Singing Tesla Coil - Building a Musically Modulated Lightning Machine

Sam Redmond - Applied Science Research
Advisor: Dr. James Dann
Menlo School, 50 Valparaiso Avenue, CA 94027

September 5, 2014

Abstract

A musically modulated Tesla coil was developed and tested but never made fully functional. Music data is sent from an input MIDI stream, either from preexisting MIDI files (transferred from a computer) or from a MIDI-enabled piano keyboard to a microcontroller. The microcontroller then processes incoming MIDI signals and translates them into a series of electrical pulses. Software was created for the Arduino Due (a microcontroller) that allows up to nine concurrent notes to be played. The output pulses control whether the Tesla coil sparks or not. The variable frequency of sparking produces the perception of sound.

Contents

List of Figures

List of Tables

1 Introduction

Tesla coils are inherently fascinating. They mix physics, danger, and lightning. What's not to like? Adding music to the mix not only makes the project more challenging but also adds novelty –while many hobbyists have made Tesla coils, only ArcAttack, oneTesla, GeekGroup, and a few other YouTubers have made singing Tesla coils. [1] [2] [3] Only oneTesla and ArcAttack have controlled a Tesla coil with an instrument; even so, they have only been able to play two notes simultaneously. Our system supports up to nine concurrent notes.

This project doesn't have much relevance to the world at all. At no point will society desperately need MIDI-controlled lightning machines in the same way that it will need sustainable energy sources or a more efficient light bulb. However, entertainment is always in demand, and nothing draws a crowd quite like Pachelbel's Canon played on the Tesla coil.

1.1 History

1.1.1 Nikola Tesla

Nikola Tesla, the inventor of the Tesla coil, lived a dramatic life. Born in 1856 in what is now Croatia (then Serbia), Tesla studied math and physics at the University of Prague. Legend holds that Tesla, the brilliant student, came up with the idea for a brushless AC motor while on a walk, sketching the schematic in the sand with a stick.

In 1884, Nikola Tesla moved in the United States and soon found a job as an electrical engineer in Thomas Edison's headquarters. After seeing how smart Tesla was, Edison offered Tesla $50,000 – equivalent to roughly $1.1 million today – to redesign Edison's inefficient DC power generators. After Tesla worked tirelessly for months to find a solution, Edison laughed off his previous offer and refused to pay Tesla the money, claiming "Tesla, you don't understand our American humor." Tesla left Edison's company.

On his own, Tesla attempted to create the Tesla Electric Light company, but failed and resorted to digging ditches for $2 a day for a brief period. Later, Tesla found a few investors for his research on alternating current. In 1887 and 1888, Tesla was awarded over 40 patents for his work on AC, and spoke at a conference, attracting the attention of George Westinghouse. Westinghouse was a huge proponent of the AC system, and had recently set up the first AC power system in Boston. Westinghouse hired Tesla, bought his patents, and gave him a research lab. However, Tesla's and Westinghouse's AC was a direct competitor to Edison's DC. Therefore, the greedy, power-hungry Edison began to publicly denounce the AC system. In 1889, a convict was killed by electrocution using AC current, overseen by Edison. Edison electrocuted cats, dogs, calves, horses, and an orangutan with AC current to show the public just how dangerous AC could be. Perhaps Edison's greatest act of cruelty came in 1903, when he electrocuted Topsy the 6-ton circus elephant with 6000 volts AC in front of a group of 1500 spectators. [4] [5] (Warning: the videos are gruesome with regards to animal cruelty)

Ultimately, though, AC power won out. For one, lightbulbs that ran on AC current were bright white, compared to Edison's dull yellow DC bulbs. The biggest reason that AC was preferred over DC involves the transfer of electricity. AC power lines can run at much higher voltages than DC power lines, which allows for a lower current to transfer the same amount of power ($P = IV$). The lower current of AC power means that less power is dissipated in the transfer cables (since $P = I^2R$), so cables can be longer. Whereas DC power required substations every 2 or so miles, AC power lines could run for hundreds of miles with negligible power losses.

While Westinghouse went off to supply the nation with AC power, Tesla continued to invent. In 1893, he demonstrated the AC system at the World Columbian Exposition in Chicago, and in 1895 he built a hydroelectric power plant at Niagara Falls. Tesla lost many notes and machines when his lab burned down in 1895 and began moving around the country, staying in Colorado Springs before moving to New York. Tesla got funding from J.P. Morgan to build a worldwide communications network with a central hub at Tesla's Wardenclyffe lab on Long Island. However, Morgan pulled the funding, and Wardenclyffe was destroyed.

Tesla's last years were spent in the New Yorker Hotel, feeding (and 'communicating with') the city pigeons. He became obsessed with the number three and abhorred germs, causing historians to question his mental health. Nikola Tesla died on January 7, 1943, alone and with significant debts. His work on dynamos, electric motors, radar, X-rays, and magnetic fields helped to inspire the next generation of inventors. Modern-day radio transmission relies on principles that Tesla discovered. Even though Guglielmo Marconi is often considered the 'father of the radio,' Tesla's research predates Marconi's and in 1943, the Supreme Court voided four of Marconi's patents, giving due credit to Tesla. Perhaps most importantly, Tesla's AC power system is the worldwide standard today. [6] [7] [8]

1.1.2 The Tesla Coil

In 1891, Nikola Tesla invented one of his most famous devices, the Tesla coil. To be fair, electrical coils existed before Nikola Tesla. Ruhmkorff coils, named after Heinrich Ruhmkorff and designed by Nicholas Callan in 1836, converted low voltage DC to high voltage DC spikes. [9] Tesla began working with these transformers, but soon switched to devices of his own invention. Tesla's capacitors consisted of metal plates immersed in oil, and his spark gap used two metal balls and an air jet to clear ionized air, creating more abrupt discharges. The secondary coil was comprised of two inductors, one to couple the fields and one 'resonator.' Later, Tesla developed higher voltage power transformers, and used a rotating spark gap to short the LC circuit. Voltage gain was significantly increased when Tesla began loosely, rather than tightly, coupling the two inductors, using air as the core rather than metal. Tesla's patent was for a 'high-voltage, air-core, self-regenerative resonant transformer that generates very high voltages at high frequency.'

Modern Tesla coils have not changed much from Tesla's ultimate design. Modern designers have simplified the circuit, removing the helical resonator that previously was connected in series with the secondary coil. Since Tesla's original goal was power transmission, his top conductors often had a large radius of curvature to minimize any electrical losses from corona effects or discharges to the secondary coil or surrounding objects. Modern builders emphasize long sparks more than efficient power transmission, so modern Tesla coils tend to use toroidal top conductors rather than Tesla's original spherical conductors. [10]

1.1.3 Modern Musical Tesla Coils

Musical Tesla coils were first seen in public demonstrations in March 2006 by Joe DiPrima and Oliver Greaves. In 2007, he renamed his group ArcAttack, a musical performance group which continues to showcase musical Tesla coils today. [11] Steve Ward and Jeff Larson were also early popularizers of the singing Tesla coil, alternatively called a Zeusaphone (a play on words on the tuba-like instrument, the Sousaphone, using Zeus, the Greek god of lightning) or a Thoremin (a play on words combining 'theremin,' the electronic musical instrument, with Thor, the Norse god of thunder. [12] Ward's original blog post discusses the physics and engineering challenges of constructing a musical Tesla coil. [13]

Since then a number of alternate musical Tesla coil groups have sprung up, most notably oneTesla, a group out of MIT that sells musical Tesla coil kits. The company was created after a wildly successful Kickstarter campaign. [14] The oneTesla kit, however, only allows for a small Tesla coil with support for only two concurrent notes. Nevertheless, communities such as those at oneTesla and the high voltage enthusisasts at 4HV have helped boost the popularity of musical Tesla coils. [15]

My Interest

Personally, I can't imagine a better project. It involves a significant amount of programming and electrical engineering, but also has some elements of mechanical engineering. I love music too, and MIDI blends music and electronics. I also enjoy the danger associated with these extremely high voltages.

2 Theory

This section is a theoretical discussion of physics concepts from electromagnetism, Tesla coils, Arduino features, MIDI, and miscellaneous circuit elements.

2.1 Physics Background

2.1.1 Maxwell's Equations

As with all electromagnetic systems, we begin with Maxwell's equations, the fundamental equations that unify electricity and magnetism.

In order, they are Gauss's Law (1 and 5), Gauss's Law for Magnetism (2 and 6), Faraday's Law of Induction (3 and 7), and Ampère's Circuital Law (4 and 8) with a correction by Maxwell, shown first in their integral form and second in their differential form.

$$\oiint_{\partial\Omega} \vec{E}\cdot \mathrm{d}\vec{S} = \frac{1}{\varepsilon_0}\iiint_{\Omega} \rho \mathrm{d}V = \frac{q_{encl}}{\varepsilon_0} \tag{1}$$

$$\oiint_{\partial\Omega} \vec{B}\cdot \mathrm{d}\vec{S} = 0 \tag{2}$$

$$\oint_{\partial\Sigma} \vec{E}\cdot \mathrm{d}\boldsymbol{\ell} = -\frac{d}{dt}\iint_{\Sigma} \vec{B}\cdot \mathrm{d}\vec{S} \tag{3}$$

$$\oint_{\partial\Sigma} \vec{B}\cdot \mathrm{d}\boldsymbol{\ell} = \mu_0 \iint_{\Sigma} \left(\vec{J} + \varepsilon_0 \frac{\partial \vec{E}}{\partial t}\right)\cdot \mathrm{d}\vec{S} \tag{4}$$

$$\nabla \cdot \vec{E} = \frac{\rho}{\varepsilon_0} \tag{5}$$

$$\nabla \cdot \vec{B} = 0 \tag{6}$$

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \tag{7}$$

$$\nabla \times \vec{B} = \mu_0 \left(\vec{J} + \varepsilon_0 \frac{\partial \vec{E}}{\partial t}\right) \tag{8}$$

In these equations, μ_0 is the permeability of free space and ε_0 is the permittivity of free space. Both are universal constants ($\mu_0 \equiv \frac{10^{-7}}{4\pi}\frac{V\cdot s}{A\cdot m}$ and $\varepsilon_0 \approx 8.8542 * 10^{-12}\frac{F}{m}$), related by $c = \frac{1}{\sqrt{\varepsilon_0 \mu_0}}$, where c, the speed of light in free space, is defined to be $299,792,458\frac{m}{s}$. The concepts of relative permeability and relative permittivity in a material can be used to define the speed of light in a material.

$\vec{E}$ is the electric field at a given point in space and $\vec{B}$ is the magnetic field at a given point in space.

∇ is a mathematical operator on functions defined on $\mathbb{R}^n$. In 3D, this operator can be thought of as a 3-vector of partial derivatives, $\nabla = \langle \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\rangle$. For vector-valued functions $f = \langle P(x,y,z), Q(x,y,z), R(x,y,z)\rangle : \mathbb{R}^3 \mapsto \mathbb{R}^3$ where P, Q, and R are real-valued functions from $\mathbb{R}^3$ to $\mathbb{R}$, $div\vec{f} = \nabla \cdot f = \frac{\partial P}{\partial x} + \frac{\partial Q}{\partial y} + \frac{\partial R}{\partial z}$ is the divergence of f and $\nabla \times f = \langle \frac{\partial R}{\partial y} - \frac{\partial Q}{\partial z}, \frac{\partial P}{\partial z} - \frac{\partial R}{\partial x}, \frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}\rangle$ is the curl of f. Note that divergence is real-valued and curl is vector-valued. Divergence can be thought of as flux per unit volume at a point. The curl is usually only applied to 3-dimensional functions and measures rotation at a point. The vector-valued functions we're dealing with are $\vec{E} : \mathbb{R}^3 \mapsto \mathbb{R}^3$ and $\vec{B} : \mathbb{R}^3 \mapsto \mathbb{R}^3$.

Ω is a closed volume with bounding surface $\partial\Omega$ and locally planar differential surface $\mathrm{d}\vec{S}$, directed perpendicularly out of the surface and with magnitude equal to the area of the plane. Σ is an open surface with bounding contour $\partial\Sigma$ and locally linear differential line $\mathrm{d}\boldsymbol{\ell}$ parallel to $\partial\Sigma$.

The quantity ρ is the charge density at a point in space, in units of $\frac{C}{m^3}$, related to total charge enclosed in a surface by $Q_{encl} = \iiint_\Omega \rho \mathrm{d}V$. $\vec{J}$ is current density, in units of $\frac{A}{m^2}$, related to net current though a surface by $I_{encl} = \iint_\Sigma \vec{J} \cdot \mathrm{d}\vec{S}$.

Why do we bother with both forms of the equations? The integral forms describe electromagnetic behavior in a global region of space, whereas the differential forms describe local electromagnetic behavior at a point.

We'll focus on Faraday's Law of Induction and Ampère's Circuit Law. In English, Faraday's law states that a time-varying magnetic field will induce an electric field, and Ampére's law states that a magnetic field is induced not only by an electrical current, but also by a time-varying electric field.

2.1.2 More on Ampère's Law

We'll ignore Maxwell's correction for now and focus on the main component of Ampere's law that approximately posits that a current will induce a magnetic field. (See Figure 1, bottom-left.) For convenience, we'll rewrite Ampère's Law as:

$$\oint_{\partial\Sigma} \vec{B} \cdot \mathrm{d}\boldsymbol{\ell} = \mu_0 I_{encl} \tag{9}$$

where the variables are the same as before, with the exception that I_{encl} refers to the total current enclosed by Σ.

Consider an infinitely long, straight wire oriented in the Z direction, carrying a current of I. (Note: If the wire were finitely long, we would have to apply Maxwell's correction to Ampère's Law to obtain a rigorous result.) Let a distance r be given. Define Σ as the disk $x^2 + y^2 < r^2$. Then $\partial\Sigma$ is the circle $x^2 + y^2 = r^2$. The current enclosed by Σ is simply I (since the wire pierces our Ampèrian surface), so the right side of Equation 9 simplifies to $\mu_0 I$. By symmetry, the magnitude of the magnetic field $\|\vec{B}\|$ is the same at all points on $\partial\Sigma$, and the angle θ between $\vec{B}$ and dℓ is constant, so we find:

$$\begin{aligned}\oint_{\partial\Sigma} \vec{B} \cdot \mathrm{d}\ell &= \|B\| cos\theta \oint_{\partial\Sigma} \mathrm{d}\ell = (\|B\| cos\theta)(2\pi r) = \mu_0 I_{encl} \\ \|B\| cos\theta &= \frac{\mu_0 I}{2\pi r}\end{aligned}$$

From Gauss's Law for Magnetism, $\vec{B}$ is a conservative field, so $\vec{B}$ must be tangent to our path of integration (so that, when dotted with the normal, the integrand becomes zero). Therefore, $\theta = 0$ and $cos(\theta) = 1$, so we get our first result: At a distance r from a line of current, the magnitude of the magnetic field is $\frac{\mu_0 I}{2\pi r}$. Its direction is arbitrarily designated (definitionally) by the right hand rule: with the thumb of the right hand aligned with the direction of the current, the magnetic field follows the direction that the fingers naturally curl.

Our next goal is to determine the magnetic field inside a solenoid. Consider an ideal

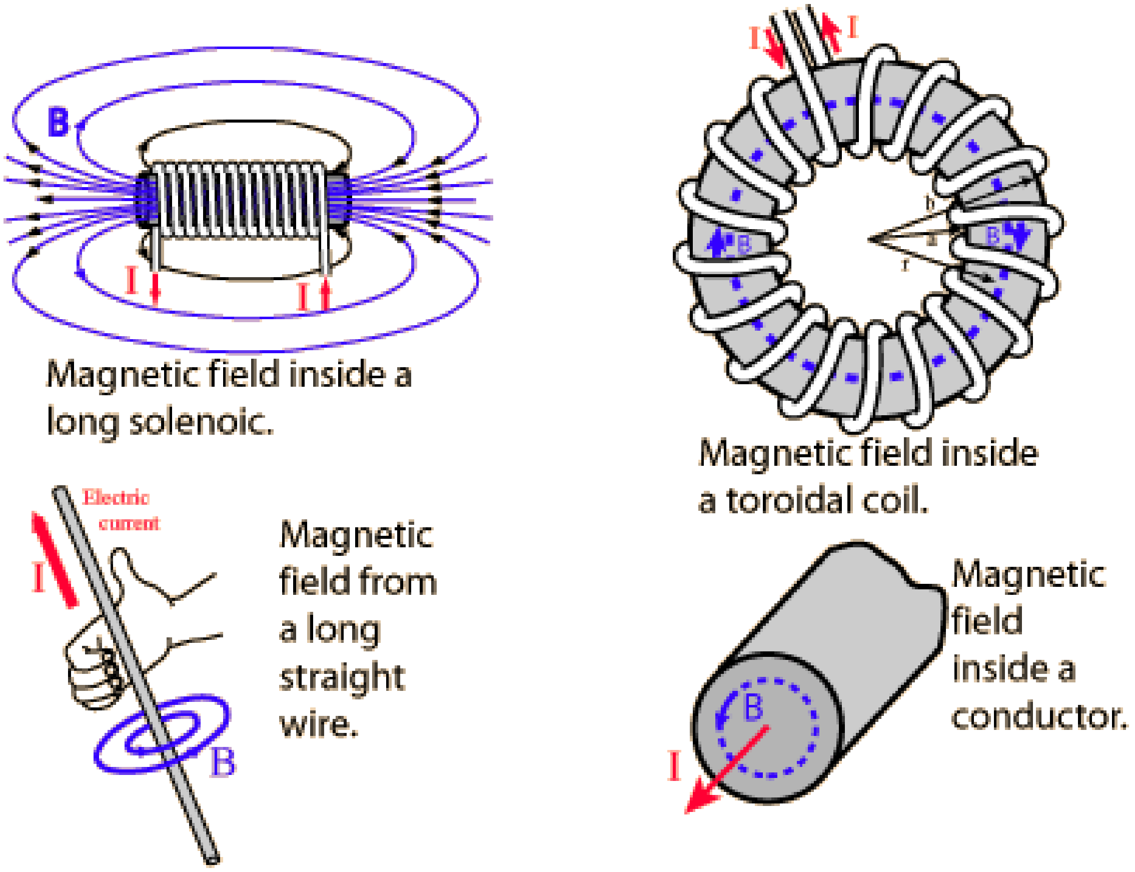

B
I
I
Magnetic field inside a long solenoic.
I
I
b
a
r
B
B
Magnetic field inside a toroidal coil.
Electric current
I
B
Magnetic field from a long straight wire.
Magnetic field inside a conductor.
B
I

Figure 1: Visualizations of the magnetic field created by a current-carrying wire in common configurations. [16]

infinitely long solenoid: a helical coil of wire with turn density n (units of $\frac{turns}{m}$), current I, wrapped around an (infinitely long) imaginary cylinder of radius r (see Figure 1, upper left). Without loss of generality, let the axis of the cylinder be the z-axis (given by $\hat{k} =< 0, 0, 1 >$), and let the current be flowing in the positive z direction (parallel to $\hat{k}$). By symmetry and by applying the right hand rule repeatedly, we argue that the magnetic field on the interior of the solenoid is parallel to $\hat{k}$ and that the magnetic field on the outside of the solenoid is parallel to $-\hat{k}$.

Consider a positively-oriented Ampèrian square ABCD in the x-z plane height h and width w. Let AB and CD be parallel to the x-axis, let BC be parallel to the z-axis and outside the cylinder, and let DA be parallel to the z-axis and inside the cylinder. Since AB and CD are perpendicular to the magnetic field both inside and outside of the solenoid, their contributions to the loop integral drop out, since the dot product of $\vec{B}$ and the tangent line is zero. The contribution from BC is also zero, since the magnetic field outside of this ideal solenoid is zero. This is because all magnetic field lines must form complete loops (Gauss's Law for Magnetism), and while the ratio of volume outside the cylinder to volume inside the cylinder increases without bound, the magnetic field lines are effectively spread through an infinite region, and are thus 'diluted' to zero strength. Therefore, the only nonzero component of the loop integral occurs on the interior segment (DA). Additionally, the vector dot product $\vec{B} \cdot d\boldsymbol{\ell}$ becomes the algebraic product $B\,dl$ along this segment because the tangent line and magnetic field lines are parallel. The net current I_{encl} enclosed by this rectangular region is given by hnI, since there are hn points where a wire with current I pierces the region.

Applying Ampère's Law, we find that:

$$\oint_{ABCD} \vec{B} \cdot d\boldsymbol{\ell} = \int_D^A B\,dl = B \int_D^A dl = Bh = \mu_0 hnI = \mu_0 I_{encl}$$
$$B = \mu_0 nI$$

The direction is given by a variation on the right hand rule: with the fingers curling in the direction of current flow in the solenoid, the thumb points along the direction of the magnetic field.

2.1.3 More on Faraday's Law

Faraday's Law is all about how a changing magnetic field can induce an electric field. Again, we rewrite Faraday's Law to intuitively understand it better:

$$\mathscr{E} = -\frac{d\Phi_B}{dt}$$

where $\mathscr{E}$ is the induced electromotive force (emf) and Φ_B is the magnetic flux through some Gaussian surface. If a magnetic field is established inside a solenoid by an external source, the solenoid will experience an induced voltage. This principle is key in transferring energy from the primary circuit to the secondary circuit in a Tesla coil.

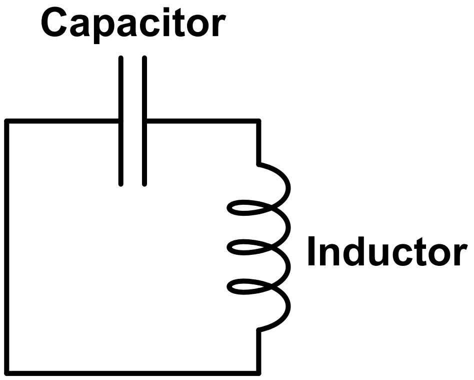


Figure 2: Undriven LC circuit

The negative sign in front of $-\frac{d\Phi_B}{dt}$ means that the induced voltage opposes the change in magnetic field. This principle is called Lenz's Law, and means that an inductor opposes the flow of current through it (since current would establish a magnetic field).

2.1.4 Inductors

Inductors have a property known as inductance, commonly given the symbol L, which characterizes this opposition to current. Mathematically,

$$\mathscr{E} = -L\frac{di}{dt}$$

In this case, $\mathscr{E}$ is often referred to as the backemf.

2.1.5 LC Circuits

Consider the undriven LC circuit shown in Figure 2. The capacitor has capacitance C, and the inductor has inductance L. Consider what happens when the capacitor is fully charged. At first, the inductor poses no backemf, so current discharges through the capacitor. However, as the magnitude of current flowing through the inductor increases, the magnetic field through the inductor increases as well, so the inductor experiences a backemf opposite to the direction of current flow. This occurs until the flow of current in one direction stops completely. However, at this point, charge has built up on the opposite plate of the capacitor, so the cycle begins again in reverse. We can calculate the frequency of this oscillation using what we know about circuits, capacitors, and inductors.

Mathematically, we know from Ampère's Law (Kirkhhoff's Voltage Rule) and conservation of charge (Kirkhhoff's Current Rule) that $V_C + V_L = 0$ and $i_C = i_L$. Additionally, we know that $V_L(t) = L\frac{di_L}{dt}$ and $Q = V_C C \implies \frac{dQ}{dt} = i_C(t) = C\frac{dV_C}{dt}$. Rearranging, we have:

$$\frac{d^2 i(t)}{dt^2} + \frac{1}{LC} i(t) = 0$$

Define $\omega_0 = \frac{1}{\sqrt{LC}}$. The solution to this differential equation yields the result

$$i(t) = I_0 sin(\omega_0 t + \phi)$$

If we let ϕ, the phase shift, be zero, we see that this LC circuit oscillates in a sinusoidal pattern, much like the simple harmonic oscillators we know and love. Specifically, the LC circuit oscillates with frequency $f = \frac{1}{2\pi\sqrt{LC}}$ Hz. In every cycle, charge flows from one plate of the capacitor, through the inductor, to the other plate and back. In fact, the typical resonance conditions hold; if there exists a driving voltage that oscillates at the same frequency and is in phase with the oscillations in the LC circuit, then the amplitude of the oscillation monotonically increases without bound. Of course, in real life, wires have resistance, so there is some damping factor. However, with the Tesla coil, the rate of increase of amplitude far outweighs the damping effects.

2.2 Tesla Coil

The main concepts behind all Tesla coils are the same: an oscillating primary LC circuit induces resonance in a secondary LC circuit, building up voltage on the secondary's capacitor until it breaks down. We'll first focus on the theory behind the spark gap Tesla coil, and then describe how our Tesla coil's design slightly differs.

2.2.1 Spark Gap

The basic circuit for a spark gap Tesla coil is shown in Figure 3a. The spark gap, primary inductor, and capacitor are all connected in series. Charge builds up on the capacitor from the AC power source (usually sent through a transformer in order to step up the voltage), until the voltage across the spark gap is high enough to breakdown and ionize the air between the leads of the spark gap. After all, the spark gap is a capacitor in disguise, and therefore has some breakdown voltage. Ionized air is a great conductor (compared to unionized air, which is a great insulator). The capacitor can then discharge through the spark gap until the air deionizes. The ionization voltage threshold is higher than the deionization voltage. As the capacitor is short-circuiting through the spark gap, it is also discharging through the inductor, forming an oscillating LC circuit. The frequency of oscillation of a Tesla coil's primary circuit is so high (MHz range) that the AC source (at 60 Hz) can be thought of as DC, and thus does not affect the frequency of the resonant circuit.

