McGill University

ECSE 426: Microprocessor Systems

Final Project Report

Diego Macario

Saki Kajita

Wen Bo Zhang

Stephen Cambridge

Table of contents

1. Abstract

The goal of the project described in this document was to design a multicomponent system to explore the interactions between embedded peripherals and a smartphone device. This involved developing an interface to support the communication between an STM32F4 Discovery board, an STM32F401RE Nucleo board and an Android smartphone. Two-way communication between the Nucleo board and the smartphone was achieved over Bluetooth Low Energy (BLE) by fitting the Nucleo board with an IDB04A1 BLE shield. Additionally, communication from the Discovery board to the Nucleo board was achieved via the Serial Peripheral Interface (SPI) protocol, while communication from the Nucleo board to the Discovery board was achieved using a system of binary codes.

The final system allows for temperature and orientation measurements to be sent from the Discovery board to the smartphone, which displays these values on its screen. The Discovery board also supports a double tap function which raises a notification on the smartphone when a double tap is detected, and the smartphone has controls for the LEDs included on the Discovery board.

2. Problem

From a high level perspective, the problem being addressed in this document consists in designing a multicomponent system with an STM32F4 Discovery board, an STM32F401RE Nucleo board, an IDB04A1 BLE shield and an Android smartphone. This multicomponent system has to be capable of:

- Concurrently sampling the temperature sensor and accelerometer included on the Discovery board.
- Transmitting temperature, pitch and roll measurements from the Discovery board to the Nucleo board using the SPI communication protocol.
- Transmitting temperature, pitch and roll measurements from the Nucleo board to the smartphone via the BLE shield.
- Displaying temperature, pitch and roll measurements on the smartphone.
- Displaying a notification on the smartphone when the Discovery board detects a double tap.
- Allowing the user to control the pattern displayed by the 4 LEDs included on the Discovery board with instructions entered through the smartphone.

The tasks that need to be accomplished in order to enable these functionalities can be broken down as follows:

- Configuring the analog to digital converter (ADC): The ADC must be configured to poll the temperature sensor hardwired within the board at a frequency of 100 Hz [1].

- Configuring the timer used to manage the ADC: A hardware timer must be configured to acquire the values provided by the ADC, start new ADC conversions, and manage the temperature sensor thread.
- Converting the temperature sensor measurements to Celsius: The values reported by the ADC must be converted to Celsius using an appropriate formula.
- Filtering the temperature sensor measurements: The converted temperature values must be filtered using a one-dimensional Kalman filter. This will involve choosing appropriate values for the parameters that define the state of the filter.
- Configuring the accelerometer: The LIS3DSH 3-axial digital accelerometer must be configured to acquire X, Y and Z acceleration measurements at a frequency of 25 Hz [2]. It must also be configured to raise interrupts whenever a new set of measurements is available, which will be used to manage the accelerometer thread.
- Calibrating the accelerometer: The accelerometer must be calibrated using the least squares method in order to calculate the pitch and the roll with 4° accuracy [2].
- Filtering accelerometer measurements: The calibrated X, Y and Z acceleration measurements must be filtered using 3 independent one-dimensional Kalman filters (one for each direction). This will involve choosing appropriate values for the parameters that define the state of each filter.
- Calculating the pitch and the roll: The pitch and the roll must be calculated using tri-axis tilt sensing, which makes the sensitivity to changes in the tilt constant over 360° of rotation [3].
- Interpreting the data provided by the accelerometer to detect double taps: The effects of a double tap on the X, Y and Z acceleration measurements must be studied to determine how to reliably detect such an event.
- Configuring the double tap interrupt mechanism: This task will involve configuring the necessary components to raise a notification on the smartphone whenever a double tap is detected.
- Configuring the display LEDs: The 4 LEDs included on the Discovery board must be configured so that their brightness can be controlled via Pulse Width Modulation (PWM).
- Configuring the SPI communication between the Discovery board and the Nucleo board: This task will involve configuring an SPI module on each board, and it will also involve defining one of them as the master and the other as the slave.
- Defining the threads and their priorities: This task will involve creating 4 threads (one for the temperature sensor, one for the accelerometer, one for the LED display and one to support the communication between the Discovery and the Nucleo boards). Their priorities must be defined based on how critical they are to the operation of the system.
- Configuring the Bluetooth communication: The BLE shield must be configured to send measurements to the smartphone and to receive instructions from it.

- Developing the Android application: An Android application must be implemented to receive measurements via Bluetooth and to display them on the smartphone's screen. It must also be capable of sending instructions in the form of integer values to the Nucleo board via Bluetooth.

3. Theory and Hypothesis

3.1 Temperature Sensor Data Conversion

The sensor available for this experiment generates a voltage that varies linearly with temperature. Its supported temperature range goes from -40 to 125 °C [5], and its conversion range goes from 1.8 to 3.6 V [6]. After processing the measured analog values with an ADC they can be converted to Celsius using the expression below:

$$Temperature\ (in\ °C) = \frac{V_{sense} - V_{25}}{Avg\_Slope} + 25 \tag{1}$$

Where V_{sense} corresponds to the voltage reported by the temperature sensor, V_{25} corresponds to the voltage reported when the sensor is at a temperature of 25 °C and Avg_Slope corresponds to the average slope of the temperature vs. V_{sense} curve. Note that V_{25} is typically 0.76 V and that Avg_Slope is typically 2.5 mV/°C [6].

3.2 One-dimensional Kalman Filter

In order to filter the data acquired by the temperature sensor and the accelerometer we require 4 independent one-dimensional Kalman filters. One of these will be used by the temperature sensor, while the other three will be used by the accelerometer to filter its X, Y and Z measurements.

This type of filter is a state-based adaptive estimator [7], which means that it receives as its input a series of measurements and generates more precise estimates based on the influence of previous predictions. The effect of past values is contained in the state of the filter, which is updated each time a new measurement is received. The state itself is defined by 5 variables, which are: $\boldsymbol{x}$ for the filtered value, $\boldsymbol{q}$ for the process noise, $\boldsymbol{r}$ for the sensor noise, $\boldsymbol{p}$ for the estimated error, and $\boldsymbol{k}$ for the Kalman gain. Equations (2) to (6) are used to update and maintain the state of the filter:

$$x = x \tag{2}$$

$$p = p + q \tag{3}$$

$$k = \frac{p}{p + r} \tag{4}$$

$$x = x + k \times (measurement - x) \tag{5}$$

$$p = (1 - k) \times p \tag{6}$$

Where equations (2) and (3) represent the prediction of the Kalman filter, and equations (4) to (6) calculate the measurement update [8].

3.3 Accelerometer Calibration

In order to calibrate the LIS3DSH 3-axial digital accelerometer to measure the tilt with 4° accuracy it is necessary to follow the least square method. This technique uses raw data measurements collected at 6 stationary positions to determine 12 calibration parameters [3]. The entire process is based on the following expression:

$$Y = w \cdot X \tag{7}$$

Where X is a 4×3 matrix that contains the 12 calibration parameters, w is a $6n \times 4$ matrix that contains the raw X, Y and Z measurements collected at the 6 stationary positions, and Y is the earth gravity vector. Figure 1 below presents the axes used to define the 6 stationary positions, while table 1 specifies each position and its corresponding vectors.

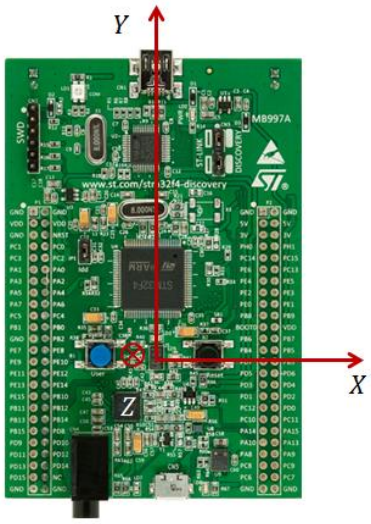

Figure 1: Positive X, Y and Z directions with respect to the accelerometer.

Table 1: Stationary positions and their corresponding w and y vectors.

