VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Self Tuning Buck Converter

Niels Clayton

Supervisors: Daniel Burmester and Ramesh Rayudu

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

Switch-mode power supplies are commonly used in a wide variety of consumer and professional appliances to transform DC voltages with high efficiency. One such switch-mode supply is the buck converter, which steps down a DC voltage. The current buck converter design process requires that a specific output filter be designed around the switching frequency of the converter, the required output voltage, and the desired inductor ripple. This filter design process often results in the selection of non-standards components that are difficult to purchase or manufacture, increasing costs and leading to design compromises. This project will implement a control system to actively control the inductor current ripple by modulating the switching frequency of the converter. This will allow engineers to design buck converters directly for the inductor current ripple they wish to tolerate, eliminating the issues of designing this output filter.

Contents

Chapter 1: Introduction

Historically power distribution has been primarily in the form of AC (Alternating current). This is credited to the fact that AC power is more efficient to transmit over long distances and made it easy to step up and down the voltages efficiently with transformers [1]. However, with the invention of solid-state electronics such as the metal–oxide–semiconductor field-effect transistor or MOSFET for short, it has become possible to efficiently step up and step down direct current (DC) voltages. This has been achieved by the invention of the switch-mode power supply, which has facilitated the continued reduction of size and increase in efficiency of electronics [2].
Today, switch-mode power supplies can be found in a wide variety of consumer and professional electronics, with some examples being laptops, phones, and any form of DC charger. Their widespread usage when compared to other DC-DC converters such as linear regulators can be attributed to their far greater efficiency. One such switch-mode power supply is the buck converter, which will step down a DC input voltage to a lower DC output voltage.

1.1 Project Motivation

Although buck converters are a widespread technology, they are not without their limitations and drawbacks. The current buck converter design process requires that a specific output filter be designed around the switching frequency of the converter, and the desired inductor ripple. This filter design process will often result in the converter requiring discrete passive components that are non-standard and hard to source. This will usually result in the designer having to make compromises in their design for either the cost or the performance of the converter.
Another drawback of this design process is the static nature of both the filter and the switching components once they have been selected. This results in the buck converters desired inductor current ripple only being achieved at a very specific designed output voltage or load. This means that with current buck converter designs varying the desired output voltage or varying the output load will cause the inductor current ripple to vary. This is an issue, as very few loads are static and will not change during their operation.

1.2 Project Goals

This project aims to eliminate the need to design the output stage of a buck converter. By implementing a control system that varies the switching frequency of the converter, we will be able to directly manipulate the inductor current ripple. This project aims to produce a proof of concept buck converter that is capable of operating at 12V, with an output range of 3-10V and precision of $\pm$5%. The converter will also be able vary its switching frequency between $1kHz$ and $100kHz$, allowing for selection of inductor current ripple between 20% and 50% with precision of $\pm$5%. All of this must be implemented while maintaining the standard functionality of the converter. For full system requirements please refer to **??**.

TODO Expand on the project goals, possibly convert the previous section into a bulleted list of project goals and requirements.

This will be referenced in the implementation and design sections many times, so it needs to clearly outline what we are looking to achieve in this project.

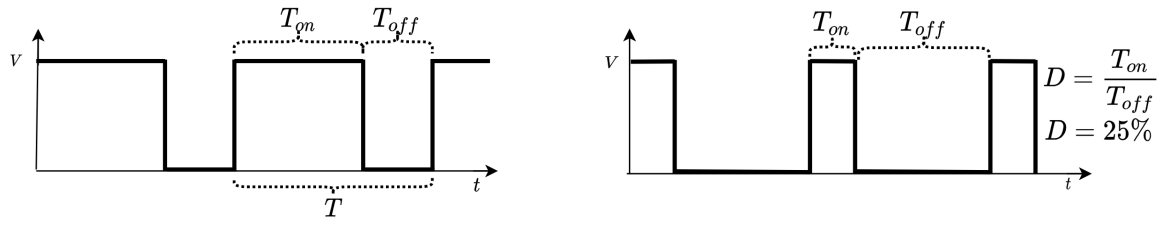
Chapter 2: Background

A literature research was performed to inform design decision made in this project, and to evaluate any existing research. It will discuss buck converter design factors and topologies, as well as the various different methods of PWM generation. In performing this literature research, we searched Google Scholar, Engineering Village, and Te Waharoa to find designs that utilised variable frequency PWM.
These searches returned no research relevant to the designs of this project, with the only related work focusing on the electromagnetic noise reduction using randomised frequency modulation [3, 4]. Because of this, research was instead performed to inform the design of the buck converter and the generation of PWM signals.

2.1 Pulse Width Modulated Signal Generation

Pulse width modulation (PWM) is a digital signal generation technique shown in Figure 2.1a, in which the Period T of the signal is held constant, while the ratio of its logic high period T_{on} to logic low period T_{off} is modulated. This ratio of high period to the low period is referred to as the duty cycle of the PWM signal and is often expressed as a percentage, this can be seen in Figure 2.1b.

PWM signals are used in a wide variety of applications for both digital and analogue electronics. PWM is often used to generate analogue signals from digital components by varying the average voltage of the digital PWM signal over time [5]. PWM is also used to control the switching elements contained within switch mode power supplies using this same principle, as discussed in Section 2.2. With regard to this project, we will be looking to generate a PWM signal that can be modulated in both duty cycle and frequency.



(a) Pulse width modulated signal

(b) Duty cycle calculation

Figure 2.1: Pulse width modulated signal characteristics

2.1.1 Analogue PWM Signal Generation

Designing a PWM signal generator using analogue components has three distinct stages required to generate the signal. These stages can be seen in Figure 2.2, and include clock

generation, triangle wave generation, and signal comparator stages [6].

The clock generation stage generates a square wave clock signal at a set frequency. This is usually done using a quartz crystal oscillator, or another form of resonating oscillator circuit. The triangle wave generating state must take the clock signal from the previous stage, and produce a triangle wave of the same frequency. This stage is most often done using a standard op-amp integrating circuit with unity gain at the resonating frequency of the clock source. The final signal comparator stage will convert this triangle wave into a PWM signal. Using a comparator, a reference voltage can be applied to the non-inverting input, and then the triangle wave can be applied to the inverting input. This will produce a pulse train with the same frequency as the clock source, where the period of T_{on} and T_{off} is set by the reference voltage.

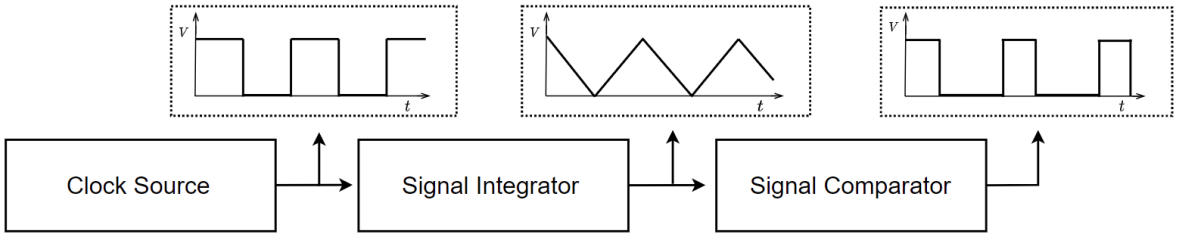


Figure 2.2: Stages of analogue PWM generation

2.1.2 Digital PWM Signal Generation

Designing a PWM signal generator with digital components is far simpler than the method described in Section 2.1.1, and can be done using either a microcontroller or a Field Programmable Gate Array (FPGA). By using an internal timer that is continually incrementing at a known period we can set a period for our PWM. Then by toggling a digital I/O when a compare variable is equal to the value of the timer. we are able to generate a PWM signal with a variable duty cycle [7]. This can be achieved on most microcontrollers, however the maximum frequency and duty cycle accuracy will be dependant on individual clock speed and internal register sizes.

