Power Electronic Feedback Control of a DC-DC Converter Using an Arduino Uno

Mr. Junhyung Park, United States Air Force Academy

Mr. Park is a third-year Electrical and Computer Engineering student at the United States Air Force Academy. Mr. Park has devoted his interests to satellites, rovers, rocketry, optics technology, robotics, and the Internet of Things with USAFA Blue Horizon Rocketry Club. Mr. Park has research experience in developing a LoRaWAN transmission system for the weather balloon that shares the flight data with the ground station through the Internet of Things network. He has also developed DC-DC buck converter power feedback control infrastructure with Arduino Uno and tested the capabilities of a hyperspectral camera for identifying small ordnances on a runway as a part of the United States Air Force's Rapid Airfield Damage Assessment System (RADAS). As a part of USAFA SPARK Innovation Tank, Mr. Park is also a problem solver with experience in pitching ideas at Falcon Tank and Junction Asia Hackathon. Mr. Park hopes for the world that he will change with his ideas one day.

John Ciezki, United States Air Force Academy

Power Electronic Feedback Control of a DC-DC Converter Using an Arduino Uno

Junhyung Park
Electrical and Computer Engineering
United States Air Force Academy
USAF Academy, United States
c25junhyung.park@afacademy.af.edu

Dr. John Ciezki
Electrical and Computer Engineering
United States Air Force Academy
USAF Academy, United States
john.ciezki@afacademy.af.edu

Abstract

Rapid prototyping systems to implement feedback controls are expensive. Other alternatives such as analog controllers that include waveform generators complicate tuning by requiring components to be physically swapped. Thus, this research explores the limits of configuring a commercially available digital microcontroller for doing power electronics research. The authors used an Arduino Uno to control the output voltage of a DC-DC converter. The setup of the Arduino Uno and the experiments that were performed are documented. The range of switching frequencies and sampling frequencies that could be used to successfully satisfy the control requirement are presented.

Index Terms

Arduino, PI feedback control, PWM signal, Power, DC-DC Buck Chopper

I introduction

In the academic environment, the development of power electronics systems is frequently slowed by the design and implementation of feedback controls. There are several attractive hardware-in-the-loop rapid prototyping systems, such as dSPACE or Speedgoat, but these systems are expensive on the order of thousands of dollars and impractical for a system that must be deployed or compact. Alternatives include analog controllers which employ some combination of PWM (Pulse Width Modulation) integrated circuit chip or custom waveform generator and comparator, and passive components wrapped around op-amps. This is a productive venture, but it generally takes significant time to fabricate and is generally more difficult to quickly tune its operation. Digital control is typically more compact and software adjustable, but it requires greater confidence in coding and experience with embedded systems generally. In contrast, Arduino Uno is a hobbyist-style microcontroller that can support switching frequencies up to 100 kHz with abilities to perform PWM, ADC, and interrupts, making Arduino Uno a suitable platform to teach embedded systems, digital control, and power electronics [1]. This research explores further limits of configuring and using an Arduino Uno for doing power electronics research. The ultimate goal is to assemble a DC power electronics microgrid and study energy management issues. Central to this endeavor is developing a capable controller that students can deploy rapidly with some success.

II dc-dc buck chopper

A DC-DC buck chopper is a circuit that efficiently steps down a DC voltage to a new level. The basic principles discussed apply to other types of PWM converters. The buck chopper circuit is illustrated in Fig. 1. The circuit is controlled by cyclically opening and closing switch S. This periodic rate is called the switching frequency and drives the sizing of the circuit inductance and capacitance. When the PWM signal is high, the switch S is closed. When the PWM signal is low, switch S is open and diode D conducts. The amount of time that S is closed divided by the switching period is called the duty cycle. For an ideal buck chopper operating in continuous conduction mode (inductor current always positive), the output voltage is approximately equal to the duty cycle times the input voltage. In our experiment, the PWM signal is created by a microcontroller.

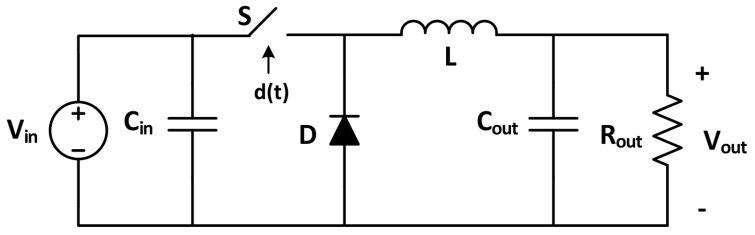

Fig. 1. Buck Chopper Circuit Topology

III embedded solutions

An embedded solution can be created using any number of microprocessors, microcontrollers, single-board computers, FPGAs, or DSP boards. Some of the more common solutions seen at the undergraduate level include the Raspberry Pi single-board computer, the BeagleBone Black microcontroller, or an Arduino microcontroller. We've found that a Raspberry Pi requires a bit more familiarity with operating systems and languages like LINUX and Python. Our students were introduced to the Arduino Uno in an early course and this effort sought to build on that native familiarity. Other Arduino boards are listed in Fig. 2. The Uno is a minimalist hobby platform. It operates at 16MHz with 14 digital Inputs/Outputs (I/O), 6 analog inputs, 6 pulse-width-modulation outputs, and no on-board Digital-to-Analog Converters (DAC). For projects with greater software or hardware requirements, boards like Mega, Due, or Yun might be considered.

Board	Clock (MHz)	Dig I/O	Ana IN	Ana OUT	PWM	Flash (KB)	SRAM (KB)	EE PROM (KB)	Proc Volt (V)	Interr Pins
UNO	16	14	6	0	6	32	2	1	5	2
Nano	16	14	8	0	6	32	2	1	5	2
Mini	16	14	8	0	6	32	2	1	5	2
Leonardo	16	20	12	0	7	32	2.5	1	5	5
Micro	16	20	12	0	7	32	2.5	1	5	5
Mega	16	54	16	0	15	256	8	4	5	6
Due	84	54	12	2	12	512	96	x	3.3	54
Yun	400	20	12	0	7	32	2.5	1	3.3	5

Fig. 2. Arduino Board Comparison

IV architecture

The flow diagram and architecture for this investigation are each shown in Fig. 3 and Fig. 4. Note that the output voltage of the buck chopper is first divided down with resistors R_1 and R_2 so there is no possibility of exceeding the voltage limits of the Arduino Uno ADC. The ADC is asked to sample this scaled output voltage at a rate called f_{sample} [2]. This voltage value is then passed to the PI-control algorithm which will update the value of the duty cycle. This updated duty cycle is then used to create a PWM signal at the desired duty frequency and required switching frequency $f_{switching}$. The buck chopper is realized using a commercially-available Taraz board that includes the input capacitance, main switch S and diode D, and the gate driver electronics needed to convert the PWM control signal to appropriate turn-on and turn-off signals. Normally, the switching frequency and sample frequency would be the same. However, here we are choosing to switch at a higher frequency so that the inductor and output capacitor can be more reasonably sized (at lower switching frequencies, the size of the inductance and capacitance gets prohibitive). At this higher switching frequency of 10-50kHz, the Arduino Uno is not able to complete the PI-control algorithm calculations. This is a fundamental limitation of the Uno. So, we selected a more modest sample frequency of 1kHz which will in turn limit the control response speed of the feedback control system.

