# Using the Arduino Uno to Teach Digital Control of Power Electronics

Lukas Müller, Masihuddin Mohammed, and Jonathan W. Kimball
Missouri University of Science and Technology
Department of Electrical and Computer Engineering
301 W. $16^{th}$St.,Rolla, MO 65401 USA
kimballjw@mst.edu

***Abstract*—There has been tremendous growth in digital control of power converters, and yet many power electronics students have limited experience with embedded systems. The Arduino Uno is an excellent learning platform for embedded programming, but has limited capabilities using the standard libraries. This work explores the use of the Arduino Uno for power electronics applications. With some extra libraries or with direct access to a few registers, the Arduino Uno can support switching frequencies around 100 kHz, suitable for an educational environment. Implementation of key features, such as PWM and analog-to-digital converters, is discussed, along with an example experiment.**

## I. Introduction

Throughout the years, digital control has become more accessible and cheaper than its analog counterpart. In recent years a number of microcontroller prototyping platforms have become available. The most popular among them is the Arduino line [1]. Arduino controllers are easy to use, program, and integrate in electrical projects. A number of previously proposed lab exercises have used the Arduino Uno as the main microcontroller [2]–[4]. In most of these applications, however, the Arduino Uno controlled a slow electrical system, such as a servo motor. The Arduino Uno has limited computational power as it is based on an 8-bit microcontroller. Therefore, it was considered too slow to control power electronic circuits [5]. In an educational environment, though, the Arduino Uno can be adapted to illustrate the use of digital controllers in power electronics.

Many students come to power electronics with limited knowledge or interest in programming. The Arduino platform provides an easy entry point to teach digital control. A template has been constructed that performs many of the key steps: setting up the PWM, ADC, and interrupts. Within the interrupt service routine, the student inserts the digital controller to process the ADC value and set a new duty ratio. Also, for plotting purposes, the serial port may be used with MegunoLink [6]. As students gain familiarity and comfort with embedded control, they can explore the details of PWM and ADC implementation. In doing so, they must learn concepts that are needed on a more conventional power converter control platform, such as setting up specific registers according to datasheets. These aspects of the Arduino language are more like conventional C programming. Ultimately, the interested student would move on to a conventional platform, such as the Texas Instruments C2000 family, but with an understanding of the overall software architecture common to all embedded control systems for power converters.

This paper begins with a discussion of the Arduino platform in section II, then proceeds through the key peripherals and their configuration for power converter applications in sections III and IV. Section V walks through a typical experiment that a student might perform, with results and a full code listing provided in the appendix.

## II. Arduino Uno Hardware and IDE

The Arduino Uno is based on the popular Atmel ATmega 328P. Older Arduino Uno Boards utilized the ATmega8 or ATmega168, however, boards featuring those microcontrollers are no longer available. The ATmega 328P is a low power 8-bit AVR RISC-based microcontroller. The ATmega supports operating frequencies up to 20 MHz, however it is clocked at 16 MHz on Arduino Uno boards. The 328P features a variety of useful peripherals to interact with its environment.

The 328P has an 8 channel Analog to Digital Converter (ADC) with 10 bit resolution [7], but the DIP version only supports 6 channels. The standard ADC clock frequency of the 328P is 125 kHz. The ADC requires approximately 13 clock cycles to complete a conversion, providing a standard maximum sampling rate of 9.6k samples. The 328P does support ADC frequencies up to 1 MHz, allowing for sampling rates close to 77k samples [8]. A method to easily increase the sampling frequency in software is shown in a later section.

Another important feature of the ATmega 328P is its three hardware timers (0,1 and 2). Timer 0 and 2 are 8-bit timers, while Timer 1 is a 16-bit timer. The timers can be used to generate the PWM signal required to control the power converter. At 100 kHz, the PWM resolution is approximately 7.3 bits. In the standard Arduino programming environment, the PWM frequency is set to 490 Hz, which is far too low to be useful in any power electronics application. Methods to access the fast PWM will be elaborated on in later sections. The timer can also be useful to generate timed interrupts.

Besides the PWM and ADC, the 328P also features an analog comparator to interact with analog signals. The analog comparator is capable of generating an interrupt on a rising edge, falling edge or a toggle of the comparator. Popular control strategies such as peak or valley current mode control require an analog comparator to be implemented [9]. The ATmega 328P can therefore be used to realize these control methods as well.

978-1-4673-6847-6/15/$31.00 ©2015 IEEE

The ATmega 328P features a USART to communicated with other devices. On the Arduino Uno Board, the serial communication from the 328P is routed through an additional microcontroller to allow communication over USB. The Arduino programming software has all the required features to communicate and program the Arduino Board over the USB connection. This makes communication between a computer and the Arduino Uno easy and straightforward.

