Final Report for ECE 445, Senior Design, Spring 2013

TA: Mustafa Mukadam

01 May 2013

Project No. 4

Angad Bector

Joel Spadin

Ruichen Zhao

By

MIDI Controlled Slide Guitar

**Abstract**

We designed and built a single-stringed, slide guitar-like instrument which automatically responded to MIDI input. We were able to make the system respond quickly enough to be played in real time by a performer with a MIDI instrument such as a keyboard or a wind controller.

The instrument uses a single guitar D string with a motorized slide which moves along the string to control its pitch. At one end of the string, a wheel with two guitar picks rotates to excite the string or to stop its vibrations. A magnetic pickup converts the vibrations to an electric signal which is then processed by a microcontroller to add audio effects.

Because it uses only a single guitar string, it has a range of 1¾ octaves which limits its use as a performance instrument. This could be overcome by creating an array of strings with a single controller.

Contents

[1. Introduction 1](#_Toc355135279)

[1.1 Purpose 1](#_Toc355135280)

[1.3 Subprojects 1](#_Toc355135281)

[1.3.1 Power Supply 2](#_Toc355135282)

[1.3.2 MIDI System 2](#_Toc355135283)

[1.3.3 Motor System 2](#_Toc355135284)

[1.3.4 Audio System 2](#_Toc355135285)

[2. Design 2](#_Toc355135286)

[2.1 Power Supply 2](#_Toc355135287)

[2.2 Microcontroller 3](#_Toc355135288)

[2.3 MIDI Input 3](#_Toc355135289)

[2.4 MIDI Software 4](#_Toc355135290)

[2.5 Motors, Motor Controllers and Mechanical System 5](#_Toc355135291)

[2.7 Motor Control Software 6](#_Toc355135292)

[2.8 Pickup 7](#_Toc355135293)

[2.9 Audio Software 8](#_Toc355135294)

[2.10 Digital to Analog Converter 11](#_Toc355135295)

[3. Design Verification 13](#_Toc355135296)

[3.1 Power Supply 13](#_Toc355135297)

[3.2 Microcontroller 13](#_Toc355135298)

[3.3 MIDI System 13](#_Toc355135299)

[3.4 Motors and Motor Controllers 13](#_Toc355135300)

[3.5 Motor Control Software 14](#_Toc355135301)

[3.6 Audio Software 15](#_Toc355135302)

[3.7 Digital to Analog Converter Circuit 15](#_Toc355135303)

[3.8 Overall System Performance 16](#_Toc355135304)

[4. Costs 17](#_Toc355135305)

[4.1 Parts 17](#_Toc355135306)

[4.2 Labor 17](#_Toc355135307)

[5. Conclusion 19](#_Toc355135308)

[5.1 Future work 19](#_Toc355135309)

[5.2 Ethical considerations 19](#_Toc355135310)

[References 21](#_Toc355135311)

[Appendix A Requirement and Verification Table 22](#_Toc355135312)

[Appendix B Mechanical and Circuit Design 26](#_Toc355135313)

[Appendix C Software Diagrams 34](#_Toc355135314)

[Appendix D Design Data 36](#_Toc355135315)

# 1. Introduction

We designed and built a single-stringed, slide guitar-like instrument which automatically responded to MIDI input. We were able to make the system respond quickly enough to be played in real time by a performer with a MIDI instrument such as a keyboard or a wind controller.

The instrument uses a single guitar D string with a motorized slide which moves along the string to control its pitch. At one end of the string, a wheel with two guitar picks rotates to excite the string or to stop its vibrations. A magnetic pickup converts the vibrations to an electric signal which is then processed by a microcontroller to add audio effects. This audio data is converted back to an analog signal and then output on a 3.5 mm audio jack for use with standard speakers.

## 1.1 Purpose

The purpose of this project was to build a unique musical instrument which could be controlled in real time from a MIDI controller or from a prerecorded performance. While similar instruments have been created, none are commercially available, and none are designed to respond properly to unconventional MIDI controllers such as wind controllers. We wanted our project to react quickly enough and modify the output sound using configurable audio effects so as to be an expressive instrument that responds well to both keyboard controllers and wind controllers.

## 1.2 Subprojects

We split our project into three separate subprojects which were each responsible for one function in the instrument. Additionally, we had a single power supply to provide power to each component and a National Instruments Single Board RIO (sbRIO) upon which the software for each subproject ran. We considered each of these components to be shared among the subprojects.

Figure 1, System overview



### 1.2.1 Power Supply

The power supply provided 24 V power to the microcontroller and to the motors. It needed to provide stable power under load so that the microcontroller would continue to function normally.

### 1.2.2 MIDI System

The MIDI system was responsible for reading the data stream from a MIDI instrument and interpreting it as note and controller data. It then supplied these interpreted values to the motor and audio software. This system consisted of a MIDI input circuit and MIDI software.

