Lab 2 – Report

Utilization of processor's temperature sensor to determine operating temperature of STMF32 Discovery board

Group 7
Ravi Chaganti – 260469339
Hoai Phuoc Truong – 260526454

ECSE 426
Department of Electrical and Computer Engineering
McGill University

17th October, 2015

Table of Contents

1. Abstract

The main objective of this experiment is to understand the importance of embedded microprocessor systems and their role in key measurements of various parameters including temperature of processor. Our aims in this lab are to get acclimatized to the STMF32 discovery board, gauge the working model of the processor in terms of hardware units and its software library components in Keil, and produce readings from sensors that are enabled within the processor. We will be monitoring the temperature of the processor to protect it from overheating.

2. Problem Statement

In our experiment, our primary focus is to take measurements from the temperature sensor, which is located under the microprocessor's chip, to get the operating temperature of the STMF32 discovery board. The temperature measured from the sensor must be first converted from analog to digital readings by using the ADC (analog to digital conversion) to digitize the temperature values in the form of signals (voltage format). After this conversion, the signals must be converted to the standard temperature format (Celsius).

The temperature readings obtained from the sensor after the ADC conversion are subjected to various disturbances including electromagnetic interference, thermal noise and quantization noise. To solve this issue, a highly efficient filter must be implemented which would minimize the signal-noise-ratio (SNR). One of the best and simple approaches is the moving average filter which acts as a low-pass filter to streamline and smooth the signals that are obtained from the temperature sensor. However, this approach would also require the perfect determination of filter depth (D) to smoothen the signals.

To display the temperature readings in a visually appealing way, an analogue meter panel must be used which would display the operating temperature of the processor. This would require implementing the servo motor on one of the GPIO pins on the board and pulse width modulation (PWM) signals to time the readings from the sensor.

An alarm must be designed for the processor to monitor the temperature of the processor that is measured by the sensor and to make sure that it doesn't cross over a certain threshold. All these designs must take place with respect to the interrupt handler which will define the sampling frequency by implementing a SysTick timer.

3. Theory and Hypothesis

In our implementation of the moving average filter to smoothen the values of the output signal, we will be using a circular buffer as it is connected end-to-end and is easy to use to buffer continuous data streams. In this model, we will be taking the average of the number of elements based on the following formula[1]:

$$y[i] = \frac{1}{M}\sum_{j=0}^{M-1} x[i+j]$$

where M is the number of elements (defined by depth (D) or size of the circular buffer).

To implement the analogue meter panel, we will be using pulse width modulation (PWM) theory [2]to set the setup high or low for taking measurements. The voltage source is provided to the setup (encoded with PWM) by a repeated series of on and off pulses.

We will be using pulse durations for 2 points: 0 degrees and 180 degrees to convert the angle of the analogue meter panel to time which will be used for blocking time. One of the main hypothesis that we have established to calculate the angle of the analogue meter panel is that the relationship between the temperature of the processor and the angle of the panel is linear.

4. Implementation

4.1 ADC & Temperature Sensor Initialization and Interrupt Handler

To implement the ADC, we first initialized the ADC struct (Figure 3 in Appendix) to enable the ADC peripherals and power for bus APB2 which is connected to ADC1 peripheral. To wake up the temperature sensor, ADC_TempSensorVrefintCmd(ENABLE) method is called and channel 16[3] is set for the ADC defined above as it is internally connected to the temperature sensor (Figure 4 in Appendix). This implementation can be seen in system_config.c and system_config.h.

The sampling frequency was setup by using ARM's SysTick_Config timer method which takes in a frequency parameter based on the system's core clock and SysTick frequency. The system's core clock operates at 168MHz and the SysTick frequency defined for our setup to sample measurements is 50Hz. We defined an interrupt handler by setting the system_ticks as 1 and run the while loop to control the frequency of temperature measurements (Figure 5 in Appendix). This implementation can be seen in main.c.