Naturally, the ATmega 328P limits the complexity and speed of the controller. There are other Arduino platforms that may be considered for the same purpose, most notably the newly-released Arduino Zero. These will be explored in future work.

All Arduino Uno boards are supported by the Arduino Integrated development environment (IDE) [1]. The Arduino IDE can be used to both program and communicate with the Arduino boards. The Arduino IDE utilizes its own programming language, which is based on C. The main advantage of the Arduino programming language is its ease of use. A variety of functions used to interact with the peripherals are predefined in the IDE. Some of these peripherals include: External Interrupts, Timer with PWM, ADC, and Serial Communication. The Arduino website offers a comprehensive description off all the supported functions for the Arduino boards [1].

## III. PWM

Pulse-width-modulation (PWM) is a fundamental concept in power electronics and warrants special attention. The ATmega 328P's three timers each feature two output compare units that can be used to generate PWM signals. The 328P's timer are also configured to support two distinct PWM modes: Fast-PWM and Phase-Corrected-PWM.

Fast-PWM is implemented by using the timer to count upwards until a specified top value is reached. Once the value is reached the counter is reset. The PWM is generated by comparing the current count of the timer to a specified compare value. If the counter value is below the compare value the pin has a configurable state (high or low). Once the counter is equal to the compare value, the state is flipped. The pin is reset to its original state when the timer resets. This is usually termed "edge-aligned" PWM.

Phase-Corrected-PWM works similar to Fast-PWM in the way it utilizes a counter and the compare registers. However, the time reset condition is different for the Phase-Corrected-PWM. In Fast-PWM the counter only counts up and is reset to its bottom value when it reaches the top value. For the Phase-Corrected version the counter counts up until it reaches its top value and then counts back down to its bottom value. The compare register works similar to the Fast-PWM, however, the pin-state is flipped every time the counter matches the compare register value. This is usually termed "center-aligned" PWM.

The implementation of the Fast-PWM and Phase-Corrected-PWM are similar, so the question arises how the two operation modes differ from one another. Fast-PWM, as the name implies, maintains a higher PWM resolution at high frequencies compared to the Phase-Corrected-PWM. This comes from the fact that with the same top counter value, the Fast-PWM frequency is roughly twice that of the Phase-Corrected PWM (Fast only counts up, Phase-Corrected counts up and back down which takes twice as long). On the other hand, the Phase-Corrected-PWM, as its name implies, ensures that the phase between the PWM signals are always the same. For the Fast-PWM, one of the edges is always at the beginning of the switching cycle and the timing of the other edge changes based on the duty cycle. In Phase-Corrected PWM the center of the signal is always at the same location and both switching edges move farther from and closer to this center based on the duty cycle. This paper will only address Fast-PWM, which is typically the best choice for buck converters.

### A. Fast-PWM library

A variety of add-on libraries have been written for the Arduino IDE, which provide access to the advanced features of the Atmega 328P. There are a number of timer libraries, which enable easy access to a fast PWM. One of those libraries is simply called "TIMER1" [10]. The library allows the PWM frequency to be adjusted to any frequency below 1 MHz. In addition, the library allows the PWM frequency to be set using a 10-bit value regardless of the switching frequency or resolution of the chosen PWM configuration. One has to keep in mind that the resolution of the PWM will still be limited by the switching frequency. The library and its function calls are available at [10]. A pre-written library is the quickest and easiest way to setup a PWM signal, however, it consumes more resources of the microcontroller than setting the PWM up by hand.

### B. Manual Configuration for Fast-PWM

The timers can be setup by hand to generate a PWM signal on either of its compare outputs. Setting up the PWM will eliminate any potential overhead that might be present in libraries such as "TIMER1." It also gives the programmer much more control over the program. Lastly, setting up the PWM by hand requires less programming space, freeing it up for other purposes. On the other hand, the registers will have to be set up by hand to enable PWM at the desired frequency, duty cycle and pin mode. This is an excellent advanced topic after students are comfortable with the embedded control process.

The operation of timer x is controlled by its timer control register (TCCRx), where x can be 0,1, or 2. The timer control register is 16bits wide, therefore, the register is split between TCCRxA and TCCRxB. The PWM mode, time prescaler and output mode are configured using this register. The required bits that have to be set to configure the PWM a certain way are given below.