### 1.2.3 Motor System

The motor system was responsible for translating data from the MIDI system into motor motions to excite the guitar string. This system consisted of motor control software, two motors with motor controllers and encoders, and the mechanical system which the motors moved. The first motor drove a belt which moves a bar along the string to change the frequency of the string vibration. The second rotated a wheel with two guitar picks to either start or stop the string vibration.

### 1.2.4 Audio System

The audio system was responsible for converting the string vibration into an electric signal, sampling it to create a digital signal, applying audio effects, and then converting the signal back to an analog one so that it can be output on speakers. This system consisted of the pickup, audio software, and digital to analog converter circuit.

# 2. Design

## 2.1 Power Supply

The maximum power required by each component of our project is listed in Table 1. We originally wanted to use two power supplies: one to power the sbRIO and another to power the motors. This would isolate the microcontroller from any voltage fluctuations caused by the motors. Each supply would need to convert 120 V AC power to 24 V DC. We were able to find a power supply to power the sbRIO, but the only power supply we found that met our requirements for the motors was much larger than necessary. We were unable to find specifications for this supply, but it provided 24 V with respect to ground and had an 8 A fuse, so we inferred that it could provide up to 192 W. This supply provided enough power to power the sbRIO as well, so we simplified our design by replacing the two power supplies with a single one.

Table 1, Power requirements

|  |  |  |  |
| --- | --- | --- | --- |
| Part | Maximum Current (A) | Voltage (V) | Power (W) |
| Single Board RIO 9632 | 1.8 | 17-30 | 7.75 |
| Picking Motor | 1 | 24 | 24 |
| Pitch Motor | 1 | 24 | 24 |
| Total | 3.8 | 24 | 55.75 |

## 2.2 Microcontroller

We initially considered using an Arduino as the microcontroller for our project, but determined that its clock frequency was not high enough to handle communication with our digital to analog converter. We also considered using a Texas Instruments Launchpad board, but we wanted to avoid programming in C if possible. We then discovered the National Instruments Single Board RIO devices, which had more than enough digital and analog IO pins for our application and could be programmed in LabVIEW. LabVIEW would allow us to design and debug our software very easily, and some of us had prior experience programming with it, so the sbRIO was ideal for our project. We were able to acquire a sbRIO 9632 from the senior design course, which we then used for all the software components of our project.

The sbRIO had 5 V pins to power digital logic circuits. It could provide a total of 2 A to logic circuits at   
5 V, for 10 W of power [1]. The MIDI input circuit contained only one current-drawing element: the 6N138 opto-isolator, which would draw a maximum of 100 mW [2]. When powered at 3.3 V, the PCM5100a DAC would draw a maximum of 148.5 mW [3]. The power dissipated in the LD1117V33 voltage regulator for the DAC can be approximated as where Vin = 5 V and Vout = 3.3 V. Assuming the Iin is the maximum current drawn by the PCM5100a of , the power dissipated in the regulator is 76.5 mW at maximum. We did not have data sheets for the optical encoders used for motor control, but as the rest of the digital circuits should draw 325 mW at maximum and the encoders should not require large amounts of power to operate, the sbRIO was more than capable of powering all of our 5 V circuitry.

## 2.3 MIDI Input

The first step in the data flow of our project is receiving MIDI data. MIDI data is sent as a serial communication at 31250 baud, so it requires only a single input line. The MIDI specification [4] states that the MIDI input to a device should be electrically isolated from the output of another device using the standardized circuit shown in Figure B-1. The input from the MIDI source is either at 0V or 5V and is fed into the opto-isolator. The output voltage of the opto-isolator mirrors that of the input, and a pull-up resistor is used to hold the output to 5V when no MIDI source is connected to the input. This prevents stray currents in either connected device from damaging the other.

For MIDI communication to operate properly at 31250 baud, the opto-isolator must have a maximum rise time of 2µs. The MIDI specification suggests using the 6N138 opto-isolator, but an equivalent or faster chip could be used. We opted to use the 6N138 for simplicity, as that would not require us to make any modifications to the standard circuit. We used a 270Ω pull-up resistor instead of the specified 280Ω, as the small difference in resistance would not change the operation of the circuit, and 280Ω resistors were not readily available.

We implemented this circuit using the PCB shown in Figure B-2 and Figure B-3. The circuit’s power, serial data out, and ground pins were connected to 5V, GPIO, and ground pins on the sbRIO, respectively. The sbRIO’s GPIO pins are 5 V tolerant, so we did not need to use any conversion circuitry.

## 2.4 MIDI Software

The serial communication from a MIDI source must be decoded before we have any useable information about the notes that a performer is playing. We found the official MIDI specification to be difficult to understand and to contain lots of implementation details which were irrelevant to our project, so we instead based our implementation on a condensed article written by David Van Brink titled *David’s MIDI Spec* [5].

