development board, shown in figure 2.1, is based on the STM32H723xG microcontroller, a high-performance 32-bit MCU built around the Arm Cortex-M7 core operating at up to 550 MHz. This core includes a single and double precision floating-point unit (FPU) and supports a comprehensive set of DSP instructions, making it well-suited for computationally intensive tasks such as real-time sensor fusion. The microcontroller features 1 MByte of Flash memory and up to 564 KBytes of RAM.

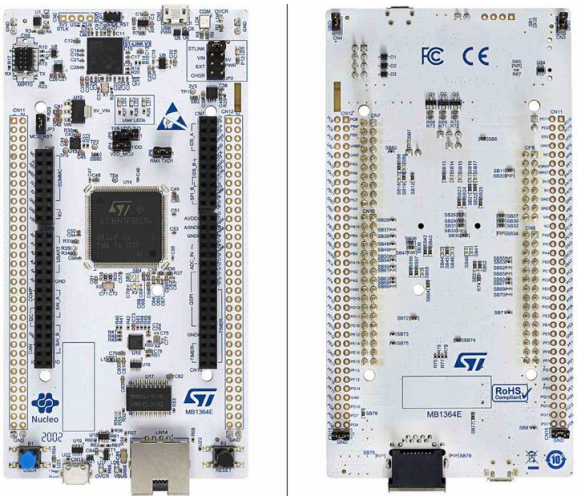

Figure 2.1: Nucleo - H723ZG board

2.2 X-NUCLEO IKS4A1

The **X-NUCLEO-IKS4A1**, shown in figure 2.2, is a motion MEMS and environmental sensor expansion board. It integrates a comprehensive suite of sensors, including:

- **LSM6DSO16IS**: a 6-axis IMU combining an accelerometer and a gyroscope
- **LIS2MDL**: a 3-axis magnetometer
- **LPS22DF**: to measure pressure
- **SHT40AD1B**: to measure humidity
- **STTS22H**: to measure temperature

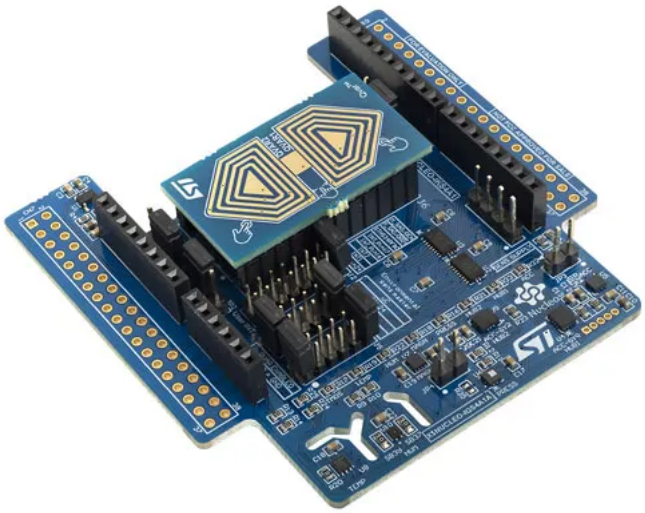

Figure 2.2: IKS4A1 board

Communication with the sensors is handled via the **I²C**. The I²C (Inter-Integrated Circuit) protocol is a widely adopted two-wire serial communication standard designed for low-speed communication between integrated circuits. It utilizes two bidirectional lines: SCL (Serial Clock) for synchronization and SDA

(Serial Data) for transmitting information, enabling efficient communication between a master device (e.g., a microcontroller) and multiple slave devices.

2.3 Connection between the two boards

The Nucleo - H723ZG board and the X-NUCLEO-IKS4A1 expansion board are interconnected through Arduino Uno R3-compatible headers. These standardized connectors allow straightforward stacking of expansion boards without the need for additional wiring. The IKS4A1 board uses the I^2C interface for communication to the `D14` (SDA) and `D15` (SCL) pins on the Arduino header, which correspond to PB9 and PB8 on the STM32H7, respectively. Power and ground connections are also routed through these headers, ensuring reliable electrical interfacing between the two boards.

