

Lab 4:

Multithreaded, interrupt-driven sensor reading and peripheral control

ECSE 426 Microprocessor Systems

Group 18

Chuan Qin 260562917

Wei Wang 260580783

March 21th, 2016

Table of Contents

| | | |
|-------|--|----|
| 1 | Abstract | 3 |
| 2 | Problem Statement | 3 |
| 3 | Theory and Hypothesis | 4 |
| 4 | Implementation | 5 |
| 4.1 | Temperature mode..... | 5 |
| 4.2 | Accelerometer mode | 6 |
| 4.3 | Kalman Filter..... | 7 |
| 4.4 | Four Digit Seven-segment Display | 8 |
| 4.4.1 | Hardware Timer | 10 |
| 4.4.2 | Temperature Mode Display | 10 |
| 4.4.3 | Accelerometer Mode Display | 11 |
| 4.5 | Alphanumeric Keypads | 12 |
| 4.6 | Real-time Operation System Threads | 14 |
| 5 | Testing and Observations..... | 16 |
| 5.1 | Kalman Filter's parameters | 16 |
| 5.2 | Threads Observations | 18 |
| 6 | Conclusion | 20 |
| 7 | Appendix | 21 |
| 8 | References | 24 |

1 Abstract

The purpose of this experiment is design multithreaded programs as a component of the system, which required two working sensors (Analog-to-digital converter and accelerometer) and software modules. Since the readings of both sensors includes different levels of noise, the Kalman filter with associated sets of parameters is in use for minimizing the noisy signals. The designed program need to communicate with sensors and a display device. The CMSIS-RTOS API provides software component for thread management. Users are allow to select display mode with keypad switches. The applications of both sensors are responsible for reporting when data is ready through sending and receiving signals mechanism. The four digits 7-segment display performs the responses obtained from ADC or accelerometer according to user command.

2 Problem Statement

Several operating functions appear to execute simultaneously. In fact, only one thread of execution is being processed at a time, and quickly switching between the threads provides the illusion of concurrent executions. Multithreading is an efficient approach for the real-time system design. This project aimed at designing a multithreaded program that monitors the temperature of the chip, detects the tilt orientation of the board through three-dimensional Cartesian coordinate, and converts the results before displaying the temperature and angle values using four digits 7-segment display.

First, the mode A is to digitize temperature readings in analog voltage retrieved from the sensor at a particular frequency using the analog to digital converter (ADC). During the conversion, there are few forms of noises are introduced, including electromagnetic noise, thermal noise, and quantization noise. Once the conversion is completed, a software filter is implemented to reduce noisy signal in order to increase the accuracy of temperature values. The data readings performance uses a four digits 7-segment display. The operating process's temperature is increasing smoothly, and when the displayed values on 7-segment display keep flashing on and off, it represents an alarm indicating that the temperature is too high and crosses a threshold.

Second, the system calculates the orientation of the board at a frequency of 25 Hz using the sensor LIS3DSH. The angle can be determined based on the calibrated parameters and the measurements of the acceleration in three dimensions through the accelerometer. Every time the data of accelerometer is ready, an interrupt in the processor is generated by the MEMS sensor. The position sensor is unstable; and by using filter, it eliminates the uncertainties of position's representation. The roll and pitch angles of the board can be performed using four digits 7-segment display. The orientation outputs keep blinking on and off when the temperature sensor notice that the processor is overheating even the system is currently at accelerometer mode.

Last, two main modes are controlled using the switches on the keypad. The keypad scanner must detects whether a button is pressed, identifies which button is entered, and generates an output associated to the pressed key. It is important to generate only one signal output code when the key is pressed. Also, it needs to avoid multiple press key at the same time. The system employs multiple

threads of functions such as temperature sensor, position sensor, display, and keypad. The thread switches need to be rapid enough to have an appearance of all functions working concurrently. There are few challenges encountered while dealing with the specifications listed above. More specifically, they are converting ADC measurement from voltage to degree Celsius, calibrating the position sensor, finding the appropriate set of Kalman filter parameters, eliminating the keypad debounce, determining the threads, designing overheating system alarm, and setting the priority of threads.

3 Theory and Hypothesis

Temperature meter and accelerometer are expected to have evolved in terms of steady, accuracy, and reliability as a consequence of need for enhanced technology. Ideally, the digital output of ADC should remain constant until it reaches a transition region while the analog input is increasing. In fact, for the temperature sensing system, ADC generates a certain amount of noisy signal during each data conversion. On the other hand, a tri-axial accelerometer represents acceleration as a voltage, and few noises come from converting the motion into a voltage signal. In this case, different levels of noise from different sensors affect their measurement accuracy, and therefore both sensors need a different set of state parameters for Kalman filter in order to minimize the effect of noisy signals from ADC measurement and from 3-axis measurement of position sensor.

All the sampling data given by ADC are in voltage format. By knowing the reference voltage, the supply voltage, and a 12-bit analog-to-digital converter, the formulas (6) in section 4.4.2 can be applied to convert temperature sensor output to degrees Celsius. With a 3-axis accelerometer, the roll and pitch tilt angle can be calculated according to the x, y, and z values of the board. Since the 7-segment display provided in this experiment can only display 3 digits with a degree symbol, temperature values and tilt angles must be converted before performing the data on the display. During both modes' display, we give a specific period of time for display to be on and off repeatedly when the system detects that the temperature crosses the threshold, which is set to be 35°C in this case.










Two keys, A and B, are used to tell display to perform temperature mode and accelerometer mode, respectively. The key oscillates multiple times before settling in a very short time; it occurs while pressing the switch and releasing the switch. Then, the program captures all the oscillations and considers them as multiple presses instead of one. To clean up switch bounce, we set a flag for keypad to make sure that one key code is recorded by the program when the key is pressed and held.

We identify that the program has four major tasks to be performed concurrently as following:

- Scan the keypad to decode the pressed button for switch
- Obtain the temperature of processor from ADC
- Calculate the orientation based on the three axes values of the board
- Show the output of temperature, roll tilt angle, or pitch tilt angle on the display

Each function is considered as a thread of execution. The operating system scheduler is responsible for deciding which thread to run and when according to its thread priority. Data dependency is one of the challenges of multithreading; more specifically, ADC, accelerometer, and display are dependent upon the results of the keypad, so the program need to assure that threads are being executed in the proper order. The program is expected to run each threads and alternate rapidly. In the table below, it shows a basic idea of chronological trace of how we expect the threads to execute. For temperature mode, we do not need to consider accelerometer. However, for the position sensor, we also need to run ADC for checking whether the temperature is higher than the threshold. The time of execution varies for each function.

Table 1 – Ideal real-time trace of all the threads for two different modes

| Threads | Temperature Mode | | Accelerometer Mode | |
|----------------------|---|---|---|---|
| idle |  | |  | |
| Thread_Keypd | |  | |  |
| Thread_ADC | |  | |  |
| Thread_Accelerometer | | |  | |
| Thread_Segment | |  | |  |

4 Implementation

4.1 Temperature mode

The system generates many analog signals including temperature, which need to be converted into digital data using ADC peripheral. It is easier to use standard peripheral functions. First, we enable ADC clock and set prescaler to allow ADC to work at the board clock frequency divided by 6 (84MHz / 6), which is equal to 14MHz [1]. Also, the input voltage is converted into a 12-bit number given a maximum value of 4096, so the resolution should be 12-bit.

```
ADC1_Handle.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV6;
ADC1_Handle.Init.Resolution = ADC_RESOLUTION_12B; [3]
```

Next thing is to select single conversion mode for ADC [3]. It means that ADC does only one conversion and then stop in this case. The readings given by ADC Conversion is stored into a 16-bit register. Once it is done, the End of Conversion flag (EOC) is set. So, ADC will no longer do any conversion until we ask for. We right justified that data which indicates that LSB is located at the rightmost bit [2].

```

ADC1_Handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
ADC1_Handle.Init.NbrOfConversion = 1;
ADC1_Handle.Init.DataAlign = ADC_DATAALIGN_RIGHT;

```

Since the single conversion mode is in use, we can disable other modes such as scan mode, continuous conversion mode, and discontinuous conversion mode. Then, we need to select ADC Channel 16 because it is where temperature sensor is internally wired to [2], [3].

