ECSE 426 - Microprocessor Systems Timers, Interrupts, Threads, and Real Time Operating Systems

(Laboratory Assignment 4)

Isaac Chan*, Tiffany Wang†
*School of Electrical and Computer Engineering
McGill University, Montreal, Quebec
Email: isaac.chan@mail.mcgill.ca
Student ID: 260624096
†School of Electrical and Computer Engineering
McGill University, Montreal, Quebec
Email: tiffany.wang@mail.mcgill.ca
Student ID: 260684152

***Abstract*—In this report, we demonstrate the STM32F407's capability to output a pulse-width modulation (PWM) signal and through the capabilities of well-defined threads, timers, and interrupts, read the DC-mean of this voltage signal and display it on an LED board. We show two separate methods of completing this task, the first involving the careful organization of a variety of interrupts and timers, the second using a multithreading approach in order to maximize efficiency and minimize power consumption. We also demonstrate the program's ability to adapt to discrepancies between the input and output voltage by modifying the duty-cycle of the PWM signal.**

I. Problem Statement

The primary objective of this experiment is to be able to output a variable DC-mean voltage through the PWM pin, and to be able to read and interpret this voltage through an Analog-to-Digital (ADC) pin (Walden 1999). We would sample the analog signal at a rate of 50 Hz, and pass it through a Finite Impulse Response (FIR) filter (Cetin et. al 1997) as the read signal would have an inherent noise component. The filter's coefficients were defined via the window design method, and this technique was chosen in order to scale the magnitude of the filtered data to a more comprehensible size. We then take the measured data and compare it to the input data, whereupon we adjust the duty_cycle such that the measured data is now coinciding with the input data.

The primary challenge faced in this experiment was the implementation of interrupts and thread. The difficulty of balancing a variety of threads and modifying their various priorities allows for many mistakes to be made in terms of resource consumption. This issue was particularly prevalent when attempting to balance the concurrency of the ADC thread, the PWM thread, and the LED thread as only specific configurations of their priorities and the use of their timers would allow all 3 to function. The solution to this problem was to allow all 3 of those threads to have identical priorities, as the interleaving of the 3 threads to their respective resources allow them to function without dominating the processor. A system level block diagram is shown in Appendix A.

II. Theory and Hypothesis

This system hinges on multiple signal processing, microprocessor, interrupt, timer, and operating system theories to function. We detail their functional dynamics below:

A. Multithreading

Multithreading is the ability for a processor to execute multiple tasks separately. Threads differ from processes in that multiple threads share the same resources (whilst holding certain protected memory spaces in RAM) while processes allocates resources during initialization. This allows for hardware that has limited resources and processing power to still have multiple tasks execute. The main advantage of multithreading is that if there are cache misses, other threads can take advantage of unused resources. Additionally, if a thread isn't consuming resources at any moment, other threads would take advantage of those resources whereas the alternative would be to leave them idle, hence increasing overall processor productivity. A visuald description of multithreading is shown in Figure 1

B. Timers

Every microcontroller inherently has a set of timers embedded in the system. The timers can count from 0 to 255 (8-bit system) or 65535 (16-bit systems). In order to accommodate for the possibility of counting up to larger values, a prescaler is typically used to adjust counters. The prescaler value determines how often a timer increments. For example, if a prescaler value is set to 64, then the timer associated with that prescaler value would only increment itself once every 64

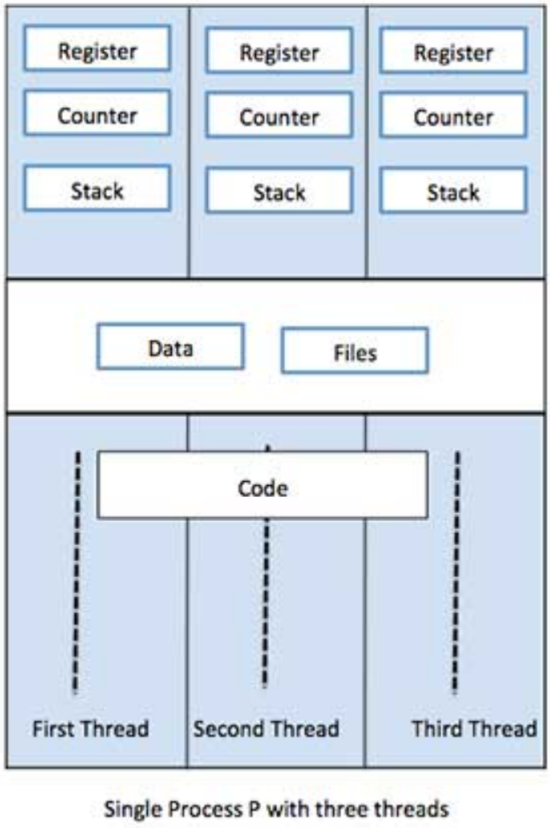

Fig. 1. Multithreading Visual Depiction

clock cycles, thus allowing the timer to account for values 64x larger than its original limit. Therefore, the prescaler value, combined with the limit of the counter and the frequency of the clock is what is used to determine the length of the timer.