Chapter 3

Data Fusion

3.1 Kalman Filter

The Kalman filter is a robust and widely-used algorithm designed to estimate the internal state of a dynamic system from a sequence of noisy and uncertain measurements. It is especially useful in the field of robotics for sensor fusion, where combining inputs from multiple sensors leads to a more accurate understanding of both the robot's position and its surroundings. The algorithm follows a recursive approach consisting of two primary phases: prediction and correction.

- **Prediction phase:** During this step, the filter forecasts the system's next state based on a mathematical model of the system's behavior and any applied control signals. It also predicts the uncertainty (covariance) associated with this estimated state.
- **Correction phase:** In this step, the predicted state is refined using newly acquired measurement data. The Kalman gain is computed to determine how much influence the new measurement should have relative to the prediction.

3.2 Kalman Filter Working Principle

The Kalman Filter estimates both the system state and the associated uncertainty by maintaining a probabilistic representation of the state. The system state is represented by a state vector $\mathbf{x}$, while the uncertainty of this estimate is described by the error covariance matrix $\mathbf{P}$.
During the prediction phase, the filter propagates the current state estimate forward in time using a system model. This step accounts for possible uncertainties in the system dynamics through the process noise covariance matrix $\mathbf{Q}$, which models unpredicted disturbances and modeling errors.
In the correction phase, sensor measurements are incorporated using the measurement model. The measurement noise covariance matrix $\mathbf{R}$ characterizes the reliability of the sensor data. The Kalman Gain $\mathbf{K}$ is then computed to optimally balance the confidence between the predicted state and the new measurement.
This recursive structure allows the filter to continuously refine its estimates in real time, making it well suited for applications such as localization, tracking, and sensor fusion in robotics.

The operation of the Kalman Filter can be summarized as a continuous recursive loop consisting of the following steps:

1. Start from a previous state estimate and its associated error covariance.
2. Predict the next system state using the system model.
3. Predict the uncertainty associated with the predicted state.
4. Acquire a new sensor measurement.
5. Compute the Kalman Gain based on the predicted uncertainty and the measurement noise.
6. Update the state estimate using the new measurement.

7. Update the error covariance and repeat the process for the next time step.

The figure 3.1 summarizes the entire procedure described above.

