

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

1	Introduction	1
1.1	Project Motivation	1
1.2	Project Goals	1
2	Background	3
2.1	Pulse Width Modulated Signal Generation	3
2.1.1	Analogue PWM Signal Generation	3
2.1.2	Digital PWM Signal Generation	4
2.2	Buck Converters	4
2.2.1	Buck Converter Design	5
2.3	Current Sensing	6
2.3.1	Current Sense Amplification	6
2.3.2	Hall Effect Sensors	6
2.4	Control Systems	7
3	Design	8
3.1	System Specifications and Architecture	8
3.1.1	System Architecture	9
3.2	PWM Signal Generator	10
3.2.1	Analogue PWM Generator Design	10
3.2.2	Digital PWM Generator Design	11
3.3	System State Sensing	11
3.3.1	Current Sensor Selection	11
3.3.2	Mean Inductor Current Sensing	11
3.3.3	Peak Inductor Current Sensing	12
3.3.4	Output Voltage Sensing	12
3.4	Control System	12
3.4.1	Output Load Voltage Controller	12
3.4.2	Inductor Current Ripple Controller	12
4	Implementation	13
4.1	PWM Generation	13
4.2	Inductor Current Peak to Peak Sensor	13
4.3	Control System	13
4.4	Software Architecture	13
5	Evaluation	14
5.1	PWM Generation	14
5.2	Inductor Current Ripple Sensor	14
5.3	Control System	14

6	Conclusions & Future Work	15
A	System Specification Derivation	18
B	PWM Generation Figures	20
C	Peak Detector Figures & Tables	23
D	Control System Figures	30
E	Schematic & Printed Circuit Board	35
F	Source Code	37

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 current. 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

The goal of this project is to develop a testing platform through which the effects of variable buck converter switching frequency on inductor current ripple can be observed and controlled. The aim of this is to eliminate the need to design the output stage of the buck converter by implementing a control system to maintain a specified inductor current ripple, and output voltage.

To achieve this goal, a proof of concept buck converter will be designed to operate as the testing platform. This converter will require a variable frequency pulse width modulated

(PWM) signal generator, and inductor current ripple sensing. This will allow for direct manipulation and control of inductor current ripple. The converter will also need to implement the full functionality of existing buck converter designs, including controlled output voltage regulation.

It is also important to note that as this project aims to produce a testing platform to be used as a proof of concept, there will be continued work undertaken after the projects conclusion. Because of this, it is important that the design and implementation undertaken in this project considers the ease of use and future expansion of the project.

The following list of specifications outlines the requirements the designed test platform will meet:

1. Operate with a 12V DC input supply voltage
2. Provide a selectable DC output voltage between 3V and 10V
3. Provide an output voltage precision of at least $\pm 5\%$ of the targeted output voltage
4. Provide a selectable inductor current ripple between 20% and 50% of the total output current
5. Provide a variable converter switching frequency between 1kHz and 100kHz
6. Provide an inductor current ripple precision of at least $\pm 5\%$ of the target inductor current ripple
7. Operate with variable load sizes between 10Ω and 20Ω

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.

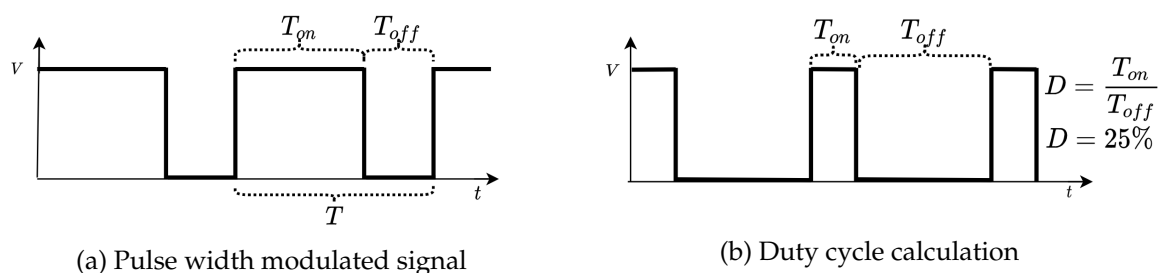


Figure 2.1: Pulse width modulated signal characteristics

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.

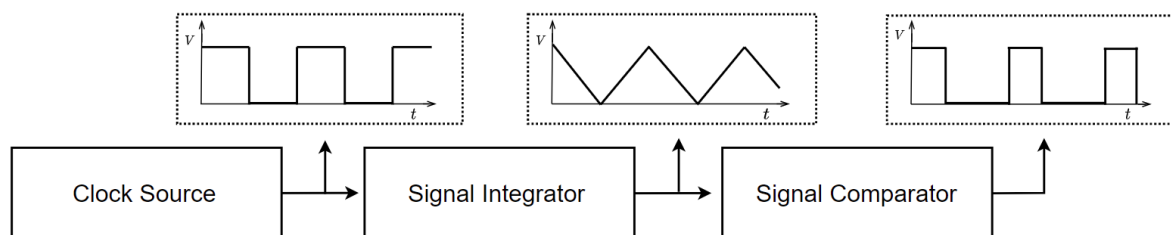


Figure 2.2: Stages of analogue PWM generation

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 timing register that is continually incrementing at a known period we can set a period for our PWM. It is possible to vary the period of the PWM by simply increasing or decreasing the timing registers clock. Next, by toggling a digital I/O when a compare variable is equal to the current value of the timer, we are able to generate a PWM signal with a variable duty cycle [7]. This timing architecture can be simply implemented within an FPGA, but it can also commonly be found within the hardware of most microcontrollers. It should be noted however, that the range of achievable frequencies and duty cycle accuracy will be dependant on an individual microcontrollers 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 through 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).

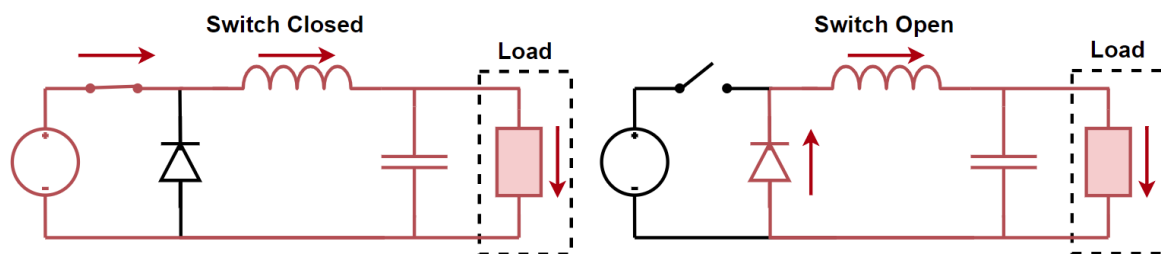


Figure 2.3: Operating states of a buck converter

2.2.1 Buck Converter Design

The design of a common buck converter has two primary considerations, the output voltage of the converter V_o , 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.

$$V_o = D \cdot V_{in} \quad (2.1)$$

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.

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

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

Current sensors are transducers that convert an input current to an easily measured output voltage that is proportional to the current through the sensor. Current sensors operate on one of two principles, Ohm's law, Faraday's and Ampere's law [11]. Ohm's law describes how when current flows through a resistive element a voltage drop will occur. Sensors operating under this principle will measure the voltage drop across a known resistance, and are referred to as direct sensors as they directly effect the circuit. Faraday's and Ampere's law describes how a changing electric flux will induce a changing magnetic field, and a changing magnetic field will induce a changing magnetic flux. Sensors operating under this principle will use the magnetic field induced by the current through the sensor to produce a voltage proportional to that current, and are referred to as indirect sensors as they make no direct contact with the circuit being sensed.

2.3.1 Current Sense Amplification

Current sense voltage amplification sensors are a form of direct sensor, working on the 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 of known gain. This combination allows for accurate amplification of the shunts voltage drop, increasing the precision of measurements, and allowing for drastically smaller shunt resistors.

2.3.2 Hall Effect Sensors

Hall effect sensors are a form of indirect sensor, working on the principles of Faraday's and Ampere's law as discussed above. They function by measuring the magnetic flux density, meaning that they are commonly used to sense the presence of a magnets or magnetic fields. However since a current flowing through a wire will produce a magnetic flux, they can also be used to measure current flow.

Due to 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. This provides large safety advantages, eliminates sensing power losses, 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 within the surrounding environment. 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 [12].

2.4 Control Systems

Control systems are integral to the operation of any system for which you wish to achieve a desired outcome. In their simplest, a control system has the purpose of obtaining a desired output from a system, with a desired performance, when a given input is provided.

There are two main topologies of control system, open loop and closed loop. An open loop control system requires no feedback of the output, and therefore no sensing. Because of this, open loop controllers are often very easy and fast to implement within a system because of this, however they have no knowledge of the system's state and are therefore unable to correct for errors such as external disturbances.

On the other hand, closed loop controllers require feedback from the current output of a system, and therefore will require sensing to be implemented. This added sensing requirement will add complexity to the system, however a closed loop controller provides the ability to compensate for errors in the system's output.

The performance of a specific control system is often discussed in terms of its steady state settling time, and its steady state error. The steady state settling time of a controller describes the time it takes for a controller to reach its final value ('steady state') when provided with a given input. The slower the steady settling time of a controller, the slower it will react to a given input, and vice versa. The steady state error of a controller describes the error between the desired output and the final output of the system once its steady state has been achieved.

Chapter 3: Design

The design processes outlined within this report will draw heavily on the background research discussed in Chapter 2. These designs aim to effectively implement the outlined system requirements discussed by Section 1.2 in a robust and repeatable manner.

3.1 System Specifications and Architecture

Section 1.2 outlines a list of specifications that this project must achieve. For each item in this list, design specifications can be derived to ensure that our designs are capable of meeting the outlined requirements. This list of specifications will also help to shape the architectural approach taken on this project.

Requirement 1: Operate with a 12V DC input supply voltage

To meet this requirement, the buck converter must be designed to be 12V DC compatible. This will require all designed hardware, including additional digital and analog circuitry to be capable of operating with this supply.

Requirement 2: Provide a selectable DC output voltage between 3V and 10V

To meet this requirement, the designed buck converter must have the ability to change the duty cycle of its PWM switching signal. This will allow for variation of the output voltage based on Equation (2.1).

Requirement 3: Provide an output voltage precision of at least $\pm 5\%$ of the targeted output voltage

To meet this requirement, the designed buck converter must have a control system to regulate the output voltage. To achieve a steady state error of less than $\pm 5\%$, feedback should be implemented, requiring output voltage sensing. We can also specify that a minimum duty cycle resolution of 1.25% will be required to achieve this. For the derivation of these values, please see Appendix A.