2.2 Buck Converters

The buck converters is a variant of a switch mode power supply that steps down a DC input voltage to a DC output voltage. They are commonly used in a wide variety of consumer and professional appliances such as laptops, phones, and chargers due to their high efficiency compared to other DC-to-DC step down converters such as linear regulators [8].

The basic operational components of a buck converter can be seen below in Figure 2.3. From this we see that a buck converter has three main elements, the input voltage source, two switching components, and an output filter across the load. In the case of Figure 2.3, the first switching component is an actively controlled switch such as a MOSFET or transistor, and the second a passive switching diode. This configuration of an active and a passive switch is known as the non-synchronous buck converter topology, if the passive diode were to be replaced with a second active switch the topology would be considered synchronous. Although both topologies function under the same fundamental principles, the non-synchronous topology is easier to implement with the drawback of higher losses and

therefore lower efficiency.

It can also be seen from Figure 2.3 that a buck converter has two operating states that are controlled through the activation of these switching components. By toggling these switching components at high speed though the use of PWM, we can control the current flowing through the inductor of the output filter. By controlling this current we are also able to directly control the current through, and voltage across the output load of the converter. Using this, buck converters will often have a feedback control system in their design to be able to actively control and regulate the output voltage during usage. This controller will vary the duty cycle of the the switching PWM signal, thereby varying the output voltage of the buck converter as shown in Equation (2.1).

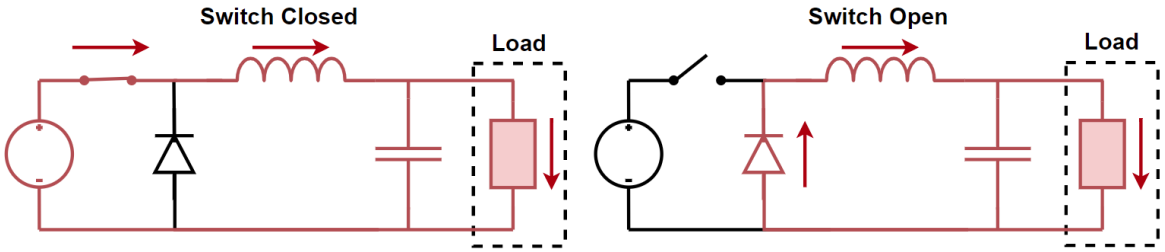


Figure 2.3: Operating states of a buck converter

2.2.1 Buck Converter Design

The design of a common buck converter has two primary considerations, the output voltage of the converter V_0, and the inductor current ripple of the converter Δi_L. These considerations can be specified by designing the buck converter using Equation (2.1) & Equation (2.2) [9, 10].

When designing a buck converter the first design specification that must be met is the output voltage. In Equation (2.1) the output voltage can be directly related to the input voltage V_{in} and the switching duty cycle D. Using this equation it is possible to directly set the output voltage of the buck converter by varying this duty cycle.

$$V_o = D \cdot V_{in} \tag{2.1}$$

Once the output voltage has been specified, the inductor current ripple can be calculated and specified with Equation (2.2). This equation allows for the inductor current ripple to be directly related to the inductor size L, and the PWM switching frequency f_s. This allows the the specification of the inductor current ripple through the varying of these two values.

$$\Delta i_L = \frac{V_o \cdot (1 - D)}{L \cdot f_s} \tag{2.2}$$

These two equations will be used to inform the designs and specifications of this project, and will be discussed in detail in Section 3.1.

2.3 Current Sensing

Discuss the varying methods of current sensing that are available, and what each of their advantages and disadvantages are.

2.3.1 Hall Effect Sensors

brief overview of hall effect sensors. They function by measuring the magnetic flux density, meaning that they are commonly used to senses the presence of a magnet. However since a current flowing though a wire will produce a magnetic flux, they can also be used to measure current flow.
Because of this operation, they are able to sense a current without contacting or altering the circuit. This means that they are commonly used in the measurement of high voltage or high current circuits, as they will remain completely isolated from the circuit being sensed, and will also not dissipate power from the sensed circuit. This provides large safety advantages, and can often decrease the complexity of the sensing circuit.
Hall affect sensors do however suffer from a lack of precision. Due to their operation, their measurements will constantly be offset by any ambient magnetic flux. This means that they are not commonly used for the sensing of small signal currents, as the noise floor of the Earth's own magnetic field can often be larger than the signal being sensed.

2.3.2 Current Sense Amplification

Current sense amplification works on the basic principle of Ohm's law $I = \frac{V}{R}$. By sensing the voltage dropped across a known value resistive element (Often called the current shunt), it is possible to calculate the current that is flowing through the element. This form of current sensing is very simple to implement in theory, however it has the effect of altering the system being sensed by adding a resistive load, and thereby increasing the losses of the system.
To mitigate the effects of this sensing on a circuit, a smaller resistive load can be used. However, this will also decrease the measurable voltage across the load, and therefore decrease the precision of a taken measurement. Because of this, shunt based current sensors are often paired with an operational amplifier known gain. This combination allows for accurate amplification of the shunt voltage drop, increasing the precision of the measurement, and allowing for drastically smaller shunt resistors.

2.4 Control Systems

Discuss the basics of control theory, and how a controller can be implemented on a digital system within discrete time.

- Discuss in very general terms what a control system is what what it seeks to do in a system.
- Discuss what the control system will be doing in the case of this project. Talk about how a controller will be used to control both the output voltage of the converter, and the inductor ripple of the converter.
- Discuss the design, use, and implementation of a PID controller within a system.

2.4.1 PID Controllers

Chapter 3: Design

The deign processes outlined within this report will draw heavily on the background research discussed in Chapter 2. These designs aim to effetely implement the outlined system requirements in a robust and repeatable manner.

3.1 Defining & Justifying System Specifications

Based on the system requirements that have been outlined in Chapter 1, a set of system specifications can be created to inform our design decisions.

It has been specified that the final system design will be able to select for both the output load voltage, and the inductor current ripple of the buck converter. From this requirement we can identity that two separate control systems should be designed, one to regulate the output voltage of the converter, and one to regulate the inductor current ripple.
Next we can identify that to control the output load voltage and the inductor current ripple, we must be able to actively effect their current state. Based on Equation (2.1) we can see that by varying the duty cycle of the converters PWM signal, we are able to directly control the output voltage. Similarly, from Equation (2.2) we can see that by varying the converters PWM switching frequency we are able to directly control the inductor current ripple. From this we can specify that our designs of the PWM generation must be capable of varying both the duty cycle and the switching frequency of the output PWM signal independently and simultaneously.

