ECSE 426 - Microprocessor Systems

**Lab 3&4 Report**

Group 15

Yan Ren 260580535

Fei Feng 260575129

1. ***Abstract***

For Lab 3 and Lab 4, the main goal is to design a system that generates a desired DC voltage based on PWM pulses, and use the ADC converter to read the voltage as a feedback to control the voltage level. When the user enters the desired voltage and press enter on the keypad, the 7-segment will display the current voltage value accordingly. For the keypad, # is used as an enter button, \* is used as a delete button, pressing \* for 1-2 seconds will restart the operation of each scenario at any time, holding \* for more than 3 seconds will put the system into sleep mode, and holding # for more than 3 seconds will wake up the system back to the operating mode. In either of phases or button presses, the display should be stable and ON except for the sleep mode. In sleep mode, the display should be OFF and the output voltage should be 0. For Lab 4, the requirements are mostly the same, but they will be achieved by building a multithreaded system using CMSIS-RTOS. Additionally, the power will be further reduced in sleep mode.

1. ***Problem Statement***

Based on the main goals of these two experiments, the objective is divided into following several major parts:

Properly setting up the PWM mode.

* Choose the PWM pulse frequency, resistor and capacitor values by calculation.
* Build the rectifier circuit, set up a PWM channel and apply it as input voltage of the circuit.
* Tune the duty cycle of PWM to make the desired voltage as output.

Feedback operation by using ADC.

* Use another timer as a source of sampling time for ADC.
* Calculate the sampling frequency based on PWM pulse frequency.
* Filter the data by FIR and choose proper filter parameters.

Setting up the keypad and maintaining the 7-segment LEDs display.

* Initialize a group of GPIO (General Purpose Input/Output) ports for keypad.
* Write the keypad scanning algorithm.
* Write the algorithm for handling button holds and bounce-free button presses.

Controller block implementation.

* Adjust the PWM pulse width based on the ADC reading values.
* Use state machine to switch between different modes.

For Lab 4, following problems are added based on the above statements.

* Implement all functions by using CMSIS-RTOS, the system should be multithreaded.
* Reduce the power in sleep mode as much as possible.

1. ***Theory and Hypothesis***
   1. *PWM pulse generation*

For generating the PWM pulse, we set channel 1 of TIM3 to PWM Generation CH1 by using STM32CubeMX. TIM3 mounts to APB1, and its clock frequency is 42MHz. However, since TIM has internal PLL which increases frequency by 2, the true clock frequency is 84MHz. According to equation (2) which is mentioned in section 3.3 below, the PWM frequency needs to be set to satisfy that R\*C\*f=10. Since we choose the 10 capacitor and resistor, we set the PWM frequency to 2500 Hz by calculation. Then, according to equation

(1)

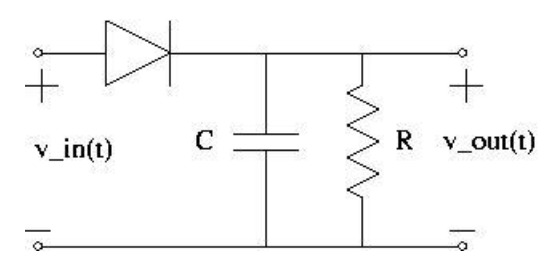
we set the period equals to 99, and prescaler equals to 335.

* 1. *ADC timer trigger*

By using STM32CubeMX we set another TIM2 to externally trigger ADC. We configure resolution = 8B, ExternalTrigConvEdge = rising edge, and ClockDivison = DIV1. Considering that ADC needs to acquire enough data from PWM to ensure the accuracy, the frequency of ADC has to be faster than the frequency of PWM. Therefore, we set the ADC frequency equals to 10000 Hz. Then we use equation (1) above again to calculate the period and prescaler for TIM2. We got period equals to 9, and prescaler equals to 839.