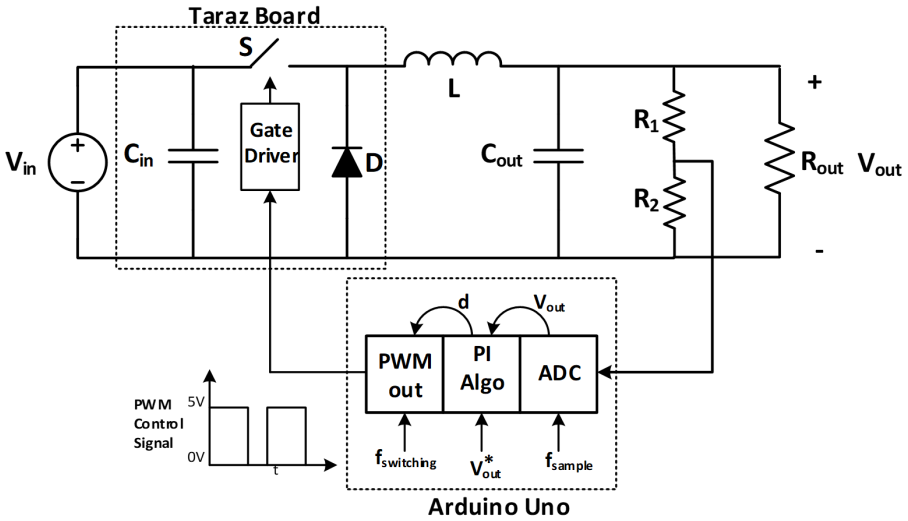

Fig. 3. Prototype Architecture

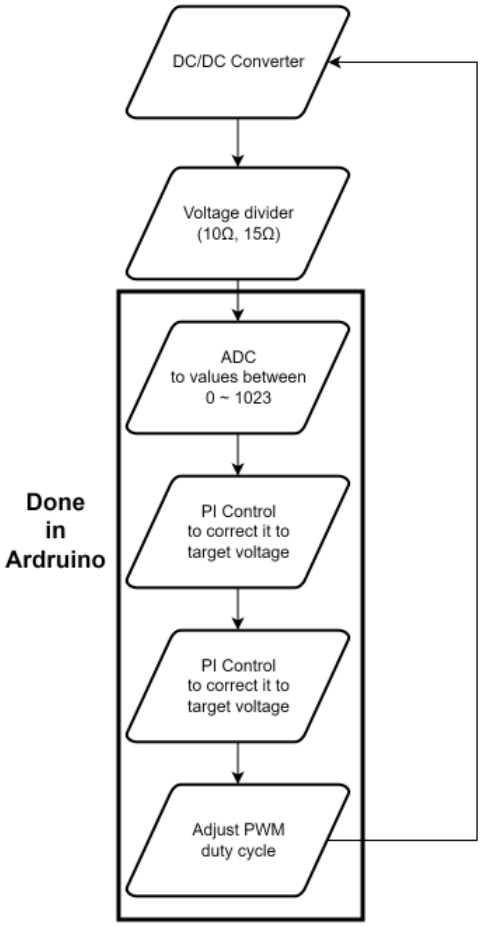

Fig. 4. Arduino Feedback Loop Flow Diagram

V prototype

The goal of this work is to explore the possibility and limit of an Arduino Uno to monitor the load voltage, execute the PI control algorithm, update the duty cycle of the PWM signal, and output the PWM signal. The components used to build the prototype according to schematics in Fig. 5 and Fig. 6 are as follows:

- Taraz GDA-2A2S1 DC-DC Buck Chopper x1
- DC Power Supply x2
- Inductor (220μH) x1
- Capacitor (470μF) x1
- Arduino Uno x1
- 150W BK Precision DC Electronic Load x1
- Multimeter x1
- Scope x2
- Resistor (15kΩ x1, 10kΩ x1, 33Ω x2, $\frac{1}{4}$W)

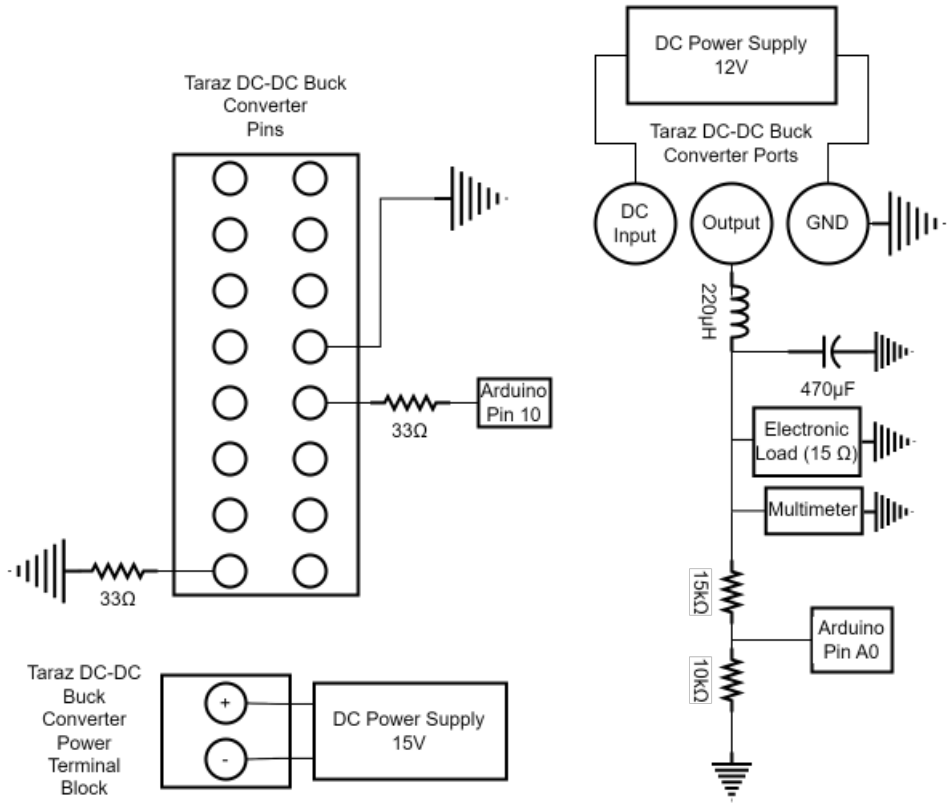

Fig. 5. Schematics of the Taraz DC-DC Buck Chopper

The test setup for our studies includes a GDA-2A2S1 Taraz power electronics board that conveniently implements a half-bridge switcher with gate drivers, input capacitor filter, and connection points. The board was configured so that only the top switch in the half-bridge was being gated. All that is required is for the input voltage and a PWM control signal to be connected. The output of the half-bridge is then connected to a breadboard containing an inductor and capacitor to realize the buck chopper topology shown in Fig. 1. The specification for this project was to step 12V down to 6V, and then have the controller regulate the 6V for changes in load resistance and for a step change in desired output voltage down to 4V.

The controllable electronic load for the DC-DC converter was a 150W BK Precision unit that could handle voltages up to 60V. A variable power supply was used to provide the 12V input and a second one was used to provide the 15V needed to power the Taraz board control circuitry. Digital multimeters and an oscilloscope were used to monitor various signals. The nominal load on the DC-DC converter was set to 15 Ω. At 6V out and 20kHz switching frequency, the converter remains in continuous conduction mode (CCM) up to 17.6 Ω. For a 4V output voltage, CCM is maintained up to 13.2 Ω at 20kHz.

An output voltage divider was specified to guarantee that even if the input 12V was passed through the DC-DC converter to the output, the divided voltage would not exceed the 5V maximum input to the Arduino Uno. The output voltage would be scaled as follows:

$$V_{Scaled} = V_{Output} \frac{10k\Omega}{(10+15)k\Omega} \tag{1}$$

The voltage across the 10 kΩ resistor is then input into the 10-bit ADC in Arduino Uno at pin A0. The ADC output value is an integer between 0 to 1023, with 1023 representing 5V and 0 as 0V.

The ADC value is then used to perform a PI calculation for the PWM duty cycle. An error from the desired voltage is first calculated. Then, the error value and the error value from the previous feedback loop are used to perform the PI control. The PI control coefficient values are calculated offline and verified in Matlab and then transferred over to the Arduino Uno code. The calculations for deriving the coefficient values are covered in the ”PI Controller” section later in the paper. The output of the PI control code would update the OCR1B value that controls the duty cycle of the PWM signal. Based on a different PWM frequency and the sampling frequency, the PI control coefficients would need to be updated in Matlab and transferred to the Arduino Uno code.

```
error = refVolt - voltage;
```

```
//PI control coefficients calculated from Matlab: OCR1B += error*ki + error*ki +
    error*kp - previouserror*kp
OCR1B += error*0.0633 + previousError*0.0633 + error*0.0407 - previousError
    *0.0407; //20kHz PWM, 1kHz sampling frequency

previousError = error;
```

In case the PI control corrects the OCR1B value beyond its bit limit, a safety control was also included to set OCR1B at 390 if the correction goes beyond or at 10 if the correction becomes too low or negative.

```
if(OCR1B > 390) {
    OCR1B = 390;
}

if(OCR1B < 10) {
    OCR1B = 10;
}
```

The code does not implement integrator anti-windup at this point. This feedback loop would continuously repeat at the sample frequency.

In the experiment, the desired voltage is first set to 6V.

```
int refVolt = 492; //6V

void setup() {
    OCR1A = 399;
    OCR1B = 0;
}
```

Then, after 10 seconds, as shown in the code, the desired voltage is changed to 4V to verify the Arduino's ability to correct the voltage. Since the desired voltage is changed, the feedback loop will make changes to OCR1B accordingly to meet the desired voltage.

```
if (millis() > 10000) {
  refVolt = 327; //4V
}
```

Based on the schematics, a prototype was built, and the voltage correction experiment conducted. The PWM frequency was set to 20kHz, and the sampling rate was initialized at 1kHz.