MIDI data is sent as a series of messages. Each message consists of one or more 8-bit bytes: one status byte followed by zero or more data bytes. The MSB of 1 denotes a status byte, while 0 denotes a data byte. The remaining seven bits of a status byte denote the type of message. The meaning of the seven bits of each data bytes depends on what type of message is being sent. For the purposes of our instrument, we only needed to listen for Note Off, Note On, Controller Change, and Pitch Bend messages as shown in Table 2. The status bytes of these messages contain a channel number c, which is used to address one device when multiple devices are chained. Our instrument does not support chaining, so we simply ignore the channel number.

To illustrate what each message means, imagine a keyboard with pitch bend and modulation wheels. When a performer plays a note, a Note On message is sent giving the pitch of the note and its velocity (how hard the performer struck the key) as values between 0 and 127. When the performer releases a note, a similar Note Off message is sent giving the pitch. The velocity here is usually sent as 0. Alternatively, the instrument may send a Note On message with zero velocity, which is interpreted like a Note Off. When the performer moves the modulation wheel, a Controller Change message is sent with controller number 2 (modulation wheel) and the position of the wheel as a number between 0 and 127. When the performer moves the pitch bend wheel, a Pitch Bend message is sent with the position of the wheel as a number between 0 and 16383. 8192 indicates the center position, which corresponds to no bend in pitch. The bend value is split into two 7-bit numbers so that it can be transmitted.

MIDI messages are sent as a 31250 baud serial communication. The data line is held high when no data is being transmitted. When a device wants to send a message, it first sets the data line low for one bit, transmits 8 bits of data, and then holds the line high for one bit. There is no clock line, so the start bit (the initial low bit) is used to synchronize the receiver to the sender. The receiver simply waits for the line to go low, then reads the input eight times at proper intervals and then resumes waiting for another transmission. 31250 Hz translates to a period of 32 µs, and we chose to read during the middle of each bit period to avoid reading during the middle of a bit transition, so our algorithm for receiving MIDI data is as follows:

1. Wait for the data line to go low
2. Wait 48 µs (32 µs until the end of the start bit + 16 µs until the middle of the MSB)
3. Read the data line and push its value onto an 8-element Boolean array
4. Wait 32 µs
5. Repeat steps 3-4 seven more times
6. Combine the Boolean array into an 8-bit integer and push it onto a FIFO queue
7. Restart at step 1

Table 2, MIDI messages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Status Byte | Message Type | Message Size | Byte 1 | Byte 2 |
| 0x8c | Note Off | 2 | Pitch | Velocity |
| 0x9c | Note On | 2 | Pitch | Velocity |
| 0xBc | Controller Change | 2 | Controller Number | Value |
| 0xEc | Pitch Bend | 2 | Bend LSB | Bend MSB |

We implemented this algorithm inside the sbRIO’s FPGA, so MIDI input cannot be interrupted by other processes running on the microcontroller. The FPGA pushes each byte onto a FIFO queue and uses Direct Memory Access to share the raw serial data with the microcontroller. The microcontroller then runs software which interprets the data as MIDI messages.

The software which interprets MIDI messages stores a MIDI state object which holds the pitch and velocity of the last received note, whether a note is currently being played, a pitch bend offset value, and an array of the current values for each of the 128 continuous controllers. It also sends commands to the motor control software to tell it to pick the string. Since our instrument is monophonic, we do not need to keep track of whether every note is on or off. Each time a new Note On message is received, it simply overwrites the last received note.

The software runs in a 5 ms loop and works as follows:

1. Check the serial data queue for new data. If it is empty, stop and wait for the next loop execution.
2. Read the queue into an array.
3. Interpret each byte in the array in order and update the MIDI state as shown in Figure C-1 and Figure C-2.
4. If any byte generated a pick event, send a pick event to the motor control loop.
5. Copy the current state into a shared variable so it can be accessed by other software loops.

## 2.5 Motors, Motor Controllers and Mechanical System

In the early stages of our design phase, we were searching for linear actuators to drive our pitch bar along the length of the string. However, we quickly discovered that even the most basic linear motor costs much more than we could afford and so we had to come up with a cheaper alternative.

We then decided to use two rotational motors to drive the pitch bar and the pick wheel. The pick wheel would be mounted directly onto the shaft of one motor, while the pitch bar would be on a belt which in turn would be driven by the other motor. We got two Minerta Motors F-Series motors with encoders from the Everitt Power Lab. These were standard DC brushed motors that took in 5-30V DC at 1.2 A peak current. Each motor had an optical encoder with 2000 counts per revolution which we used for position feedback.