---

[1] Smith, S. (1997). Moving Average Filters. In The scientist and engineer's guide to digitalsignal processing (pp. 277-280). San Diego, Calif.: California Technical Pub.

[2] Barr, M. (2001). Pulse Width Modulation. In Embedded Systems Programming (pp. 103-104)

[3] STM32F405xx Datasheet - production data. (n.d.). Retrieved October 19, 2015, (pp. 5) from http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf?s_searchtype=keyword

To digitize the measurements from the temperature sensor, ADC_SoftwareStartConv(ADC1) and ADC_GetConversionValue(ADC1) methods were called to make this conversion and the final converted value was stored in the parameter 'result' (Figure 6 in Appendix). These implementations can be seen in temperature.c and temperature.h.

4.2 Digital Signal to Degrees Celsius Conversion

To convert digital signal (which is stored in the 'result' parameter) into degrees Celsius, the following equation[4] was derived (Figure 4 in Appendix):

$$temperature\ reading\ (degrees\ Celsius) = \frac{\frac{result \times 3000}{4096} - V\_25}{AVERAGE\_SLOPE} + 25$$

where the constant parameters in the equation are the following[5]:
result: conversion from digital signal
V_25 = 760 mV
AVERAGE_SLOPE: 2.5
4096: 12 bits → 2^{12}

This implementation can be seen in temperature.c and temperature.h.

4.3 Signal filter design

To streamline and smoothen the digital signal (which is converted to degrees Celsius), we used a moving average filter design. We defined a circular buffer with methods to create operations of insertion, deletion and getting the elements. A summary of all operations can be seen in Figure 7 in Appendix. This implementation can be seen in utils/circular_buffer.c, and utils/circular_buffer.h.

The algorithm to calculate the moving average filter was obtained from the moving average filter theory[6]. We defined depth (D), which is the number of elements which will used in order to calculate the average of those elements. If the size of the number of elements is less than the depth, then the average is calculated with respect to the size and not the depth. If size is greater than the depth, then the average is calculated with respect to the depth. We stored these averages dynamically in the circular buffer by appending and removing them continuously so as to save memory of the circular buffer (Figure 8 in Appendix). This implementation can be seen in ma_filter.h and ma_filter.c.

---

[4] RM0008 Reference Manual. (n.d.). Retrieved October 19, 2015, from http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf

[5] STM32F405xx Datasheet - production data. (n.d.). Retrieved October 19, 2015, (pp. 6, 37, 134) from http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf?s_searchtype=keyword

[6] Smith, S. (1997). Moving Average Filters. In The scientist and engineer's guide to digitalsignal processing (pp. 277-280). San Diego, Calif.: California Technical Pub.

The method ma_filter(float input) takes take the temperature readings, which are in the degrees Celsius form, and implements the moving average filter with respect to the depth (D). To predict the approximate value of depth (D), we implemented the following code in MATLAB which will generate random numbers for sine wave and calculate the filter of all the numbers:

1. $t = linspace(0{,}2\times pi, 100)$
2. $y = sint(t)$
3. $f = 3\times y + transpose(rand(100{,}1))$
4. $g = filter\ (\frac{1}{14} \times ones(1{,}14), 1, f)$
5. $plot(f)$
6. $plot(g)$

Equation 1 defines 100 numbers ranging from 0 to 2π. Equation 2 is the sine wave of the input t. Equation 3 defines a function f which is subjected to random numbers that is added to the sine wave. Equation 5 plots the graph of function f which is random because of noise. Equation 4 defines a function g, which filters the function f with respect to the filter depth (D) (which is 14) and equation 6 plots the graph of function g. Estimation of depth (D) is explained in the Observations section.

4.4 Analogue Meter Panel Design

To display the temperature readings, we connected a servo motor to the processor and attached a pointed arrow to the motor which rotates in both clockwise and anticlockwise directions. We drew a protractor on cardboard of angles between 5 degrees and 175 degrees. The motor has 3 cables: red (for the 5V connection), black (for ground connection), and yellow (connected to GPIOD_7 for the PWM signal). We initialized the GPIO pin using the GPIO struct (Figure 9 in Appendix). The yellow cable was connected to GPIOD_& using a male-to-female connector which allowed us easily separate the yellow cable from red and black cables to avoid cable interference. GPIO initialization can be seen in system_config.c and system_config.h.