The TCCR1 register controls the function of timer 1 [7]. The WGM11,WGM12 and WGM13 bit in the TCCR1 register are set high to enable fast PWM with fully adjustable switching frequency. In this mode, Timer 1 will count until it reaches the value stored in ICR1. However, a suitable prescaler has to be selected before the ICR1 value can be calculated to obtain a suitable PWM frequency. The CSxx bits in the TCCR1 register are used to set the timer 1 pre-scaler. The most suitable prescaler can be calculated using

$$P = \frac{f_{328P}}{65535 \times f_{pwm}} \tag{1}$$

where $P$ is the most desirable prescaler, $f_{328P}$ is the operating frequency of the ATmega 328P (16 MHz on the Arduino Uno) and $f_{pwm}$ is the desired PWM frequency. The value of $P$ must be rounded up to the next available prescaler value. For switching frequencies commonly encountered in power electronics, a prescaler of 1 will be required. This prescaler can be set by setting CS10 = 1 and the remaining prescaler select bits CS11 and CS21 to zero [7]. Timer 1 can be configured to different prescalers using the configuration shown in [7]. With the prescaler known, the exact PWM frequency can be set by calculating the required ICR1 value:

$$ICR1 = \frac{f_{328P}}{P \times f_{pwm}} - 1 \tag{2}$$

The 328P's timer register must be configured to generate the PWM signal on one of the output pins. Timer one has two outputs OCR1A and OCR1B (for channel A pin 9 and channel B pin 10 respectively). These outputs can configured by setting the control bits in the TCCR1A register. A PWM channel can be configured to the non-inverting mode by setting the COM1X1 bit to one and the COM1X0 bit to zero, where X is the letter of the channel. In the non-inverting mode, the PWM signal will be high at the beginning of the switching cycle and then turn low. The inverting mode can be enabled by setting both COM1X1 and COM1X0 to 1. In this configuration the PWM signal will be low at the beginning of each switching cycle. Configuring one channel in non-inverting and the other channel to inverting mode allows two signals to be generated that are $180^{\circ}$ phase-shifted. However, this configuration does not provide dead time during one of the two switching transitions of the signals.

The duty cycle is then set by the channel compare values for timer 1, OCR1A and OCR1B (for channel A pin 9 and channel B pin 10 respectively). The required OCR1x value for a given duty cycle in the non-inverting mode is determined by:

$$OCR1X = ICR1 \times d \tag{3}$$

where $d$ is the duty cycle ranging between 0 and 1. In the inverting mode, $d$ in (3) would be replaced with $(1-d)$. OCR1x must be rounded to the nearest whole number. The duty cycle of a given channel can be adjusted anywhere in the program by setting OCR1X to the necessary value. The OCR1X register is double buffered, therefore, the register will not update until the next PWM cycle. This prevents random switching operations whenever the duty cycle value is updated.

The sample code given below demonstrates how timer 1 can be configured in the setup function to generate a PWM signal at 100 kHz, in non-inverting mode, and a duty cycle of 50% on the Arduino Uno pin 9.

```
pinMode(9, OUTPUT);
TCCR1A = 0;
TCCR1B = 0;
//Clear Timer 1 Control Register to
//ensure the timer is properly configured
ICR1 = 159;               //Set TOP value for
    Timer 1
OCR1A = 79;               //Set Channel A duty
     cycle to 50%
TCCR1A |= (1 << COM1A1);  //Set channel A to
    non-invert mode
TCCR1A |= (1 << WGM11);
TCCR1B |= (1 << WGM12)|(1 << WGM13);
//Enables Fast PWM with ICR1 as TOP
TCCR1B |= (1 << CS10);    //Starts timer with
    prescaler of 1
```

## IV. Other functions

### A. Timer based Interrupts

A timer based interrupt also needs to be generated for most converter control schemes. The standard Arduino IDE, does not provide a function for timer interrupts. Libraries are, however, available to set up and generate these interrupts easily. Timer 1 is used for PWM purposes and Timer 0 is used by the Arduino IDE for delay and debugging functions. Timer 2 is not assigned any functionality yet, therefore it can be used to generate the timed interrupt. The library "FrequencyTimer2" can be used to provide easy access to a Timer 2 based interrupt [11]. The library uses three predefined functions to setup, enable and execute a desired interrupt at a fixed frequency. The library and documentation for it can be found at [11].

Similar to the PWM, the timer 2 interrupt can also be configured by hand. Once again, configuring the interrupt by hand is more difficult to perform, however, it provides performance benefits. Timer 2 is controlled in a similar manner to timer 1. The TCCR2 register controls the behavior of the timer. In this case the timer is setup to act as simple counter though to trigger an interrupt at a fixed frequency. The WGM21 bit in the TCCR2 register is set high to allow generation of an interrupt at a specified frequency. Similar to the timer 1 PWM setup, an appropriate prescaler has to be determined for timer 2. The available prescalers range from 1 to 1024. Equation (1) can be modified to find the required prescaler for timer 2:

$$P_{int} = \frac{f_{328P}}{256 \times f_{int}} \tag{4}$$

where $f_{int}$ is the frequency at which the interrupt is suppose to occur. $P_{int}$ needs to be rounded up to the next available prescaler value. The bits required to set the timer to a certain prescaler can be found in [7]. Additionally, as with the PWM, the appropriate overflow value needs to be calculated to obtain the desired frequency. Unlike, the PWM setup the overflow is tied to one of the compare registers, OCR2A or OCR2B. Which register triggers the interrupt is determine by configuring the interrupt mask register TIMSK2. To use OCR2A, the OCIE2A bit in TIMSK2 would be set high [7]. The required value of the compare register can be calculated using a modified version of (2):