C. Pulse-Width Modulation

Pulse-Width Modulation (PWM) is a modulation technique used to encode a message in a pulsing signal. The two primary uses of PWM is to encode a signal and to control the DC-mean of an output voltage. For example, should a microcontroller only be able to output 3V or 0V, and the user desired a 1.5V DC-mean, PWM could be used such that the microcontroller would modulate between a 3V output 50% of the time and a 0V output 50% of the time, thus creating a 1.5V DC-mean. The value that would be read from a PWM signal is therefore calculated as:

$$\bar{y} = \frac{1}{T}\int_0^T f(t)dt \tag{1}$$

where f(t) is the frequency of the pulse width with period T, consisting of low value y_min, high value y_max, and a duty cycle D ϵ [0, 1](illustrated in Figure 2). And so, we expand

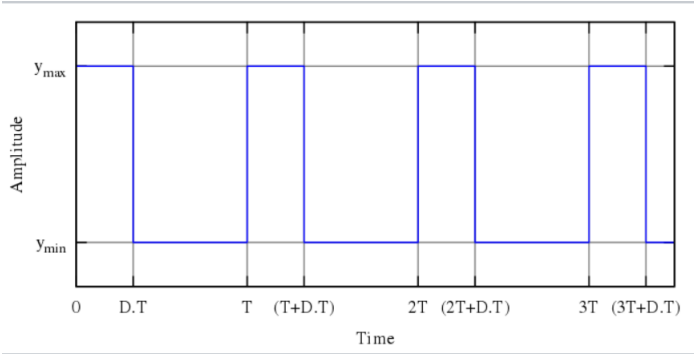

Fig. 2. A standard PWM signal, illustrating the use of the duty cycle

the expression to consist of

$$\begin{aligned}\bar{y} &= \frac{1}{T}(\int_0^{DT} y_{max}dt + \int DT^T y_{min}dt) \\ &= \frac{1}{T}(D*T*y_{max} + T(1-D)y_{min}) \\ &= D*y_{max} + (1-D)*y_{min}\end{aligned} \tag{2}$$

Where the final equation assumes a perfect square wave (ideal case only, as a preliminary examination of the fourier transform would demonstrate that the output wave would not be an ideal square wave, and thus, only the integral form of the formula can be used for the PWM calculation)

D. Analog-to-Digital Conversion

An analog-to-digital converter converts an analog signal (Walden et. al 1999) (in this case - voltage), to a floating point precision digital signal. As this process requires the quantization of a continuous input, it by definition has to introduce loss of information. Additionally, as the ADC does the conversion periodically by sampling the input, it limits the bandwidth allowable bandwidth of the input signal.

The performance of an ADC is quantified by its bandwidth, resolution, and signal-to-noise ratio. In our instance, as the analog signal was converted from the board's own digital signal, the bandwidth is trivially fully comprehensive of the information. We measure the dynamic range as a function of the resolution and the effective number of bits (ENOB) as a measure of the signal-to-noise ratio. We compute the resolution Q with the following formula

$$Q = \frac{E_{FSR}}{2^M} \tag{3}$$

where E_{FSR} is is the full scale voltage range, which is given by

$$E_{FSR} = V_{RefHi} - V_{RefLow} \tag{4}$$

where V_{RefHi} and V_{RefLow} are 3 and 0V respectively, and where M is 10 bits.

We also alleviate the issue of the signal-to-noise ratio by passing the ADC information through a FIR filter, which is described below.

E. FIR Filter

A Finite Impulse Filter, or an FIR filter, is filter whose response to an impulse is of finite duration as it eventually converges to 0. This effectively parameterizes the digital interpretation of an analog signal such that it reduces the signal to noise ratio.

In our causal discrete-time system, the FIR filter of order N that we apply can be summarized with the following:

$$\begin{aligned} y[n] &= b_0 x[n] + b_1 x[n-1] + ... + b_N x[n-N] \\ &= \sum_{i=0}^{N} b_i * x[n-i] \end{aligned} \tag{5}$$

whereupon we can see that this operation is a discrete convolution function. This filter brings about multiple advantages.