Position	Direction	w vector	y vector
1	Z Down	$w_1 = [A_{x1}\ A_{y1}\ A_{z1}\ 1]_{nx4}$	$y_1 = [0\ 0\ 1]_{nx3}$
2	Z Up	$w_2 = [A_{x2}\ A_{y2}\ A_{z2}\ 1]_{nx4}$	$y_2 = [0\ 0\ -1]_{nx3}$
3	Y Down	$w_3 = [A_{x3}\ A_{y3}\ A_{z3}\ 1]_{nx4}$	$y_3 = [0\ 1\ 0]_{nx3}$
4	Y Up	$w_4 = [A_{x4}\ A_{y4}\ A_{z4}\ 1]_{nx4}$	$y_4 = [0\ -1\ 0]_{nx3}$

5	X Down	$w_5 = \left[A_{x5}\ A_{y5}\ A_{z5}\ 1\right]_{nx4}$	$y_5 = [1\ 0\ 0]_{nx3}$
6	X Up	$w_6 = \left[A_{x6}\ A_{y6}\ A_{z6}\ 1\right]_{nx4}$	$y_6 = [-1\ 0\ 0]_{nx3}$

Note that the vectors corresponding to different positions do not have to have the same n dimension, but this is recommended for consistency. Also note that n must be at least 1000 in order to appropriately calibrate the accelerometer [3]. With the measurements completed, the w and Y vectors can be composed using equations (8) and (9), respectively.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}_{6nx4} \tag{8}$$

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}_{6nx3} \tag{9}$$

Finally, the calibration parameters can be found by solving equation (7), which results in the following expression:

$$X = [w^T \cdot w]^{-1} \cdot w^T \cdot Y = \begin{bmatrix} ACC_{11} & ACC_{12} & ACC_{13} \\ ACC_{21} & ACC_{22} & ACC_{23} \\ ACC_{31} & ACC_{32} & ACC_{33} \\ ACC_{41} & ACC_{42} & ACC_{43} \end{bmatrix} \tag{10}$$

This calibration matrix can then be applied to raw measurements as they are read by the accelerometer using equations (11), (12) and (13).

$$X_{calibrated} = X_{raw} \cdot ACC_{11} + Y_{raw} \cdot ACC_{21} + Z_{raw} \cdot ACC_{31} + ACC_{41} \tag{11}$$

$$Y_{calibrated} = X_{raw} \cdot ACC_{12} + Y_{raw} \cdot ACC_{22} + Z_{raw} \cdot ACC_{32} + ACC_{42} \tag{12}$$

$$Z_{calibrated} = X_{raw} \cdot ACC_{13} + Y_{raw} \cdot ACC_{23} + Z_{raw} \cdot ACC_{33} + ACC_{43} \tag{13}$$

3.4 Pitch and Roll Calculation

The pitch is defined as the angle between the positive x axis and the local horizontal plane, while the roll is defined as the angle between the positive y axis and the local horizontal plane [3]. These two forms of tilt are illustrated in figure 2 below.

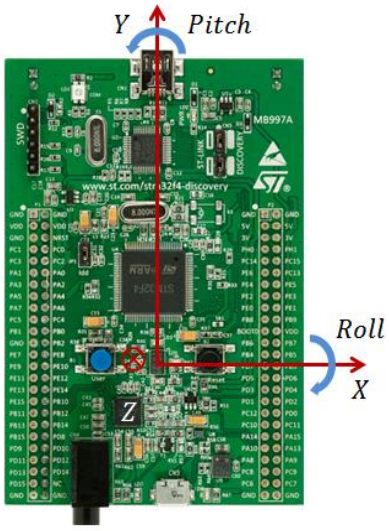

Figure 2: Pitch and roll angles measured with respect to the local horizontal plane.

In order to achieve a constant sensitivity to changes in the tilt over 360° of rotation, one must employ tri-axis tilt sensing [3]. This technique uses X, Y and Z measurements to calculate the pitch and roll with the following expressions:

$$Pitch = \tan^{-1}\frac{A_x}{\sqrt{\left(A_y\right)^2 + (A_z)^2}} \tag{14}$$

$$Roll = \tan^{-1}\frac{A_y}{\sqrt{(A_x)^2 + (A_z)^2}} \tag{15}$$

Where A_x is the projection of the gravity vector onto the X axis, A_y is the projection of the gravity vector onto the Y axis and A_z is the projection of the gravity vector onto the Z axis. In order for these expressions to provide values that range from 0 to 360° it is necessary to modify them based on the quadrant of rotation they are being applied in. In the case of the pitch, the quadrants are defined with respect to the positive X and Z directions when the Y direction is parallel to the horizontal plane, as illustrated in figure 3. Table 2 presents the signs of A_x and A_z in each quadrant, and the necessary adjustments to make the $Pitch$ value provided by equation (14) range from 0 to 360°.

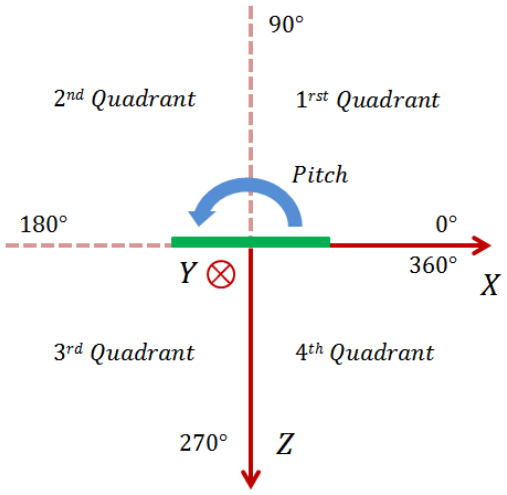

Figure 3: Rotation quadrants for the pitch.

Table 2: Necessary adjustments to make the pitch range from 0 to 360°.

Quadrant	Sign of A_x	Sign of A_z	Pitch
1	-	+	$-Pitch$
2	-	-	$180 + Pitch$
3	+	-	$180 + Pitch$
4	+	+	$360 - Pitch$

As for the roll, the quadrants are defined with respect to the positive Y and Z directions when the X direction is parallel to the horizontal plane, as illustrated in figure 4. Table 3 presents the signs of A_y and A_z in each quadrant, and the necessary adjustments to make the $Roll$ value provided by equation (15) range from 0 to 360°.

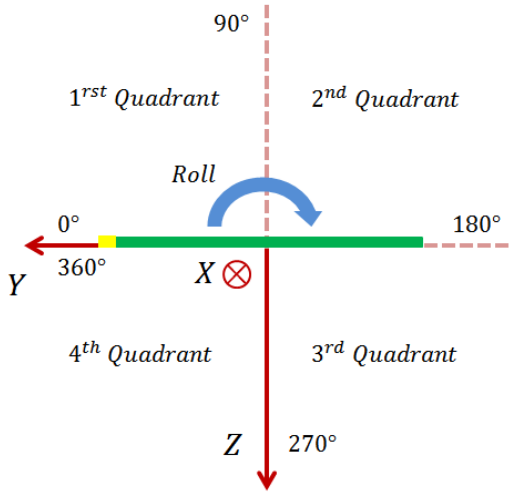

Figure 4: Rotation quadrants for the roll.

Table 3: Necessary adjustments to make the roll range from 0 to360°.

Quadrant	Sign of A_y	Sign of A_z	Roll
1	-	+	$-Roll$
2	-	-	$180 + Roll$
3	+	-	$180 + Roll$
4	+	+	$360 - Roll$

3.5 Timer Speed Configuration

The speed of the hardware timers included in the STM32F4 Discovery board can be defined by selecting $Prescaler$ and $Period$ values that satisfy the following equation:

$$Desired\ timer\ frequency = \frac{Timer\ input\ frequency}{Prescaler \times Period} \tag{16}$$

Note that the $Prescaler$ must never exceed 65535, since it is a 16-bit register, and that the $Period$ must never exceed 65535 or 4294967295, depending on whether a 16-bit or 32-bit timer is being configured [9].

3.6 Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a technique for getting analog results with digital means. This process begins with a digital signal that is used to create a square wave, which switches between 0 and 5 V. By changing the portion of the time that the signal is on versus the time that the signal is off, it is possible to generate voltages between 0 and 5 V. In order to get varying analog values, we must modulate the duration of the time that the voltage is at its maximum (5 V). By repeating this on-off pattern fast enough we can obtain a steadily increasing or decreasing voltage that ranges between 0 and 5 V, which can make the brightness of an LED increase or decrease progressively. To determine the pulse length we must use an external timer and configure it so that we get the desired voltage increase or decrease. The pulse length can be defined by selecting *TIM_Period* and *Duty Cycle* values that satisfy equation 17:

$$Pulse\ Length = \frac{(TIM_{Period} + 1) \cdot Duty\ Cycle}{100} + 1 \tag{17}$$

Where *Duty Cycle* is the voltage between 0 and 5 V expressed in percentage [10].