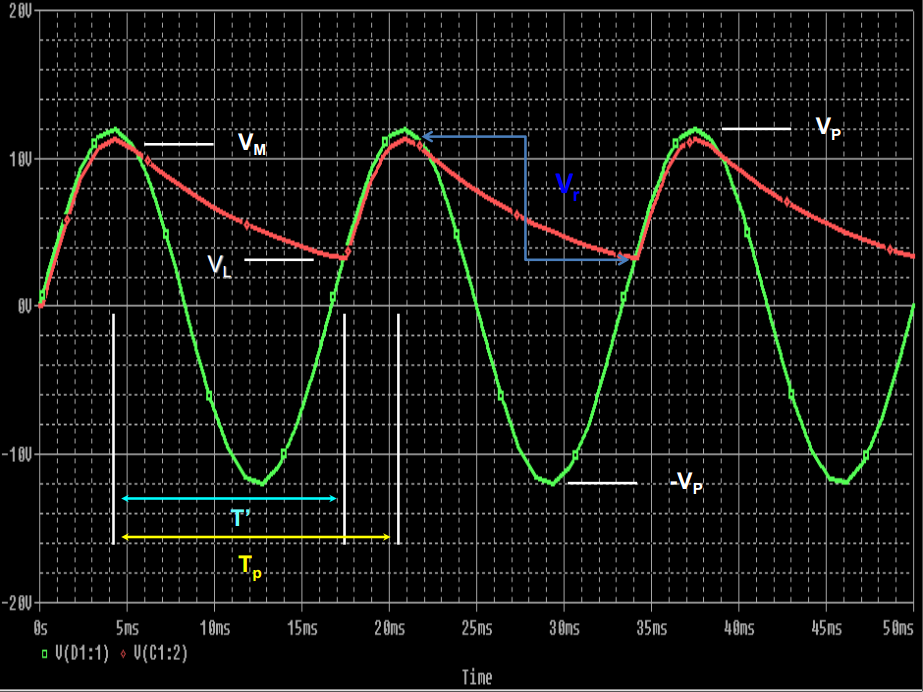
* 1. *Rectifier*

The rectifier circuit in this lab is implemented based on following:



*Figure 1 - Rectifier Circuit*

This is a typical design of Half-wave Rectifier with smoothing capacitor. A rectifier is a circuit which converts the Alternating Current (AC) input power into a Direct Current (DC) output power. The power diode in a half wave rectifier circuit passes just one half of each complete wave of the AC supply in order to convert it into a DC supply shown as below.

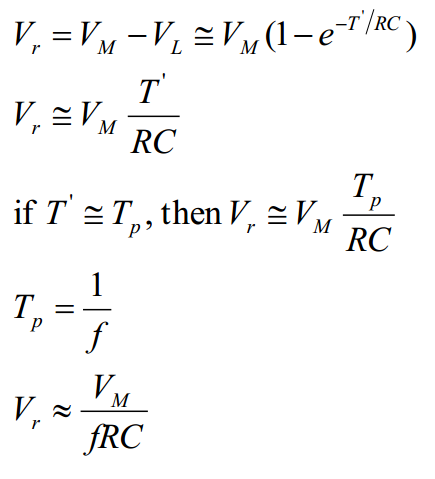


*Figure 2 - DC Supply Conversion [1]*

During the rectification process the resultant output DC voltage and current are therefore both “ON” and “OFF” during every cycle. As the voltage across the load resistor is only present during the positive half of the cycle (50% of the input waveform), this results in a low average DC value being supplied to the load [2].

The variation of the rectified output waveform between this “ON” and “OFF” condition produces a waveform which has large amounts of “ripple” which is an undesirable feature. The resultant DC ripple has a frequency that is equal to that of the AC supply frequency. Maximum range of fluctuations of the “DC” voltage is called Ripple Voltage(Vr).

Very often when rectifying an alternating voltage we wish to produce a “steady” and continuous DC voltage free from any voltage variations or ripple. One way of doing this is to connect a large value Capacitor across the output voltage terminals in parallel with the load resistor as shown below. This type of capacitor is known commonly as a “Reservoir” or Smoothing Capacitor. The smoothing capacitor, resistor, ripple voltage and output voltage satisfy the following[1].



We will use this relation

(2)

to determine the electric elements when building the rectifier circuit.

* 1. *Nested Vector Interrupt Controller*

The STM32f4 board supports up to 240 programmable Interrupts. Each one of them can be assigned a priority. The lower the priority number is, the higher the priority of that interrupt is. As a result, a signal with priority 0 is serviced before a signal with priority 1, if they happen at the same time. Furthermore, there are two types of priorities; preempt priority and subpriority. When two interrupts that have the same preempt priority happen at the same time, the one with the lower subpriority will be serviced first.