4.2 Accelerometer mode

The board generates several kinds of analog signals including gravity. To collect and use the data of gravity generated by accelerometer, we apply the functions provided by the LIS3DSH driver to configure tri-axis MEMS accelerometer. In the driver, the sensor is connected to SPI (Serial Peripheral Interface) interface. The SPI1 here is the master that controls that clock line and generates data for sending to the slave.

All the three axes (X, Y, and Z) are enabled to retrieve gravity data. The results will continuously updating, so the low and high registers can be updated until all values are ready. We set the full scale to 2g because the smaller the full scale is, the more accurate result will be generated. The sampling rate is set to 25Hz as required.

```

LIS3DSH_Ini.Axes_Enable = LIS3DSH_XYZ_ENABLE;
LIS3DSH_Ini.Continuous_Update = LIS3DSH_ContinuousUpdate_Enabled;
LIS3DSH_Ini.Full_Scale = LIS3DSH_FULLSCALE_2;
LIS3DSH_Ini.Power_Mode_Output_DataRate = LIS3DSH_DATARATE_25;

```

We add an interrupt signal of EXTI0 to the sensor; it notifies the system every time the data is ready. The first step to implement the interrupt is to define the IRQ Handler of EXTI0, which pins are connected to line zero, in the file `stm32f4xx_it.c`. The INT1 pin is connected to GPIO Pin PE0 in the board, and therefore, this signal from sensor indicates a new data is ready to be processed. Each time the board starts, the system know what the EXTI0 interrupt is from the file `startup_stm32f40xx.s`.

Then, we overwrite the Callback function for this interrupt. It reads the raw data from accelerometer, calibrates the coordinates, and filters out the measurements error for each axis value. Then next steps is to enable the data ready interrupt of this sensor by setting the signal to active high and the type to pulse. The last thing is to enable the interrupt using the NVIC, and then, we set interrupt of position sensor to be the highest (pre-emption priority and sub-priority are both zero). The reason is that there is no other interrupts should effect the process of getting raw data from accelerometer.

4.3 Kalman Filter

Kalman filter is an algorithm applied after retrieving readings from ADC and accelerometer for the purpose of minimizing the effects of measurement error. The implementation of this type of filter is based on the immediate last state and the observed physical result.

Table 2 – Kalman filter state structure parameters

| Kalman Filter State Parameters | Variables |
|--------------------------------|-----------|
| Process Noise Covariance | q |
| Measurement Noise Covariance | r |
| Value | x |
| Estimation Error Covariance | p |
| Kalman Gain | k |

The algorithm can be separated into two types: prediction equations and update equations. The first type predicts the current state for the input according to the previous state values as Equation 1 and 2 in [4].

$$p_x = p_{x-1} + q_{x-1} \quad (1)$$

$$k_x = \frac{p_x}{p_x + r_{x-1}} \quad (2)$$

where x presents the state number

The second part is to calculate the filtered result using the estimate and the measurement given by the system. Then, this update process is repeated continuously.

$$x_x = x_{x-1} + k_x(\text{measurement} - x_{x-1}) \quad (3)$$

$$p_x = (1 - k_x)p_x \quad (4)$$

Since the Kalman filter keeps record of the previous state, hence, we need separate states for all the necessary variables such as temperature and tri-axis coordinates. In Figure 1, we had already determine the appropriate set from previous lab. Instead of testing for a small range of temperature, we use hairdryer to heat up the processor for better reading observations, which increases the range up to around 65°C. We know that the initial filter parameters are q, r, p, and k which is equal to 0.07, 3, 0.1, and 0, respectively. The initial x value depends on the first reading temperature. Later in the section 5.1, we need to define the parameters which give the best results. To test different values, we use MATLAB to generate graphs of inputs and outputs.

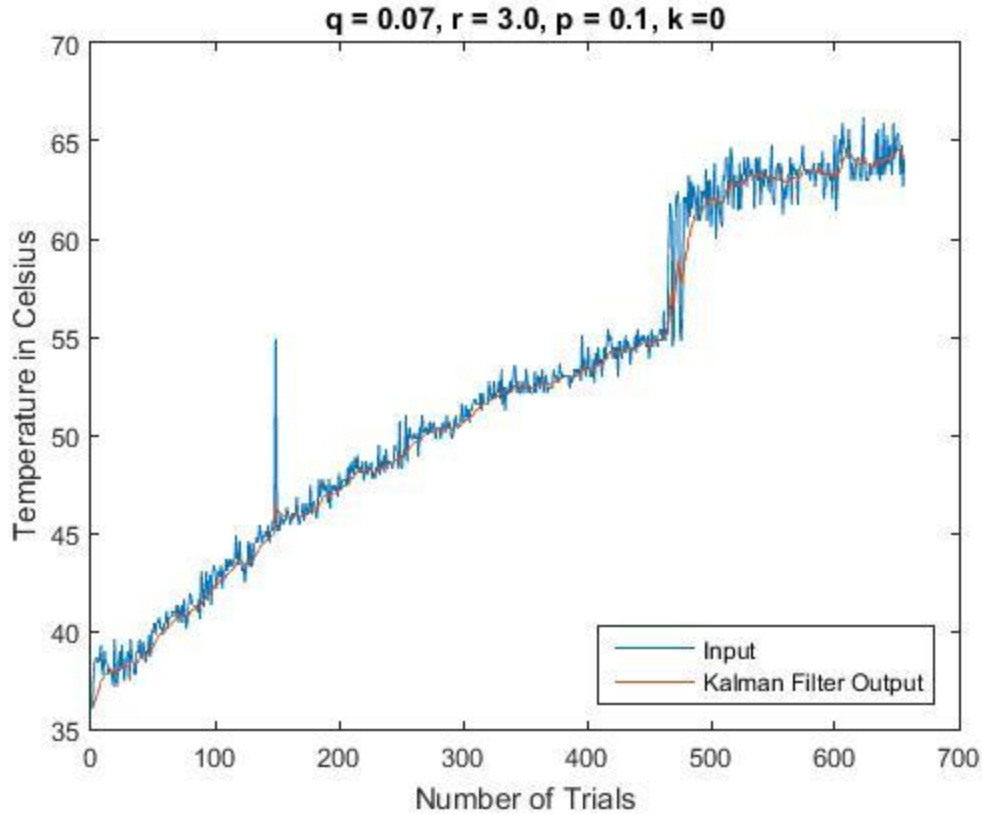


Figure 1 – Performance of Kalman Filter with appropriate parameters for temperature

4.4 Four Digit Seven-segment Display

There are two main steps required before the display: configuring the GPIO for different type of display, and connecting the display pins to its corresponding GPIO pins. There are multiple GPIO pins on the board. These pins can be programmed as input; they can also be used to perform outputs. In order to be able to configure pins of a specific port for different types of display, we must first enable its pin clock. Then, we define the pins' mode to be push-pull output since there is no external voltage applied, activate pull up that configures inputs to be high, select clock speed between range 25MHz to 100MHz, and initialize the pins required for displays.