While the spark gap is 'closed', the primary Tesla coil circuit acts exactly as an LC circuit. We know that an oscillating LC circuit will induce alternating magnetic fields in the interior of its inductor. In a Tesla coil, another inductor (the secondary) is concentric with the primary, but usually has a much high turn density. From Faraday's Law of Induction, this means that a voltage ($\mathscr{E}$: electromotive force) is induced in the wires of the secondary coil. This induced voltage will charge the secondary circuit's capacitor (really, the top toroid is just one plate of a capacitor, where the other plate is the physical ground), and then the secondary coil will begin displaying resonant LC behavior, albeit initially at a lower energy level. Each time the primary oscillates, it induces some voltage in the secondary. Essentially, the current in the primary is driving oscillations in the secondary.

Consider what happens if the angular frequency of each circuit is tuned to the same value. This is possible because the resonant frequency of an LC circuit only depends on the inductance of its inductor and capacitance of its capacitor. Letting $\omega_1 = \omega_2$, the oscillating current through the primary circuit establishes a magnetic field on the interior of the primary's inductor, which then induces a voltage in the secondary's inductor. This voltage constitutes a 'push' of the secondary circuit. If these 'pushes' are well timed – that is, if the time of each push coincides with the peaks of the current in the secondary and is in the same direction – then the secondary coil will resonate. The behavior is just like pushing a child on a swing from both ends: if you push every time the child is at a maximum altitude, you deliver energy to the child's oscillation, and thus increase the amplitude of his swinging. Similarly, the resonant 'pushes' delivered by the primary are synced in such a way that they increase the energy in the oscillations of the secondary. As the energy rises, so does the amplitude of the voltage in the secondary (although it is still oscillating). Eventually, the voltage is high enough that the air around the top load breaks down as charges try to spark to ground (the other plate of the top 'capacitor'). The spark will ionize the air, turning it to plasma and rapidly increasing its volume, before the volume subsides again. This expansion and contraction of the air's volume constitutes a pressure wave, which can be interpreted by the human ear as sound, albeit a square, rather than sinusoidal, wave.

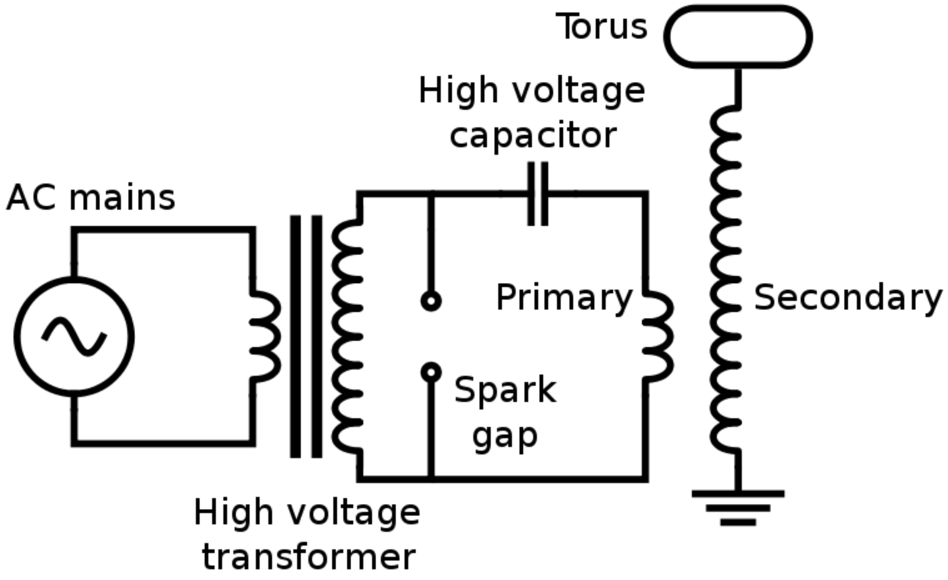
Another way to think of a Tesla coil is as a transformer. It takes relatively low voltage AC (15,000V) from the transformer and steps it up to a higher voltage, corresponding to the turn density ratio of the secondary to the primary. Of course, this stepped-up voltage is not enough to cause electrical discharge, which is where resonance comes into effect. The windings are loosely coupled (lots of air space between them), but this is mostly to protect the primary circuit from the induced $\mathscr{E}$ it experiences from the oscillations in the secondary.

An alternative circuit configuration is shown in Figure 3b. However, in this schematic, high frequency, high voltage oscillations across the high voltage capacitor are mirrored onto the transformer's secondary winding. Since we're using a (relatively weak) neon sign transformer, continual operation of a Tesla coil in this configuration will damage the transformer's internal components and is not recommended.

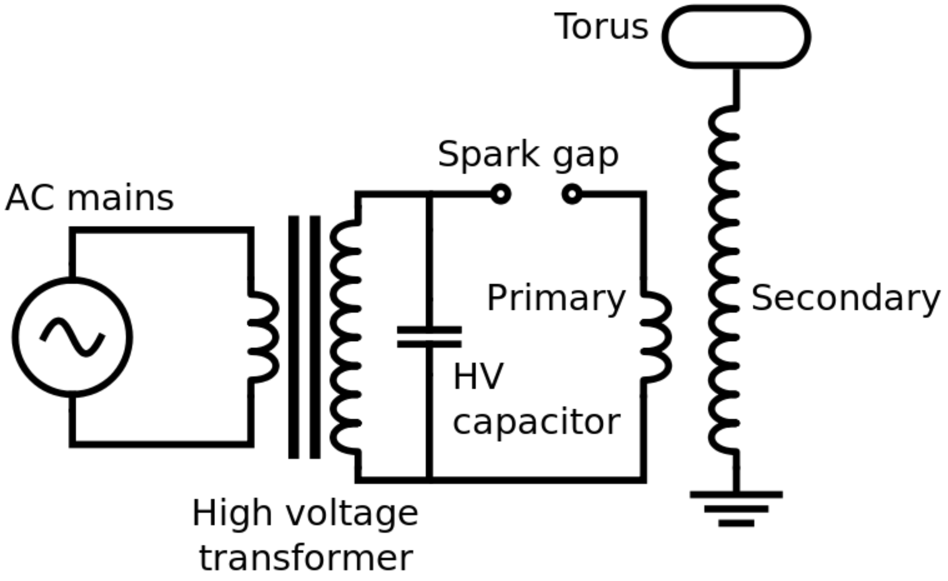
2.2.2 Solid State (DRSSTC)

The previous section described the theory of a spark gap Tesla coil. However, a spark gap Tesla coil only operates at one specific frequency. We want to control the frequency of sparks from the top of the secondary. For an A4 note (440 Hz), we'd make sparks 440 times per second, and for middle C (262 Hz), we'd make sparks 262 times per second.

Therefore we need something that simulates the functionality of the spark gap in a spark



(a) Common Tesla Coil Configuration



(b) Alternate Tesla Coil Configuration

Figure 3: Two schematics for a Tesla coil. The top circuit is preferred, as it protects the transformer by shorting across the spark gap. In practice, the secondary coil has many, many more turns (two to three orders of magnitude more) than the primary coil. The toroid acts as a capacitor to ground, in that effectively one plate is the toroid, and the other is the grounding rod or the actual ground.

gap Tesla coil, but which is controllable by an electrical signal. The best component would be a relay rated for high voltage, high current, and fast switching speeds. However, components like this cost upwards of $2,000, well outside our budget. An alternative approach is to use transistors. The type of transistor best suited to our needs is an IGBT, or Integrated Gate Bipolar Transistor. An IGBT is basically a one-directional switch that closes when the voltage difference between the gate and collector is positive ($V_{GE} > 0$). We'll likely arrange the IGBTs in an H Bridge pattern to allow current to flow both directions in the LC circuit. IGBT's are useful because they can withstand both high current and high voltage applications.

2.3 Arduino

What is an Arduino?

There are many types of Arduino boards, such as the Uno, Mega, Leonardo, Due, and more, but all Arduino boards share common features. An Arduino board allows an electrical engineer to interact with a microcontroller (a tiny computer) by uploading programs to be executed and by connecting wires to pins which ultimately lead to the microcontroller. The Arduino board also contains a number of other features, like power indicators, resistors, and more. The essential feature of an Arduino is its ability to execute programs uploaded by a programmer into the microcontroller's memory. These programs are written in the Arduino programming language, which is an extension of C++ that includes various Arduino- and microcontroller-specific features (including **Serial**, `digitalWrite`, `analogRead`, direct register access, and more). Behind the scenes, though, all Arduino code is just C++.

To be technical, the microcontroller doesn't actual execute the high-level commands that we write in C++ or the 'Arduino' programming language; rather, the developer's computer compiles the 'Arduino' code (which is ultimately comprised of C++ constructs) into machine code (pure binary) using `avr-gcc`, a compiler developed by the Free Software Foundation specifically for use on Atmel microcontrollers (the type found on Arduino boards). `avr-gcc` is accompanied by `avr-libc`, a library of useful C tools, `avrdude` (AVR Downloader/UploaDEr), a project for in-system programming, `avr-gdb`, a debugger, and more. [17] The compiled machine code is then uploaded onto the board using a USB-to-USB cable. Most boards, including the Arduino Uno and Arduino Mega, use a Standard-A (default computer USB port) to Standard-B converter, but some boards, like the Arduino Due, use a Standard-A to Micro-B converter. An onboard USB-to-Serial converter then converts the incoming USB-encoded data into Serial data that can be fed directly into the microcontroller's flash memory. Additionally, all Arduino boards allow for direct programming of the contents of the microcontroller's memory using the 2x3 pin programming header (known as 'flashing the firmware/software'); however, this technique is dangerous, and should not be used. There are parts of the microcontroller's memory that should not be modified, such as the portion that contains the bootloader, which is a chunk of code that decides when to start executing the uploaded code. (Essentially, it waits for a pause after a stream of incoming data and then runs the uploaded program.) In summary, the Arduino executes a compiled version of an engineer's C++ code.

Our Goal

For this project we developed software to convert incoming MIDI signals to outgoing electrical pulses. (See Subsection 2.4.) This task is challenging. Much like the brain, the Arduino's job can be separated into three steps: receiving input data, processing data, and sending output data. The theory behind each of the steps is discussed in the following sections.

2.3.1 Serial Input (and Output)

Serial communication refers to the encoding of binary information in a sequence of alternating high voltages and low voltages, corresponding to either a binary 1 (high voltage) or a binary 0 (low voltage). For most Arduino boards, 'high voltage' is defined as 5V and corresponds to a binary 1. 'Low voltage' is defined as 0V and corresponds to a binary 0. Some Arduino boards, like the Due, use 3.3V as 'high' and 0V as 'low.'

Every Arduino comes equipped with at least one Serial port, also known as a UART (Universal Asynchronous Receiver/Transmitter) or USART (Universal Synchronous Asynchronous Receiver/Transmitter). This Serial port is given the name **Serial**. **Serial** handles communications in two ways: it can communicate with the computer via USB, and it can communicate with other electrical devices via the digital pins RX (0) and TX (1). RX means receiver and TX means transmitter. [18]

Some Arduino boards come with more than just the standard Serial port. For example, both the Arduino Due and the Arduino Mega 2560 R3 (an improvement over the older Arduino Mega) have additional Serial ports beyond just Serial. These boards have Serial1 on pins 19 (RX) and 18 (TX), Serial2 on pins 17 (RX) and 16 (TX) and Serial3 on pins 15 (RX) and 14 (TX). [19] [20] These extra built-in UARTs are extremely useful because a programmer can employ Serial ports other than RX0 and TX1, which are tied to both Serial and the computer's USB connection.

It is worth mentioning in passing that a SoftwareSerial library exists that allows arbitrary digital pins to be used as Serial I/O; however, this solution is both slower than the native UARTs built into the Mega's hardware (as it has to replicate in software an optimized hardware circuit) and has limitations, such as restricted data flow (just one SoftwareSerial can receive data at a given time, and each board has particular limitations as to which pins are available for SoftwareSerial). [21] So, although SoftwareSerial exists, the hardware of the Mega and Due is simpler, faster, and more elegant.

When interfacing with the computer, serial communication is easy. Serial output can be viewed as a sequence of characters via the Arduino environment's Serial monitor. We use this fact extensively to help debug. Serial input can be passed to the Arduino's Serial buffer via the text input field at the top of the Serial monitor. While less useful, this feature could be used for debugging in other projects.

Serial I/O also operates on an Arduino's pins. By default, pin 0 receives data (input), and pin 1 transmits data (output). It's interesting, although not strictly relevant, that when a serial output call is made (via `Serial.println` and similar functions), it appears as though the data is only sent to the computer monitor; however, the serial data is also transmitted on pin 1. This quirk can be used to daisy-chain Arduinos.

An RX pin (such as RX19 for `Serial1`) reads in data at a set rate (the baud rate) and stores it in a 1024-byte wide Serial buffer. The buffer implements Queue (first in first out) logic. When the RX pin is brought high (at 5V by default) *relative to the Arduino's ground*, a logic 1 is appended to a register, and when it is brought low, a 0 is added. When eight bits are added to this register, the entire byte is appended to the Serial buffer and the process resets. Bytes can be drained from the Serial buffer by functions like `Serial.read`.

Bits in the Serial buffer (and in the Arduino in general) are ordered from least-significant to most-significant. This architecture, shared by most Serial data transfer systems, is referred to as little-endian. Therefore, the sequence of voltages 5V-0V-0V-0V-0V-5V-5V-0V is stored in the buffer as '10000110,' but represents a byte with a value of '01100001,' or 0x61 or 97. This byte can be interpreted as a character using the ASCII encoding (in this case, 97 corresponds to 'a'), or can be interpreted via another protocol, such as MIDI.

The Arduino's Serial I/O must have a specified baud rate passed as a parameter to the `Serial.begin(int baudRate)` command. This value, expressed in units of bps ('bits per second'), corresponds to the number of times per second the Arduino measures the voltage on an RX pin and adds either a 0 or a 1 to a register. For instance, `Serial.begin(1000)` will check the RX0 pin every millisecond, but `Serial.begin(100000)` will check the RX0 pin every $10\mu s$.

2.3.2 Timers

The Arduino usually executes programs synchronously, that is, executing them line by line. However, all Arduinos come with a few timers, which can be used to execute code asynchronously.

A timer is functionally a counter. By default, on every iteration of a clock cycle, a timer's counter is incremented by 1. When the timer hits its maximum value, it overflows to 0 and begins the cycle again. Functions can be attached to timers so that when a certain condition is hit, a specified block of code executes. This is a useful tool if one wants to take a measurement exactly every second (for, say, a space launch), or wants to only pulse power to a machine.

Most Arduinos have just 3 timers: Timer0, Timer1, and Timer2; but the Arduino Mega 2560 has six timers: Timer0 through Timer5. This is the key development that will allow us to play five or six notes at once.

A given timer has certain registers associated with it. One of the the key aspects of a timer is its maximum size. This size determines the upper bound of what a given timer can count to. Timer0 and Timer2 are 8-bit timers, meaning that they can count to $2^8 = 256$ before overflowing. Timer1, Timer3, Timer4, and Timer5 are all 16-bit timers, meaning that they can count to $2^{16} = 65536$ before overflowing.

Sometimes, the desired period of a timer wouldn't fit into its counter register (either 1 or

2 bytes) if the counter was incremented on every clock cycle (each clock cycle is $\frac{1}{16000000} = 62.5ns$). We certainly couldn't make a 1Hz timer with increments every 62.5ns. To solve this problem, there is the notion of a prescaler. A prescaler is a scaling factor that changes the conversion of clock cycles to incrementation. For instance, if Timer1's prescaler value is 64, the Arduino only increments Timer1's counter every 64 clock cycles, rather than every one. However, the set of available prescaler values differs for Timer0, Timer2, and Timer1/3/4/5.

We can write our own code to be executed when a timer hits a specified condition; these code blocks are called ISRs, or Interrupt Service Routines. An ISR is passed an overflow condition, but otherwise works just like a normal function.

Be careful when manipulating timers, as it could cause an accidental malfunction in library procedures. delay() and delayMicroseconds() both rely on Timer0, so if you change Timer0's prescaler, be sure to divide by the appropriate amount when passing parameters. The Servo library uses Timer1 extensively, so any changes to Timer1 will affect those calls. Lastly, the tone() function, which generates pulse widths corresponding to given frequencies, uses Timer2.

2.3.3 Port Manipulation

To be a useful component in our assembly, the Arduino must be able to output voltages, quickly turning on and off. The Arduino standard library of functions provides digitalWrite, which is sufficient to set a pin's output voltage to either HIGH (3.3V on Arduino Due, 5V for all other boards) or LOW (0V) under usual circumstances. However, the conditions we're operating in are far from usual, and for our purposes, digitalWrite is too slow.

Before we talk more about optimization, we have to talk about clock speed. A processor's clock speed is a measure of how many instructions the processor can execute per second. For example, my Macbook Pro has a 2.4GHz processor from Intel. The Arduino Mega's clock speed is 16MHz, meaning it can execute 16 million instructions per second. The Arduino Due's clock speed is 84MHz, meaning it can execute 84 million instructions per second. But what exactly is an instruction? The definition varies from processor to processor, but think of it as an action the computer performs.

Arduino code, being C++ in disguise, compiles to assembly, and it is this assembly that is uploaded to the microcontrollers. Assembly is a programming language with limited commands that interface directly with the processor. Some commands read from a specific point in memory, while others deal with moving information from memory to registers and back. Assembly can be directly translated to binary code (machine code). Each assembly command takes a specific number of clock cycles to execute. For instance, the Arduino assembly commands cbi (clear bit i) and sbi (set bit i) both take two clock cycles. Jumping back to the start of a while loop is an rjmp (relative jump) and also takes two clock cycles. So, if we want to maximize the switching speed of the Arduino we have to minimize the number of clock cycles a single switch takes to execute.

digitalWrite takes two parameters: an integer corresponding to the pin of interest, and either HIGH or LOW specifying whether to bring the pin to 5V or 0V. However, the internal code of digitalWrite() has a slow chain of conditional checking to extract the internal pin number from the pin parameter. Therefore, there must be a faster way.

There is! Each pin of the Arduino is associated with a particular bit in three different hardware registers: PORTx, PINx, and DDRx, where x is a letter between A and L (on the ATmega2560 – older micro controllers have fewer registers). Each register is eight bits (1 byte) long, and therefore each register contains information about eight different pins. For instance, information about Pin D5 is contained in the fifth bit from the right (fifth least significant) in registers PORTD, PIND, and DDRD.

DDRx contains information about pin configurations. A 1 means that the pin is configured as input; a 0 means it is configured as output. PORTx controls whether a pin is HIGH or LOW, and PINx reads the status of pins that were configured as inputs using DDRx. Both DDRx and PORTx are read/write, but PINx is usually read only. However, the ATmega2560 allows writing a logic 1 bit to a PINx register to toggle the state of the corresponding bit in the corresponding PORTx. That is exactly what we need in order to toggle pulses to the primary circuit.

There are benefits and drawbacks to using direct port manipulation. The code is harder to debug and much harder for an outside programmer to understand. It is much less portable than digitalWrite and digitalRead, since the pin mappings are different on the ATmega2560, the ATmega328, the ATmega168, and the ATmega8. Additionally, it is very easy to cause malfunctions in built-in systems, especially when mistakenly setting registers directly, rather than and-ing or or-ing them with given bitstrings.

The biggest benefit for our purposes is the fast speed of direct port access compared to the clunky digitalWrite. Additionally, direct port access allows multiple pins to be set at exactly the same time (not useful to us, but interesting nonetheless). Finally, this approach can reduce program size in the event that the program is almost too large to fit in the board's memory.

The ATmega2560, the processor which underlies the Arduino Mega 2560, has the pin mappings specified in Figure 4. Some pins have an internal name of Pxn, where x is a letter between A and L, and n is a number between 0 and 7. These pins corresponds to the nth bit from the right in registers PORTx, PINx, and DDRx.

We've discussed the pin mappings for the ATmega2560 because they are easier to understand and because they are more analogous to the popular Arduino Uno. However, the Arduino Due actually uses an entirely different setup to toggle output states, which we will discuss briefly. The Arduino Due provides a plethora of 4-byte registers (it's main advantage over the older Arduino models) that control the input and output of various pins.

The Arduino Due has analogous registers to DDRx and PORTx. We'll ignore PINx for now. The REG_PIOx_OER, or Register Parallel Input/Output X Output Enable Register, determines whether a pin is configured as input or output. REG_PIOx_SODR, or Set Output Data Register, will set a pin high if the corresponding data bit is brought high. REG_PIOx_CODR, or Clear Output Data Register, will set a pin low if the corresponding data bit is brought high. The abstraction of PORTx and PINx into x_SODR and x_CODR allows a more sensible treatment of pin output values. Instead of using PORTx and PINx as direct access to the pin's output values, the programmer can write instructions in the SODR

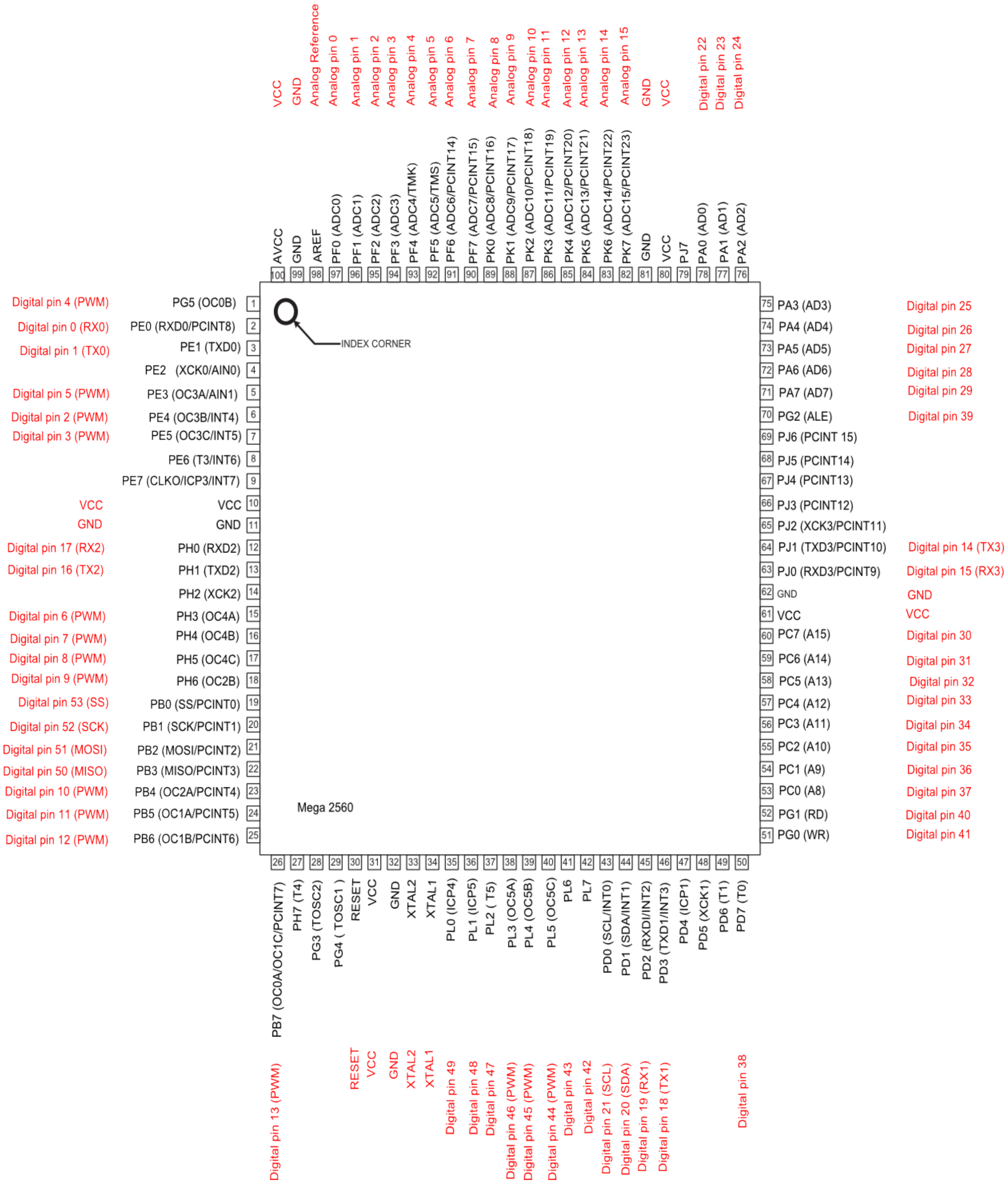


Figure 4: Pin mapping for the ATmega2560, the controller that underlies the Arduino Mega 2560.

Table 1: A description of the bytes found in a typical MIDI event transmission

Name	Range	Description
Status	0x0-0xF	Type of MIDI message
Channel	0x0-0xF	Channel number
Data Byte 1	0x0-0xFF	First data byte (see Table 2)
Data Byte 2	0x0-0xFF	Second data byte (see Table 2)

and CODR register which then carry over to the actual output pins. 'x' can be any letter between A and F.

A more comprehensive analysis of the Due's pins is shown in Table 6 in Appendix C.

2.4 MIDI

2.4.1 What is it?

MIDI, or Musical Instrument Digital Interface, is a standard that allows two electronic instruments to communicate. It was developed around the time electronic music started surfacing in the 1970s. Various manufacturers formed the MIDI Manufacturers Association (MMA), a coalition that has created two standards – General MIDI Level 1 (GM1), and General MIDI Level 2 (GM2) – while still allowing companies to develop their own unique version through the use of System Exclusive messages.

GM1 specifies the use of MIDI events to communicate information in the form of instructions. It specifies 128 instruments (denoted as 'programs') and 64 percussion sound patches. It also specifies a set of controller actions. A comprehensive list of these options is given in Appendix D.