Measuring time duration in the Arduino Uno is normally done through the Serial command and system clock. However, Serial could not be used in this experiment due to the Interrupt already using ISR [4], which Serial requires also. Therefore, the digital pin 7 was set as an output to signal the beginning and end of each cycle of reading voltage and the PWM duty cycle adjustment.

Additionally, the trigger function of the oscilloscope was used to capture the commanded step change in output voltage from 6V to 4V. If measured continuously from the beginning, the range of the measurement would not specifically capture the voltage correction phase. To trigger the oscilloscope, the digital pin 13 was set as an output to trigger the oscilloscope measurement.

The signal in Fig. 6 indicates the reading of the digital pin 7 of Arduino. It is high at 1kHz interrupt frequency.

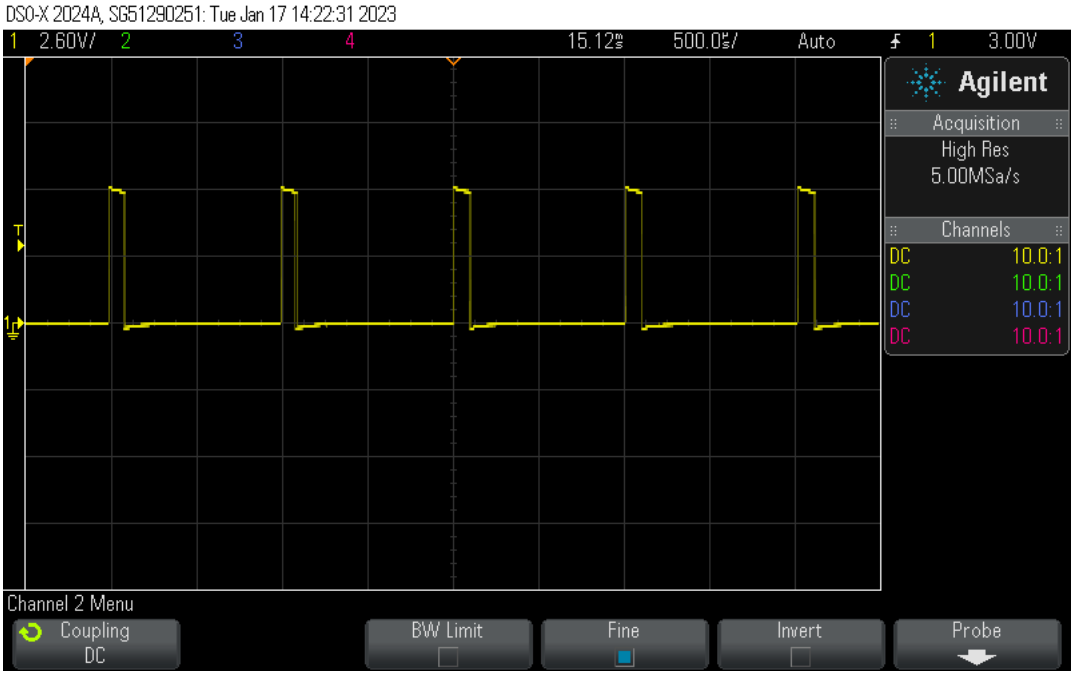

Fig. 6. Oscilloscope Measurement of the Digital Pin 7 for Measuring the Time Duration of Each Cycle

In Fig. 7, the yellow line on the oscilloscope indicates the digital pin 13 output from Arduino that triggered the measurement. The green line on the oscilloscope is the output voltage reading. As a result of the experiment, the voltage was regulated successfully from 6V to the step change of reference voltage of 4V. The control action took around 20ms before stabilizing to 4V.

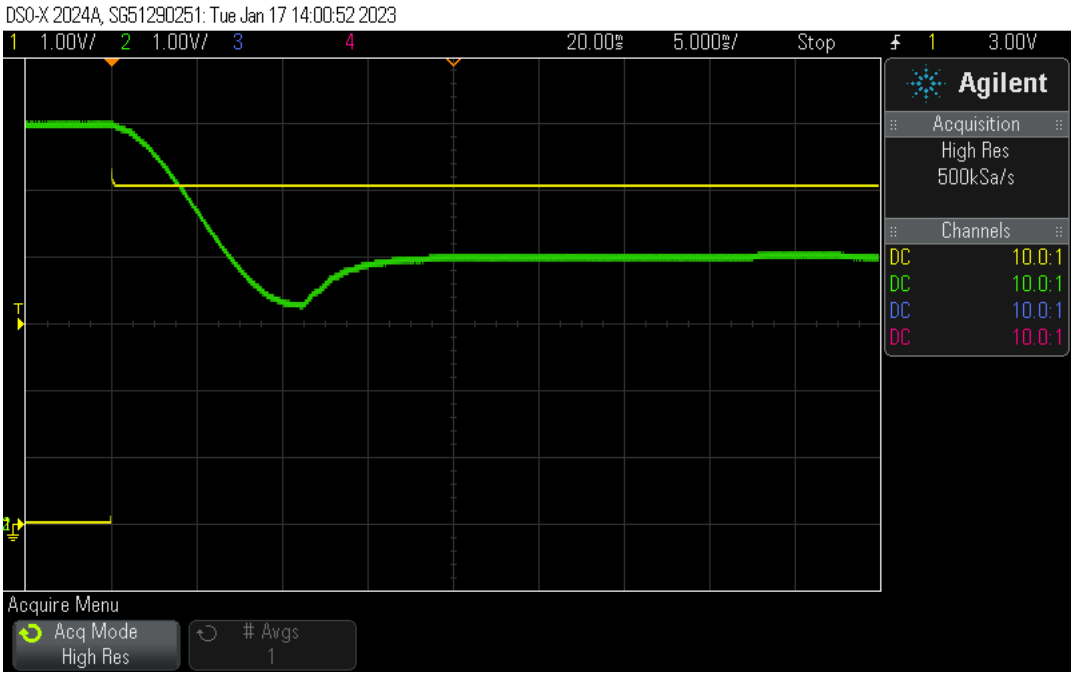

Fig. 7. Oscilloscope Measurement of the Prototype Voltage Correction Phase with 20kHz PWM and 1kHz Sampling Frequency

VI pi controller

Due to inevitable losses in the buck circuit elements and that these losses will vary with output load resistance, a feedback control loop is required to regulate the output voltage and provide a satisfactory transient response. It does this by dynamically

adjusting the duty cycle. Most control algorithms first create an error signal between the desired reference and the actual output voltages. In this study, we focused on a proportional-plus-integral (PI) controller which uses a weighted combination of proportional error and integrated error as shown in (2).

$$d(t) = K_p(V^*_{out} - V_{out}) + K_I \int (V^*_{out} - V_{out})dt \tag{2}$$

The integral component reduces steady-state error while the proportional component provides a means to influence stability and transient performance. The control synthesis process establishes the gains K_p and K_I.

A digital controller can be synthesized in two ways: (1) a continuous-time controller can be designed and then discretized or (3) the plant is first discretized and then a discrete-time compensator is synthesized using digital-domain techniques. We use the former as it uses concepts that every student sees in the ECE program. The process for a PI-controlled buck chopper is as follows.

First, the transfer function for the plant must be specified as shown in (2).

$$G_p(s) = \frac{K_{pwm}K_{sen}V_{in}(R_cC_s + 1)}{(\frac{R+R_C}{R})LCs^2 + [\frac{L}{R} + (R_C + R_L)C + \frac{R_cR_LC}{R}]s + (1 + \frac{R_L}{R})} \tag{3}$$

It includes the effects of capacitor ESR (R_C) and inductor series resistance (R_L). The transfer function has a left-half-plane (LHP) zero and two under-damped poles. The gains K_{pwm} and K_{sen} model the modulation gain and the output voltage divider. The output of the controller sets OCR1B which directly sets the duty cycle, with a value of 0 yielding d = 0 and 799 (for a frequency of 10kHz) yielding d = 1. Thus, the gains are as follows:

$$K_{pwm}Ksen = \frac{5}{12} * \frac{1024}{5} * \frac{1}{800} = 0.1067 \tag{4}$$

The open-loop resonance point is quite close to (5).

$$\omega_{res} = \frac{1}{\sqrt{LC}} \approx 3.1krad/s \tag{5}$$

We can then add the integrator from our PI control and plot the frequency response. We choose the gain crossover frequency to be one decade below the resonant frequency as follows:

$$\omega_{gc} = 0.1 * \omega_{res} \approx 310rad/s \tag{6}$$

This is to avoid requiring further loop shaping with additional poles and zeros. As a result, this allowed us to find the required gain that the controller must produce at 310rad/s. Finally, the zero of the PI-controller was placed at the resonant frequency so we would not disturb the gain crossover frequency while adding 45° of phase at ω_{res}. This approach gives both good phase margin and gain margin, though it will certainly not yield the fastest response.