Requirement 4: Provide a selectable inductor current ripple between 20% and 50% of the total output current

To meet this requirement, the designed buck converter must have the ability to change the frequency of its PWM switching signal. This will allow for variation of the inductor current ripple based on equation Equation (2.2).

Requirement 5: Provide a variable converter switching frequency between 1kHz and 100kHz

To meet this requirement, the PWM switching frequency must be selectable across the specified range. This, in tandem with requirement 4, allow for the specification of the inductor range for which these requirements can be met. For the derivation of these values, please see Appendix A.

Requirement 6: Provide an inductor current ripple precision of at least $\pm 5\%$ of the target inductor current ripple

To meet this requirement, the designed buck converter must have a control system to regulate the inductor current ripple. To achieve a steady state error of $\pm 5\%$, feedback should be implemented, requiring inductor current ripple sensing. We can also specify that a minimum frequency selection resolution of 200Hz will be required to achieve this. For the derivation of these values, please see Appendix A.

Requirement 7: Operate with variable load sizes between 10Ω and 20Ω

To meet this requirement, all components of the buck converter must be designed to handle the maximum output current of 1A that will occur with a 10Ω load at 10V. It will also be important that any current sensing is designed to operate within the range defined by these loads. Finally it is required that control systems are implemented to maintain requirements 3, & 6 as the load varies.

3.1.1 System Architecture

Based on the specifications outlined in the above section, an architecture that outlines the system design has been developed. This architecture has been divided into three subsystems that must each be developed. This architecture system can be seen in Figure 3.1.

Subsystem one implements a PWM generator. From requirements 2, & 5 it has been identified that a PWM generator capable of varying both the duty cycle and the frequency of it's output signal independently must be designed. This system can be seen in Figure 3.1 outlined in green.

Subsystem two implements system state sensing. From requirement 3 & 6 it has been identified that an output voltage sensor, and an inductor current ripple sensor must be designed. This system can be seen in Figure 3.1 outlined in blue.

Subsystem three implements two control systems. From requirements 3, 6, & 7 it has been identified that control systems should be designed to ensure that the specified steady state errors of the output voltage and inductor current ripple are met.

One control system will take the output load voltage measurement, and control the PWM duty cycle. The other control system will take the inductor current ripple, and control the PWM frequency. This system can be seen in Figure 3.1 outlined in red.

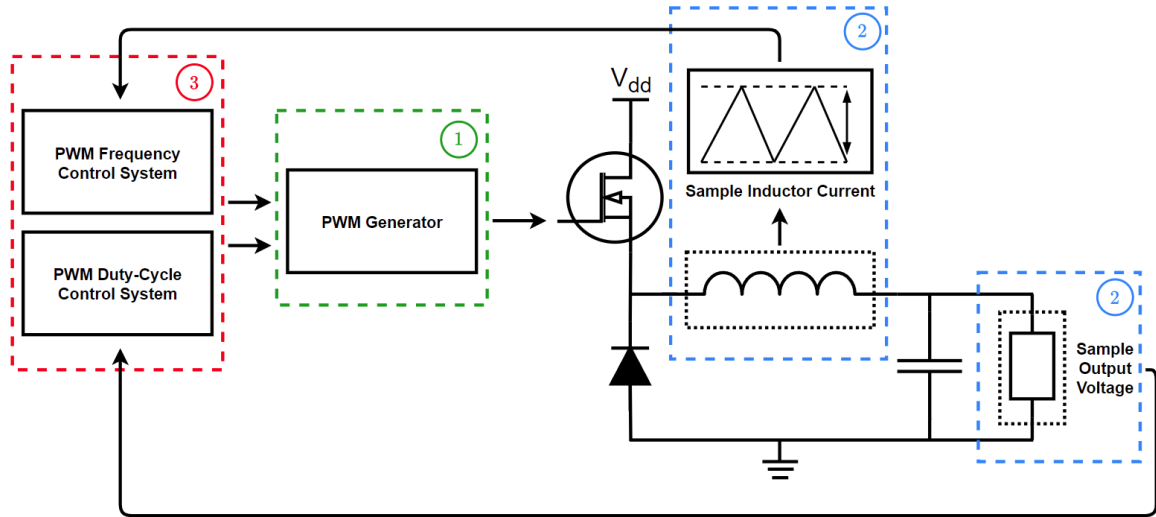


Figure 3.1: System architecture design, with three internal subsystems

3.2 PWM Signal Generator

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 affect the overall design of the system architecture. This section will discuss these designs and finalise the design of the PWM generator for this system.

3.2.1 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 its performance, and can be seen in Appendix B. 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)

3.2.2 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 [13]. 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 B.

3.3 System State 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 peak current value, mean current, and output voltage).

3.3.1 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 2.3.2. 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.

3.3.2 Mean Inductor Current Sensing

discus how by low pass filtering the signal we are easily able to obtain the mean of the analog signal. Using this, the output of the current sense amplifier is input into a $10Hz$ cut-off frequency low pass filter, that will easily filter the entire frequency range. Then talk quickly about how this value can be easily sampled using the internal ADC as it vary greatly and will not need the precision.

3.3.3 Peak Inductor Current Sensing

discuss why it is that peak sensing is important, and how two designs have been created to achieve this task. The precision rectifier is my own design and not based on any existing designs, the sample and hold based sensor is based on designs from these resources [14, 15]

Precision Rectifier Peak Voltage Detection

Discuss the operation, and simulations of the Precision Rectifier Peak Voltage Detection.

Sample and Hold Peak Voltage Detection

Discuss the operation, and simulations of the Sample and Hold Peak Voltage Detection. Then discuss that due to it's simpler design, it's implementation will be far easier, the possible issue it may have will be easier to diagnose, and it's cost will be much lower. then specify that this was the final design selected for this section.

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.

3.3.4 Output Voltage Sensing

discuss how the output voltage control does not require high levels of precision, and such will be using the internal ADC of the ESP32 microcontroller.

3.4 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.

3.4.1 Output Load Voltage Controller

discuss the structure of the controller. It will take the output voltage as in input, convert that to the theoretical duty cycle required to produce that output, and then compare that to the target to get the error. This will then output a new duty cycle for the system.

Show the matlab simulations, and reference the state space model that needs to still be put in the appendix. discuss how the system was tuned using the matlab 'pidtune()' function, and converted to discrete time to be implemented on a microcontroller.

3.4.2 Inductor Current Ripple Controller

I'm unsure on what to put here... I never designed the controller, and was planning to just perform a parameter sweep of the proportional, integral, and differential constants. maybe I should remove this section?

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 only 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 its $V_{GS_{on}}$ 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 & Future Work

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. Familant 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/statcom technologies: 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] P. Jain, "Engineers garage - current sensors." Online, June 2012.
- [12] Texas Instruments, *Precision Magnetics – Solving the Stability Challenge in Isolated Current Sensing*, Aug. 2020.
- [13] Espressif Systems, *ESP32: Technical Reference Manual*, 4.4 ed., 2021.
- [14] R. Elliott, "Peak detection circuits." Online, Mar. 2017.
- [15] H. Kelley and G. Alonso, "Ltc6244 high speed peak detector," techreport, Analog Devices, 2020.

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: System Specification Derivation

PWM Duty Cycle Resolution Calculations

It has been specified that the output voltage error should be within $\pm 5\%$ of the output voltage. This allowable error will be smallest with the smallest output voltage, and will therefore define the smallest allowable duty cycle resolution.

Calculate the allowable output voltage error:

$$\begin{aligned} V_{error} &= 3 \cdot 0.05 \\ &= 0.15V \end{aligned} \tag{A.1}$$

Calculate the minimum number of discrete voltage steps based on Equation (2.1)

$$\begin{aligned} N_{steps} &= \frac{V_{in}}{V_{error}} = \frac{12}{0.15} \\ &= 80 \end{aligned} \tag{A.2}$$

Calculate the resolution in bits:

$$\begin{aligned} N_{bits} &= \log_2(N_{steps}) = \log_2(80) \\ &= 6.3 \approx 7 \end{aligned} \tag{A.3}$$

This gives a minimum duty cycle resolution of 7 bits, or 128 discrete steps.

Minimum & Maximum Inductor Size Calculations

It has been specified that the operating frequency of the buck converter should be between 1kHz & 100kHz, with an inductor current ripple between 20% & 50%. Based on these specifications, Equation (2.2) can be used to derive the minimum and maximum inductor sizes that the system will continue to function with.

Derivation Constants:

$$R_{load} = 10\Omega, V_{in} = 12V, V_{max} = 10V, V_{min} = 3V, f_{min} = 1\text{kHz}, f_{max} = 100\text{kHz}$$

All calculations will be performed using the minimum allowable inductor current ripple of 20%, calculated below. This will ensure that all greater current ripple percentages are achievable by the system.

$$i_{min} = \frac{V_{min}}{R_{load}} \quad (A.4)$$

$$= 0.3A$$

(A.5)

$$\Delta i_{min} = i_{min} \cdot 0.2$$

$$= 0.06A$$

Minimum Inductor Size

Using Equation (A.6), we see that the minimum achievable inductor size will be at the maximum switching frequency. It can be noted that due to the quadratic nature of this equation, the maximum switching frequency for a constant ripple will occur at an output of $V_o = \frac{v_{in}}{2}$, with a duty cycle of 50%.

$$L = \frac{V_o \cdot (1 - D)}{\Delta i_L \cdot f_s} \quad (A.6)$$

Calculate the minimum inductor value:

$$L_{min} = \frac{\frac{v_{in}}{2} \cdot (1 - 0.5)}{\Delta i_{min} \cdot f_{max}} \quad (A.7)$$

$$= \frac{6 \cdot (1 - 0.5)}{100000 \cdot 0.06}$$

$$= 0.5mH$$

Maximum Inductor Size

From Equation (A.6), we can see that the maximum inductor value will be at the lowest switching frequency, with the greatest output voltage, and therefore the greatest duty cycle.

