

Implementation of an sPWM signal

☆ 119 stars

🔗 73 forks

☆ Star

👁 Watch ▾

<> Code

🕒 Issues 4

🔗 Pull requests

💬 Discussions


▶ Actions

📁 Projects

📖 Wiki

🔗 master ▾

⋮



Irev-Dev Merge pull request #15 from shuyuan-liu/master ...

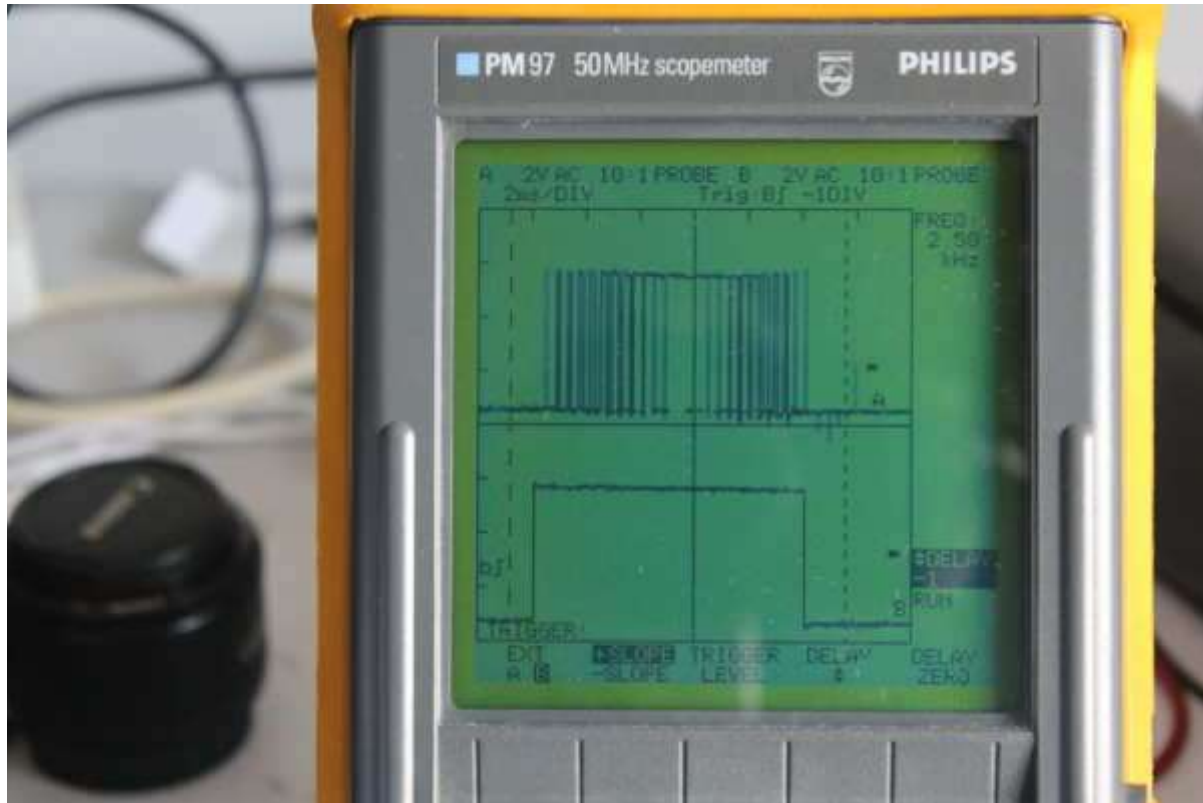
on Jul 9 ⌚ 55

[View code](#)

☰ README.md

Arduino-Atmel-sPWM

Implementation of an sPWM signal on Ardunio and Atmel micros



Introduction

The aim of this repo is to help the hobbyist or student make rapid progress in implementing an sPWM signal on a arduino or atmel micro, while making sure that the theory behind the sPWM and the code itself is understood.

Please also note that:

- It's assumed the reader has a basic understanding of C programming
- If you plan on making an inverter please read the safety section
- Feel free to collaborate on improving this repo

Table of Contents

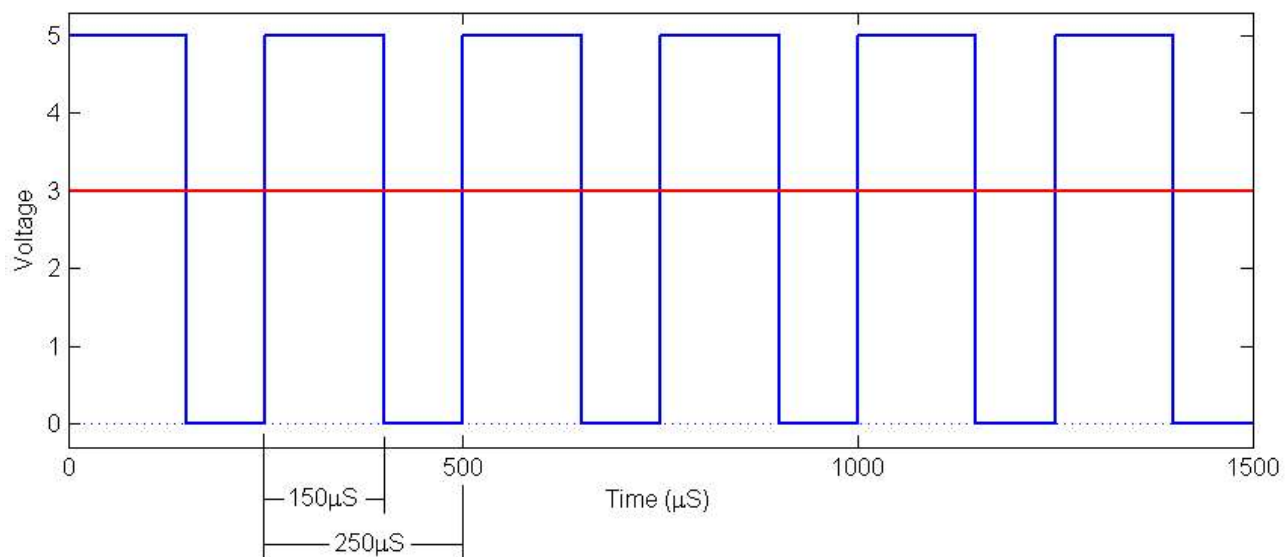
- [Brief Theory](#)
 - [Basic PWM](#)
 - [Typical micro implementation](#)
 - [Sinusoidal PWM](#)
- [Code & Explanation](#)
 - [sPWM_Basic](#)
 - [sPWM_Generate_Lookup_Table](#)

- [Testing the Signal](#)
 - [Viewing Pulse Widths with an Oscilloscope](#)
 - [Viewing the Filtered Signal with an Oscilloscope](#)
 - [Listening to the Signal](#)
- [Compatibility](#)
- [Safety](#)

Brief Theory

Basic PWM

Pulse width modulation's (PWM) main use is to control the voltage supplied to electric circuits, it does this by rapidly switching a load on and off. Another way of thinking of it is to consider it as a method for a digital system to output an analogue signal. The figure below shows an example of a PWM signal.