MIDI has 16 channels of communication on which it can send or receive data. Usually, however, Channel 10 is set aside for percussion tracks. [24]

2.4.2 Information Protocol

The basic unit of MIDI communication is a MIDI event, which is essentially an instruction. This event is normally a set of three bytes which encodes four pieces of information: the command byte, the first data byte, and the second data byte. The command byte is broken into four bits which represent a status value and a channel value. The status value communicates which type of MIDI message follows, and the channel value communicates which channel the event should influence. This information is summarized in Table 1.

The first bit of any incoming MIDI message is a 1, so the range of possible status values is really 8 to F. The status value determines what type of MIDI message the incoming event is, and also determines how the data bytes should be interpreted. This information is shown in Table 2. The channel values range from 0 to F, specifying any of the 16 possible channels of

Table 2: A description of the data bytes expected by the possible status values

Status	Value 1	Value 2	Comments
8	Note	Velocity	Note off
9	Note	Velocity	Note on (if velocity = 0, note off)
A	Note	Value	Polyphonic pressure
B	Controller #	Value	Controller change
C	Program	-	Program change
D	Value	-	Channel pressure
E	Value	Optional Value	Pitch bend
F	Value	Optional Value	System Exclusive (SysEx) or Realtime Message

MIDI communications (ranging from '0000,' which is channel 1, to '1111,' which is channel 16). Therefore, the entire command byte ranges from 0x80 to 0xFF (128 - 255).

For pitch bend (E), if a machine sends two value bytes, they are interpreted as a LSB and MSB, whereas if it sends just one, that byte is interpreted as a MSB.

Note that some status commands (C and D, and sometimes E and F) are associated with fewer than one data byte. However, a majority of the events that are sent are either note on or note off events, both of which require two data bytes. We'll focus on these events at length.

For both note on and note off events, the first data byte corresponds to the note pitch, and the second data byte corresponds to the velocity, or volume, of the note. A note off command always has velocity zero, but some electronic instruments transmit note on messages with a velocity of zero to indicate that the note should be turned off.

The MIDI note value ranges from 0 to 7F (0 to 127), as does the velocity, where a higher value corresponds to a louder volume. The MIDI pitch value is associated with musical notes via Table 3. [24]

2.4.3 Decoding MIDI

The Arduino has to convert the sequences of bits it gets from a MIDI stream to sensible information about the MIDI event. To further understand how to convert Serial data to MIDI events, we'll practice with a few sequences of bits.

First, let's pretend the MIDI stream sends the following data bits in sequence:

100100010011010001101011

The first step is to convert these to bytes: our command byte (1001 0001) is 0x91, or 145; our first data byte (00110100) is 0x34, or 52; and our second data byte (01101011) is 0x6B, or 107. The 9 (status value) in the command byte indicates that this MIDI event is a note on event. The 1 (channel value) in the command byte indicates that this MIDI event takes place on channel 2. Since this is a note on command, the first data byte corresponds

Table 3: Translation table between MIDI note values (as received by a note on or note off message) and musical notes.

Octave #	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

to a MIDI note pitch of 52, which (using Table 3) means E4. Finally, the second data byte corresponds to a velocity of 107, which is about 84% of the maximum. Putting it all together, we conclude that this MIDI event is an instruction to play E4 on channel 2 at 84% volume.

Let's now dive into the hardware, looking at real voltages. MIDI operates at a baud rate of 31,250 bps. This means that each bit has a duration of exactly 32μs. MIDI uses the 8-N-1 data encoding protocol, which specifies exactly how data bits are transferred over a wire. (Note that the phrase 'data bit' is being used in a more general context here than above.) In the 8-N-1 protocol, each byte of information is associated with more than 8 voltage 'bits.' For every eight data bits (one byte), there are no parity bits (the 'N' in 8-N-1) and 1 stop bit. All Serial communications are prefaced with a start bit to inform the listener (in our case, the Arduino) that new data is arriving. Therefore, each MIDI byte is really encoded in ten bits: a start bit (low), eight data bits, and a stop bit (high). Therefore A MIDI source can transmit at most 3,125 bytes every second.

MIDI has no error-detection or error-correction capabilities due to a lack of parity bits, and therefore the MMA requires that the maximum cable length be 50 feet to minimize the degradation of the signal due to random noise pickup.

One last note about the MIDI protocol. Bits are transmitted (as in almost all Serial systems) from least significant bit to most significant bit. This may appear 'backwards,' since we often like to write bytes from most significant bit to least significant bit. This quirk is a feature that allows computers to process data more quickly.

Figure 5 shows a real oscilloscope graph of voltage on an Arduino's RX pin relative to the Arduino's ground. Let's try to determine what MIDI event was sent. Each box is 100μs wide, so each bit is a little less than two ticks wide.

The problem is left as an exercise for the reader.

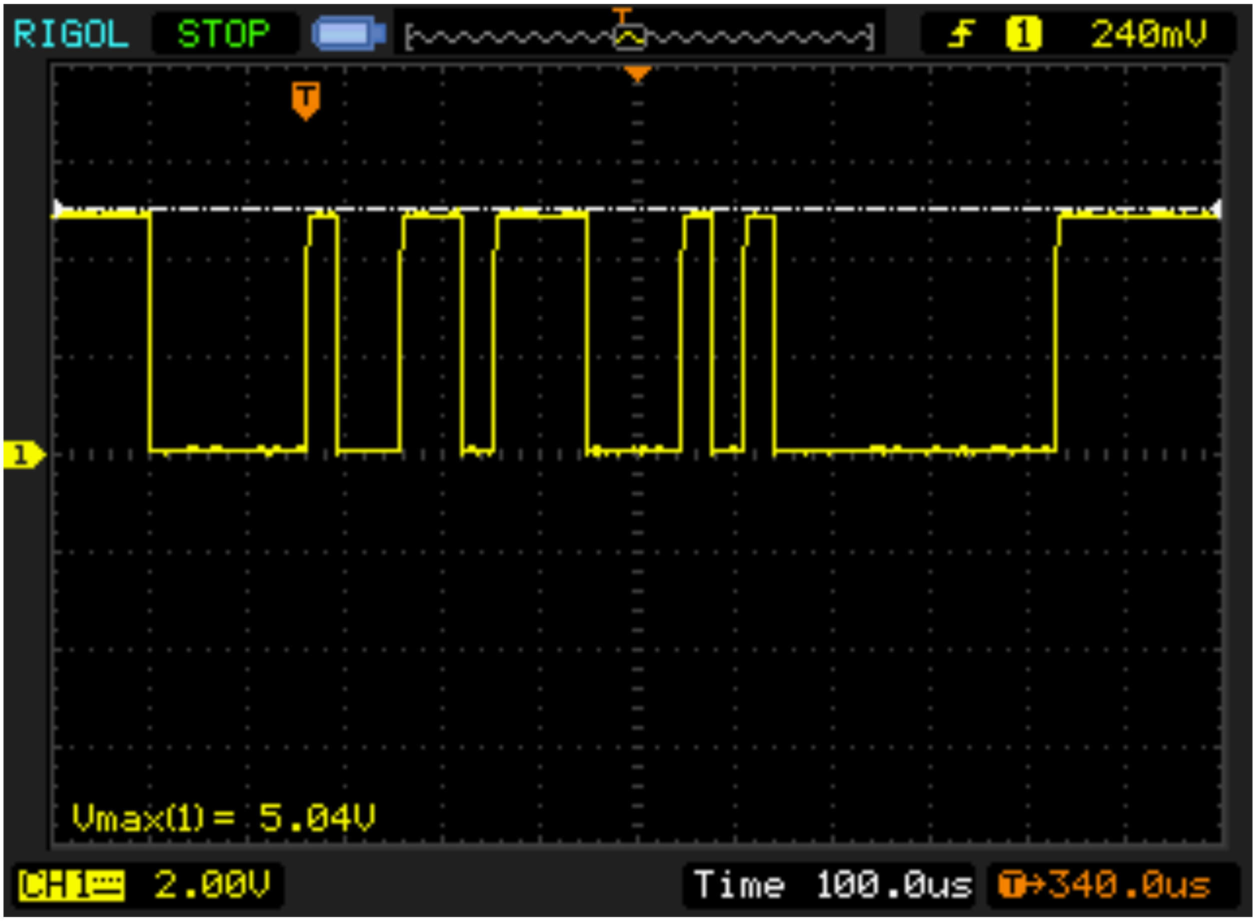


Figure 5: Voltage incident on the Arduino's RX pin when a specific MIDI event is sent. Try to figure out what event was sent.

Software has been developed to decode incoming MIDI signals in the exact same way we just did. (See Appendix E.)

Conventional MIDI cables consists of five pins in a half-circle configuration. If looking directly at a male MIDI plug, these pins are, from left to right: 1, 4, 2, 5, 3. Pins 1 and 3 are not used, and only exist for forward-compatibility in MIDI hardware.

MIDI hardware is not capable of both transmitting and receiving messages on the same cable, so often MIDI contains two cables, one to transmit and one to receive. For the transmitter, pin 4 is always at 5V, and pin 5 is either at 0V or 5V. On the receiver the same is true, with the addition that pin 2 is grounded. This ground provides the shielding for the signal pair of wires.

2.5 Circuit Elements

In addition to the conventional circuit elements described previously, we use unconventional components in our circuitry. The following sections describe the theory of each of these components. Specifications are described later on, when we discuss the practical application of these idealized components to our overall goal of creating a musical Tesla coil.

A note about active and passive components: passive components are those that do not require an external voltage supply (more than just the circuit connections) to function. Examples of passive components include resistors, capacitors, diodes, switches, etc. Active components require an external power source, conventionally denoted V_{cc}, originally named after the common collector of an NPN transistor based circuit. This voltage supply usually powers internally logic components and other active components inside the the overall active component.

2.5.1 Opto-isolator

An opto-isolator is a circuit element that connects two electrically disconnected circuits via optical transmission. A photodiode (think LED) on one side of the circuit will transmit a signal if and only if there is a voltage present across the ends. This light is fiber-optically transferred to a photo-transistor on the other circuit that activates when the light is detected. Many opto-isolators employ internal mechanics different from these, but this is the general idea. A sample schematic is shown in Figure 6.

Opto-isolators are extremely useful because they allow two circuits to have separate grounds. We'll use this fact to isolate the two halves of our MIDI circuit.

2.5.2 Fiber Optical Transmitter and Receiver

Electricity is not the only way to send signals. We can also send signals via light. To do so, we need a device that can convert electrical signals to light signals and a device that can convert light signals back to electrical signals.

An LED (Light Emitting Diode) is one such component. Given an applied voltage across its leads (past some threshold voltage), an LED will light up. If the voltage difference is too high, the LED will burn out as the physical and chemical structures responsible for the emission of light begin to break down. Since an LED is a diode, it only allows current through in one direction, so the polarity of LEDs matters. All fiber optic transmitters are essentially glorified LEDs – perhaps they can turn on and off very quickly, or perhaps their spectral output is less broadband – but fundamentally a fiber optic transmitter can be thought of as a fancy LED.

On the other hand, we'll need some device to sense an incident light source and produce some electrical signal. These components are called phototransistors. All semiconductor materials respond to a wide range of frequencies, some more than others. When light of any wavelength hits a material with excess electrons (N-type) or electron holes (P-type), it alters the concentration of charge carries in the material, thus changing the material's electrical properties, specifically with regards to conduction. All semiconductor materials respond to applied light in this way, but the largest response is to infrared light (about 800 nm, in the near-infrared portion of the electromagnetic spectrum). In fact, any transistor can be converted into a phototransistor with a transparent case and an extremely strong source

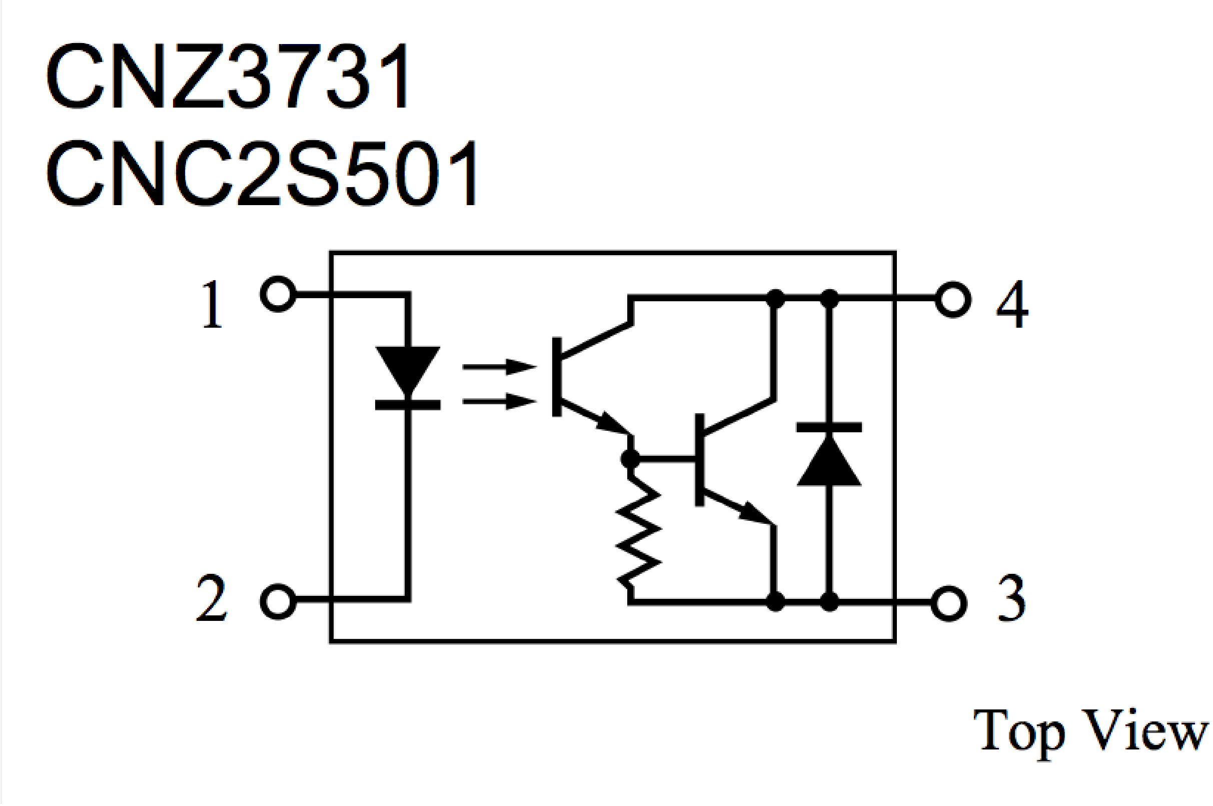


Figure 6: Internal Circuit Diagram for the CNZ3731 opto-isolator. We are not using this opto-isolator (although we do have it in the lab) but its schematic is relatively straightforward.

of IR. Phototransistors on the market use focusing lenses, doping, and other techniques to amplify this natural effect to the point where the final component is effectively a light-controlled on/off switch, although in reality it demonstrates the same resistive heating that normal transistors do when switching states. [25] As with the fiber optic transmitter, these fiber optic receivers are often tuned for desired parameters, such as switching speed, minimum incident light, or maximum output load.

We still need some way to transmit light information from our fiber optic transmitter (a specialized LED) to the fiber optic receiver (a phototransistor). A fiber optic cable is a specialized flexible cable that traps light inside for very long distances. Google Fiber is using similar technology to transmit data at up to 1000 Mbps, or 125 megabytes per second. [26]

2.5.3 Voltage Regulator

A voltage regulator is an electrical component that accepts some variable voltage and outputs a specific voltage. For example, a voltage regulator could accept a fluctuating voltage – perhaps 19V plus or minus 0.5V – and output a consistent 15V. Voltage regulators rely on a feedback mechanism to keep modifying the output voltage until it exactly matches a reference voltage stored in the internals of the regulator.

While some voltage regulators use electromechanical feedback to control the output voltage, our voltage regulators use a feedback loop consisting of an op-amp connected to a Darlington pair of transistors (which enhances current gain) to modify the output. An example circuit of a voltage regulator is shown in Figure 7.

2.5.4 NOT Gate

A NOT gate performs a unary logical not operation on an input signal. Essentially, if the input is low, the output is high, and if the output is high, then the input is low. Each NOT gate defines its own cutoff between a low and a high input signal. A simple NOT gate needs four pins: V_{cc}, GND, input, and output. Specifically, if the input voltage is below some threshold, the output will be at (approximately) V_{cc}, and if the input voltage is above some threshold, the output voltage will be at GND. A NOT gate can be constructed with one transistor. If there is sufficient voltage at the base of our bipolar junction transistor, then the transistor will act as a closed switch, allowing voltage to drain to ground. However, if there is no voltage at the base of the transistor, then the transistor will act as an open switch and the output will be at a nonzero voltage.

The class of circuits that uses BJTs and resistors in this way is known as TTL, or transistor-transistor logic. The other large class of circuits is CMOS, or complementary metal-oxide-semiconductor, which uses a different configuration of semiconductor devices to perform logic and amplification. While other classes exist (RTL, resistor-transistor logic, and DTL, diode-transistor logic, to name a few), TTL and CMOS are by far the most prevalent. Even though there are fundamental differences between TTL and CMOS circuits, we'll treat them as identical with regards to the overall logic performed on input signals. [27] [28] [29] Technically, all our ICs use TTL.

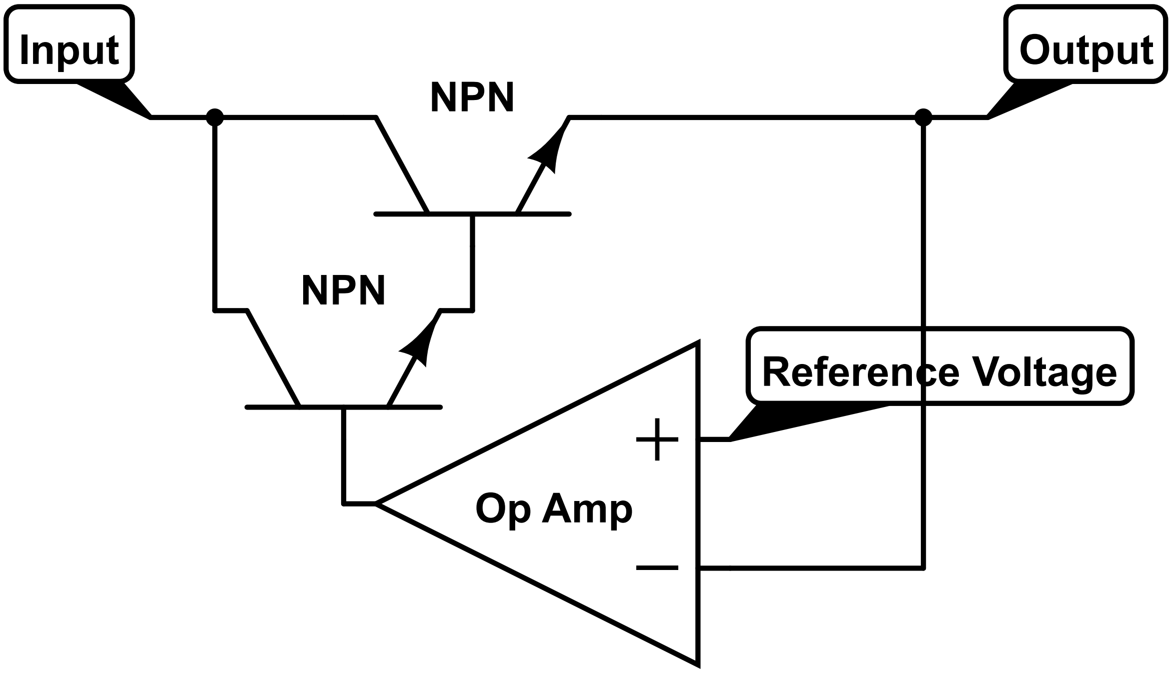


Figure 7: A sample circuit for a voltage regulator. The op-amp modifies the input voltage via a feedback loop so that the output voltage nearly matches an internal reference voltage.

More complicated logic gates exist, performing operations such as NAND, OR, AND, and many more. Binary logic operations such as these will require two transistors to handle two inputs. These logic gates can be combined in various ways to produces adders, memory, and many other high-level components relevant to a modern computer. In fact, it is possible (although hardly advisable) to make a computer entirely out of NAND gates.

2.5.5 D-Type Flip-Flop

A flip-flop is an active, stateful electrical component –that is, its output at any point in time is determined not only by its inputs at that moment, but also by previous inputs. In this way, the flip-flop is a piece of memory. Specifically, the flip-flop can be in one of two states:high or low. It is possible to build a memory unit, perhaps for a computer, using flip-flops, since each one can store and retain one bit of information. Although the flip-flop is internally just an complicated circuit built of transistors, resistors, and other electrical components, we will treat it as a black box which has memory. It is conventional for flip-flops provide both an output pin and it's complement, to reduce the unnecessary use of NOT gates.

Conventional (SR) flip-flops have two inputs: a set (S) and a reset (R). If the R pin is brought high, the flip-flop is brought low. If the S pin is brought high, the flip-flop is brought high too. If both pins are low, then the flip-flop maintains its state. Internally, SR flip-flops are cross-coupled NOR gates. Draw the circuit diagram to verify that the extra inputs to the NOR gates for a set-reset pair.

We use a D-type flip-flop, which is slightly more complex. In addition to the set and reset lines, there is both a clock line and a data line (hence the D). At a specific point in the clock's cycle (perhaps on the rising edge), the value of the data line is mirrored to the state of the flip-flop, which can be read as the flip-flop's output. The set and reset lines on a D-type flip-flop work the same way they do on an SR flip-flop, and are often used to force the flip-flop into a particular state, regardless of the clock or data lines.

2.5.6 Integrated Gate Bipolar Transistor

An integrated gate bipolar transistor (abbreviated IGBT) is a transistor designed for switching quickly at high voltages and currents. It uses a particular arrangement of semiconductor materials to achieve this goal. Notably, instead of the three regions of the familiar bipolar junction transistor (PNP or NPN), IGBTs have four alternating regions (NPNP or PNPN). It is this configuration that gives IGBTs their switching power.

2.5.7 Gate Drive Transformer

The gate drive transformer we use is a 1:1:1 voltage transformer that allows a voltage difference across its input leads to be mirrored across two pairs of output leads. A 1:1:1 gate drive transformer consists of three wires that are twisted around each other, then spiraled through a metal toroid. Just as in any other transformer, the changing current through the input wire creates a changing magnetic field in the metal toroid. This changing magnetic field induces an electromotive force, which essentially acts as a voltage, in the pair of output wires. Since the coil ratio is 1:1:1, the voltage ratio is also 1:1:1.

2.5.8 Current Transformer

A current transformer decreases the magnitude of alternating current in a circuit. Current transformers are often used to measure the frequency of high voltage alternating current, which cannot be fed directly into measuring instruments. Usually the lowered current is used directly for measurement; however, we will use it as a feedback transformer so that the logic circuit can 'sense' the oscillating current in the power circuit. A current transformer works using Ampère's Law and Faraday's Law. As current alternates in the measured wire, a magnetic field is established in the current transformer's core. This alternating magnetic field induces an electromotive force in the measuring wire, which then drives a proportional current (although phase shifted by a quarter of a cycle).

3 Design

The design of this project is separable into three distinct sections. The first is the experimental design of the project itself – its construction as the union of independent modules. The second is the design of the MIDI to Arduino circuit. The last is the design of the primary Tesla coil circuit. Some physical elements were designed and implemented, but to such a small degree as to be uninteresting.

Overview

The project is designed to break apart into manageable, independent modules. These modules are shown in Table 4.

Table 4: A breakdown of the units necessary for a successful project

From	To	Means
Piano/Computer	MIDI Stream	Built in to Q61 Keyboard
MIDI Stream	Arduino Serial In	Optocoupler circuit (see Section 3.1)
Arduino Serial In	MIDI events	Code (MIDI library)
MIDI Events	Timers	Code (see Section 2.3.2)
Timer Handlers	Electrical Pulses	Code (direct port access)
Electrical Pulses	Primary Circuit (logic half)	Transistors/Fiber Optics
Primary Circuit (logic half)	Primary Circuit (power half)	Many logic and power ICs
Primary Circuit (power half)	Resonance in Secondary	Tuned LC Circuits
Voltage in Secondary	Electrical Arcs	Breakdown

With this design, each small problem could be tackled individually, then merged together to form a successful final project. A rough flow chart of the project is shown in Figure 8.

3.1 MIDI to Arduino Circuit

The goal of the MIDI to Arduino circuit is to convert incoming MIDI signals into Serial data that the Arduino can read and understand. In this project, we only use the MIDI output of an Alesis Q61 piano keyboard or a computer, but in theory the incoming MIDI data stream could come from any MIDI-enabled instrument. In order to use a computer as a MIDI source for the Arduino, a USB-to-MIDI cable must be employed, since no computers have native MIDI out ports.