Calculate the maximum duty cycle:

$$D_{max} = \frac{V_{max}}{V_{in}} \quad (A.8)$$

$$= \frac{10}{12} = 83.3\%$$

Calculate the maximum inductor size:

$$L_{min} = \frac{V_{max} \cdot (1 - D_{max})}{\Delta i_{min} \cdot f_{min}} \quad (A.9)$$

$$= \frac{1 \cdot (1 - 0.83)}{1000 \cdot 0.06}$$

$$= 27.78mH$$

PWM Frequency Resolution Calculations

I need to put this here from desmos: <https://www.desmos.com/calculator/oscpnz2ozn>

Appendix B: PWM Generation Figures

Analogue PWM Generation

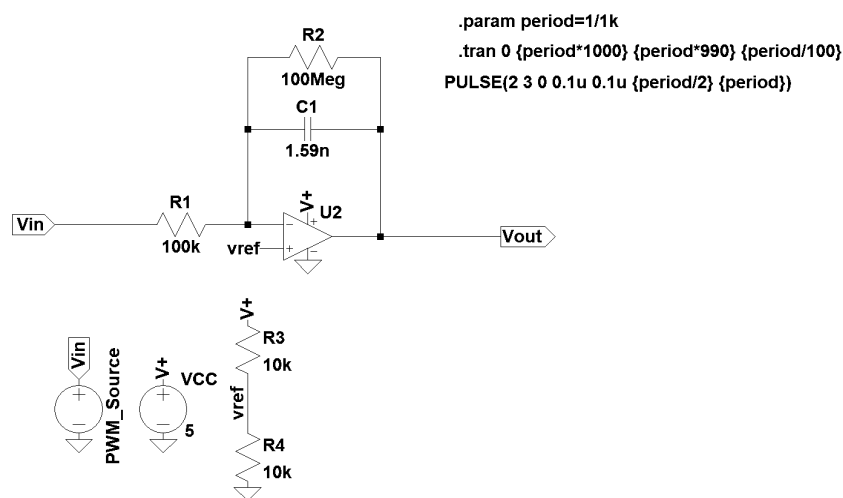


Figure B.1: Operational amplifier integrator LTSpice circuit simulation

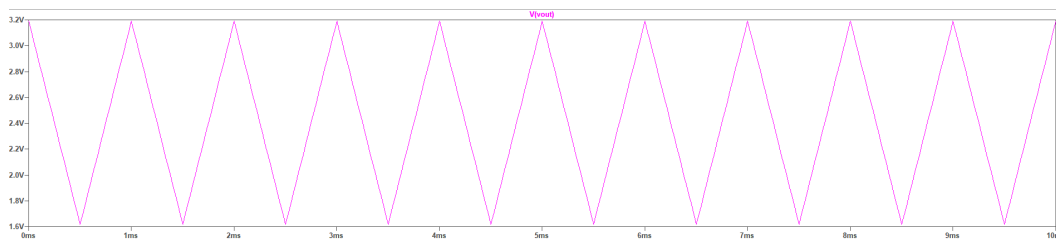


Figure B.2: Operational amplifier integrator simulation with a 1kHz square wave input. Output triangle wave amplitude of 1.5V

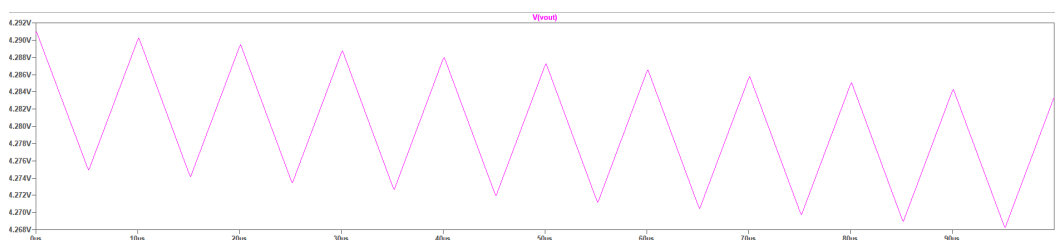
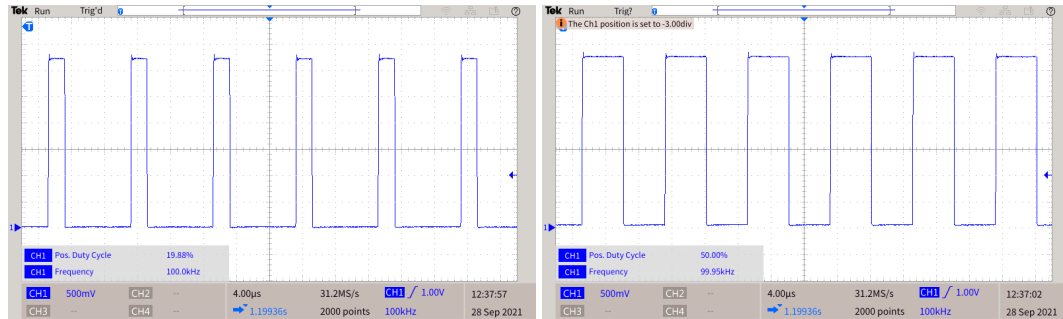


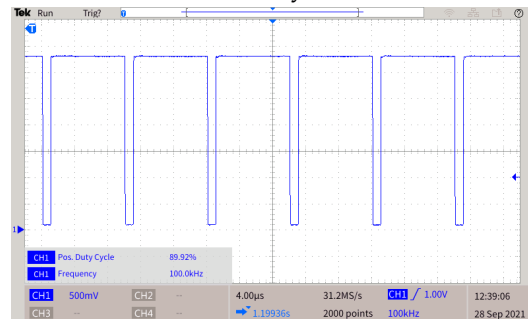
Figure B.3: Operational amplifier integrator simulation with a 1kHz square wave input. Output triangle wave amplitude of 20mV

Digital PWM Generation

PWM Duty Cycle Control



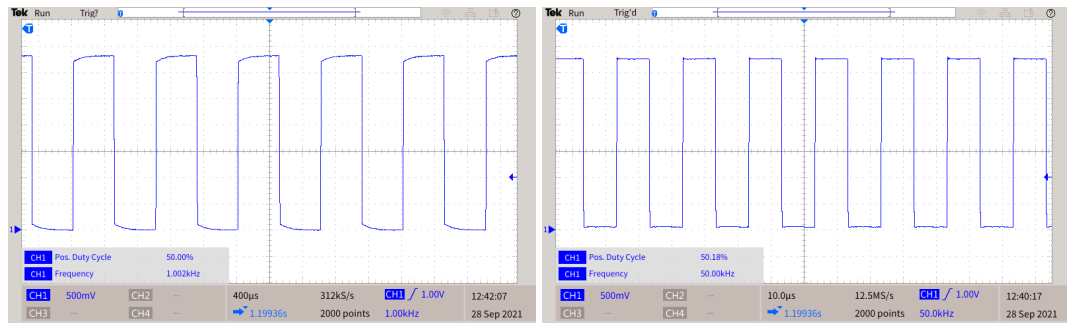
(a) Digital PWM Generation with a 20% duty cycle (b) Digital PWM Generation with a 50% duty cycle



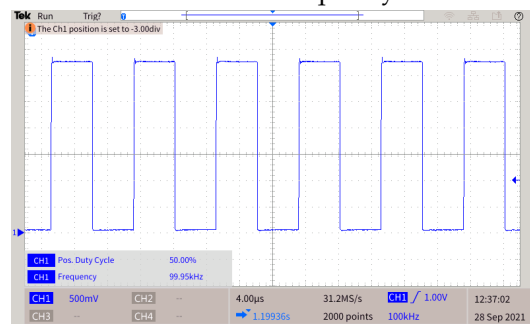
(c) Digital PWM Generation with a 90% duty cycle

Figure B.4: Digital PWM: Varying PWM duty cycle with a 100kHz frequency

PWM Frequency Control



(a) Digital PWM Generation with a 1kHz frequency (b) Digital PWM Generation with a 50kHz frequency



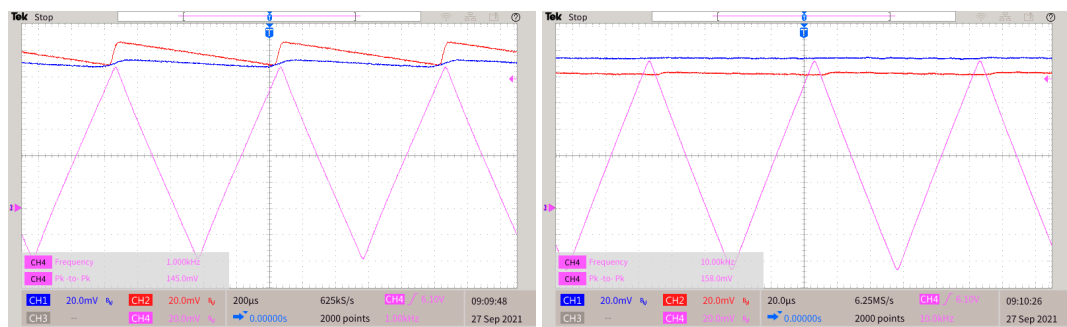
(c) Digital PWM Generation with a 100kHz frequency

Figure B.5: Digital PWM: Varying PWM frequency with a 50% duty cycle

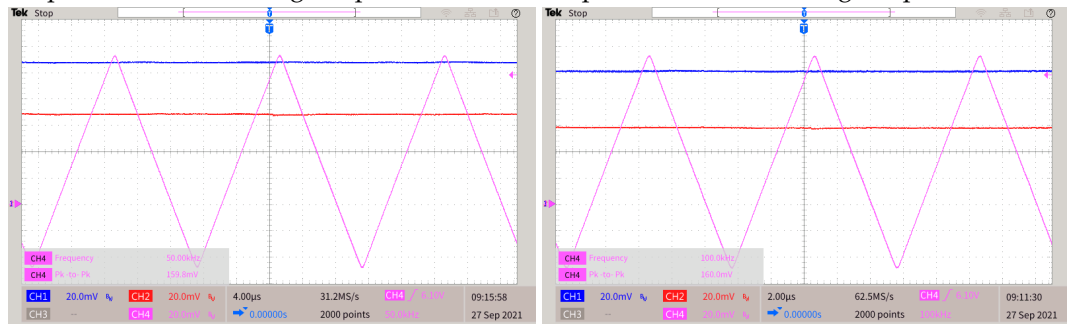
Appendix C: Peak Detector Figures & Tables

Initial & Final Peak Detector Output Comparison

Peak Detector Output for a 150mV Peak to Peak Triangle Input



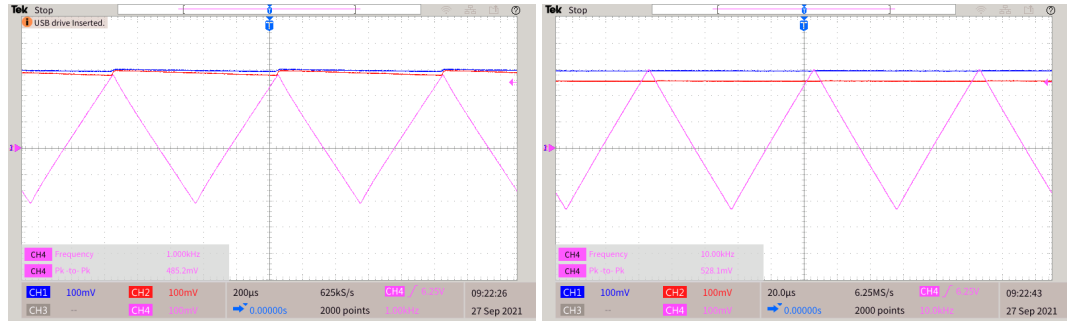
(a) Initial (Red) & final (Blue) peak detector outputs for a 1kHz triangle input. (b) Initial (Red) & final (Blue) peak detector outputs for a 10kHz triangle input.



