ECSE 426 - Microprocessor Systems
Lab Report 2: Timers, Interrupts, Multithreaded, Interrupt-Driven Readings and Peripheral Control

Harley Wiltzer (260690006)
Matthew Lesko (260692352)

March 19, 2018

Contents

1 Abstract . . . 1
2 Problem Statement . . . 1
3 Theory and Hypothesis . . . 2
3.1 Rectifier . . . 2
3.2 Multithreading and Semaphores . . . 2
3.3 Hypothesis . . . 2
4 Implementation . . . 3
4.1 PWM Pulse Generation . . . 3
4.2 ADC and Timers . . . 3
4.3 Filtering . . . 3
4.4 Alphanumeric Keypad . . . 4
4.5 Finite State Machine . . . 4
4.6 Controller . . . 5
4.7 Multithreading . . . 5
5 Testing and Observations . . . 5
5.1 PWM and Rectifier . . . 5
5.2 Filtering . . . 5
6 Conclusion . . . 6

A GPIO Configuration Parameters 7

B ADC Configuration Settings 9

C TIM2 Configuration Settings 10

D TIM3 Configuration Settings 11

E HAL Cube MX Autogenerated Code 12

F HAL GPIO Pin Configuration 13

G Theory References 14

List of Figures

1 Finite State Machine 5
2 Filtered and Unfiltered Values VS Time (ms), Using Unmodified FIR Filter 6
3 Filtered and Unfiltered Values VS Time (ms), Using High Coeffcients for Early Values 7
4 Filtered and Unfiltered Values VS Time (ms), Using High Coeffcients for Late Values 8
F.1 HAL's Visualization of the STM32F4's Pins Configuration 13

List of Tables

1 Abstract

The purpose of experiment 3 is for the programmers to gain experience in utilizing timers and interrupts to accomplish a task which involves converting an analog pulse to digital and displaying its voltage on an LED display, effectively a voltmeter. The purpose of experiment 4 is for the programmers to gain exposure in designing a multithreaded application on a real time operating system (RTOS) running on an embedded system. The task of experiment 4 involves copying over experiment 3's program and subdiving several of its features each to its own concurrently running thread, with the goal of optimizing power usage. This report will explain in detail how the programmers implemented the problems stated below, as well as the challenges they faced, the testing they had done, and the conclusions they have made. By the end of the report, the reader shall understand how the timers available on the STM32F4 board can be used to activate peripherals and generate a pulse, and understand the implementation of multithreaded programs on embedded systems.

2 Problem Statement

The problem is for the developers to first implement a solution for generating a PWM pulse, whose voltage is set by an input on a keypad, that is then fed to a rectifier. Afterwards, the device shall feed the rectifier's output to an ADC to be converted to a digital signal, and have the signal's mean voltage be automatically displayed on an LED display. Furthermore, the program has to be implemented with the use of concurrently-running threads running on an RTOS. The problem can be divided into the following tasks:

- Setting up a timer to act as a PWM pulse generator;
- Setting up a timer to activate the ADC to take an analog sample and convert it to digital;
- Designing a rectifier circuit component that takes the PWM pulse as input and feeds the output to the ADC;
- Testing and optimization of an FIR filter that reduces noise from the output signal of the ADC;
- Setting up the alphanumeric keypad so that the user may input their desired voltage to be displayed on an LED display;
- Mainting the 7-segment display;
- Coding a controller function that automates the changes to be made on the PWM's duty cycle so that the previous or default voltage updates to the target voltage on the LED display;
- Coding a finite state machine function that enables the user to Enter and Delete digits for the target voltage, Reset the target voltage, and put the device to Sleep by using the keypad;
- Implementating the program's features using CMSIS-RTOS and multithreading;
- Reducing the power consumption of the device when it is in sleep mode using CMSIS-RTOS.

3 Theory and Hypothesis

Since this experiment uses components from experiment number 2, any theory for the identical components that has been mentionned in the previous lab report will be skipped. If you need to see theory for those components, and it is not mentionned here, please refer to Lab Report number 1.