Incoming MIDI data, transmitted over a MIDI-to-MIDI or a USB-to-MIDI cable, is converted into a breadboard-compatible format by a female MIDI connector (5 pin DIN 180° connector) from SparkFun. This component allows male MIDI cables to be directly 'plugged into' a breadboard without the need for extraneous soldering. [30] The MIDI connector we use is shown in Figure 9. As per the datasheet, the pins correspond to MIDI connections 3,

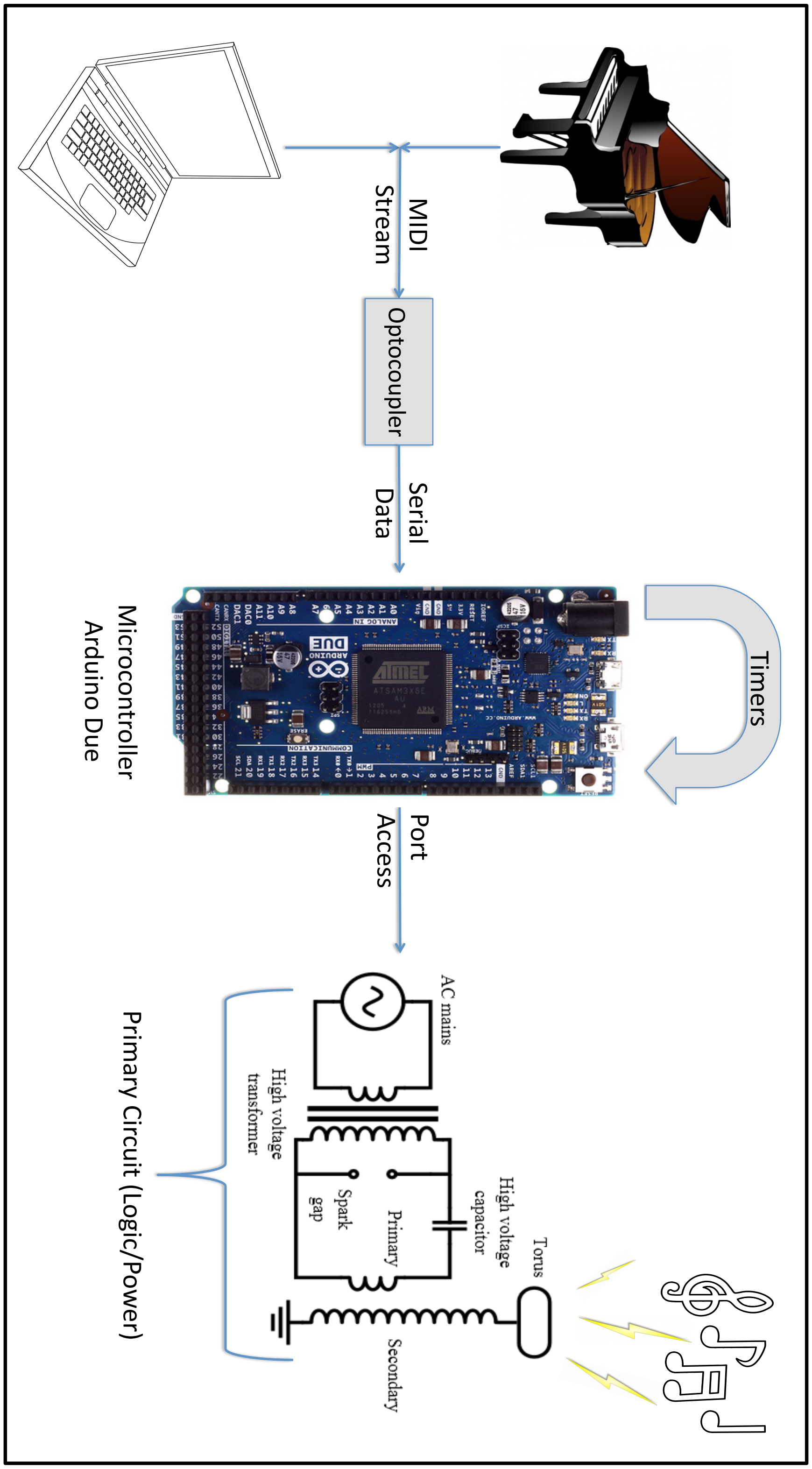
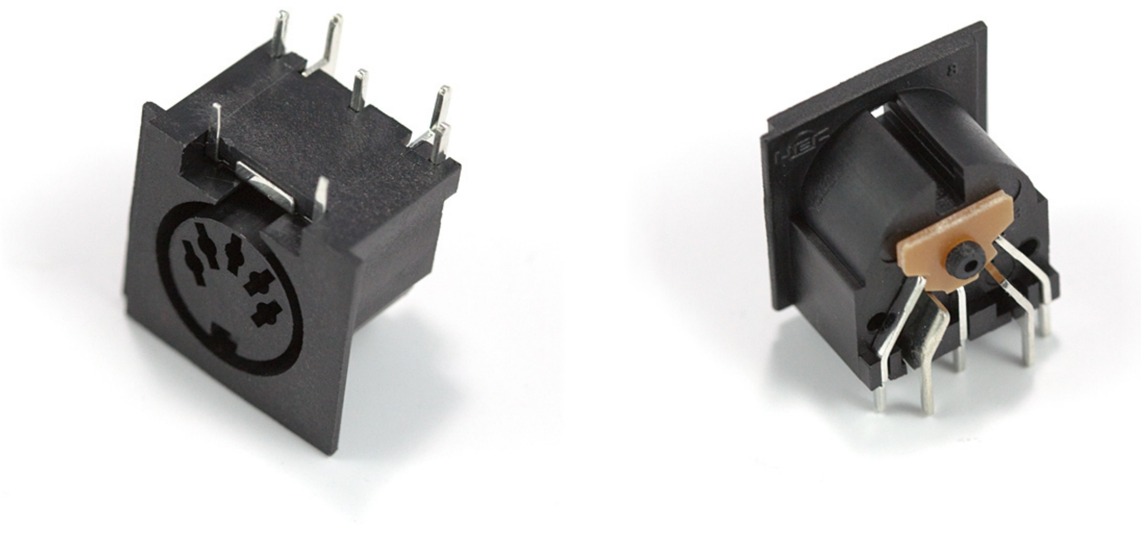


Figure 8: Abstract diagram of the flow of data in the project and the tools used to process it.



(a) Front view **(b)** Rear view

Figure 9: Two angles of the SparkFun breadboard-friendly MIDI connector.

5, 2, 4, 1 from left to right when viewed from the front with the pins in a convex up (smiling) configuration.[31]

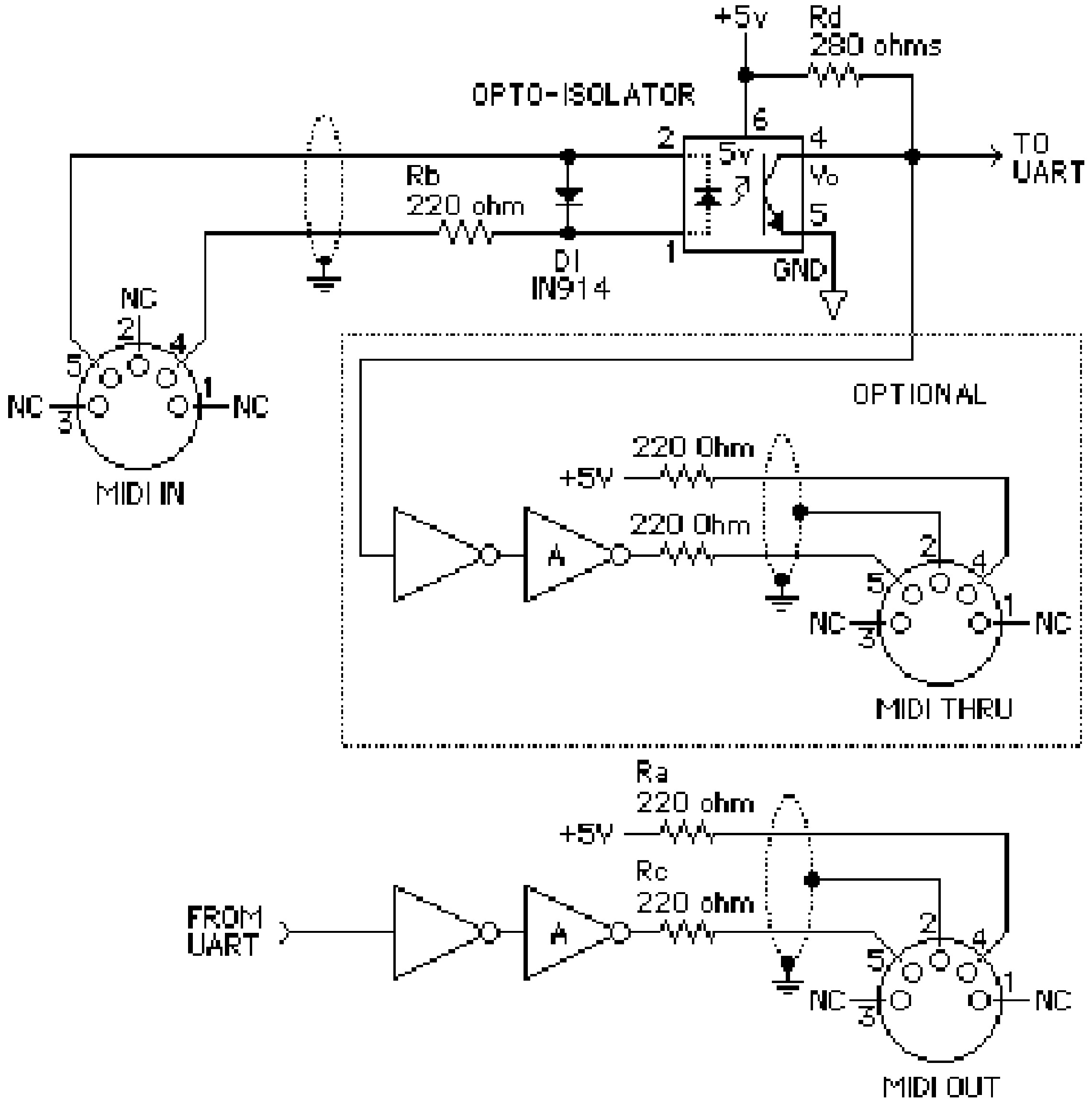
The rest of the circuit roughly follows the MIDI to DIN electrical specifications, published by the MIDI Manufacturers Association. [32] Their circuit is shown in Figure 10.

Our slightly modified circuit is shown in Figure 11. We have only changed the diode (from 1N914 to 1N4148), the optocoupler (from PC-900 to 6N138) and the value of the drain resistor (280Ω to 470Ω).

We use an optocoupler to isolate the MIDI side of the circuit from the Arduino side of the circuit for multiple reasons. First, the optocoupler allows us to maintain separate grounds on the MIDI and Arduino side of the circuit. Since a MIDI signal consists of a series of voltage spikes, if the signal and receiver circuits are not optically isolated, loops of current through the common ground will corrupt the signal.

Another benefit of the optocoupler is to fix any signal degradation that may have occurred over the MIDI cable. After all, MIDI provides no error-correction procedure in its protocol, so random fluctuations may (and do!) significantly interfere with the quality of the signal. Therefore the optocoupler not only removes minor noise fluctuations but also brings the 'high' voltage of the MIDI signal to a constant amount. (See Section 4.1.)

After trying a variety of optocouplers, we settled on the 6N138 optocoupler. Although it is an active component, its switching speed is extremely fast (less than $10\mu s$ for both rise and fall).[33] On the master side, pin 2 is the cathode of the internal photodiode (LED), and pin 3 is the anode. On the slave side, pin 8 is power, pin 7 can increase switching speed if it has a pull-up resistor, pin 6 is the output, and pin 5 is ground. The arrangement of



Copyright 1985 MIDI Manufacturers Association

Figure 10: Electrical specification from the MIDI Manufacturers Association for the conversion of a MIDI In signal to a UART (top circuit). Specifications are also given for MIDI Thru (middle circuit) and MIDI Out (bottom circuit).

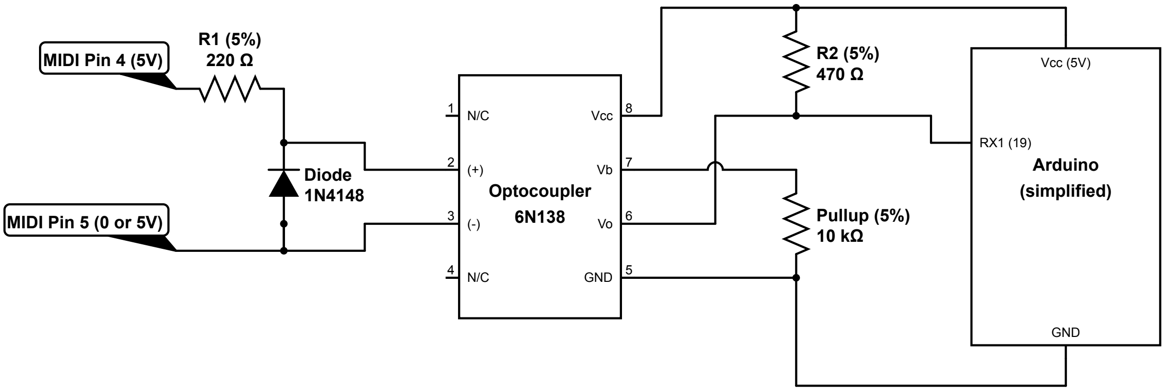


Figure 11: Schematic of the MIDI to Arduino circuit. The MIDI transmission is carried on pins 4 and 5.

the Darlington pair of transistors in this component ensures that pin 6 will be at ground if and only if there is a voltage across pins 2 and 3. The schematic for the 6N138 is shown in Figure 12.

In Figure 11, R1 drops the voltage across the optocoupler's inputs in order to protect the photodiode (LED) in the rare case that the MIDI provider does not adhere to the suggested standard of providing MIDI power from 5V through a 220Ω resistor. The maximum recommended voltage across the LED is 1.7V, and it can break down at as little as -5V, so the resistor keeps the LED from breaking. [33] The photodiodes of other optocouplers may break down at even lower voltages, so the resistor is a good safety precaution, although not strictly speaking necessary. The 1N4148 diode is a fast-switching diode capable of recovering from forward conduction in at most 4 nanoseconds. It has replaced the 1N914 originally suggested by the MIDI Manufacturers Association, because it has the same electrical specifications but slightly lower leakage current ($25nA$ vs. $5\mu A$) and a slightly higher maximum average forward current (150mA vs 75mA). [34] We use this diode in order to minimize feedback loops which distort the MIDI signal, and to protect the photodiode of the optocoupler from improperly wired MIDI pins, where pin 5 is always at a higher voltage than pin 4.

The optocoupler setup maintains the logic of MIDI pin 5, which carries the MIDI data. If the MIDI source sends a logic 1, MIDI pin 5 will be at 5V, so there will no voltage difference between pin 2 and pin 3 of the optocoupler. Therefore, pin 6 of the optocoupler will not be connected to ground, and the Arduino's RX pin will read a high voltage (by way of R2), corresponding to the MIDI source's original logic 1. We increased the value of the drain resistor because the Arduino Due only accepts voltages on I/O pins between 0 and 3.3V. By increasing the value of the drain resistor, we lower the voltage incident on the Arduino's RX

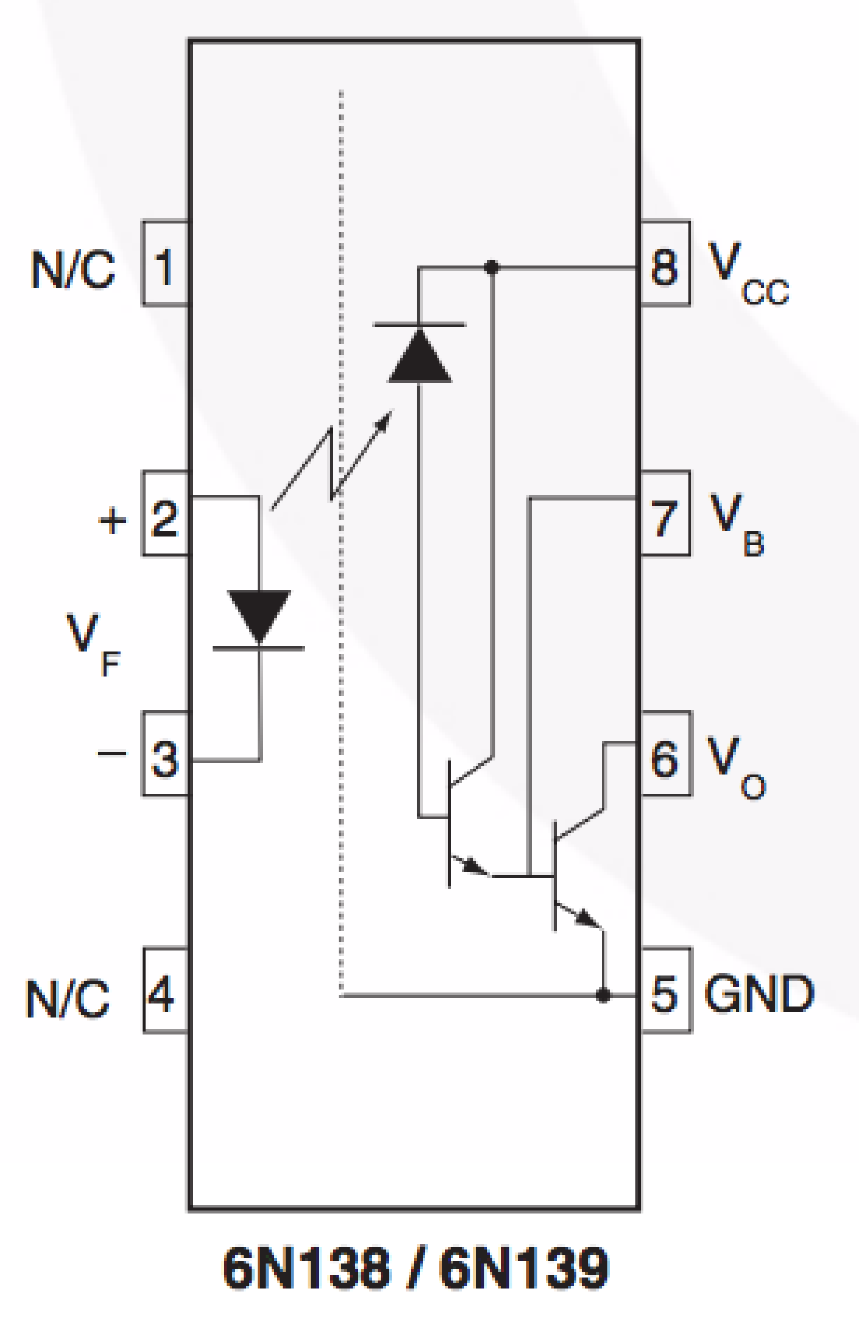


Figure 12: Schematic for the 6N138 Optocoupler. Pin 2 is the cathode, and pin 3 is the anode of the control circuit. On the slave circuit side, pin 8 is power, pin 5 is ground, pin 6 is output voltage, and pin 7 (optional connection) is a pull-up that increases switching speed. Pin 6 can drain to ground if and only if there is a voltage across the input LED.

pin (which is an I/O pin) so that we are less likely to damage the Arduino's microcontroller.

Conversely, if the MIDI source sends a logic 0, MIDI pin 5 will be at 0V, so there will be a voltage difference across the photodiode. Then, pin 6 of the optocoupler will be connected to ground (through the 'inner' transistor in the Darlington pair), and so the Arduino's RX pin will see 0V, correctly corresponding to the original logic 0 from the MIDI source.

The incoming Serial data is processed by a MIDI library which calls function handlers in response to different MIDI status values.

3.2 Primary Circuit

The overall goal of the primary circuit is to create a series of electrical breakdowns (from the top conductor to ground) exactly when a signal is on. To repeat, we want to be able to control exactly when arcs occur by sending electrical signals.

The original design for this Tesla coil's primary circuit is an adaptation of the circuit used by oneTesla. (See Figure 13.)

The overall circuit is comprised of two halves. There is a power half, which contains high voltage and high frequency electricity, and a logic half, which controls the current in the power half. We'll begin by looking at the power half, and conclude by discussing the logic half.

3.2.1 Power Circuit

The primary circuit's power half consists of two parts: (1) a voltage double and rectifier, and (2) a half bridge.

Ultimately, the Tesla coil is powered by electricity from a wall socket. There are usually three holes in an electrical socket. These three pins are associated with hot, neutral, and ground. The hot connection fluctuates at 120V AC at 60HZ. The neutral connection is held at zero voltage, referenced to Earth's voltage. The ground pin is optional and just for safety; it is frequently connected directly to ground via a stake in the ground or something similar. The neutral pin is used to carry current back to ground, whereas the ground pin is not.

Let's take a closer look at the voltage double and rectifier. When the AC power supply voltage is negative, the top diode conducts and the top capacitor charges. When the AC power supply voltage is positive, the bottom diode conducts and the bottom capacitor charges. The voltage drop across each capacitor is equal to the voltage of the AC power supply, so the overall voltage drop across both capacitors (which are in series) is twice the source voltage – hence, a doubler. Note that the output voltage measured across the resistive load will still pulsate, although all the voltage is in the same direction and the maximum voltage is twice the source voltage of our power supply.

We use 2 MUR460 power diodes (rated at 60A, 1000V DC) to rectify the AC, and two large capacitors to smooth out the bumps (3188FH222M350AMA2 - 2200 uF +- 20%, 250VDC, 400 surge, 85 max c ambient).

The half bridge is what directly drives the oscillations in the primary circuit that allow the Tesla coil to build up. In theory, we pulse the DC voltage across the LC circuit, initiating

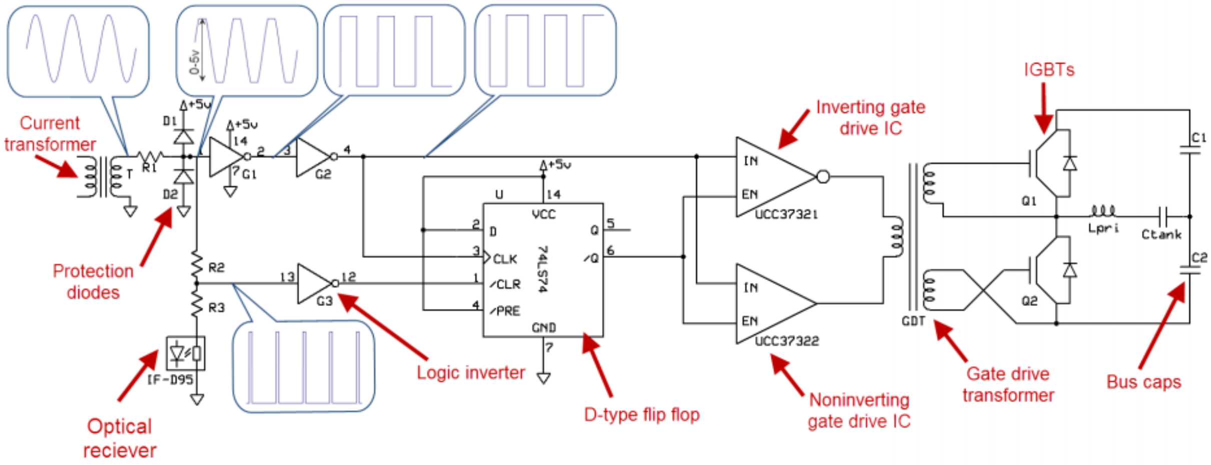


Figure 13: Schematic published by oneTesla for the primary circuit. Before the current transformer is a rectifier and doubler circuit that converts 120V AC to 240V DC

oscillations. We alternate the direction of the voltage across the tank capacitor and inductor to induce AC current in the primary circuit, much like the oscillations in an LC circuit. For this to work, we need to switch the DC voltage at the resonant frequency of the Tesla coil.

3.2.2 Logic Circuit

The logic circuit is divided into two parts. There is a signal side and an output side.

The two inputs to the logic circuit are an interrupter signal and a feedback signal. The interrupter signal is from the output of the fiber-optic receiver, and corresponds to the electrical pulses sent out by the Arduino. The feedback signal is from a current transformer jumped off of the main primary circuit (the middle line in the half bridge). A combination of the interrupter signal and feedback signal controls the output of a D-type flip flop, which then determines the output states of two complementary gate drivers. The differential voltage signal between the two gate drivers is routed through a 1:1:1 gate drive transformer, which allows easy switching of the transistors.

The feedback signal is driven by the output of a 300:1 Triad current transformer, which allows us to ‘keep an eye on’ the oscillating current in the primary circuit.

The AC signal is squared up by sending it twice through an inverter, and this signal is fed into our flip flop in conjunction with the inverted interrupter signal. The clocked AC signal is also sent to the inputs of two complementary gate drivers, and the enable line of the drivers is fed by the output of the flip flop. The differential voltage from the output of the drivers powers the gate drive transformer.

3.3 Physical Design

Many of the existing physical objects are left over from Batchelder's project. His components emphasized interchangeability, which makes modifying his system easier. While I left most of his components intact, I will modified the existing primary coil. It was very crooked, with sloppy soldering that allowed sparks to short-circuit to the secondary (through insulation!) from the primary. To that end, I am prototyping a 3D-model of a holder for the primary coil which locks in place (preferably locks into the table) and adds structural integrity to the primary coil. Since the 3D printer's platform is too small for the entire piece to be printed at once, I'm creating separate interlocking parts. The prototype is shown in Figure 14. Since I am still determining measurements, this diagram has no specific dimension and is only intended to give an idea of what the supports will look like.