To drive our motors, we used Cytron MD10C motor controllers. These are bi-directional DC motor drivers that can be given a 3.3 V or 5 V PWM (pulse width modulated) signal at up to 10 kHz to control motor speed. The drivers take in an input voltage of 14-25 V DC with a 10A maximum current and connect directly to the motors. Since we wanted to use a 24 V supply, the motors would not draw much more than 1 A, and our microcontroller could easily generate PWM signals at 3.3 V, these controllers were perfect for our application.

We did not need to design any circuitry for the motor controllers. We simply connected the motors to the controllers and connected the digital IO of the controllers and encoders to the sbRIO as shown in Figure B-4.

Our mechanical system was designed by the ECE machine shop in Everitt lab. Diagrams and images of the system can be found at the end of Appendix B.

## 2.7 Motor Control Software

To control the motors, we decided to use PID control to control the position of each motor as it is very simple to implement and generally very effective for positional control. We used one PID controller per motor. Each controller is given the current position of the motor, the desired position, and a set of PID gains and outputs a motor speed command. The motor command is determined from the error between the current position and desired position. The error is scaled by the P (proportional) gain, the integral of the error is scaled by the I (integral) gain, and the derivative of the error is scaled by the D (derivative) gain. These three terms are added together to get the motor command. By properly tuning the PID gains, the control loop will move the motor to the desired position with minimal or no overshoot.

We implemented encoder feedback and PWM output in the sbRIO’s FPGA using code provided by National Instruments [6] [7]. We then scaled the encoder values so that we could work with degree positions instead of encoder count values. Since the encoders have 2000 counts per revolution, this gave us a resolution of .

For our position control to be correct, we first needed to find a reference position for each motor. When we turn the instrument on, it automatically enters a calibration routine which disables PID control and applies a small control signal to each motor. This causes the picking wheel to move until a pick hits the string and the pitch slide to move until it reaches the end of the guitar neck. Once the positions of each motor have stopped changing for 500 ms, we reset the encoder positions and re-enable PID control.

Since our picking wheel had two guitar picks, we needed to rotate the wheel 180° on every pick command. This command was sent by the MIDI software whenever a note was started when no note was previously playing, or when a note was started while a note was previously playing and the new note had a velocity higher than some tunable threshold. This allows a performer to control when the instrument picks during legato phrases. To allow the string to vibrate freely and to allow us to strike the string quickly after receiving a pick command, we gave the picks a 15° offset from the string. We also were able to stop the string from vibrating by removing this offset so that the pick touched the string. When a note was released, the pick would then be pressed against the string for a moment before returning to its usual 15° offset. This allows a performer to play staccato notes. We manually tuned the PID gains so that the motor would move as fast as possible initially, then smoothly come to a stop without overshoot so that it would not strike the string multiple times. These gains caused the pick to strike the string when trying to dampen it however, so we made another set of gains for this purpose. LabVIEW’s PID implementation allows for changing PID gains, so we simply select a set of gains depending on whether we are striking or stopping the string.

To implement pitch control, we first needed a conversion from motor position in degrees to displacement of the pitch. By manually moving the pitch slide increments of 10 cm and recording the motor positions, then averaging the results, we determined that a 44.8° rotation of the motor translated to a 1 cm movement. We then needed an accurate conversion from pitch values to slide positions. We used a standard electric guitar D string, and the pitch slide could move past the end of the neck, so the highest pitch our instrument could play was E♭ on the first fret. This corresponds to MIDI note number 51. Using the first fret as position 0, we then manually measured the distance to each successive fret to find approximate positions for each pitch. We later used a tuner to adjust each position to ensure that each pitch was in tune. Our final slide positions are given in Table D-1. Since pitch bends can set the pitch to a fractional note number, we use linear interpolation between each setpoint. Though the conversion between pitch and position is exponential in nature, for these small changes it is approximately linear, and pitches are not usually held to fractional note numbers for very long during a performance, so this did not cause our instrument to sound out of tune. Motor control then converts the current motor rotation and desired pitch into slide positions and feeds them into the PID controller.

We also implemented a portamento effect to pitch control. When portamento was enabled, rather than immediately changing the pitch setpoint when a new note was received, the setpoint would smoothly move to the new pitch at a configurable rate given in note numbers per second. After manually tuning the PID gains such that the pitch slide would quickly reach the correct position without portamento enabled, we discovered that the slide would repeatedly overshoot and then correct itself during a smooth portamento transition. To counter this, we again used a separate set of PID gains when the setpoint changed smoothly rather than in discrete steps. This allowed us to have both quick and smooth pitch transitions.

While systematic methods for PID tuning exist, the methods we tried did not give us acceptable motor control. Instead, we tuned our gains manually while the control software was running by increasing P until the motor responded quickly to a difference in position, and then increased I and D to smooth out the transition and stabilize the control loop. Finally, we applied small changes to each of the gains to see whether or not they improved the control performance. Our final PID gains for each of the four cases can be found in Table D-2.