In the common cathode display, all 7-segment data lines are connected together to ground. Each segment, labelled from A through to G in Figure A-1 [6], can be selected individually by a high signal. More specifically, the segment will be dark when it is set to be '0', and it will be light when it is set to be '1'.

There are 12 resistors of 390 Ohm added while connecting the wire from GPIO pins to the 12 pins (7 segments, one decimal point, and 4 digit) of display. The purpose of the resistor is to protect the display by reducing current flowing to a segment. In Table 4, we set individual data lines to '1' in order to produce various numbers from 0 through 9. As shown in the Figure A-1, we notice that each digit contains a pin for decimal point (DP). We connect DP pin to GPIO Port E Pin₁₄.

According to the values of temperature, there is only one decimal point required for the performance. The DP pin will be light when ones digit is on. Furthermore, we connect a transistor to each common cathode terminals as a switch to specify which digit should be illuminated. Each digit must perform sequentially from left to right because all four digits share the same segment pins. We need to extract each digit from a given output and display it accordingly (Table 3). The delay time between each digit display is controlled using hardware timers instead of using software timers.

Table 3 – The common cathode pins configuration using GPIO Port C

| Terminal | Digit 1 | Digit 2 | Digit 3 | Digit 4 |
|------------|---------|---------|---------|---------|
| Digit | Tens | Units | Tenths | Degree |
| Pin Number | Pin_15 | Pin_1 | Pin_2 | Pin_3 |

Table 4 – Seven-segment display code and pins configuration using GPIO Port E

| Number | A | B | C | D | E | F | G |
|---------------|-------|-------|-------|--------|--------|--------|--------|
| | Pin_7 | Pin_8 | Pin_9 | Pin_10 | Pin_11 | Pin_12 | Pin_13 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| Degree Symbol | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

The temperature threshold is set to be 35°C. While the system is operating, the chip's temperature is smoothly increasing. Once the temperature crosses the threshold, the display is being on and off repeatedly within a short time, so it seems like the output values are flashing. The on-off ratio of the display device 1, which means that the time of output performance is as same as the time of not performing. A period of time is 2 milliseconds. In addition, the device display values for 3 periods, which 6 milliseconds, and turns off for the same time period.

4.4.1 Hardware Timer

A hardware timer of a microcontroller is an independent counter. We need to specify the hardware timer; the timer3 is used for display's delay time. In the beginning, we enable the TIM3 clock, and then, we initialize the timer to run with a frequency divider, known as prescaler, of 84, and a period of 100. The frequency for the counter is 1 MHz (84 MHz / 84). Next thing is to select the counting mode for operation. In this case, we want the timer to count from zero to its maximum value at the given speed. The timer will automatically reset to zero once it reaches 100 with 1 microseconds cycle.

```
TIM_Handle.Init.Prescaler = 84 - 1;  
TIM_Handle.Init.CounterMode = TIM_COUNTERMODE_UP;  
TIM_Handle.Init.Period = 100 - 1;
```

After the timer configuration, we start the timer3 with the help of HAL library. Then, we add an interrupt for the timer called "TIM3_IRQn", by setting up the Nested Vectored Interrupt Controller (NVIC), so we receive an interrupt when the timer part is done and the interrupt time is 100 microseconds. This interrupt has the highest priority 0, which is the same pre-emption priority as the accelerometer interrupt but different sub-priority.

```
HAL_TIM_Base_Init(&TIM_Handle);  
HAL_TIM_Base_Start_IT(&TIM_Handle);  
HAL_NVIC_SetPriority(TIM3_IRQn, 0, 1);  
HAL_NVIC_EnableIRQ(TIM3_IRQn);
```

We overwrite the Callback function to count the internal variable for display. Every time the interrupt goes into interrupt service routine (ISR), it increases a variable by one. That variable is considered as a counter for changing display digit. The time delay between each digit performance is set to be 500 microseconds. At this refresh rate, the flickering problem becomes invisible.

4.4.2 Temperature Mode Display

The temperature values converted by ADC are in voltage format. Note that the data returned from ADC is not equivalent to V_{sense} . We need to determine the maximum 12-bit value would occur at the supply voltage of 3V, which can be defined by following equation:

$$V_{sense} = \frac{3000mV}{4095} \cdot value \quad (5)$$

where value = the temperature reading obtained from ADC

We can convert readings into Celsius format using the Equation 6 in [1].

$$Temperature (^{\circ}C) = \frac{V_{sense} - V_{25}}{Average Slope} + 25 \quad (6)$$

*where the reference voltage at 25°C (V_{25}) = 0.76 V
and Average Slope = 0.0025 V/°C*

The sampling frequency for temperature is required to be 100Hz that indicates we obtain a reading from sensor every 10 milliseconds. To accomplish this, we declare an extern variable which is expecting to increase by one when hardware interrupt is triggered. We consider this variable as a counter for amount of interrupts; once it reaches 100 times, the counter resets and the system asks ADC to do conversion. For every iteration, the counter resets and increases according to the interrupts. The system can get a reading from ADC every 0.01 second. Instead of displaying all the converted data from ADC, the display takes the average of ten recent data.

4.4.3 Accelerometer Mode Display

Since the raw data generated by the accelerometer is not accurate, we need to calibrate it using the method of least square to make the error smaller before calculating the angle [7]. The first step is to gather the Cartesian coordinates systems of 6 stationary position in Table 5. After these 6 measurements, we get a 6x4 matrix. More specifically, the first three elements of each row is the data of x, y, and z, respectively. Each entries of the fourth column is equal to 1. The matrix Y is a 6x3 matrix that represents the expected coordinate's values for each 6 stationary positions referred to Document 15 [7].

Table 5 – XYZ coordinates of 6 stationary position

| Expected Measurements (g) | | Actual Measurements (mg) | | |
|---------------------------|----|--------------------------|--------------|-------------|
| | | X | Y | Z |
| X | 1 | 1008.085999 | 14.579000 | 18.849000 |
| | -1 | -996.974025 | -25.986000 | 17.445999 |
| Y | 1 | 22.570000 | 1042.001953 | 53.680000 |
| | -1 | 64.903999 | -1031.754028 | 11.102000 |
| Z | 1 | -20.740000 | -105.469002 | 1004.731018 |
| | -1 | -44.957001 | -12.505000 | -980.208984 |

Table 6 – 12 calibration parameters of matrix X

| Coefficient | Value |
|-------------|--------------------|
| ACC10 | -0.004835453911860 |
| ACC20 | 0.018264113153855 |
| ACC30 | -0.021482451164122 |
| ACC11 | 0.994138083545817 |
| ACC21 | -0.021832092768331 |
| ACC31 | -0.001001098568320 |
| ACC12 | 0.018260880674360 |

| | |
|-------|--------------------|
| ACC22 | 0.961011356119953 |
| ACC32 | -0.021178545356665 |
| ACC13 | 0.012006312335592 |
| ACC23 | 0.044758274939461 |
| ACC33 | 1.006413535148700 |

Then, we get the 4x3 matrix X using Equation 7 which contains the coefficients required to calibrate the gravity's raw data of all axes, each calculated values are recorded in Table 6. Each time when we obtain one raw data from position sensor, we use Equation 8 to calibrate in callback function of EXTI0 interrupt [7].