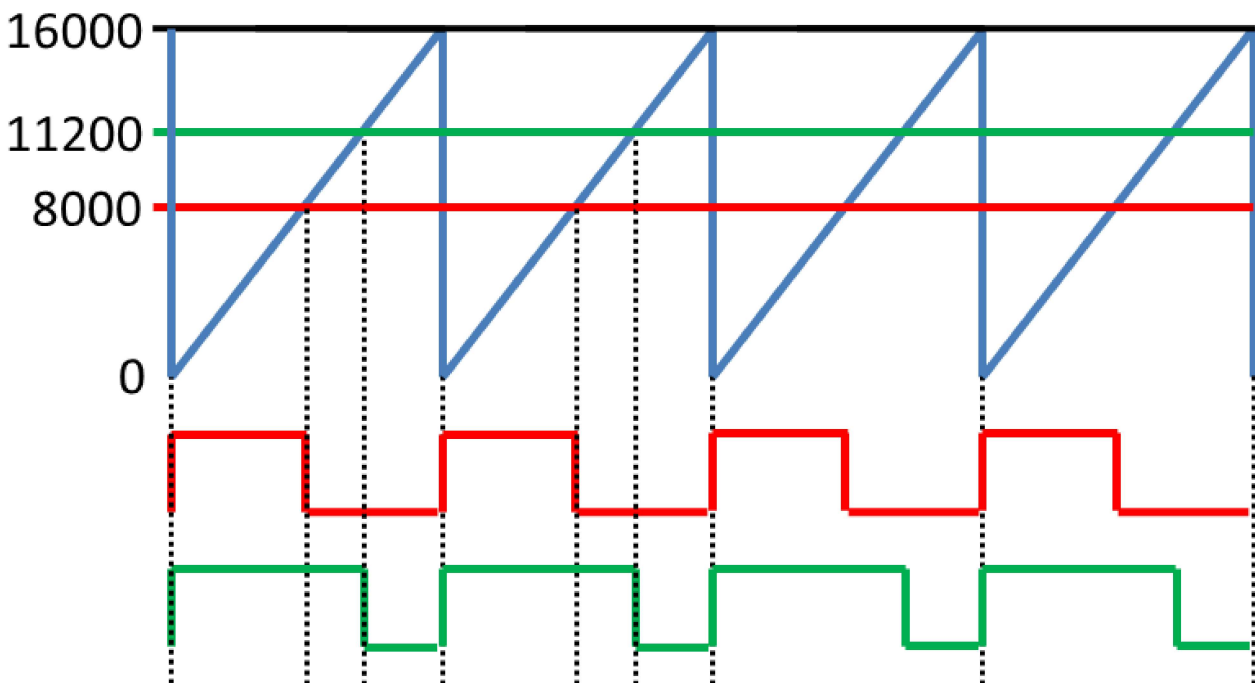
There are two properties to a PWM signal, the frequency which is determined by the period of the signal and the duty cycle which is determined by the high-time of the signal. The signal in figure above has a period of 250μs which means it switches at 4KHz. The duty-cycle is the percent high time in each period, in the last figure the duty-cycle is 60% because of the high-time and period of 150μs and 250μs respectively. It is the duty-cycle that determines average output voltage. In figure above the duty-cycle of 60% with 5V switching voltage results in 3V average output as shown by the red line. After filtering the output, a stable analogue output can be achieved. the figure below shows PWM signals with 80% and 10% duty-cycles. By dynamically changing the duty-cycle, signals other than a flat output voltage can be achieved.



Typical microcontroller PWM implementation

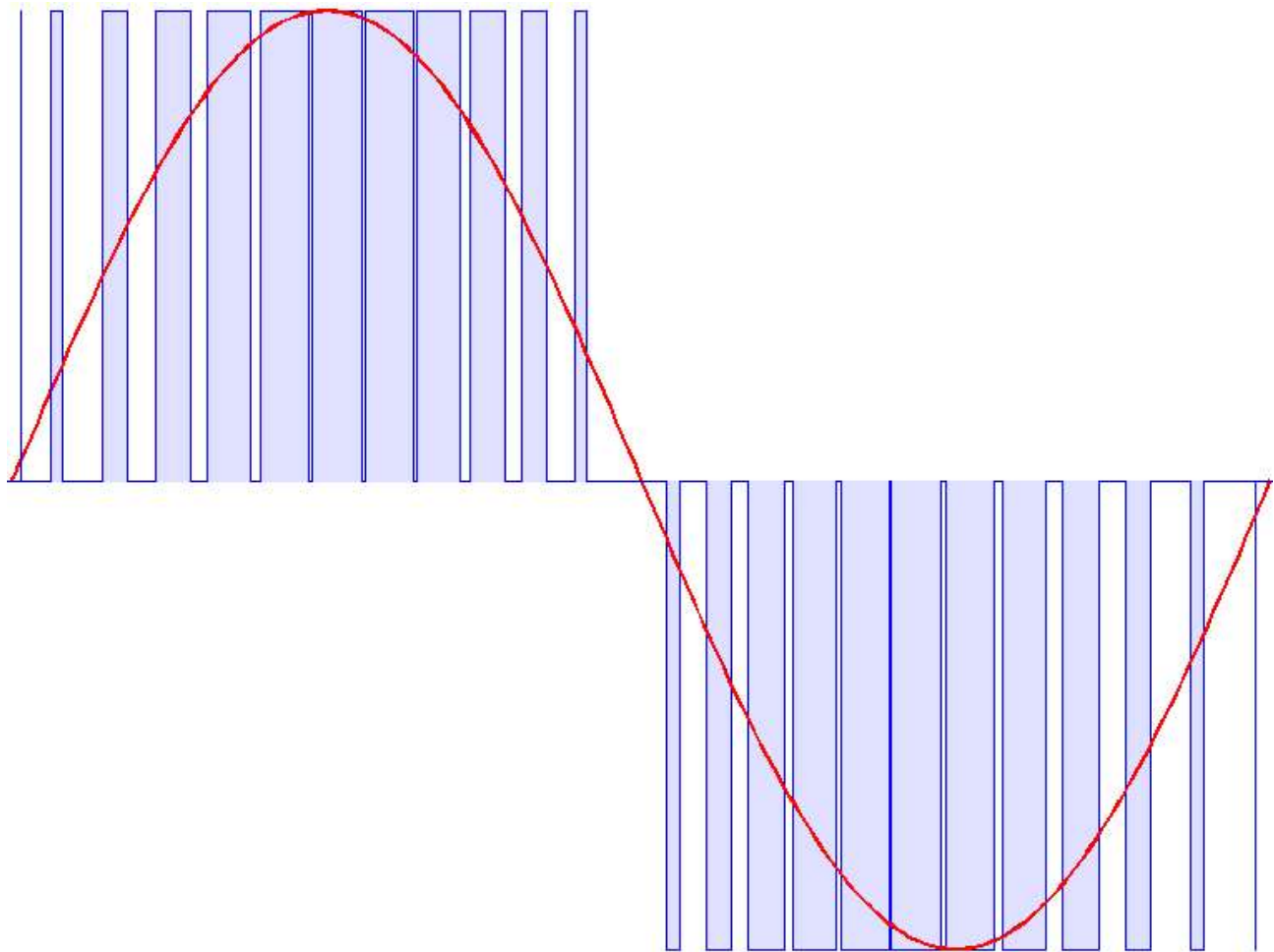
This section describes how micro-controllers use timers/counters to implement a PWM signal, this description relies heavily on the figure below. Here the blue line represents a counter that resets after 16000, this gives the period of the PWM and also the switching frequency (f_s), if this micro has a clock source of 16MHz, then f_s will be $16 \times 10^6 / 16 \times 10^3 = 1\text{KHz}$.

The two other values of 11200 and 8000 shown as the green and red line, they represent compare output values of the microcontroller. When the counter reaches this values, the microcontroller can change the output of some pins. In this case two pins are set to HIGH when the counter resets and then Set low when the counter reaches each of the output compare values. This determines the high time and therefore duty-cycle of the PWM. It's important to understand how a PWM is typically set up in a microcontroller, as the explanation of the code in the next section will not make sense otherwise.



Sinusoidal PWM

A sinusoidal PWM (sPWM) signal can be constructed by dynamically changing the duty-cycle. The result is short pulses at the zero-crossings and long pulses at the wave peaks. This can be seen in the figure below.