Next, we discretized our controller transfer function using the Bilinear Transform. This moves us into the z-domain, where f_{samp} is the sampling frequency, E(z) is the error signal, and Y(z) is the output of the controller.

$$G_c(z) = \frac{Y(z)}{E(z)} = K_p + \frac{K_I}{\frac{2}{T_{samp}}\frac{z-1}{z+1}} = K_p + \frac{K_I(z+1)}{2f_{samp}(z-1)} \tag{7}$$

After cross-multiplying and converting into a difference equation, we get:

$$y(k) = y(k-1) + (K_p + \frac{K_I}{2f_{samp}})e(k) + (\frac{K_I}{2f_{samp}} - K_p)e(k-1) \tag{8}$$

Alternatively, we can regroup and express as:

$$y(k) = y(k-1) + K_p[e(k) - e(k-1)] + (\frac{K_I}{2f_{samp}})[e(k) + e(k-1)] \tag{9}$$

This algorithm requires us to store the previous value of the controller output, y(k-1), and the previous value of the voltage error, e(k-1). Including additional poles and zeros in the controller will increase the order of the required difference equation, adding more terms, and requiring more quantities to be stored. Future efforts will explore how this complexity impacts the allowable sampling frequency and the related speed of response.

The performance of the digital control is impacted by ADC quantization and the available duty cycle resolution. Both can result in unacceptable tracking and sustained oscillations as the duty cycle ping pongs between two discrete values. More advanced controllers provide a greater ability to shape the response at the cost of more analog components or more lines of software. For this research, the goal is to show that an inexpensive microcontroller, on the order of $40, can be used to realize an acceptable PI-control response for a buck chopper.

VII simulation

A simulation was performed in Matlab's Simulink as shown in Fig. 8. The prototype components were replicated as best as possible. Tustin's Transformation accomplishes the conversion of the continuous time compensator to the corresponding digital compensator [3]. The steps of the experiment explained in previous sections were also replicated by using Simulink's function blocks.

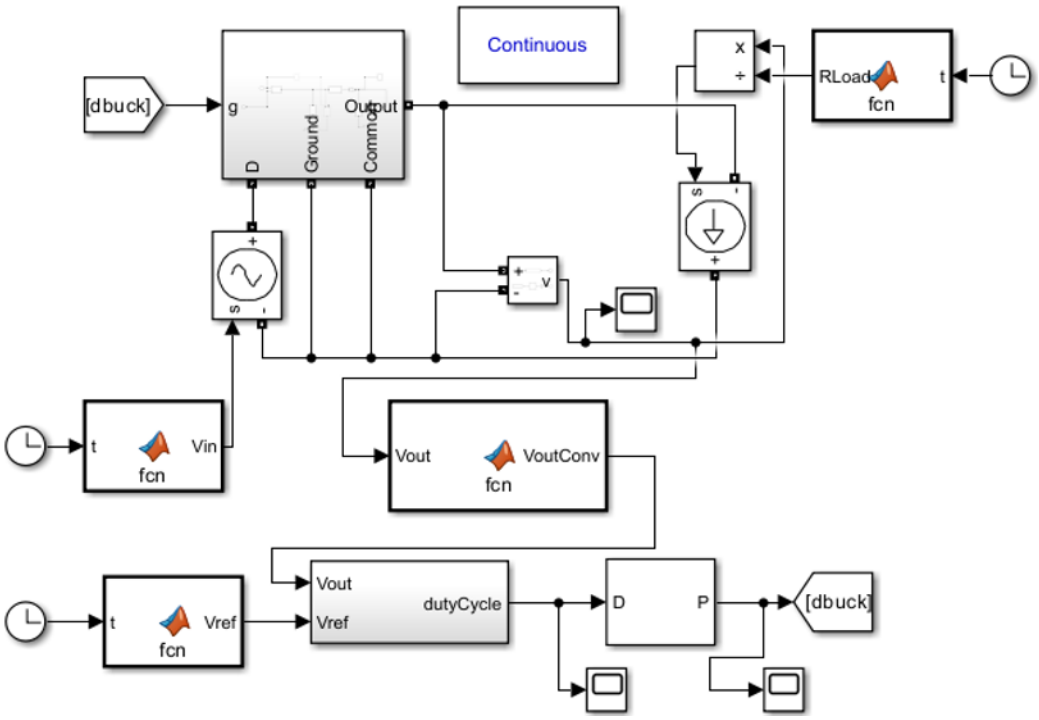

Fig. 8. Simulink Simulation Setup

According to the simulation result in Fig. 9, the time that the simulation took for correcting voltage from 6V to 4V with 20kHz and 1kHz sampling frequency was 35.284ms. Thus, the simulation showed a longer settling time than the 20ms measured in hardware. Differences might be accounted for by software delays in the Arduino that may not have been perfectly emulated in the simulation or by differences in how the Simulink block implements the PWM. Besides the time, the shape of the voltage correction matched the simulation and prototype. Overall, the prototype and simulation provided good correction. The voltage successfully outputs the user's set desired voltages stably using the PI-control and feedback system with the Arduino.

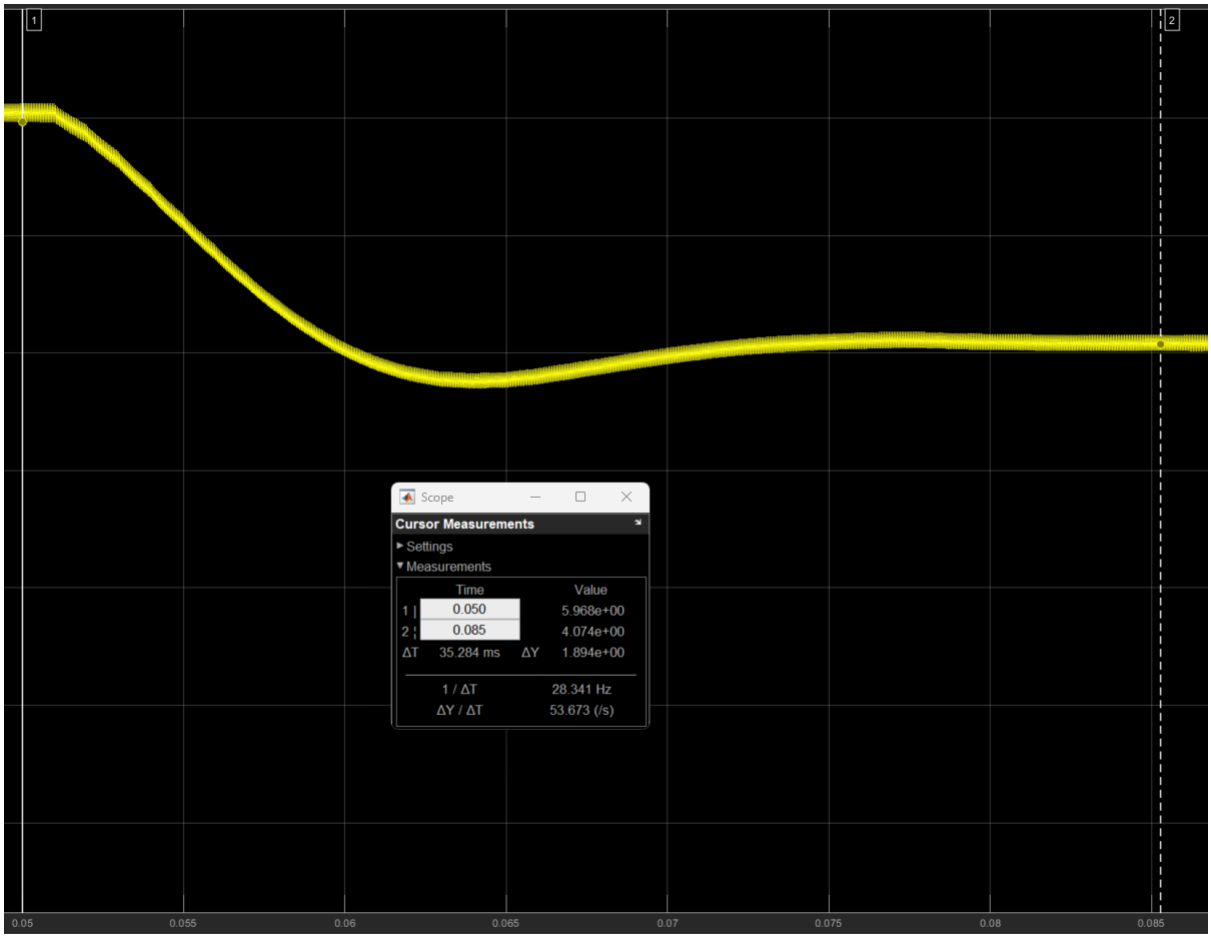

Fig. 9. Simulink Simulation of Voltage Correction with 20kHz PWM and 1kHz Sampling Frequency

VIII limitations

Due to the capabilities of the Arduino Uno, researchers did not typically think about using it for PWM control. The experimental test was repeated for the PWM frequencies and sampling frequencies listed in Fig. 10 to explore limitations in the approach.