3.7 Bluetooth Low Energy (BLE)

Bluetooth low energy (BLE) is a wireless personal area network technology that was developed and marketed for its low energy consumption and low costs. It is usually used to communicate with small peripheral devices.

3.7.1 Generic Access Profile (GAP)

Before any communication can occur, each Bluetooth device must be paired. The Generic Access Profile is what manages advertisement and connection between devices. Each BLE device is either a peripheral or a central device. Peripheral devices are usually small low-power devices that connect to central devices. In this project, the BlueNRG shield expansion board will be the peripheral. The central device will be an Android based smartphone [11]. Figure 5 illustrates the state machine that defines BLE.

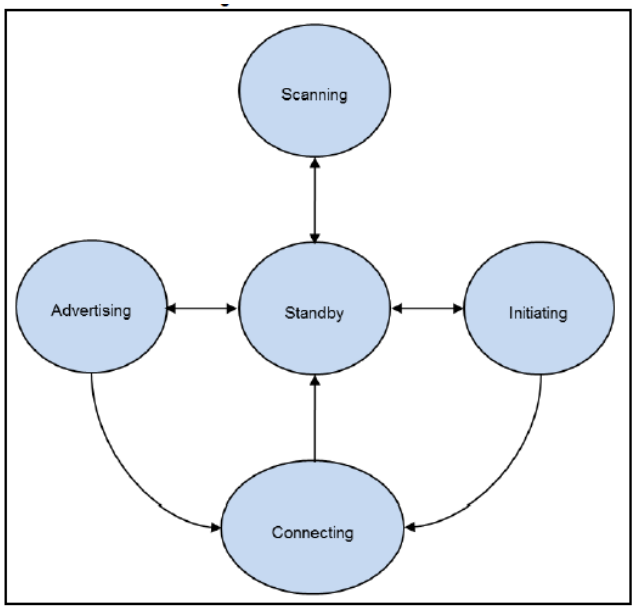

Figure 5: BLE state machine.

In advertising state, the device is transmitting advertising channel packets and possibly listening to responses triggered by these packets. In standby, no packet is transmitted or received. In scanning state, the device looks for advertising channels. In initiating state, the device responds to the advertiser's packets and attempts to initiate a connection. When a connection is made, the initiating device will assume the master role and the advertising device will be the slave [12].

With GAP, a device advertises itself by transmitting a packet of up to 31 bytes at regular intervals of 20ms. To save power, these intervals can be extended to 2 seconds. If a central device is interested in a scan response, the advertising device can respond with additional data [11]. This process is illustrated in figure 6.

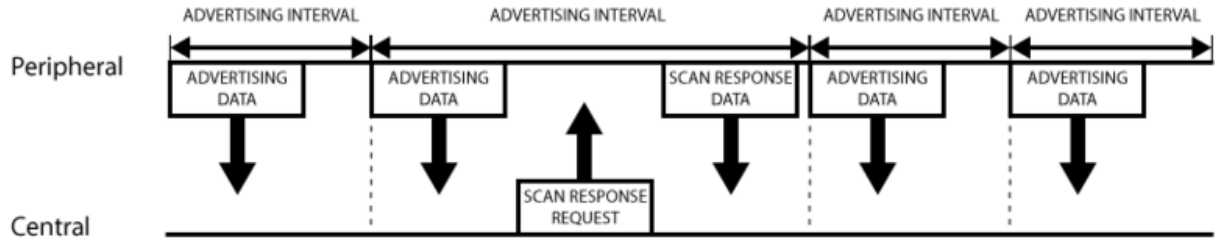

Figure 6: Advertising process.

Once a connection is established between the peripheral and a central device, the broadcasting can be stopped and a two way communication is initiated using the GATT.

3.7.2 Generic Attribute Profile (GATT)

The Generic Attribute Profile defines the way two BLE devices pass information back and forth using concepts such as services and characteristics. When using GATT, each connection is exclusive. A BLE peripheral can only be connected to one central device at a time.

The profile is a pre-defined collection of services defined on the peripheral. As such, it does not exist on the peripheral. The profile should be implemented on the central device to be able to recognise the services offered [13]. Services are used to divide data and contain chunks of data called characteristics. Each service is identified by a 16-bit UUID for officially adopted BLE services or 128-bit UUID for custom ones. By default when a connection is made, there will be a GAP service and a device information service. Characteristics are what encapsulates single data points and are also identified by UUID. Each characteristic has at least two attributes: the main attribute and a value attribute that holds the data. The main attribute defines the value attribute's handle and UUID that tells the client which handle to read to access the value attribute. Characteristics with the notify property will also have a configuration attribute. To enable notifications from those characteristics, the client must write to that configuration [11]. To transfer data back and forth, one can write to and read from a characteristic. The structure of services and characteristics is part of the Attribute Protocol (ATT).

In a GATT transaction, the peripheral, known as the GATT server, holds the ATT lookup data. The GATT client, which could be a smartphone, sends requests to the server and initiates transactions. The server will suggest a "Connection Interval" and the central device will check after each interval to see if the data has changed. This process is illustrated in figure 7.

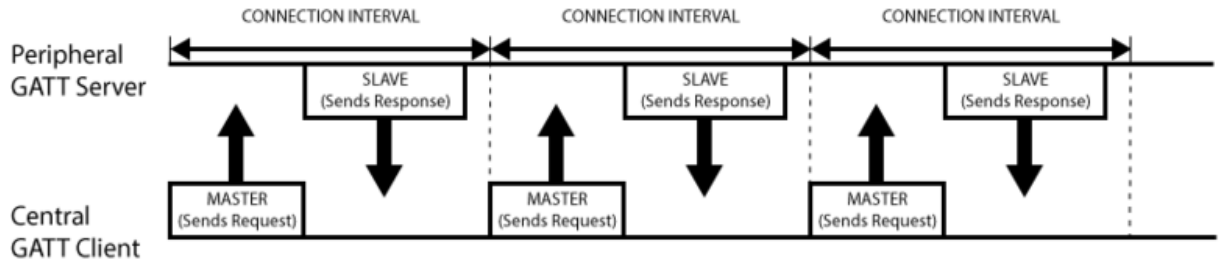

Figure 7: GATT transaction process.

3.8 Hypothesis

We believe that the best way to implement this system is to use configure the Discovery board as the slave and the Nucleo board as the master. By doing this, the Nucleo board can request temperature, pitch and roll measurements from the Discovery board, and it can transmit them to the smartphone, acting as an interface between the Discovery board and the smartphone. We also believe that in order to detect double taps reliably it will be necessary to implement a circular buffer and to base the detection algorithm of the average of the buffer, since using raw values can cause the system to 'detect' double taps when the board is being rotated.

4. Implementation

4.1 Slave

The slave of the system is the Discovery board, whose functions were separated into 4 threads. The operation of these threads, and the way they interact are presented in sections 4.1.1 through 4.1.5.

4.1.1 Temperature Sensor Thread

4.1.1.1 ADC Configuration

The first step to acquire temperature measurements using the STM32F4 Discovery board is to configure the ADC1. Since we only needed to work with channel 16, which is the channel that is hardwired to the temperature sensor, we disabled the Scan Conversion Mode. This in turn enabled the Single Conversion Mode, which is appropriate for our application. We then selected the resolution to be 12 bits in order to increase the precision of each conversion, and we chose to align the data to the right for convenience. We also chose to have the End of Conversion (EOC) flag be enabled at the end of each conversion so that it could be used to control the flow of the program. Additionally, we disabled the Continuous and Discontinuous Conversion Modes, the external triggers and any properties associated with Direct Memory Access (DMA) Requests, since none of these are relevant for our application. We also selected a channel Rank of 1 to explicitly make channel 16 the first one to be converted, even

though there are no other channels involved in our system. Finally, we chose the Clock Prescaler to be 4 and the Sampling Time to be 480 cycles. Recall that the ADC1 receives the APB2 peripheral clock, which is set to 84 MHz, and that the maximum clock frequency for the ADC1 is 36 MHz [6]. By using a Clock Prescaler of 4 the ADC1 receives a clock frequency of 21 MHz, which is below its maximum possible value. As for the Sampling Time of 480 cycles, we chose it that way so that it would be larger than the maximum total conversion time of the ADC1, which is reported to be 16.4 µs [6].