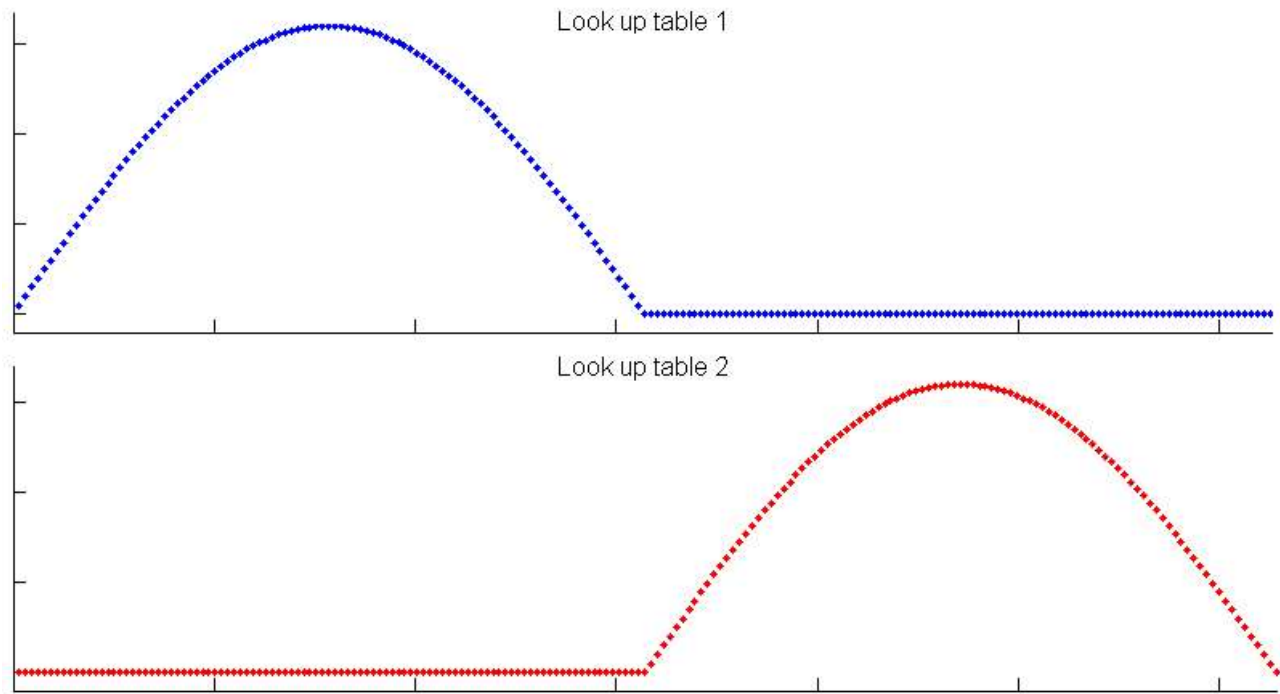
The figure shows negative pulses which is not possible on most micro-controllers. Instead normally this is implemented with two pins, one pulsing the positive half of the sin wave and the second pulsing the negative half, this is how it is implemented in this repo as having the signal split across two pins makes sense if the signal is going to be used to control a H-bridge.

Code and Explanation

In this chapter we'll step through the code found in the folder `sPWM_basic`, and then the difference between it and the code found in the folder `sPWM_generate_lookup_table` will be discussed. Code found in `sPWM_atmel` is for use on an atmel chip without using the arduino IDE. The code toggles a pin for every period of the sine output in order to make it osilloscope friendly.

sPWM_Basic

The following C code implements an sPWM on an Atmel micro-controller. The signal is generated on two pins, one responsible for the positive half of the sine wave and the other pin the negative half. The sPWM is generated by running an (ISR) every period of the PWM in order to dynamically change the duty-cycle. This is done by changing the values in the registers OCR1A and OCR1B from values in a look up table. There are two look up tables for each of the two pins, lookUpTable1 and lookUpTable2 and both have 200 values. The first half of lookUpTable1 has sin values from 0 to π and the second half is all zeroes. The first half of lookUpTable2 is all zeroes and the second half has sin values from 0 to π as shown in figure below.



The code assumes implementation on an Arduino Uno but is likely compatible with other boards. see [Compatibility](#).

Lets walk through the code.

```
#include <avr/io.h>
#include <avr/interrupt.h>

// Look up tables with 200 entries each, normalised to have max value of 1600 which
int lookUp1[] = {50 ,100 ,151 ,201 ,250 ,300 ,349 ,398 ,446 ,494 ,542 ,589 ,635 ,681
int lookUp2[] = {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
```

We are going to address the registers on the atmel chip as well as using interrupts so the `<avr/io.h>` and `<avr/interrupt.h>` headers are necessary. From there we have two arrays which have a two half sinusoidal signal entered in

```

void setup(){
  // Register initilisation, see datasheet for more detail.
  TCCR1A = 0b10100010;
  /*10 clear on match, set at BOTTOM for compA.
    10 clear on match, set at BOTTOM for compB.
    00
    10 WGM1 1:0 for waveform 15.
  */
  TCCR1B = 0b00011001;
  /*000
    11 WGM1 3:2 for waveform 15.
    001 no prescale on the counter.
  */
  TIMSK1 = 0b00000001;
  /*0000000
    1 TOV1 Flag interrupt enable.
  */
}

```

Here the timer register have been initialised. If you are interested in the details you can look up the ATMEGA328p datasheet, but for now what's important is that we have set up a PWM for two pins and it call in interrupt routine for every period of the PWM.

```

ICR1 = 1600; // Period for 16MHz crystal, for a switching frequency of 100
sei(); // Enable global interrupts.
DDRB = 0b00000110; // Set PB1 and PB2 as outputs.
pinMode(13,OUTPUT);
}

```

ICR1 is another register that contains the length of the counter before resetting, since we have no prescale on our clock, this defines the period of the PWM to 1600 clock cycles. Then we enable interrupts. Next the two pins are set as outputs, the reason why `pinMode()` is not used is because the pins might change on different arduinos, they might also change on different Atmel micros, however you are using an arduino with a 328p, then this code will work. Lastly `pinMode()` is used to set pin 13 as an output, we will use this later as a trigger for the oscilloscope, however it is not necessary.

```

void loop(){ /*Do nothing . . . forever!*/}

```

Nothing is implemented in the loop.

```

ISR(TIMER1_OVF_vect){
    static int num;
    static char trig;
    // change duty-cycle every period.
    OCR1A = lookUp1[num];
    OCR1B = lookUp2[num];

    if(++num >= 200){ // Pre-increment num then check it's below 200.
        num = 0;      // Reset num.
        trig = trig^0b00000001;
        digitalWrite(13,trig);
    }
}

```

This interrupt service routine is called every period of the PWM, and every period the duty-cycle is changed. This is done by changing the value in the registers OCR1A and OCR1B to the next value of the look up table as these registers hold the compare values that set the output pins low when reached as per figure .

Therefore in each period the registers OCR1x are loaded with the next value of their look up tables by using num to point to the next value in the array, as each period num is incremented and checked that it is below 200, if it is not below 200 it is reset to 0. The other two lines involving trig and digitalWrite are there to toggle a pin as a trigger for an oscilloscope and does not impact the sPWM code.

sPWM_generate_lookup_table

The rest of this section discusses some modifications to this code, namely we can make generate the lookup table at the start of the code, the benefit of this is we change the switching frequency as well as the sPWM frequency. Code for this can be found in the sPWM_generate_lookup_table folder. The start of the code looks like this:

```

#include <avr/io.h>
#include <avr/interrupt.h>

#define SinDivisions (200) // Sub divisions of sinusoidal wave.

static int microMHz = 16; // Micro clock frequency
static int freq = 50;     // Sinusoidal frequency
static long int period;   // Period of PWM in clock cycles.
static unsigned int lookUp[SinDivisions];
static char theTCCR1A = 0b10000010; //variable for TCCR1A

void setup(){

```



```
double temp; //Double variable for <math.h> functions.

period = microMHz*1e6/freq/SinDivisions;// Period of PWM in clock cycles

for(int i = 0; i < SinDivisions/2; i++){ // Generating the look up table.
    temp = sin(i*2*M_PI/SinDivisions)*period;
    lookUp[i] = (int)(temp+0.5);        // Round to integer.
}
```

Notice that only the first half of the sine wave is generated, because of the way this code implements the sPWM where each of the two pins are responsible for different halves of the signal, only half the sine wave is needed. However it does require a modification to the interrupt service routine.

```
ISR(TIMER1_OVF_vect){
    static int num;
    static int delay1;
    static char trig;

    if(delay1 == 1){/*delay by one period because the high time loaded into OCR1A:B
        theTCCR1A ^= 0b10100000;// Toggle connect and disconnect of compare output A a
        TCCR1A = theTCCR1A;
        delay1 = 0;                // Reset delay1
    } else if(num >= SinDivisions/2){
        num = 0;                    // Reset num
        delay1++;
        trig ^=0b00000001;
        digitalWrite(13,trig);
    }
    // change duty-cycle every period.
    OCR1A = lookUp[num];
    OCR1B = lookUp[num];
    num++;
}
```