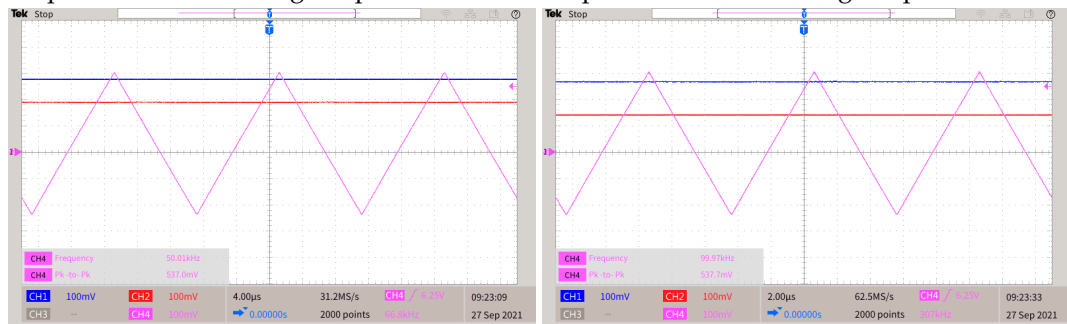
(c) Initial (Red) & final (Blue) peak detector outputs for a 50kHz triangle input. (d) Initial (Red) & final (Blue) peak detector outputs for a 100kHz triangle input.

Figure C.1: Initial & Final Peak Detector Output at varying frequencies for a 150mV peak to peak triangle input.

Peak Detector Output for a 500mV Peak to Peak Triangle Input



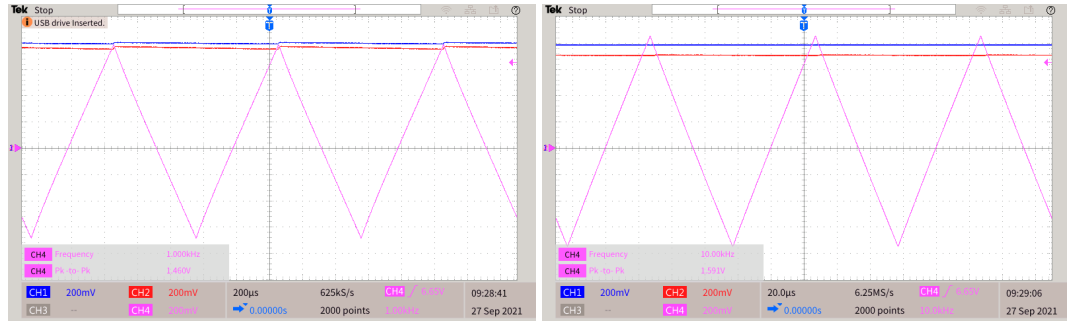
(a) Initial (Red) & final (Blue) peak detector outputs for a 1kHz triangle input. (b) Initial (Red) & final (Blue) peak detector outputs for a 10kHz triangle input.



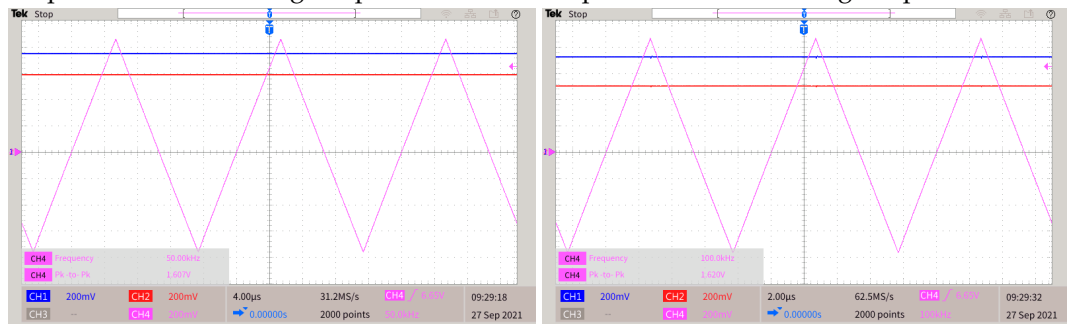
(c) Initial (Red) & final (Blue) peak detector outputs for a 50kHz triangle input. (d) Initial (Red) & final (Blue) peak detector outputs for a 100kHz triangle input.

Figure C.2: Initial & Final Peak Detector Output at varying frequencies for a 500mV peak to peak triangle input.

Peak Detector Output for a 1500mV Peak to Peak Triangle Input



(a) Initial (Red) & final (Blue) peak detector outputs for a 1kHz triangle input. (b) Initial (Red) & final (Blue) peak detector outputs for a 10kHz triangle input.



(c) Initial (Red) & final (Blue) peak detector outputs for a 50kHz triangle input. (d) Initial (Red) & final (Blue) peak detector outputs for a 100kHz triangle input.

Figure C.3: Initial & Final Peak Detector Output at varying frequencies for a 1500mV peak to peak triangle input.

Initial & Final Peak Detector Output Error Plots

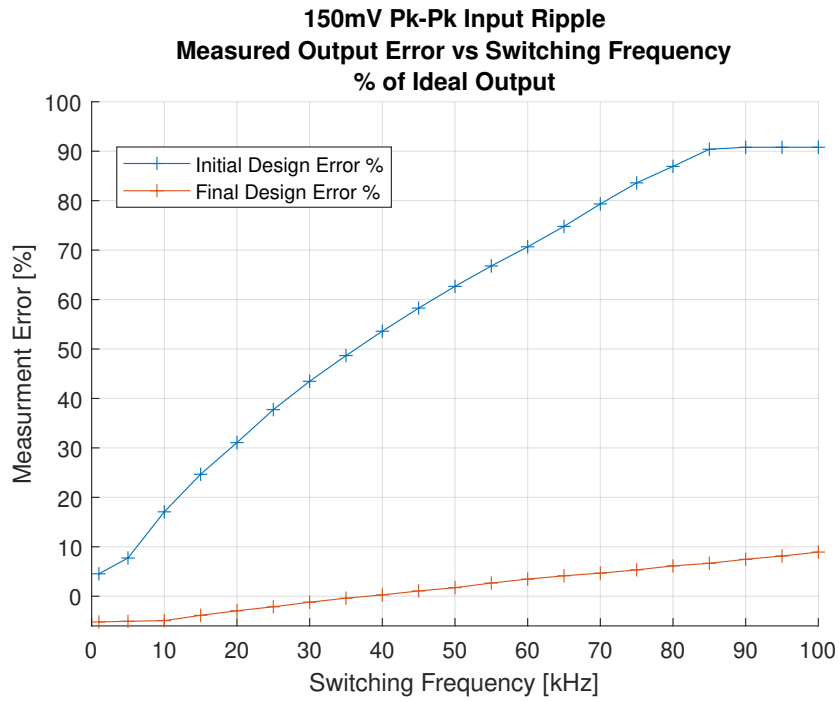


Figure C.4: Initial & final peak detector output error across frequencies for a 150mV peak to peak input. Displayed as a percentage of the ideal output.

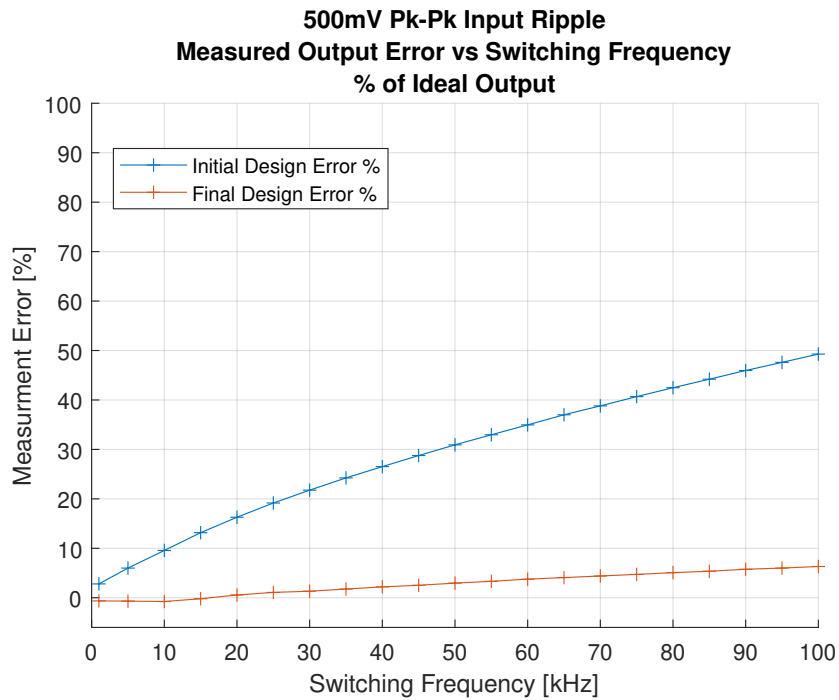


Figure C.5: Initial & final peak detector output error across frequencies for a 500mV peak to peak input. Displayed as a percentage of the ideal output.