4.1.1.2 Hardware Timer Configuration

In order to sample the temperature sensor at a frequency of 100 Hz we decided to use the TIM4 16-bit hardware timer. This timer is connected to the APB1 peripheral clock, which is set to 84 MHz [9]. Using equation (16) we found that a $Prescaler$ of 840 and a $Period$ of 1000 results in a timer frequency of 100 Hz. We then chose to have the counter count up for convenience, and we assigned its associated TIM4_IRQn an interrupt priority of 2.

4.1.1.3 Temperature One-dimensional Kalman Filter

For this project we used the same one-dimensional Kalman filter parameters we determined qualitatively in experiment 2. Table 4 specifies the value of each parameter and figure 8 illustrates the performance of the filter with values obtained directly from the board.

Table 4: Initial temperature sensor one-dimensional Kalman filter parameters.

q	r	p	k	x
0.0001	0.5	0.1	0	Initial value

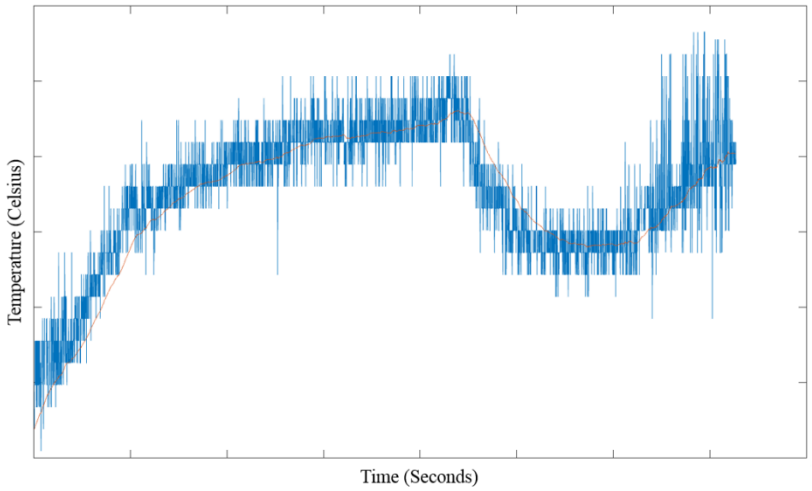

Figure 8: Unfiltered and filtered temperature vectors for **q** equal to 1e-4, **r** equal to 0.5, **p** equal to 0.1, **k** equal to 0, and **x** equal to the initial measurement.

Note that the blue line corresponds to the unfiltered measurements, while they red one corresponds to the filtered ones. As can be observed, the red line is smooth and follows the trend defined by the blue line.

4.1.1.4 Thread Operation

The temperature sensor thread uses the osSignalSet and osSignalWait functions to control its execution. osSignalSet is called inside the HAL_TIM_PeriodElapsedCallback, which means that it is executed every time TIM4 raises an interrupt (100 times per second). osSignalWait is the first instruction inside the loop of the temperature sensor thread, and it is called with the osWaitForever parameter. Using this set up we guarantee that the loop of the temperature sensor thread is only executed 100 times per second. We also guarantee that the thread is sleeping when the ADC does not have new values available.

When the thread is awake it acquires a new temperature measurement by executing HAL_ADC_GetValue. It then converts the measurement to Celsius using equation (1) and it filters it using the one-dimensional Kalman filter described in section 4.1.3. The resulting value is then shared with the display thread, as we will see later on, and a new conversion is started by executing HAL_ADC_Start.

4.1.2 Accelerometer Thread

4.1.2.1 Accelerometer Configuration

The first step to acquire X, Y and Z acceleration measurements is to configure the LIS3DSH 3-axial digital accelerometer. In order to obtain sets of acceleration measurements every 40 ms we set the Output Data Rate (ODR) equal to 25, which makes the accelerometer generate Data Ready interrupts at a frequency of 25 Hz. We then enabled the X, Y and Z axes since calculating the pitch and the roll using tri-axis tilt sensing requires all 3 quantities, as evidenced in equations (14) and (15). We also disabled the Continuous Update Mode, which in turn sets the Block Data Update (BDU) bit to 1. This was done in order to guarantee that the output register is not updated until the MSB and LSB readings are completed. The Bandwidth of the Anti-aliasing Filter was chosen to be 50 Hz, since frequencies above half the sampling frequency (a quantity also known as the Nyquist frequency) appear as aliases [14]. Finally, the Full Scale was defined as 2 g in order to maximize the resolution of the accelerometer for performing accurate tilt measurements.

In terms of the Data Ready interrupt, it was selected to be Active High and Pulsed. This signal is generated on the INT1 line, which is hardwired to pin 0 of port E [9]. For this reason it was necessary to configure this pin in rising interrupt Mode. The Pull-down option was also enabled since the pulses are Active High, as previously mentioned. The speed was

chosen to be Low since this corresponds to a frequency of 2 MHz, which is fast enough to handle interrupt pulses. Finally, the EXTI0_IRQn was given a priority of 1.

With pin 0 of port E configured to generate interrupts whenever it receives a pulse on the INT1 line, we then had to include the EXTI0_IRQHandler and the HAL_GPIO_EXTI_Callback, which are executed whenever the accelerometer enables the Data Ready interrupt.

4.1.2.2 Accelerometer Calibration

For this project we used the same calibration matrix we found in experiment 4. This calibration matrix was obtained following the procedure described in section 3.3, and it is presented below:

$$X = \begin{bmatrix} 0.001001487971751 & -0.000012886855747 & 0.000007862071445 \\ -0.000026697002793 & 0.000971123491151 & -0.000004490648009 \\ 0.000004161225213 & 0.000002812607406 & 0.000960328656662 \\ -0.014532292617973 & 0.000585555300052 & -0.017547993185863 \end{bmatrix} \quad (17)$$

Note that in order to acquire 1000 sets of X, Y and Z acceleration measurements in each of the 6 positions it was necessary to hold the board in each position for 40 seconds while printing the measurements. The matrix itself was obtained by taking these measurements to MATLAB and solving equation (7). The values that compose the matrix are applied to new measurements before they are filtered using equations (11), (12) and (13).

4.1.2.3 X, Y and Z One-dimensional Kalman Filters

After the X, Y and Z acceleration measurements are calibrated, they are then filtered using their corresponding one-dimensional Kalman filters. These 3 filters are independent, but they are initialized with the same parameters in order to make the data consistent.

For this project we used the same one-dimensional Kalman filter parameters we determined qualitatively in experiment 4. Table 5 specifies the value of each parameter and figures 9, 10 and 11 illustrate the performance of the X, Y and Z filters with values obtained directly from the board.

Table 5: Initial X, Y and Z one-dimensional Kalman filters parameters.

q	r	p	k	x
0.05	0.75	0.1	0	Initial value

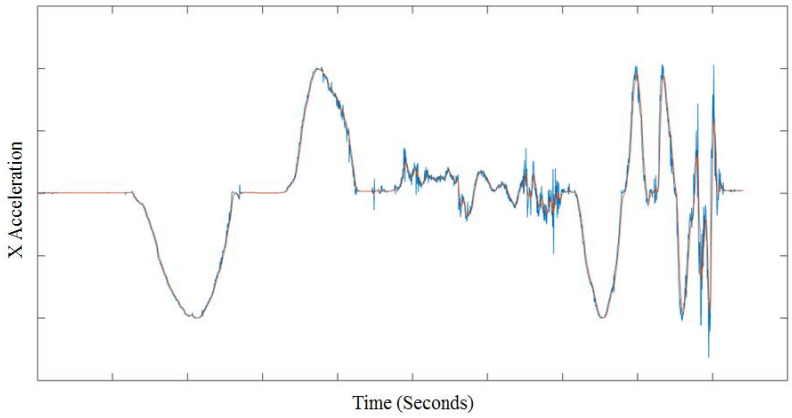

Figure 9: Unfiltered and filtered *X* acceleration vectors for **q** equal to 0.05, **r** equal to 0.75, **p** equal to 0.1, **k** equal to 0, and **x** equal to the initial measurement.