## 2.8 Pickup

The pickup uses a magnetized core and a coil with many winds to convert the motion of a guitar string through the magnetic flux of the core into an electrical signal. We originally planned to fabricate a humbucking pickup, which uses two coils with opposite polarities such that environmental noise cancels out while the string vibration adds in each coil. However, we discovered that commercially available magnetic coils do not come magnetized and we did not have access to tools to magnetize them, so we used a commercial guitar pickup instead. Commercial humbucking pickups were large and expensive, so we used a single coil pickup.

We also originally planned to use a pre-amplifier to boost the low-level signal from the pickup, but discovered that the sbRIO’s analog inputs were sensitive enough to read the signal from our pickup without amplification. The only circuitry we had to add was a high impedance resistor between each of the positive and negative outputs of the pickup and the analog ground of the sbRIO so that the sbRIO’s differential analog input would function correctly. This circuit is shown in Figure B-5.

## 2.9 Audio Software

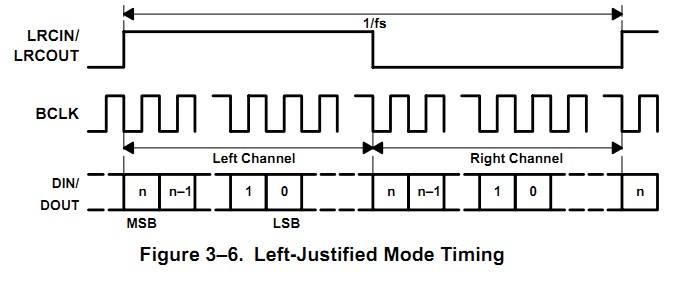
In order to make our instrument sound like an electric guitar, we wanted to process the sound of the string vibration by adding effects such as distortion and reverberation. We determined that trying to implement these effects using analog circuits would be inflexible and needlessly complex. If we wanted to change the effects after initially designing them, we would have to redesign the analog circuits or build in methods to tune them. Instead, we decided to implement all of our effects digitally using the sbRIO. This would give us the flexibility to add, remove, and modify the effects at any time.

We chose to sample the analog signal from the pickup at the standard frequency of 44.1 kHz. As the upper limit of human hearing is approximately 20 kHz, this allowed us to represent all audible frequencies without aliasing. This was also one of the frequencies supported by our digital to analog converter. Our DAC also supported input in either I2S or Left-Justified format—two digital formats for transmitting audio data. We chose to implement Left-Justified format because it was the simpler of the two.

We first implemented the audio input and output in the FPGA. The FPGA has a 40 MHz clock, so we determined that we should sample the input every clock cycles. Since the audio output had a component which needed to run at double the sampling rate, we chose to sample every 906 cycles instead or 907. This changes our sampling frequency to 44.15 kHz, which introduces a 0.1135% error, which is well within the 5% error tolerance of the DAC [3].

The audio output is synchronized to the input loop and controls two clock lines and one data line to implement Left-Justified format. The proper timing for this format is shown in Figure 2. The first clock signal, the left-right clock, runs at the sampling rate and tells the DAC whether we are outputting a sample for the left or right speaker. We set this output high for 453 cycles, then low for another 453 cycles each loop. The second clock signal, the bit clock, runs at twice the sampling rate times the number of bits per sample. We use 32-bit samples, so the bit clock runs at 64 times the sampling rate. 64 does not divide evenly into 906 however, so we use a period of 14 clock cycles and insert 15 cycle periods at regular intervals in order to reach 906 cycles over 64 periods. The bit clock is held low for 7 cycles and then high for either 7 or 8 cycles to achieve the proper period. During each cycle of the left-right clock, we ready the next audio sample and split it into a 32-bit Boolean array. We then output one bit of the sample per bit clock cycle in the next loop. Since our instrument generates a single audio signal, we simply output the sample twice so that the same audio is played from the left and right speakers.

Figure 2, Left-Justified format timing



We initially tried processing audio in the sbRIO’s microprocessor, but it was unable to handle the MIDI, audio, and motor software together without using very large audio buffers and introducing delays of 100ms or more. This amount of delay was unacceptable, so we instead implemented the processing inside the FPGA. This limited the complexity of effects we could implement, but since samples were now processed in real time instead of being buffered and processed in chunks, this reduced our delay to a single sample. Since we later added an FIR filter to reduce noise, we were only able to implement two audio effects before we reached the limits of the FPGA’s capabilities: distortion and aftertouch.

We wanted to emulate the distortion used in analog guitar effects circuits, but modeling these effects digitally would require more complex calculations than we could perform inside the FPGA. Instead, we found a simple approximation of these effects which could be implemented on the FPGA [8]. The effect works by clamping the amplitude of the signal—effectively cutting off the peaks of the signal and introducing high frequency components which give the audio a harsher tone. This effect is shown in Figure 3. We allow the clamping value to be changed in order to tune the amount of distortion applied.