$$OCR2X = \frac{f_{328P}}{Pint \times f_{int}} - 1 \tag{5}$$

where OCR2X is the appropriate compare register. Lastly, global interrupts need to be enabled using the sei() command. With these setting an interrupt service routine will be triggered at the specified frequency. The name of the interrupt TIMER2_COMPX_vect, where X is the compare register used. A sample code to set up a interrupt with a frequency of 100 kHz on timer 2 is shown below.

```
void setup(){
TCCR2A = 0;
TCCR2B = 0;
//Clear Timer 2 Control Register to insure
// the timer is properly configured
OCR2A=159;
//Set overflow value to fix to specific
//frequency
TCCR2A |= (1 << WGM21);
//Set timer 2 to CTC Mode
TIMSK2 |= (1 << OCIE2A);
//Set interrupt on compare match
TCCR2B |= (1 << CS20);
//Start timer with prescaler of 1
sei();
//enables global interrupts
}

ISR (TIMER2_COMPA_vect)
{
//Code to be executed at fixed frequency
}
```

It is important to keep the code executed in the interrupt service routine as brief as possible, or limit the interrupt execution frequency as the processor might get hung up continuously in the ISR.

### *B. Analog to Digital Converter*

The operation of the ADC can be adjusted as well to allow for faster sampling speeds. The default clock frequency of the Arduino Uno's ADC is set to 125 kHz. This frequency is generated using a prescaler of 128 for the ADC. Atmel specifies that the ADC supports clock frequencies up to 1 MHz though [7], [8] without significant loss in accuracy. Faster ADC frequencies are possible, but are not characterized by Atmel [8]. The ADC clock prescaler can be adjusted to change the ADC frequency. The prescaler is set by changing the ADPSX bits in the ADC control and status register ADCSRA [7]. Also, the ADC may be used in free-running mode for faster samples.

The following code sample can be used to adjust the ADC frequency in the Arduino IDE to 1 MHz.

```
const unsigned char PS_128 = (1 << ADPS2)
| (1 << ADPS1) | (1 << ADPS0);
//Defines prescaler of 128
const unsigned char PS_16 = (1 << ADPS2);
//Defines prescaler of 16
ADCSRA &= ~PS_128;
//removes bits set by arduino
//library for 128 prescaler
ADCSRA |= PS_16;
//set ADC prescaler to 16
```

The ADC function in the Arduino IDE can be executed much faster with a higher ADC clock frequency.

Even with an increased clock cycle the analogRead function still takes a long time to execute. This is due to the fact that the analogRead function operates the ADC in single shot mode. When the function is called the ADC is enabled, taking a sample of the next 13 ADC clock cycles before returning the measurement. During the sampling process the microcontroller is idling, not performing any functions. The time spent between analog to digital conversions can be utilized to execute the compensator code of the controller. This would allow the Atmega 328P to update the PWM duty cycle during each sample, allowing much higher bandwidths. The ADC can be configured in free running mode, setting the ADC to continuously sample the selected input and generating an interrupt when the next result is available. To set up a free running ADC, the control register have to configured by hand. The ADMUX, ADCSRA, ADCSRB and DIDR0 register control the function of the ADC. The prescaler configuration was already described above. The ADMUX register is used to select the pin from which the ADC will read and select a voltage reference. The ADC input can be selected by setting the bits in the ADMUX register as shown in [7]. Similarly the voltage reference can be selected [7]. A common reference is the internal 5 V reference, which is selected by setting REFS1 to 0 and REFS0 to 1. The ADCSRA and ADCSRB register are set to configure the actual operation behavior of the ADC. In ADCSRA, ADEN is set high to enable the ADC. A analog to digital conversion is started by writing a one into ADSC. The ADC is set up to trigger automatically of an external source by setting the ADATE bit high. The trigger sources is selected by the ADCSRB register. To configure the ADC in free running mode all bit in ADCSRB should be written low, with the exception of ACME, which should be set high. Lastly, the ADC interrupt needs to be enabled to read out the ADC periodically, This is accomplished by writing the ADIE bit in the ADCSRA register high. Using this configuration a new ADC reading will be available after every ADC interrupt in the ADCL and ADCH registers. ADCL contains the lower 8 bits of the ADC readout, while ADCH will contain the 2 most significant bits. A sample code to set up the ADC in free running mode is given below.

```
void setup(){
ADMUX = 0;
ADCSRA = 0;
ADCSRB = 0;
ADMUX |=(1 << REFS0);
ADCSRA|=(1 << ADEN) | (1 << ADATE)
| (1 << ADIE) | (1 << ADPS2);
ADCSRB|=(1 << ACME);
sei();
bitWrite(ADCSRA, 6, 1);
}
void ISR(ADC_vect){
aval = ADCL;
aval +=ADCH << 8;
}
```

Setting up the ADC in free running mode not only frees up processor time, but also generates an interrupt at a fixed frequency. Instead of tying the execution of the controller algorithm to the timer 2 interrupt, it can be connected to the ADC interrupt directly. This allows the PWM duty cycle to be recalculated between consecutive samples of the ADC, enabling higher bandwidths.