* 1. *Sensor calibration and filtering*

Raw data from the ADC can be filtered using an FIR filter. To find the right filter coefficients and order, we can use MATLAB to experiment on an array of values collected beforehand from the sensor. In general increasing the order of the filter results in a smoother graph where noise and small vibrations are eliminated. Reducing the order of the filter has the opposite effect. The value of the coefficients are used to control the impact of values when calculating the moving average.

* 1. *Detecting key presses on a 4\*3 keypad*

The technique used to detect which key was pressed on a simple alphanumeric keypad consists in first detecting to which column the pressed key belongs. To do this we first set the pins connected to the columns to logic 1 using pull-up resistors, and the pins connected to the rows to logic 0 using the pull-down resistors. Therefore, when a key is pressed, the pin of that specific column becomes logic 0. Once we know in which column the pressed key is, we set the rows to logic 1, and the columns to logic 0. As a result, the row that contains the pressed key will be read logic 0. This procedure can be done in a very short period of time, and one way to detect pins going from logic 1 to logic 0 is to continuously read them [3].

* 1. *Serial Peripheral Interface*

It is a synchronous serial communication interface for short distances. It consists of five logic signals: SCLK(serial clock), MOSI (data output from the master to the slave), MISO (data output from the slave to the master), SDIO (bidirectional input/output), SS (slave select).

* 1. *Multithreading*

Multithreading technique allows the processor to execute more than one tasks concurrently by quickly switching between processes. This technique is used widely in operating systems. It is especially efficient when threads are independent of each other. However, when more than one thread has access to the same resource, this technique is prone to error. Extra care must be taken in such situation. Moreover, communication among threads can be problematic and could cause locks.

* 1. *FreeRTOS*

Real time operating system serves real-time applications that process data as it comes in. Dependability is valued more than performance in such system. There are two major ways to design an FreeRTOS: Event-driven where tasks are handled according to their priorities and time-sharing, where each task is assigned with a fixed amount of time and they will take turns to be processed by the processor [4].

In FreeRTOS, intertask communication and resource sharing will define the way the scheduler work. One also need to keep in mind that accessing sharing data at the same time is unsafe and unwanted. There are three common approaches to implement the a dependable scheduler:

1. Message Passing:

When using message passing approach, sharing resources are only used by exactly one task at a time and threads will communicate using a global variable.

2. Masking:

Masking is another common approach. One of the processes can disable interrupts when it is at its critical sessions. Doing so will allow tasks to finish its process without having the risk of getting interrupted by a task that has higher priority.

3. Mutexes/ Semaphores

Mutexes are used for the purpose of preventing race conditions. It prevents threads from modifying a variable that is already being modified by another thread. When a thread enters its critical section and tries to access a shared resource, it first sets the mutex to 1, works on said resource, and sets the mutex back to 0 when it is done. If another thread tries to access that same shared resource while the mutex is set to 1, it will wait until it is set back to 0 [4].

Semaphores are similar to mutexes in the sense that they restrict access. They are mainly used to signal when a thread can access a resource or has to wait. Contrary to mutexes, they are not binary, and are best used when controlling access for multiple threads.

1. ***Implementation***
   1. *Selections of PWM pulse frequency, resistor, and capacitor*

The largest capacitor available in lab is 10. We use this as smoothing capacitor in rectifier. We chooseresistor and PWM pulse frequency to be 2.5 kHz.

By using the relation mentioned in section 3.3

We verify Vr is around 10% of the , which is within the acceptable range of the ripple voltage.

* 1. *TIM3 and PWM mode configuration*

To configure the PWM to be 2.5 kHz.

Parameters for the TIM\_HandleTypeDef structure [5]

|  |  |  |
| --- | --- | --- |
| Parameter | Option selected | Purpose |
| Instance | TIM3 | Select the timer |
| Prescaler | 336 | Used to reduce the timer frequency. |
| CounterMode | TIM\_COUNTERMODE\_UP | Count up |
| Period | 100 | Value that the counter will count to or from depending on if it is counting up or down. |
| ClockDivision | TIM\_CLOCKDIVISION\_DIV1 | Clock Prescaler |
| RepetitionCounter | 0 | No repetitions. |

Notice the Period is 100, therefore the range of adjusting PWM Timer Pulse is [0, 99].