The outlined requirements also specify the level of precision that will be required from the output load voltage and the inductor current ripple. This allows us to specify the tolerable error for the system design, and calculate the minimum required system specifications. Based on the buck converter design equations from Section 2.2.1, The following specifications have been identified:

- Maximum allowable duty cycle step size

$$V_{error} = V_{min} \cdot error = 0.15V \tag{3.1}$$

$$D_{step} = \frac{V_{error}}{V_{in}} = 0.0125 \tag{3.2}$$

$$N_{step} = \frac{1}{D_{step}} = 80 \tag{3.3}$$

- Maximum & minimum inductor sizes

$$L_{max} = \frac{V_{max} \cdot (1 - D_{max})}{f_{min} \cdot I_{min}} = 27.7mH \tag{3.4}$$

$$L_{min} = \frac{\frac{V_{in}}{2} \cdot (1 - 0.5)}{f_{max} \cdot I_{min}} = 0.5mH \tag{3.5}$$

- Maximum allowable frequency step size

$$f_{step} = \frac{V_{max} \cdot (1 - D_{max})}{(I_{min} - I_{Error}) \cdot L_{max}} - f_{min} = 52Hz \tag{3.6}$$

$$N_{steps} = \frac{(f_{max} - f_{min})}{f_{step}} = 1881 \tag{3.7}$$

The derivation of these values can be found in Appendix B.

TODO Refactor this to be a list of specifications

From these equations we can build a list of final specifications to inform the design of our PWM generator and buck converter. The PWM generator must provide a minimum voltage step size of 0.0125V, for a resolution of 80 voltage steps between 3V and 10V. The PWM generator must also be able to provide a minimum frequency step size of 52Hz, for a resolution of 1881 frequency steps between 1kHz & 100kHz. Finally we can also specify that the buck converter must be capable of functioning with inductor values between 0.5mH & 27.7mH.

By designing the PWM generator and the buck converter to these specifications, we are able to guarantee that we can always achieve the requirements outlined in Chapter 1.

TODO Discuss the minimum and maximum sense current, along with the bandwidth requirement of the sensors

Design the system specifications around the current sensing and voltage sensing. What is the required bandwidth of the sensor? What is the required precision?

3.2 System Architecture & Design

To achieve the specifications that have been outlined in Section 3.1, it is important to design the system architecture around them. In Figure 3.1 an overview of the system architecture can be seen, with three main design sections outlined. These sections each represent a significant segment of work that must be completed for the final artefact of this project to be achieved.

The first section of work that must be completed is the design of the PWM generation, denoted 1 in Figure 3.1. This PWM generator will be used to control both the output voltage and the inductor current ripple, and as such must be able to modulate both the duty cycle and the frequency of the PWM to the precisions required.

The second section of work is the design of the sensing elements required by the system, denoted 2 in Figure 3.1. These elements will be used to measure both the output voltage and the inductor current ripple, and therefore must be able to achieve the required precisions and sampling rates.

Finally the third section of work is the design and implementation of the two control systems, denoted 3 in Figure 3.1. These control systems will be responsible for maintaining the desired output voltage and inductor current ripple of the buck converter. This system will therefore be responsible for facilitating the final functionality of the project, combining sections 1 & 2.

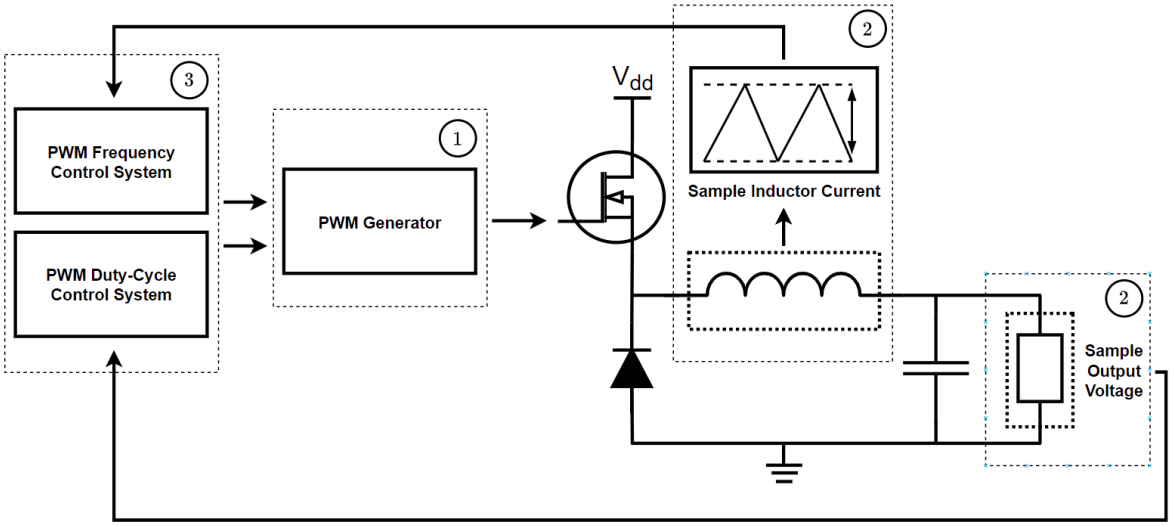


Figure 3.1: High level system overview

3.3 PWM Generation

In the design of the PWM generator, both the analogue and digital designs discussed in Section 2.1 were considered, designed, and tested for. Each of these designs presented pros and cons that would affects the overall design of the system architecture. This section will discuss these designs and finalise the design of the PWM generator for this system.

3.3.1 Analogue PWM Generator Design

As discussed in Section 2.1.1, the design of the analogue PWM signal generator requires three stages, each of which will have its own design requirements based on the specifications outlined in Section 3.1.

The clock generation stage will be responsible for setting the frequency of the final PWM signal. This specifies that the clock source have a variable frequency output range between $1kHz$ and $100kHz$, with a minimum step size of $52Hz$. Research was done on a variety of clock sources, looking at voltage-controlled oscillators (VCO's), signal generator IC's, and even the basic 555 timer. From this VCO's were identified to operate at much higher frequencies than those used in this project. It was also identified that signal generator IC's

often require selections of passive components to operate effectively, increasing their complexity. For this reason the variable frequency 555 timer circuit was selected, as it provided the required specifications.

The next section designed was the signal integrator stage. This stage consisted of a basic op-amp integrator circuit, with a design requirement that it be able to integrate the clock signal across the frequency range required. This circuit was designed and implemented in an LTSpice simulation to evaluate it's performance, and can be seen in Appendix C.1. From this simulation it was noted that the integrator's frequency response was similar to that of a first order low pass filter, and greatly attenuated the integrated signal. For this reason it was decided that analogue PWM generation would not be implemented in this system, as it presented many issues.

TODO insert images of the designed integrator circuit, and it's frequency response (bode plot)

3.3.2 Digital PWM Generator Design

As discussed in Section 2.1.2, The design of the digital PWM generator is far simpler than that of the analogue, and can be implemented in a wide variety of methods. In this project microcontrollers and FPGA's have been considered.
Based solely on the capabilities of the platform, the PWM generator would be best designed and implemented on an FPGA as it would allow for superior speed and precision. However FGPA design brings a lot of difficulties, primarily in the prototyping and testing stages. Because of this, due to the limited time from of this project, we have decided to implement this PWM design using a microcontroller.
The selection of the microcontroller is highly dependant on the clock frequency and design of the PWM peripherals, as it must be capable of achieving the specifications outlined in Section 3.1. A large selection of microcontroller data-sheets were reviewed to identify their specifications, including AVR, STM8, Espressif, and teensy based microcontrollers. From this review it was decided that the ESP32 microcontroller would be best suited to this project [11]. This microcontroller is capable of outputting a maximum PWM frequency $125kHz$ with a duty cycle resolution of 9 bits (512 voltage steps). From here a short C program was written to test the PWM functionality of the ESP32, and it was confirmed that it met the required specifications. The source code and images of this PWM signal can be found in Appendix C.2.