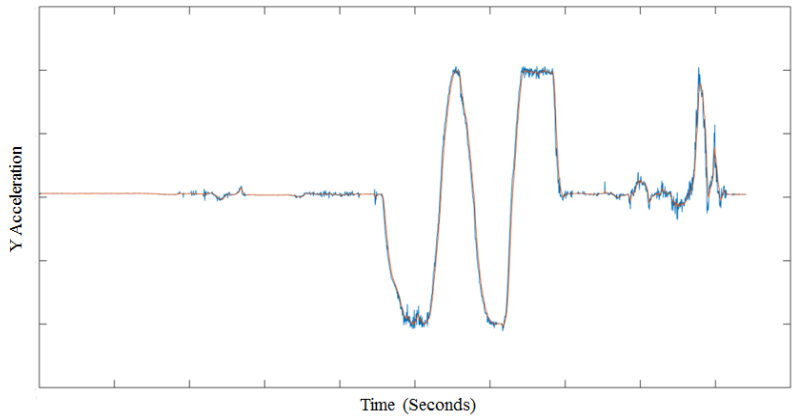

Figure 10: Unfiltered and filtered *Y* acceleration vectors for **q** equal to 0.05, **r** equal to 0.75, **p** equal to 0.1, **k** equal to 0, and **x** equal to the initial measurement.

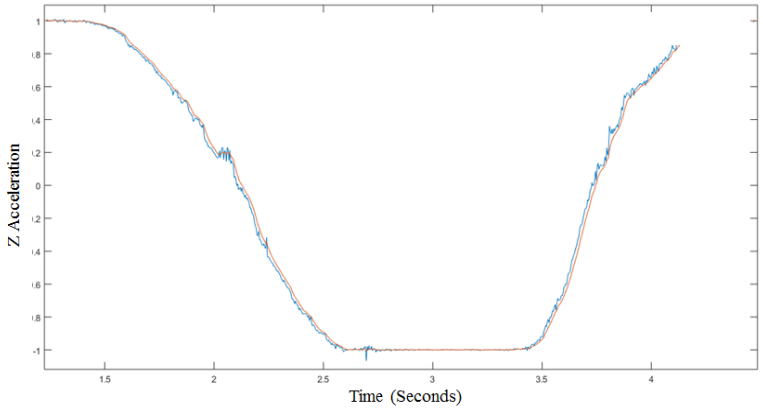

Figure 11: Unfiltered and filtered *Z* acceleration vectors for **q** equal to 0.05, **r** equal to 0.75, **p** equal to 0.1, **k** equal to 0, and **x** equal to the initial measurement.

Note that the blue lines correspond to the unfiltered values, while the red lines correspond to the filtered ones. The parameters that define these filters were chosen to make the output data smooth and accurate with respect to the original values.

4.1.2.4 Double Tap Detection

In our design, double tap detection is achieved through comparison of immediate acceleration values to a moving average of previous values. It is presumed that the user would be tapping on the face on the board, so we look for taps in the z-direction only.

The detection algorithm operates on a 25Hz interval, the same at which the accelerometer is polled. At each evaluation, the z-acceleration is placed into a circular buffer containing 100 values, each new one overwriting the oldest in the array. This represents the most recent four seconds of z-accelerations, which are used to calculate a rolling average. A single tap is registered if the immediate z-acceleration value from the accelerometer exceeds the rolling average by a specified threshold.

To prevent tap noise from drastically changing the rolling average, which is meant to be the “normal” value of the acceleration, the following three values are omitted from the acceleration buffer, being replaced by the current average (so as to have no effect on the subsequent averages).

After a single tap is registered, there is a 0.8 second window for a second tap to be registered and trigger a double tap interrupt. If another does not occur, an internal 'single-tap' flag will be reset. This ensures that two taps spaced too far apart do not count as a double-tap.

In order to keep only meaningful double-taps triggering the routine, we use the pitch and roll of the board (calculated earlier by the accelerometer read functionality) to control an enable signal for the double-tap detection. While the moving buffer is constantly being updated, taps are only checked for and registered if the board is in a flat position – thus noise while rotating the board to test the other accelerometer functions does not trigger taps.

When a double-tap is registered, an interrupt flag is raised on GPIO pin E6, connected to the Nucleo board. The Nucleo then transmits a signal to the smartphone and a notification is set.

4.1.2.5 Thread Operation

The accelerometer thread uses the osSignalSet and osSignalWait functions to control its execution. osSignalSet is called inside the HAL_GPIO_EXTI_Callback, which means that it is executed whenever the accelerometer enables its Data Ready interrupt (25 times per second). osSignalWait is the first instruction inside the loop of the accelerometer thread, and it is called with the osWaitForever parameter. Using this set up we guarantee that the loop of the

accelerometer thread is only executed 25 times per second. We also guarantee that the thread is sleeping when the accelerometer does not have new values available.

When the thread is awake it acquires new X, Y and Z acceleration measurements by executing LIS3DSH_ReadACC. It then calibrates the measurements, filters them using their corresponding one-dimensional Kalman filters, and uses them to calculate the pitch and the roll with equations (14) and (15). One of these two values is shared with the display thread depending on the mode of operation, as we will see later on.

4.1.3 LED Thread

4.1.3.1 LED Patterns

Following the requirements specified for this project, we configured the 4 LEDs included on the Discovery board so that they can be used to display different patterns based on values entered by the user.

In total we had five different patterns: LEDs turned off, LEDs turned on with PWM, LEDs displaying a circular pattern (clockwise or counter clockwise, depending on the sign of the frequency value entered by the user), LEDs displaying a zigzag pattern (also clockwise or counter clockwise) and LEDs pulsing with progressively increasing and decreasing brightness. The codes the user could enter through the smartphone and their associated patterns and features are summarized in table 6.

Table 6: LED pattern codes and their features.

Pattern number	Pattern	Input variable
0 (Default)	LEDs OFF	-
1	LEDs ON	Integer between 1 and 10: Determines the brightness of the LEDs. • Brightness: 1 is minimum brightness, 10 is maximum brightness.
2	Rotating LEDs	Integer between -10 and 10: Determines the speed and direction of rotation. • Speed: 0 is the slowest, 10 and -10 are the fastest. • Direction: Positive values result in clockwise rotation, negative values result in counter clockwise rotation.
3	Custom LEDs	Integer between -10 and 10: Determines the speed and direction of rotation. • Speed: 0 is the slowest, 10 and -10 are the fastest. • Direction: Positive values result in clockwise rotation, negative values result in counter clockwise rotation.

4	Pulsing LEDs	Integer between 1 and 10: Determines the speed of pulsation of the LEDs. • Speed of pulsation: 1 is the slowest, 10 is the fastest.

4.1.3.2 PWM Configuration

In order to enable PWM, which is used by 2 of the 5 patterns, we began by configuring external timer TIM4. Since the frequency received by TIM4 is 84 MHz, we set the Prescaler property to 1 and the Period property to 8399, which allowed us to work with a PWM frequency of 10 kHz. We then configured the timer so that it could be used for PWM by setting the OCMode property to TIM_OCMODE_PWM1. Additionally, we set the Output Compare Pulse property to a width value calculated using equation 17, and we also initialized all the channels of TIM4 using the HAL_TIM_OC_ConfigChannel function.

We then configured the GPIO pins connected with the 4 LEDs so that they could be used for PWM. First, the GPIO Mode property of these pins was set to GPIO_MODE_AF_PP instead of GPIO_MODE_OUTPUT_PP. We set them to Alternating Function (AF) mode so that the timer could have full control over them. We then set the GPIO Alternate property of the pins to GPIO_AF2_TIM4, which connected them to the PWM function produced by TIM4.

Since only 2 patterns use PWM, we made a function that executes the configuration steps described in this section. This allows the system to switch the configuration of the LED pins depending on the pattern.

4.1.3.3 Thread Operation

Unlike the other 3 threads, the LED thread does not use osSignals or interrupts to control its execution. It simply uses the osDelay function to sleep for amounts of time that depend on the pattern and frequency input values.

Every LED pattern function, except the default one (LEDs turned off), has a loop inside of it. The system only breaks out of these loops when the pattern or frequency values are modified by the user. When this occurs, the system switches to the loop that corresponds to the pattern entered by the user, or it returns to the same loop but with the new frequency value. For the default pattern, since the thread does not have to wake up to change the state of the LEDs, it simply wakes up every 250 ms to check if the pattern or frequency values have been modified.