$$X = [W^T * W^{-1}] * W^T * Y \quad (7)$$

$$\begin{bmatrix} A_{x1} \\ A_{y1} \\ A_{z1} \end{bmatrix} = \begin{bmatrix} ACC_{11} & ACC_{12} & ACC_{13} \\ ACC_{21} & ACC_{22} & ACC_{23} \\ ACC_{31} & ACC_{32} & ACC_{33} \end{bmatrix} \cdot \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} + \begin{bmatrix} ACC_{10} \\ ACC_{20} \\ ACC_{30} \end{bmatrix} \quad (8)$$

After calibrating and filtering the three dimensional coordinates, we choose the method of tri-axis tilt sensing, which has the highest sensitivity among different angles, to calculation the board's orientation. In this experiment, we need to determine both roll and pitch angle in Equation 9 and 10. The rotation around longitudinal axis is called roll, and the motion about side-to-side axis is pitch.

$$Roll = \arctan\left(\frac{A_{x1}}{\sqrt{(A_{y1})^2 + (A_{z1})^2}}\right) \quad (9)$$

$$Pitch = \arctan\left(\frac{A_{y1}}{\sqrt{(A_{x1})^2 + (A_{z1})^2}}\right) \quad (10)$$

4.5 Alphanumeric Keypads

A 4x4 keypad used for switching between temperature mode and accelerometer mode by different keypresses. The keypad used consists of 16 buttons, which are organized in a row and column matrix as shown in Figure A-2. Each row and column of the keypad is associated with a pin, so there are 8 pins in total, which are configured as shown in Table 7, 8. This experiment has to handle switch de-bouncing correctly to avoid registering multiple numbers from a single press. In order to achieve this, we set a flag to avoid the system interpret a key to be entered repeatedly if it is pressed once and held down. For example, every time a key is pressed, the flag is set to be '1', and it becomes '0' when the key is released. When the flag is triggered, keypad scanner generates only one unique code of the pressed button.

Table 7 – Keypad column pins configuration using GPIO Port D

| Column | Column 1 | Column 2 | Column 3 | Column 4 |
|------------|----------|----------|----------|----------|
| Pin Number | Pin_10 | Pin_1 | Pin_2 | Pin_3 |

Table 8 – Keypad row pins configuration using GPIO Port D

| Row | Row 1 | Row 2 | Row 3 | Row 4 |
|------------|-------|-------|-------|-------|
| Pin Number | Pin_9 | Pin_5 | Pin_6 | Pin_7 |

After the keypad scanner detects a pressed button, it outputs the unique code associated to the entered character. The initial state for all buttons is logic '1', and it becomes '0' when the key is pressed. First, we initialize each column bits to be output with logic '0' and each row bits to be input. The system reads the pins from each column and shift left by 4 digits. Second, we set each row bits as output with '0' and each column as input. The program reads the pins and simply concatenate it with the reading from previous step. Then, it decodes the given button according to the character codes shown in the table below. In this experiment, temperature mode is represented as button A and accelerometer mode is button B. Also, the key one and two correspond to the roll and pitch of accelerometer mode, respectively.

Table 9 – Keypad buttons code

| Key pressed | Column | Row |
|-------------|--------|------|
| 1 | 1110 | 1110 |
| 4 | | 1101 |
| 7 | | 1011 |
| * | | 0111 |
| 2 | 1101 | 1110 |
| 5 | | 1101 |
| 8 | | 1011 |
| 0 | | 0111 |
| 3 | 1011 | 1110 |
| 6 | | 1101 |
| 9 | | 1011 |
| # | | 0111 |
| A | 0111 | 1110 |
| B | | 1101 |
| C | | 1011 |
| D | | 0111 |

4.6 Real-time Operation System Threads

We should by now have working devices (keypad, temperature sensor, accelerometer, and display) and software components. Threads are an approach for a system to split all the works into more than one thread that is running concurrently. CMSIS-RTOS provides software functions to support multithreading mechanism; moreover, the API allows defining, creating, and controlling thread functions. In this experiment, the program has several threads to execute, including reading and decoding keypad, measuring the temperature, calculating the tilt angles, and displaying the temperature or orientation of the board with over-heating alarm system based on user's selection. Each of the executions mentioned above is considered as a thread with different priority levels if necessary.

According to the RTX kernel's features, the input frequency of the RTOS kernel timer is 168 MHz with a RTX timer tick interval value of 100 microseconds. We set the timer tick the same frequency as the TIM3 interrupt. By default, the maximum number of user threads running simultaneously is 6, and each thread has its own stack of size 800 bytes beside the stack of the main thread which is 1024 bytes. The application checks the stack overflow at thread switch, although the execution time will slightly be increased because of enabling this feature. Hence, all the threads are initialized to have same priority. It is important to know that greater number indicates higher priority, which is opposite to the setting for interrupt priority. We implements Round-Robin scheduling as an approach to support equal priority threads switching, and each thread executes for 500 microseconds before switching.

The program needs to know which requests make which threads go. We use signals as inter-thread communication method for the purpose of enabling threads to send a flag to others and enabling threads to wait for the flag from others. More specifically, threads for both sensors might wait for unique signals from thread of keypad to process. Then, after data is ready, the threads of sensors send a unique signal to segment thread indicating that output is ready to be displayed. The operations of signal and wait are atomic, meaning the thread will not be interrupted by any other thread until the end. Threads are forced to sleep for 1 millisecond after sending the signal. Since temperature and position sensors are independent functions, there is no need to implement semaphores to access shared resources together.

Table 10 – Threads priority configuration

| Thread name | Priority Type | Number |
|----------------------|-----------------------|--------|
| Thread_Keypad | osPriorityAboveNormal | +1 |
| Thread_ADC | osPriorityNormal | 0 |
| Thread_Accelerometer | osPriorityNormal | 0 |
| Thread_Segment | osPriorityBelowNormal | -1 |