3.4 Inductor Current Peak to Peak Sensing

Discuss the purpose of this sensors, and the requirements of the sensing system in regards to its range, precision, and bandwidth. Also discuss the required outputs of the sensor (we need to detects the current peak value, and the mean current).

3.4.1 Current Sensor Selection

Talk about how hall effect sensors were identified to not be suitable for this design due to the reasons outlined in Section 2.3.1. From here the discuss the selection of a current sense amplifier (it's gain, bandwidth, and precision), and then the selection and design of the shunt resistor. Then discuss the expected voltage output of this current and the expected waveform.

Based on the specifications outlined in Section 3.1, it can be identified that directly sampling the output waveform from this sensors is not feasible due to it's large possible bandwidth. From there then discuss the design of varying analog circuits to attempt to identify the peak current and mean current.
I want to show circuit designs, and simulations for both the precision rectifier peak detection design, and the capture and hold peak detection design.

3.4.2 Precision Rectifier Peak Voltage Detection

Design, and simulations of the Precision Rectifier Peak Voltage Detection.

3.4.3 Sample and Hold Peak Voltage Detection

Design, and simulations of the Sample and Hold Peak Voltage Detection.

3.4.4 Current sensing digitisation

Discuss the selection of the ADC for measuring the peak ripple and mean ripple currents. Talk about the minimum required ADC precision, and bandwidth requirements.

3.5 Control System

The control system designs will inform the selection of the sensors used within the system design. This section will cover the selection of the controller topology (PI & PID), some basic modelling of the system, and then the selection of the sensors required to generate the correct feedback signals.

3.5.1 Output Load Voltage Controller

3.5.2 Inductor Current Ripple Controller

Chapter 4: Implementation

4.1 PWM Generation

go ask Daniel about this because currently all I can think to say is about the bootstrap circuit and the code. But the bootstrap circuit is not important to the design, and was inly used due to covid since I will be using a gate driver in the final design.

TODO Gate driving and Bootstrapping

Discuss how the micro will be unable to directly drive the gate of the switching MOSFET, as it's $V_{GS_{on}}$ will be too large. Because of this a gate driver will need to be selected. The driver will need to be driven using the 3.3V logic from the ESP32, and will need timings fast enough to function correctly at 100kHz. I can also discuss how during the lock-down I was unable to purchase a gate driver, and so I built a bootstrapping circuit from components I located around the house.

4.2 Inductor Current Peak to Peak Sensor

Do I just show a photo of the breadboard and say that I built it on that?

4.3 Control System

4.4 Software Architecture

Discuss the ESPidf HAL (Hardware Abstraction Layer), and how it provides greater control of the system resources than the more commonly used arduino platform.
Then discuss how the system operates within the freeRTOS real time operating system, what allows for easy multitasking between the different time sensitive control loops that will have to be run on the micro.
Finally discuss the simple to use API that was implemented, which aims to abstract away the HAL layer. This API is implemented using collection of single header libraries written in the 'C' programming language, with each implementing only the core functionality of the hardware they are interacting with. This makes the software platform robust and simple to move to different embedded platforms, with such a move only requiring the re-implementation of the core functions of each library.

Chapter 5: Evaluation

The evaluation of the system will be conducted using a range of load resistances, evaluating its performance with 10Ω, 15Ω, and 20Ω output loads, using a constant supply voltage of 12V DC. S

- The buck converter will be able to take input voltages up to 12V DC
- The buck converter must maintain the basic functionality outlined in eq. (2.1)
- The buck converter will have an output voltage range between 3V and 10V DC
- The output voltage accuracy will be within $\pm$5% of the target output voltage
- The user will be able to define the inductor ripple between 20% and 50%
- The inductor ripple accuracy will be within $\pm$5% of the defined inductor ripple
- The buck converter will have a switching frequency range of 1kHz to 100kHz
- The control system will have no steady state error

5.1 PWM Generation

5.2 Inductor Current Ripple Sensor

5.3 Control System

Chapter 6: Conclusions

Bibliography

[1] E. Earley, "What's the difference between ac and dc?." Online, Sept. 2013.

[2] G. Bocock, "History of switch mode power supplies (smps)." Online.

[3] J. R. Roman, "PWM regulator with varying operating frequency for reduced EMI," Mar. 2001.

[4] Y. L. Familiant and A. Ruderman, "A Variable Switching Frequency PWM Technique for Induction Motor Drive to Spread Acoustic Noise Spectrum With Reduced Current Ripple," *IEEE transactions on industry applications*, vol. 52, no. 6, pp. 5355–5355, 2016.

[5] W. Tareen, M. Aamir, S. Mekhilef, M. Nakaoka, M. Seyedmahmoudian, B. Horan, M. A. Memon, and N. A. Baig, "Mitigation of power quality issues due to highpenetration of renewable energy sources in electricgrid systems using three-phase apf/statcomtechnologies: A review," in *Power Electronics in Renewable Energy Systems* (T. Suntio and T. Messo, eds.), pp. 105–133, 2019.

[6] J. Caldwell, *Analog Pulse Width Modulation*. texas instruments, June 2013.

[7] S. Colley, "Pulse-width modulation (pwm) timers in microcontrollers." Online, Feb. 2020.

[8] N. Mohan, *Power Electronics: A First Course*. Don Fowley, Oct. 2011.

[9] N. Mohan, *Power Electronics a First Course*, ch. 4: Switch Mode DC-DC Converters: Switching Analysis, Topology Selection and Design, pp. 38–68. Don Fowley, 2012.

[10] B. Hauke, "Basic Calculation of a Buck Converter's Power Stage," tech. rep., Texas Instruments, Aug. 2015.

[11] Espressif Systems, *ESP32: Technical Reference Manual*, 4.4 ed., 2021.

Acknowledgments

Thank you Danny B for being great :-)
Thank you ECS Techs for putting up with my questions
And thank you to my wonderful girlfriend for not kicking me out when I was working on this stupid fucking project really late

Appendix A: Peak Detector Figures & Tables

A.1 Peak Detection Error Tables

150mV Peak to Peak Input Signal Initial and Final Design Peak Detector Output Error Across Frequencies				
Frequency (kHz)	**Final Design Error (mV)**	**Final Design % Error**	**Initial Design Error (mV)**	**Initial Design % Error**
1	-3.90	-5.2	3.40	4.533333333
5	-3.80	-5.066666667	5.80	7.733333333
10	-3.70	-4.933333333	12.80	17.06666667
15	-2.90	-3.866666667	18.50	24.66666667
20	-2.20	-2.933333333	23.30	31.06666667
25	-1.60	-2.133333333	28.30	37.73333333
30	-0.90	-1.2	32.60	43.46666667
35	-0.30	-0.4	36.50	48.66666667
40	0.20	0.266666667	40.20	53.6
45	0.80	1.066666667	43.70	58.26666667
50	1.30	1.733333333	47.00	62.66666667
55	2.00	2.666666667	50.10	66.8
60	2.60	3.466666667	53.00	70.66666667
65	3.10	4.133333333	56.10	74.8
70	3.50	4.666666667	59.50	79.33333333
75	4.00	5.333333333	62.70	83.6
80	4.60	6.133333333	65.20	86.93333333
85	5.00	6.666666667	67.80	90.4
90	5.60	7.466666667	68.10	90.8
95	6.10	8.133333333	68.10	90.8
100	6.70	8.933333333	68.10	90.8

Table A.1: Table of output errors at varying frequencies for both the initial and final peak detection design with a 150mV peak to peak input signal.