The free running ADC offers better controller performance, however, it does require the PWM generation and the ADC to be synchronized to one another. To ensure stable operation of the compensator, the ADC has to sample the signal at an

integer multiple of the PWM frequency. The ADC and PWM clocks are generate by the same source, therefore, they can be synchronized with the right configuration. In free running mode, the 328P's ADC will sample the selected signal every 13 ADC clock cycles [7]. The ADC sampling cycles are related to the ATmega 328P's machine cycles by the prescaler. The total number of machine cycles between ADC samples is 13 multiplied by the selected ADC prescaler. As seen in the previous section, the number of machine cycles during each PWM cycle is equal to the PWM prescaler multiplied by the TOP value of the PWM timer (ICR1). To ensure the ADC samples are synchronized with the PWM signal the following equality has to be satisfied:

$$nP_{PWM} \times ICR1 = 13P_{ADC} \tag{6}$$

where $n$ is the number of PWM cycles between samples. The ADC frequency is difficult to adjust as it is determined entirely by the ADC prescaler. The interval between samples and PWM frequency is more flexible, therefore, the interval should be chosen to work with the selected ADC frequency.

In conclusion, if a high bandwidth with a more complex compensator are desired and the PWM frequency is flexible, then the free running ADC is the preferable option. If the PWM frequency is critical or the frequency has to be varied during operation, then the ADC should be operated in one shot mode.

### C. Analog Comparator

As described in the introduction, the ATmega 328P features one analog comparator. This comparator can be used in analog type control schemes such as peak or valley current mode control. The analog comparator is not supported by any standard Arduino libraries. Additionally, there are no add-on libraries available to handle the analog comparator. Therefore, the only way to configure the analog comparator is using the respective control registers [7]. The analog comparator is configured by the ACSR and ADCSRB registers [7]. The standard analog comparator inputs are pin 6 and 7 on the Arduino Uno. The ADCSRB register can be used to modify the analog comparator inputs and select one of the ADC pins. The ACME bit in the ADCSRB register can be set high to enable the selection of different pinouts. The ACSR control register determines the general behavior of the analog comparator. To enable the comparator the ACD bit should be written low; unlike the other peripherals described in this paper a logic 0 turns the comparator on. The ACIE bit enables the analog comparator interrupt with a logic 1. The comparator interrupt flag is the ACI bit. When activating the ACIE bit, the ACI bit should be cleared once, as turning on the ADC can cause a false triggering of the comparator interrupt. The triggering mode of the comparator is set using the ACIS1 and 0 bits. The analog comparator can be set to trigger on an output toggle (ACIS1=0,ACIS=0), a rising edge (ACIS1=1,ACIS=0) or falling edge (ACIS1=1,ACIS=1) [7]. Using the described registers the analog comparator operation can be adjusted for the desired operating mode. The comparator interrupt may be used to implement peak or valley current mode control or protection modes.

## V. Example Experiment

The goal of the proposed platform is to introduce power electronics students to digital control concepts. A template has been written in the Arduino IDE that establishes all of the discussed settings. The PWM and ADC pins are set, the PWM frequency is set to 100 kHz, and registers are set to put the ADC into its fastest possible mode. At the beginning of the sketch, a variety of constants are defined, so that students may readily change, e.g., the output voltage reference.

The interrupt service routine is essentially blank. The laboratory assignment contains the following pseudocode to help students start:

```
error = reference-sampled;
accum += error;
accum = constrain(accum,Min,Max);
pwm = Ki*accum + Kp*error;
pwm = constrain(pwm,0,PERIOD);
```

They must translate this into properly formatted code with properly declared variables. They must also design the compensator, which is of the form

$$H(z) = K_p + \frac{K_i}{z-1} \tag{7}$$

The controller must be designed for a buck converter with the following parameters: 12 V input, 5 V output, 370 $\mu$H, 470 $\mu$F, 100 kHz, with a load of 2.5 $\Omega$.

First, students use MATLAB to create a Bode plot of the plant model and then design a compensator with appropriate gain and phase margins and a bandwidth of approximately 100 Hz. Fig. 1 and Fig. 2 illustrate the loop gain and step response, respectively, for control gains of $K_p = 0.030293$ and $K_i = 0.0047852$. The gain margin is infinite, and the phase margin is $99.4^\circ$ at 89.1 Hz.