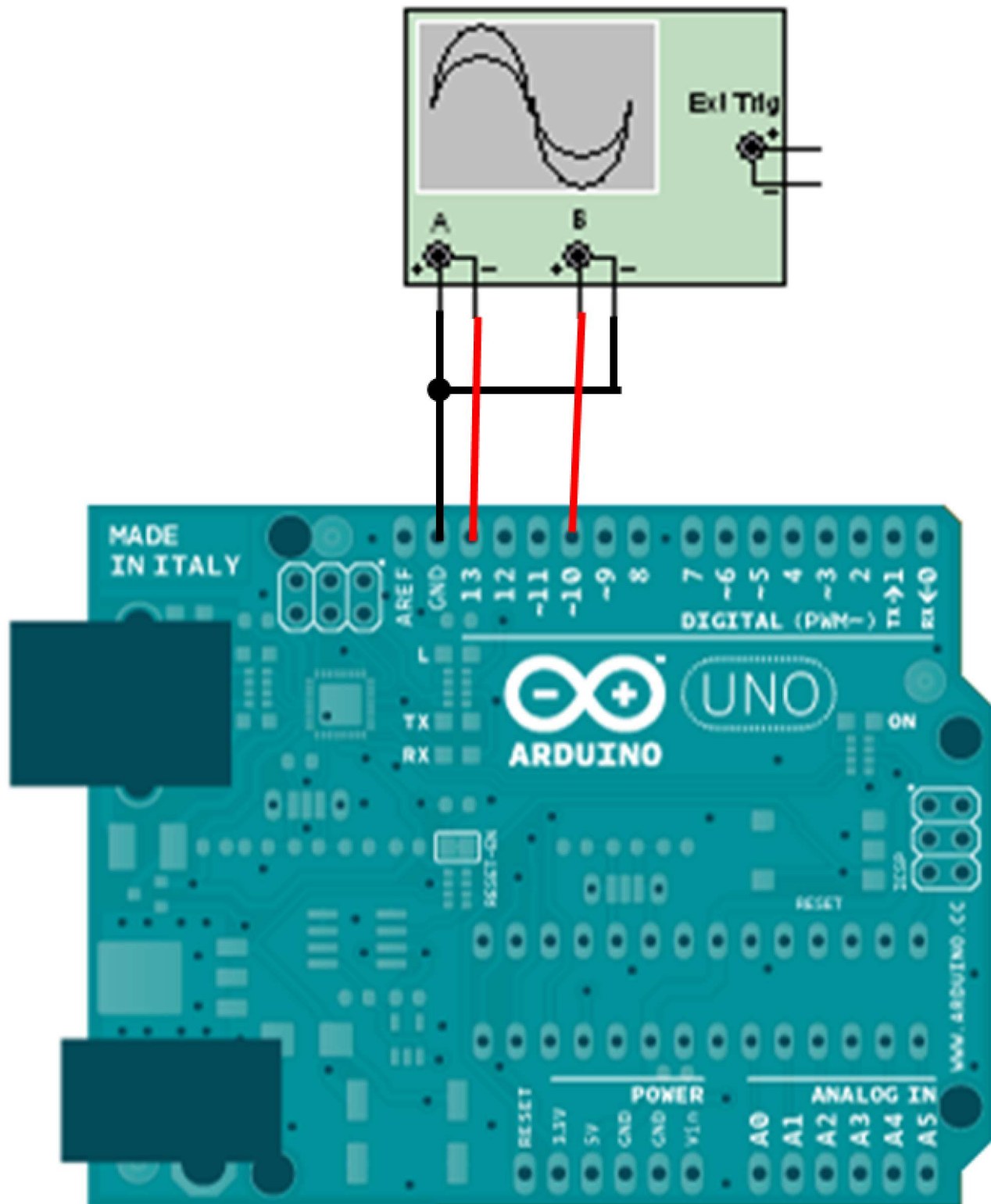
Both output compare registers ORC1x have the same values loaded into them each period, however the output compare for each pin is enabled and disabled in turns by toggling two bits in the TCCR1A register. It is toggle each time the look up table index (num) is reset, however it is delayed by one clock cycle, this is because when values are loaded into OCR1x registered, is buffered whereas changes in TCCR1A is implemented immediately, see the 328p datasheet for more details.

Testing the Signal

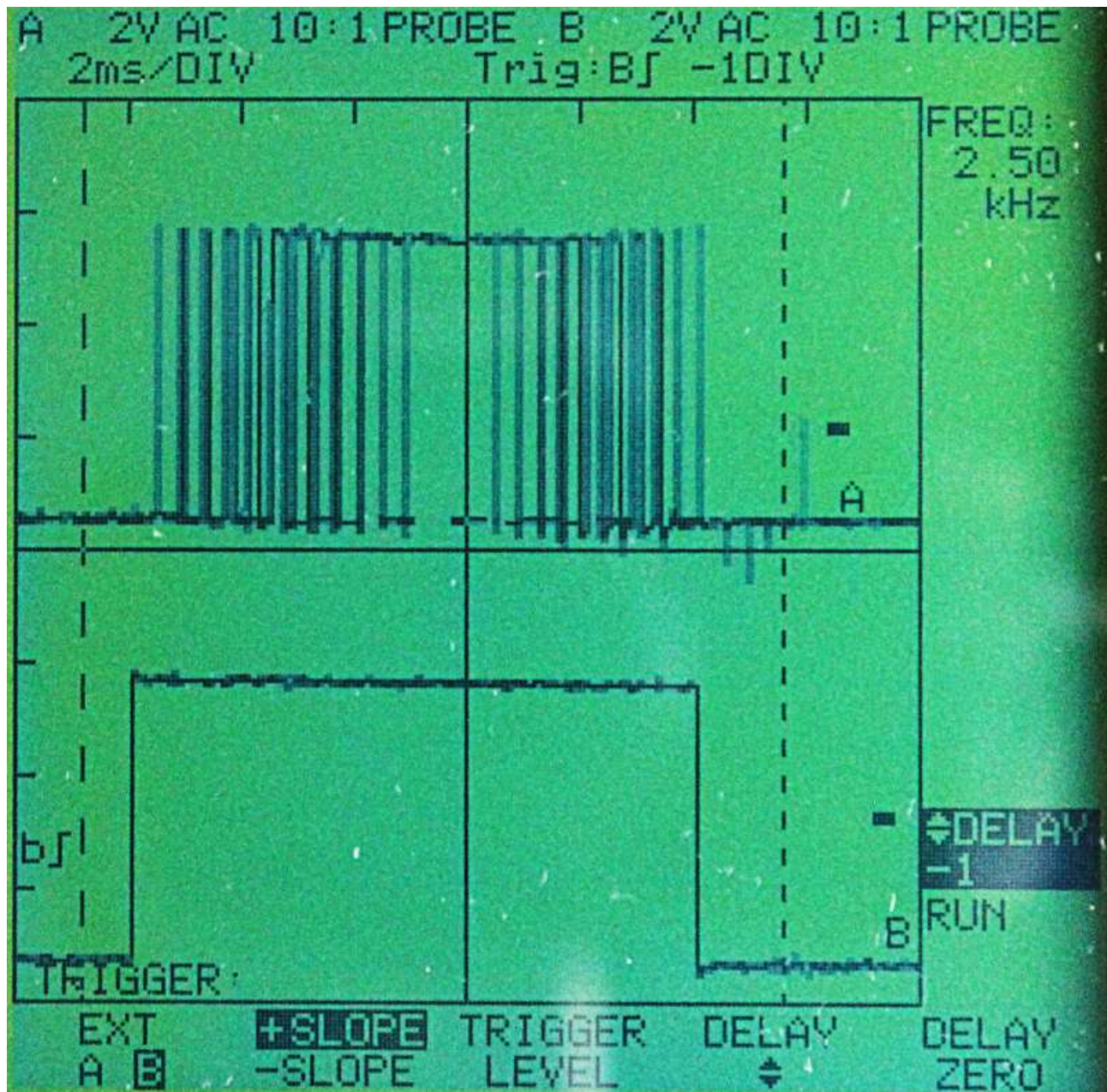
This chapter discusses ways to test and monitor the sPWM signal, the first section discusses using an oscilloscope which is the best method to verify the signal however an alternate and cheaper method is to use a small speaker. The 50hz signal and the underlying switching frequency is easy to hear.