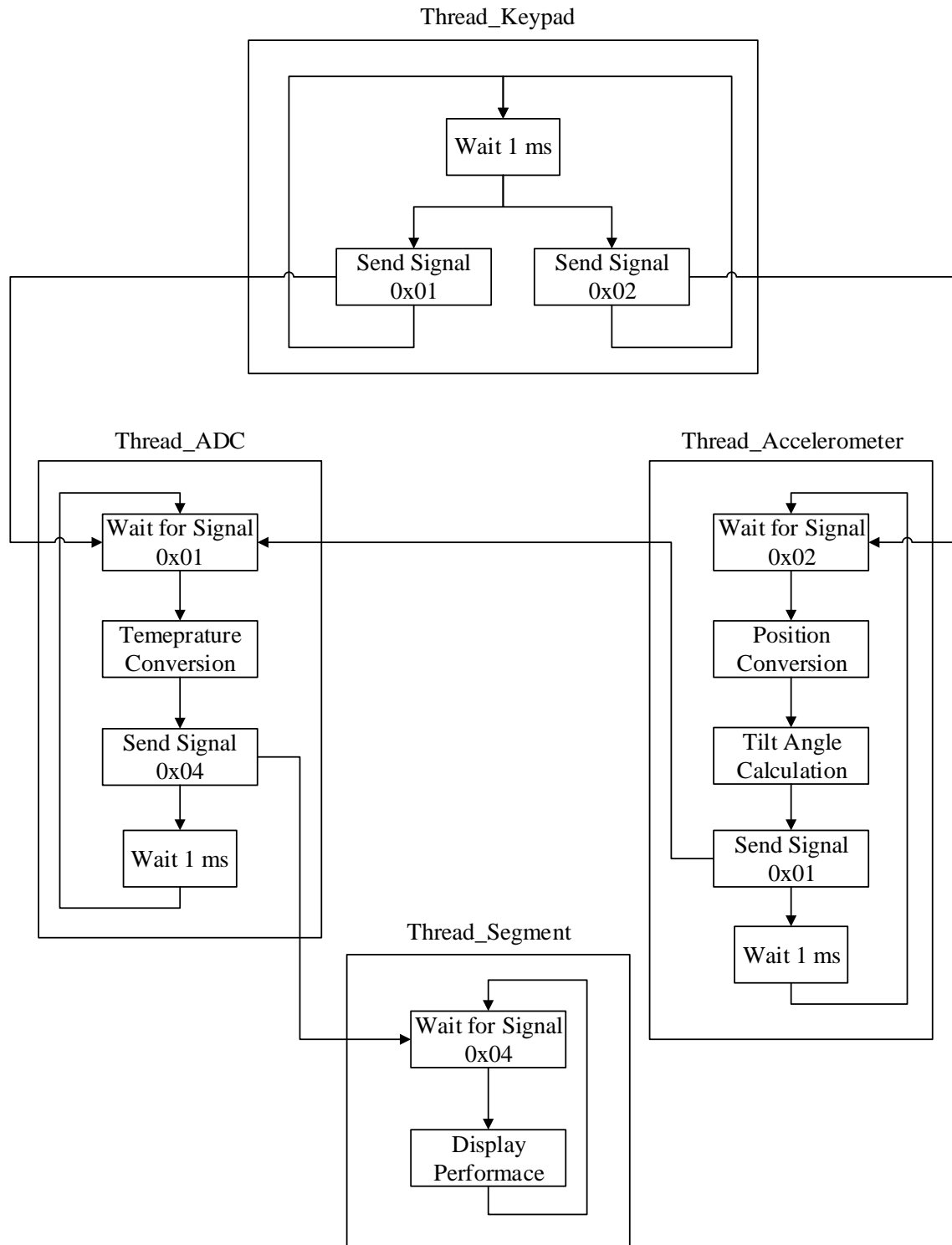


Figure 2 – Overall threads design diagram

The priority of each thread is listed in Table 10. Additionally, Figure 2 represents our design for this multithreading system. According to the design specifications, the accelerometer can be ignored while operating in temperature mode. However, it is necessary to monitoring the

temperature in the background while the system is computing the current roll and pitch angle of the board. The reason that the temperature sensor must run all the time regardless the selected mode is for overheat checking. The display makes performing data flashing on and off every time the measured temperature is above the threshold. For example, when the user selects accelerometer mode, the position sensor will be activated after receiving the signal from keypad. Once the angle computation is done, a signal will be sent to ADC in order to verify the processor's temperature. Then, temperature meter signals display to show the orientation of the board. These instructions are being executed continuously in short time, so the system appears to have all simultaneously running threads. Later in the section 5.2, we will discuss how the actual thread trace is different from what we expect.

5 Testing and Observations

5.1 Kalman Filter's parameters

We use accelerometer to measure tri-axis acceleration force and output corresponding values. The three dimensional Cartesian coordinates are introduced with noisy signals. According to the initial values of parameters given in lab 1, we can see that the filtered outputs still have noise in Figure 3 for x-axis. We should determine the better values of each filter parameters to reduce the error caused by vibration noise. To achieve this goal, we adjust the process noise covariance (q) and measurement noise covariance (r). Instead of testing for a small range of coordinate value, we rotate the board from 0° to 90° along each axis. We collect that all x , y , and z values increasing from 0 to 1.

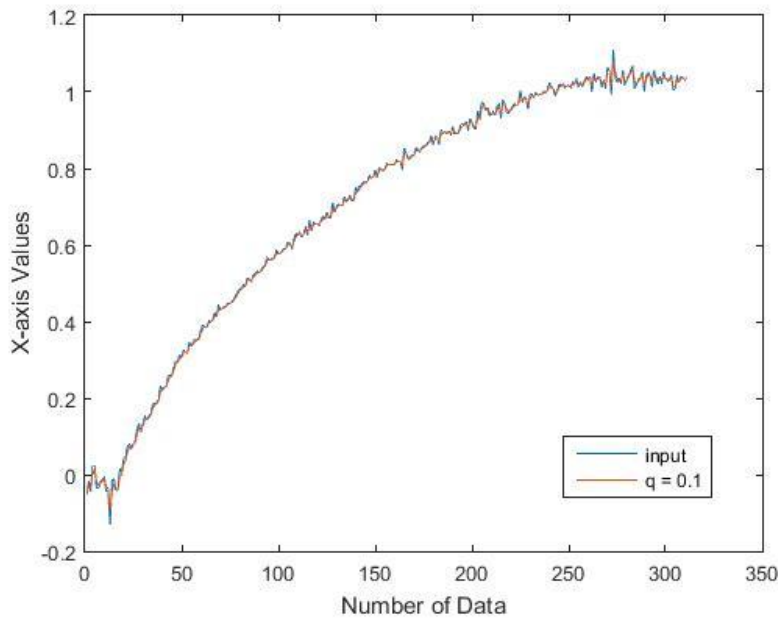


Figure 3 – Performance of Kalman filter using original parameters for x-axis

Since the steps of finding filter parameters are the similar for all three axes, we will only discuss x-axis as an example in this section. We start by decreasing the value of q by scale of 10. In Figure 4, we notice that the filtered data are more stable when the process noise is between 0.01 and 0.001. When the q is too small, the filtered output no longer follow the inputs, which means it leads to its insensitivity. By slightly modifying process noise covariance, we obtain the better q value for the filter.

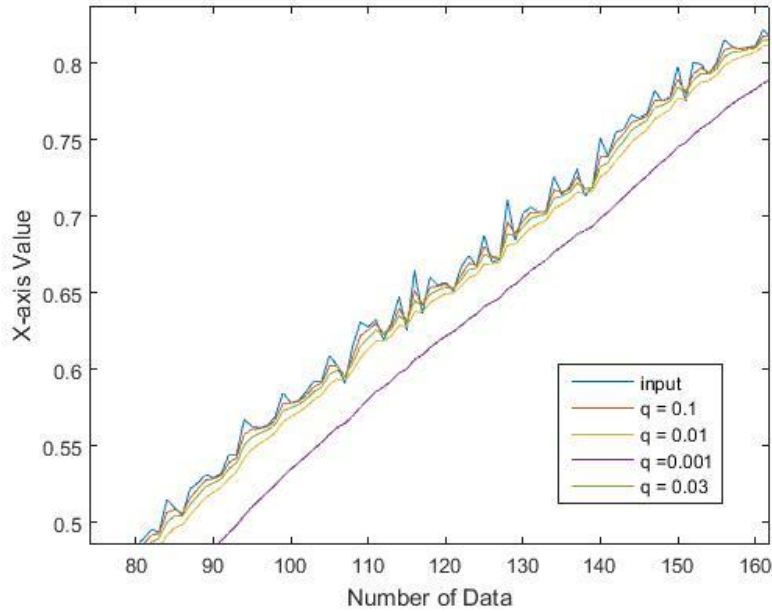


Figure 4 – Performance of Kalman Filter using different q values for x-axis

Furthermore, we now adjust the measurement noise with the new q values. We observe the changes of filtered outputs while decrementing r value because the filter cannot remove the error properly if r is too large. Then, the filter is no more accurate. When the r is equal to 0.02, the plot of results is closer to the plot of inputs compare to the plot when r is equal to 0.1. This observation indicates that the best value of r should be between 0.02 and 0.1. In the Figure 3, the output data for $r = 0.048$ overlaps the output plot for $r = 0.05$. Therefore, the Kalman filter provides the best results for x-axis coordinate by cancelling the error caused by noise when $q = 0.03$ and $r = 0.048$ without changing any p and k variables. By repeating the same process, we can determine each set of filter parameters for y-axis and z-axis in Appendix (Figure A-3, A-4, A-5, and A-6).