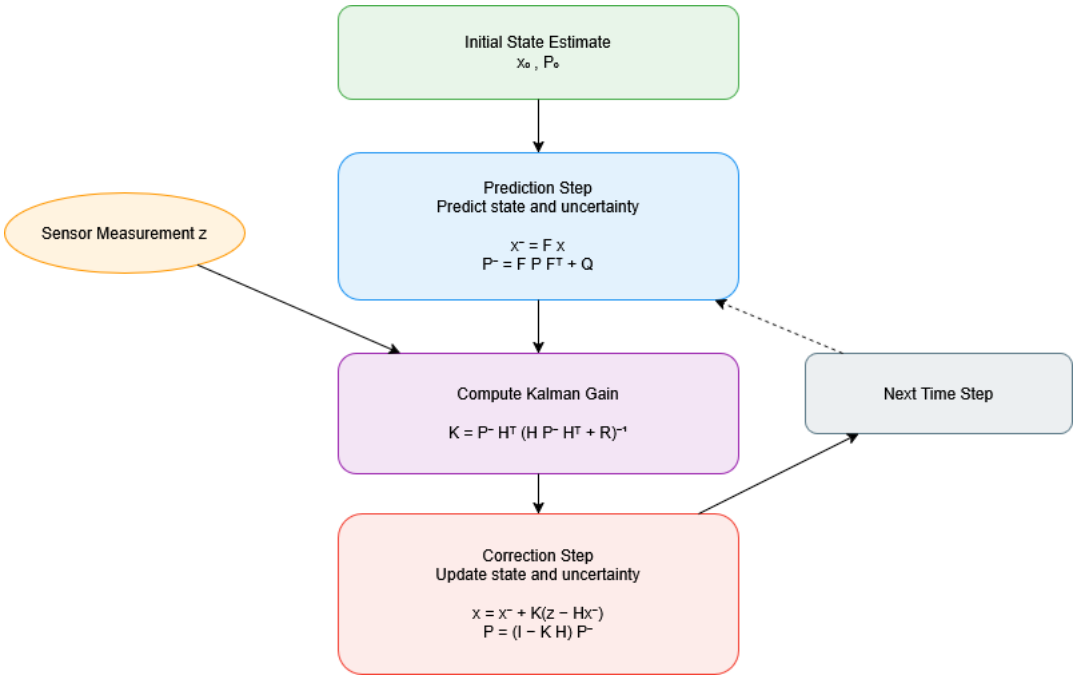

Figure 3.1: Kalman filter flow chart

3.2.1 Kalman filter vs Extended Kalman filter

As discussed in the previous paragraph, the standard Kalman filter is used when working with **linear models**. However, many real-world systems are nonlinear, and this is where the **Extended Kalman Filter** (EKF) comes in. The EKF extends the standard Kalman Filter to handle **nonlinear** systems by linearizing the process and measurement models around the current estimate using a first-order Taylor expansion. While this allows the EKF to be applied to a broader range of problems, the linearization introduces approximation errors, which can affect the accuracy and stability of the filter compared to the standard Kalman Filter in truly linear systems.

Chapter 4

Firmware

This chapter presents the firmware development process carried out for the STM32-based embedded system, which integrates motion sensors to estimate orientation and heading. The firmware is designed to run on the STM32H723ZGT6 microcontroller and interfaces with the X-NUCLEO-IKS4A1 expansion board, which includes an accelerometer, gyroscope, and magnetometer. The chapter outlines the key steps involved in initializing and configuring the sensors, acquiring and processing the raw sensor data, and implementing sensor fusion using STMicroelectronics' MotionFX library. Additionally, it describes how real-time calibration, relative orientation tracking, and UART-based data transmission were integrated into the system. UART is a hardware-based asynchronous serial communication protocol that enables reliable point-to-point data exchange without a shared clock signal. The microcontroller streams the estimated orientation and calibration parameters over the TX/RX lines using a predefined baud rate and frame format, allowing real-time monitoring, logging, and debugging of the system behavior on a PC.

4.1 Used sensors initialization

In order to enable accurate motion sensing and sensor fusion, it is essential to properly initialize and configure the onboard sensors at the beginning of the firmware execution.

The **sensors_init()** function fulfills this role by setting up the inertial and magnetic sensors.
It begins by initializing the LSM6DSO16IS sensor for both accelerometer and gyroscope data acquisition, as well as the LIS2MDL sensor for magnetometer measurements. Once initialized, it enables all three sensing components to begin capturing motion data.
Following this activation, the function calls **set_sensors_scale()** to configure the full-scale settings or sensitivity ranges of the sensors, ensuring that the data output aligns with the expected physical units.
If any step in this process fails, the function invokes the **Error_Handler()** to safely manage the error.

```
1  void sensors_init(){
2    IKS4A1_MOTION_SENSOR_Init(IKS4A1_LSM6DSO16IS_0,
       MOTION_ACCELERO | MOTION_GYRO);
3    IKS4A1_MOTION_SENSOR_Init(IKS4A1_LIS2MDL_0, MOTION_MAGNETO);
4      IKS4A1_MOTION_SENSOR_Enable(IKS4A1_LSM6DSO16IS_0,
       MOTION_ACCELERO);
5    IKS4A1_MOTION_SENSOR_Enable(IKS4A1_LSM6DSO16IS_0, MOTION_GYRO
       );
6    IKS4A1_MOTION_SENSOR_Enable(IKS4A1_LIS2MDL_0, MOTION_MAGNETO)
       ;
7
8    if (set_sensors_scale() != BSP_ERROR_NONE)
9    {
10       Error_Handler();
11   }
12 }
```