4.1.4 Communication Thread

4.1.4.1 Sending Instructions from the Master to the Slave

In order to send instructions from the Nucleo board (master) to the Discovery board (slave) we used a system of GPIO pins to compose and receive binary numbers. The steps involved in completing this system were:

- Configuring 5 GPIO pins in output mode on the master, so that it can compose binary numbers ranging from 00000_2 (0_{10}) to 11111_2 (31_{10}).
- Configuring 5 GPIO pins in input mode on the slave, so that it can read the binary numbers composed by the master.
- Configuring an additional GPIO pin (C0) in output mode on the master, which is enabled when it wants the slave to read an instruction.
- Configuring a GPIO pin (E2) in rising interrupt mode on the slave, which triggers an interrupt whenever the master wants it to read an instruction.

With these 12 pins it is possible to transmit 32 different instructions from the master to the slave. The pins used to compose and receive the binary numbers are presented in table 7, while the instruction codes and their meanings are presented in table 8.

Table 7: GPIO pins used to compose and receive instructions.

Bit	Master (Output Mode)	Slave (Input Mode)
1 (LSB)	C8	C9
2	C6	C8
3	C13	C6
4	C5	C5
5 (MSB)	C4	C4

Table 8: Instruction codes and their meanings to the slave.

Instruction Code	Function
00000_2 (0_{10}) to 01010_2 (10_{10})	These instructions set the PWM and rotation frequency of the LEDs to numbers between 0 and 10
01011_2 (11_{10}) to 10100_2 (20_{10})	These instructions set the PWM and rotation frequency of the LEDs to numbers between -1 and -10
10101_2 (21_{10})	Turns all LEDs off
10110_2 (22_{10})	Turns all LEDs on
10111_2 (23_{10})	Makes the LEDs follow a circular pattern
11000_2 (24_{10})	Makes the LEDs follow a zigzag pattern
11001_2 (25_{10})	Makes the LEDs glow while changing their brightness
11101_2 (29_{10})	Makes the slave send the roll to the master via SPI
11110_2 (30_{10})	Makes the slave send the pitch to the master via SPI
11111_2 (30_{10})	Makes the slave send the temperature to the master via SPI

Figure 12 illustrates the connections used to send instructions between the master and the slave in a more intuitive format:

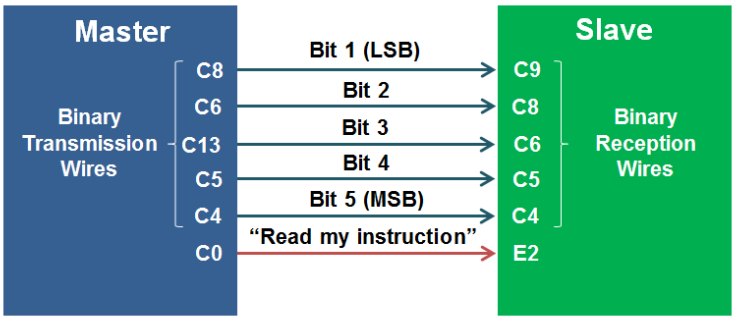

Figure 12: Connections used to transmit instructions from the master to the slave. Note that the arrows point from pins configured in output mode to pins configured in input mode.

4.1.4.2 Sending Measurements from the Slave to the Master

In order to send temperature, pitch and roll measurements from the Discovery board to the Nucleo board, we configured SPI3 on the Nucleo board in master mode and SPI2 on the Discovery board in Slave mode. Note that we did not use these SPI modules to transmit instructions from the master to the slave because of several synchronization issues we encountered during the development process, which made it necessary to implement the scheme described in section 4.1.4.1.

Since the system clock of the master is set at a frequency of 32 MHz, we used a Baud Rate Prescaler of 256 to operate the SPI modules at the lowest possible frequency, which is sufficient for our purposes. Aside from configuring the SCK, MISO and MOSI pins of the two SPI modules being used, we also configured a "Data Ready" pin (E7) on the Discovery board. As the slave, the Discovery board was programmed to enable this pin when it is ready to transmit the data the master requested. This raises an interrupt in the Nucleo board through pin C1, which was configured in rising interrupt mode. When this interrupt occurs, the master begins transmitting dummy bytes and checking the register in which received bytes are stored, which in turn clocks the slave, allowing it to transmit the desired measurement. This process is described in more detail in section 4.1.4.3.

Table 9 presents the pins used to support the SPI communication, while figure 13 illustrates these connections in a more intuitive format:

Table 9: Pins used to support SPI communication between the master and the slave.

Function	Master (Output Mode)	Slave (Input Mode)
SCK	C8	C9
MISO	C6	C8

MOSI	C13	C6

Figure 13: Connections used to transmit measurements from the slave to the master using the SPI communication protocol.

4.1.4.3 Thread Operation

Supporting the communication between the Nucleo board and the Discovery board requires a total of 10 wires with the schemes described in the previous 2 sections. This number could have been reduced to 4 wires with a complete, bi-directional implementation of the SPI communication protocol. This implementation was not achieved, however, because of issues related to synchronization, as noted in section 4.1.4.2. The lack of synchronization caused the bytes sent between the master and the slave to be shifted, corrupting the measurements. But despite the higher number of wires used by our system, the communication between the Nucleo board and the Discovery board is fully reliable and responsive. The steps executed during a complete exchange are as follows:

- The master begins by composing a binary number with the pins used to issue instructions. For the purpose of this explanation, let's assume the master enables all 5 pins, composing the instruction code used to tell the slave that it wants the temperature.
- The master then enables the "read my instruction" pin, which triggers an interrupt on the slave that makes it read the 5 wires used to receive instructions.
- The slave composes the received instruction and determines that the master is requesting the temperature.
- The slave takes the most recent temperature measurement, which is a floating point number, and divides it into 4 bytes. It then places the first byte in the Transfer Register, and it enables the "data is ready" pin. While it waits for the master to begin clocking it, it checks the

TXE flag to know when the Transfer Register is empty. When it is, it fills it with the next byte.

- The "data is ready" line triggers an interrupt that makes the master begin transmitting dummy bytes, which clocks the slave. It continues to do this while the RXNE flag is low, which means that it stops when the Receive Register is not empty. When this occurs, it retrieves the received byte and beings transmitting dummy bytes again. This process is repeated until 4 bytes have been received.

Note that this was achieved using transmit and receive functions written by ourselves, instead of with the HAL_SPI_Receive and HAL_SPI_Transmit library functions. Also note that these operations are only executed when the master enables the "read my instruction" pin, which means that the communication thread is sleeping when the master has not requested anything.

4.1.5 Thread Priorities and Shared resources

The temperature sensor, accelerometer and communication threads were given high priorities using the osPriorityHigh parameter. The LED thread was given a normal priority using the osPriorityNormal parameter. We chose to give the temperature sensor, accelerometer and communication threads higher priorities because they are critical for the operation of the system. If the communication thread is interrupted while the master is transmitting dummy bytes, the master will have to wait until the communication thread wakes up, which delays the operation of its other services. Similarly, without the values the temperature sensor and accelerometer threads provide, the system cannot update what is being displayed on the smartphone.

In terms of the shared resources, there are only 5 in the entire system. These variables, and the way they are employed by each thread are summarized in table 10 below:

Table 10: Shared resources and their uses by each thread.

	Communication Thread	**LED Thread**	**Accelerometer Thread**	**Temperature Sensor Thread**
measured_temperature	Read	-	-	Write
Pitch	Read	-	Write	-
Roll	Read	-	Write	-
LED_Frequency	Write	Read	-	-
LED_Pattern	Write	Read	-	-

As can be observed in the table above, each resource is only shared by two threads and each one is only written by one the two threads that share it. It is for this reason that our system does not require mutexes in order to guarantee thread synchronization.

4.2 Master

The master of the system is the Nucleo board, which supports the BLE connection with the smartphone through the BLE shield. Its operation and the Android application that runs on the smartphone are described in sections 4.2.1 and 4.2.2.

4.2.1 BLE Server

The BLE server is implemented on the Nucleo board with the BLE Shield. The expansion board has a BlueNRG network processor that communicates with the main board using SPI1 in interrupt mode. The main processor can access the BlueNRG host and controller using the application command interface (ACI) [15]. This architecture is illustrated in figure 14.

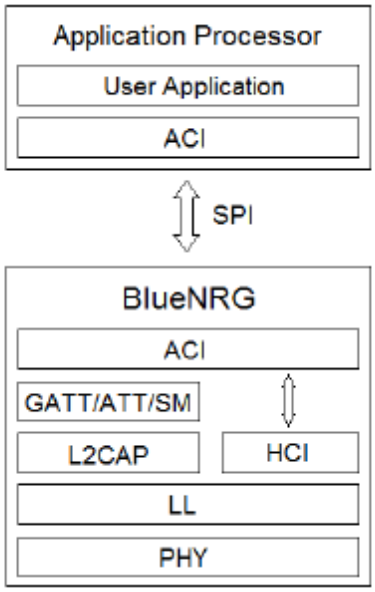

Figure 14: GATT transaction process.