Parameters for the TIM\_OC\_InitTypeDef structure

|  |  |  |
| --- | --- | --- |
| Parameter | Option selected | Purpose |
| OCMode | TIM\_OCMODE\_PWM1 | Clear counter when compared values match. |
| OCPolarity | TIM\_OCNPOLARITY\_LOW |  |
| Pulse | [0,99] | The value of the pulse is a variable that depends on the user input voltage. The range is [0,99], 0 is for 0% duty cycle, 100 is for 100% duty cycle. |

Finally, we call the HAL\_TIM\_PWM\_ConfigChannel() function 4 times (once for each channel, or LED) in a while loop, in order to constantly adjust the pulse. Each time the config channel function is called we must also call HAL\_TIM\_PWM\_Start\_IT() function.

* 1. *Choice of sampling frequency*

ADC and TIM2 configuration

ADC frequency is set to be 10kHz, which is higher than the 2.5kHz PWM frequency.

Parameters for the ADC\_ChannelConfTypeDef structure

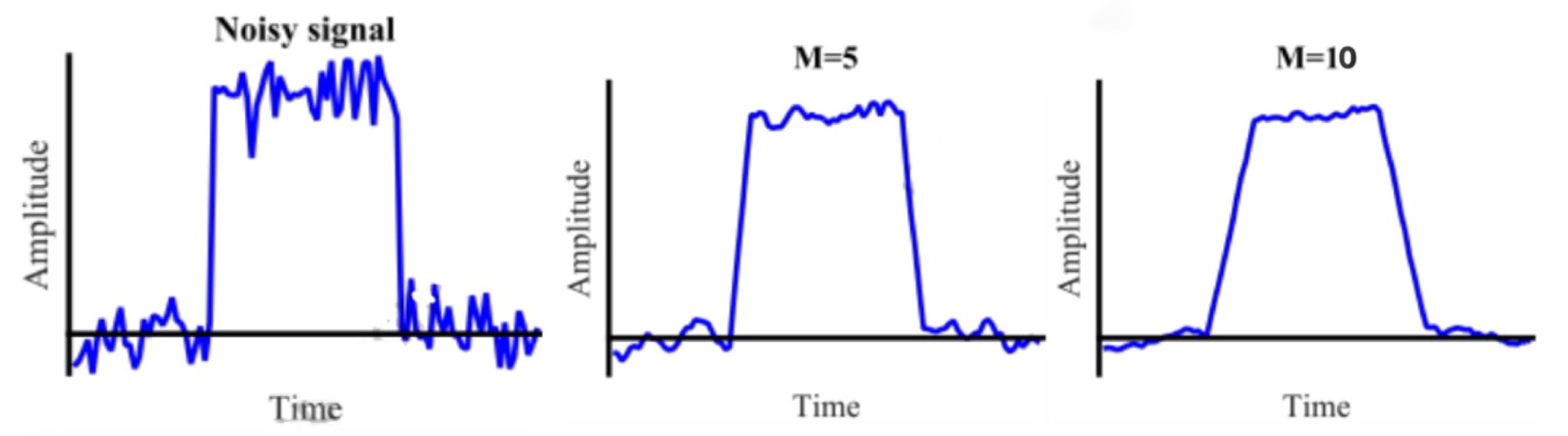
|  |  |  |
| --- | --- | --- |
| Parameter | Option selected | Purpose |
| ExternalTrigConvEdge | ADC\_EXTERNALTRIGCONVEDGE\_RISING | ADC conversion is triggered externally on rising edge |
| ExternalTrigConv | ADC\_EXTERNALTRIGCONV\_T2\_TRGO | ADC is triggered externally by timer2 |
| Resolution | ADC\_RESOLUTION\_8B | The value of the ADC is 8 bits |

Parameters for the TIM\_HandleTypeDef structure [5]

|  |  |  |
| --- | --- | --- |
| Parameter | Option selected | Purpose |
| Instance | TIM2 | Select the timer |
| Prescaler | 839 | Used to reduce the timer frequency. |
| CounterMode | TIM\_COUNTERMODE\_UP | Count up |
| Period | 9 | Value that the counter will count to or from depending on if it is counting up or down. |
| ClockDivision | TIM\_CLOCKDIVISION\_DIV1 | Clock Prescaler |

* 1. *Choice of filter parameters*

In Lab 3 and Lab 4, we use the same filter as in previous labs, which is a moving average FIR filter. According to [6], the impulse response h[n], which means the parameters of this filter, equals to 1/M, where M is the number of input signal samples. For choosing the number of input signal samples, we found that although a longer filter performs a better smoothing action, it translates into wider edges during the transition [6]. It means that output results may become inaccurate because of using a too long filter. Hence, we decide to keep the number of input signal samples as 5 instead of 10. Therefore we have five input signal samples, so all the five parameters are 1/5 = 0.2 in our case.