4.2 Reading sensor data

The **read_insert_sensors()** function is responsible for acquiring and preprocessing raw sensor data from the onboard motion sensors before passing it to the MotionFX sensor fusion library. It retrieves acceleration, gyroscope, and magnetometer data from the LSM6DSO16IS and LIS2MDL sensors using the **IKS4A1_MOTION_SENSOR_GetAxes()** function. Once data acquisition for each sensor is successfully completed, the retrieved data are formatted in order to satisfy the **MFX_input_t** struc-

ture used in the MotionFX library. To do so, raw readings are converted into the appropriate physical units; specifically, the accelerometer and gyroscope readings are scaled from milligravity and millidegrees per second to gravity (g) and degrees per second (dps), respectively, by dividing by 1000. The magnetometer values are first scaled by a factor of 0.15 µT per count and normalized (dividing by 50), as required by MotionFX, then corrected by subtracting previously calculated hard-iron biases stored in **magCalOutput.hi_bias**.

```
void read_insert_sensors(){
    IKS4A1_MOTION_SENSOR_Axes_t raw_acc, raw_gyro, raw_mag;

  if(IKS4A1_MOTION_SENSOR_GetAxes(IKS4A1_LSM6DSO16IS_0,
   MOTION_ACCELERO, &raw_acc) == BSP_ERROR_NONE)
  {
    // MFX_input_t structure needs acc [g] gyro [dps] and [uT
   /50]
    mfx_input.acc[0] = raw_acc.x / 1000.0f;
    mfx_input.acc[1] = raw_acc.y / 1000.0f;
    mfx_input.acc[2] = raw_acc.z / 1000.0f;
  }

  if(IKS4A1_MOTION_SENSOR_GetAxes(IKS4A1_LSM6DSO16IS_0,
   MOTION_GYRO, &raw_gyro) == BSP_ERROR_NONE)
  {
    mfx_input.gyro[0] = raw_gyro.x / 1000.0f;
    mfx_input.gyro[1] = raw_gyro.y / 1000.0f;
    mfx_input.gyro[2] = raw_gyro.z / 1000.0f;
  }

  if(IKS4A1_MOTION_SENSOR_GetAxes(IKS4A1_LIS2MDL_0,
   MOTION_MAGNETO, &raw_mag) == BSP_ERROR_NONE)
  {
    mfx_input.mag[0] = (raw_mag.x * 0.15f / 50.0f) -
   magCalOutput.hi_bias[0];
    mfx_input.mag[1] = (raw_mag.y * 0.15f / 50.0f) -
   magCalOutput.hi_bias[1];
    mfx_input.mag[2] = (raw_mag.z * 0.15f / 50.0f) -
   magCalOutput.hi_bias[2];
  }
}
```

The raw magnetometer data are multiplied by the **sensitivity value** which is a scale factor that converts the raw output from the sensor (LSB) into a physical unit. The obtained physical unit is then divided by 50 to match the MFX_input_t required measure unit.

Figure 4.1 reports the table that summarizes the conversion process of raw sensor data into the units required by the MotionFX library.

Parameter	Raw unit	Operation	Final unit
Accelerometer	mg	/ 1000	g
Gyroscope	mdps	/ 1000	dps
Magnetometer	LSB	×0.15 / 50	µT/50