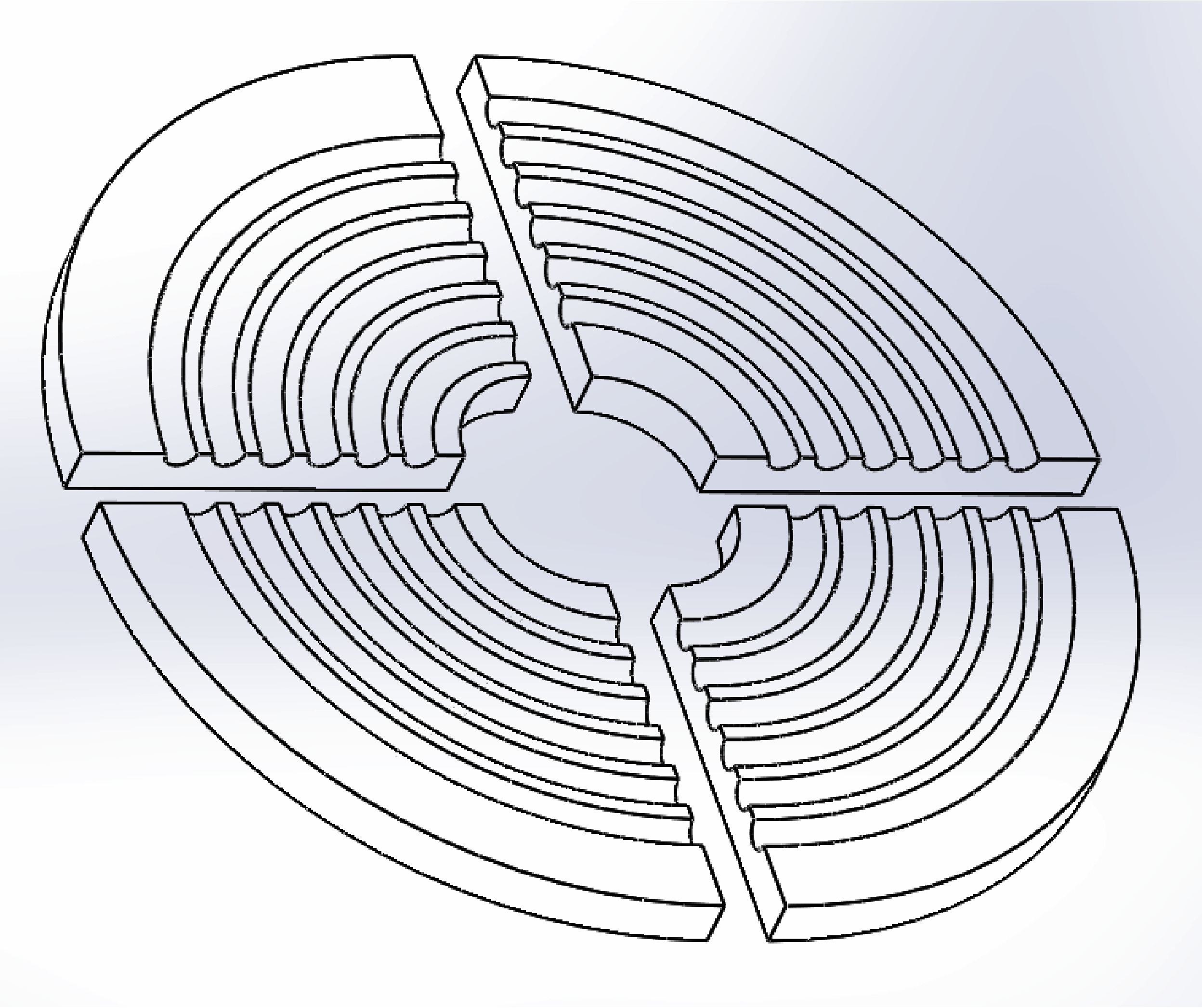


Figure 14: SolidWorks model of the primary coil support.

Also, time permitting, I'd like to redesign the Tesla coil stand to make it more stable. As of now, one of the table's legs is not physically attached to the tabletop and many connections are made by epoxy rather than screws.

4 Results

4.1 MIDI to Arduino

MIDI Before and After Optocoupler

The optocoupler performed admirably in cleaning up the incoming MIDI signal by reducing noise, raising the reference voltage while still maintaining the different bits due to its very fast switching speed.

For consistency in this section, all graphed yellow lines represent the voltage difference between MIDI pins 4 and 5 at the MIDI connector (before the optocoupler), and all graphed blue lines represent the voltage difference between the output (pin 6) of the optocoupler and the Arduino's ground (after the optocoupler).

Figure 15 shows the impact of the optocoupler on the signal quality. Miscellaneous noise is removed, and the max voltage is raised to a level the Arduino can recognize as a binary 1. Note the inverting action of the optocoupler circuit: when the voltage difference between MIDI pins 4 and 5 is low, the optocoupler outputs a high voltage, as expected.

Figure 16 encapsulates the fast logic of the optocoupler. From Subfigure 16a, we see that the propagation delay from the start of a rising signal edge to the start of a falling output edge is $2.6\mu s$. Subfigure 16b shows that the propagation delay from the start of a falling signal edge to the start of a rising output edge is $1.8\mu s$.

Figure 17 shows the falling and rising durations for the output of the optocoupler. Subfigure 17a and Subfigure 17b give the falling and rising times as $300ns$ and $3\mu s$ respectively.

Combining our results, we see that the overall propagation time to a logical low (T_{PHL}) is $2.6{+}0.3 = 2.9\mu s$, and the overall propagation time to a logical high (T_{PLH}) is $1.8{+}3 = 4.8\mu s$. These values are consistent with the typical values of $T_{PHL} = 1.5\mu s$ and $T_{PLH} = 7\mu s$ for the 6N138 optocoupler, specified on page 4 of the optocoupler's data sheet. [33] One reason our values differ from the specifications is that, as described in Figure 24 of the datasheet, the cutoff for both 'sufficiently low' and 'sufficiently high' is 1.5V, whereas our tests consider 'sufficiently low' to be 0V and 'sufficiently high' to be 4V. If we use Fairchild's cutoffs to estimate our optocoupler's switching speeds, we find $T_{PHL} = 2.5\mu s$ and $T_{PLH} = 2.4\mu s$, which is more self-consistent. Either way, the optocoupler is clearly switching incredibly quickly. It is this high-speed switching that allows for the creation of such high-quality MIDI data for the Arduino to read.

Arduino Reads MIDI data

The software works exactly as intended. Using the MIDI library, the Arduino was able to read in a sequence of voltages and call a desired function, such as HandleNoteOn or HandleSystemReset according to the type of MIDI signal. Sample Arduino **Serial** output might be NOTE ON: Channel 1, Pitch 47, Velocity 78. Computed Frequency 123.47Hz.

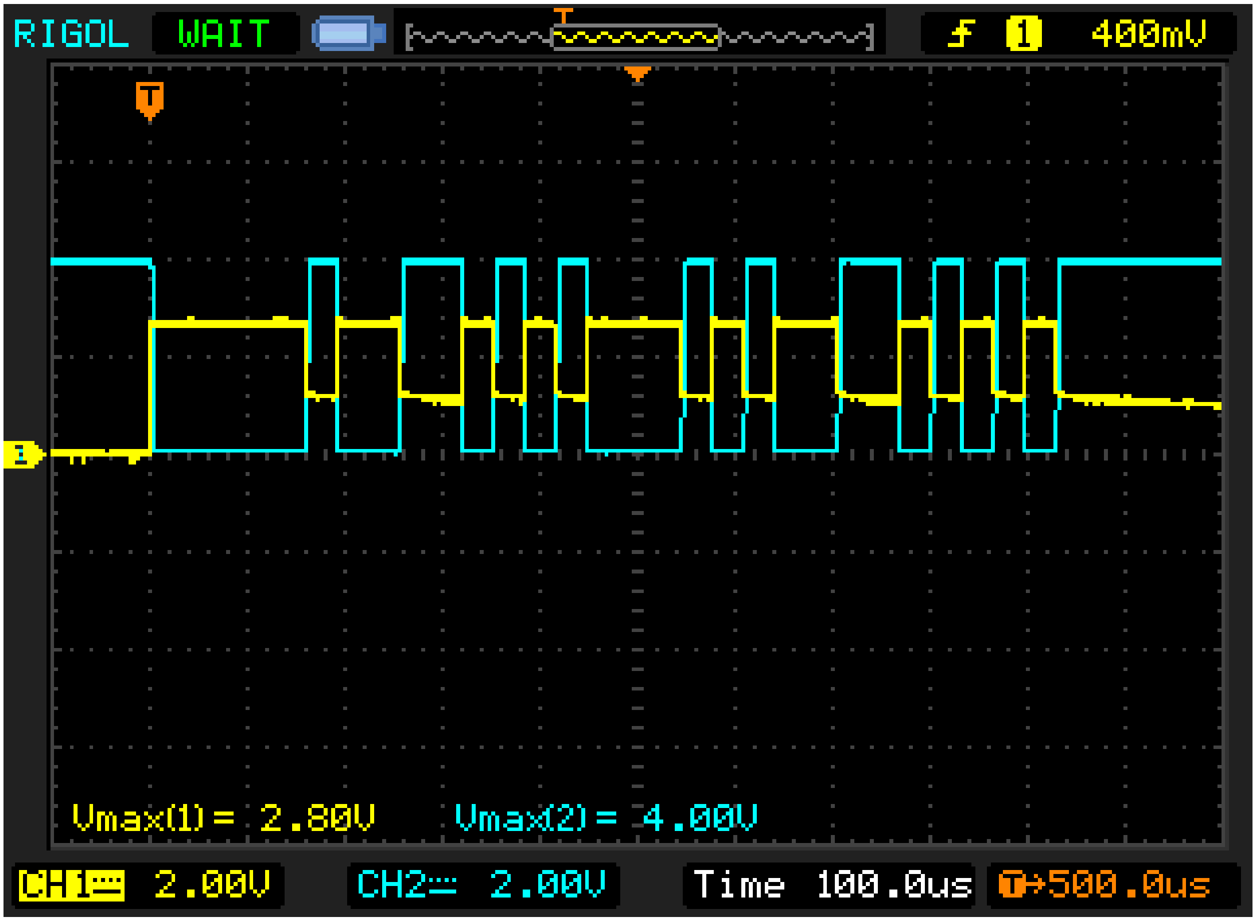
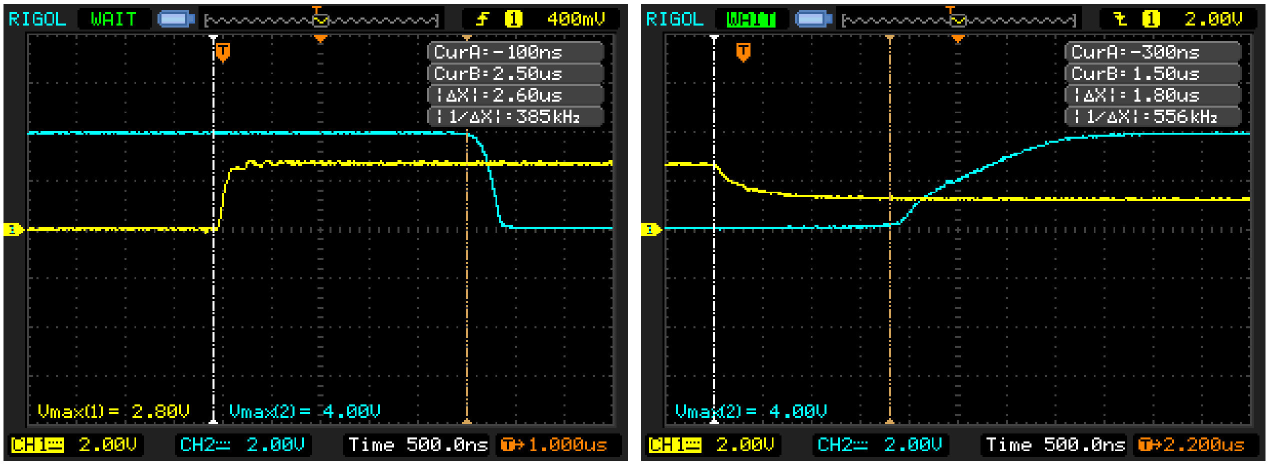


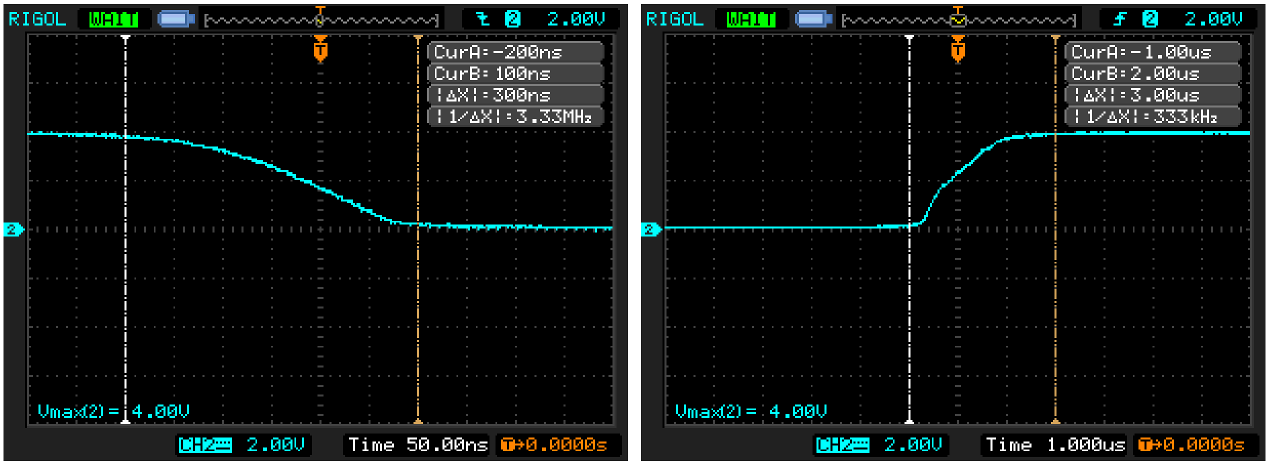
Figure 15: Incoming MIDI data before and after the optocoupler.



(a) Rising Signal Edge Propagation Delay

(b) Falling Signal Edge Propagation Delay

Figure 16: Signal propagation delay of optocoupler. The voltage and time scales are identical.



(a) Duration of Falling Edge

(b) Duration of Rising Edge

Figure 17: Fall time and rise time for the optocoupler output. The time scales are emphatically different, but the voltage scales are the same.

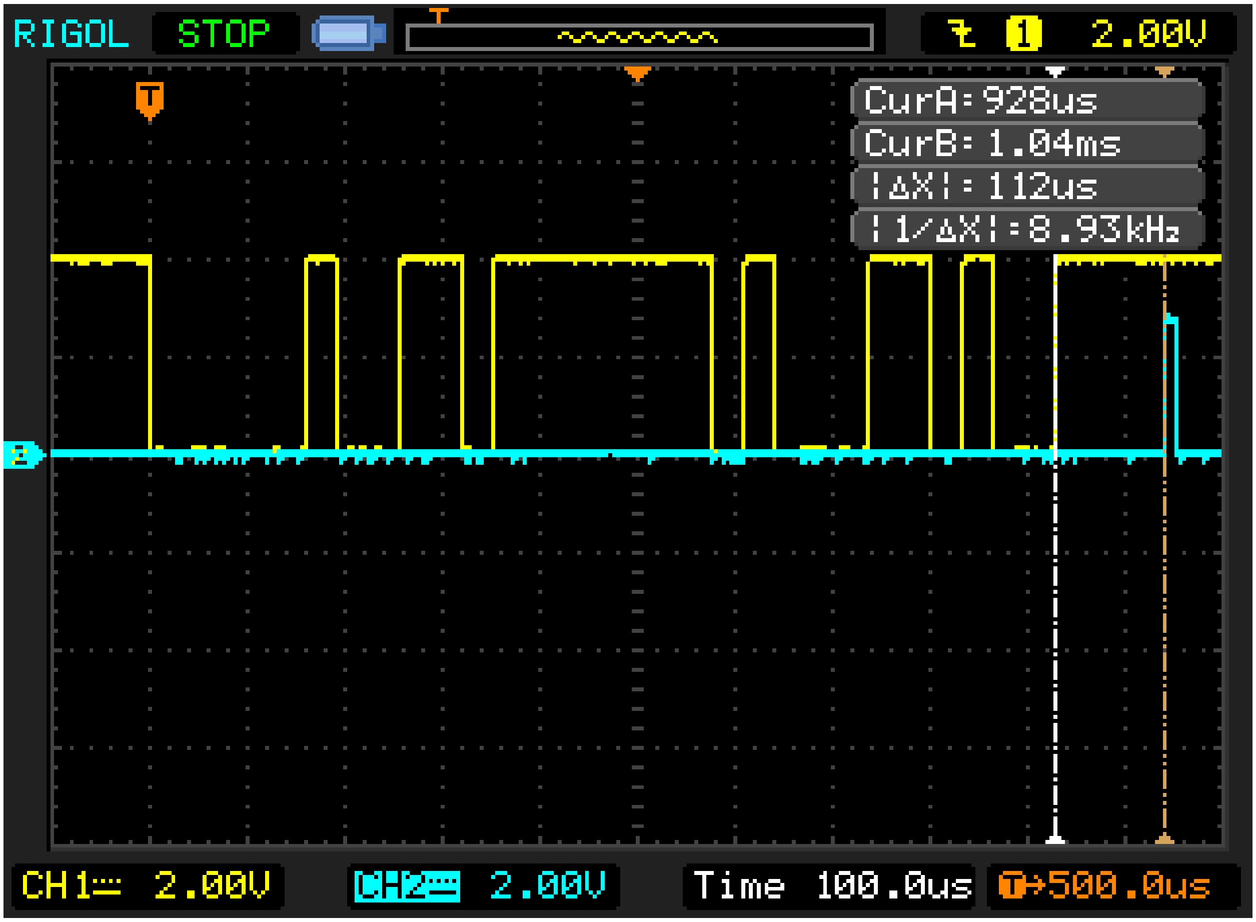
4.2 Arduino In (MIDI) to Ardiuno Out (PWM)

All of the optimizations we made in the Arduino code were effective. These results are the most satisfactory, because they prove that all of the code came together to function as desired.

Processing Delay

The biggest speed challenge was in interpreting a sequence of high and low voltages as a MIDI message and starting a timer at the appropriate frequency. Our code utilizes the power of the MIDI library as well as the Due Timer library to quickly process data and start the timer. Figure 18 shows the overall delay between the incoming MIDI message and the first outgoing electrical pulse. Shockingly, the Arduino was able to respond to the incident stream of voltages in just $112\mu s$! However, this defines the upper bound of the possible delay, because the final MIDI data byte in reality ends with a high (1) stop bit. Since the MIDI protocol operates at 31,250 bps, this final high bit (which is 'hidden' in the post-MIDI voltage stream) has a duration of $32\mu s$. At what point during the $32\mu s$-long high bit does the Arduino read in the voltage as a digital 1 into the Serial input buffer? We have no idea. As far as our research suggests, there is no documentation on precisely when the Arduino reads in the final high stop bit. Therefore, the processing delay could be as low as $80\mu s$! If we assume the Arduino reads in the voltage value uniformly at random from the $32\mu s$ interval, then the expected value of the processing delay is $96\mu s$, still less than $100\mu s$.

This processing delay, however, takes into account the delay incurred both by the MIDI library and by the Due Timer library. Figure 19 shows the processing delay to interpret the MIDI signal and call the associated function handler (tasks performed by the MIDI library). The blinkLED method was called from within the body of the NoteOn method,



CurA: 928us
CurB: 1.04ms
|ΔX|: 112us
|1/ΔX|: 8.93kHz

Figure 18: The processing delay of the Arduino from reading in a MIDI signal to the initiation of a timer's first generated output pulse. Since the Arduino could read the final high stop bit anywhere in the last bit's time interval, the delay could be as little as $80\mu s$.

albeit after three `if` statements. Therefore, the visible delay represents the time taken to completely analyze and interpret the incoming MIDI signal. As before, we don't know when in the last $32\mu s$ the Arduino actually finishes reading the final MIDI data byte (returning it to a `Serial.read` call) – hence the lack of cursors; however, we can approximate that the MIDI processing code took between $0\mu s$ and about $20\mu s$ to execute. Again, if we assume the Arduino read in the code at the halfway point, we get an expected processing duration of $4\mu s$ – extremely fast!

Note: The following tests were conducted with the user output disabled (`useSerial=False;`, `useLCD=False;`). If the booleans `useSerial` or `useLCD` are set to `True`, the processing delay will be much greater (empirically about 10 to $15ms$, depending on the length of the output message). Since this latency is incurred only once (when starting the note), it doesn't matter much to the overall sound. Nevertheless, it is satisfying to see the rapid response of the code.

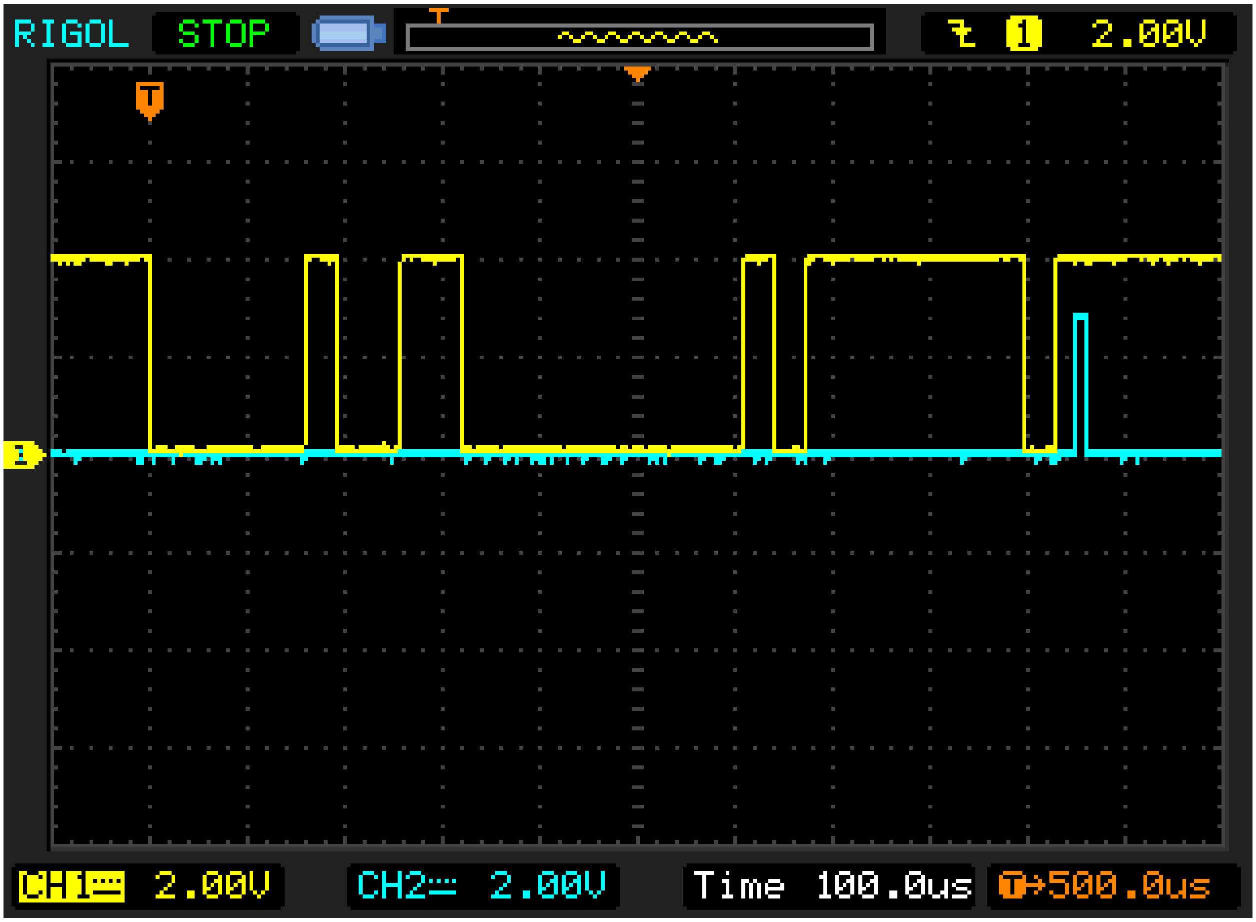


Figure 19: The processing delay between a MIDI signal and the execution of an associated function.