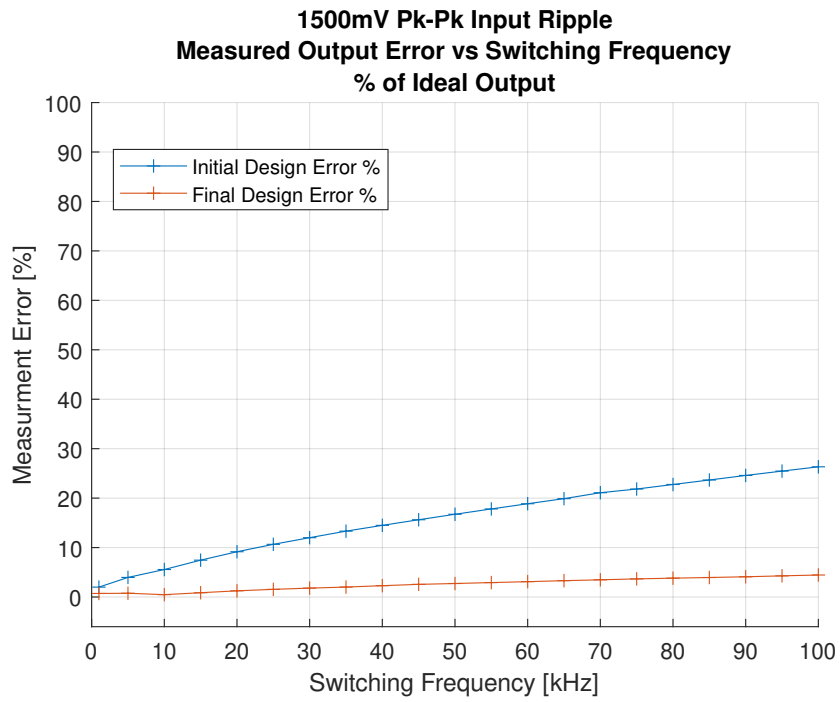


Figure C.6: Initial & final peak detector output error across frequencies for a 1500mV peak to peak input. Displayed as a percentage of the ideal output.

Initial & Final Peak Detector Output 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	4.53333333
5	-3.80	-5.06666667	5.80	7.73333333
10	-3.70	-4.93333333	12.80	17.06666667
15	-2.90	-3.86666667	18.50	24.66666667
20	-2.20	-2.93333333	23.30	31.06666667
25	-1.60	-2.13333333	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	0.26666667	40.20	53.6
45	0.80	1.06666667	43.70	58.26666667
50	1.30	1.73333333	47.00	62.66666667
55	2.00	2.66666667	50.10	66.8
60	2.60	3.46666667	53.00	70.66666667
65	3.10	4.13333333	56.10	74.8
70	3.50	4.66666667	59.50	79.33333333
75	4.00	5.33333333	62.70	83.6
80	4.60	6.13333333	65.20	86.93333333
85	5.00	6.66666667	67.80	90.4
90	5.60	7.46666667	68.10	90.8
95	6.10	8.13333333	68.10	90.8
100	6.70	8.93333333	68.10	90.8

Table C.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

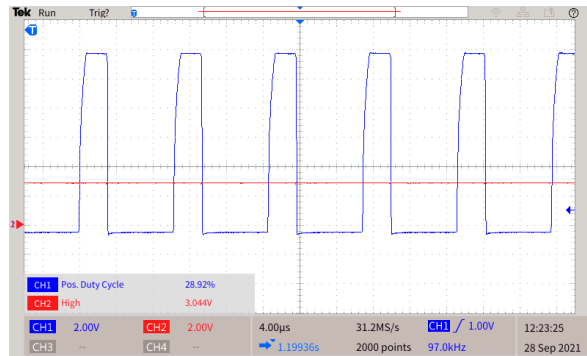
Table C.2: Table of output errors at varying frequencies for both the initial and final peak detection design with a 500mV input signal.

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	10.653333333
30	13.50	1.8	90.00	12
35	15.10	2.013333333	99.90	13.32
40	17.20	2.293333333	108.70	14.493333333
45	19.20	2.56	117.20	15.626666667
50	20.50	2.733333333	125.50	16.733333333
55	21.90	2.92	133.60	17.813333333
60	23.30	3.106666667	141.50	18.866666667
65	24.80	3.306666667	149.20	19.893333333
70	26.10	3.48	158.20	21.093333333
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	23.666666667
90	30.70	4.093333333	184.40	24.586666667
95	32.10	4.28	191.00	25.466666667
100	33.40	4.453333333	197.50	26.333333333

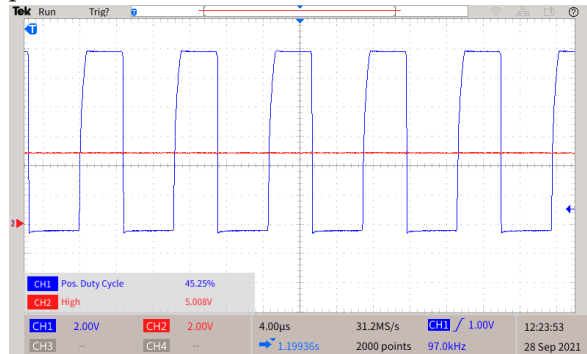
Table C.3: Table of output errors at varying frequencies for both the initial and final peak detection design with a 1500mV input signal.

Appendix D: Control System Figures

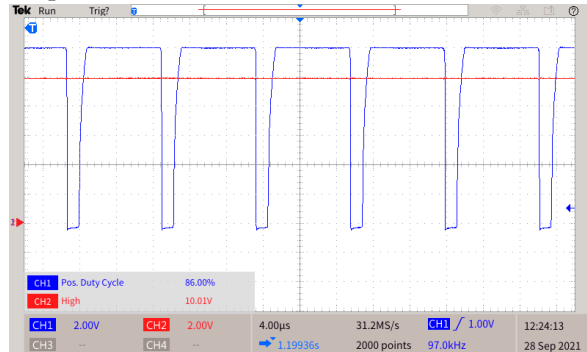
Variable Output Voltage Duty Cycle Control



(a) Controlled duty cycle of 28.9% for a 3V DC output



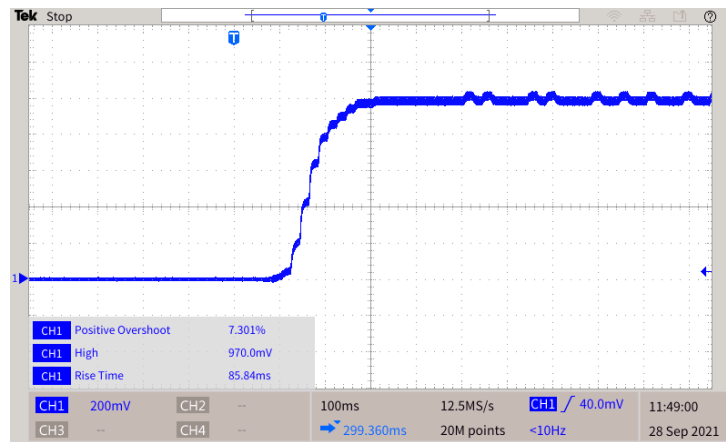
(b) Controlled duty cycle of 45.25% for a 5V DC output



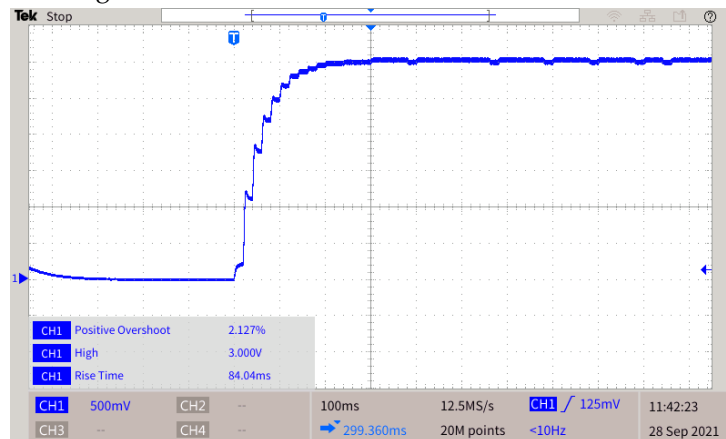
(c) Controlled duty cycle of 86.0% for a 10V DC output

Figure D.1: PWM duty cycle controlled by the voltage output control system (Blue), achieving 3V, 5V and 10V DC (Red).

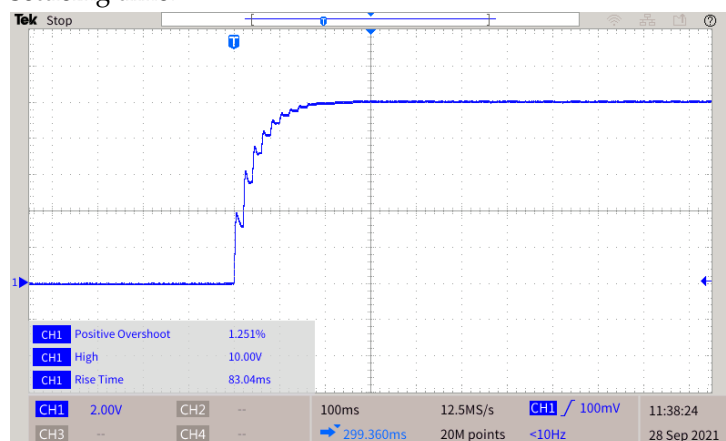
Output Voltage Control System Step Response Rise Time



(a) Output voltage controller 0V to 1V step response with 86ms settling time.



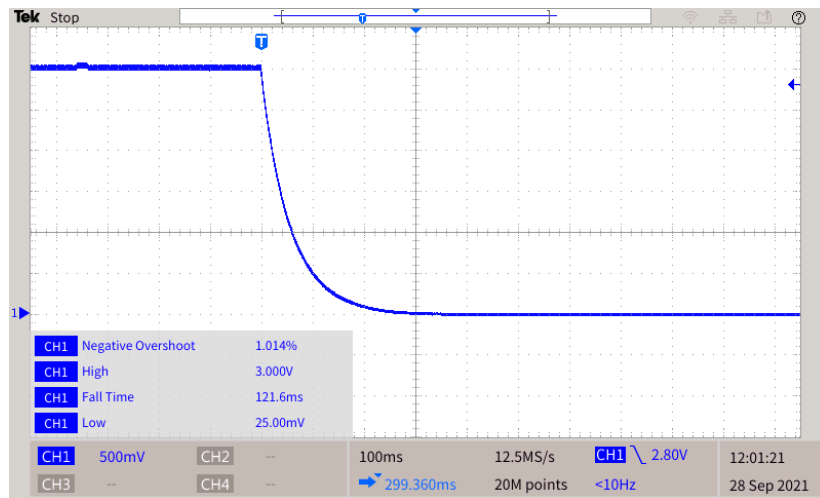
(b) Output voltage controller 0V to 3V step response with 84ms settling time.