We implemented the rotation of the motor using the PWM signal theory[7]. We set the pulse width to 470µs which points to 0 degrees and 2200µs which points to 180 degrees after tuning the motors (explained more in detail in Observation section) based on trial and error method. We converted the PWM signal from degrees to time (in µs) by using the following formula:

$$wait\_time = 0\_DEGREE + \left(\frac{angle}{MAX\_ANGLE} \times (180\_DEGREE\ -\ 0\_DEGREE)\right)$$

where 0_DEGREE = 470µs, 180_DEGREE = 2200µs, MAX_ANGLE = 180°

[7] Barr, M. (2001). Pulse Width Modulation. In Embedded Systems Programming (pp. 103-104)

We defined a method blocking_wait as:

$$blocking_{wait} = wait_{time} \times ONE\_MICRO\_SECOND\_DELAY$$

where ONE_MICRO_SECOND_DELAY is 19 units (a constant derived from hit and trial method for our setup). This method will set it high so as to measure the PWM signals during this time. This implementation can be seen in the files motor_interface.c and motor_interface.h.

Finally, we designed the protractor on cardboard such that 5 degrees points to 20 degrees Celsius and 175 degrees points to 60 degrees Celsius. We derived a simple linear equation in the method temperature_to_angle (float temperature) which is given below:

$$angle = temperature \times slope + constant$$

where slope is 4.25 and constant is -80. This can be seen in util/lab2_util.c. Our setup implementation can be seen in the following picture:

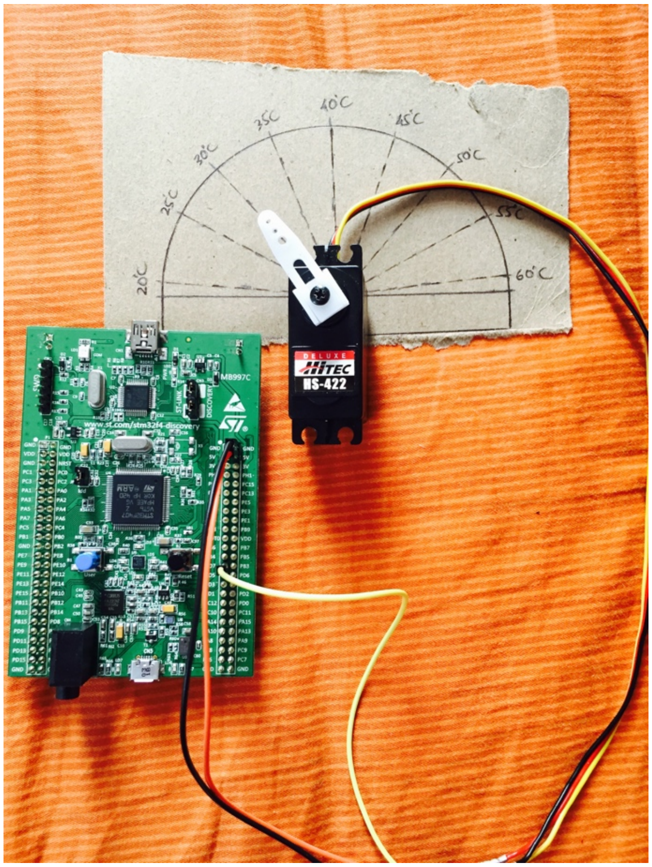

Figure1: servo motor connection to processor

4.5 Overheating Alarm

Once the ADC struct is initialized, we then initialized the GPIO struct with all its definitions (Figure 9 in Appendix)). We enabled the AHB1 peripheral clock to power both GPIOB and GPIOD for pins GPIO_Pin_12, GPIO_Pin_13, GPIO_Pin_14, and GPIO_Pin_15 to highlight LEDs 3,4,5,6. This implementation can be found in system_config.c and system_config.h.