PWM Freq	Sampling Freq	6V (V)	4V (V)	Settling Time
20kHz	1kHz	6.00∼6.01	4.00∼4.01	20ms
40kHz	1kHz	5.96∼5.97	3.93∼3.94	5ms
80kHz	1kHz	5.85∼5.86	3.95∼3.96	6ms
20kHz	2kHz	5.91∼5.92	3.98∼3.99	20ms
20kHz	4kHz	5.88∼5.89	3.87∼3.90	15ms

Fig. 10. Limitations Testing

As the experiment switching and sampling frequency values increased, the Arduino Uno was found to be inadequate to perform accurate voltage control due to the resolution issues. In all cases with the increased experiment values, the voltage output was under the set desired voltages (6V and 4V) although the time it took to reach the steady output voltage decreased. As the switching frequency increased, the oscillation in the output voltage was more pronounced as shown in Fig. 11. Additionally, the 6V to 4V transition was not smooth, unlike the test case with 20kHz PWM and 1kHz sampling frequency. Similar results are reflected in the Simulink simulation from Fig. 12. This phenomenon was anticipated from the duty cycle resolution issue that 80kHz PWM frequency creates.

$$OCR1A = 0 \sim 99$$

$$Resolution_{DutyCycle} = \frac{1}{(99-0)+1} = 1\% \tag{10}$$

Thus, the duty cycle cannot be expressed in a decimal form with 1% duty cycle resolution compared to 0.25% resolution from 20kHz PWM frequency with the same calculation process. Only a whole number percentage of the duty cycle can be expressed. As the PWM frequency increases, the inaccuracy of the duty cycle will be greater, which will cause more oscillates and inaccurate output voltages.

Fig. 11. Oscilloscope Measurement of the Prototype Voltage Correction Phase with 80kHz PWM and 1kHz Sampling Frequency

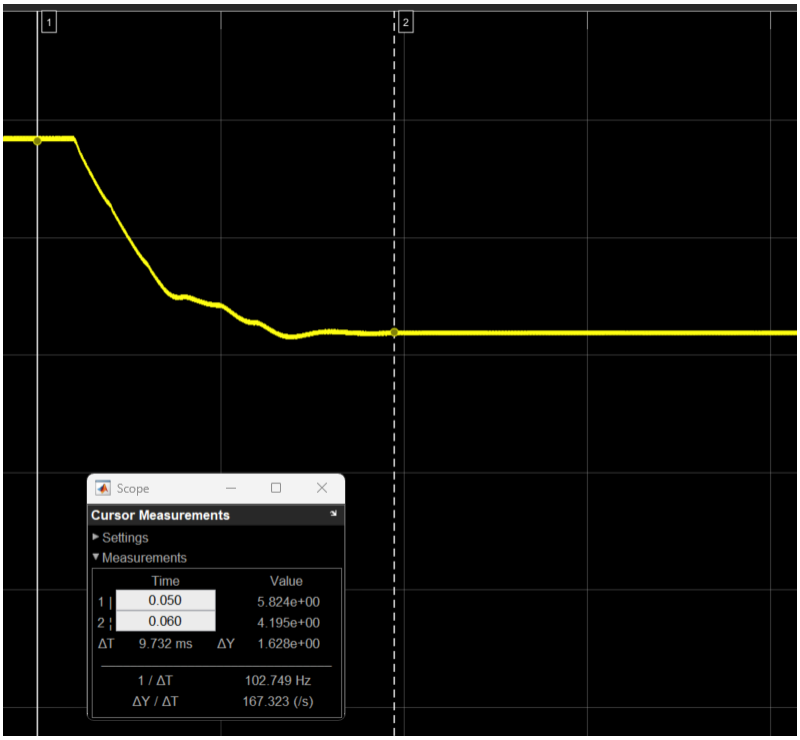

Fig. 12. Simulation of the Voltage Correction Phase with 80kHz PWM and 1ms Sampling Frequency

As the sampling frequency increased, the output voltage was significantly lower than the desired voltage as shown in Fig. 13. In the Simulink simulation shown in Fig. 14, the output voltage is also lower than the desired voltage. However, the transition was still smooth, and the output voltage was consistent without oscillations. According to Fig. 15, the time that the correction calculation took (around 100μs as indicated by the time the signal is high) was still less than the sampling time (250μs), so the possibility of the correction calculation not finishing before the next cycle starts is not an issue.

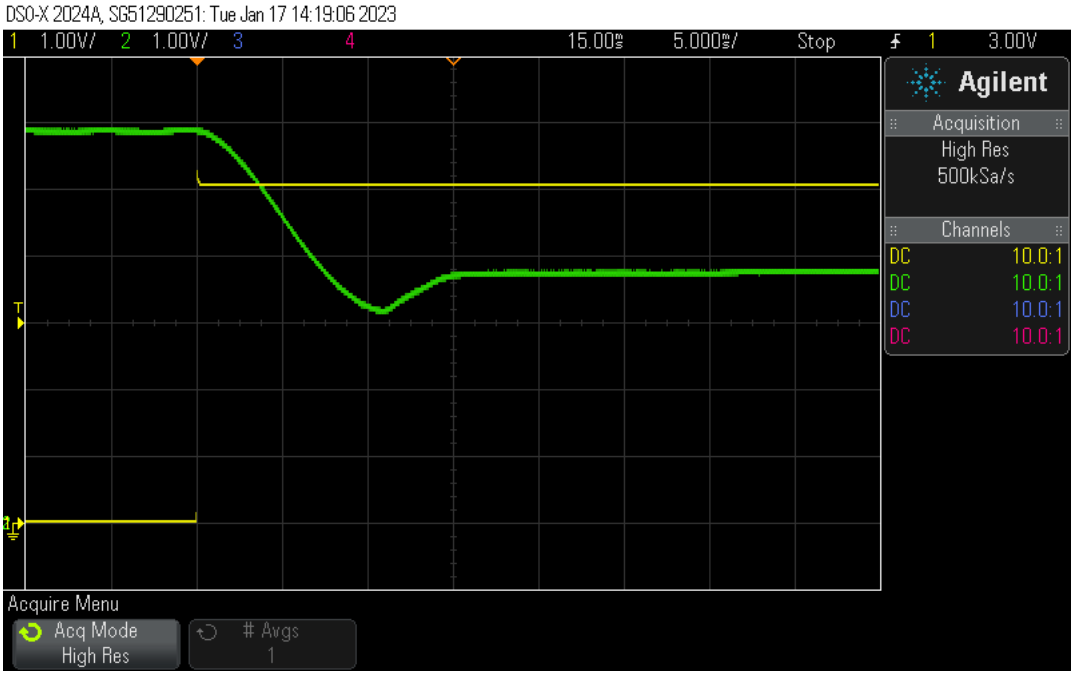

Fig. 13. Oscilloscope Measurement of the Prototype Voltage Correction Phase with 20kHz PWM and 4kHz Sampling Frequency