1) The filter is inherently BIBO stable
2) Requires no feedback, and as a result, the relative error of each of the outputs is the same as rounding errors would not be compounded through summation

The coefficients b_i were chosen using the Window design method, where we scaled the coefficients based on the size of the window such that the output of the filter is on the same level of magnitude as its input.

F. Interrupts

There are effectively 2 methods for peripherals in a system to interact with the CPU. Polling or interrupts (Corbet et. al 2014). Polling, while much simpler to implement, has one fundamental flaw: When the CPU requires the peripheral to make a computation, the CPU will continuously poll the peripheral until the computation is completed, and will therefore not make any other computations in the interim. However, an interrupt method will allow the CPU to move onto other tasks until the peripheral is completed with its computation, by which time the peripheral will send an interrupt to the CPU and pass along the information that the CPU has requested. The interrupt system allows for no wasted clock cycles whereas the polling system will experience a lot of latency due to its fundamental inefficiency.

III. Implementation Testing

In this section, we will discuss the following 8 components of our system, and we will present the methodologies used to implement the system, issues that we encountered during its implementations, and the solution that was used to remedy the problem.

1) ADC
2) PWM
3) Keypad
4) Threads

A. ADC

The value we read from the ADC would be between 1 to 1024 bits, and is evenly spaced between 0 and 3V, and therefore, a conversion is necessary. The equation that we use to show this conversion is shown below

$$DV = \frac{AV * 3}{1024} \tag{6}$$

where DV is the Discrete Voltage, and AV is the Analog Voltage, we multiply by 3 as that is the VDD, and divide it by 1024 as that is our resolution. The ADC was implemented by using the TIM1 channel as the rising edge trigger. We set the frequency of the ADC sampling to be less than that of the PWM (and that of an indivisible value, for that matter) such that we could ensure that the samples obtained from the PWM signal would be independent and identically distributed, thus ensuring a proper calculation of the DC-mean value.

B. PWM

The timer that is set for the PWM display is set on channel 1 on TIM3, which is the PWM signal. The PWM signal itself is set on 750MHz, and a high frequency was chosen because it would allow for greater sampling accuracy. This is equivalent to providing a higher number of samples, which would in turn reduce the standard of the point-estimator for the mean, which is calculated as:

$$\sigma_\theta = \sqrt{\frac{pq}{n}} \tag{7}$$

Where we assume an ideal square wave, which leads to a Bernoulli Random Distribution. The p and q=(1-p) values are therefore dictated by the duty cycle.

This was possible because we had restricted the values of our capacitor and our resistor values to 10uF and 390 ohms respectively, which allowed our frequency to be more flexible as the frequency calculation is computed as such:

$$f = \frac{1}{2 * \pi * R * C} \tag{8}$$

.

Where we determined the prescaler value of 84000 by the following equation

$$PRESCALER = \frac{84MHz}{DesiredFreq * Period} + 1 \tag{9}$$

C. Keypad

The keypad was handled with a state-machine controller that had 3 separate states (Wait, output, sleep). The keypad itself was an active low system, whereupon pressing the button would provide the board with a 0 voltage whereas the default non-pressed signal would equal the VDD voltage. We set the row pins to output and into push-pull mode, and the column pins to input and pull up. In order to prevent any button from being pushed multiple times, we added

a debouncer module that would not allow the value to be re-recorded in the system until a set number of iterations had been passed. It is important to note that the keypad will not function unless the user enables the pull-up resistors, as the algorithm is based on detecting the changes from 1 to 0 and recording the patterns. The states that were utilized for the keypad are detailed here

1) Wait: This was the default state of the keypad, whereupon it was waiting for an input from the user. An update_voltage function was used to keep track of the values that the user enters, and is the implementation of the queue that would delete the oldest digit and place the newest digit in the place of lowest magnitude (the 10^{-2} value in our case), and the star key would be used to delete the last digit. The pound key would then enter that voltage and that voltage would be used to compute the duty cycle that is used in the PWM output.