500mV Peak to Peak Input Signal Initial and Final Design Peak Detector Output Error Across Frequencies				
Frequency (kHz)	**Final Design Error (mV)**	**Final Design % Error**	**Initial Design Error (mV)**	**Initial Design % Error**
1	-1.60	-0.64	7.00	2.8
5	-1.70	-0.68	15.00	6
10	-1.90	-0.76	23.90	9.56
15	-0.50	-0.2	32.90	13.16
20	1.40	0.56	40.70	16.28
25	2.70	1.08	47.90	19.16
30	3.30	1.32	54.40	21.76
35	4.40	1.76	60.60	24.24
40	5.50	2.2	66.30	26.52
45	6.30	2.52	71.90	28.76
50	7.40	2.96	77.30	30.92
55	8.30	3.32	82.40	32.96
60	9.40	3.76	87.40	34.96
65	10.20	4.08	92.50	37
70	11.00	4.4	97.00	38.8
75	11.80	4.72	101.70	40.68
80	12.70	5.08	106.20	42.48
85	13.40	5.36	110.50	44.2
90	14.40	5.76	114.90	45.96
95	15.00	6	119.00	47.6
100	15.80	6.32	123.20	49.28

Table A.2: Table of output errors at varying frequencies for both the initial and final peak detection design with a 500mV input signal.

1500mV Peak to Peak Input Signal Initial and Final Design Peak Detector Output Error Across Frequencies				
Frequency (kHz)	**Final Design Error (mV)**	**Final Design % Error**	**Initial Design Error (mV)**	**Initial Design % Error**
1	5.40	0.72	15.00	2
5	5.70	0.76	29.70	3.96
10	3.60	0.48	41.70	5.56
15	6.50	0.866666667	55.90	7.453333333
20	9.40	1.253333333	68.70	9.16
25	11.60	1.546666667	79.90	10.65333333
30	13.50	1.8	90.00	12
35	15.10	2.013333333	99.90	13.32
40	17.20	2.293333333	108.70	14.49333333
45	19.20	2.56	117.20	15.62666667
50	20.50	2.733333333	125.50	16.73333333
55	21.90	2.92	133.60	17.81333333
60	23.30	3.106666667	141.50	18.86666667
65	24.80	3.306666667	149.20	19.89333333
70	26.10	3.48	158.20	21.09333333
75	27.50	3.666666667	163.80	21.84
80	28.60	3.813333333	170.70	22.76
85	29.60	3.946666667	177.50	23.66666667
90	30.70	4.093333333	184.40	24.58666667
95	32.10	4.28	191.00	25.46666667
100	33.40	4.453333333	197.50	26.33333333

Table A.3: Table of output errors at varying frequencies for both the initial and final peak detection design with a 1500mV input signal.

Appendix B: System Specification Derivation

The system specification derivation equations have been input into the graphing platform Desmos.com. This has allowed me to visually inspect these equations and form conclusions. The full working and step by step derivation is also available on Desmos.com using the following links.

Duty cycle equation derivation:
https://www.desmos.com/calculator/8c7wmbyzw4

Inductor sizing equation derivation:
https://www.desmos.com/calculator/v7ntrescw5