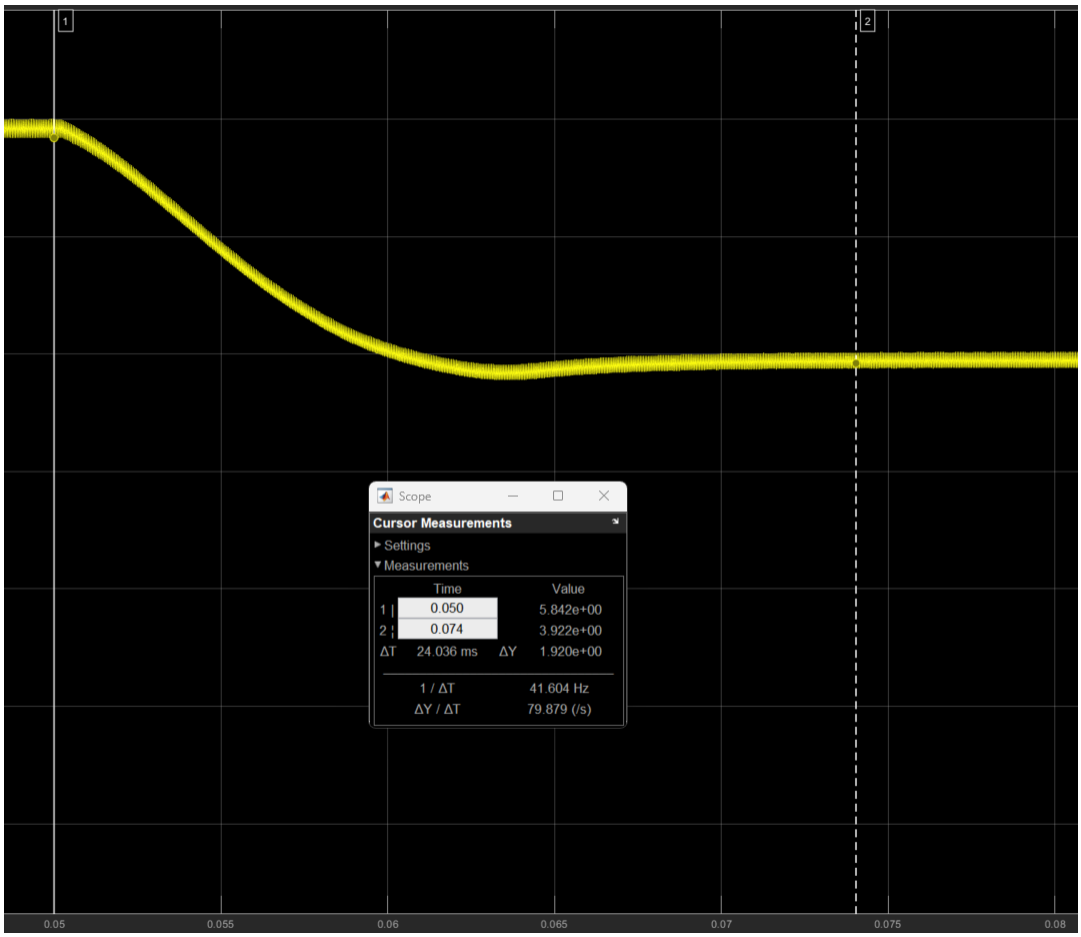

Fig. 14. Simulation of the Voltage Correction Phase with 20kHz PWM and 4kHz Sampling Frequency

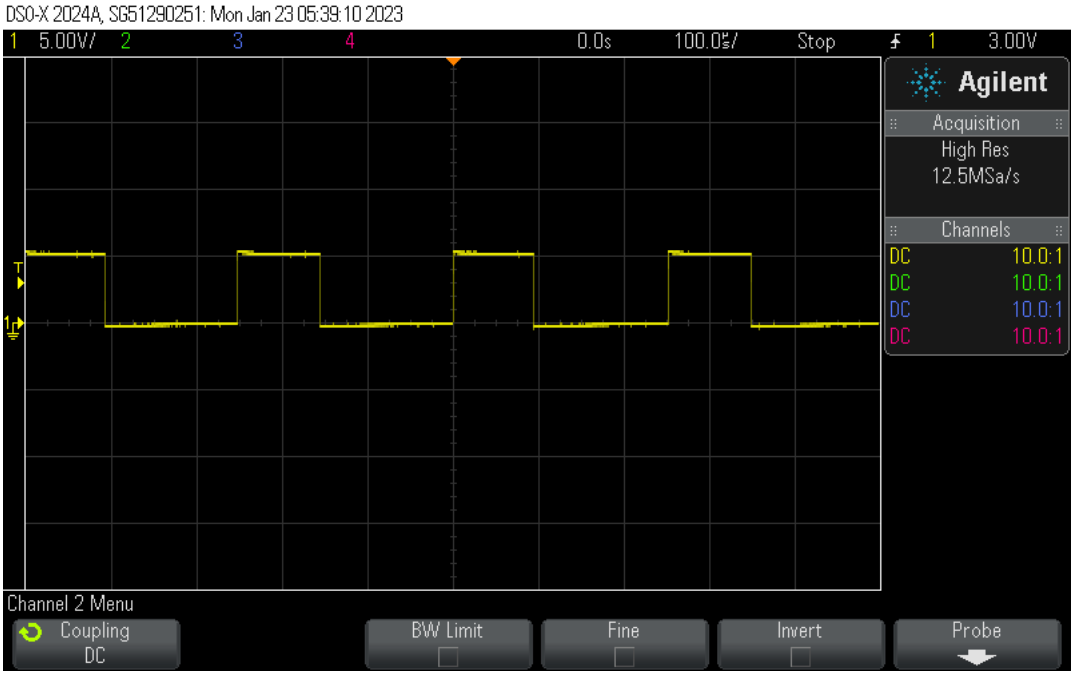

Fig. 15. Calculation Finishes Before the Next Cycle Begins

The most probable reason for the lower voltage than the desired voltage is that there is a delay of two to three cycles before a new PWM signal duty cycle is applied.

This phenomenon is examined in Fig. 16. Here, to investigate the delay, we created an open-loop test where we alternatively set a 50% or 87.5% duty cycle (green waveform). What we see is as the yellow signal toggles to cause the change, we must wait either two or three cycles for the change to take effect. As a result, a higher sampling rate causes more inaccurate output voltage because the accurate duty cycle correction was not applied in time. This is one of the limitations of the Arduino Uno, and the reason for this phenomenon is a topic that needs to be investigated further.

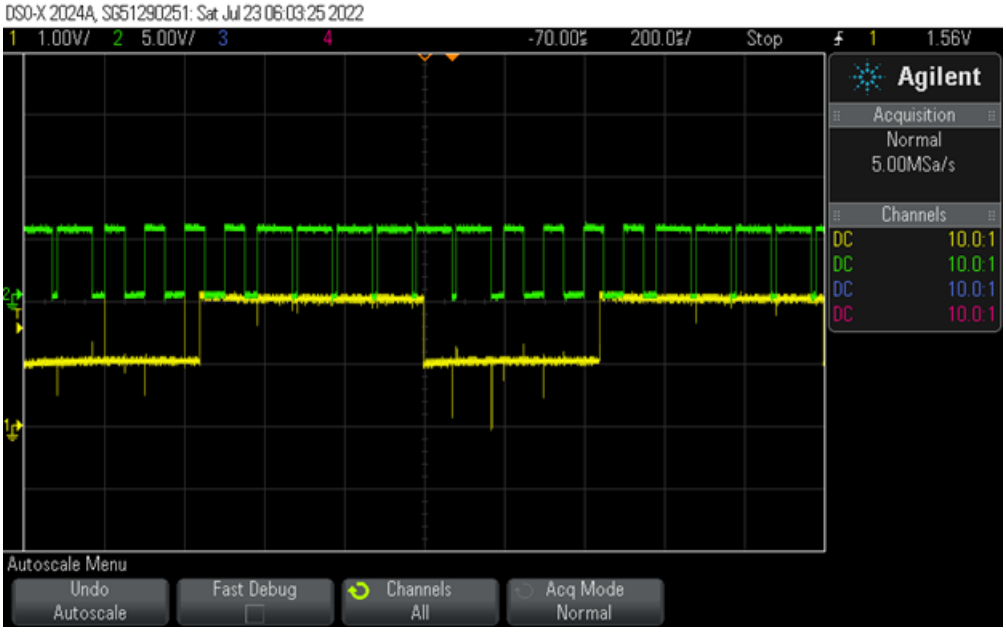

Fig. 16. Delays When PWM Signal Duty Cycle Changes

IX conclusion

We investigated the possibility and limit of using an Arduino Uno to perform output voltage regulation for a buck chopper circuit. We concluded that the architecture investigated successfully regulated the output voltage with a 20kHz PWM frequency setting and 1kHz sampling frequency and should work for lower values as well.

However, limitations existed when performing with higher PWM frequencies or sampling frequencies. If the switching application requires higher frequencies a different microcontroller platform or an analog solution must be considered.