We defined a parameter ALARM_THRESHOLD which is the maximum temperature after which the LEDs will start circulating to show that temperature of the processor is overheating. We used a counter and set it to 'greater than 10' to circulate the LEDs. SET_LED method will set all the GPIO bits and led_all_off method will clear all the LEDs by reseting the GPIO bits. This implementation can be found in alarm.c, alarm.h, led_interface.c and led_interface.h. The figure below describes the FSM flowchart of all the LEDs and RESET states:

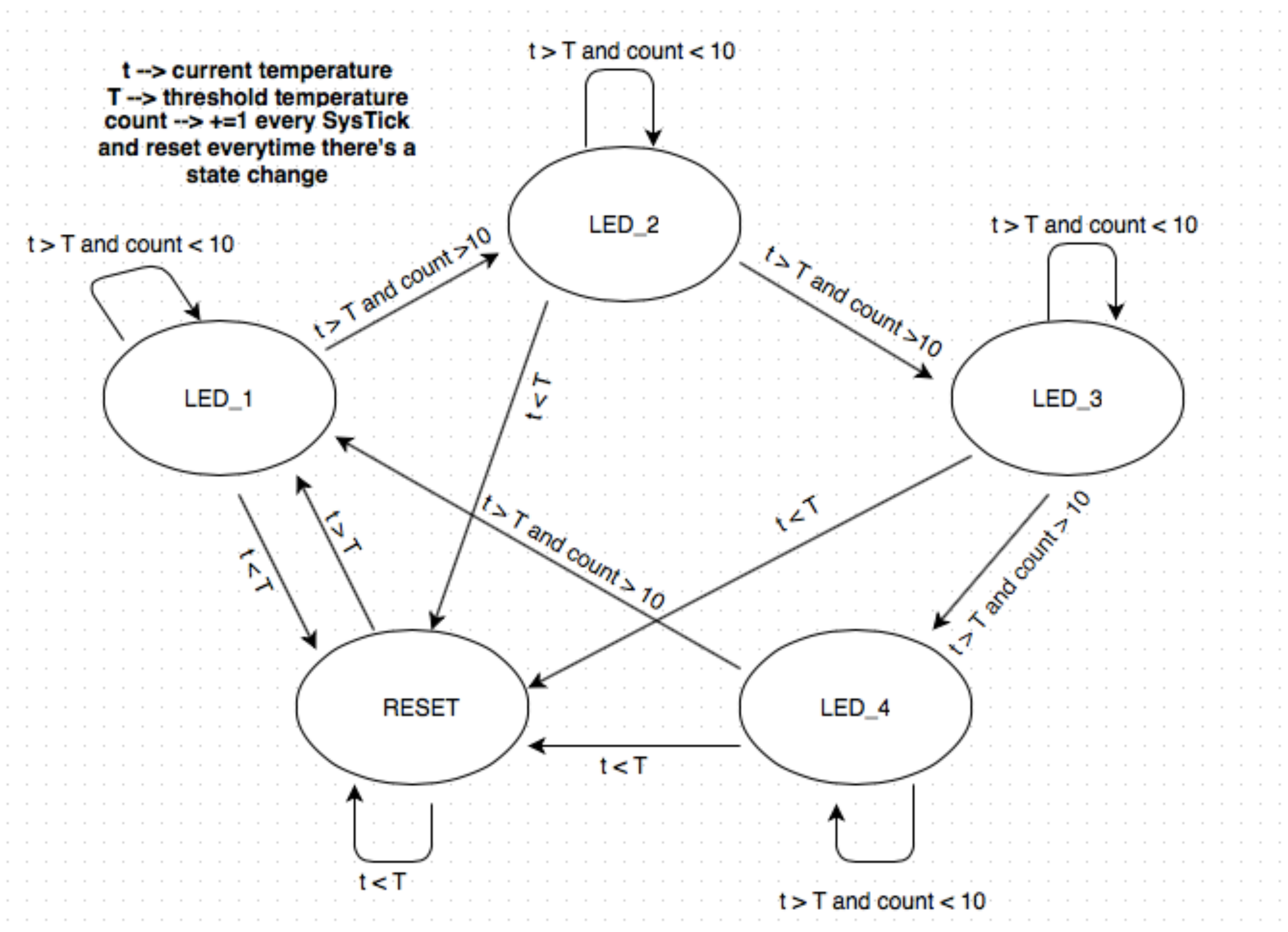

Figure2: state diagram of LEDs and RESET in alarm implementation

5. Observation

To check the raw data of the temperature of the processor (after ADC and signal to temperature conversion), we added the parameter temperature_reading to the watch panel and monitored it by blowing hot air from a hair-dryer. We noticed that once the temperature was going above 60 degrees Celsius (which was our defined threshold), the 4 LEDs started circulating to display overheating.

To verify the analogue meter panel, we connected the servo motor's yellow cable to GPIOD_7, red cable to 5V and black cable to the ground. We started blowing hot air from the hair-dryer and saw that the LEDs began to circulate again after we crossed the threshold temperature. However, the arrow pointer of the servo-motor was fluctuating and was inconsistent due to signal noise (defined depth(D) as 4). We increased the depth(D) from 4 to 14 after verifying with the filter program written in MATLAB to smoothen the clockwise and anticlockwise rotation of the arrow pointer. The rotation became very consistent after this change of depth(D) to 14.