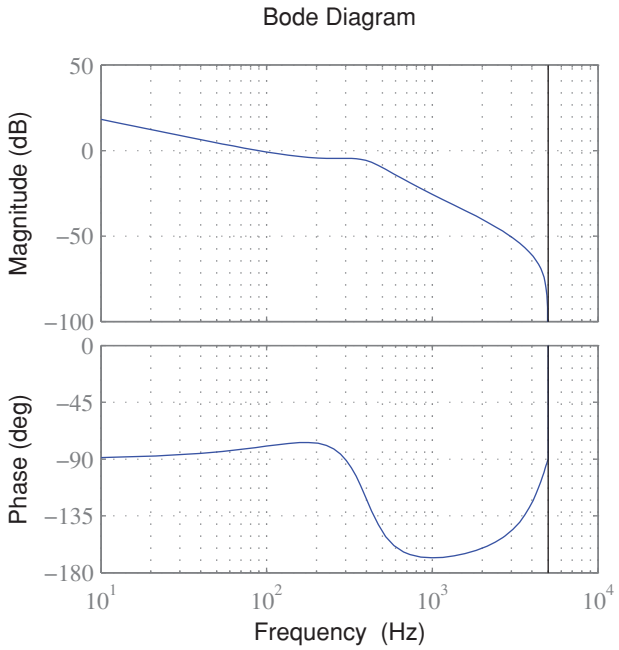


Fig. 1. Bode plot of loop gain for buck converter with gains as noted in the text.

Next, the gains must be adapted to the embedded platform. The output voltage is sensed through a 2:1 voltage divider,

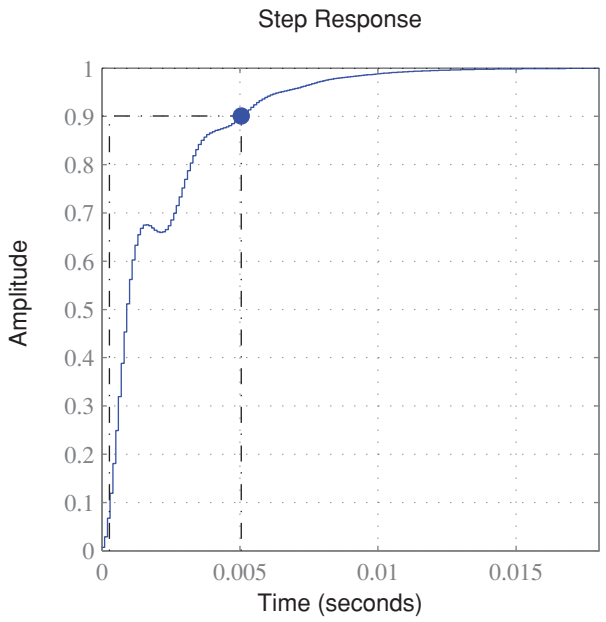


Fig. 2. Closed-loop step response of the small-signal model of buck converter with gains as noted in the text. The rise time, 5 ms, is indicated.

and the ADC gives a 10-bit result on a 5 V scale. Therefore, if the output voltage is 5.0 V, the ADC result is 512. For 100 kHz switching, the PWM limit is 160 counts. All factors considered, the controller sensing-to-actuation gain is

$$\frac{1}{2} \times \frac{1024}{5.0} \times \frac{1}{160} = 0.640 \tag{8}$$

So that the microcontroller can perform integer math, the control gains are scaled by $2^{16}$, and then the result is right-shifted by 16 places. Thus, the gains entered in the code are $K_p = 3102$ and $K_i = 490$.

Finally, the pseudocode is converted to real code with the derived gains. In the main loop, the serial port is used for datalogging via MegunoLink [6]. A typical result, the response at power-up, is shown in Fig. 3. The complete code listing is provided in the appendix. Notice that the majority of the code has the general look and syntax of conventional embedded C code. However, the use of the Arduino environment and libraries simplifies some of the configuration and the datalogging.

## VI. Conclusion

The Arduino Uno platform is an excellent tool for teaching embedded control. With appropriate libraries, or direct access to registers of the ATmega 328P used on the board, much higher capabilities are available than the basic Arduino IDE supports. These extended capabilities support power converter control at a level suitable for a teaching laboratory. A template allows students to focus first on the essential control functions, that is, the functions necessary to implement the feedback loop without worrying about register configurations. This is a stepping stone to more sophisticated control implementations.