Table 4.1: Raw data transformation table

4.3 Magnetometer calibration

The **start_mag_cal** function is responsible for performing the magnetometer calibration procedure using the *MotionFX* library. It begins by initializing variables for storing raw magnetometer data. The function transmits an initial UART message to indicate that calibration phase has started. Within a **do-while** loop, it repeatedly acquires magnetometer readings from the LIS2MDL sensor, scales the raw data using the sensor's sensitivity (0.15 µT/LSB), and stores it in the *magCalInput* structure along with a timestamp. This data is then processed by **MotionFX_MagCal_run**, which performs the calibration. The calibration output parameters are then retrieved and the calibration output parameters are retrieved using **MotionFX_MagCal_getParams**. The calculated bias values (originally in µT/50) are scaled back to µT for readability, and the calibration quality is printed on the UART.
A short delay is introduced between iterations to avoid overwhelming the calibration routine. The loop continues until the calibration quality reaches the required level (*MFX_MAGCALGOOD*) or the maximum number of attempts is reached. After the loop, a final message is sent over UART, indicating if the final calibration quality is good or not.

```
1 void start_mag_cal(void) {
2     IKS4A1_MOTION_SENSOR_Axes_t mag_data;
```

```
3     char msg_mag[128];
4     uint32_t timeout = 0;
5     const uint32_t max_attempts = 10000;
6
7     HAL_UART_Transmit(&huart3, (uint8_t*)"Calibrating
    magnetometer...\r\n", 29, HAL_MAX_DELAY);
8
9     do {
10        IKS4A1_MOTION_SENSOR_GetAxes(IKS4A1_LIS2MDL_0,
    MOTION_MAGNETO, &mag_data);
11        magCalInput.mag[0] = (float)mag_data.x * 0.15f / 50.0f;
12    magCalInput.mag[1] = (float)mag_data.y * 0.15f / 50.0f;
13    magCalInput.mag[2] = (float)mag_data.z * 0.15f / 50.0f;
14        magCalInput.time_stamp = HAL_GetTick();
15
16        MotionFX_MagCal_run(&magCalInput);
17        MotionFX_MagCal_getParams(&magCalOutput);
18
19        timeout++;
20        sprintf(msg_mag, "Calibration Quality: %d. Offsets ( T
    ): X=%.2f Y=%.2f Z=%.2f\r\n",
21                magCalOutput.cal_quality,
22                        magCalOutput.hi_bias[0] * 50.0f, //
    convert from  T /50 to  T
23                        magCalOutput.hi_bias[1] * 50.0f,
24                        magCalOutput.hi_bias[2] * 50.0f);
25        HAL_UART_Transmit(&huart3, (uint8_t*)msg_mag, strlen(
    msg_mag), HAL_MAX_DELAY);
26        // delay to avoid having too frequent MagCal_run calls
27      HAL_Delay(10);
28
29    } while((magCalOutput.cal_quality != MFX_MAGCALGOOD) && (
    timeout < max_attempts));
30
31    if(magCalOutput.cal_quality == MFX_MAGCALGOOD) {
32
33        sprintf(msg_mag, "Calibration SUCCESSFUL. Quality: %d\r
    \n", magCalOutput.cal_quality);
34
35    } else {
36        sprintf(msg_mag, "Calibration FAILED. Quality: %d\r\n",
    magCalOutput.cal_quality);
37    }
38
39    HAL_UART_Transmit(&huart3, (uint8_t*)msg_mag, strlen(
    msg_mag), HAL_MAX_DELAY);
40 }
```

4.4 Main Function

At the beginning of the main() function, the system and peripherals are initialized by calling several functions:

- **HAL_Init()**: sets up the Hardware Abstraction Layer, preparing the STM32 for use.
- **MPU_Config()**: configures the Memory Protection Unit, which is often used for security and stability purposes in STM32H7 applications.
- **SystemClock_Config()**: sets up the system clock to ensure the microcontroller operates at the correct frequency.
- **MX_GPIO_Init()**: configures the General-Purpose Input/Output (GPIO)
- **MX_USART3_UART_Init()** initialize the UART (USART3) interface used to show the data on the serial monitor.
- **MX_CRC_Init()**: initializes the CRC (Cyclic Redundancy Check) peripheral of the STM32 microcontroller. This peripheral is used to compute checksums efficiently, which is useful for verifying data integrity. In this context, it's specifically required by the MotionFX library, which uses CRC to validate internal data structures and ensure correct processing during sensor fusion.