3.1 Rectifier

A rectifier is a component that converts alternating current (AC) to direct current (DC). It does so by only allowing a one-way flow of electrons, by the use of a diode. The diode allows electric current only in the forward bias condition and blocks electric current in reverse bias condition, this allows it to act like a rectifier. The output of the diode only contains a positive half cycle, as opposed to the input having a positive and a negative half cycle [1]. The rectifier component of this experiment's device contains a diode connected in series with a parallel connection of a resistor and a capacitor.

3.2 Multithreading and Semaphores

The idea of multithreading is to have multiple threads execute concurrently. This allows for parallelism, more effecient use of a processor, and faster execution time. This is similar to the notion of concurrently-running processes, however, threads are not processes. One can have multiple threads running in one process, thus allowing for less overhead because the threads all share the same data [2]. Since, the threads may be sharing resources, this requires some kind of mutual exclusion to prevent threads from falling into deadlock or race-condition situations. The solution is semaphores. Semaphores allow one program to use a shared resource without having other programs use the resource at the same time. This is done by having a program wait until a semaphore is available to use before it can have access to the shared resource and execute its code. First a program waits for a semaphore, and once it is available, holds the semaphore by decrementing the semaphore's count, and executing its code. If the semaphore has a value of 0, no other program that is waiting for the same semaphore can execute. When the program using the shared resource no longer has need for the resource, it releases the semaphore by incrementing the semaphore's count, thus allowing other programs that are waiting to execute, to use the resource [2]. This resolves the problem of deadlocks and race-conditions.

3.3 Hypothesis

The programmers should be able to implement a solution for experiment 3 given the time allocated for the project. The challenges one expects to face are learning how to use the keypad and implementing a solution with it, testing of the resistor and capacitor components for the rectifier, and implementing a solution for the different features available on the keypad with the use of a finite state machine. The programmers should be able to implement a solution for experiment 4 given the time allocated for the project, since the majority of the device and code is the same from experiment 3 and the students should have had experience with multithreading and semaphores.

4 Implementation

4.1 PWM Pulse Generation

The developers were tasked with designing a system that generates PWM pulses and feeds the voltage to a rectifier circuit. Under the TIM3 configuration settings, a timer, one can see that it is configured to generate a pulse at a frequency of 1MHz, by inheriting the system's clock of 168MHz and having a period of 168, hence having 168 MHz divided by 168, yielding 1 MHz. The students have chosen to use this frequency after having tested lower frequencies, and came to the conclusion that in order for the duty cycle to control the output voltage, the frequency had to be elevated to at least this amount of frequency. After testing and observing, the students noticed that a frequency of 1 MHz was suitable enough to have the duty cycle control the output voltage. The STM32F4's TIM3 hardware timer is configured to generate and output a PWM pulse channel that is assigned to a GPIO pin on the board, which can be viewed in the GPIO pin configuration table generated by STMCubeMX. By using a circuit wire, the programmers can feed the output of the PWM pulse channel to a circuit component on a bread board, more specifically, the rectifier. The rectifier circuit holds the charge in the capacitor while a load resistance discharges the capacitor, enabling the user to influence the duty cycle to control the output voltage level. The required range for the output voltage is between 0.5V and 2.8V, hence it is required to test different configurations for the resistor and capactitor to deliver a range of output voltages that meet the requirement. The students have chosen a capacitor of 5uF and a resistor of 390 Ohms. This configuration was chosen because it achieved an output voltage range of 0.4V to roughly 2.1V. The programmers were able to observe this phenomenon after testing multiple different configurations and came to the conclusion that the resistor and capacitor values chosen were suitable for the task.

4.2 ADC and Timers

Previously, the students used STM's built-in SysTick to enable the ADC. For this experiment, the students had configured the ADC to be triggered by an STM32F4 timer, specifically TIM2. Under the TIM2 configuration settings, one can see that the timer is configured with a prescaler of 83999 and a period of 1, which by inherting the system clock of 168 MHz, entails the timer to a frequency of 500 Hz. The reason behind this decision is so that the ADC gets activated frequently enough to take samples more often. This would allow it to be a more responsive component and allow the output voltage's RMS to be updated more accurately. One can see under the ADC's configuration settings that the trigger for the ADC's activation is the TIM2's trigger event.