See Figures 10, 11, 12, and 13 in Appendix for the various plots and depths (D) and how we obtained the correct depth of 14. In figure 5.1, we plotted a sine wave graph which is subjected to random signal noise. In figure 11, we estimated the depth to be 4 which smoothened the curve to some extent but not entirely. We corrected the depth to 14 in Figure 13 (Appendix) and obtained a smooth graph of the sine wave function.

We also observed that our conversions from temperature to angle and angle to wait time had linear characteristics and worked well when applied in the formulas in sections 4.2 and 4.4.

6. Conclusion

Using temperature sensor of the processor to monitor the operating temperature of the processor was a very convenient way to detect if the processor is overheating or not. Our observations were consistent with the requirements of this experiment after our derivation and implementation of temperature-digital signal, angle-time_wait, and temperature-angle linear equations, ADC, interrupt handler, circular buffer for data filtering, analogue meter panel and alarm overheating.

7. Appendix

References

1. Smith, S. (1997). Moving Average Filters. In The scientist and engineer's guide to digitalsignal processing (pp. 277-280). San Diego, Calif.: California Technical Pub.

2. Barr, M. (2001). Pulse Width Modulation. In Embedded Systems Programming (pp. 103-104)
3. RM0008 Reference Manual. (n.d.). Retrieved October 19, 2015, from http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf
4. STM32F405xx Datasheet - production data. (n.d.). Retrieved October 19, 2015, (pp. 37, 134) from http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf?s_searchtype=keyword