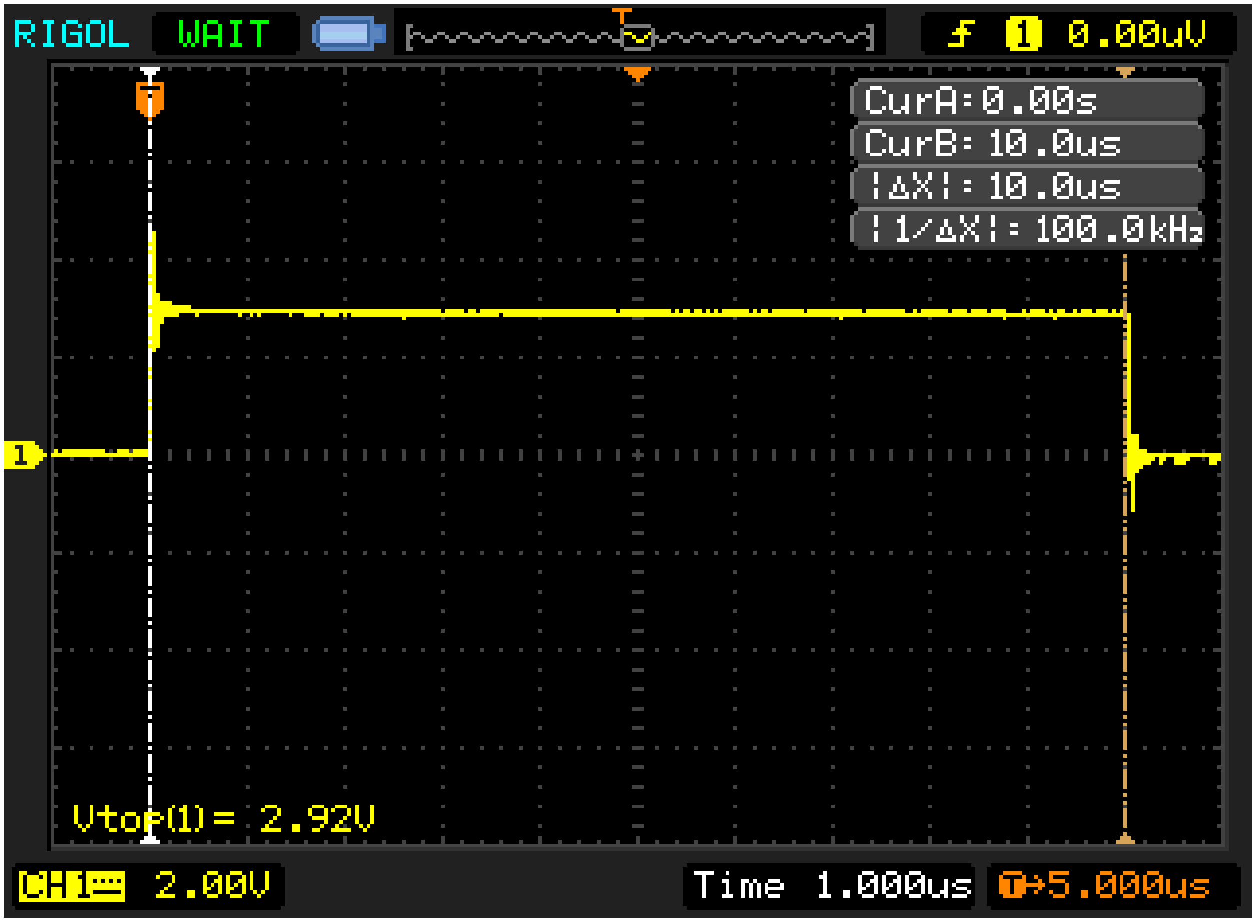
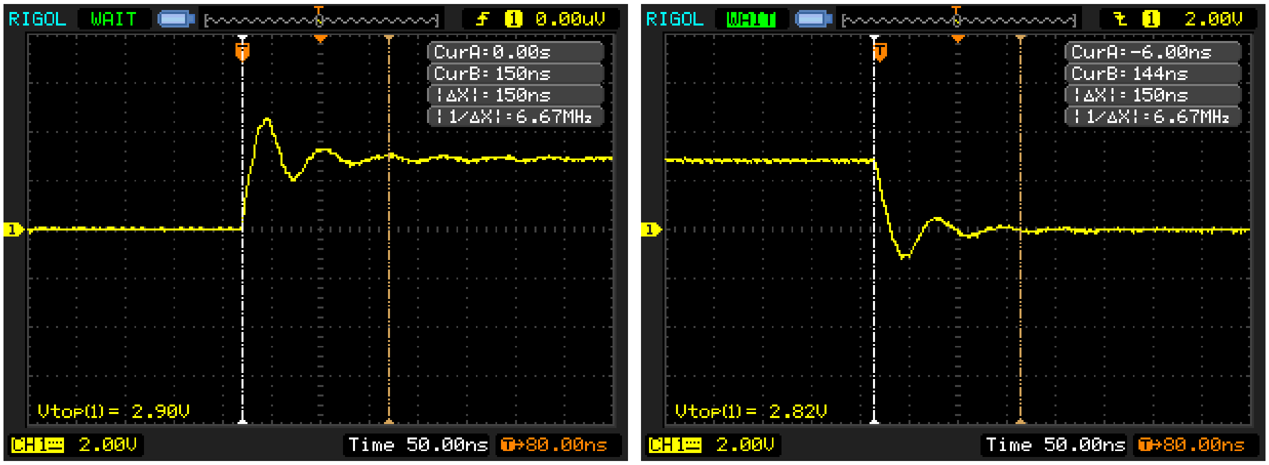


Figure 20: The pulse generated by the Arduino. The duration is controlled by the built-in delayMicroseconds function, which is accurate enough for our purposes.

Characteristics of Output Pulse

We programmed the Arduino to generate a sequence of output pulses that are (a) a specific duration and (b) as square as possible. For (a), we used the Arduino's built-in delayMicroseconds function, which is accurate to values as low as $3\mu s$, at least in Atmel-based microcontrollers. Figure 20 gives an overview of the pulse generated by the Arduino as well as a measurement of its width: exactly $10\mu s$, as expected.

The most impressive feature of these generated pulses is the sharp rise and fall edges. Figure 21 shows a zoomed-in version, giving an approximate measurement of the rise and fall times on the edges of this pulse. The Arduino takes $150ns$ to switch from a stable low output voltage to a stable high output voltage or vice versa. However, if we look even more closely, as in Figure 22, we see that the inital rise time (before stabilization) is $12ns$, and the initial fall time (before stabilization) is $14ns$. At the Arduino Due's clock speed of 84MHz, these blisteringly fast switches correspond to roughly one clock cycle ($\frac{1s}{84{,}000{,}000} = 11.9ns$). Therefore, these figures are proof that direct port manipulation is an extremely fast


RIGOL WAIT 0.00uV
CurA: 0.00s
CurB: 150ns
|ΔX|: 150ns
|1/ΔX|: 6.67MHz
Vtop(1)= 2.90V
CH1 2.00V Time 50.00ns T→80.00ns

RIGOL WAIT 2.00V
CurA: -6.00ns
CurB: 144ns
|ΔX|: 150ns
|1/ΔX|: 6.67MHz
Vtop(1)= 2.82V
CH1 2.00V Time 50.00ns T→80.00ns

(a) Stable rise time of generated pulse.

(b) Stable fall time of generated pulse.

Figure 21: Stable transition times of the generated pulse. The fast speed is due to the use of direct port manipulation rather than digitalWrite.

alternative to the clunky digitalWrite, which can take up to $20\mu s$ to execute.

True Delay Between Notes

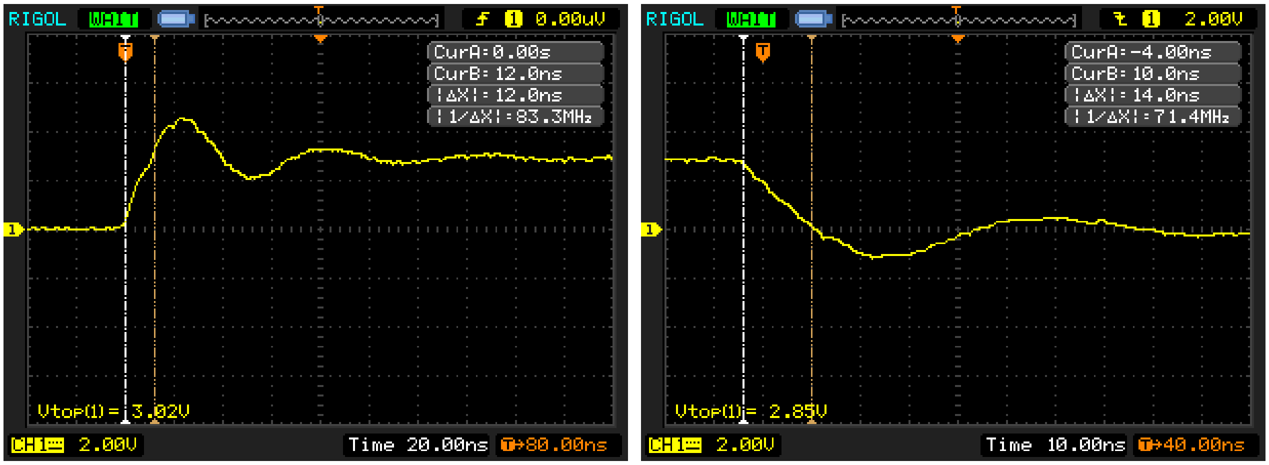
Another key component of our code design was the use of asynchronous timers to ensure that the pulses corresponding to any note will be perfectly spaced according to that note's frequency.

Figure 23 displays the measured period of the highest-frequency note (MIDI Note 127), which corresponds to $12543.854Hz$, or an ideal period of $79.720\mu s$. Even though measuring the period with the oscilloscope's cursors introduces some error in resolution, the apparent period generated by the Arduino in response to MIDI note 127 is $79.6\mu s$, a near perfect replica of the desired result (off by 0.1% or less).

Figure 24 shows the measured period of the lowest-frequency note (MIDI Note 0), which corresponds to $8.176Hz$ or an ideal period of $122.312ms$. At such a large time scale, the oscilloscope only gives resolution to the millisecond. Nevertheless, we were right on the money, measuring a period of $122ms$. For this trial, we increased the pulse width to $1000\mu s$ so that the pulses would be visible even at such a zoomed-out view. Nevertheless, this increased pulse width did not affect the overall period of the pulses (as one might expect by the delay call in the blinkLED function), further proof that the asynchronicity is working as desired.

Support for Multiple Notes

The primary reason for using asynchronous timers to control pulse generation was to support multiple notes at once. Figure 25 demonstrates two concurrent notes. Visually, it is hard to distinguish more than two concurrent notes; however, this system supports up to nine



(a) Instantaneous rise time of generated pulse.

(b) Instantaneous fall time of generated pulse.

Figure 22: Instantaneous transition times of the generated pulse. The fast speed is due to the use of direct port manipulation rather than digitalWrite.

concurrent notes, each running on a different one of the Arduino Due's nine asynchronous timers.

Transmission Across Fiber Optics

While the Arduino was able to generate these output pulses, we needed a way to quickly send them over to the logic circuit. We used a fiber optic transmission system to send these signals from the Arduino's output to the logic circuit. While the original design featured a transistor (to raise the 3.3V output from the Arduino to the more conventional 5V input to the fiber optic transmitter), we ultimately decided to power the fiber optic transmitter directly from the Arduino's output pin to save time and remove complexity.

Figure 26 shows the turn-on and turn-off delay times between the voltage across the fiber optic transmitter (shown in yellow) and the voltage out of the fiber optic receiver (shown in blue). Interestingly, these components seemed to be switching much slower than their specifications. The fiber optic transmitter specifies a rise and fall time of $100ns$ [35] and the receiver specifies a rise and fall time of $70ns$ [36].

4.3 Primary Circuit

The primary circuit was not successful as a whole. However, portions of the circuit functioned exactly as expected. Unfortunately, data is not readily available for all of the following claims, due to time constraints and a make-it-work-first-record-data-later mindset.

On the power side, the doubling rectifier was very successful. The diodes rectified the AC from the wall socket (although we used a variac as a proxy), and the capacitors smoothed out the bumps in the AC. In fact, the voltage difference across the power supplies looked smoother than the DC input we were using from a power supply to power the logic side.

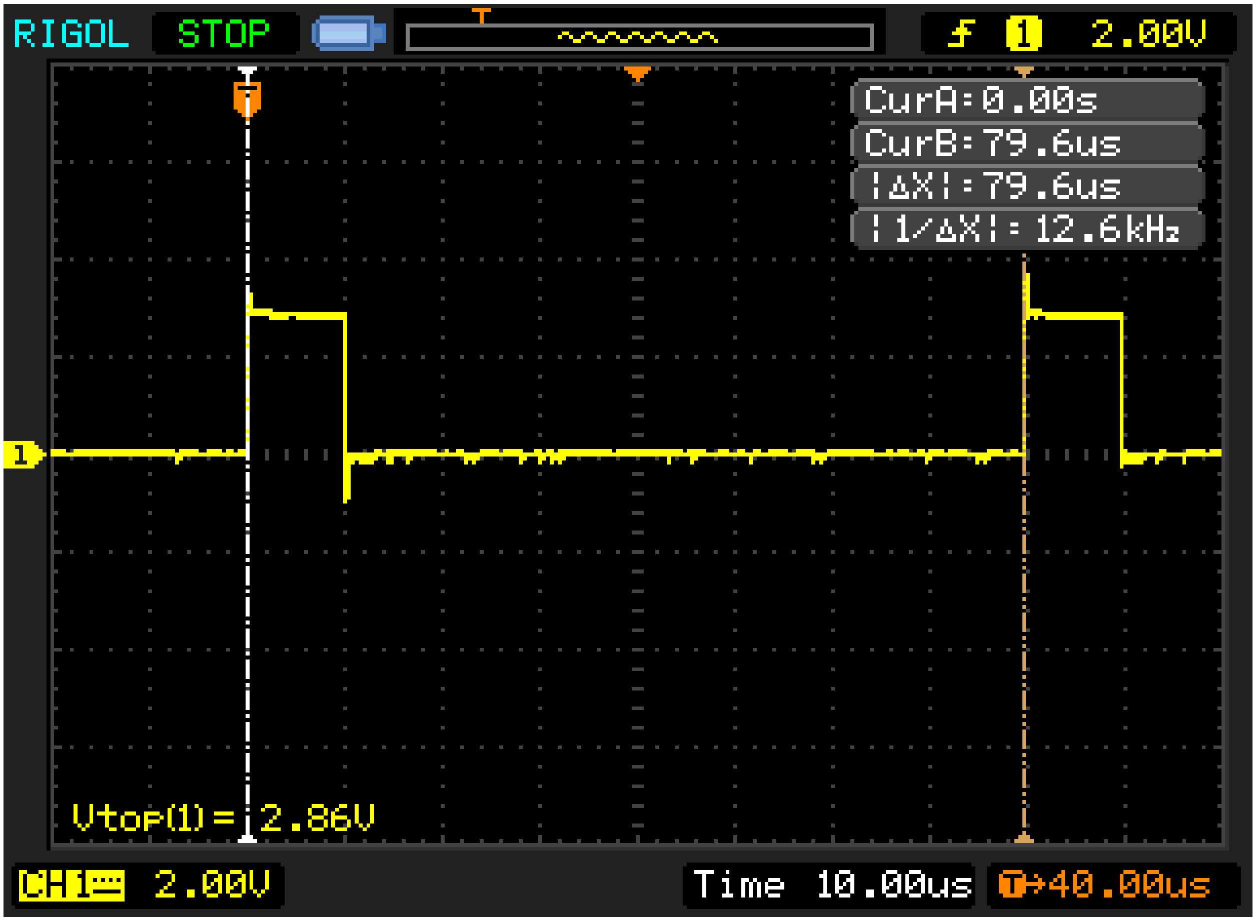
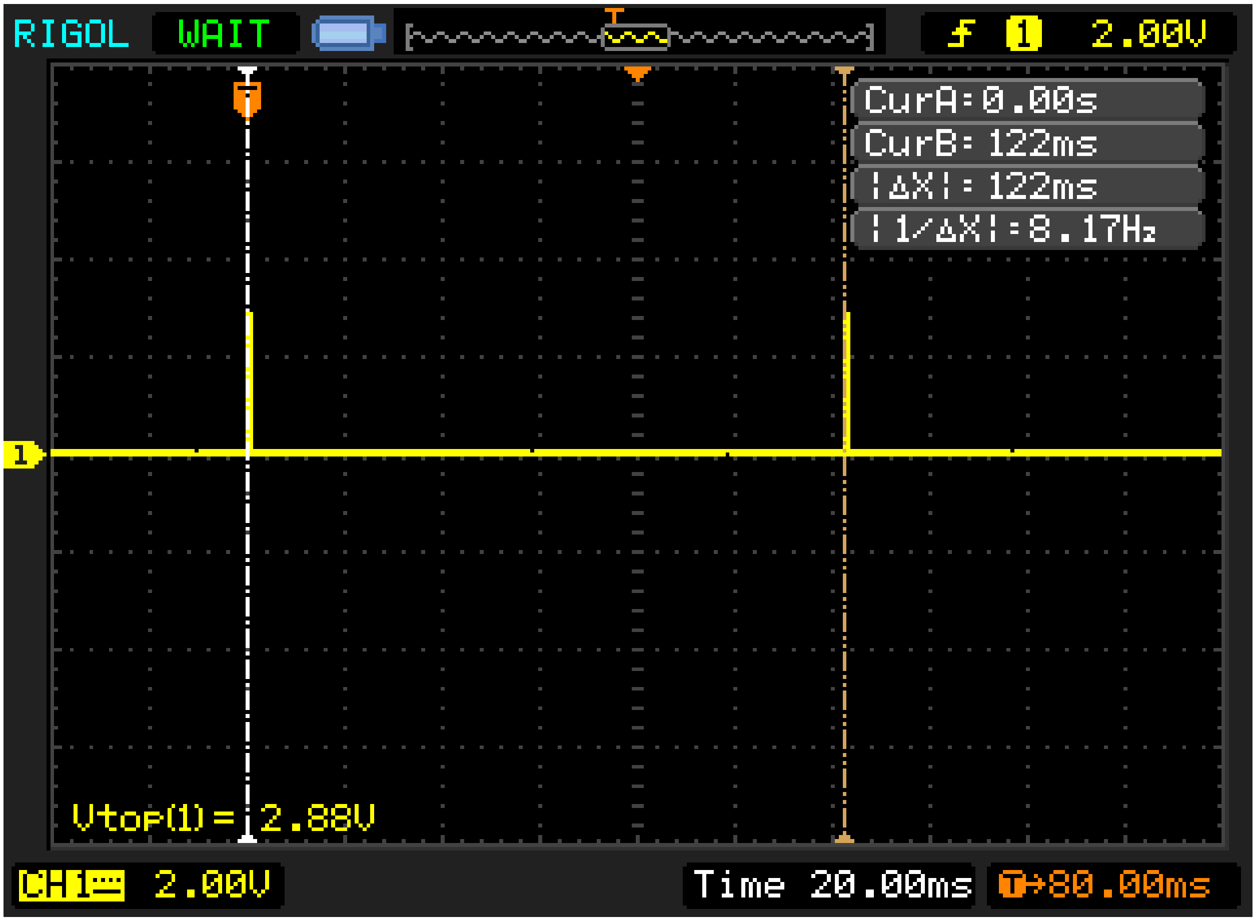


Figure 23: The measured time interval between pulses corresponding to MIDI Note 127. The expected value is 79.72Hz, and the measured value is 79.6Hz - almost perfect. Note: The pulse width is 10μs in this image.



CurA:0.00s
CurB:122ms
|ΔX|:122ms
|1/ΔX|:8.17Hz

Vtop(1)=2.88V

CH1 2.00V
Time 20.00ms
T→80.00ms

Figure 24: The measured time interval between pulses corresponding to MIDI Note 127. The expected value is $79.72Hz$, and the measured value is $79.6Hz$ - almost perfect. Note: For visualization purposes only, the pulse width is $1000\mu s$ in this image.

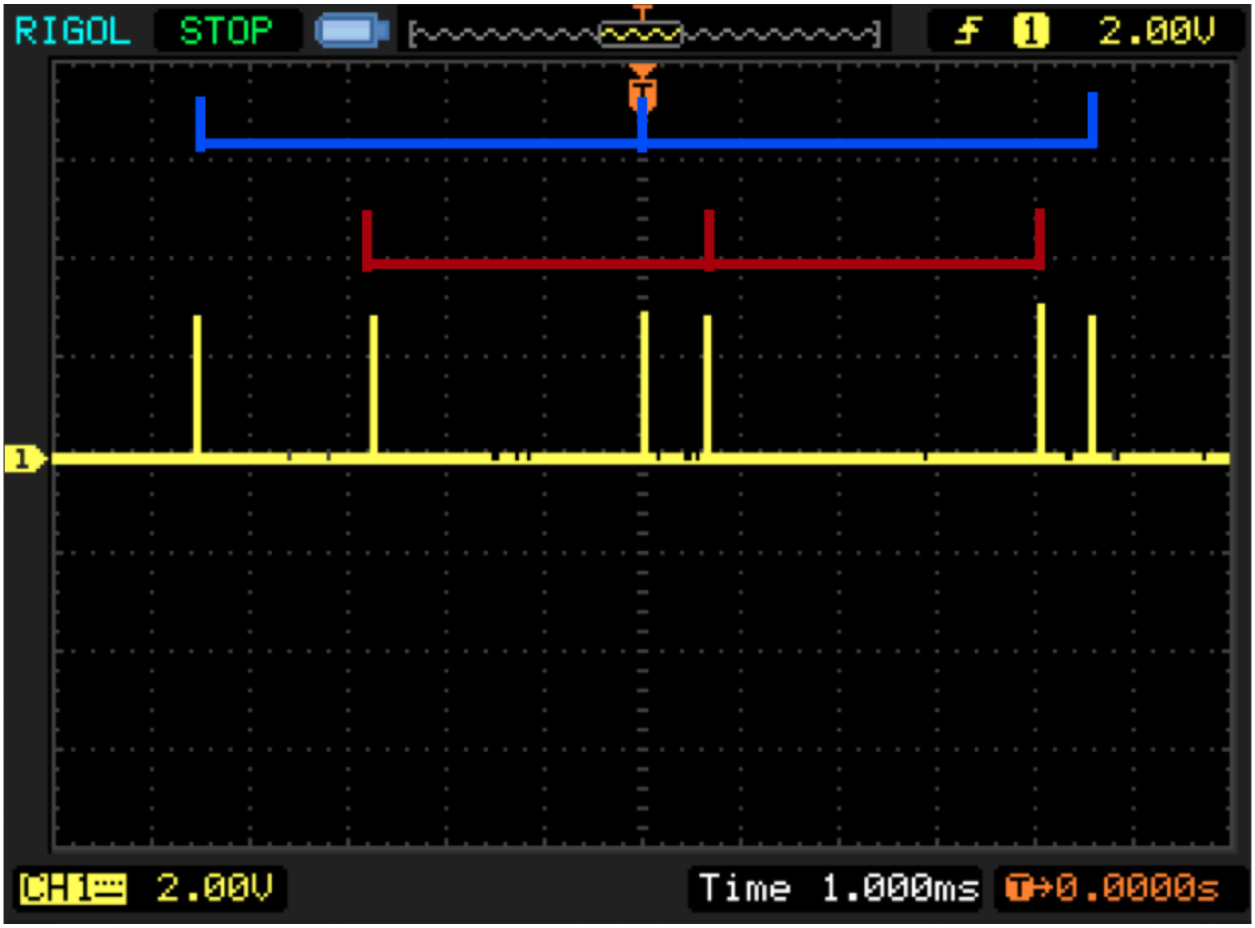
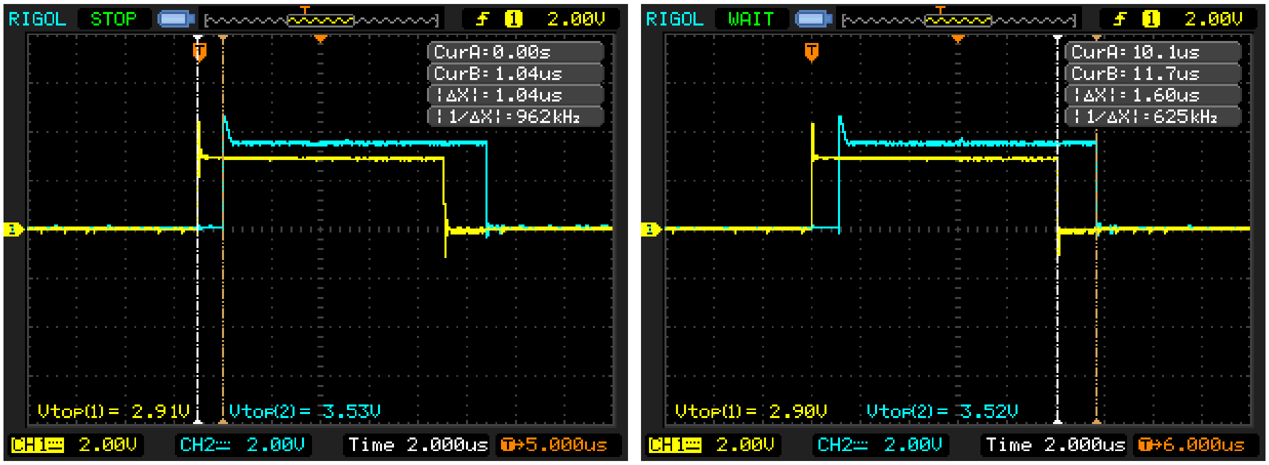
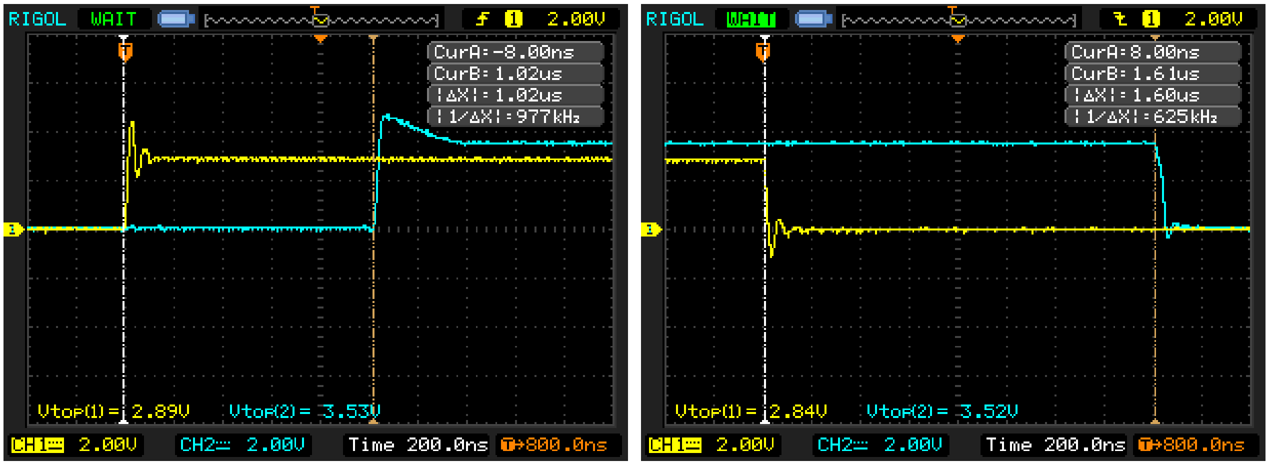


Figure 25: Two concurrent notes being played. The blue sequence of notes (first, third, and sixth spike) correspond to one frequency, and the red sequence of notes (second, fourth, and fifth spike) correspond to another.