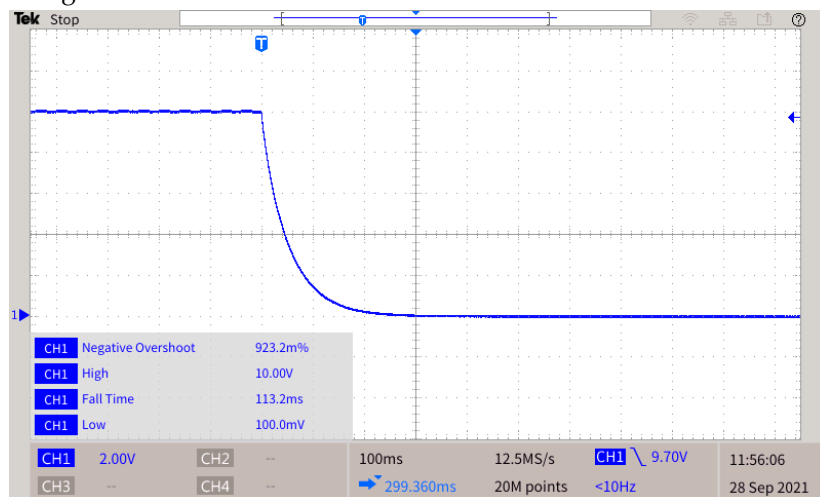
(c) Output voltage controller 0V to 10V step response with 83ms settling time.

Figure D.2: Output voltage control system step response, reacting to 1V, 3V and 10V output steps.

Output Voltage Control System Step Response Fall Time



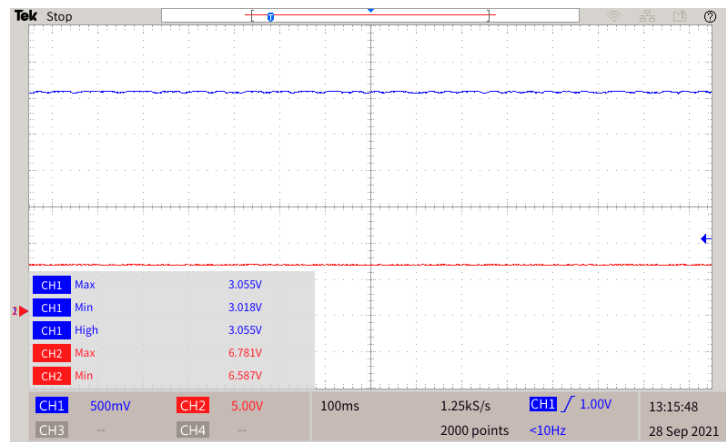
(a) Output voltage controller 3V to 0V step response with 121ms settling time.



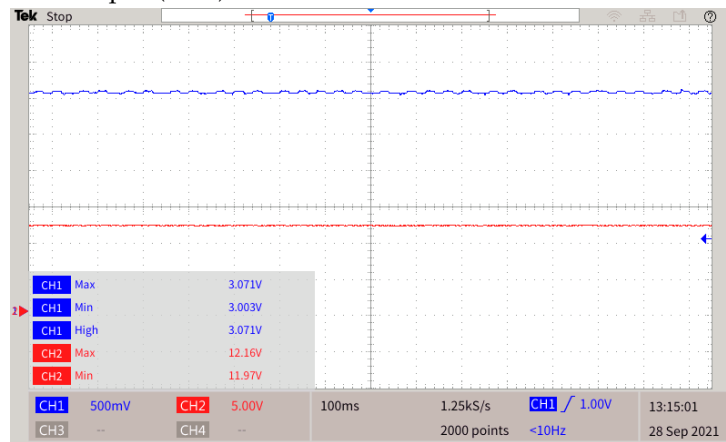
(b) Output voltage controller 10V to 0V step response with 113ms settling time.

Figure D.3: Output voltage control system step response, reacting to -3V and -10V output steps.

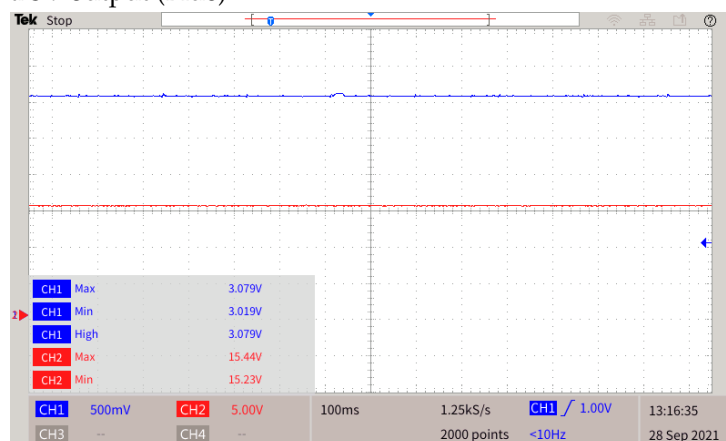
Output Voltage Control System with Variable Supply Voltage



(a) Output voltage controller regulating a 6V supply (Red) for a 3V output (Blue)



(b) Output voltage controller regulating a 12V supply (Red) for a 3V output (Blue)



(c) Output voltage controller regulating a 16V supply (Red) for a 3V output (Blue)

Figure D.4: Output voltage controller regulating variable supply voltages for constant output.

Output Voltage Control System Supply Voltage Step

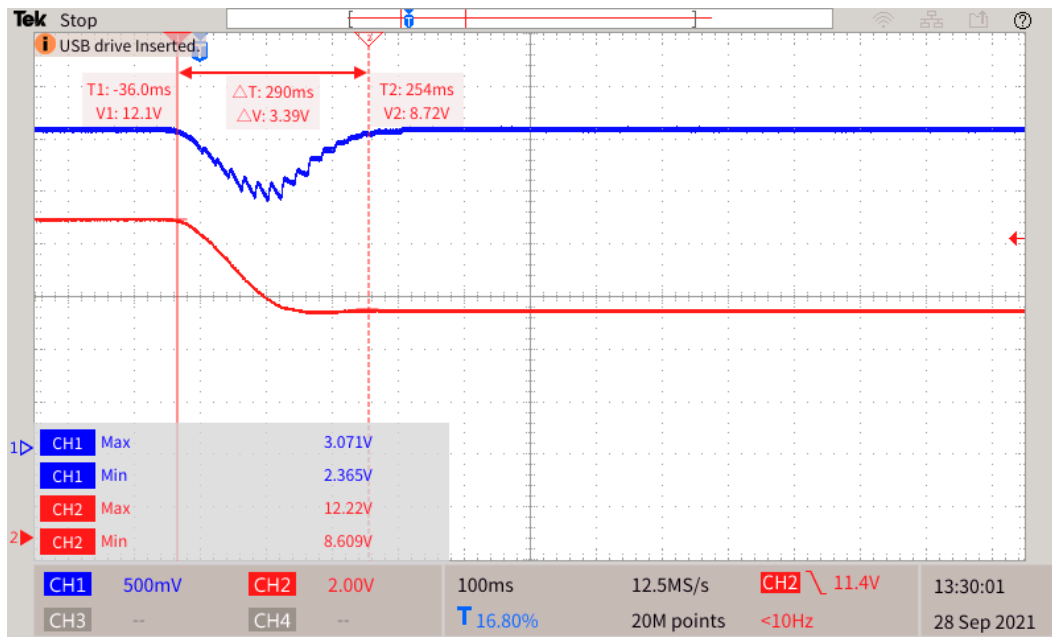


Figure D.5: Output voltage controller response (Blue) to a 12V to 8V supply voltage step (Red), with a recovery time of 290ms

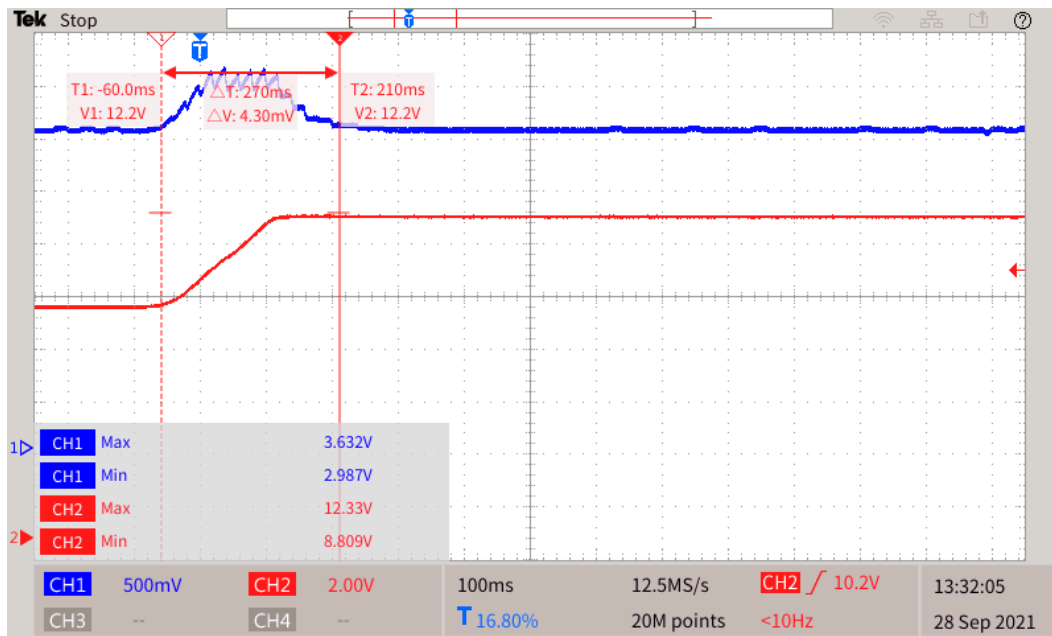


Figure D.6: Output voltage controller response (Blue) for a 8V to 12V supply voltage step (Red), with a recovery time of 270ms