PWM frequency equation derivation:
https://www.desmos.com/calculator/ekjhcrt9zg

Appendix C: PWM Generation Figures

C.1 Analogue PWM Generation

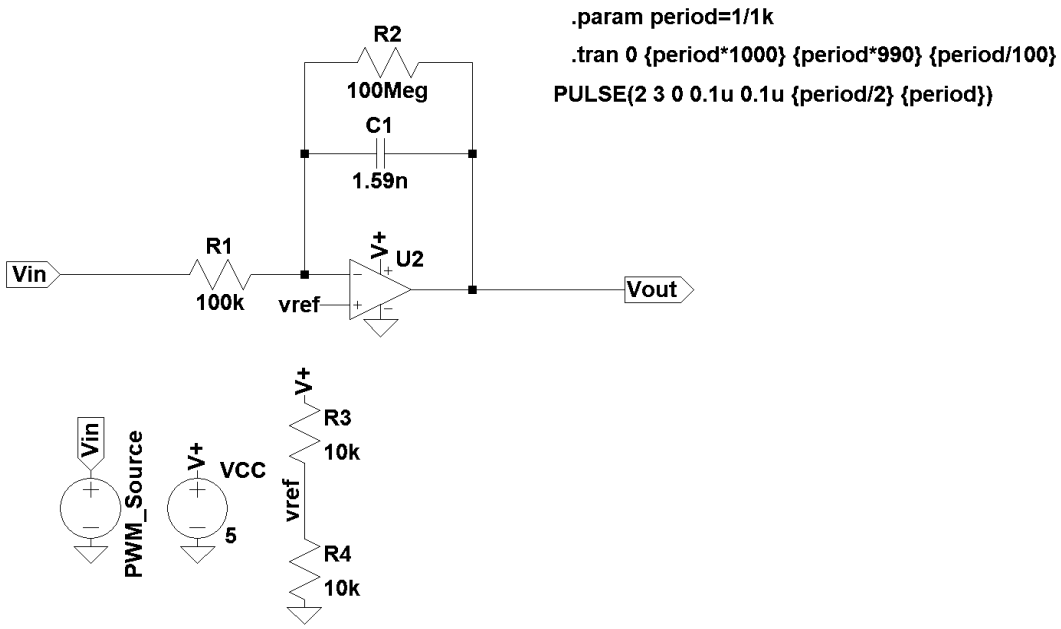


Figure C.1: Analogue PWM LTSpice circuit

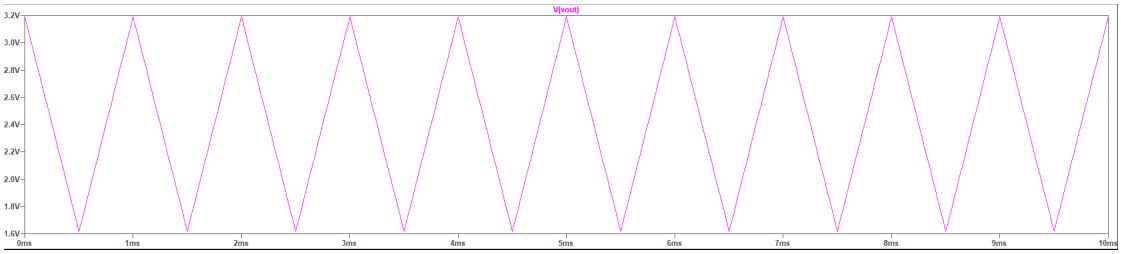


Figure C.2: Analogue PWM LTSpice simulation $1kHz$

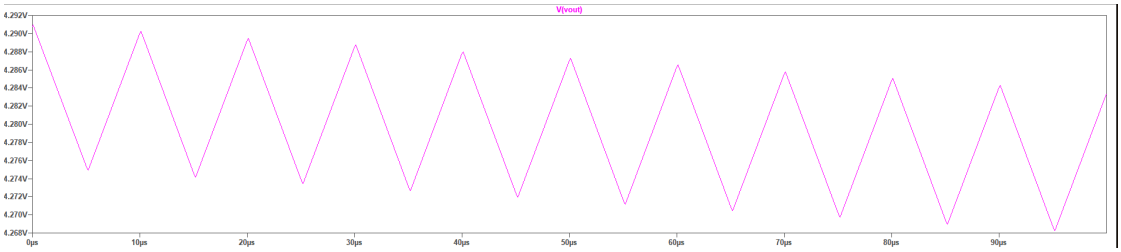
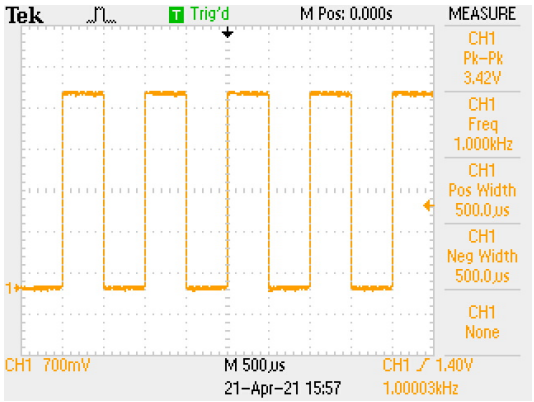
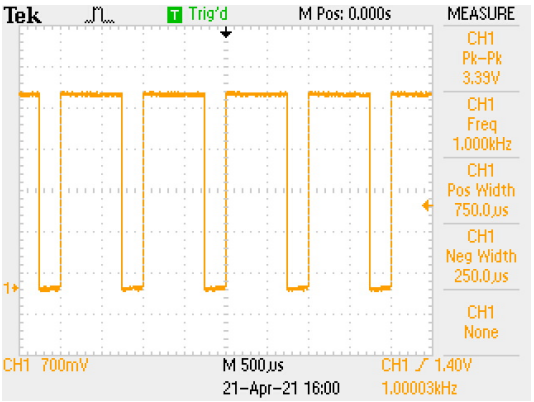


Figure C.3: Analogue PWM LTSpice simulation $100kHz$

C.2 Digital PWM Generation

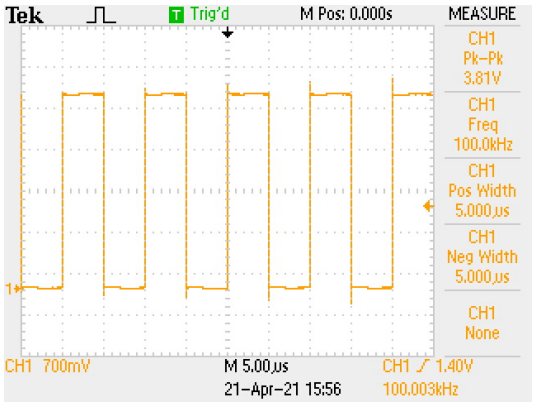


(a) Digital PWM Generation at $1kHz$ and a 50% duty cycle

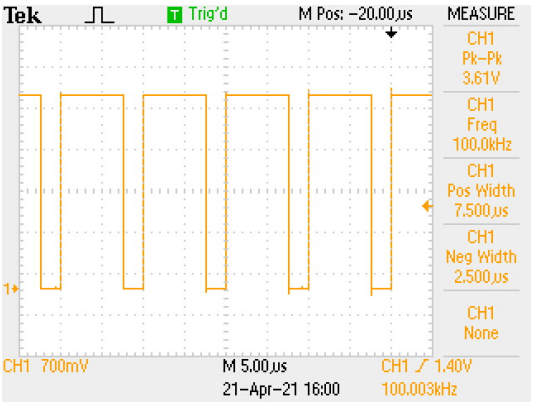


(b) Digital PWM Generation at $1kHz$ and a 75% duty cycle

Figure C.4: Digital PWM Generation at $1kHz$



(a) Digital PWM Generation at $100kHz$ and a 50% duty cycle



(b) Digital PWM Generation at $100kHz$ and a 75% duty cycle

Figure C.5: Digital PWM Generation at $100kHz$

Appendix D: Source Code

D.1 Single Header Libraries: Hardware Drivers

D.1.1 adc.h

```
#ifndef BUCK_CONVERTER_ADC
#define BUCK_CONVERTER_ADC

#include <math.h>
#include "driver/adc.h" // Include the ADC driver

// ADC struct to hold setup data
typedef struct esp_adc
{
    // Rolling average buffer
    uint16_t *buffer;
    uint8_t span;

    // ADC channel to sample
    uint8_t adc_channel;

} esp_adc;

#endif // BUCK_CONVERTER_ADC


#ifdef BUCK_CONVERTER_ADC_IMPL

/*
 * Set up the adc to have a 12bit width, and 11dB attenuation
 */
esp_err_t adc_init(esp_adc* adc){

    // ADC set up
    esp_err_t status; // Status variable to check if adc initialisation was
    successful

    status = adc1_config_width(ADC_WIDTH_BIT_12); // Set adc width to 12 bits
    if(status != ESP_OK) return status;

    status = adc1_config_channel_atten(adc->adc_channel, ADC_ATTEN_11db); // Set
    adc attenuation to 11dB
    return status;
}


```

```
40 /*
41  * Read the raw value from the adc
42  */
43 uint16_t adc_read(esp_adc* adc){
44     // Read the raw value from the provided adc channel
45     return adc1_get_raw(adc->adc_channel);
46 }
47 
48 /* Compute the rolling average of the raw ADC values.
49  * The circular buffer will store the index of the next element to be
50  * replaced in it's final index.
51  */
52 float rolling_average(esp_adc* adc, int raw_value){
53 
54     // Check if there is an averaging buffer
55     if(adc->span == 0)
56     {
57         printf("Can not compute average with a span of 0\n");
58         return -1;
59     }
60 
61     // get the index of the value to replace, and then replace it
62     int next_index = adc->buffer[adc->span];
63     adc->buffer[next_index] = raw_value;
64 
65     // Increment the index of the last value and check if it past the end of the
       array
66     next_index++;
67     adc->buffer[adc->span] = (next_index >= adc->span) ? 0 : next_index;
68 
69     // Calculate the rolling average
70     float total = 0;
71     for(int i = 0; i < adc->span; i++)
72     {
73         total = total + adc->buffer[i];
74     }
75 
76     return total/adc->span;
77 }
78 
79 /* Compute the conversion from the adc reading to a voltage.
80  *
81  * The output of the adc has been characterised and then a polynomial has been fit
       to the curve.
82  * This provides an error of < 1% for readings between 200mV and 3100mV
83  */
84  float IRAM_ATTR adc_conversion(float acd_reading)
85 {
86     return (pow(acd_reading , 4) * -7.6813211494455e-15) +
87            (pow(acd_reading , 3) *  5.03088719249885e-11) +
88            (pow(acd_reading , 2) * -1.06609443189713e-7) +
89            (0.00085850726668 * acd_reading) + 0.09077205072441;
90 }
91 
92 #endif
```