The next step in the research might consider using the controllable DC-DC converter to regulate the charging and discharging of a battery. The architecture can be transformed into a current feedback architecture so that the current can be measured to find the charge in and out of the battery. Thus, one can track the battery charging and discharging processes.

An example of application from this study is the usage of the digital feedback control architecture in a large-scale energy storage system. In the age of renewable energy, a consistent and reliable supply of energy from the storage system is critical. Therefore, each storage unit and electronic component will need a digital control system. By monitoring and controlling the charging and discharging process of the DC energy system with this study's digital feedback control architecture, more accurate supply of the desired energy level is possible.

In conclusion, the power electronic feedback architecture introduced in this study is cheap and accessible for undergraduate-level power electronic lesson, but with limitations. Therefore, this architecture can be used as a practical example when teaching power electronics. It provides a real-world application of output voltage regulation for a buck chopper circuit using a relatively accessible and user-friendly Arduino Uno. However, the performance of the feedback control could be improved by using more powerful simulation and test solutions, such as Speedgoat or dSPACE, but with a cost premium. A power electronics lab could be envisioned that would explore adjusting PI gains both in simulation and in hardware. The performance with and without the integral control could also be examined to show the effect on steady-state error. Additional benefits in the classroom could be to also examine transient performance using the single-shot capability of the oscilloscope.

references

[1] L. Müller, M. Mohammed and J. W. Kimball, "Using the Arduino Uno to teach digital control of power electronics," 2015 IEEE 16th Workshop on Control and Modeling for Power Electronics (COMPEL), Vancouver, BC, Canada, 2015, pp. 1-8, doi: 10.1109/COMPEL.2015.7236487. [Accessed August 2022].

[2] "Fast sampling from analog input," February, 2015. [Online]. Available: http://yaab-arduino.blogspot.com/2015/02/fast-sampling-from-analog-input.html. [Accessed August 2022].

[3] M.G. Feemster, "A systematic approach for development and simulation of digital control algorithms using SIMULINK," 120th ASEE Annual Conference & Exposition, June, 2013. Available: https://monolith.asee.org/public/conferences/20/papers/700/view. [Accessed August 2022].

[4] "The do's and don'ts of using Arduino interrupts," March, 2022. [Online]. Available: https://www.digikey.com/en/maker/blogs/2022/the-dos-and-donts of-using-arduino-interrupts. [Accessed August 2022].

appendix

complete code

```
//PI control variables
int error = 0;
int previousError = 0;

//Voltage variables
int voltage;
int refVolt = 492;

boolean execute = false; //Variable that initiates the analog reading and control
    cycle

void setup() {
    analogReference(EXTERNAL); //sets the reference voltage to outside source(5V)

    pinMode(13, OUTPUT);    // sets the digital pin 13 as output
    pinMode(7, OUTPUT);
    pinMode(A0, INPUT); //Sets pin A0 to peform analog reading

    cli();//stop interrupts

    //PWM setup
    TCCR1A = (1<<COM1A1) + (1<<COM1B1) + (1<<WGM11) + (1<<WGM10);
    TCCR1B = (1<<WGM13) +  (1<<CS10); // prescaler = 1 (none)
    OCR1A = 399;
    OCR1B = 0;
    DDRB |= (1<<PB1)|(1<<PB2);


    //Timer2
    TCCR2A = 0;    // set entire TCCR1A register to 0
    TCCR2B = 0;    // set entire TCCR1A register to 0

    TIMSK2 = (1<<OCIE2A); // enable Timer2 time compare interrupt

    TCCR2A = 1<<WGM21; //Set CTC mode
    TCCR2B = (1<<CS22) | (0<<CS21) | (1<<CS20); // set 128 prescaler

    OCR2A = 125; //For 1kHz sampling frequency with prescaler 128

    sei();//allow interrupts
}

ISR (TIMER2_COMPA_vect) {
    PORTD = B10000000;  //Beginning of the loop pin 7
    OCR2A = B1111101; //125 1kHz sampling frequency with prescaler 128
    execute = true;
}

void loop() {
    while (execute) {
        if (millis() > 10000) {
          refVolt = 327;
```

```
        PORTB = B00100000; //Digital pin 13 HIGH
      }

      voltage = analogRead(A0);

      error = refVolt - voltage;
      //OCR1B += ki + ki + kp - kp
      OCR1B += error*0.0633 + previousError*0.0633 + error*0.0407 - previousError
          *0.0407; //20kHz, 1kHz sampling frequency

      if(OCR1B > 399) {
        OCR1B = 390;
      }

      if(OCR1B < 0) {
        OCR1B = 10;
      }

      previousError = error;

      PORTD = B00000000; //End of the cycle
      execute = false; //Get out of the loop until the next interrupt
    }
}
```

arduino

A. Structure

Our microcontroller, the Arduino Uno, must accomplish three tasks. It must sample the output voltage at a sufficient rate to avoid aliasing. Second, it must implement the control algorithm to update the duty cycle. Third, it must modify the PWM output to realize the updated duty cycle at the desired switching frequency. We can set the PWM frequency and duty cycle using bit-level operations. We can establish a sampling frequency using a software-programmed interrupt. We can investigate whether the sampling interval is long enough to execute our control algorithm. The following sections will provide the details on how to do this for an Arduino Uno. The goal might be that such "architecture" code could then be provided to the student to modify the DC-DC converter control algorithm without being hamstrung by the low-level programming code.

The structure of the Arduino code consists of three main stages: setup(), interrupt, and loop(). After importing necessary libraries and setting global variables, the setup() adjusts the environment settings for Arduino to perform certain tasks, such as ADC (Analog Digital Converter) reading, creating PWM signals, and initializing interrupts. When the setup() is complete, Arduino transitions to loop() which runs forever or until the Arduino is powered off.

Interrupts allow the software to jump from the main loop and execute a specific piece of time-critical code. For instance, for a control loop, one may want to exit the main code to periodically update the required PWM duty cycle. The interrupt may occur at a regular timed interval which we call the sampling frequency. The code in the Interrupt Service Routine (ISR) must be executed within the sampling period to maintain synchronization, so the tasks in the ISR must be short. In our experiment, we toggled a boolean variable to True in the ISR. This boolean variable triggers the control loop. After a cycle of the control loop, the boolean variable is toggled back to False and awaits until the next sampling period.

The Arduino Uno (ATMega328o microprocessor) has a default PWM frequency of about 980Hz on output pins 5 and 6 or 490Hz on pins 3, 9, 10, or 11. A PWM signal can be established by simply including the following command in the setup section of the code for a static PWM signal or within a loop if the signal duty cycle will be adjusted.

```
analogWrite( pin, dutyCycle )
```

where "pin" is one of the available PWM pins 3, 5, 6, 9, 10, or 11, and "dutyCycle" is an integer between 0 (0%) and 255 (100%).

The ATmega chip has three PWM timers, controlling the available 6 PWM outputs. By manipulating the chip's timer registers directly, one can obtain more control than the analogWrite function provides. The Timer/Counter Control Registers, TCCR1A and TCCR1B for Counter 1, hold the bits that set the mode, the prescaler that sets the frequency, and the bits that enable the output. The Output Compare Registers, OCR1A and OCR1B for Timer 1, hold the bits that set the levels when the counter

toggles its output and thus the duty cycle. OCR1A/B registers have 16 bits and OCR2A/B for Timer 2 have 8 bits. To have the Arduino Uno produce PWM frequencies much greater than the default value of 980Hz and dynamically adjust the duty cycle at a reasonable update rate, these internal registers must be configured. This requires consultation with Arduino data sheets and knowledge of making bit-level changes. This paper will document how that needs to be done and what are the limitations. The goal would be that students would have this part of the code provided to them, so they can instead focus on realizing and testing the control algorithm.

In the efforts documented here, the Arduino Uno produced PWM signals with Phase Correct mode. In Phase Correct mode, there is a counter (TCNT1) that counts up to the maximum count, set by OCRnA, where n is equal to one for counter 1 and two for counter 2. The maximum count controls the frequency of the PWM signal. After the counter reaches the maximum count, it counts downwards back to zero as shown in Fig. 17. This signal is output on OC1A (Arduino Uno pin 9).