Aftertouch is the name given by MIDI to the ability to change the volume of a note after it is struck. This makes an instrument respond well to a MIDI wind controller, as the performer can change the volume of the sound at any time by changing the amount of air blown into the controller. To implement this, we first needed to determine the amplitude of the input signal so that we could rescale it to match the amplitude given by the MIDI controller. We used an envelope follower algorithm [9] which is essentially a nonlinear lowpass filter. It rises quickly with the absolute value of the audio signal and falls more slowly, so it rides along the peaks of the signal. We then divide the desired amplitude by the current amplitude and then scale the audio signal by this factor. This effect is shown in Figure 4. This allows us to rescale the audio signal to any arbitrary envelope, but the input signal will eventually become very small since the string will stop vibrating. At this point, we would be amplifying random noise, so once the input amplitude drops below a threshold value, we give up on rescaling and pass the signal through unmodified. To get the desired amplitude envelope, we simply fed a MIDI controller value such as Breath or Volume to the FPGA. We left this controller number configurable, as different MIDI controllers may use different controller numbers.

Figure 3, Distortion effect

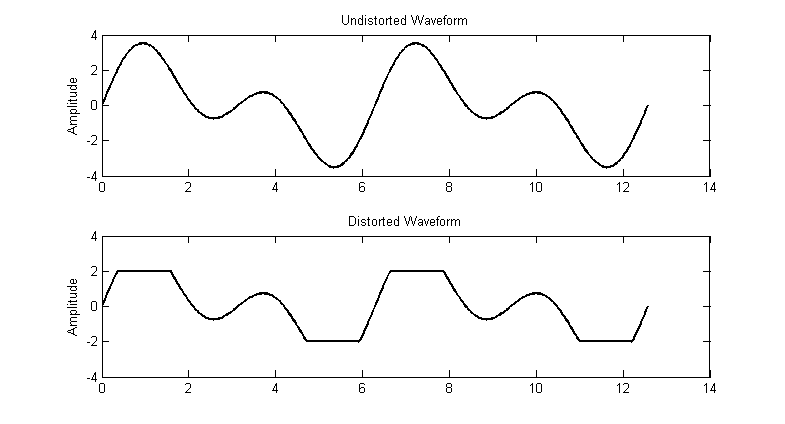


Figure 4, Aftertouch effect

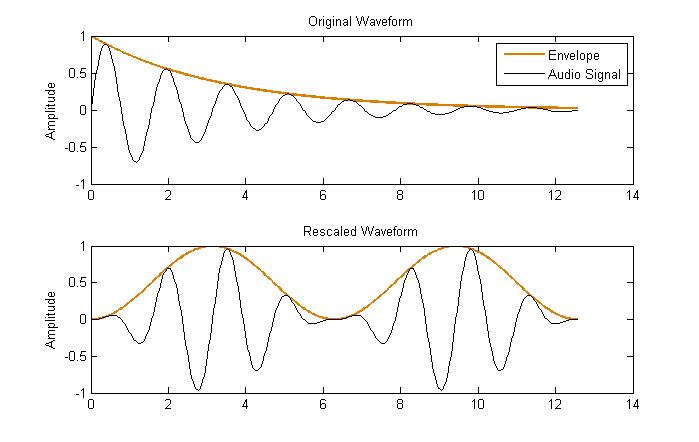


Figure 5, Audio signal flow



When we tested the audio system as a whole, we noticed that the pickup would collect a lot of high frequency noise from the environment. We eliminated this by adding a lowpass filter in the audio software before applying any other effects. The lowest frequencies in the noise were around 3500 Hz, and the fundamental frequency of the highest pitch playable on our guitar is around 800 Hz, so we designed a FIR filter with a stopband starting at 3000 Hz in MATLAB. We needed to use as low a number of filter coefficients as possible to avoid exceeding the capabilities of the FPGA, but large enough that the edge of the passband was at least 800 Hz. We found that a 31 tap filter would give us passband and stopband edges of 1000 Hz and 3000 Hz respectively. The frequency response of this filter is shown in Figure D-1. We implemented this filter by modifying a code sample provided by a National Instruments engineer [10]. This introduced an extra 31 samples of delay and removed overtones from high notes, but removed the high frequency noise entirely.

The final audio data flow is as show in Figure 5. The pickup signal is sampled and then fed into the lowpass filter. The result is fed into the distortion effect and the envelope follower. Either the distorted or undistorted signal is selected depending on whether distortion is enabled. This signal is then multiplied by the scaling factor given by the envelope follower. Either the rescaled or the unscaled signal is then selected depending on whether aftertouch is enabled. The result is converted to left justified format and sent to the DAC circuit.