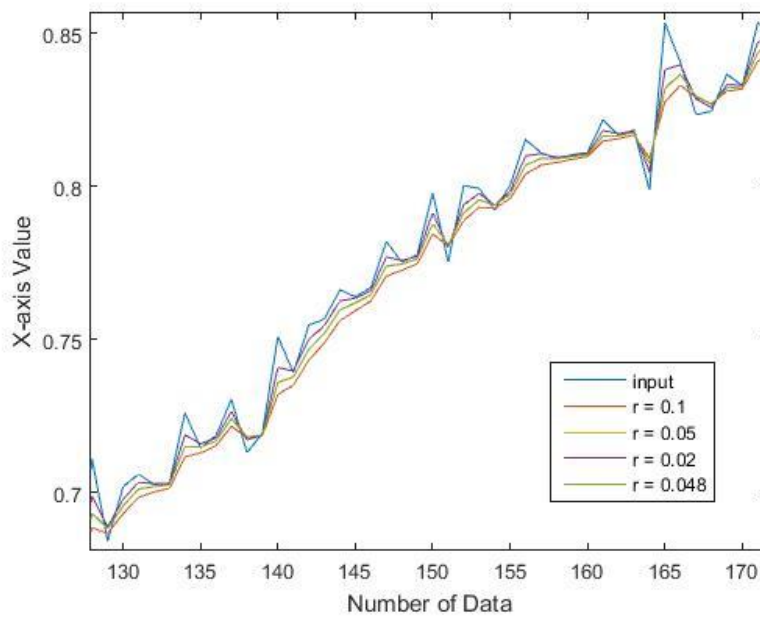


Figure 5 – Performance of Kalman Filter using different r values with $q = 0.03$ for x -axis

5.2 Threads Observations

Using the System and Thread Viewer in debugging mode, we can clearly observe the stack usage of each thread for the program. In Figure 6, we use 5 threads in total. Besides the 4 threads that we designed, there is a thread called idle. The scheduler switches to the idle thread when there is no thread being executed. Also, the program's runtime effects the stack usage, and we notice that most of the threads do not use more than half of the default stack memory. It means that there is no stack overflow problem in our current design. We can modify each thread's stack size accordingly.

Property

Value

System

| Item | Value |
|------------------------------|---------------------------------------|
| Tick Timer: | 0.100 mSec |
| Round Robin Timeout: | 0.500 mSec |
| Default Thread Stack Size: | 800 |
| Thread Stack Overflow Check: | Yes |
| Thread Usage: | Available: 6, Used: 5 + os_idle_demon |

Threads

| ID | Name | Priority | State | Delay | Event Value | Event Mask | Stack Usage |
|-----|----------------------|-------------|----------|-------|-------------|------------|------------------------------|
| 2 | Thread_Keypad | AboveNormal | Wait_DLY | 31 | | | cur: 8%, max: 12% [96/800] |
| 3 | Thread_ADC | Normal | Wait_DLY | 27 | 0x0000 | 0x0001 | cur: 27%, max: 27% [216/800] |
| 4 | Thread_Accelerometer | Normal | Wait_AND | | 0x0000 | 0x0002 | cur: 27%, max: 53% [424/800] |
| 5 | Thread_Segment | BelowNormal | Running | | | | cur: 19%, max: 31% [248/800] |
| 255 | os_idle_demon | None | Ready | | | | |

Figure 6 – Capture of system and thread viewer of the application

The Event Viewer provides a visual real-time thread trace. By debugging, the application runs the threads in the following order: keypad, ADC, accelerometer, and segment. In Figure 7, it is the events trace when user select to display temperature mode. The events pattern of temperature mode is similar to our ideal trace in hypothesis. However, the trace of accelerometer looks different. As ADC and acceleration have the same priority, the system starts to run both operations at the same time by applying Round-Robin scheduler. When the system recognizes the key for mode selection, it sends a signal to accelerometer thread and it suspends the ADC thread. As the trace shown in Figure 8, the system suspends the ADC and executes accelerometer. It jumps back to resume ADC function until the computation of board's orientation is finished. Lastly, the display thread will wait for the signal sent by ADC to run. The execution time for display is longer than other threads because of the time delay between each digit display. Since the segment thread has the lowest priority, no thread will disturb the display. The timing trace of idle on both graphs suggests that there is no thread running and the highest thread is put to sleep.