2) Output: The conversion between an input voltage and duty cycle was computed using a linear regression technique, whereupon we used an oscilloscope to collect sample data points that drew a relationship between duty cycle and voltage values. The linear regression would therefore compute weights that would allow us to draw a "translation" function that would translate our input voltage to a relevant duty cycle. The linear regression problem was defined using an MSE error function as shown:

$$MSE(w) = \frac{1}{n}\sum_{i=1}^{n}(w^T x - \hat{Y}_i)^2 \tag{10}$$

And the linear regression problem is defined as

$$\hat{Y} = w_1 * x + w_0 * \tag{11}$$

This in turn generates a convex optimization problem, where the proof of convexity is omitted. However, as this problem is now a closed and bounded convex optimization problem, we can guarantee that there is a global minimum solution. Furthermore, as we can see that the quadratic term allows for this function to be strictly convex, we can be further assured that this optimization problem has a unique global minimizer. Additionally, the global minimizer can be found through a simple rearrangement of terms, whereupon we attain a closed form solution for the optimization problem as such:

$$w = (X^T X)^{-1}(X^T Y) \tag{12}$$

Using those weights, we are able to create a linear function (which is a fair assumption given the nature of duty cycle and how it relates to voltage - refer to section 2c).

After the duty cycle is set and the output is fed through the PWM pin, the state machine returns to the wait state and awaits the next subsequent input

3) Sleep: The sleep state activates after the star key is pressed for 3 seconds and is utilized as a power reduction function. In this state, the LED, PWM, and ADC modes are all turned off and the only thing that's left on is the keypad pin that would await the star key (pressed for 1-3 seconds) to take it back to the wait state. As a result, the star key had to have a counter to calculate how long it had been pressed for, and a release debouncer to ensure that there were no false negatives in the system.

D. Multithreading

This section is specially reserved Lab 4, when we proceeded to use multithreading to complete this task, we utilized 4 different threads

1) PWM: The PWM thread ran on a normal priority and continuously checked to see if the read voltage is within 5% of the desired voltage and adjusts the duty cycle accordingly. In this adjustment, we utilized the osSignalWait function to update the signal and that is what allowed the threads to interleave, therefore only adjusting after the ADC signal is set, and therefore, the PWM adjust will not pre-empt the ADC reading the output.

2) ADC: The ADC thread ran on a normal priority and was run on a polling mode (and not on an interrupt mode). Instead, the ADC pin was initialized and deinitialized at each reading in order to maximize power efficiency. The ADC pin read every 20 ms from the PWM signal as a 50Hz system had proved functional in the past. Finally, as this ADC ran on an 8-bit system, a different conversion was required this time and is shown below

$$DV = \frac{AV * 3}{256} \tag{13}$$

3) Keypad: The keypad thread was given the highest priority as the user input took precedence above all other utilities in the system. As a result, the keypad was constantly for user input, and the implementation for the most part is identical to the Lab 3 implementation (as described above). However, we used the osDelay function for the debouncer instead to keep in line with the FreeRTOS utility functions. In the sleep mode of the keypad function, the other three threads are suspended in order to maximize power efficiency - whereupon we would deinit the ADC, PWM, and all the LEDs to save power. When we exit sleep momde, we would reinit those services. The keypad, as a result of this implementation, necessitated a polling implementation and not an interrupt implementation.

4) LED: The LED continuously had 3 digits interleaving, representing the 1s place, the 10^{-1} place, and the 10^{-2} place. The LED would continuously output the voltage, but would only update the voltage read from the ADC every 5 ADC iterations, equivalent to 100ms (or 0.1s).

5) Memory Allocation: To prevent stack overflow, we allocated 128 bytes of memory to the PWM, ADC, and LED threads and 256 bytes of memory to the Keypad thread. Giving this protected memory to each of the threads allowed them to have sufficient protected memory for their functions and therefore prevented any unwanted overwriting of information.

A full block diagram of the implementation of the multithreading system is shown in Appendix A. The thread diagram could not be displayed as the Keil Debugger system has known issues with the FreeRTOS operating system.

IV. Conclusion

In this experiment, we developed 2 systems of utilizing PWM, ADC, and LED outputs. The first was the more obvious method to use timers, and while the implementation was more straightforward, it has clear inefficiencies in both memory usage and resource allocation. These inefficiencies are recmedied in lab 4, when we used a multithreading approach to tackle the problem. The ability of multithreading to minimize idle resource time, to use protected and shared memory resources, and to interleave processing allows it to be the most efficient implementation currently available for a task such as this one

References

[1] A. E. Cetin, O.N. Gerek, Y. Yardimci, "Equiripple FIR filter design by the FFT algorithm," IEEE Signal Processing Magazine, pp. 60-64, March 1997.

[2] Cartwright, Kenneth V. "Determining the Effective or RMS Voltage of Various Waveforms without Calculus". Technology Interface. 8 (1): 20 pages, Fall 2007

[3] Walden, R. H. . "Analog-to-digital converter survey and analysis". IEEE Journal on Selected Areas in Communications. 17 (4): 539550 1999.