Figures

1. ADC stuct initialization

```
void adc_init(void) {
    ADC_InitTypeDef adc_init_s;  // Structure to initialize definitions of ADC
    ADC_CommonInitTypeDef adc_common_init_s; // ADC Common Init structure definition
    ADC_DeInit(); // Deinitializes all ADCs peripherals registers to their default reset values.
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); // Enable Power for bus APB2 connected to peripheral ADC1.
    adc_common_init_s.ADC_Mode = ADC_Mode_Independent; // Configures the ADC to operate in independent mode.
    adc_common_init_s.ADC_Prescaler = ADC_Prescaler_Div2; //Select the frequency of the clock to the ADC (devide clock frequency by 2). The clock is common
    adc_common_init_s.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled; //Configures the Direct memory access mode for multi ADC mode.
    adc_common_init_s.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles; //Configures the Delay between 2 sampling phases.
    ADC_CommonInit(&adc_common_init_s); //Initializes the ADCs peripherals according to the specified parameters in the ADC_CommonInitStruct.

    adc_init_s.ADC_Resolution = ADC_Resolution_12b; // Configures the ADC resolution to 12 bits.
    adc_init_s.ADC_ScanConvMode = DISABLE; // Specifies whether the conversion is performed in
    adc_init_s.ADC_ContinuousConvMode = DISABLE;
    // Will just convert single value when triggered. If enabled it will start a new conversion as soon as it finishes one
    adc_init_s.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None; //Select the external trigger edge and enable the trigger of a regular group. Can s
    adc_init_s.ADC_DataAlign = ADC_DataAlign_Right; // Specifies whether the ADC data  alignment is left or right. If right, MSB in register set to 0 and da
    adc_init_s.ADC_NbrOfConversion = 1; // Specifies the number of ADC conversions that will be done using the sequencer for regular channel group.!
    //wake up temperature sensor
    ADC_TempSensorVrefintCmd(ENABLE);
    ADC_Cmd(ADC1, ENABLE); // Enables the specified ADC peripheral (turn it on).
    ADC_RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_480Cycles); // Configures for the selected ADC a regular channel, rank in the sequencer
    ADC_Init(ADC1, &adc_init_s);
}
```

Figure3

2. Temperature sensor initialization

```
    //wake up temperature sensor
    ADC_TempSensorVrefintCmd(ENABLE);
    ADC_Cmd(ADC1, ENABLE); // Enables the specified ADC peripheral (turn it on).
    ADC_RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_480Cycles); // Configures for the selected ADC a regular channel, rank in the sequencer
    ADC_Init(ADC1, &adc_init_s);
```

Figure4

3. Interrupt Handler

```
/*interrupt handler
*/

void SysTick_Handler(){
    system_ticks = 1;
}
```

Figure5

4. read_temperature method

```c
void read_temperature(void) {
    ADC_SoftwareStartConv(ADC1);
    //Starting Conversion, waiting for it to finish, clearing the flag, reading the result
    while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET); //Could be through interrupts (Later)
    //EOC means End Of Conversion
    float result = ADC_GetConversionValue(ADC1); // Result available in ADC1->DR
    temperature_reading = (((result * 3000) / 4096) - V_25) / AVERAGE_SLOPE + 25;
}
```

Figure6

5. circular buffer's operations

```c
int circular_buffer_init(circular_buffer* empty, BUFFER_TYPE* data, uint16_t size);
int circular_buffer_is_full(circular_buffer* buffer);
int circular_buffer_is_empty(circular_buffer* buffer);

int circular_buffer_append(circular_buffer* buffer, BUFFER_TYPE* new_value);
int circular_buffer_remove_last(circular_buffer* buffer, BUFFER_TYPE* value);

int circular_buffer_get_first(circular_buffer* buffer, BUFFER_TYPE* value);
int circular_buffer_remove_first(circular_buffer* buffer, BUFFER_TYPE* value);

int circular_buffer_clear(circular_buffer* buffer);
int circular_buffer_size(circular_buffer* buffer, uint16_t* size);

int circular_buffer_get(circular_buffer* buffer, uint16_t index, BUFFER_TYPE* value);
```

Figure7

6. Calculating moving average of the filter

```c
        if (size < MA_FILTER_DEPTH) {
            float sum = average * size + input;
            circular_buffer_append(&cb_data, &input);
            average = sum / (size + 1);
            return average;
        } else {
            float sum = average * MA_FILTER_DEPTH;
            float removing = 1;
            circular_buffer_remove_first(&cb_data, &removing);
            circular_buffer_append(&cb_data, &input);
            sum -= removing;
            sum += input;
            average = sum / MA_FILTER_DEPTH;
            return average;
```