The expansion board is first initialised using the following library functions: BNRG_SPI_Init(), HCI_Init() and BlueNRG_RST(). The board can then be configured as a server, and the required services (accelerometer, temperature, double tap and LED) can be added using the aci_gatt_add_serv() function. Each characteristic is then added with its appropriate properties using the aci_gatt_add_char() function. The pitch, roll and temperature characteristics have read and notify properties. The double tap characteristic only has the notify property, while the intensity and pattern characteristics of the LED service have the write property. Each of the characteristics and services are added using a unique UUID that will also be implemented in the android application in the form of a lookup table [15].

The aci_gap_set_discoverable() function is called to start advertising the BLE server until a connection is made. After that, the HCI_Process() function must be called

periodically to process the various events and trigger the right callback functions. The aci_gatt_update_char_value() function is used whenever a new value is stored to a characteristic. The data to be updated must first be broken down into an array of bytes. Whenever a read request is made by the client, the Read_Request_CB() function is triggered and inside it the aci_gatt_allow_read() function is called to allow a read operation. Whenever a characteristic is written by the client, the Attribute_Modified_CB() function is called and depending on the data and characteristic handle, the right LED operation commands will be sent to the Discovery board using the binary communication system.

4.2.2 Android Client Application

The temperature The android application is based on the sample android application [16]. When the application is started, the first activity is the device scan activity. By pressing the scan button on the top right menu, the device scans for a period of 5 seconds for any BLE peripheral devices nearby. Whenever a device is detected, it is added to a list in the middle of the screen. Figure 15 illustrates the device scan activity.

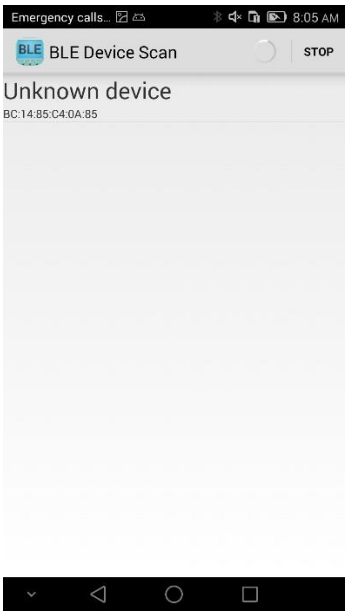

Figure 15: Device scan activity.

By pressing on any of the devices listed, the application will respond to the advertisement and pair up with the device. When a device is selected, the application will jump into the device control activity. Figure 16 illustrates the device control activity.

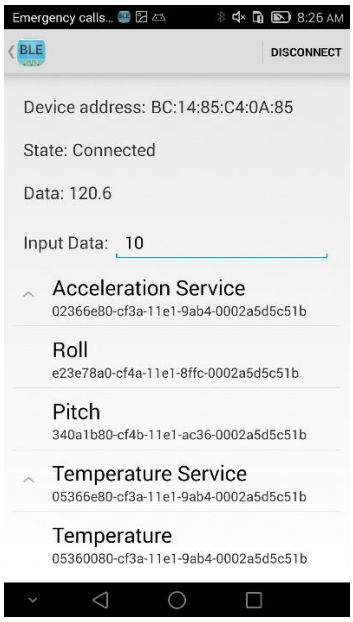

Figure 16: Device control activity.

Upon connection with the selected device, a BluetoothLE service will be created and run in the background. The application will try to detect and list any services and characteristics in the form of an expendable list. The application will try to match any UUID it finds with a lookup table of UUID's to see if any known characteristics or services are found.

If any characteristic with the read property is pressed, such as roll, pitch or temperature, a read request will be made using the readCharacteristic() function [17]. Whenever a data change is received by the background service, it will broadcast an update with the new value. Inside the device control activity, a receiver will take and display that data in the form of a floating point number in the data text field. Additionally, these characteristics and the double tap characteristic have the property of notify. When any of them are pressed twice, descriptor.setValue(BluetoothGattDescriptor.***ENABLE_NOTIFICATION_VALUE***) [18] is executed to write to the configuration of the pressed characteristic on the server side, prompting notifications. When notifications are enabled, the server will continuously send new data that will update the data text field. Additionally, whenever the double tap characteristic has been changed, which occurs when the Discovery board is tapped twice, a notification will be trigger. The notification will happen even if the application is idle. Figure 17 illustrates how double tap notifications are reported in the application.

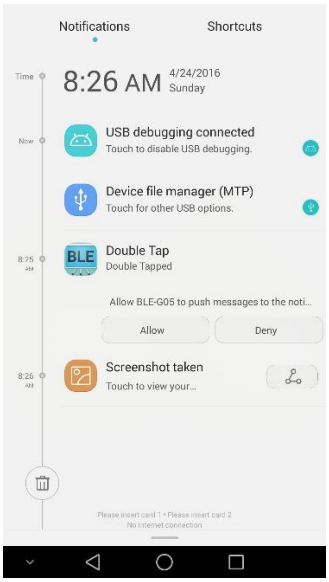

Figure 17: Double tap notification.

For transmitting data to the server from the client, a number can be given to the input data text field. After doing this, pressing any of the LED service characteristics will write the value in the data text field to the pressed characteristic with the writeCharacteristic() [17] function.

5. Testing and Observations

The final system, including all of its components and connections, is illustrated in figure 18 below:

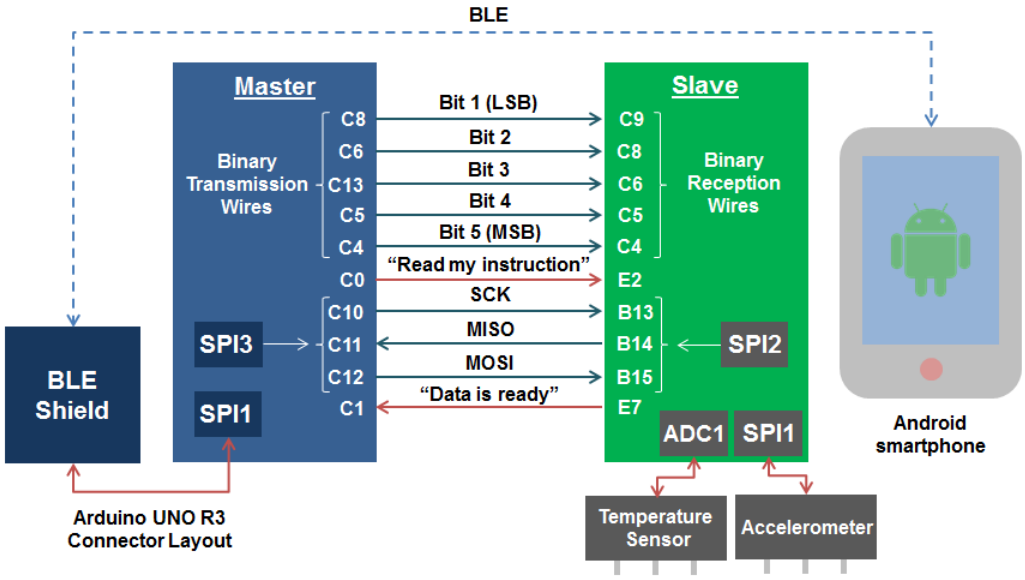

Figure 18: Schematic of the system components and their connections.