## 2.10 Digital to Analog Converter

In order to convert the digital audio signal to an analog one, we chose to use a Texas Instruments PCM5100a DAC. The PCM5100a is designed for audio conversion and supports input in either I2S or Left-Justified format, and it supports a wide range of sampling frequencies including 44.1 kHz. It automatically determines the input frequency and can automatically generate a system clock if one is not provided. It also requires 3.3 V digital logic inputs, which are compatible with the 3.3 V GPIO on the sbRIO. This made the PCM5100a a perfect fit for our application.

The PCM5100a required 3.3 V power, but the sbRIO only supplied 5 V power, so we used a LD1117V33 3.3 V linear regulator to step down the power from the sbRIO to 3.3 V. The PCM5100a data sheet provided a sample application circuit, which we determined we could use with only a single modification [3]. Since we did not use a pre-emphasis filter, the DAC’s de-emphasis filter was unnecessary, so we held that input low to disable the filter. Our final circuit, shown in Figure B-6, consists of a network of decoupling capacitors to stabilize power to the DAC and a lowpass filter on each of the output lines to eliminate high frequency noise.

We implemented this circuit with the PCB shown in Figure B-7 and Figure B-8. The power, data, and ground pins of the circuit were connected to power, GPIO, and ground pins on the sbRIO respectively.

# 3. Design Verification

## 3.1 Power Supply

The various components of our project required 24 V power ±5%. We used a voltmeter to check the voltage of the supply and then used the potentiometer on the supply to adjust it to 23.97 V.

## 3.2 Microcontroller

The sbRIO required between 17 and 30 volts to operate properly. This is satisfied by the requirements of the power supply, and when the power supply was plugged in to a wall socket the board powered on.

For the 5 V circuits to operate properly, the board must supply 5 V ±10% on its 5 V power pins. We verified this with a voltmeter and found that they supplied 4.98 V.

Before we could program and use the board, it needed to be properly configured. We used National Instruments’ included software to configure the board to communicate with our computers. After configuration, the status LED turned off to indicate no error and we were able to connect to and program the board using LabVIEW.

## 3.3 MIDI System

For the MIDI system to operate properly, the microcontroller must be running and it must supply 5 V ±10%. This is satisfied by the requirements of the microcontroller.

The MIDI input circuit must transfer MIDI input to the microcontroller at proper digital logic levels. Logic low must be less than 0.8 V and logic high must be greater than 3.0 V for the sbRIO to properly receive MIDI data. We verified this using a 5 V power source and a voltmeter to determine that a 0 V input produced an output less than 0.8 V and a 5 V input produced an output greater than 3.0 V.

We originally wanted to test the reliability of the data transfer by sending a 10,000 bit stream of MIDI data and comparing the sent and received values. Repeated tests would allow us to test whether the bit error rate was 1 in 10,000 or less. We were unable to find software that would allow us to send such a large arbitrary data stream over MIDI, so we were unable to perform this test. However, during the operation of our project, we never observed any data loss or corruption, so we can conclude that the communication is reliable enough for our purposes.

Finally, we needed to verify that the MIDI software properly interpreted MIDI messages. It must read note on and off messages with the proper pitches and velocities, update controller values properly, and receive pitch bend values properly. We tested this by using MIDI OX—a PC program for sending and receiving MIDI data—and an Electric Wind Instrument—a MIDI wind controller—to send particular MIDI messages. We then verified that the sent data matched the data interpreted by the MIDI software.

## 3.4 Motors and Motor Controllers

The motors controllers must be supplied between 14 and 25 volts. To get as much power out of each motor as possible, we chose to run the controllers at 24 V ±5%. This requirement is satisfied by the requirements for the power supply.

We first needed to verify that the motors themselves worked properly and in both directions. We verified this by connecting the motors to a 24 V power supply and verifying that they spun. We then reversed polarity on the input and tested that the motors spun in the opposite direction.

Second, we needed to verify that the motor controllers worked properly. We did this by powering the controllers and connecting them to a voltmeter to check that the controllers could output +24 V and   
-24 V when pressing on the forward and reverse test buttons included on the controllers. It was not necessary to check the exact voltages, as this would only affect the maximum speed of the motors.

After verifying that the motors and controllers worked, we needed to verify that the motor controllers could drive the motors in both directions and at varying speeds. We did this using an Arduino and code provided by Cytron for testing the motor controllers. We were able to control the direction and speed of the motors by changing the duty cycle of a 10 kHz PWM signal.

Next, we repeated the same test using our own code running on the sbRIO. We first used an oscilloscope to verify that our PWM signal ran at the proper frequency and that we could properly modulate its duty cycle. Then we connected the sbRIO to the motor controllers and were able to control the motors’ direction and speed just as with the Arduino and sample code.