After initializing the system and peripherals, the **sensors_init()** function is called to configure and activate the onboard sensors, such as the accelerometer, gyroscope, and magnetometer. This function typically sets up the I^2C or SPI interfaces and prepares the sensors for data acquisition. In particular, SPI (Serial Peripheral Interface) is a synchronous serial communication protocol based on a master–slave architecture, in which the microcontroller acts as the master and the sensors as slaves. Data are exchanged over separate lines for transmission and reception and are synchronized by a dedicated clock signal, enabling high-speed and low-latency communication. Following the function **MotionFX_initialize(mfxstate)** initializes the MotionFX sensor fusion engine, allocating and configuring the internal state

structures required for computation.
Finally, **MotionFX_enable_9X(mfxstate, MFX_ENGINE_ENABLE)** enables the 9-axis fusion mode, which combines data from the accelerometer, gyroscope, and magnetometer to estimate the device's orientation and heading.

```
int main(void) {
    HAL_Init();
    MPU_Config();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_CRC_Init();
    MX_USART3_UART_Init();
    sensors_init();
    MotionFX_initialize(mfxstate);
    MotionFX_enable_9X(mfxstate, MFX_ENGINE_ENABLE);
```

The main loop continuously runs after initialization to perform real-time sensor data acquisition, fusion, and output. It starts by checking if the user button is pressed. When pressed, it triggers a magnetometer calibration by calling **MotionFX_MagCal_init()** and a custom **start_mag_cal()** function, followed by a short delay. If the initial yaw angle has not been set yet and the output yaw is valid, it stores the current yaw as a reference for relative heading calculation.
The function **read_insert_sensors()** is then called to read the raw data from the accelerometer, gyroscope and magnetometer and populate the MotionFX input structure.
The loop computes the elapsed time (delta_time) since the last update using **HAL_GetTick()**, which is essential for time-dependent sensor fusion algorithms like the Kalman filter.
Next, the functions **MotionFX_propagate()** and **MotionFX_update()** compute the fused orientation (yaw, pitch, roll), linear acceleration, and heading. The relative yaw is calculated by subtracting the initial reference yaw from the current yaw, allowing orientation tracking from a known direction.
Finally, all processed sensor data and orientation results are formatted into a string and sent over UART using **HAL_UART_Transmit()**, allowing for real-time monitoring via

the serial terminal. A short delay (**HAL_Delay(10)**) ensures the loop runs at a manageable frequency, reducing CPU load and UART congestion. The figure 4.2 shows the flow of function calls.