[4] ARM, Application Note 179 - Cortex M3 Embedded Software Development, http://infocenter.arm.com/help/topic/com.arm.doc.dai0179b/AppsNote179.pdf, March 2007

[5] Jonathan Corbet; Alessandro Rubini; Greg Kroah-Hartman (2005). "Linux Device Drivers, Third Edition, Chapter 10. Interrupt Handling", Dec 2014

[6] Dennis Dempsey and Christopher Gorman, "Digital-to-Analog Converter," U.S. Patent 5,969,657, filed July 27, 1997, issued October 19, 1999.

Appendix A
Multithreading block diagram

The block diagram of the multithreading system is shown in Figure 3

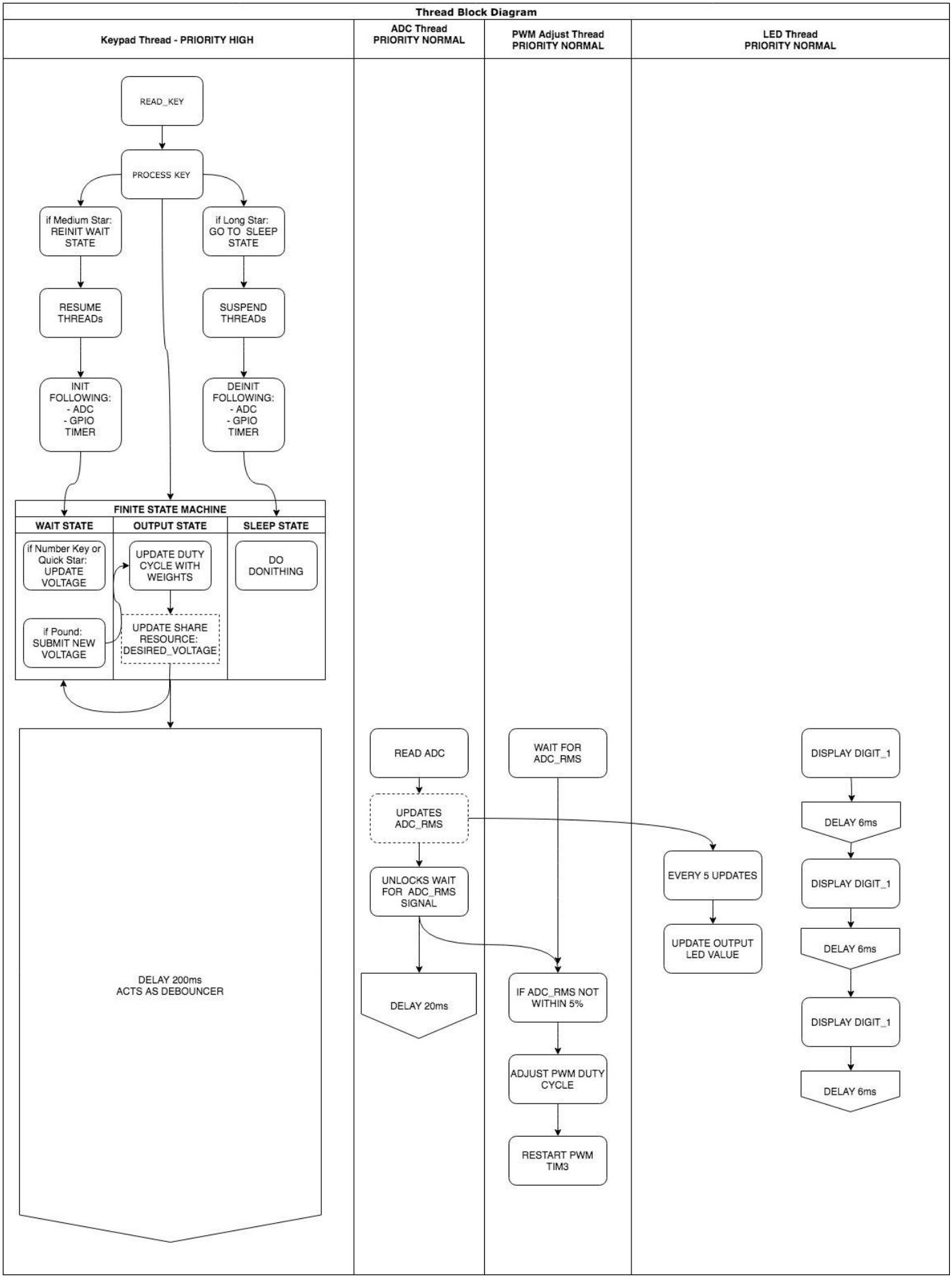

Fig. 3. Multithreading Block Diagram