Viewing Pulse Widths with an Oscilloscope

Here we aim to view the individual pulses that make up the sPWM to see if they look how we would expect, that is thin pulses becoming thick and then thin again. If you are using the code `sPWM_generate_lookup_table` I recommend lowering the switching frequency of the sPWM by changing `#define SinDivisions(200)` to `#define SinDivisions (50)`. This will make the pulses easier to see as there will be fewer of them. If you are using the Uno pin 13 will be toggled every period of the sine and so can be used as a trigger for the oscilloscope. the figure below shows the wiring to the Arduino Uno.

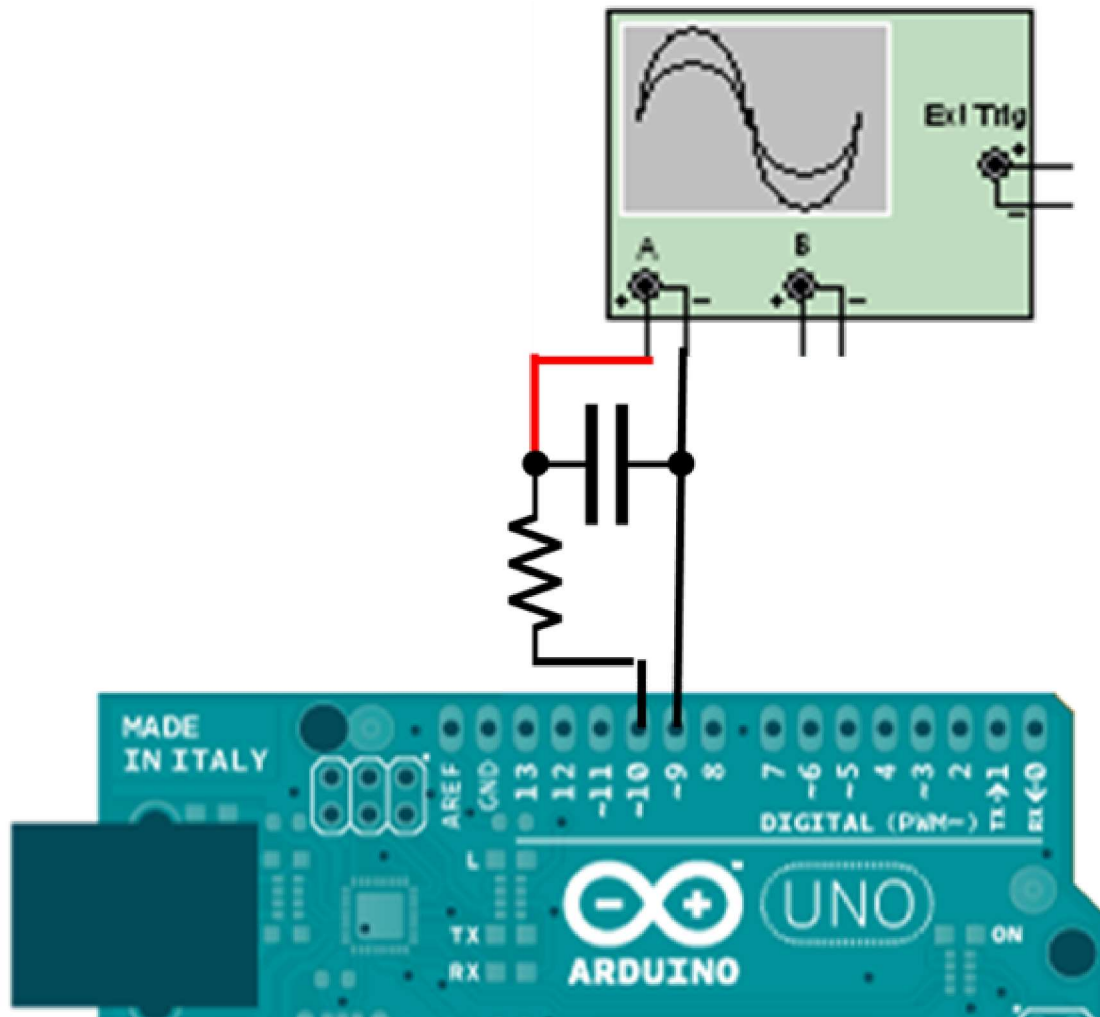


Notice we are only looking at half the signal from one of the output pins. The wave form produced should look like the figure below. Notice the change from thin to thick back to thin pulses.