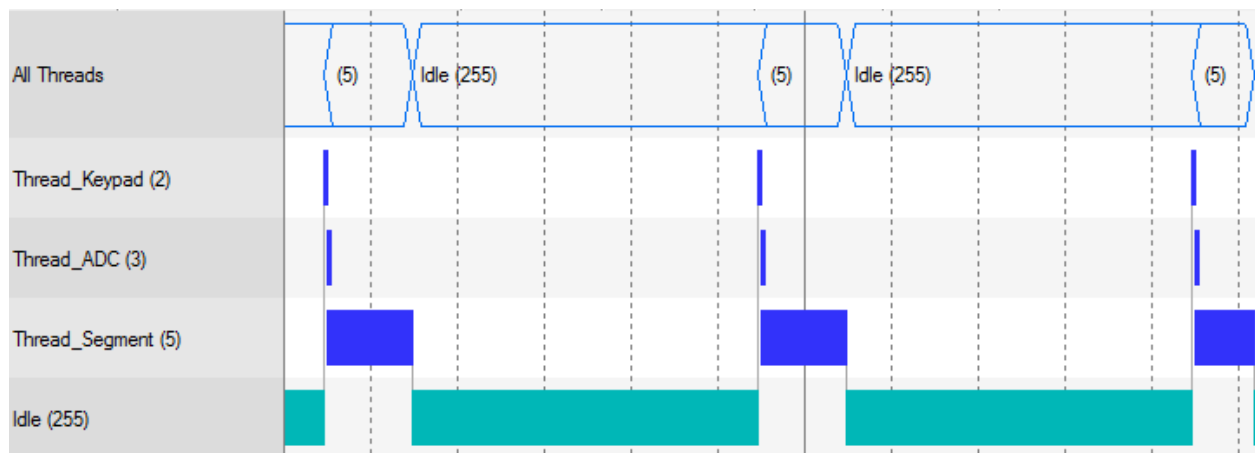


Figure 7 – Capture of events viewer for temperature mode

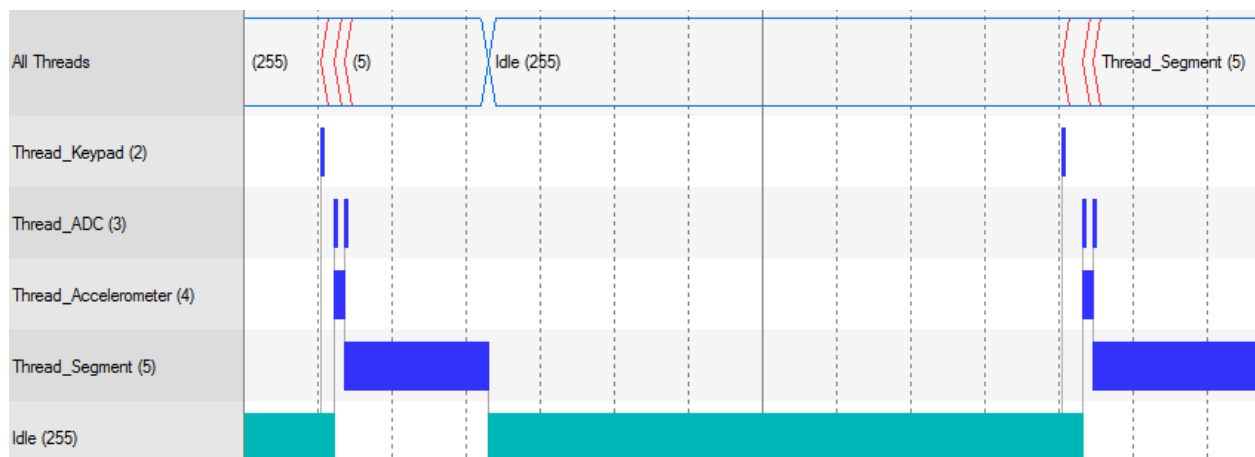


Figure 8 – Capture of events viewer for accelerometer mode

6 Conclusion

In this experiment, we write an applications that use multithreading mechanism which make the system appear to run many execution at the same time. In fact, the system handles each threads within the executions and switches between threads for a very short time slice. In detail, the user command interface is designed to be a thread, and the data processing components are in completely different threads. Signal and wait is the technique used for inter-thread communication. Once a thread finishes its job, it notifies the next thread to run. The next thread must be blocked and waiting until the signal is received.

In the RTOS scheduler, the thread with higher priority always run, so we set the thread priority of keypad scanner to be a highest. After user decides which mode to be displayed, temperature sensor and accelerometer will reacts accordingly. Since both operations are independent and do not share any data, their threads have the same level of priority. Normally, the Round-Robin scheduling policy is applied when threads have equal priority. But in our case, the thread of execution order is dependent on the signal sent from keypad thread; especially, for accelerometer mode. The display output can only performed when the temperature or angle data is ready. Hence, it means that the thread of display has the lowest priority among all threads and it is the last operating function to execute.

With the row and column arrangement of the keypad, we manage to read the state of four switches of a row or of a column at one time. Every keypress generates a unique code to let display know which mode to perform. The display shows data reported by the sensors in a sequential fashion, so we use an independent hardware timer to operate time delay between each digit's performances. We alter the refresh rate about 500 microseconds to make the transition fast enough to fool human's eyes. Furthermore, the display outputs are blinking on and off when the system reaches a high temperature.

The software modules can be split into two major part that are responsible for data readings: temperature and orientation computations. The first part is to get a data from built-in temperature sensor using single conversion mode of ADC peripheral on the STM32 processor. When the conversion complete, we implement Kalman filter with parameters of $q = 0.07$ and $q = 3$ to reduce the noise introduced while sampling. Then, the display outputs the temperature in degree Celsius format on a four digit 7-segment display. The second part is to measure the three dimensional coordinates of the board based on gravity. We calibrate xyz values according to the measurements of 6 stationary positions in order to minimize the measurement uncertainty of accelerometer, and use software filter to eliminate the noise. All three axes have the same measurement noise covariance (r) which is equal to 0.048. The parameter q for x, y, and z axes is 0.03, 0.03, and 0.07, respectively. The application converts the coordinate systems to angle and sends the data to display. Overall, the temperature meter and position sensor are both stable, sensitive, and accurate.

7 Appendix

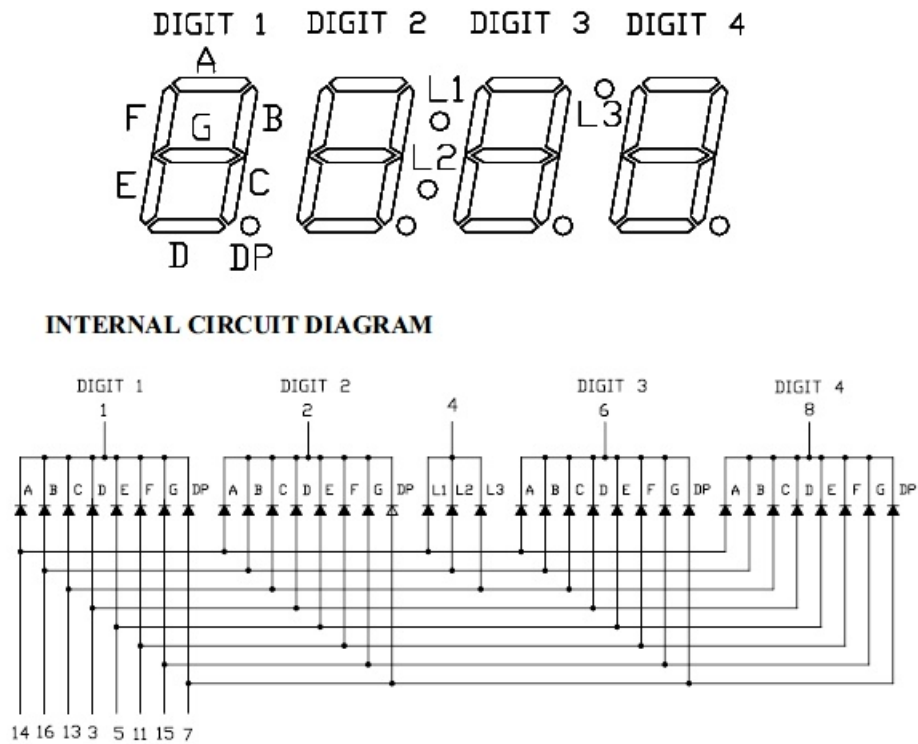


Figure A-1 – Four digit segments arrangement [6]

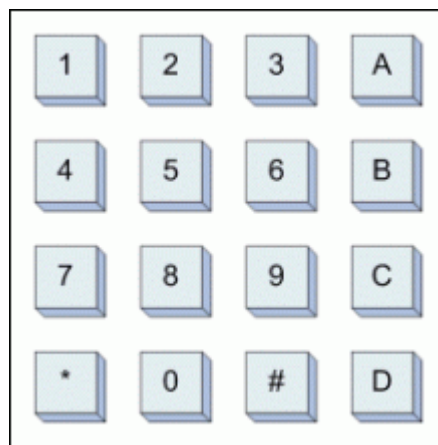


Figure A-2 – Keypad layout [8]