4.3 Filtering

The devices uses a modified FIR filter from experiment 2 in order to reduce the noise of the ADC's signal. The two modifications are the following:

- Increasing from 5 coefficients to 10 being used in the moving average;
- Setting the first five coefficients to 0.05 and the last five coefficients to 0.15.

In this way, the 5 earliest samples hav have a significance of 0.05 and the 5 later samples have a significance of 0.15 when calculating the average of the 10 samples. The programmers drew this conclusion by trial and error and the empirical evidence proved that the above modifications to the FIR filter significantly reduced the signal's noise. The programmers have tested mutliple configurations, in which this report shall demonstrate three of the tested configurations. The graphs the programmers have used to determine the optimized filter can be seen in the Testing and Observations: Filtering, section. After testing multile configurations, the three that are graphed in the testing section further prove that the programmers have chosen an accurate filter with coefficients: [0.15, 0.15, 0.15, 0.15, 0.15, 0.05, 0.05, 0.05, 0.05, 0.05].

4.4 Alphanumeric Keypad

In order to allow a user to either control the output voltage of the rectifier circuit or put the system to sleep, it was required to integrate a keypad peripheral. The keypad has buttons arranged in a 4×3 matrix which is interpreted via the 7 pins that the keypad offers. In order to read a given column of the keypad, one may set the keypad's corresponding column pin `HIGH` and subsequently read all four row pins. The reading of the four row pins and the column pins are passed through a lookup table (implemented in software) which returns the corresponding key that was pressed (assuming only one key is pressed at any given time).

The overall strategy for processing keyboard input is as follows:

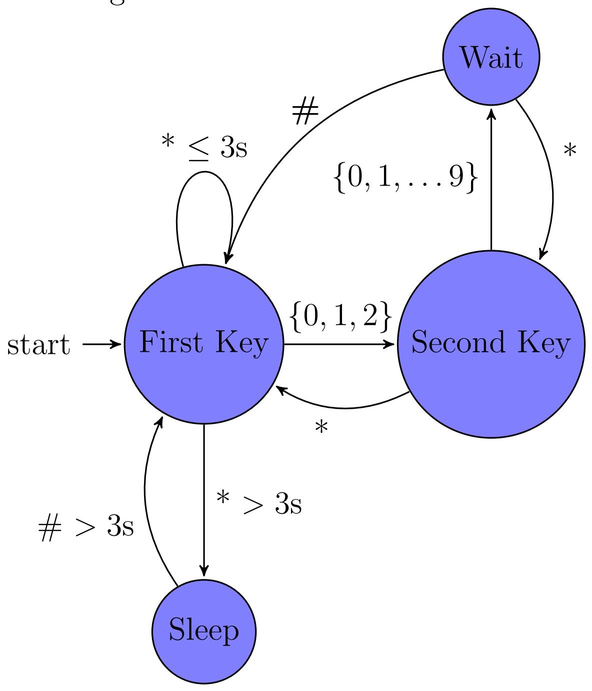
1. Set the pin corresponding to the first keypad column HIGH.
2. Poll the row pins for a short period (on the order of 100ms).
3. If the row pins indicate that a button was pressed, search lookup table for the appropriate button reading and latch it in a `reading` variable
4. Repeat steps 2 and 3 for each of the remaining column pins.
5. Once all column pins have been read, alert other tasks of the `reading` variable (via a signal in the context of an RTOS, for example) and reset `reading = -1`. Note that it is important to alert other tasks when no button has been pressed as well (indicated by `reading = -1`) in order to distinguish between a user pressing a button and a user holding a button.
6. Repeat

The above procedure assumes the keypad controller is operating concurrently with the rest of the system, which was true in the case of Lab 4 which made use of an RTOS. However, during lab 3, this was not at all the case. Since no signals could be sent, the finite state machine would wait for the whole keyboard to be scanned before computing its next state. Clearly, this is rather inefficient as the finite state machine would, first of all, have to wait approximately 300ms before each state transition. Secondly, since most of the time the user is not pressing a button, the finite state machine would end up polling for button presses needlessly for the majority of the CPU time. With the threading and signal implements in FreeRTOS, this efficiency was increased greatly. In the implementation of Lab 4, as discussed below.