## Appendix: Code Listing

```
#include "TimerOne.h" //Library for PWM
#include "FrequencyTimer2.h" //Library for
    Controller Interrupt
#include <avr/io.h> // AVR Device specific I/O
     definitions
#include <stdint.h> // Declare sets of Integer
     types

//Setting up constants
const uint8_t PWM1_PIN = 9; // Setting PWM1 on
     Pin9
const uint8_t Vout_Pin = 3; //Vout Measurment
    ADC Pin
const int PWM1_Max = 100;// 16 MHz ticks
const int PWM1_Min = 0;
const int MAX = 160;// PWM period in 16 MHz
    ticks

//PI Controller Tuning Constants
const int KpScaled = 3102;
const int KiScaled = 490; // Scaled value, by
    0.64*2^-16
const int KpiScale = 16; // Scaling power of 2
const int VoutADC = 512; // scaled version of
    setpoint, in ADC counts
const int INTMAX = 21400; // Integrator value,
     to constrain Vout Integral

//Different Prescaler Settings for ADC
const unsigned char PS_4 = (1 << ADPS1); //
    Defines prescaler of 4
const unsigned char PS_8 = (1 << ADPS1) | (1
    << ADPS0); // Defines prescaler of 8
const unsigned char PS_16 = (1 << ADPS2); //
    Defines prescaler of 16
const unsigned char PS_128 = (1 << ADPS2) | (1
     << ADPS1) | (1 << ADPS0); // Defines
    prescaler of 128

//Setting up program memeory
volatile int PWM1 = 0;
long int PWM1d = 0;
int IntVout = 0; //Current ADC value of output
     voltage
int incomingByte = 0; //Serial Buffer Variable
long int Vouterror = 0;
long int Voutintegral = 0;

void setup(){
  Serial.begin(115200); // Set Serial to
      fastest possible speed
  // Requires MegunoLink Software to read

  pinMode(9, OUTPUT);
  TCCR1A = 0;
  TCCR1B = 0;
  //Clear Timer 1 Control Register to ensure
      the timer is properly configured
  ICR1 = MAX-1;  //Set TOP value for Timer 1,
      MAX=160
  OCR1A = PWM1_Min;  //Set Channel A duty
      cycle to min initially
  TCCR1A |= (1 << COM1A1);  //Set channel A to
       non-invert mode
  TCCR1A |= (1 << WGM11);
```