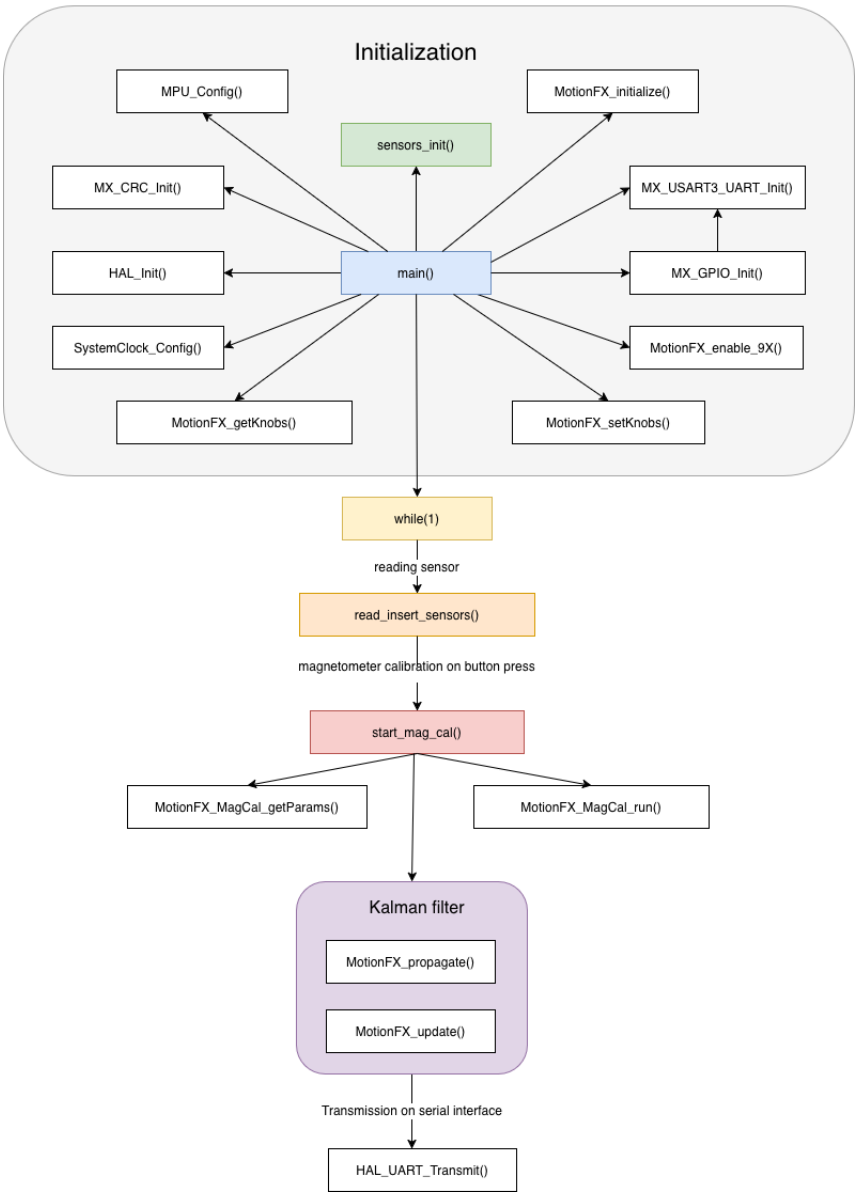

Figure 4.2: Flow chart function calls

4.5 Final Results

This project successfully implemented a complete framework for acquiring, processing, and fusing inertial sensor data in order to estimate the orientation of an embedded system in three-dimensional

space. By integrating accelerometer, gyroscope, and magnetometer measurements from the X-NUCLEO-IKS4A1 expansion board with the H723ZG Nucleo board, a full real-time sensor fusion pipeline was developed and validated.
Although the system is capable of producing yaw, pitch, and roll estimates, the results are not yet perfectly stable or accurate under all operating conditions. This behavior highlights the intrinsic complexity of inertial sensor fusion, especially in the presence of sensor noise, magnetic disturbances, and calibration inaccuracies. Nevertheless, the implemented solution demonstrates the correct functioning of the acquisition chain, data scaling, MotionFX integration, magnetometer calibration routine.
Final results are shown in figure 4.3

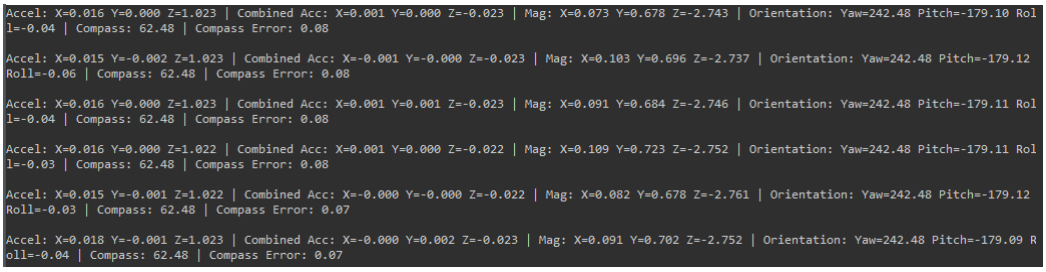

Figure 4.3: Final results