4.5 Finite State Machine

A finite state machine was implemented to control the functionality of the system according to the state the system is in. The state is changed by inputs to the keypad from the user.

Figure 1: Finite State Machine



4.6 Controller

4.7 Multithreading

5 Testing and Observations

5.1 PWM and Rectifier

The students tested multiple configurations of resistor and capacitor values for the rectifier circuit. Here were their findings: One can see that having a high resistor (390 Ohms) and low capacitor (5uF) allow the output voltage to yield a range between 0.4V and 2.1V, making it the best suitor for the requirement of delivering a wide range of output voltages.

5.2 Filtering

One can draw comparisons from the following graphs generated by a run-time variables monitoring and visualization tool, STM Studio [source]:

For the graph above, the configuration for the filter's coefficients is [0.2, 0.2, 0.2, 0.2, 0.2]. In regular blue are the unfiltered values, and in light blue are the filtered values. This graph demonstrates the values read at runtime of the unfilted and filtered values that passed through the unmodified FIR filter. One can observe that there is a considerable amount of noise left from filtering.
For the graph above, the configuration for the filter's coefficients is [0.05, 0.05, 0.05, 0.05, 0.05, 0.15, 0.15, 0.15, 0.15, 0.15]. This filter considers the five earliest values to each have a significance of 0.15 and the five later values to each have a significance of 0.05. One can observe by the graph that this filter is a considerable improvement to the unmodified version. Although, the programmers