(a) (b)

Figure 26



(a) (b)

Figure 27

The current transformer, too, seemed to be functioning well. However, we were not able to induce oscillations in the primary circuit, so it is unclear whether the current transformer would have worked. It did, however, give us an accurate readout of the current rise in an RC circuit (which is how the primary circuit behaved with malfunctioning transistors).

The transistors were able to switch very, very quickly. However, towards the end, we ran into trouble making sure the transistors still functioned as NPNs, because we had perhaps blown them out.

On the logic side, the myriad of protective capacitors and resistors would only be relevant in the case of a ground surge, which we were unable to replicate. All power indicator LEDs and fans worked as expected. The voltage regulators output a constant voltage (after many hours of fiddling and burning ourselves, however). While we were unable to take conclusive data on the output of the flip flop or the gate drivers, we did get to see the not gate in action.

The output of the current transformer should have been AC, so we approximated it with the output of a variac. To be fair, the current transformer's output would have been at the frequency of the Tesla coil (in the high KHz to MHz range), whereas the frequency of the variac is 60Hz. So while the variac did not let us test the speed of our logic side, it did allow us to see the progression of logic signals in the logic side.

5 Future Work

I would like to expand on this project in a few ways.

MIDI files are pure binary, which means that they are relatively small. It would be great to load a playlist of MIDI songs onto an SD card and allow the Arduino to read a data stream from the SD card. A switch could control the mode (incoming MIDI stream or built-in MIDI songs). Furthermore, it would be convenient to prototype a circuit that allows for fast switching between MIDI from the piano and MIDI from the computer. As of now, I have to unplug one connection's MIDI cable and replug another's. However, it would be possible to implement a radio button interface (i.e. only one active at a time) that could allow easy toggling between various MIDI input ports.

While hard, it is theoretically possible to interlace the PWM outputs of two Arduinos, each controlled by an independent MIDI source, allowing for multiple-instrument music played on one Tesla coil. The challenge would be handling overlapping signals. More simply, multiple singing Tesla coils could be constructed, one for each of a band's common instruments.

6 Acknowledgements

I would like to acknowledge a few people for their contributions to this project. First, I must thank Dr. Dann for his ongoing help and willingness to spend time with me, both in the theoretical and practical realms, analyzing and debugging circuits and ICs. Mr. Ward always

demonstrated an interest in the project and offered great advice. Early in the process, Niles and I spent a lot of time discussing possible approaches to musically modulating the primary signal. Kayla and the rest of the staff at oneTesla helped me conceptually understand the primary circuit. My parents have been extremely supportive throughout these five months. Lastly, I'd like to thank TI for shipping so many free 'sample' ICs.

References

[1] What We Do. *Arc Attack.* http://www.arcattack.com/#wwd

[2] oneTesla. *oneTesla.* http://onetesla.com/

[3] Geek Group. *The Geek Group.* http://thegeekgroup.org/

[4] Critical Thinking Video Series: Thomas Edison Electrocutes Topsy the Elephant, Jan. 4, 1903. *ProCon.* http://deathpenalty.procon.org/view.resource.php?resourceID=003749

[5] Topsy (elephant). *Wikipedia.* http://en.wikipedia.org/wiki/Topsy_(elephant)

[6] The History of Nikola Tesla - a Short Story. *Vimeo.* http://vimeo.com/13327616

[7] Nikola Tesla. *History.* http://www.history.com/topics/inventions/nikola-tesla

[8] Nikola Tesla: Biography. *Biography.* http://www.biography.com/people/nikola-tesla-9504443

[9] Induction Coil. *Wikipedia.* http://en.wikipedia.org/wiki/Induction_coil

[10] Tesla Coil. *Biblioteca Pleyades.* http://www.bibliotecapleyades.net/tesla/esp_tesla_17.htm

[11] Singing Tesla coil. *Wikipedia.* http://en.wikipedia.org/wiki/Zeusaphone

[12] Tesla Coil: Popularity. *Wikipedia.* http://en.wikipedia.org/wiki/Tesla_coil#Popularity

[13] Musical Tesla Coils. *Steve's High Voltage.* http://www.stevehv.4hv.org/MusicalSSTCs.htm

[14] About oneTesla. *oneTesla.* http://onetesla.com/about

[15] News. *4HV.* http://4hv.org/news.php

[16] Ampère's Law Diagram. *HyperPhysics.* http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/imgmag/amplaw2.gif

[17] AVR Libc Home Page. *Savannah.* http://www.nongnu.org/avr-libc/

[18] Serial. *Arduino.* http://arduino.cc/en/reference/serial

[19] Arduino Mega 2560. *Arduino.* http://arduino.cc/en/Main/ArduinoBoardMega2560

[20] Arduino Due. *Arduino.* http://arduino.cc/en/Main/ArduinoBoardDue

[21] SoftwareSerial Library *Arduino.* http://www.arduino.cc/en/Reference/SoftwareSerial

[22] Port Registers. *Arduino.* http://arduino.cc/en/Reference/PortManipulation

[23] MOC3020 thru MOC3023 Optocouplers/Optoisolators. *Texas Instruments.* http://www.ti.com/lit/ds/symlink/moc3020.pdf

[24] MIDI Programming - A Complete Study: Part 1 - Midi File Basics. *Pete's QBASIC Site*http://www.petesqbsite.com/sections/express/issue18/midifilespart1.html

[25] How Phototransistors Operate. *Practical Coilgun Design.* http://www.coilgun.info/theory/phototransistors.htm

[26] A Different Kind of Internet and TV. *GoogleFiber.* https://fiber.google.com/about/

[27] What are the Basic Differences Between CMOS and TTL Signals? *National Instruments.* http://digital.ni.com/public.nsf/allkb/2D038D3AE1C35011862565A8005C5C63

[28] Transistor–transistor logic. *Wikipedia.* http://en.wikipedia.org/wiki/Transistor%E2%80%93transistor_logic

[29] CMOS. *Wikipedia.* http://en.wikipedia.org/wiki/CMOS

[30] MIDI Connector - Female Right Angle. *SparkFun.* https://www.sparkfun.com/products/9536

[31] Midi Jack Datasheet. *SparkFun.* https://www.sparkfun.com/datasheets/Prototyping/Connectors/MIDI-RA.pdf

[32] MIDI Electrical Specification Diagram & Proper Design of Joystick/MIDI Adapter. *MIDI Manufacturers' Association.* http://www.midi.org/techspecs/electrispec.php

[33] Single-Channel: 6N138, 6N139 - Low Input Current High Gain Split Darlington Optocouplers. *Fairchild Semiconductors.* http://www.fairchildsemi.com/ds/6N/6N138.pdf

[34] 1N4148 Small Signal Fast Switching Diodes Datasheet. *Vishay Semiconductors.* http://www.vishay.com/docs/81857/1n4148.pdf

[35] Plastic Fiber Optic Red LED *Industrial Fiber Optics.* http://i-fiberoptics.com/pdf/if-e96edatasheet.pdf

[36] Plastic Fiber Optic Photologic Detectors *Industrial Fiber Optics.* http://i-fiberoptics.com/pdf/ifd95.pdf

[37] General MIDI Level 1 Sound Set *MIDI Manufacturer's Association.* http://www.midi.org/techspecs/gm1sound.php

A Parts List

Table 5 lists all the parts necessary to build a singing Tesla Coil *that we don't already have.* Many essential components, including the coil itself, are already in the Whitaker Lab, and thus are left off of this list.

Table 5: Equipment List

Item	Supplier	Cost [+ Shipping] ($)	Ordered?	Received?
USB to MIDI in/out cable	HDE	5	Yes	Yes
20' MIDI to MIDI cable	Hosa	6 [+ 2]	Yes	Yes
Breadboard-friendly MIDI jacks	SparkFun	2	Yes	Yes
Q61 MIDI-enabled Keyboard	Alesis	113 [+ 25]	Yes	Yes
6N138 Optocoupler	Digikey	1	Yes	Yes

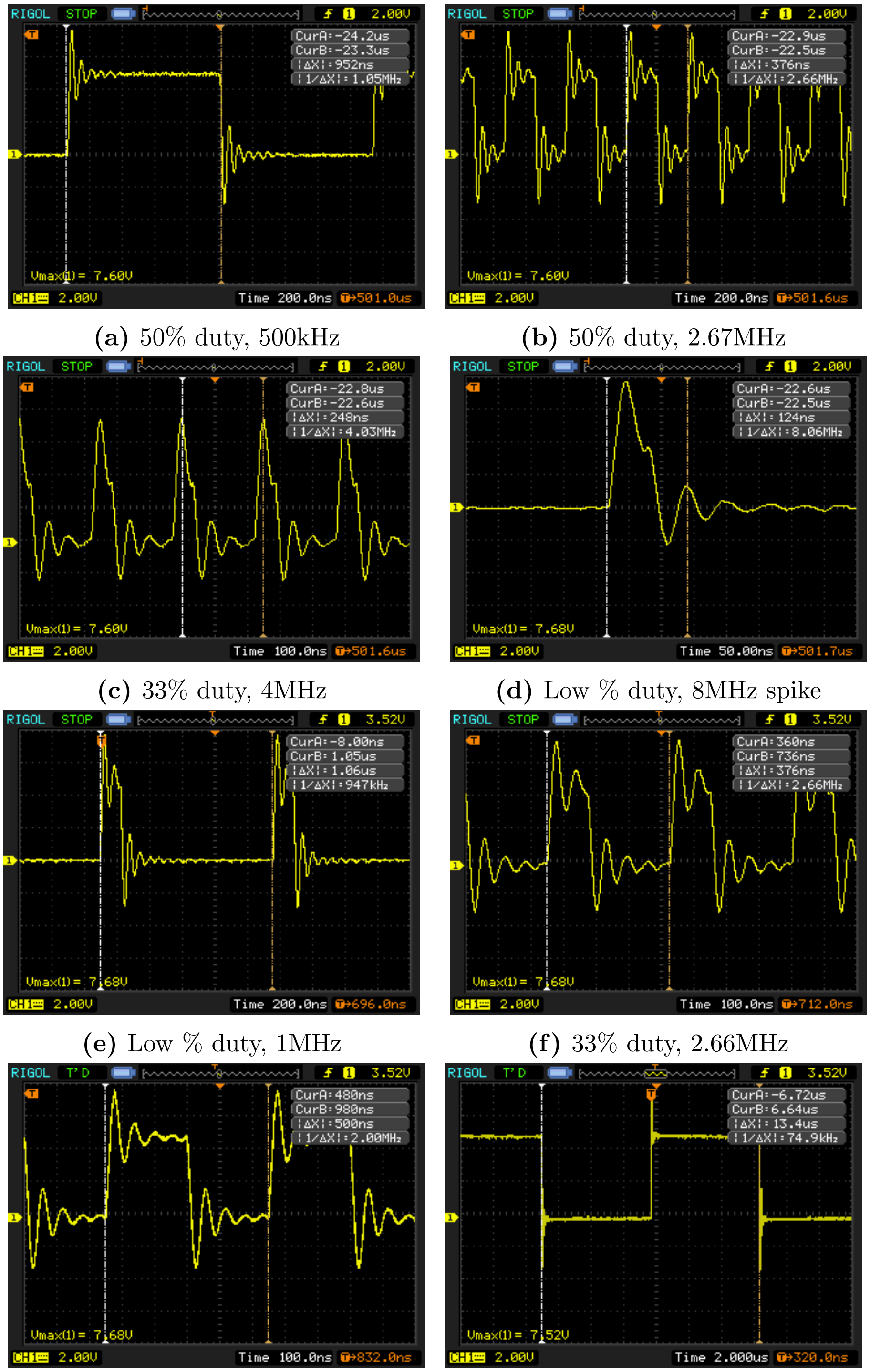
B Evidence of Port Manipulation

Figure 28 demonstrates the use of direct port access to control the voltage output of an Arduino's pin. The voltage measured is the potential difference between the pin and the Arduino's ground.

Things of note: the extremely odd overhead of calling `loop()` instead of looping in `setup()`; PINE's (toggle) performance compared to PORTE's (separate commands for on/off) performance; and most importantly, the incredible delay introduced by using `digitalWrite()` rather than direct port access. The code associated with each output is shown below.

Listing 1: Port Manipulation

```
// Digital pin 5 is PE3 on the ATMega2560, so it's controlled by the 8's place of DDRE,
    PORTE, and PINE
/* //A
void setup() {
 DDRE |= B00001000; //Configure pin 5 for output
}
void loop() {
 PINE = B1000;
} */

/* //B
void setup() {
  DDRE |= B00001000; //Configure pin 5 for output
  while (1) {
    PINE = B1000;
  }
}
void loop() {} */

/* //C
void setup() {
  DDRE |= B00001000; //Configure pin 5 for output
  while (1) {
    PINE = B1000;
    PINE = B1000;
  }
}
void loop() {} */

/* //D
void setup() {
  DDRE |= B00001000; //Configure pin 5 for output
}
void loop() {
  PINE = B1000;
  PINE = B1000;
} */

/* //E
void setup() {
  DDRE |= B00001000; //Configure pin 5 for output
}
void loop() {
  PORTE |= B00001000; //Turn on
  PORTE &= B11110111; //Turn off
} */

/* //F
void setup() {
  DDRE |= B00001000; //Configure pin 5 for output
  while (1) {
    PORTE |= B00001000; //Turn on
    PORTE &= B11110111; //Turn off
  }
}
void loop() {} */

/* //G
void setup() {
  DDRE |= B00001000; //Configure pin 5 for output
  while (1) {
    PORTE |= B00001000; //Turn on
    PORTE |= B00001000; //Do nothing for 2 cycles
    PORTE &= B11110111; //Turn off
  }
}
void loop() {} */

/* //H
void setup() {
  DDRE |= B00001000; //Configure pin 5 for output
  while (1) {
    digitalWrite(5,HIGH);
    digitalWrite(5,LOW);
  }
}
void loop() {} */
```



(a) 50% duty, 500kHz

(b) 50% duty, 2.67MHz

(c) 33% duty, 4MHz

(d) Low % duty, 8MHz spike

(e) Low % duty, 1MHz

(f) 33% duty, 2.66MHz

(g) 50% duty, 2MHz

(h) 50% duty, 75kHz (slow!)

Figure 28: Direct port manipulation corresponding to different code fragments

C Arduino Due Pinout

Table 6 lists the internal and external names of all of the pins on the Atmel SAM3X8E ARM Cortex-M3 CPU, the microcontroller that is the basis of the Arduino Due.

Figure 29 provides a graphical overview of the Arduino Due board, with color, annotations, and warnings. The diagram was created by Rob Gray.

Table 6: A comprehensive list of the internal and external names of pins on the Atmel SAM3X8E ARM Cortex-M3 CPU, which underlies the Arduino Due.

Pin Number	SAM3X Pin Name (Internal)	Mapped Pin Name (External)
0	PA8	RX0
1	PA9	TX0
2	PB25	Digital Pin 2
3	PC28	Digital Pin 3
4	connected to both PA29 and PC26	Digital Pin 4
5	PC25	Digital Pin 5
6	PC24	Digital Pin 6
7	PC23	Digital Pin 7
8	PC22	Digital Pin 8
9	PC21	Digital Pin 9
10	connected to both PA28 and PC29	Digital Pin 10
11	PD7	Digital Pin 11
12	PD8	Digital Pin 12
13	PB27	Digital Pin 13 / Amber LED ”L”
14	PD4	TX3
15	PD5	RX3
16	PA13	TX2
17	PA12	RX2
18	PA11	TX1
19	PA10	RX1
20	PB12	SDA
21	PB13	SCL
22	PB26	Digital Pin 22
23	PA14	Digital Pin 23
24	PA15	Digital Pin 24
25	PD0	Digital Pin 25
26	PD1	Digital pin 26
27	PD2	Digital Pin 27
28	PD3	Digital Pin 28
29	PD6	Digital Pin 29
30	PD9	Digital Pin 30

Continued on next page

Table 6 - continued from previous page

Pin Number	Pin Name (Internal)	Mapped Pin Name (External)
31	PA7	Digital Pin 31
32	PD10	Digital Pin 32
33	PC1	Digital Pin 33
34	PC2	Digital Pin 34
35	PC3	Digital Pin 35
36	PC4	Digital Pin 36
37	PC5	Digital Pin 37
38	PC6	Digital Pin 38
39	PC7	Digital Pin 39
40	PC8	Digital Pin 40
41	PC9	Digital Pin 41
42	PA19	Digital Pin 42
43	PA20	Digital Pin 43
44	PC19	Digital Pin 44
45	PC18	Digital Pin 45
46	PC17	Digital Pin 46
47	PC16	Digital Pin 47
48	PC15	Digital Pin 48
49	PC14	Digital Pin 49
50	PC13	Digital Pin 50
51	PC12	Digital Pin 51
52	PB21	Digital Pin 52
53	PB14	Digital Pin 53
54	PA16	Analog In 0
55	PA24	Analog In 1
56	PA23	Analog In 2
57	PA22	Analog In 3
58	PA6	Analog In 4
59	PA4	Analog In 5
60	PA3	Analog In 6
61	PA2	Analog In 7
62	PB17	Analog In 8
63	PB18	Analog In 9
64	PB19	Analog In 10
65	PB20	Analog In 11
66	PB15	DAC0
67	PB16	DAC1
68	PA1	CANRX
69	PA0	CANTX

Continued on next page

Table 6 - continued from previous page

Pin Number	Pin Name (Internal)	Mapped Pin Name (External)
70	PA17	SDA1
71	PA18	SCL2
72	PC30	LED ”RX”
73	PA21	LED ”TX”
74	PA25	(MISO)
75	PA26	(MOSI)
76	PA27	(SCLK)
77	PA28	(NPCS0)
78	PB23	(unconnected)
USB	PB11	ID
USB	PB10	VBOF

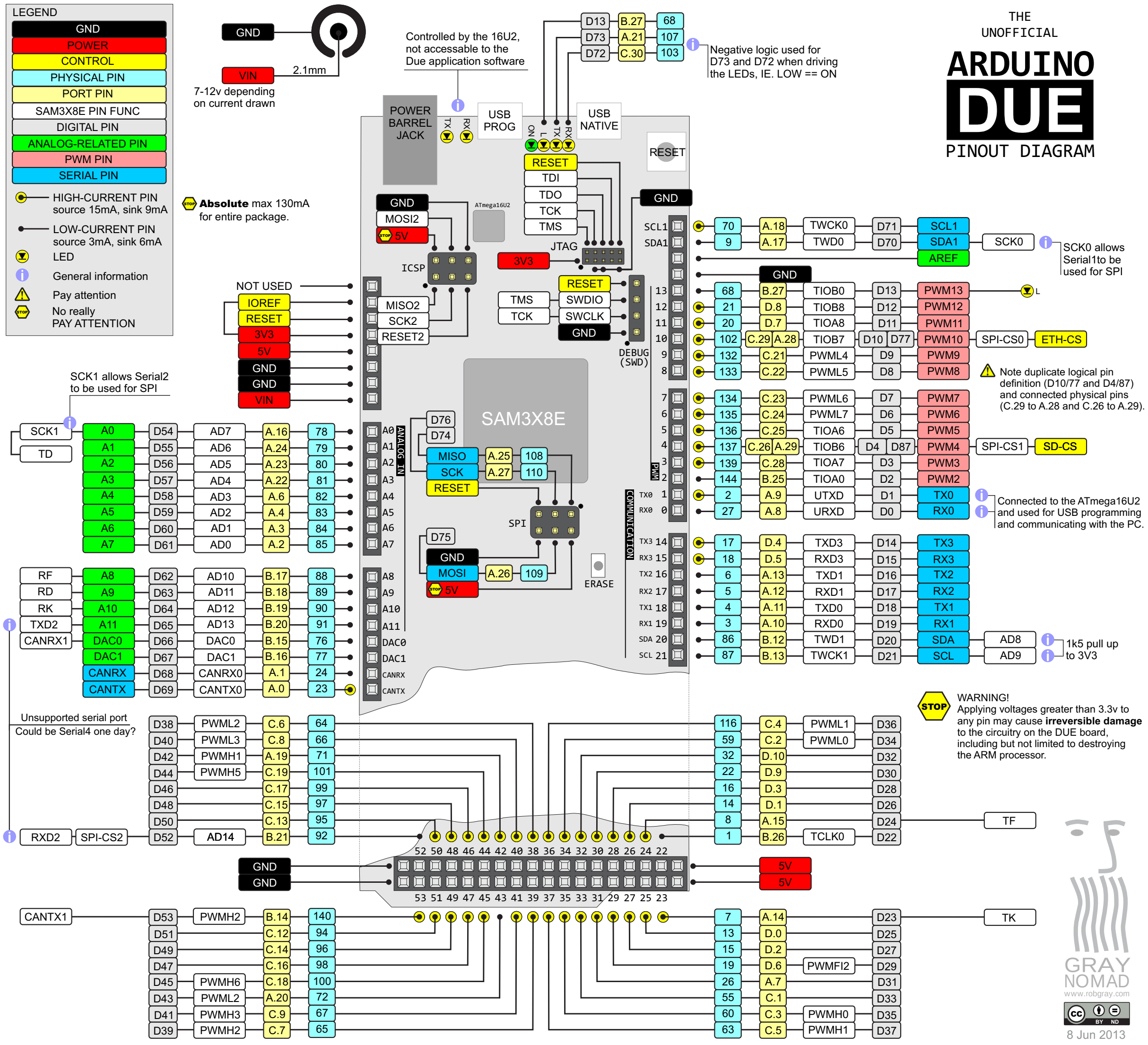


Figure 29: An unofficial, but extremely helpful, pinout diagram of the Arduino Due, complete with captions, port assignments, and warnings, created by Rob Gray. (Zoom in for more resolution)

D More MIDI Specifics

Table 7 lists families of instruments accepted by MIDI (specifically, the General MIDI Level 1 Standard). Table 8 specifically lists all the instruments (also called programs or patches) available. Table 9 lists the minimum possible set of percussion sounds available. Table 10 shows selected controllers and their corresponding values. This list is not comprehensive. [37]

Table 7: MIDI Instrument Families (PC# = Program Change Number)

PC#	Family Name	PC#	Family Name
1-8	Piano	65-72	Reed
9-16	Chromatic Percussion	73-80	Pipe
17-24	Organ	81-88	Synth Lead
25-32	Guitar	89-96	Synth Pad
33-40	Bass	97-104	Synth Effects
41-48	Strings	105-112	Ethnic
49-56	Ensemble	113-120	Percussive
57-64	Brass	121-128	Sound Effects

Table 8: All instruments specified by the General MIDI Level 1 Standard. Note that sound types in parentheses are suggested, but not reinforced.

PC#	Instrument Name	PC#	Instrument Name
1	Acoustic Grand Piano	65	Soprano Sax
2	Bright Acoustic Piano	66	Alto Sax
3	Electric Grand Piano	67	Tenor Sax
4	Honky-tonk Piano	68	Baritone Sax
5	Electric Piano 1	69	Oboe
6	Electric Piano 2	70	English Horn
7	Harpsichord	71	Bassoon
8	Clavi	72	Clarinet
9	Celesta	73	Piccolo
10	Glockenspiel	74	Flute
11	Music Box	75	Recorder
12	Vibraphone	76	Pan Flute
13	Marimba	77	Blown Bottle
14	Xylophone	78	Shakuhachi
15	Tubular Bells	79	Whistle
16	Dulcimer	80	Ocarina
17	Drawbar Organ	81	Lead 1 (square)
18	Percussive Organ	82	Lead 2 (sawtooth)
19	Rock Organ	83	Lead 3 (calliope)
20	Church Organ	84	Lead 4 (chiff)
21	Reed Organ	85	Lead 5 (charang)
22	Accordion	86	Lead 6 (voice)
23	Harmonica	87	Lead 7 (fifths)
24	Tango Accordion	88	Lead 8 (bass + lead)

Continued on next page

Table 8 - continued from previous page

PC#	Instrument Name	PC#	Instrument Name
25	Acoustic Guitar (nylon)	89	Pad 1 (new age)
26	Acoustic Guitar (steel)	90	Pad 2 (warm)
27	Electric Guitar (jazz)	91	Pad 3 (polysynth)
28	Electric Guitar (clean)	92	Pad 4 (choir)
29	Electric Guitar (muted)	93	Pad 5 (bowed)
30	Overdriven Guitar	94	Pad 6 (metallic)
31	Distortion Guitar	95	Pad 7 (halo)
32	Guitar harmonics	96	Pad 8 (sweep)
33	Acoustic Bass	97	FX 1 (rain)
34	Electric Bass (finger)	98	FX 2 (soundtrack)
35	Electric Bass (pick)	99	FX 3 (crystal)
36	Fretless Bass	100	FX 4 (atmosphere)
37	Slap Bass 1	101	FX 5 (brightness)
38	Slap Bass 2	102	FX 6 (goblins)
39	Synth Bass 1	103	FX 7 (echoes)
40	Synth Bass 2	104	FX 8 (sci-fi)
41	Violin	105	Sitar
42	Viola	106	Banjo
43	Cello	107	Shamisen
44	Contrabass	108	Koto
45	Tremolo Strings	109	Kalimba
46	Pizzicato Strings	110	Bag pipe
47	Orchestral Harp	111	Fiddle
48	Timpani	112	Shanai
49	String Ensemble 1	113	Tinkle Bell
50	String Ensemble 2	114	Agogo
51	SynthStrings 1	115	Steel Drums
52	SynthStrings 2	116	Woodblock
53	Choir Aahs	117	Taiko Drum
54	Voice Oohs	118	Melodic Tom
55	Synth Voice	119	Synth Drum
56	Orchestra Hit	120	Reverse Cymbal
57	Trumpet	121	Guitar Fret Noise
58	Trombone	122	Breath Noise
59	Tuba	123	Seashore
60	Muted Trumpet	124	Bird Tweet
61	French Horn	125	Telephone Ring
62	Brass Section	126	Helicopter
63	SynthBrass 1	127	Applause

Continued on next page

Table 8 - continued from previous page

PC#	Instrument Name	PC#	Instrument Name
64	SynthBrass 2	128	Gunshot

Table 9: Percussion sounds specified by the General MIDI Level 1 Standard. Many MIDI instruments support more than this set of sounds; however, this is the base set of sounds that *all* MIDI instruments must support.