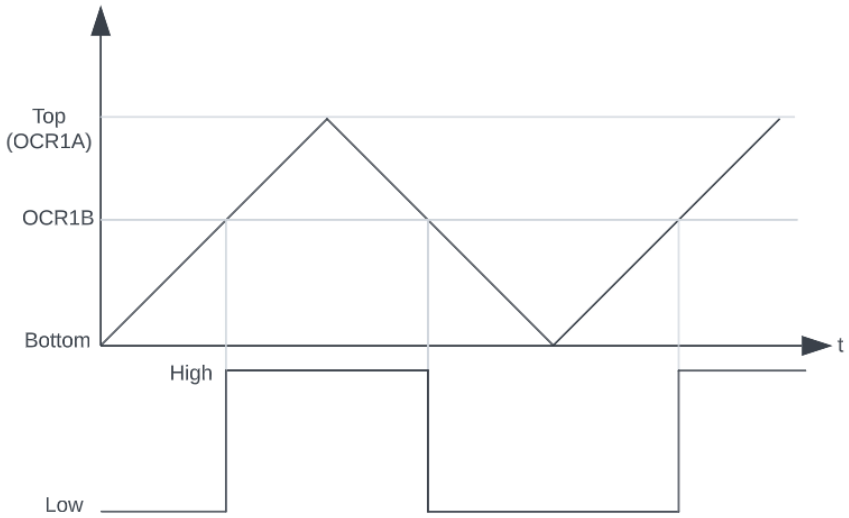

Fig. 17. Phase Correct Mode

Adjusting OCR1B changes the duty cycle by comparing the signal output from OC1A. This work used Compare Match mode, as specified in Fig. 21 row 4. OC1B (Arduino pin 10) is set high when OC1A crosses the OCR1B value while up-counting and set low when OC1A crosses the OCR1B value while down-counting, as shown in Fig. 17. The PWM signal is output on OC1B. Besides this mode, different modes of PWM other than Phase Correct mode can be set using registers TCCRnA and TCCRnB, referenced in Fig. 18, 19, and 20.

Bit No	7	6	5	4	3	2	1	0	
Identifier	**COM1A1**	**COM1A0**	**COM1B1**	**COM1B0**	**-**	**-**	**WGM11**	**WGM10**	**TCCR1A**
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	

Fig. 18. TCCRnA allocations

Bit No	7	6	5	4	3	2	1	0	
Identifier	**ICNC1**	**ICES1**	**-**	**WGM13**	**WGM12**	**CS12**	**CS11**	**CS10**	**TCCR1B**
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	

Fig. 19. TCCRnB allocations

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Time/ Counter Mode of Operation	TOP	Update of OCR1x at	TOV Flag Set On
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0xFFFF	Immediate	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x00FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x01FF	TOP	BOTTOM
4	0	1	0	0	CTC	0x03FF	TOP	MAX
5	0	1	0	1	Fast PWM, 8-bit	OCR1A	Immediate	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x00FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x01FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Freq Correct	0x03FF	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Freq Correct	ICR1	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	OCR1A	BOTTOM	BOTTOM
11	1	0	1	1	PWM, Phase Correct	ICR1	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	-	-	-
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Fig. 20. PWM Modes Setup

To set the PWM signal to a desired frequency, the OCRnA value needs to be calculated.

$$OCRnA = \frac{f_{clock}}{2K_{ps}f_{des}} - 1 \tag{11}$$

In (11), f_{clock} is 16MHz, and K_{ps} is the prescaler as shown in Fig. 22 with possible values of 1, 8, 64, 256, and 1024. OCRnA is the maximum number of counts.

For example, 20kHz PWM would require OCRnA to be set to 399 as shown in (12).

$$OCRnA = \frac{16MHz}{2(1)(20kHz)} - 1 = 399 \tag{12}$$

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM1[3:0] = 14 or 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation); for all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM (inverting mode)

Fig. 21. PWM Prescaler Setup

CS12	CS11	CS1	Description
0	0	0	No clock source
0	0	1	System clock
0	1	0	Prescaler=8
0	1	1	Prescaler=64
1	0	0	Prescaler=256
1	0	1	Prescaler=1024
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock on T1 pin. Clock on rising edge.

Fig. 22. Compare Match Modes Setup

The duty cycle is programmed by setting OCRnB to a value between 0 and OCRnA. For example, a 50% duty cycle for 20kHz PWM would require OCRnB to be set to 199.

The following code is set up to produce 20kHz PWM signals at Pin 10 in Mode 11 with a prescale of 1:

```
void setup() {
    TCCR1A = (1<<COM1A1) + (1<<COM1B1) + (1<<WGM11) + (1<<WGM10);
    TCCR1B = (1<<WGM13) +  (1<<CS10); // prescaler = 1 (none)
    OCR1A = 399;
    OCR1B = 199;
    DDRB |= (1<<PB1)|(1<<PB2);
}
```

Here the OCR1B value is fixed, but in practice, it can be updated by a control loop.

B. Voltage Analog Reading

Arduino boards contain a multichannel 10-bit ADC for pins A0 to A5, which maps a sensed 0-5V input voltage to an integer value between 0 and 1023. This gives a resolution of 4.9mV per integer step. The input range can be adjusted using analogReference() (the default value will be 5V for 5V boards or 3.3V for 3.3V boards). In our case, the analogReference was set to EXTERNAL, which is the 5V external power source applied to the AREF pin as a reference, because the reference voltage can fluctuate depending on the Arduino power source. Therefore, this work required two separate power supplies.

The coding for setting pin A0 as the analog input pin and setting reference voltage as the external power source is as follows:

```
void setup() {
    analogReference(EXTERNAL); //sets the reference voltage to external source (5V)
    pinMode(A0, INPUT); //Sets pin A0 to perform analog reading
}

void loop() {
    voltage = analogRead(A0);
}
```

Another mode of ADC that does not use analogRead() is ADC Free Running mode. This is a mode where the ADC continuously converts and interrupts at the end of each conversion. This approach does not waste time waiting for the next sample in the loop. The values are just showing up at regular times. Additionally, it improves accuracy due to jitter. However, we did not use ADC Free Running mode due to the limited number of timers that are available on the board.

C. Interrupts

An Arduino Uno can perform the timer compare interrupt using a timer register, TIMSKn. The Interrupt Service Routine (ISR) performs Top and Bottom comparisons. Due to PWM already using Timer 1, the control loop Interrupt used Timer 2. We initially considered a sampling interval of 1ms. Timer 2 control register has a different setting from Timer 1. Timer 2 has only 8 bits compared to Timer 1's 16 bits. Therefore, a higher prescaler is necessary to fit the step within 8 bits (0 to 255). In order to activate the correction cycle every 1ms, presacaler 128 was used to make the 8-bit Output Compare Register OCR2A value a whole number. A whole number of OCR2A allows the interrupt to be triggered at exactly the set time.

$$\frac{16MHz}{128} = 125KHz \tag{13}$$

$$PulseTime = \frac{1}{125KHz} = 8\mu s \tag{14}$$

$$OCR2A = \frac{1ms}{8\mu s} = 125 \tag{15}$$

The bit settings in Fig. 23 indicate how to change the prescaler setting in TCCR2B Timer2 Control Register B:

Prescaler	CS20	CS21	CS22
1	1	0	0
8	0	1	0
32	1	1	0
64	0	0	1
128	1	0	1
256	0	1	1
1024	1	1	1

Fig. 23. TCCR2B Timer2 Control Register B

The code to set up an interrupt at 1kHz is as follows:

```
void setup() {
    cli();//Stop interrupts

    //Timer2
    TCCR2A = 0;     // set entire TCCR1A register to 0
    TCCR2B = 0;     // set entire TCCR1A register to 0

    TIMSK2 = (1<<OCIE2A); // enable Timer2 time compare interrupt

    TCCR2A = 1<<WGM21; //Set CTC mode
    TCCR2B = (1<<CS22) | (0<<CS21) | (1<<CS20); // set 128 prescaler

    OCR2A = 125; //For 1kHz interval with prescaler 128

    sei();//allow interrupts
}

ISR(TIMER2_COMPA_vect){
//Interrupt tasks go in here
}
```

A different prescaler is selected for different frequency of interrupt triggers to ensure that OCR2A is a whole number. For a 2kHz interrupt frequency, a prescaler of 64 was used, and a prescaler of 32 was used for a 4kHz interrupt frequency.

There are restrictions on using interrupts in Arduino [4]. The delay(), millis(), micros(), or any other functions that rely on ISR cannot be used in an ISR. Loops or long tasks cannot be used in an ISR, otherwise, it will delay other interrupts and code. If a long task is required, a flag should be set within the ISR and perform the task within Arduino's loop(). Finally, Serial functions cannot be used in an ISR since they use interrupts as well.