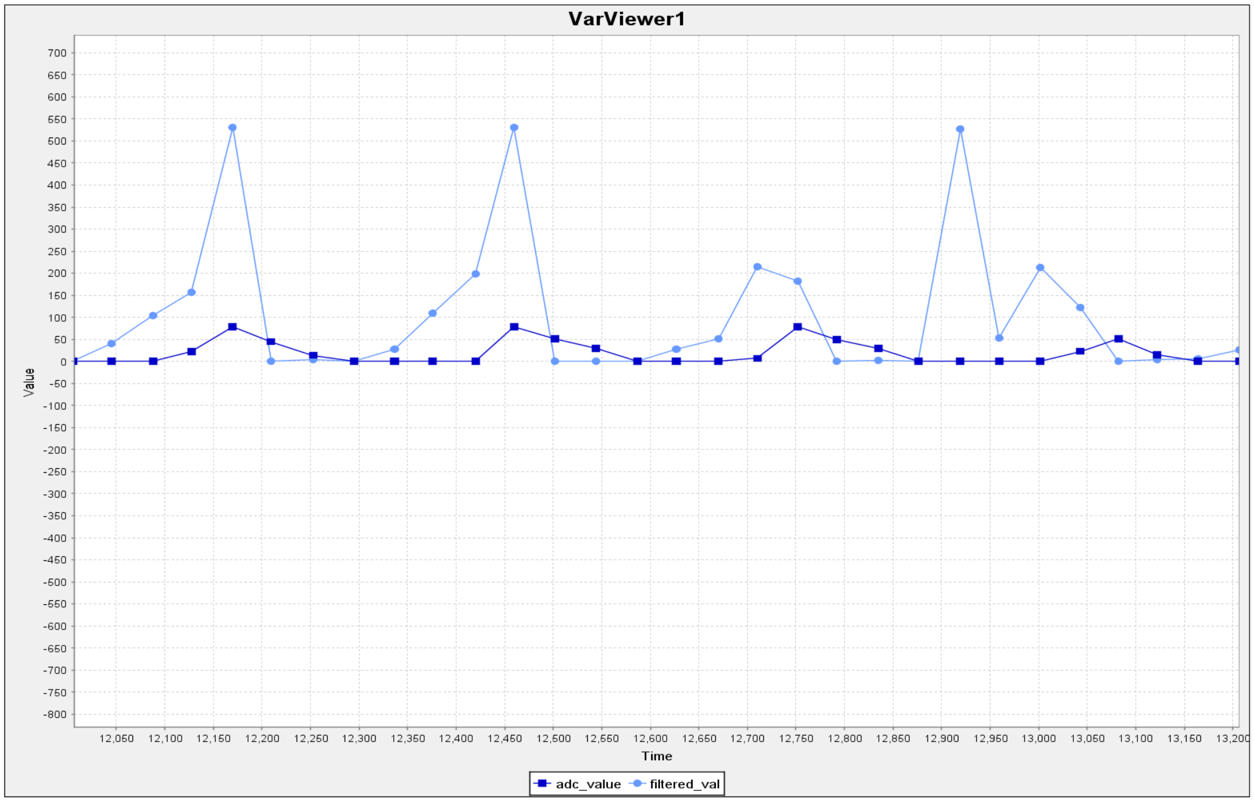


Figure 2: Filtered and Unfiltered Values VS Time (ms), Using Unmodified FIR Filter

believe that it could be optimized further.
For the graph above, the configuration for the filter's coefficients is [0.15, 0.15, 0.15, 0.15, 0.15, 0.05, 0.05, 0.05, 0.05, 0.05]. This filter considers the five later values to each have a significance of 0.15 and the five early values to each have a significance of 0.05. One can observe by the graph that this filter is an improvement to the previous version. The programmers believe that this version of the filter should perform well enough given the scope of the problem.