Appendix E: Schematic & Printed Circuit Board

Full System Schematic

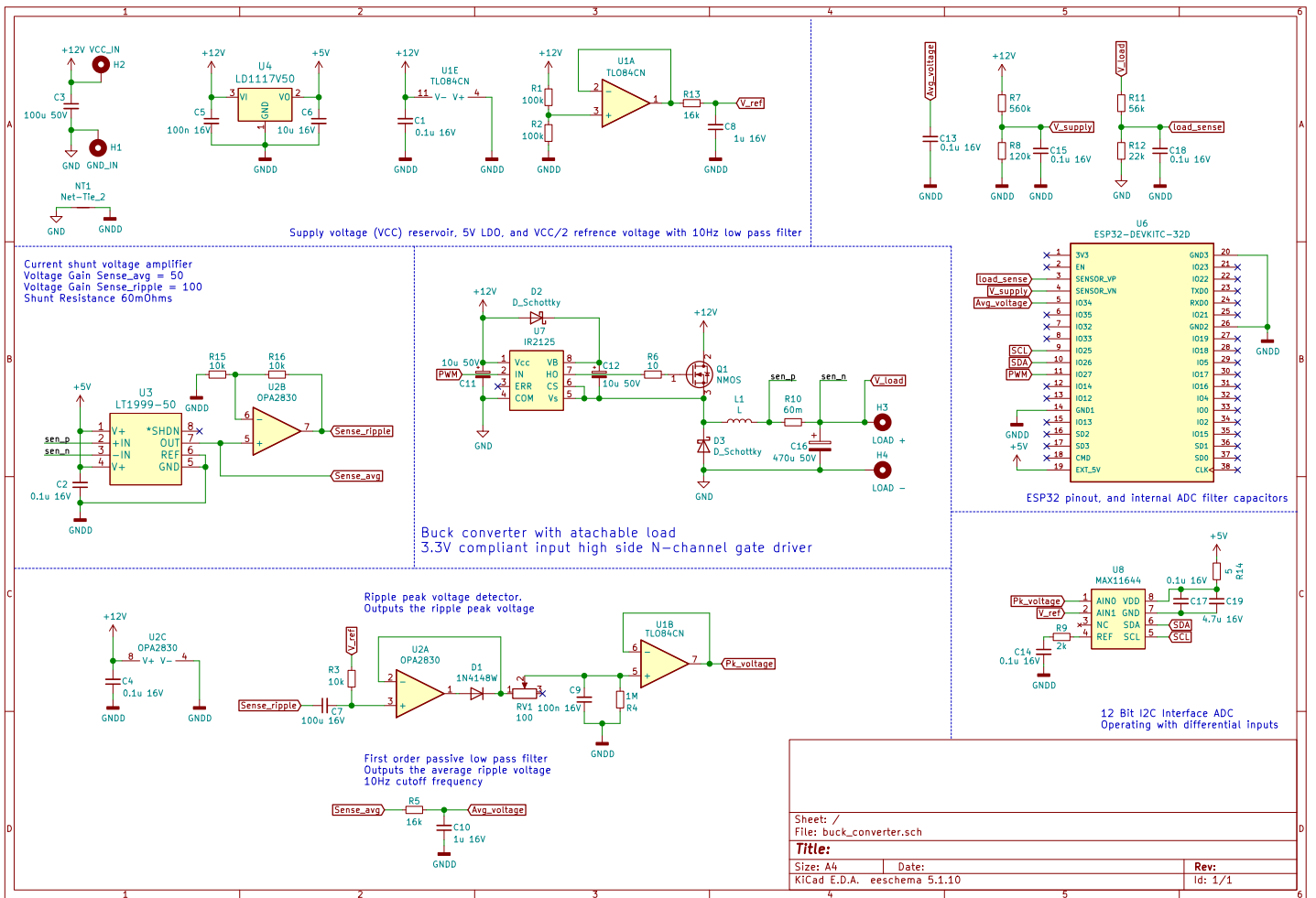


Figure E.1: Final system full schematic

Designed Printed Circuit Board

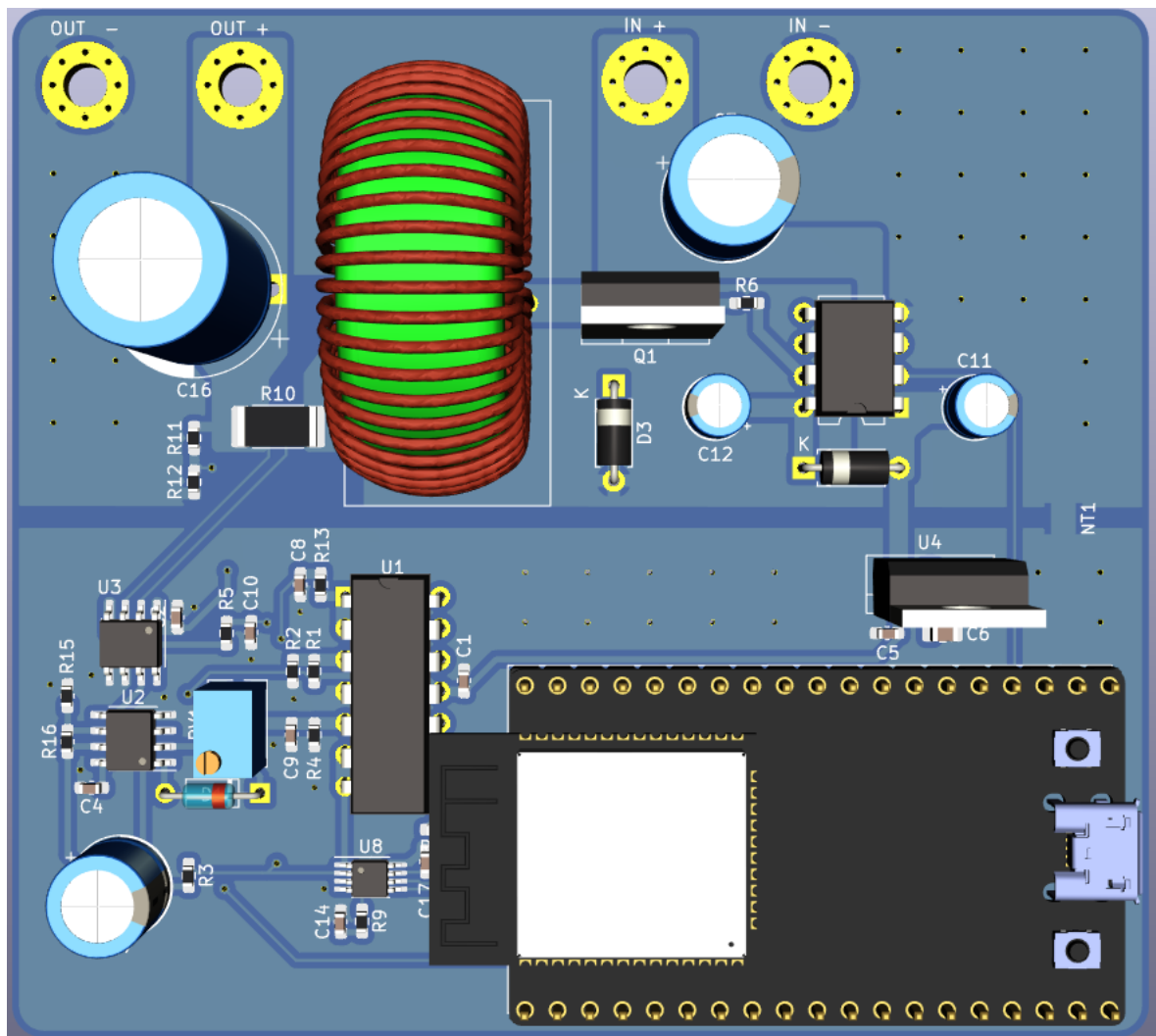


Figure E.2: Final system printed circuit board rendering

Appendix F: Source Code

Single Header Libraries: Hardware Drivers

adc.h

```
1 #ifndef BUCK_CONVERTER_ADC
2 #define BUCK_CONVERTER_ADC
3
4 #include <math.h>
5 #include "driver/adc.h" // Include the ADC driver
6
7 // ADC struct to hold setup data
8 typedef struct esp_adc
9 {
10     // Rolling average buffer
11     uint16_t *buffer;
12     uint8_t span;
13
14     // ADC channel to sample
15     uint8_t adc_channel;
16
17 } esp_adc;
18
19 #endif // BUCK_CONVERTER_ADC
20
21
22 #ifdef BUCK_CONVERTER_ADC_IMPL
23
24 /*
25  * Set up the adc to have a 12bit width, and 11dB attenuation
26  */
27 esp_err_t adc_init(esp_adc* adc){
28
29     // ADC set up
30     esp_err_t status; // Status variable to check if adc initialisation was
31     // successful
32
33     status = adc1_config_width(ADC_WIDTH_BIT_12); // Set adc width to 12 bits
34     if(status != ESP_OK) return status;
35
36     status = adc1_config_channel_atten(adc->adc_channel, ADC_ATTEN_11db); // Set
37     // adc attenuation to 11dB
38     return status;
39 }
```



```

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
66     // array
67     next_index++;
68     adc->buffer[adc->span] = (next_index >= adc->span) ? 0 : next_index;
69
70     // Calculate the rolling average
71     float total = 0;
72     for(int i = 0; i < adc->span; i++)
73     {
74         total = total + adc->buffer[i];
75     }
76
77     return total/adc->span;
78 }
79
80 /* Compute the conversion from the adc reading to a voltage.
81  *
82  * The output of the adc has been characterised and then a polynomial has been fit
83  * to the curve.
84  * This provides an error of < 1% for readings between 200mV and 3100mV
85  */
86 float IRAM_ATTR adc_conversion(float acd_reading)
87 {
88     return (pow(acd_reading , 4) * -7.6813211494455e-15) +
89            (pow(acd_reading , 3) * 5.03088719249885e-11) +
90            (pow(acd_reading , 2) * -1.06609443189713e-7) +
91            (0.00085850726668 * acd_reading) + 0.09077205072441;
92 }
93 #endif

```

code/adc.h

pid.h

```
1 #ifndef BUCK_CONVERTER_PID
2 #define BUCK_CONVERTER_PID
3
4 /*
5  * PID Controller code structure designed by https://github.com/pms67 under the MIT
6  * licence
7  * https://github.com/pms67/PID
8  */
9 typedef struct
10 {
11     /* Controller gains */
12     float Kp;
13     float Ki;
14     float Kd;
15
16     /* Derivative low-pass filter time constant */
17     float tau;
18
19     /* Output limits */
20     float limMin;
21     float limMax;
22
23     /* Integrator limits */
24     float limMinInt;
25     float limMaxInt;
26
27     /* Sample time (in seconds) */
28     float T;
29
30     /* Controller "memory" */
31     float integrator;
32     float prevError; /* Required for integrator */
33     float differentiator;
34     float prevMeasurement; /* Required for differentiator */
35
36     /* Controller output */
37     float out;
38 } PIDController;
39
40
41 #endif
42
43
44 #ifdef BUCK_CONVERTER_PID_IMPL
45
46 void PID_controller_init(PIDController *pid){
47
48     /* Clear controller variables */
49     pid->integrator = 0.0f;
50     pid->prevError = 0.0f;
51
52     pid->differentiator = 0.0f;
53     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
82         +(2.0f * pid->tau - pid->T) * pid->differentiator) /
83         (2.0f * pid->tau + pid->T);
84
85     /*
86     * Compute output and apply limits
87     */
88     pid->out = proportional + pid->integrator + pid->differentiator;
89
90     if (pid->out > pid->limMax){
91         pid->out = pid->limMax;
92     }
93
94     else if (pid->out < pid->limMin){
95         pid->out = pid->limMin;
96     }
97
98     /* Store error and measurement for later use */
99     pid->prevError = error;
100    pid->prevMeasurement = measurement;
101 }
102 #endif //BUCK_CONVERTER_PID