code/adc.h

D.1.2 pid.h

```
#ifndef BUCK_CONVERTER_PID
#define BUCK_CONVERTER_PID

/*
 * PID Controller code structure designed by https://github.com/pms67 under the MIT licence
 * https://github.com/pms67/PID
 */
typedef struct
{

    /* Controller gains */
    float Kp;
    float Ki;
    float Kd;

    /* Derivative low-pass filter time constant */
    float tau;

    /* Output limits */
    float limMin;
    float limMax;

    /* Integrator limits */
    float limMinInt;
    float limMaxInt;

    /* Sample time (in seconds) */
    float T;

    /* Controller "memory" */
    float integrator;
    float prevError; /* Required for integrator */
    float differentiator;
    float prevMeasurement; /* Required for differentiator */

    /* Controller output */
    float out;

} PIDController;

#endif


#ifdef BUCK_CONVERTER_PID_IMPL

void PID_controller_init(PIDController *pid){

    /* Clear controller variables */
    pid->integrator = 0.0f;
    pid->prevError = 0.0f;

    pid->differentiator = 0.0f;
    pid->prevMeasurement = 0.0f;
```

```
54
55      pid->out = 0.0f;
56  }
57
58
59  void IRAM_ATTR PID_controller_update(PIDController *pid, float setpoint, float
        measurement){
60
61      // Error Signal
62      float error = setpoint - measurement;
63
64      // Proportional Signal
65      float proportional = pid->Kp * error;
66
67      // Integral Signal
68      pid->integrator = pid->integrator + (pid->Ki * pid->T * (error +
        pid->prevError) * 0.5f);
69
70      /* Anti-wind-up via integrator clamping */
71      if (pid->integrator > pid->limMaxInt){
72          pid->integrator = pid->limMaxInt;
73      }
74
75      else if (pid->integrator < pid->limMinInt){
76          pid->integrator = pid->limMinInt;
77      }
78
79      // Derivative Signal with low pass filtering
80      pid->differentiator = -(2.0f * pid->Kd * (measurement - pid->prevMeasurement)
        /* Note: derivative on measurement, therefore minus sign in front of equation!
        */
81                              +(2.0f * pid->tau - pid->T) * pid->differentiator) /
82                               (2.0f * pid->tau + pid->T);
83
84      /*
85    * Compute output and apply limits
86    */
87      pid->out = proportional + pid->integrator + pid->differentiator;
88
89      if (pid->out > pid->limMax){
90          pid->out = pid->limMax;
91      }
92
93      else if (pid->out < pid->limMin){
94          pid->out = pid->limMin;
95      }
96
97      /* Store error and measurement for later use */
98      pid->prevError = error;
99      pid->prevMeasurement = measurement;
100 }
101
102 #endif //BUCK_CONVERTER_PID
```

code/pid.h

D.1.3 pwm.h

```
#ifndef BUCK_CONVERTER_PWM
#define BUCK_CONVERTER_PWM

// Hardware driver includes
#include "driver/ledc.h"

/*
 * PWM set up defines and helper macros
 */
#define DUTY_RESOLUTION LEDC_TIMER_9_BIT
#define FREQUENCY_MIN   1000
#define FREQUENCY_MAX   100000
#define DUTY_STEPS      512
#define DUTY(X)         ((1.0 - X) * DUTY_STEPS) // Duty cycle calculation macro

#endif


#ifdef BUCK_CONVERTER_PWM_IMPL

/*
 *   Set up the PWM timer and channel structs, and configure the PWM output
 */

void PWM_setup(ledc_timer_config_t *pwm_timer, ledc_channel_config_t *pwm_channel,
    uint32_t frequency, float duty_cycle){

    // Select and set up the PWM clock source and initial frequency
    ledc_timer_config_t timer = {
        .duty_resolution = DUTY_RESOLUTION,    // resolution of PWM duty
        .freq_hz = frequency,                  // frequency of PWM signal
        .speed_mode = LEDC_HIGH_SPEED_MODE,    // timer mode
        .timer_num = LEDC_TIMER_0,             // timer index
        .clk_cfg = LEDC_AUTO_CLK,              // Auto select the source clock
    };

    // Select and set up the PWM output channel and initial duty cycle
    ledc_channel_config_t channel = {
        .channel    = LEDC_CHANNEL_0,          // PWM output channel (0 - 4)
        .duty       = DUTY(duty_cycle),        // PWM duty cycle
        .gpio_num   = 23,                      // PWM output GPIO
        .speed_mode = LEDC_HIGH_SPEED_MODE,    // PWM output mode
        .timer_sel  = LEDC_TIMER_0             // PWM timer index
    };

    *pwm_timer = timer;
    *pwm_channel = channel;

    ledc_timer_config(pwm_timer);
    ledc_channel_config(pwm_channel);
}

/*
 *   Set a new PWM duty cycle for a given PWM channel
```

```
54  */
55
56 esp_err_t PWM_set_duty(ledc_channel_config_t *pwm_channel, float duty_cycle){
57
58     // Set the new PWM Duty Cycle of the given channel configuration
59     ledc_set_duty(pwm_channel -> speed_mode, pwm_channel -> channel,
       DUTY(duty_cycle));
60
61     // Update the PWM channel with the new configuration
62     return ledc_update_duty(pwm_channel -> speed_mode, pwm_channel ->channel);
63 }
64
65
66 /*
67  *   Set a new PWM frequency for a given PWM channel
68  */
69
70 esp_err_t PWM_set_frequency(ledc_channel_config_t *pwm_channel, ledc_timer_config_t
       *pwm_timer, uint32_t frequency){
71
72     if(frequency > FREQUENCY_MAX || frequency < FREQUENCY_MIN)
73     {
74         printf("Specified input frequency not within possible range: %d - %d\n",
       FREQUENCY_MIN, FREQUENCY_MAX);
75         return ESP_ERR_INVALID_ARG;
76     }
77
78     // Set the new PWM frequency of the given channel and timer configuration
79     return ledc_set_freq(pwm_channel -> speed_mode, pwm_timer -> timer_num,
       frequency);
80 }
81
82 #endif // BUCK_CONVERTER_PWM
```

code/pwm.h

D.2 Application Code

D.2.1 buck_converter.h

```
#ifndef BUCK_CONVERTER
#define BUCK_CONVERTER

// Project includes
#include "pwm.h"
#include "adc.h"
#include "pid.h"

// Standard includes for math and print functions
#include <stdio.h>
#include <math.h>
#include <string.h>

// RTOS includes for error messages and task handling
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "esp_err.h"

void init_buck();
void control_loop(void *pvParameters);

/*
 * PWM timer and channel structs
 */
ledc_timer_config_t pwm_timer;     // Create the PWM timer struct
ledc_channel_config_t pwm_channel; // Create the PWM channel struct

/*
 * Constants for ADC conversion
 */
static const float LOAD_R1 = 56237.0f; // Voltage divider values from buck output
    load
static const float LOAD_R2 = 22035.0f + 995.2f;

static const float SUPPLY_R1 = 565300.0f; // Voltage divider values from buck
    output load
static const float SUPPLY_R2 = 119700.0f;

// Internal esp ADC structs
esp_adc v_supply;

/*
 * Constants, functions, and variables for PID controller
 */
static const float VO = 3.0f;  // The desired initial output of the converter.

// Controller gains
static const float KP = 0.16f;
static const float KI = 17.1f;
static const float KD = 0.0f;