6 Conclusion

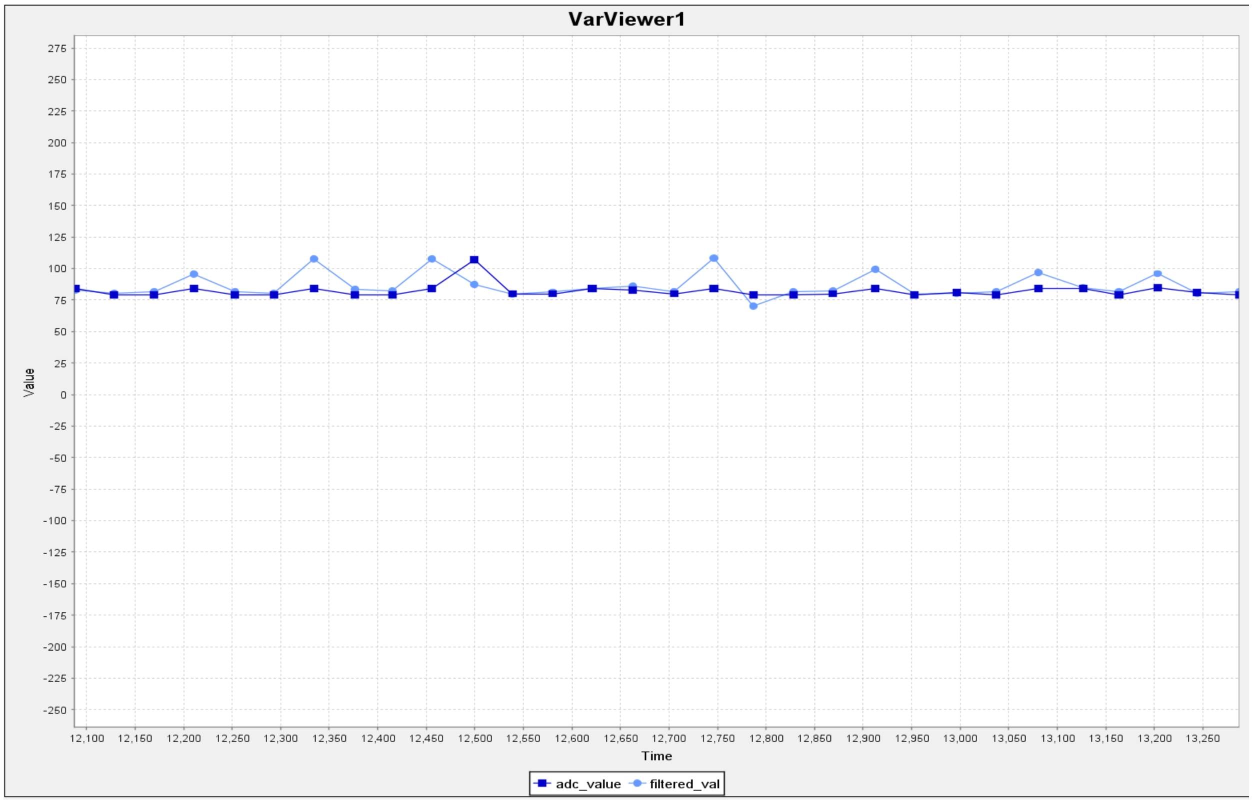


Figure 3: Filtered and Unfiltered Values VS Time (ms), Using High Coeffcients for Early Values

Appendix A

GPIO Configuration Parameters

This appendix lists the configuration parameters set for each of the different GPIO pins (or classes of GPIO pins).

User Input Button

Parameter	Value
Mode	`GPIO_MODE_IT_RISING`
Pull	`GPIO_NOPULL`

Display Mode LEDs (4 of these)

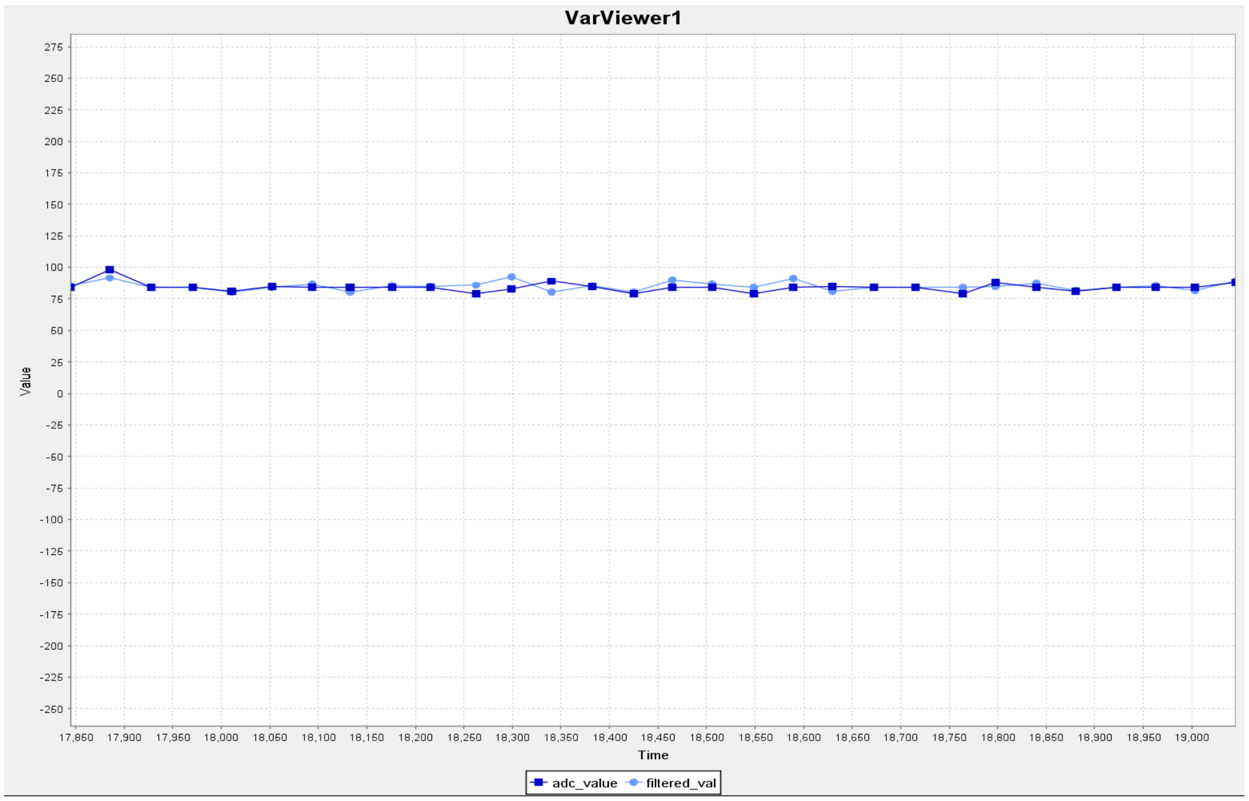


Figure 4: Filtered and Unfiltered Values VS Time (ms), Using High Coeffcients for Late Values