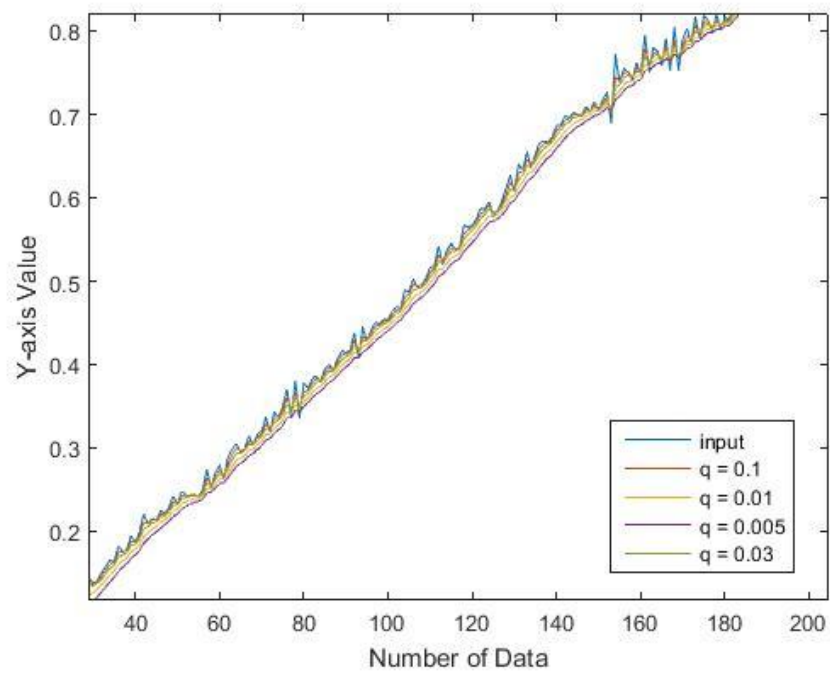


Figure A-3 – Performance of Kalman Filter using different q values for y-axis

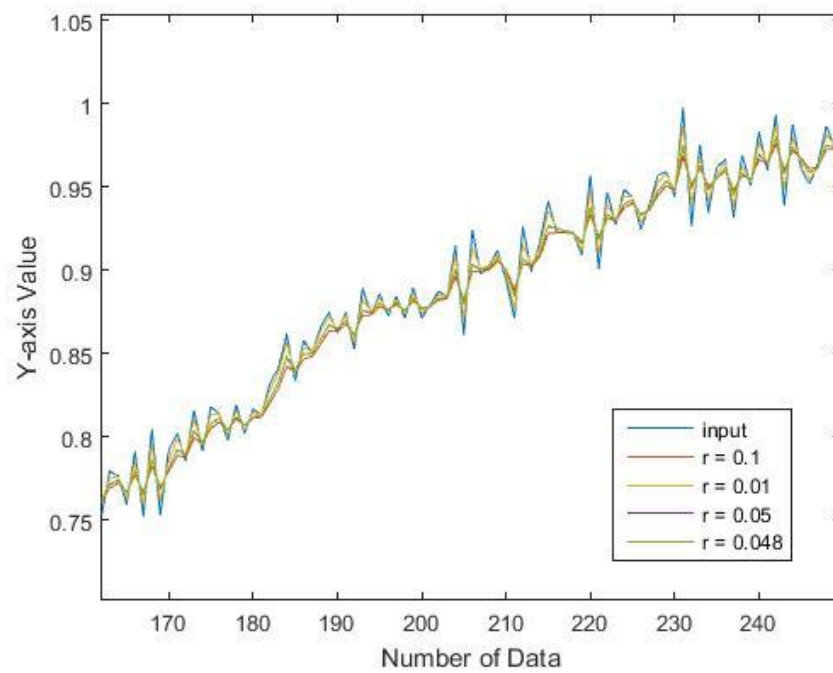


Figure A-4 – Performance of Kalman Filter using different r values with $q = 0.03$ for y-axis

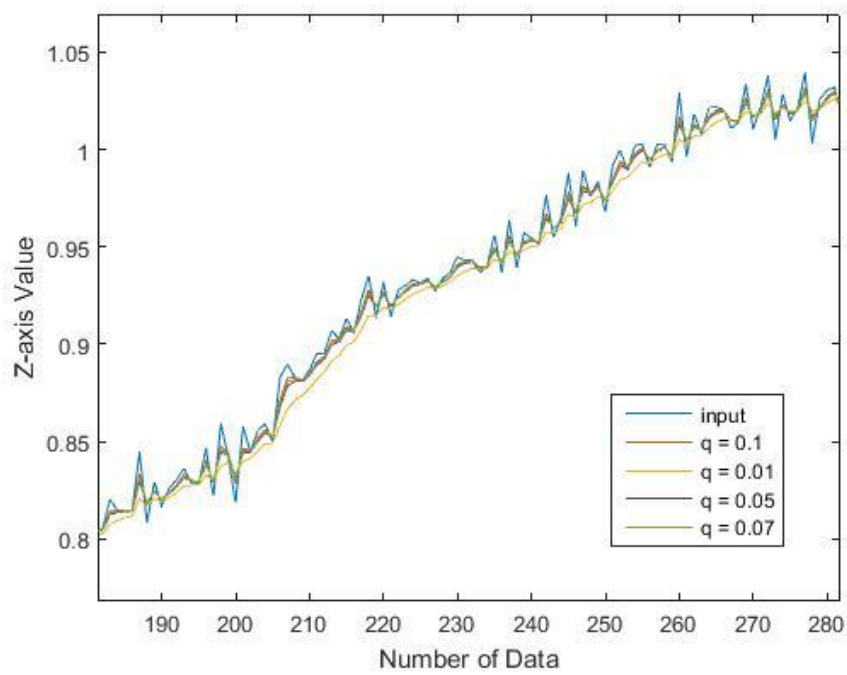


Figure A-5 – Performance of Kalman Filter using different q values for z-axis

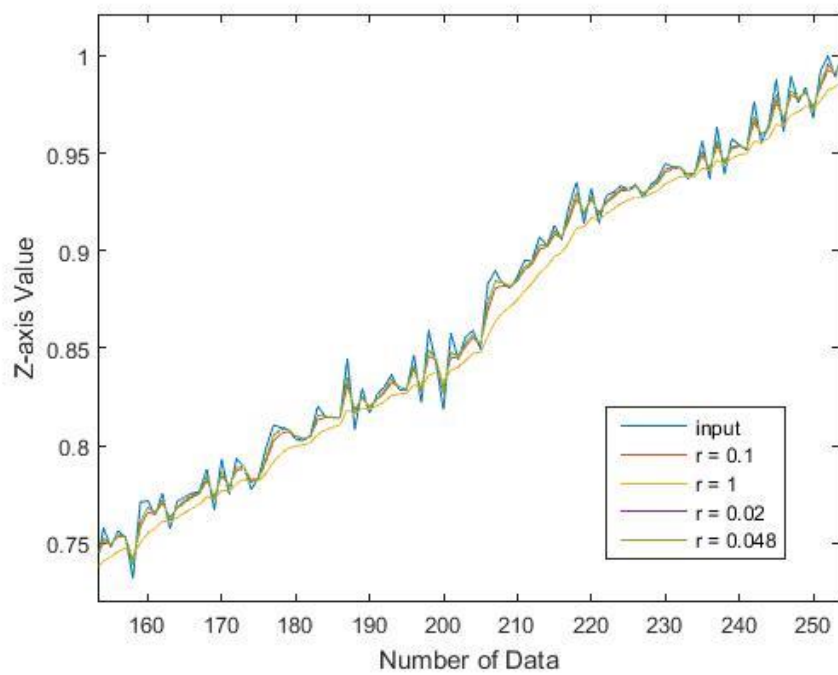


Figure A-6 – Performance of Kalman Filter using different r values with $q = 0.07$ for y-axis

8 References

- [1] STMicroelectronics, “Doc_10 – STM32F407xx,” datasheet, October 2015.
- [2] *Doc_05 – STM32F4xxx Reference Manual*, Revision 1, STMicroelectronics, September 2011.
- [3] *Doc_19 – Documentation of STM32F4xx Cube HAL drivers*, Revision 3, STMicroelectronics, September 2015.
- [4] A. Suyyagh, Doc_32 – Generic Keypad and Hitachi HD44780 tutorial, 2016.
- [5] *Doc_10 – Discovery Kit F4 Rev.C*, Revision 4, STMicroelectronics, January 2014.
- [6] R. Bhatt. Serial 4-digit Seven Segment LED Display [Online]. Available: <http://www.electronics-lab.com/project/serial-4-digit-seven-segment-led-display/> [Accessed: February 18, 2016]
- [7] *Doc_15 – Tilt angle application notes*, Revision 1, STMicroelectronics, April 2010.
- [8] Maxim Integrated Products, Inc. Using a Keypad and LCD Display with the MAXQ2000 [Online]. Available: <https://www.maximintegrated.com/en/app-notes/index.mvp/id/3414> [Accessed: March 14, 2016]