```

code/pid.h

pwm.h

```
1 #ifndef BUCK_CONVERTER_PWM
2 #define BUCK_CONVERTER_PWM
3
4 // Hardware driver includes
5 #include "driver/ledc.h"
6
7 /*
8  * PWM set up defines and helper macros
9  */
10 #define DUTY_RESOLUTION LEDC_TIMER_9_BIT
11 #define FREQUENCY_MIN 1000
12 #define FREQUENCY_MAX 100000
13 #define DUTY_STEPS 512
14 #define DUTY(X) ((1.0 - X) * DUTY_STEPS) // Duty cycle calculation macro
15
16 #endif
17
18
19 #ifdef BUCK_CONVERTER_PWM_IMPL
20
21 /*
22  * Set up the PWM timer and channel structs, and configure the PWM output
23  */
24
25 void PWM_setup(ledc_timer_config_t *pwm_timer, ledc_channel_config_t *pwm_channel,
26               uint32_t frequency, float duty_cycle){
27
28     // Select and set up the PWM clock source and initial frequency
29     ledc_timer_config_t timer = {
30         .duty_resolution = DUTY_RESOLUTION, // resolution of PWM duty
31         .freq_hz = frequency, // frequency of PWM signal
32         .speed_mode = LEDC_HIGH_SPEED_MODE, // timer mode
33         .timer_num = LEDC_TIMER_0, // timer index
34         .clk_cfg = LEDC_AUTO_CLK, // Auto select the source clock
35     };
36
37     // Select and set up the PWM output channel and initial duty cycle
38     ledc_channel_config_t channel = {
39         .channel = LEDC_CHANNEL_0, // PWM output channel (0 - 4)
40         .duty = DUTY(duty_cycle), // PWM duty cycle
41         .gpio_num = 23, // PWM output GPIO
42         .speed_mode = LEDC_HIGH_SPEED_MODE, // PWM output mode
43         .timer_sel = LEDC_TIMER_0 // PWM timer index
44     };
45
46     *pwm_timer = timer;
47     *pwm_channel = channel;
48
49     ledc_timer_config(pwm_timer);
50     ledc_channel_config(pwm_channel);
51 }
52
53 /*
54  * Set a new PWM duty cycle for a given PWM channel
55  */
56 }
```

```

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,
60      DUTY(duty_cycle));
61
62      // Update the PWM channel with the new configuration
63      return ledc_update_duty(pwm_channel -> speed_mode, pwm_channel -> channel);
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
71  *pwm_timer, uint32_t frequency){
72
73      if(frequency > FREQUENCY_MAX || frequency < FREQUENCY_MIN)
74      {
75          printf("Specified input frequency not within possible range: %d - %d\n",
76          FREQUENCY_MIN, FREQUENCY_MAX);
77          return ESP_ERR_INVALID_ARG;
78      }
79
80      // Set the new PWM frequency of the given channel and timer configuration
81      return ledc_set_freq(pwm_channel -> speed_mode, pwm_timer -> timer_num,
82      frequency);
83  }
84
85  #endif // BUCK_CONVERTER_PWM

```

code/pwm.h

Application Code

buck_converter.h

```
1  #ifndef BUCK_CONVERTER
2  #define BUCK_CONVERTER
3
4  // Project includes
5  #include "pwm.h"
6  #include "adc.h"
7  #include "pid.h"
8
9  // Standard includes for math and print functions
10 #include <stdio.h>
11 #include <math.h>
12 #include <string.h>
13
14 // RTOS includes for error messages and task handling
15 #include "freertos/FreeRTOS.h"
16 #include "freertos/task.h"
17 #include "freertos/queue.h"
18 #include "esp_err.h"
19
20 void init_buck();
21 void control_loop(void *pvParameters);
22
23 /*
24  * PWM timer and channel structs
25  */
26 ledc_timer_config_t pwm_timer;    // Create the PWM timer struct
27 ledc_channel_config_t pwm_channel; // Create the PWM channel struct
28
29 /*
30  * Constants for ADC conversion
31  */
32 static const float LOAD_R1 = 56237.0f; // Voltage divider values from buck output
33                                     load
34 static const float LOAD_R2 = 22035.0f + 995.2f;
35
36 static const float SUPPLY_R1 = 565300.0f; // Voltage divider values from buck
37                                     output load
38 static const float SUPPLY_R2 = 119700.0f;
39
40 // Internal esp ADC structs
41 esp_adc v_supply;
42
43 /*
44  * Constants, functions, and variables for PID controller
45  */
46 static const float V0 = 3.0f; // The desired initial output of the converter.
47
48 // Controller gains
49 static const float KP = 0.16f;
50 static const float KI = 17.1f;
51 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

buck_converter.c

```
1 // Include single header library function implementations
2 #define BUCK_CONVERTER_PWM_IMPL
3 #define BUCK_CONVERTER_ADC_IMPL
4 #define BUCK_CONVERTER_PID_IMPL
5
6 #include "buck_converter.h"
7
8 void init_buck()
9 {
10     // PWM set up
11     PWM_setup(&pwm_timer, &pwm_channel, 50000, 0);
12 }
13
14 /*
15  * Output voltage PID control loop.
16  * This function initialises the PID controller, and the internal ADC on the ESP32.
17  * It will then enter the main control loop, and will compute the following:
18  *
19  *     - Read the current buck converter load voltage. This is Measured using the
20  *       ESP32 ADC, through the voltage divider defined by LOAD_R1 and LOAD_R2.
21  *
22  *     - Calculate the equivalent duty cycle that would be used to produce the
23  *       current measured load voltage.
24  *
25  *     - Pass the target duty cycle, and current duty cycle through the PID
26  *       controller
27  *
28  *     - Update the buck converter duty cycle with the new controller output
29  *
30  *     - Delay the task until the selected sample period defined by TS has elapsed
31  */
32 void control_loop(void *pvParameters)
33 {
34     // Setup an input queue for receiving new target voltages
35     QueueHandle_t target_voltage_queue = *(QueueHandle_t *)pvParameters;
36
37     // Load voltage ADC set up and local variables
38     uint16_t supply_voltage_buffer[5] = {0}; // Supply voltage rolling average
39     buffer
40
41     // Variables for storage and conversion of load voltage
42     uint16_t load_adc_raw;
43     float load_adc_voltage;
44     float load_voltage;
45     float measurment_duty;
46
47     // Variables for storage and conversion of
48     uint16_t supply_adc_raw;
49     float supply_adc_voltage;
50     float supply_voltage;
51
52     // Initialise the ADC structs
53     esp_adc v_load = { // Create adc struct for reading the load voltage
```



```

53         // adc input channel 0 (GPIO 36)
54         .adc_channel = 0,
55
56         // do not allow for a rolling average
57         .span = 0,
58         .buffer = NULL};
59
60     esp_adc v_supply = { // Create adc struct for reading the supply voltage
61         // adc input channel 6 (GPIO 34)
62         .adc_channel = 6,
63
64         // 5 wide rolling average
65         .span = 5,
66         .buffer = (uint16_t *)&supply_voltage_buffer};
67
68     // Init the load voltage adc and check to see if it was successful
69     if (adc_init(&v_load) != ESP_OK)
70     {
71         printf("Load voltage ADC init error!\n"); // If set up fails print the error
72         vTaskDelete(NULL);
73     }
74
75     // Init the supply voltage adc and check to see if it was successful
76     if (adc_init(&v_supply) != ESP_OK)
77     {
78         printf("Supply voltage ADC init error!\n"); // If set up fails print the
79         error
80         vTaskDelete(NULL);
81     }
82
83     //PID set up
84     float target_voltage = V0;
85     float target_duty;
86
87     // Create the PID struct
88     PIDController voltage_controller = {
89         // Controller gains
90         .Kp = KP,
91         .Ki = KI,
92         .Kd = KD,
93
94         // Differentiator low pass filter corner frequency
95         .tau = 0.0f,
96
97         // Min and max output duty cycles
98         .limMin = 0.0f, // 0% duty cycle
99         .limMax = 0.98f, // 95% duty cycle
100
101         // Integrator wind-up min and max
102         .limMinInt = -5.0f,
103         .limMaxInt = 5.0f,
104
105         // Sample time period
106         .T = TS / 100.0f,
107     };

```

```

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

main.c

```
1 #include "buck_converter.h"
2
3 void app_main()
4 {
5
6     init_buck();
7
8     // Declare task handles
9     TaskHandle_t VO_controller = NULL;
10
11     QueueHandle_t target_voltage_queue;
12
13     // initialise the queues
14     target_voltage_queue = xQueueCreate(5, sizeof(float));
15
16     // Create the output voltage control task
17     xTaskCreatePinnedToCore(control_loop,                // Voltage control loop
18                             function
19                             "Vout_controller",           // Task name
20                             2048,                        // Task stack size
21                             (void *)&target_voltage_queue, // Function parameters
22                             2,                           // Priority of the task
23                             (app_main has priority 1)
24                             &VO_controller,             // Task handle
25                             tskNO_AFFINITY);            // Core the task has
26     been pinned to (No core selected)
27
28     float input_voltage;
29     char input_buffer[20] = {0};
30     uint8_t buf_index = 0;
31
32     while (true)
33     {
34         char input = fgetc(stdin);
35
36         if (input != 0xFF)
37         {
38             input_buffer[buf_index++] = input; // read in the character, and
39             increment the index
40             fputc(input, stdout);              //echo the character back
41
42             if ((input_buffer[buf_index - 1] == '\n'))
43             {
44                 input_voltage = strtod(input_buffer, NULL);
45
46                 xQueueSend(target_voltage_queue, (void *)&input_voltage,
47                             (TickType_t)1);
48
49                 printf("\nInput Target Voltage: %f\n", strtod(input_buffer, NULL));
50                 buf_index = 0;
51             }
52         }
53     }
54 }
```

```
50         vTaskDelay(1);
51
52         // printf("task stack unused: %d\n\n",
uxTaskGetStackHighWaterMark(V0_controller));
53         // vTaskDelay(100);
54     }
55 }
```

code/main.c