*Figure 3 - filters with different length*

* 1. *Graph of threads division*

Three threads in total with osPriority as normal for both have been implemented:

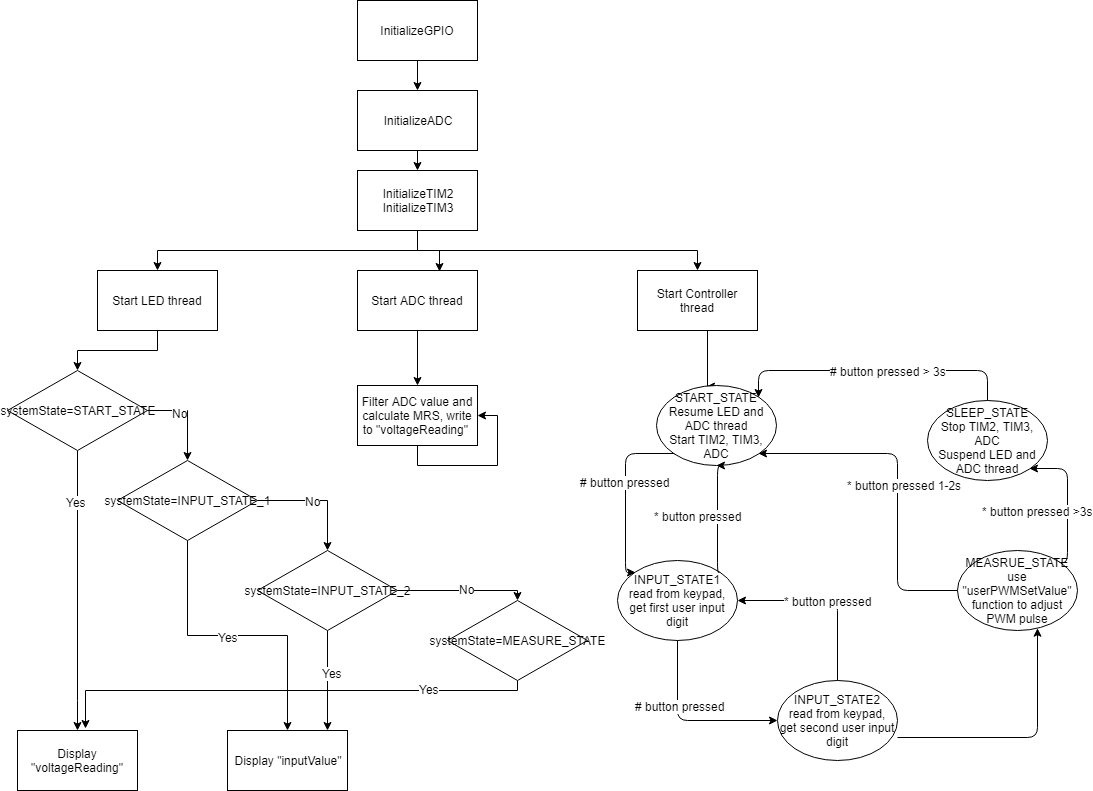
1. Controller Finite State Machine and Keypad
2. LED display
3. ADC

Detection of button pressed is implemented in controller thread and the value is stored in a shared variable locked by a semaphore. The thread and finite state machine diagram can be found on the next page.

Three threads are running in parallel. ADC data is constantly being acquired and stored in the background. LED display decides which data to present based on the state in controller thread.

Three global variables are shared among three threads. “volatile int systemState” stores the state in Controller thread. This data is written by Controller threads and read by LED display thread. “volatile float inputValue” stores the user input value from keypad. This data is written by controller threads and read by LED display thread. “volatile float voltageReading” stores the voltage readings from the ADC thread. ADC thread writes this variable and controller and display threads read this variable.