Considering it is a complex multicomponent system, we decided to begin testing it by verifying the proper operation of the different services supported by the Android application. The testing process of each service is detailed below:

- Temperature service: The temperature service was tested by comparing the values displayed on the screen of the smartphone with the original values sent by the Discovery board, which we observed by adding the measured_temperature variable to Watch 1. The clear match between the sent and displayed values certified that the floating point numbers were being decomposed into bytes, transmitted, and recomposed appropriately. We also heated the processor of the discovery board with friction to observe the temperature changing more drastically on the screen.
- Accelerometer service: The accelerometer service was tested following the same procedure used to test the temperature service, except it was done with the pitch and roll values. This stage of testing also involved switching between displaying the temperature, pitch and roll, to make sure that the system updated the values being displayed appropriately. Additionally, the board was rotated to see the pitch and the roll values displayed on the smartphone range from 0 to 360°.
- Control of the LEDs: Testing the control of the LEDs involved inputting every possible combination of valid pattern and frequency parameters through the smartphone. Invalid input values were also included to certify that the system ignored them. The on and off patterns were simple to test because they are not affected by the frequency parameter. The rotate and zigzag patterns changed their speed depending on the frequency parameter, and they also changed direction depending the sign of the frequency parameter. The glow pattern changed the brightness of the LEDs progressively, with the rate of change of the brightness defined by the frequency parameter. The transitions between the different patterns were smooth, and the system responded immediately to the values entered through the smartphone.
- Double tap service: Testing the double tap service involved 3 stages. In the first stage we certified that notifications are displayed when the Discovery board is double tapped while it lies flat with its Z vector pointing down. We then certified that taps spaced by 0.8 seconds do not cause notifications to be displayed, since we consider them to be independent taps. Finally, we verified that changing the orientation of the board does not cause notifications to be displayed either.

After verifying the proper operation of the system from a high level perspective, we used Keil's event viewer tool to observe the behaviour of the 4 threads supported by the discovery board. We first noticed how the temperature sensor thread is executed 100 times per second, while the accelerometer thread is executed 25 times per second, which matched our expectations. The number of times the LED thread is executed per second depended on the frequency parameter, with 4 times per second being the slowest frequency. As for the communication thread, the number of times it executed per second depended on the frequency at which the

master sent new instructions. We decided to have the master ask for temperature, pitch and roll values in sequence with 100 ms delays between each request. Note that we also tested using 10 ms delays, which functioned appropriately as well. With these tests we ensured that no threads execute all the time.

6. Timeline and Work Breakdown

Stephen Cambridge: In the week of March 28th I wrote a routine for a bonus feature that would allow a custom, arbitrary LED pattern and speed to be displayed on the Discovery board. This feature was unused, however, due to limitations in inter-device communication. In the week of April 4th I worked with Diego to get reliable SPI communication between the Discovery board and the Nucleo board. I also worked with Saki on implementing PWM for the LEDs. The following week Saki and I finished the LED and double-tap functionality, and I also worked with Saki and Diego on assembling the final project.

Diego Macario**:** In the week of March 21st I began familiarizing myself with the SPI communication protocol by reading its associated documentation and by attempting to make two Discovery boards communicate via SPI. I decided to start with two Discovery boards instead of a Discovery board and a Nucleo board because I felt more comfortable working with a platform that I understood well. By the beginning of the week of March 28th I succeeded in getting the two Discovery boards to communicate using the library functions HAL_SPI_Receive and HAL_SPI_Transmit, and I immediately began working on getting a Nucleo board and a Discovery board to communicate via SPI. Throughout this week and the week of April 4th I worked with Stephen to fix the synchronization issues that corrupted the data sent between the boards. During this time we wrote our own transmit and receive functions, which we used to reliably send measurements from the slave to the master, but we still observed issues when sending instructions from the master to the slave. On April 11th it was announced that we could work with protocols different from SPI, and seeing how we had not been able to resolve the synchronization problems we decided to combine our one-directional SPI with a binary system. I worked with Wen on implementing the binary system, and I also worked with Saki and Steve on integrating the different components.

Saki Kajita: In the week of March 21st I started working on the double-tap feature by collecting and analyzing data from the accelerometer. In the week of March 28th I began implementing the four main LED patterns (LEDs off, LEDs on, LEDs displaying a circular pattern and LEDs pulsing with PWM). Stephen and I finalized these features in the week of April 4th. On that same week I also worked with him on finalizing the double-tap feature, and in the week of April 11th I worked with him and Diego on assembling the final project.

Wen Bo Zhang: On the week of March 21st I began working on the android application. Basing myself on the sample application provided on the Android website, I familiarized myself with the GAP advertising procedures and the GATT 2-way communication. In the first half of

the week, by working with the sample project by STMicroelectronics for the Nucleo board, I managed to test the scanning for devices function and managed to establish a connection between the board and the smartphone application. In the following week of the 28th of March, I implemented the temperature, accelerometer and double tap services for both the board and the application. I was able to make read requests to the board and enable notifications. I was also able to receive and display data from the board in the form of floating point number. On the week of April 11th, during the first half of the week, I worked with Diego to implement a custom communication protocol between the Nucleo and Discovery boards. In the second half of the week, I implemented the LED service that had the property of write. I tested the write function and made sure the phone could send a single byte at a time to the Nucleo board. I also made a notification display whenever a double tap has been detected.

7. Conclusion

The system detailed in this document explores the interactions between embedded peripherals and a smartphone device. The challenges encountered during the implementation of the SPI communication protocol proved complex enough to warrant a change of direction in our design. The combination of the one-directional SPI with a binary communication system resulted in the addition of 6 wires to the design, but allowed us to finalize a fully reliable system. The double tap detection algorithm also proved to be effective, ignoring single taps and avoiding false notifications when the board is tilted. The different services are responsive, and they can be tuned to provide updates at different frequencies. The reduction of the shared resources down to 5 variables also benefited the system by eliminating the need for mutexes in the slave's firmware.

8. References

[1] A. Suyyagh, *Lab 2: Sensor Data Acquisition, Digitizing, Filtering, and Digital I/O*, 1st ed. ECSE 426: Microprocessor Systems, 2016, p. 2.

[2] A. Suyyagh, *Lab 3: MEMS Accelerometer, Timers and Interrupts*, 1st ed. ECSE 426: Microprocessor Systems, 2016, p. 1.

[3] *Tilt angle application notes*, 1st ed. STMicroelectronics, 2010, p. 6-7, 9, 12-14.

[4] A. Suyyagh, *A Generic Keypad and LCD Tutorial*, 1st ed. ECSE 426: Microprocessor Systems, 2016, p. 7-9.

[5] *RM0090 Reference Manual: STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs*, 1st ed. STMicroelectronics, 2011, p. 229.

[6] *STM32F405xx STM32F407xx: ARM Cortex-M4 32b MCU+FPU, 210DMIPS, up to 1MB Flash/192+4KB RAM, USB OTG HS/FS, Ethernet, 17 TIMs, 3 ADCs, 15 comm. interfaces & camera*, 6th ed. STMicroelectronics, 2015, p. 38, 132-133.

[7] A. Suyyagh, *Lab 1: One-Dimensional Kalman Filter*, 1st ed. ECSE 426: Microprocessor Systems, 2016, pp. 1-2.

[8] "Filtering Sensor Data with a Kalman Filter", *Interactive Matter Lab*, 2009.

[9] A. Suyyagh, *Programming NVIC, Peripheral Timers and External MEMS Sensors*, 1st ed. ECSE 426: Microprocessor Systems, 2016, p. 10-11, 14-15.

[10] T. Majerle, *STM32F4 PWM tutorial with TIMERs*, 2014. [Online]. Available: http://stm32f4-discovery.com/2014/05/stm32f4-stm32f429-discovery-pwm-tutorial/. [Accessed: 22-Mar-2016].

[11] *UM1873 User manual*. STMicroelectronics, 2015, p.10-12.

[12] "GAP | Introduction to Bluetooth Low Energy | Adafruit Learning System",*Learn.adafruit.com*, 2016. [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap. [Accessed: 24- Apr- 2016].

[13] "GATT | Introduction to Bluetooth Low Energy | Adafruit Learning System",*Learn.adafruit.com*, 2016. [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt. [Accessed: 24- Apr- 2016].

[14]"Analog Sampling Basics - National Instruments", *Ni.com*, 2016. [Online]. Available: http://www.ni.com/white-paper/3016/en/#toc3. [Accessed: 22- Mar- 2016].

[15] *UM1755 User manual*. STMicroelectronics, 2015, p.15, 32, 78, 109.

[16] "BluetoothLeGatt | Android Developers", *Developer.android.com*, 2016. [Online]. Available: http://developer.android.com/samples/BluetoothLeGatt/index.html. [Accessed: 24- Apr- 2016].

[17] "BluetoothGatt | Android Developers", *Developer.android.com*, 2016. [Online]. Available: http://developer.android.com/reference/android/bluetooth/BluetoothGatt.html. [Accessed: 24- Apr- 2016].

[18] "BluetoothGattDescriptor | Android Developers", *Developer.android.com*, 2016. [Online]. Available: http://developer.android.com/reference/android/bluetooth/BluetoothGattDescriptor.html. [Accessed: 24- Apr- 2016].