Parameter	Value
Mode	GPIO_MODE_OUTPUT_PP
Pull	GPIO_NOPULL
Speed	GPIO_SPEED_FREQ_LOW

Display Segment Pins (8 of these)

Parameter	Value
Mode	GPIO_MODE_OUTPUT_PP
Pull	GPIO_NOPULL
Speed	GPIO_SPEED_FREQ_LOW

Display Selector Pins (3 of these)

Parameter	Value
Mode	GPIO_MODE_OUTPUT_PP
Pull	GPIO_NOPULL
Speed	GPIO_SPEED_FREQ_LOW

Appendix B

ADC Configuration Settings

ADC Instance Parameters

Parameter	Value
Clock Prescaler	`ADC_CLOCK_SYNC_PCLK_DIV4`
Resolution	`ADC_RESOLUTION_8B`
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
External Trigger Conversion Edge	`ADC_EXTERNALTRIGCONVEDGE_RISING`
External Trigger Conversion	`ADC_EXTERNALTRIGCONV_T2_TRGO`
Data Alignment	`ADC_DATAALIGN_RIGHT`
Number of Conversions	1
DMA Continuous Requests	Disabled
EOC Selection	`ADC_EOC_SINGLE_CONV`

ADC Channel Parameters (Channel 1)

Parameter	Value
Rank	1
Sampling Time	`ADC_SAMPLETIME_28CYCLES`

Appendix C

TIM2 Configuration Settings

TIM2 Instance Parameters

Parameter	Value
Instance	`TIM2`
Clock Prescaler	`83999`
Counter Mode	`TIM_COUNTERMODE_UP`
Period	1
Clock Division	`TIM_CLOCKDIVISION_DIV1`

TIM2 Clock Source Parameters

Parameter	Value
Clock Source	`TIM_CLOCKSOURCE_INTERNAL`

TIM2 Master Configuration Parameters

Parameter	Value
Master Output Trigger	`TIM_TRGO_UPDATE`
Master Slave Mode	`TIM_MASTERSLAVEMODE_DISABLE`

Appendix D

TIM3 Configuration Settings

TIM3 Instance Parameters

Parameter	Value
Instance	`TIM3`
Clock Prescaler	`0`
Counter Mode	`TIM_COUNTERMODE_UP`
Period	`PWM_PERIOD`
Clock Division	`TIM_CLOCKDIVISION_DIV1`

TIM3 Master Configuration Parameters

Parameter	Value
Master Output Trigger	`TIM_TRGO_RESET`
Master Slave Mode	`TIM_MASTERSLAVEMODE_DISABLE`

TIM3 Output Channel Parameters

Parameter	Value
OC Mode	`TIM_OCMODE_PWM1`
Pulse	`duty_cycle * PWM_PERIOD`
OC Polarity	`TIM_OCPOLARITY_HIGH`
OC Fast Mode	`TIM_OCFAST_DISABLE`