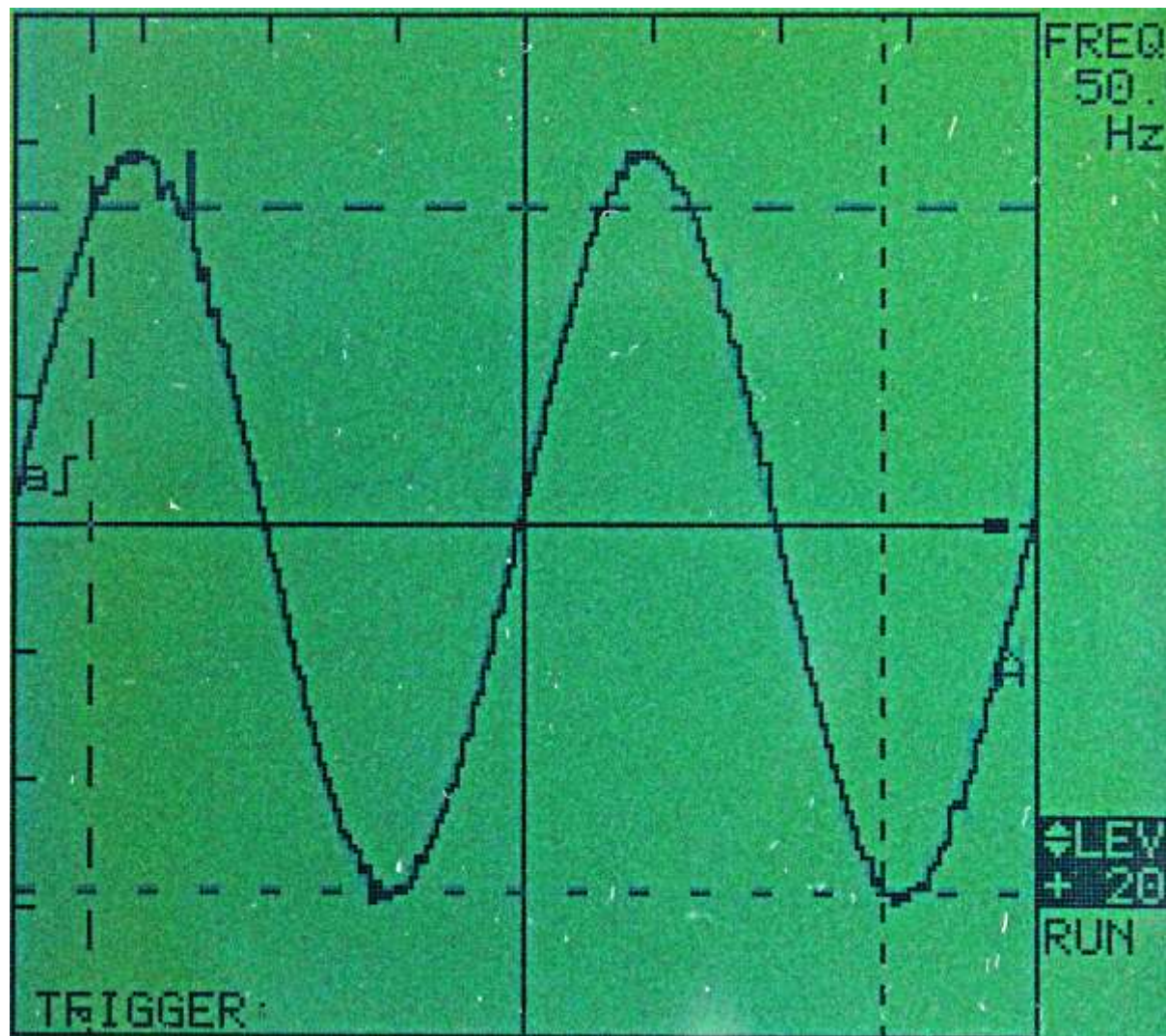


Viewing the Filtered Signal with an Oscilloscope

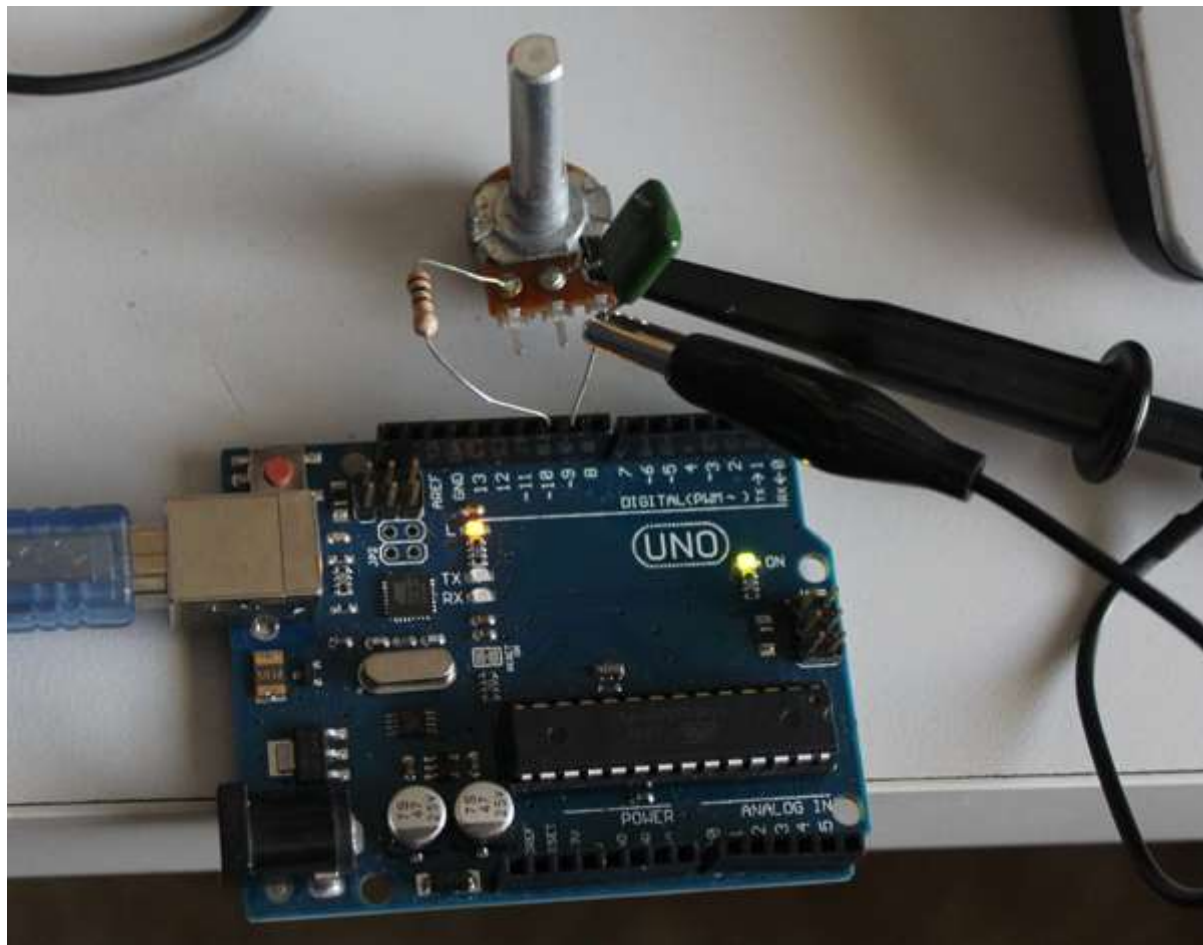
Here we aim to view a smooth sinusoidal wave by smoothing the output of the micro with a simple low-pass RC filter as shown in Figure 4.3.



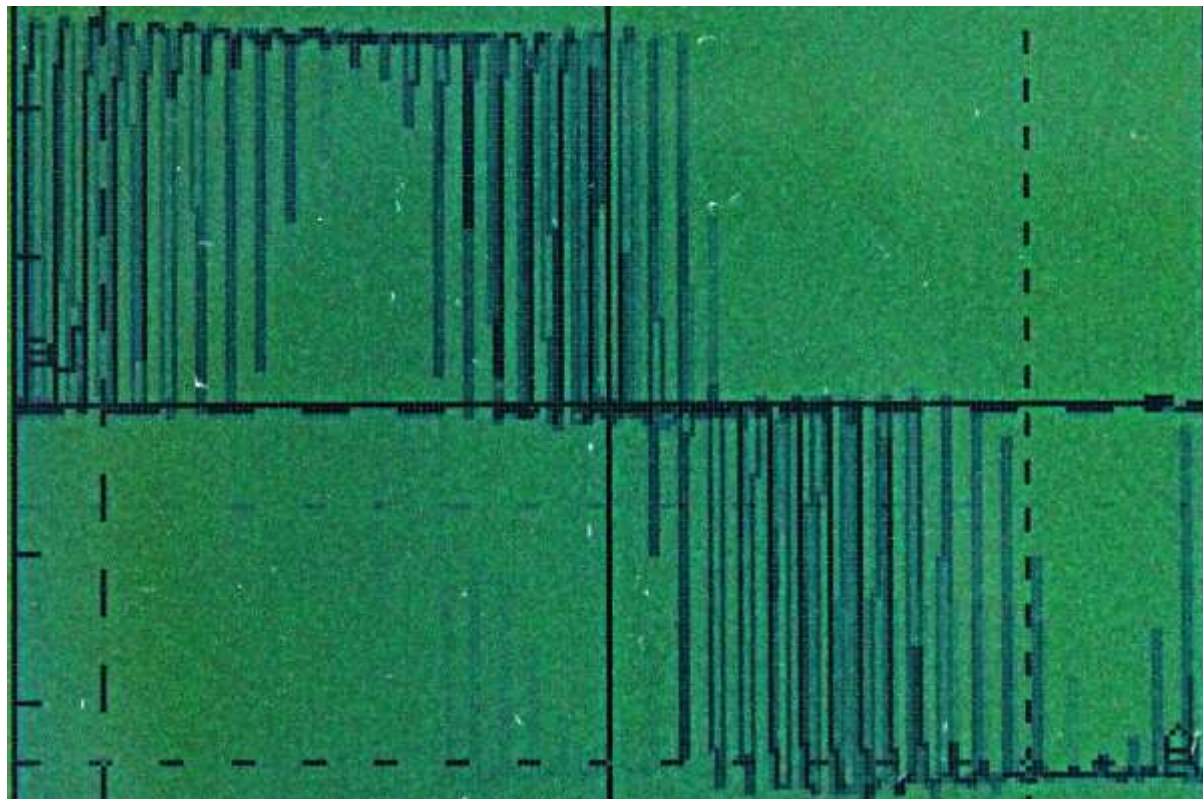
The purpose of the filter is to remove the much higher switching frequency (f_s) and leave only the 50Hz sine wave. The cut-off frequency (f_c) of the filter determined is by $f_c = 1/2\pi RC$, and it should lie somewhere between f_s and 50Hz, closer to 50Hz is more optimal. f_s is determined by the line of code `#define SinDivisions (a number)` in `sPWM_generate_lookup_table`. The switching frequency is given by $f_s = \text{SinDivisions} \times 50\text{Hz}$. Using an arbitrary number for SinDivisions may produce unexpected results, I recommend using 50, 200 and 400 to produced f_s 's of 2.5, 10 and 20 kHz respectively. The filtered signal is shown in the figure below.