```

```
51 // Controller Sample time period in ms
52 static const float TS = 2.0f;
53
54 #endif // BUCK_CONVERTER
```

code/buck_converter.h

D.2.2 buck_converter.c

```
// Include single header library function implementations
#define BUCK_CONVERTER_PWM_IMPL
#define BUCK_CONVERTER_ADC_IMPL
#define BUCK_CONVERTER_PID_IMPL

#include "buck_converter.h"

void init_buck()
{
    // PWM set up
    PWM_setup(&pwm_timer, &pwm_channel, 50000, 0);
}

/*
 * Output voltage PID control loop.
 * This function initialises the PID controller, and the internal ADC on the ESP32.
 * It will then enter the main control loop, and will compute the following:
 *
 *      - Read the current buck converter load voltage. This is Measured using the
 *        ESP32 ADC, through the voltage divider defined by LOAD_R1 and LOAD_R2.
 *
 *      - Calculate the equivalent duty cycle that would be used to produce the
 *        current measured load voltage.
 *
 *      - Pass the target duty cycle, and current duty cycle through the PID
   controller
 *
 *      - Update the buck converter duty cycle with the new controller output
 *
 *      - Delay the task until the selected sample period defined by TS has elapsed
 */

void control_loop(void *pvParameters)
{
    // Setup an input queue for receiving new target voltages
    QueueHandle_t target_voltage_queue = *(QueueHandle_t *)pvParameters;

    // Load voltage ADC set up and local variables
    uint16_t supply_voltage_buffer[5] = {0}; // Supply voltage rolling average
    buffer

    // Variables for storage and conversion of load voltage
    uint16_t load_adc_raw;
    float load_adc_voltage;
    float load_voltage;
    float measurment_duty;

    // Variables for storage and conversion of
    uint16_t supply_adc_raw;
    float supply_adc_voltage;
    float supply_voltage;

    // Initialise the ADC structs
    esp_adc v_load = {// Create adc struct for reading the load voltage
```

```
                    // adc input channel 0 (GPIO 36)
                    .adc_channel = 0,

                    // do not allow for a rolling average
                    .span = 0,
                    .buffer = NULL};

esp_adc v_supply = {// Create adc struct for reading the supply voltage
                     // adc input channel 6 (GPIO 34)
                     .adc_channel = 6,

                     // 5 wide rolling average
                     .span = 5,
                     .buffer = (uint16_t *)&supply_voltage_buffer};

// Init the load voltage adc and check to see if it was successful
if (adc_init(&v_load) != ESP_OK)
{
    printf("Load voltage ADC init error!\n"); // If set up fails print the error
    vTaskDelete(NULL);
}

// Init the supply voltage adc and check to see if it was successful
if (adc_init(&v_supply) != ESP_OK)
{
    printf("Supply voltage ADC init error!\n"); // If set up fails print the
error
    vTaskDelete(NULL);
}

//PID set up
float target_voltage = V0;
float target_duty;

// Create the PID struct
PIDController voltage_controller = {
    // Controller gains
    .Kp = KP,
    .Ki = KI,
    .Kd = KD,

    // Differentiator low pass filter corner frequency
    .tau = 0.0f,

    // Min and max output duty cycles
    .limMin = 0.0f,  // 0% duty cycle
    .limMax = 0.98f, // 95% duty cycle

    // Integrator wind-up min and max
    .limMinInt = -5.0f,
    .limMaxInt = 5.0f,

    // Sample time period
    .T = TS / 100.0f,
};

```

```
108     // initialise the PID controller
109     PID_controller_init(&voltage_controller);
110 
111     // Task variables
112     TickType_t xLastWakeTime; // Hold the time stamp of the last wake time
113 
114     // Enter the PID control loop
115     while (true)
116     {
117 
118         xLastWakeTime = xTaskGetTickCount(); // Store the current tick count
119 
120         // Read the supply voltage ADC
121         supply_adc_raw = adc_read(&v_supply);
122         supply_adc_raw = rolling_average(&v_supply, supply_adc_raw);
123         supply_adc_voltage = adc_conversion(supply_adc_raw);
124         supply_voltage = (supply_adc_voltage * (SUPPLY_R1 + SUPPLY_R2)) / SUPPLY_R2;
125 
126         // Read the load voltage ADC
127         load_adc_raw = adc_read(&v_load); // Take an adc reading
128         load_adc_voltage = adc_conversion(load_adc_raw); // Convert this value to
    the voltage at the adc
129         load_voltage = (load_adc_voltage * (LOAD_R1 + LOAD_R2)) / LOAD_R2; //
    Convert to the voltage at the buck converter load
130 
131         // Calculate the measured duty cycle and target duty cycle
132         measurment_duty = load_voltage / supply_voltage; // Calculate the duty
    theoretical duty cycle of the output
133         target_duty = target_voltage / supply_voltage; // Calculate the target duty
    cycle for the controller
134 
135         // Check for a new target voltage
136         if (uxQueueMessagesWaiting(target_voltage_queue))
137         { // If there is a new target voltage read it
138             if (xQueueReceive(target_voltage_queue, &target_voltage, (TickType_t)1))
139             {
140                 target_duty = target_voltage / supply_voltage;
141             }
142         }
143 
144         // Update the duty cycle with the PID controller
145         PID_controller_update(&voltage_controller, target_duty, measurment_duty);
146 
147         // Output the new duty cycle
148         PWM_set_duty(&pwm_channel, voltage_controller.out);
149 
150         // Delay task until the provided time period is reached
151         vTaskDelayUntil(&xLastWakeTime, TS);
152     }
153 }
```

code/buck_coverter.c

D.2.3 main.c

```
#include "buck_converter.h"

void app_main()
{

    init_buck();

    // Declare task handles
    TaskHandle_t VO_controller = NULL;

    QueueHandle_t target_voltage_queue;

    // initialise the queues
    target_voltage_queue = xQueueCreate(5, sizeof(float));

    // Create the output voltage control task
    xTaskCreatePinnedToCore(control_loop,                   // Voltage control loop function
                            "Vout_controller",              // Task name
                            2048,                           // Task stack size
                            (void *)&target_voltage_queue, // Function parameters
                            2,                              // Priority of the task (app_main has priority 1)
                            &VO_controller,                 // Task handle
                            tskNO_AFFINITY);                // Core the task has been pinned to (No core selected)

    float input_voltage;
    char input_buffer[20] = {0};
    uint8_t buf_index = 0;

    while (true)
    {
        char input = fgetc(stdin);

        if (input != 0xFF)
        {
            input_buffer[buf_index++] = input; // read in the character, and increment the index
            fputc(input, stdout);              //echo the character back

            if ((input_buffer[buf_index - 1] == '\n'))
            {

                input_voltage = strtof(input_buffer, NULL);

                xQueueSend(target_voltage_queue, (void *)&input_voltage, (TickType_t)1);

                printf("\nInput Target Voltage: %f\n", strtof(input_buffer, NULL));
                buf_index = 0;
            }
        }

```

```
50          vTaskDelay(1);
51
52          // printf("task stack unused: %d\n\n",
        uxTaskGetStackHighWaterMark(VO_controller));
53          // vTaskDelay(100);
54      }
55  }
```

code/main.c