Appendix E

HAL Cube MX Autogenerated Code

Appendix F

HAL GPIO Pin Configuration

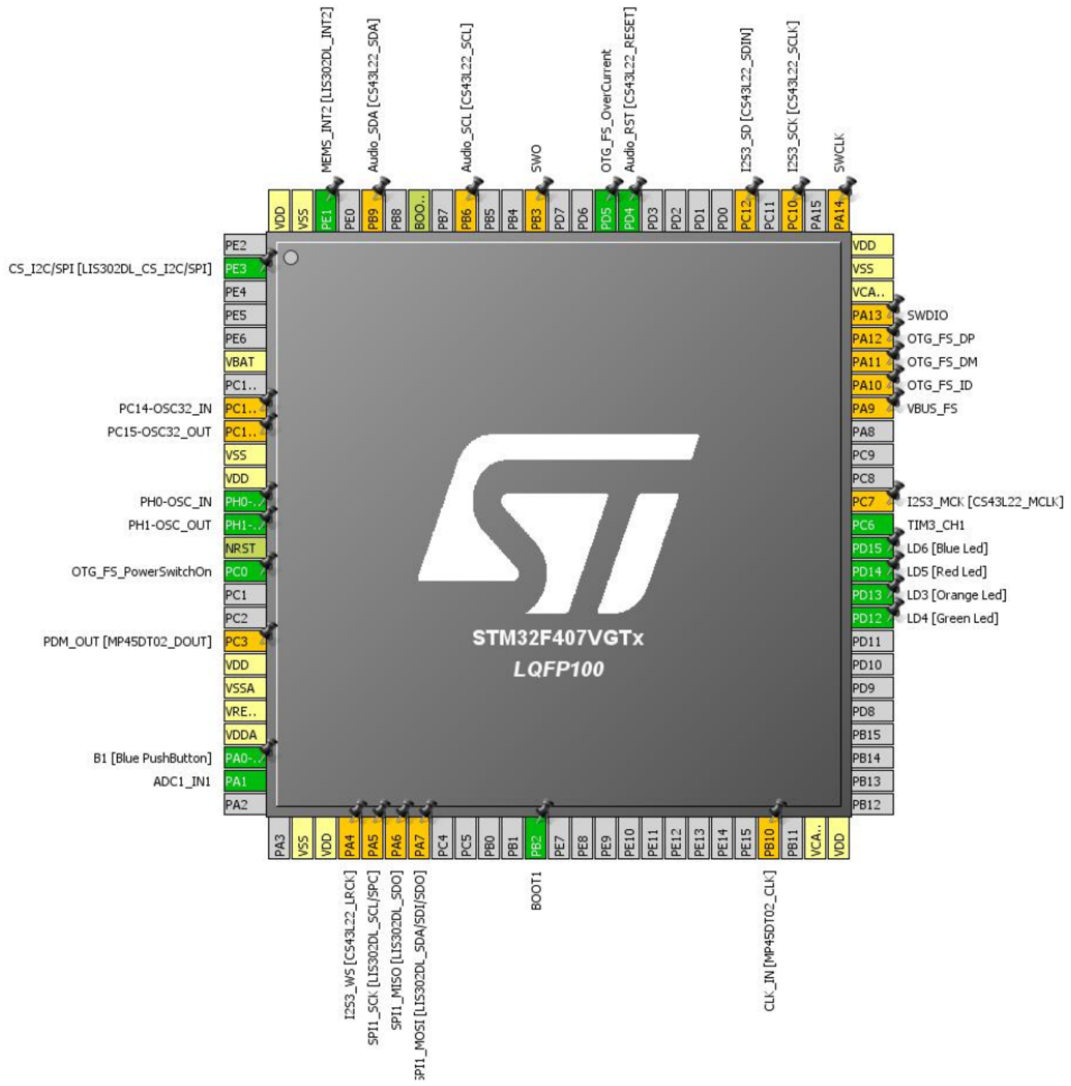


Figure F.1: HAL's Visualization of the STM32F4's Pins Configuration

Appendix G

Theory References

- 1. Tool, B. and Library, C. (2018). Rectifier Circuits — Diodes and Rectifiers — Electronics Textbook. [online] Allaboutcircuits.com. Available at: https://www.allaboutcircuits.com/textbook/sem
3/rectifier-circuits/ [Accessed 18 Mar. 2018].

- 2. Justsoftwaresolutions.co.uk. (2018). Locks, Mutexes, and Semaphores: Types of Synchronization Objects — Just Software Solutions - Custom Software Development. [online] Available at: https://www.justsoftwaresolutions.co.uk/threading/locks-mutexes-semaphores.html [Accessed 18 Mar. 2018].