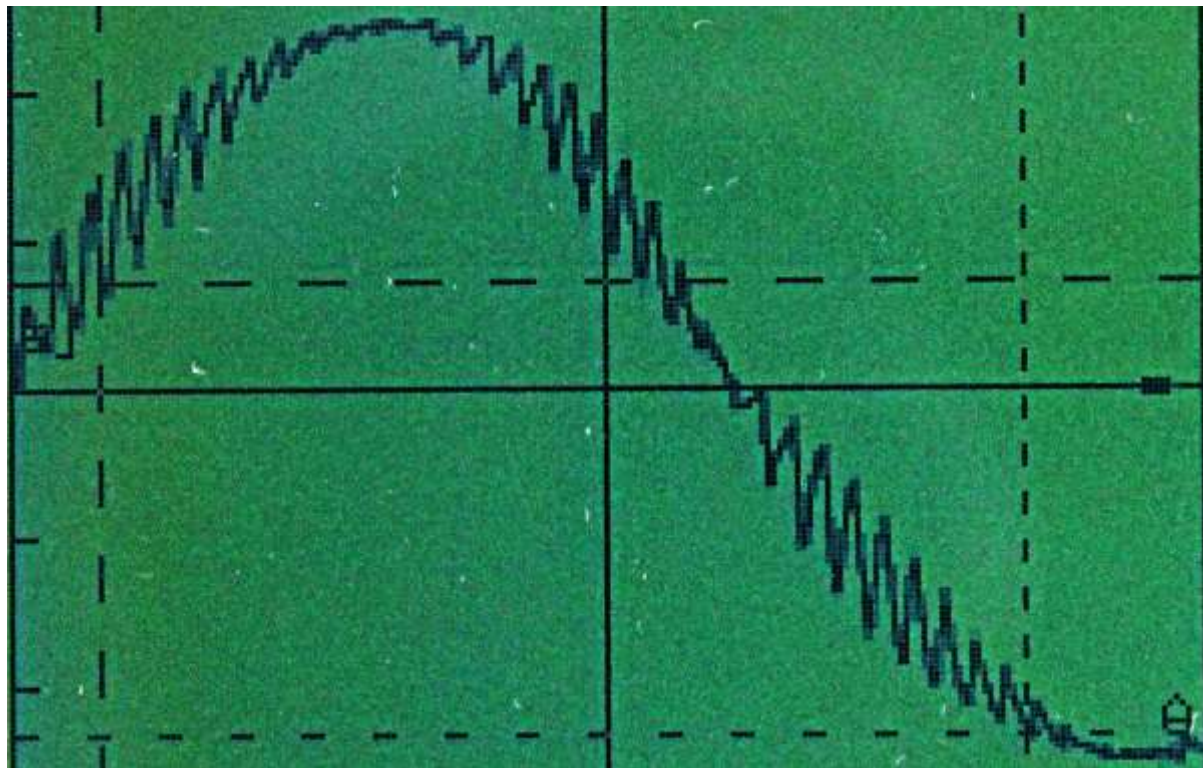
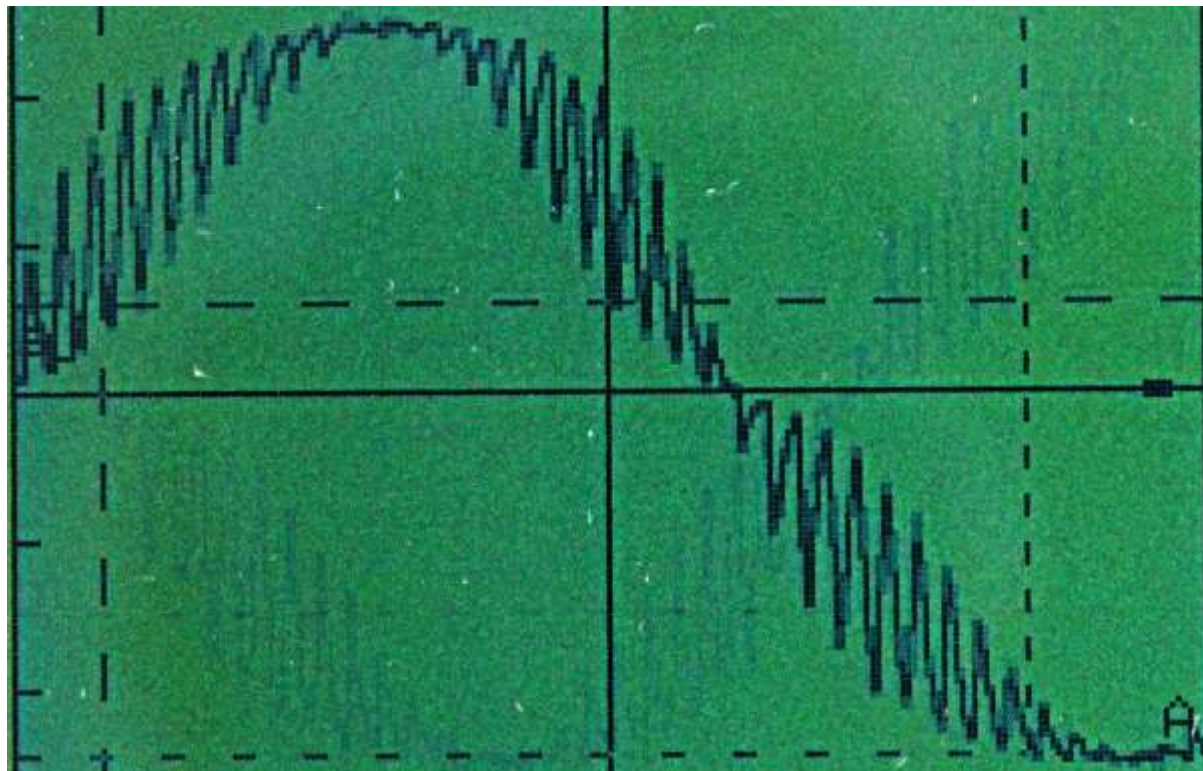