Figure8

7. GPIO pin struct

```c
void gpio_init(void) {
    GPIO_InitTypeDef gpio_init_s; // Structure to initilize definitions of GPIO
    GPIO_StructInit(&gpio_init_s); // Fills each GPIO_InitStruct member with its default value

    /* TIM3 clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
    RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOB, ENABLE); // Enables the AHB1 peripheral clock, providing power to GPIOB branch
    RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOD, ENABLE); // Enables the AHB1 peripheral clock, providing power to GPIOD branch
    gpio_init_s.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15; // Select the following pins to initialise
    gpio_init_s.GPIO_Mode = GPIO_Mode_OUT; // Operating mode = output for the selected pins
    gpio_init_s.GPIO_Speed = GPIO_Speed_100MHz; // Don't limit slew rate, allow values to change as fast as they are set
    gpio_init_s.GPIO_OType = GPIO_OType_PP; // Operating output type (push-pull) for selected pins
    gpio_init_s.GPIO_PuPd = GPIO_PuPd_NOPULL; // If there is no input, don't pull.
    GPIO_Init(GPIOD, &gpio_init_s); // Initializes the GPIOD peripheral.

    motor_init(&gpio_init_s);
}
```

Figure9

8. Plot of random numbers on a sine wave

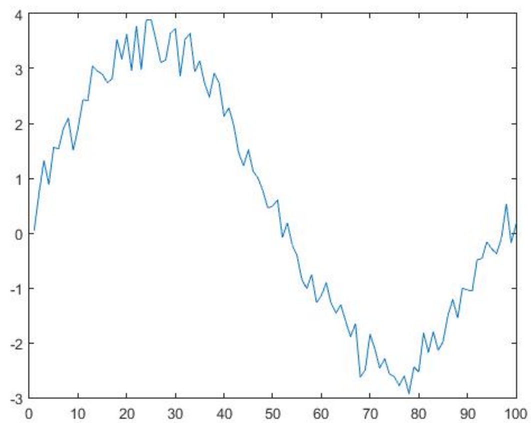

Figure10

9. Plot of random numbers after filtering using depth 4

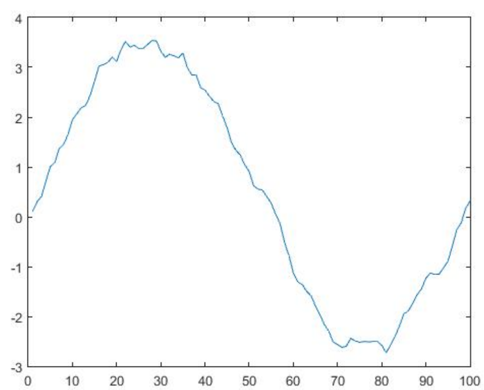

Figure11

10. Plot of random numbers after filtering using depth 7

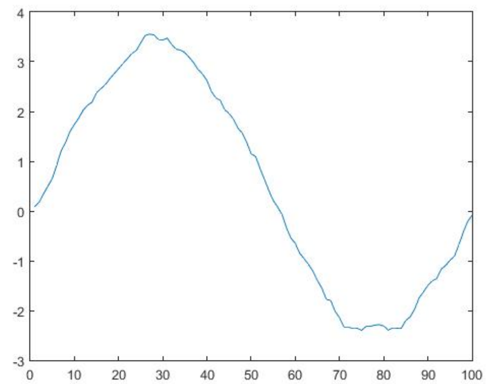

Figure12

11. Plot of random numbers after filtering using depth 14

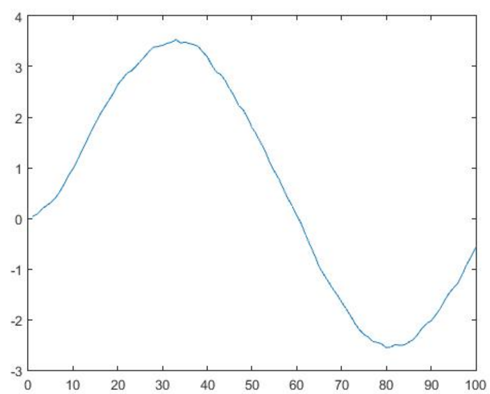

Figure13