The access of “inputValue” and “voltageReading” is protected by semaphore “keyboardSem” and “voltageReadingSem”. Only one thread is allowed in the critical section of reading and writing the global variable.



*Figure 4 - Thread and state diagram*

* 1. *Reducing the power in sleep mode*

In the SLEEP\_STATE of our finite state machine, we use following command to stop the TIM2, TIM3 and ADC:

HAL\_StatusTypeDef HAL\_TIM\_PWM\_Stop(TIM\_HandleTypeDef \*htim, uint32\_t Channel)

HAL\_StatusTypeDef HAL\_TIM\_Base\_Start(TIM\_HandleTypeDef \*htim)

HAL\_StatusTypeDef HAL\_ADC\_Stop(ADC\_HandleTypeDef\* hadc)

We also use “osStatus osThreadSuspend (osThreadId thread\_id)” to suspend the LED and ADC threads. When exit the SLEEP\_STATE, start the TIM2, TIM3 and ADC, also use “osStatus osThreadResume (osThreadId thread\_id)” to resume the LED and ADC threads.

1. ***Testing and Observations***

To test the keypad function, we connect the keypad and 7-segment LEDs display. Then we try to enter all the numbers on keypad to see if the 7-segment shows the correct number we entered. After that, we test all functions of \* and # as required, including delete the number, restart the operation, sleep mode, and wake up feature.

To ensure each part is able to work properly during the experiments, we use oscilloscope to observe the pulse and voltage. For testing the PWM pulse, firstly we manually set different duty cycles in the code, then we use oscilloscope to observe whether the graph is changed properly. After making sure that PWM works well, we use the oscilloscope to test the output from rectifier. We also use it to test if controller block works, because when our code adjusts the duty cycle correctly, the diagram of PWM will be changed accordingly. Moreover, we use oscilloscope to make sure that when the system is in sleep mode, the output voltage is 0. For lab 4 we are using FreeRTOS, and it does not have event view support. However, we observe the LED is turned off in sleeping mode and since we do not intentionally set LED pins. Also the PWM output pin is 0V. We assume the LED thread is suspended and PWM timer is stopped.

After making sure that every section works well, we test the whole system together many times to see if there is any error occurs. We try to enter different voltage values within the range on keypad, and observe both the 7-segment display and oscilloscope display to verify that system works successfully.

1. ***Conclusion***

In the course of these two labs, we use STM32CubeMX to set timers for generating PWM pulse and triggering ADC. Then the rectifier circuit converts AC inputs to DC outputs. Polling technique is used in keypad function. ADC is used to read the voltage values, and controller block with state machines adjusts the duty cycle of PWM to give the desired voltage as entered. In lab 4 CMSIS-RTOS is used to implement the whole functions into a multithreaded system. The successful implementation of these two experiments, provided us with a solid introduction to using timer to generate PWM pulse and trigger ADC, polling technique for using keypad, and designing multithreaded system by using CMSIS-RTOS.

1. ***Reference***

[1] Filebox.ece.vt.edu, 2018. [Online]. Available: https://filebox.ece.vt.edu/~LiaB/ECE2204/Lectures/Diodes/Half-Wave\_Rectifiers.pdf. [Accessed: 20- Mar- 2018].

[2] 2018. [Online]. Available: https://www.electronics-tutorials.ws/diode/diode\_5.html. [Accessed: 20- Mar- 2018].

[3] "Siddharth Chandrasekaran - EmbedJournal", Embedjournal.com, 2018. [Online]. Available: https://embedjournal.com/authors/siddharth-chandrasekaran/. [Accessed: 20- Mar- 2018].

[4] "Why RTOS and What is RTOS?", Freertos.org, 2018. [Online]. Available: https://www.freertos.org/about-RTOS.html. [Accessed: 20- Mar- 2018].

[5] "STM32F4 PWM tutorial with TIMERs - STM32F4 Discovery", STM32F4 Discovery, 2018. [Online]. Available: https://stm32f4-discovery.net/2014/05/stm32f4-stm32f429-discovery-pwm-tutorial/. [Accessed: 20- Mar- 2018].

[6] “Moving Average Filter”, wirelesspi.com, 2018. [Online]. Available: http://wirelesspi.com/moving-average-filter/ [Accessed: 19-Mar-2018].