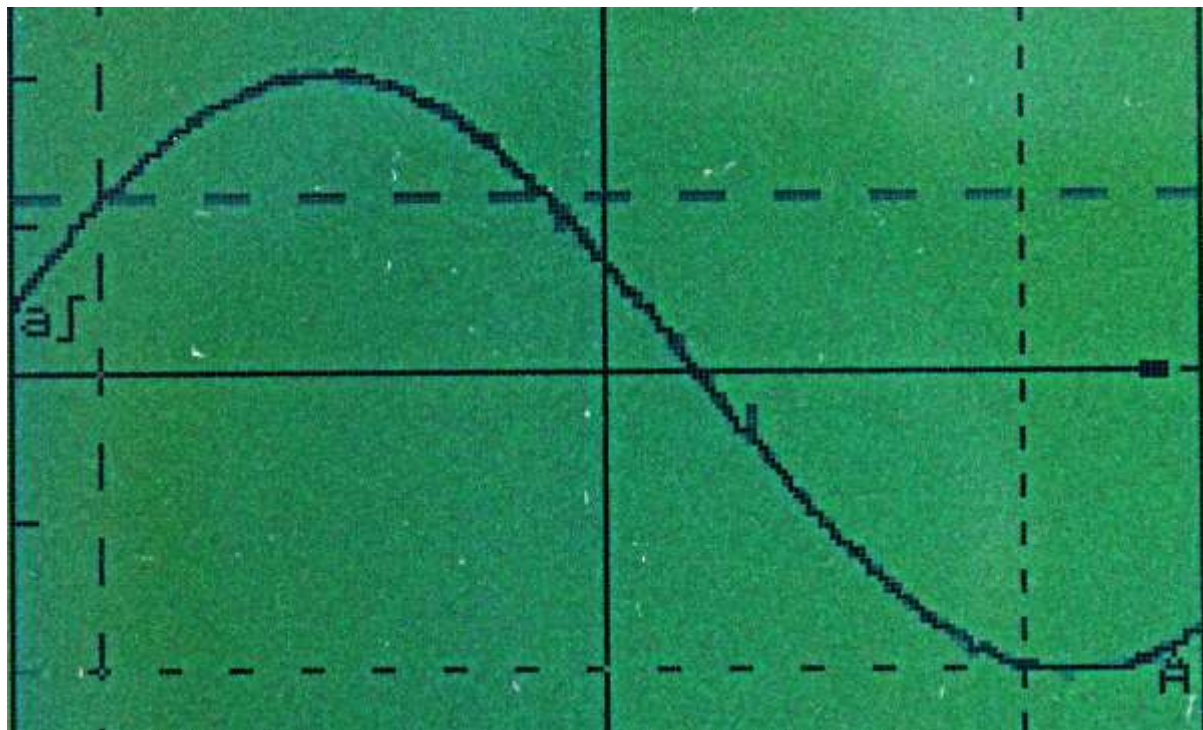
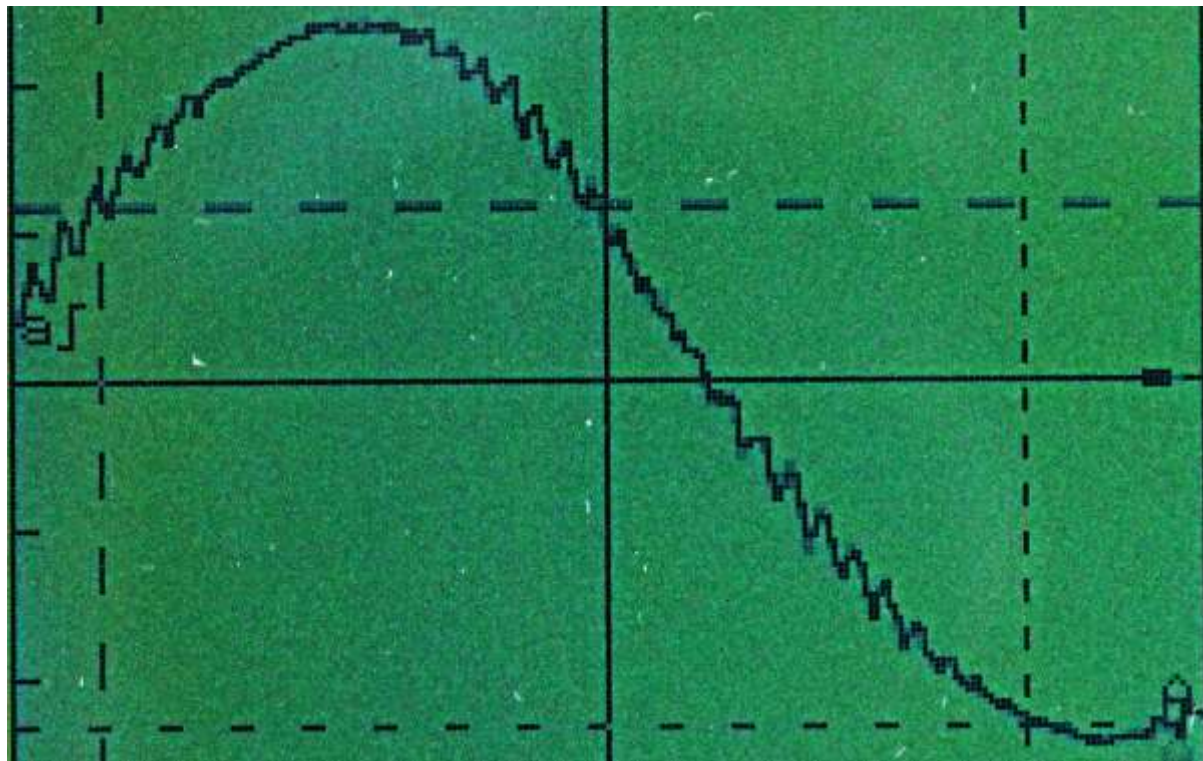
Alternatively a potentiometer can be used as a variable resistor as shown in the figure below.



This allows the f_c to be varied. The following five figures show the result of changing the potentiometer and therefore the amount of filtering.

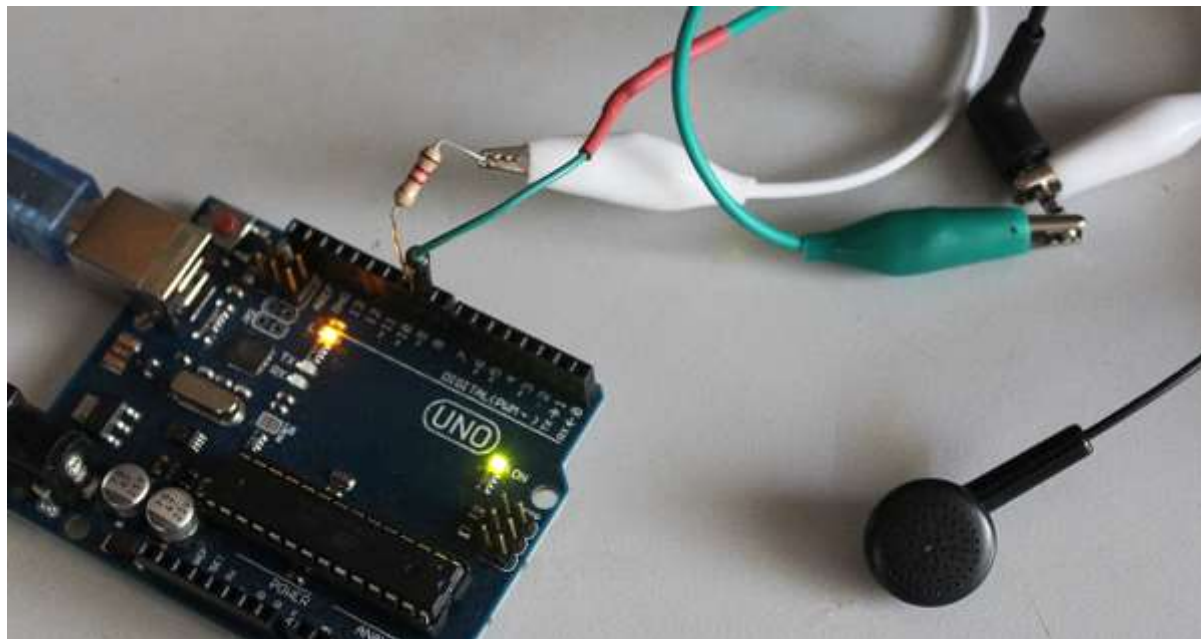






Listening to the Signal

If you don't have an oscilloscope, listening to the signal is a useful way to determine if the sPWM is working since it is easy to hear both the 50Hz hum and the switching frequency. We can use the micro to drive a small pair of head-phones directly, putting them in series with a 1K Ω resistor should protect most head-phones, as seen in Figure 4.6.



It is recommended to change `#define SinDivisions (200)` to 50, 200 and 400 in order to hear the difference in switching frequencies. Note depending on your age and hearing you might not be able to hear the switching frequency with `#define SinDivisions (400)`, as this produces a switching frequency of 20KHz, which is on the limit of human hearing.

Compatibility

Please let me know if you got the code to work on a device that's not listed here

Comparability list:

- Arduino Uno
- Arduino Nano
- Arduino mega2560

Safety

This section is to briefly discuss safety in regards to making an inverter that steps up to mains voltage whether that be 110 or 230. First of all, I don't encourage it, I'd prefer that you didn't and I take no responsibility for your actions. Remember 30mA can be lethal, mains voltage deserves respect.

If you still choose to do so, take basic precautionary steps like: Invest in some terminals and make sure that any high voltage part of the circuit is not touchable; don't modify it while it's power up; Don't do it alone.

Releases 1

 Initial Release

Latest

on Nov 14, 2015

Packages

No packages published

Contributors 6



Languages

C++ 100.0%