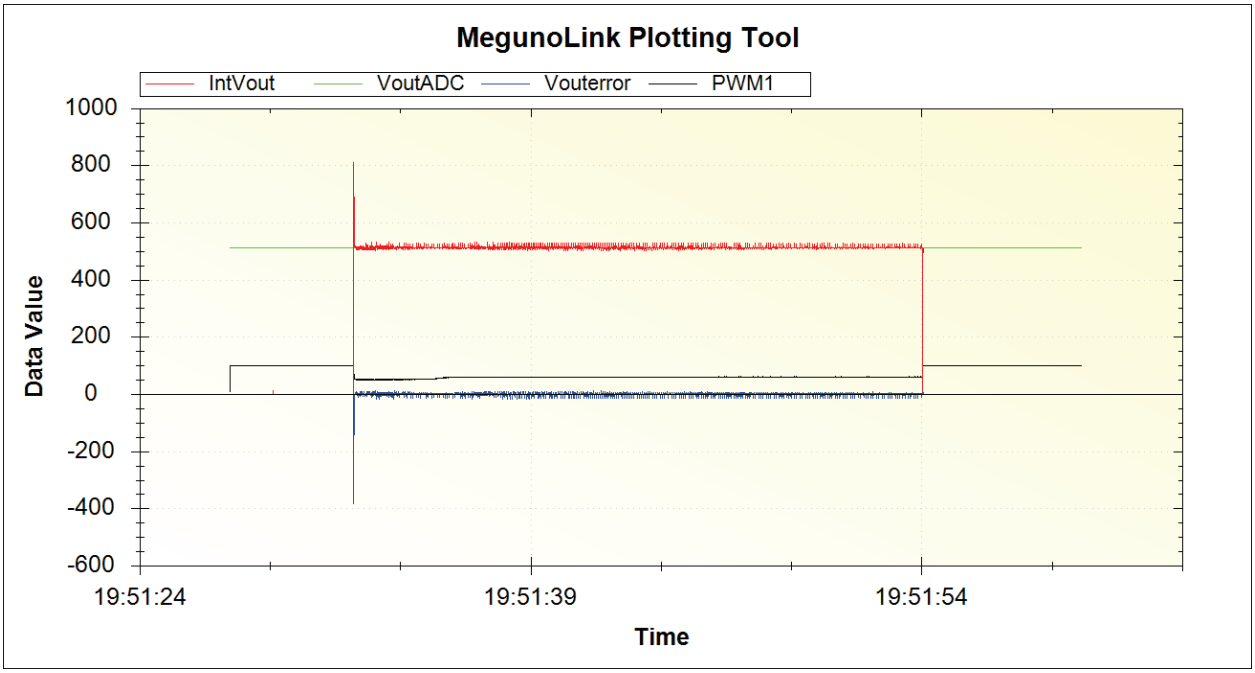


Fig. 3. MegunoLink plot for startup and regulation.

```
  TCCR1B |= (1 << WGM12)|(1 << WGM13);  //
      Enables Fast PWM with ICR1 as TOP
  TCCR1B |= (1 << CS10);  //Starts timer with
      prescaler of 1

  //Sets up ISR to execute PI loop at fixed
      time interval
  FrequencyTimer2::setPeriod(1000);//Set up
      ISR period (value in us)
  FrequencyTimer2::enable();//Initialize ISR
      timer
  FrequencyTimer2::setOnOverflow(controller);
      //Enable ISR

  //Sets up ADC for faster read than Arduino
      Default Analog Read
  ADCSRA &= ~PS_128; //Clears ADC Pre-Scaler
  ADCSRA |= PS_16; //Change Pre-Scaler to
      allow fastest possible ADC
}

void controller() //Controller ISR
{
  //Updates PWM
  OCR1A = PWM1;
  //Pull new measured values
  IntVout = analogRead(Vout_Pin); //Read
      output of converter

  //Run PI Control loop for Output voltage
  Vouterror=VoutADC - IntVout;
  Voutintegral=Voutintegral+Vouterror;
  Voutintegral=constrain(Voutintegral,-INTMAX
      ,+INTMAX);
  PWM1d=(KpScaled*Vouterror+KiScaled*
      Voutintegral);

  //Ensures duty cycle limitation of CFCW
      multiplier
  PWM1=(int) (PWM1d >> KpiScale);
  PWM1=constrain(PWM1,PWM1_Min,PWM1_Max);
  //Done with controller execution, wait for
      next iteration to update controller
}

//Main Program, handles datalogging only as
    controller is time sensitive
void loop(){
  Serial.print("{PWM1,T,"); //Plotting PWM1 vs
       time
  Serial.print(PWM1);
  Serial.println("}");
  Serial.print("{IntVout,T,"); // Plotting
      IntVout vs time
  Serial.print(IntVout);
  Serial.println("}");
  Serial.print("{VoutADC,T,"); // Plotting
      VoutADC vs time
  Serial.print(VoutADC);
  Serial.println("}");
  Serial.print("{Vouterror,T,"); //Plotting
      Vouterror vs time
  Serial.print(Vouterror);
  Serial.println("}");
}
```


## ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under award ECCS-0900940, and by the US Department of Energy Sunshot program under award DE0006341, the MARMET Center.


## REFERENCES


[1] Arduino.cc, “Arduino - reference,” www.arduino.cc/en/Reference, 2015.

[2] P. Fajri, N. Lotfi, M. Ferdowsi, and R. Landers, "Development of an educational small scale hybrid electric vehicle (hev) setup," in *Electric Vehicle Conference (IEVC), 2013 IEEE International*, Oct 2013, pp. 1–6.

[3] R. Fransiska, E. Septia, W. Vessabhu, W. Frans, W. Abednego, and Hendro, "Electrical power measurement using arduino uno microcontroller and labview," in *Instrumentation, Communications, Information Technology, and Biomedical Engineering (ICICI-BME), 2013 3rd International Conference on*, Nov 2013, pp. 226–229.

[4] D. LaSelle, "A low power control system optimized for solar thermal power generation," in *Global Humanitarian Technology Conference (GHTC), 2013 IEEE*, Oct 2013, pp. 135–141.

[5] R. Pittini, Z. Zhang, and M. Andersen, "An interface board for developing control loops in power electronics based on microcontrollers and dsps cores -arduino /chipkit /dspic /dsp /ti piccolo," in *Control and Modeling for Power Electronics (COMPEL), 2013 IEEE 14th Workshop on*, June 2013, pp. 1–7.

[6] Megunolink, "Megunolink lite," http://www.megunolink.com/megunolink-lite/, 2015.

[7] Atmel, "Atmel 8-bit microcontroller with 4/8/16/32kbytes in-system programmable flash," http://www.atmel.com/Images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet.pdf, 2013.

[8] ——, "Avr120: Characterization and calibration of the adc on an avr," http://www.atmel.com/Images/doc2559.pdf, 2006.

[9] K. Suryanarayana, L. Prabhu, S. Anantha, and K. Vishwas, "Analysis and modeling of digital peak current mode control," in *Power Electronics, Drives and Energy Systems (PEDES), 2012 IEEE International Conference on*, Dec 2012, pp. 1–6.

[10] Arduino.cc, "Arduino playground - timer1," playground.arduino.cc/Code/Timer1, 2015.

[11] ——, "Arduino playground - frequencytimer2," playground.arduino.cc/Code/FrequencyTimer2, 2015.