Key#	Drum Sound	Key#	Drum Sound
35	Acoustic Bass Drum	59	Ride Cymbal 2
36	Bass Drum 1	60	Hi Bongo
37	Side Stick	61	Low Bongo
38	Acoustic Snare	62	Mute Hi Conga
39	Hand Clap	63	Open Hi Conga
40	Electric Snare	64	Low Conga
41	Low Floor Tom	65	High Timbale
42	Closed Hi Hat	66	Low Timbale
43	High Floor Tom	67	High Agogo
44	Pedal Hi-Hat	68	Low Agogo
45	Low Tom	69	Cabasa
46	Open Hi-Hat	70	Maracas
47	Low-Mid Tom	71	Short Whistle
48	Hi-Mid Tom	72	Long Whistle
49	Crash Cymbal 1	73	Short Guiro
50	High Tom	74	Long Guiro
51	Ride Cymbal 1	75	Claves
52	Chinese Cymbal	76	Hi Wood Block
53	Ride Bell	77	Low Wood Block
54	Tambourine	78	Mute Cuica
55	Splash Cymbal	79	Open Cuica
56	Cowbell	80	Mute Triangle
57	Crash Cymbal 2	81	Open Triangle
58	Vibraslap		

Table 10: A list of the most common MIDI controllers

Controller Number#	Controller Name
1	Modulation wheel
7	Volume
10	Pan
11	Expression
64	Sustain pedal
100	Registered Parameter Number LSB
101	Registered Parameter Number MSB
121	All controllers off
123	All notes off

E Code

Software was developed for the Arduino Due to convert incoming MIDI data to an outgoing PWM signal at the frequency of the note encoded by the MIDI signal. This software supports up to nine concurrent notes, each running on one of the Due's Asynchronous Timer/Counters.

Additional software was developed for the ATmega2560, the microcontroller which underlies the Arduino Mega 2560 R3, and is available on demand. The Arduino Mega software only supports up to 5 concurrent notes, and the code is much more convoluted.

All the software is open source and licensed under the MIT License. It can be accessed at https://github.com/sredmond/DueTeslaInterrupter

The DueTeslaInterrupter file contains the main body of the Arduino program. It lets other modules do a lot of the heavy lifting.

Listing 2: DueTeslaInterrupter.ino

```
/*
 *  @file               DueTeslaInterrupter.ino
 *  Project             Tesla Coil Interrupter
 *      @brief          Interrupter for Arduino Due
 *      @version        1.0
 *  @author             Sam Redmond
 *      @date           05/11/14
 *  license             MIT License
 */

#include "system.h"
#include "DueTimer.h"
#include "MIDI.h"
#include "LCD.h"
#include "handlers.h"

void setup()
{
  setupLED(); //Configure pin 3 for output
  ledOff(); //Just in case the Due holds the I/O pins high by default

  //Setup is inexpensive
  Serial.begin(9600);
  setupLCD(); //Setup the Liquid Crystal Display

  //Print warning messages
  if (!useSerial) {
    Serial.println("Serial␣output␣is␣disabled");
  }

  //Configure MIDI to listen on all channels
  MIDI.begin(MIDI_CHANNEL_OMNI);

  // Setup MIDI callback functions
  MIDI.setHandleNoteOn(HandleNoteOn); //Only need the name of the function
  MIDI.setHandleNoteOff(HandleNoteOff);
  MIDI.setHandleStop(HandleStop);
  MIDI.setHandleContinue(HandleContinue);
  MIDI.setHandleSystemReset(HandleSystemReset);

  while (1) //Loop
  {
    MIDI.read(); //Read any incoming MIDI signals
  }
}

//We avoid using the loop method, and instead loop inside setup
void loop(){}
```

The system module was created for system-wide macros and short functions to expedite the port-writing process.

Listing 3: system.h

```
#ifndef __SYSTEM_H
#include <Arduino.h> //<Arduino.h> is the default

//Determines whether to log MIDI events to Serial or LCD
const bool useSerial = false;
const bool useLCD = false;

//Array mapping MIDI pitch values to note frequencies (in Hz)
const double frequencyTable[] = {8.1757989156, 8.6619572180, 9.1770239974, 10.3008611535,
    10.3008611535, 10.9133822323, 11.5623257097, 12.2498573744, 12.9782717994,
    13.7500000000, 14.5676175474, 15.4338531643, 16.3515978313, 17.3239144361,
    18.3540479948, 19.4454364826, 20.6017223071, 21.8267644646, 23.1246514195,
    24.4997147489, 25.9565435987, 27.5000000000, 29.1352350949, 30.8677063285,
    32.7031956626, 34.6478288721, 36.7080959897, 38.8908729653, 41.2034446141,
    43.6535289291, 46.2493028390, 48.9994294977, 51.9130871975, 55.0000000000,
    58.2704701898, 61.7354126570, 65.4063913251, 69.2956577442, 73.4161919794,
    77.7817459305, 82.4068892282, 87.3070578583, 92.4986056779, 97.9988589954,
    103.8261743950, 110.0000000000, 116.5409403795, 123.4708253140, 130.8127826503,
    138.5913154884, 146.8323839587, 155.5634918610, 164.8137784564, 174.6141157165,
    184.9972113558, 195.9977179909, 207.6523487900, 220.0000000000, 233.0818807590,
    246.9416506281, 261.625565300, 277.182630976, 293.664767917, 311.126983722,
    329.627556912, 349.228231433, 369.994422711, 391.995435981, 415.304697579, 440.000000,
    466.16376151, 493.88330125, 523.25113060, 554.36526195, 587.32953583, 622.25396744,
    659.25511382, 698.45646286, 739.98884542, 783.9908719635, 830.6093951599,
    880.0000000000, 932.3275230362, 987.7666025122, 1046.5022612024, 1108.7305239075,
    1174.6590716696, 1244.5079348883, 1318.5102276515, 1396.9129257320, 1479.9776908465,
    1567.9817439270, 1661.2187903198, 1760.0000000000, 1864.6550460724, 1975.5332050245,
    2093.0045224048, 2217.4610478150, 2349.3181433393, 2489.0158697766, 2637.0204553030,
    2793.8258514640, 2959.9553816931, 3135.9634878540, 3322.4375806396, 3520.0000000000,
    3729.3100921447, 3951.0664100490, 4186.0090448096, 4434.9220956300, 4698.6362866785,
    4978.0317395533, 5274.0409106059, 5587.6517029281, 5919.9107633862, 5919.9107633862,
    6644.8751612791, 7040.0000000000, 7458.6201842894, 7902.1328200980, 8372.0180896192,
    8869.8441912599, 9397.2725733570, 9956.0634791066, 10548.0818212118, 11175.3034058561,
    11839.8215267723, 12543.8539514160};

/*
REG_PIOC_OER = REGister - Parallel Input/Output C - Output Enable Register
REG_PIOC_SODR = REGister - Parallel Input/Output C - Set Output Data Register
REG_PIOC_CODR = REGister - Parallel Input/Output C - Clear Output Data Register
*/
//Macros for performance-intensive one-liners
#define setupLED() REG_PIOC_OER |= (1u << 28) //Sets Pin 3 as Output
#define ledOn() REG_PIOC_SODR = (1u << 28) //Turns on Pin 3
#define ledOff() REG_PIOC_CODR = (1u << 28) //Turns off Pin 3

//Function prototypes
void blinkLED(); //Blink the LED
void customDelayMicros(uint32_t us); //Custom delay function

#define __SYSTEM_H
#endif
```

Listing 4: system.cpp

```
#include "system.h"

const int delayMicros = 10; //The amount of time to delay between rise and fall
void blinkLED()
{
  ledOn();
  delayMicroseconds(delayMicros);
  ledOff();
  delayMicroseconds(delayMicros); //Make sure that we stay at most 10% duty cycle
}
```

The handlers module contains a series of functions to respond to incoming MIDI signals appropriately

Listing 5: handlers.h

```
/*
 * MIDI Event Handlers
 * These functions are automatically called by the MIDI library
 * When MIDI events are received
 */

#ifndef __HANDLERS_H
#include "system.h"

void HandleNoteOn(byte channel, byte pitch, byte velocity);
void HandleNoteOff(byte channel, byte pitch, byte velocity);
void HandleStop();
void HandleContinue();
void HandleSystemReset();

#define __HANDLERS_H
#endif
```

Listing 6: handlers.cpp

```
#include "handlers.h"
#include "system.h"
```

```
#include "DueTimer.h"
#include "LCD.h"

//Array mapping MIDI pitch values to the Timer (if any) that is controlling that note
DueTimer activeTimers[] = {NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL}; //Each slot corresponds to a particular pitch

/*
 * Handle note-on messages
 * Precondition: 1 <= channel <= 16, 0 <= pitch < 128, 0 <= velocity < 128
 */
void HandleNoteOn(byte channel, byte pitch, byte velocity)
{
  //If zero velocity, turn the note off
  if (velocity == 0) {
    HandleNoteOff(channel, pitch, velocity);
    return;
  }

  double freq = frequencyTable[pitch];
  if (useSerial) {
    //Print a message to Serial Monitor
    Serial.println("NOTE␣ON");
    Serial.println("Channel:␣" + String(channel) + ",␣Pitch:␣" + String(pitch) + ",␣Velocity:␣" + String(velocity));
    Serial.println("Computed␣Frequency:␣"+String(freq)+"Hz");
  }
  if (useLCD) {
    //Print a message to the LCD
    clearLCD();
    printToLCD(0,0,"On␣F="+String(freq)+"Hz");
    printToLCD(0,1,"P="+String(pitch)+",␣V="+String(velocity));
  }
  ledOff(); //just to be safe
  activeTimers[pitch] = Timer.getAvailable().attachInterrupt(blinkLED).setFrequency(freq).start();
//  timer.setFrequency(freq);
//  timer.start();
  blinkLED(); //Blink at the start of the first period
  ;
}

/*
 * Handle note-off messages
 * Precondition: 1 <= channel <= 16, 0 <= pitch < 128, 0 <= velocity < 128
 */
void HandleNoteOff(byte channel, byte pitch, byte velocity)
{
  if (useSerial) {
    //Print a message to Serial Monitor
    Serial.println("NOTE␣OFF");
    Serial.println("Channel:␣" + String(channel) + ",␣Pitch:␣" + String(pitch) + ",␣Velocity:␣" + String(velocity));
  }
  if (useLCD) {
    //Print a message to the LCD
    clearLCD();
    printToLCD(0,0,"Off");
    printToLCD(0,1,"P="+String(pitch)+",␣V="+String(velocity));
  }
  DueTimer timer = activeTimers[pitch]; //Error prone, if a NoteOff is sent for some note that wasn't turned on with NoteOn
  timer.detachInterrupt(); //Also stops the timer
  activeTimers[pitch] = NULL; //Clear the spot in the activeTimers array
}

void HandleStop() {
  if (useSerial) {
    Serial.println("STOP");
  }
  if (useLCD) {
    //Print to LCD
    clearLCD();
    printToLCD(0,0,"Stop!");
    printToLCD(0,1,"Hammertime");
  }
  Timer0.stop();
  Timer1.stop();
  Timer2.stop();
  Timer3.stop();
  Timer4.stop();
  Timer5.stop();
  Timer6.stop();
  Timer7.stop();
  Timer8.stop();
}

//Potentially leaves free-running timers
void HandleContinue() {
  if (useSerial) {
    Serial.println("CONTINUE");
  }
  if (useLCD) {
    //Print to LCD
    clearLCD();
    printToLCD(0,0,"Restarting...");
    printToLCD(0,1,"(Could␣crash)");
  }
  Timer0.start();
  Timer1.start();
  Timer2.start();
  Timer3.start();
  Timer4.start();
  Timer5.start();
  Timer6.start();
  Timer7.start();
  Timer8.start();
}

//Reset EVERYTHING
void HandleSystemReset() {
  if (useSerial) {
    Serial.println("SYSTEM␣RESET");
  }
  if (useLCD) {
    //Print to LCD
    clearLCD();
    printToLCD(0,0,"Resetting");
    printToLCD(0,1,"System");
  }
  Timer0.detachInterrupt();
  Timer1.detachInterrupt();
  Timer2.detachInterrupt();
  Timer3.detachInterrupt();
  Timer4.detachInterrupt();
  Timer5.detachInterrupt();
  Timer6.detachInterrupt();
  Timer7.detachInterrupt();
  Timer8.detachInterrupt();
  activeTimers = {NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL
      ,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
      NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
      NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
      NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
      NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
      NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
      NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL};
}
```

The LCD module provides a simplified interface for interacting with the LCD output.

Listing 7: LCD.h

```
#ifndef __LCD_H

void setupLCD();
void printToLCD(int x, int y, String message);
void clearLCD();

#define __LCD_H
#endif
```

Listing 8: LCD.cpp

```
#include "LiquidCrystal.h"
#include "LCD.h"
//Register Select, Enable, DB4, DB5, DB6, DB7
LiquidCrystal lcd(12,11,9,8,7,6); //We hide the LCD from the main body of the program to
    keep the LCD module relatively standalone.
void setupLCD()
{
  lcd.begin(16,2);
  lcd.setCursor(0,0);
  lcd.print("Initiating");
  lcd.setCursor(0,1);
  lcd.print("Liquid␣Crystal");
}

//Prerequisite: 0 <= x < 16, 0 <= y < 2
void printToLCD(int x, int y, String msg)
{
  lcd.setCursor(x,y);
  lcd.print(msg);
}
void clearLCD()
{
  lcd.clear();
}
```

The DueTimer library provides an incredibly simplified interface for interacting with the Arduino Due's Timer/Counters. All credit goes to Ivan Seidel. His code is hosted on GitHub at https://github.com/ivanseidel/DueTimer

Listing 9: DueTimer.h

```
/*
  DueTimer.h - DueTimer header file, definition of methods and attributes...
  For instructions, go to https://github.com/ivanseidel/DueTimer

  Created by Ivan Seidel Gomes, March, 2013.
  Modified by Philipp Klaus, June 2013.
  Released into the public domain.
*/
#ifdef __arm__

#ifndef DueTimer_h
#define DueTimer_h

#include <Arduino.h>

#include <inttypes.h>

/*
        This fixes compatibility for Arduono Servo Library.
        Uncomment to make it compatible.

        Note that:
                + Timers: 0,2,3,4,5 WILL NOT WORK, and will
                                  neither be accessible by Timer0,...
*/
// #define USING_SERVO_LIB      true

#ifdef USING_SERVO_LIB
        #warning "HEY!␣You␣have␣set␣flag␣USING_SERVO_LIB.␣Timer0,␣2,3,4␣and␣5␣are␣not␣
            available"
#endif

class DueTimer
{
protected:

        // Represents the timer id (index for the array of Timer structs)
        int timer;

        // Stores the object timer frequency
        // (allows to access current timer period and frequency):
        static double _frequency[9];

        // Picks the best clock to lower the error
        static uint8_t bestClock(double frequency, uint32_t& retRC);

public:
        struct Timer
        {
                Tc *tc;
                uint32_t channel;
                IRQn_Type irq;
        };

        static DueTimer getAvailable();

        // Store timer configuration (static, as it's fix for every object)
        static const Timer Timers[9];

        // Needs to be public, because the handlers are outside class:
        static void (*callbacks[9])();

        DueTimer(int _timer);
        DueTimer attachInterrupt(void (*isr)());
        DueTimer detachInterrupt();
        DueTimer start(long microseconds = -1);
        DueTimer stop();
        DueTimer setFrequency(double frequency);
        DueTimer setPeriod(long microseconds);

        double getFrequency();
        long getPeriod();
};

// Just to call Timer.getAvailable instead of Timer::getAvailable() :
extern DueTimer Timer;

extern DueTimer Timer1;
// Fix for compatibility with Servo library
#ifndef USING_SERVO_LIB
        extern DueTimer Timer0;
        extern DueTimer Timer2;
        extern DueTimer Timer3;
        extern DueTimer Timer4;
        extern DueTimer Timer5;
#endif
extern DueTimer Timer6;
extern DueTimer Timer7;
extern DueTimer Timer8;

#endif
#else
        #error Oops! Trying to include DueTimer on another device?
#endif
```

Listing 10: DueTimer.cpp

```
/*
  DueTimer.cpp - Implementation of Timers defined on DueTimer.h
  For instructions, go to https://github.com/ivanseidel/DueTimer

  Created by Ivan Seidel Gomes, March, 2013.
  Modified by Philipp Klaus, June 2013.
  Thanks to stimmer (from Arduino forum), for coding the "timer soul" (Register stuff)
  Released into the public domain.
*/

#include "DueTimer.h"

const DueTimer::Timer DueTimer::Timers[9] = {
        {TC0,0,TC0_IRQn},
        {TC0,1,TC1_IRQn},
        {TC0,2,TC2_IRQn},
        {TC1,0,TC3_IRQn},
        {TC1,1,TC4_IRQn},
        {TC1,2,TC5_IRQn},
        {TC2,0,TC6_IRQn},
        {TC2,1,TC7_IRQn},
        {TC2,2,TC8_IRQn},
};

// Fix for compatibility with Servo library
#ifdef USING_SERVO_LIB
        // Set callbacks as used, allowing DueTimer::getAvailable() to work
        void (*DueTimer::callbacks[9])() = {
                (void (*)()) 1, // Timer 0 - Occupied
                (void (*)()) 0, // Timer 1
                (void (*)()) 1, // Timer 2 - Occupied
                (void (*)()) 1, // Timer 3 - Occupied
                (void (*)()) 1, // Timer 4 - Occupied
                (void (*)()) 1, // Timer 5 - Occupied
                (void (*)()) 0, // Timer 6
                (void (*)()) 0, // Timer 7
                (void (*)()) 0  // Timer 8
        };
#else
        void (*DueTimer::callbacks[9])() = {};
#endif
double DueTimer::_frequency[9] = {-1,-1,-1,-1,-1,-1,-1,-1,-1};

/*
        Initializing all timers, so you can use them like this: Timer0.start();
*/
DueTimer Timer(0);

DueTimer Timer1(1);
// Fix for compatibility with Servo library
#ifndef USING_SERVO_LIB
        DueTimer Timer0(0);
        DueTimer Timer2(2);
        DueTimer Timer3(3);
        DueTimer Timer4(4);
        DueTimer Timer5(5);
#endif
DueTimer Timer6(6);
DueTimer Timer7(7);
DueTimer Timer8(8);

DueTimer::DueTimer(int _timer){
        /*
                The constructor of the class DueTimer
        */

        timer = _timer;
}

DueTimer DueTimer::getAvailable(){
        /*
                Return the first timer with no callback set
        */

        for(int i = 0; i < 9; i++){
                if(!callbacks[i])
                        return DueTimer(i);
        }
        // Default, return Timer0;
        return DueTimer(0);
}

DueTimer DueTimer::attachInterrupt(void (*isr)()){
        /*
                Links the function passed as argument to the timer of the object
        */

        callbacks[timer] = isr;

        return *this;
}

DueTimer DueTimer::detachInterrupt(){
        /*
                Links the function passed as argument to the timer of the object
        */

        stop(); // Stop the currently running timer

        callbacks[timer] = NULL;

        return *this;
}

DueTimer DueTimer::start(long microseconds){
        /*
                Start the timer
                If a period is set, then sets the period and start the timer
        */

        if(microseconds > 0)
                setPeriod(microseconds);

        if(_frequency[timer] <= 0)
                setFrequency(1);

        NVIC_ClearPendingIRQ(Timers[timer].irq);
        NVIC_EnableIRQ(Timers[timer].irq);

        TC_Start(Timers[timer].tc, Timers[timer].channel);

        return *this;
}

DueTimer DueTimer::stop(){
        /*
                Stop the timer
        */

        NVIC_DisableIRQ(Timers[timer].irq);

        TC_Stop(Timers[timer].tc, Timers[timer].channel);

        return *this;
}

uint8_t DueTimer::bestClock(double frequency, uint32_t& retRC){
        /*
                Pick the best Clock, thanks to Ogle Basil Hall!

                Timer           Definition
                TIMER_CLOCK1    MCK /  2
                TIMER_CLOCK2    MCK /  8
                TIMER_CLOCK3    MCK / 32
                TIMER_CLOCK4    MCK /128
        */
        struct {
                uint8_t flag;
                uint8_t divisor;
        } clockConfig[] = {
                { TC_CMR_TCCLKS_TIMER_CLOCK1,   2 },
                { TC_CMR_TCCLKS_TIMER_CLOCK2,   8 },
                { TC_CMR_TCCLKS_TIMER_CLOCK3,  32 },
                { TC_CMR_TCCLKS_TIMER_CLOCK4, 128 }
        };
        float ticks;
        float error;
        int clkId = 3;
        int bestClock = 3;
        float bestError = 1.0;
        do
        {
                ticks = (float) VARIANT_MCK / frequency / (float) clockConfig[clkId].divisor;
                error = abs(ticks - round(ticks));
                if (abs(error) < bestError)
                {
                        bestClock = clkId;
                        bestError = error;
                }
        } while (clkId-- > 0);
        ticks = (float) VARIANT_MCK / frequency / (float) clockConfig[bestClock].divisor;
        retRC = (uint32_t) round(ticks);
        return clockConfig[bestClock].flag;
}


DueTimer DueTimer::setFrequency(double frequency){
        /*
                Set the timer frequency (in Hz)
        */

        // Prevent negative frequencies
        if(frequency <= 0) { frequency = 1; }

        // Remember the frequency
        _frequency[timer] = frequency;

        // Get current timer configuration
        Timer t = Timers[timer];

        uint32_t rc = 0;
        uint8_t clock;

        // Tell the Power Management Controller to disable
        // the write protection of the (Timer/Counter) registers:
        pmc_set_writeprotect(false);

        // Enable clock for the timer
        pmc_enable_periph_clk((uint32_t)t.irq);

        // Find the best clock for the wanted frequency
        clock = bestClock(frequency, rc);

        // Set up the Timer in waveform mode which creates a PWM
        // in UP mode with automatic trigger on RC Compare
        // and sets it up with the determined internal clock as clock input.
        TC_Configure(t.tc, t.channel, TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC | clock);
        // Reset counter and fire interrupt when RC value is matched:
        TC_SetRC(t.tc, t.channel, rc);
        // Enable the RC Compare Interrupt...
        t.tc->TC_CHANNEL[t.channel].TC_IER=TC_IER_CPCS;
        // ... and disable all others.
        t.tc->TC_CHANNEL[t.channel].TC_IDR=~TC_IER_CPCS;

        return *this;
}

DueTimer DueTimer::setPeriod(long microseconds){
        /*
                Set the period of the timer (in microseconds)
        */

        // Convert period in microseconds to frequency in Hz
        double frequency = 1000000.0 / microseconds;
        setFrequency(frequency);
        return *this;
}

double DueTimer::getFrequency(){
        /*
                Get current time frequency
        */

        return _frequency[timer];
}

long DueTimer::getPeriod(){
        /*
                Get current time period
        */

        return 1.0/getFrequency()*1000000;
}

/*
        Implementation of the timer callbacks defined in
        arduino-1.5.2/hardware/arduino/sam/system/CMSIS/Device/ATMEL/sam3xa/include/sam3x8e.h
*/
// Fix for compatibility with Servo library
#ifndef USING_SERVO_LIB
void TC0_Handler(){
        TC_GetStatus(TC0, 0);
        DueTimer::callbacks[0]();
}
#endif
void TC1_Handler(){
        TC_GetStatus(TC0, 1);
        DueTimer::callbacks[1]();
}
// Fix for compatibility with Servo library
#ifndef USING_SERVO_LIB
void TC2_Handler(){
        TC_GetStatus(TC0, 2);
        DueTimer::callbacks[2]();
}
void TC3_Handler(){
        TC_GetStatus(TC1, 0);
        DueTimer::callbacks[3]();
}
void TC4_Handler(){
        TC_GetStatus(TC1, 1);
        DueTimer::callbacks[4]();
}
void TC5_Handler(){
        TC_GetStatus(TC1, 2);
        DueTimer::callbacks[5]();
}
#endif
void TC6_Handler(){
        TC_GetStatus(TC2, 0);
        DueTimer::callbacks[6]();
}
void TC7_Handler(){
        TC_GetStatus(TC2, 1);
        DueTimer::callbacks[7]();
}
void TC8_Handler(){
        TC_GetStatus(TC2, 2);
        DueTimer::callbacks[8]();
```

```
}
```

Note: In addition to these libraries, you must install the MIDI library into the local workspace, available at https://github.com/FortySevenEffects/arduino_midi_library/ (you'll need the src/MIDI.h and src/MIDI.cpp files. All credit for this library goes to FortySevenEffects. Furthermore, you must include the LiquidCrystal library in the local workspace, rather than refer to it as a system library. This is because the Arduino IDE 1.5 has issues with the LiquidCrystal library.