Lastly, we needed to check that our motors did not exceed the capabilities of our power supply. The power supply should not drop 5% below 24 V under heavy load and the current into each motor should not exceed 1 A. We used a voltmeter to measure the power supply voltage while continually running both motors back and forth and never saw the voltage drop more than 300 mV, which is well within our 5% tolerance. We did not have proper cabling in the lab to safely connect an ammeter in line with our motors, and we did not have time to find or make proper cabling, so we were unable to test the current through each motor. We used motor controllers and cabling capable of withstanding up to 10 A however, so the only component that might be damaged by large currents would be the motors themselves. However, we did not notice any degradation in the performance of the motors, so we do not believe the current ever exceeds 1 A.

## 3.5 Motor Control Software

For the motor control software to work, the microcontroller must be running and must supply 5 V ±10% to the encoders. This is satisfied by the requirements of the motor controller. Additionally, the motors and motor controllers must be fully functional.

To get proper position feedback, we needed to verify that the encoders accurately read motor positions. We verified this by manually rotating the motors and checking that the positions reported by the software matched the actual rotation of the motor. Once position feedback was verified, we needed to check that the control for each motor worked properly.

To verify that the picking motor worked properly, we sent pick commands and checked that the pick wheel rotated 180°, struck the string only once, and then stopped. After tuning the PID gains, we also ensured that there was no noticeable delay between playing a note and the string being struck. By sending synthetic MIDI data to the instrument instructing it to pick at ever increasing speeds, we determined that it can play 16th notes at 120 beats per minute, or 8 notes per second. Increasing the speed further caused the timing to become erratic, as the pick had not yet finished rotating before another pick command was sent.

To verify that the pitch motor worked properly, we sent the motor control software different pitches and verified that the pitch slide stopped in the correct position. We then struck the string used a tuner to adjusts the setpoints for each pitch so that the instrument was in tune.

Finally, we needed to verify that the pitch and picking motors were synchronized. We did this by playing notes with a MIDI controller and checking whether we could hear any desynchronization. For small pitch changes, there was no noticeable delay between playing a note and hearing the note at the correct pitch. For large pitch changes, there was a noticeable slide into the new pitch, but we expected this to happen, and it does not affect the ability of the instrument to be played in real time.

## 3.6 Audio Software

For the audio software to function, the microcontroller must be running. This is satisfied by the requirements for the microcontroller.

To test the Left-Justified format output of the audio software, we fed constant value samples into the software and viewed the three digital audio output lines with an oscilloscope. We verified that the clock signals ran at the proper frequencies and that the audio data line contained the proper values.

To test the rest of the audio software, we had to verify that the DAC circuit worked properly. Once that was completed, we tested that the audio input worked by plucking the string and checking that we could hear the string vibration amplified through the speakers. We then tested each audio effect by enabling them individually and checking that the distortion changed the tone of the sound as expected and the aftertouch responded properly to volume control from a MIDI wind controller.

## 3.7 Digital to Analog Converter Circuit

For the DAC circuit to function, the microcontroller must be running and must supply at least 4.4 V to the linear regulator. This is satisfied by the requirements of the motor controller. Additionally, the output of the regulator must be between 3.0 and 3.6 volts. We verified this with a voltmeter and found it to be 3.27 V.

Finally, the DAC must reproduce audio from Left-Justified audio data. Once the audio software was capable of producing Left-Justified format data, we generated a single sine wave and measured the output of the DAC using an oscilloscope and speakers. Using the FFT function of the oscilloscope, we checked that the output frequency matched the frequency of the generated sine wave. We then swept the sine wave frequency from 100 Hz to 20 kHz, repeating this check to make sure it could produce all audible frequencies. Using the speakers, we checked that the output sounded like a pure sine wave through audio equipment.

## 3.8 Overall System Performance

The performance of the motor control exceeded our expectations. For a pitch change within a whole step, we estimate that we are able to reach the desired pitch within 150 ms, and pitch slide can move the entire length of its travel in about two seconds. The picking wheel is able to pick with no noticeable delay as well. The instrument also responds well to fine pitch changes such as vibrato and can be played in real-time using a MIDI wind controller.

We had to slow down the MIDI input loop to 5ms to allow the motor control loop to run quickly, so this introduces a small amount of delay on the input. Ideally we would use a more powerful processor or find ways to improve the efficiency of our code to improve this response time, but since picking and motor control respond without very noticeable delay, this doesn’t appear to be much of an issue.

Audio processing could certainly be improved. Since the pickup signal is very low level, we only use a portion of the resolution of the sbRIO’s analog input. Additionally, the FPGA limits the number and complexity of audio effects we can implement, but since we don’t need to buffer audio and process it in chunks, the amount of delay from audio input to output is only 32 samples, or 0.7 ms at 44.1 kHz.

# 4. Costs

## 4.1 Parts

Many of the components of our project were provided for free from the department, and exact pricing for the components is unknown. For these parts, we have estimated the cost using comparable commercial parts. Table 3 shows the estimated cost of the parts for our project. While the National Instruments sbRIO is perfect for rapid development in a project such as ours, it is also very expensive at $2200 for a single board or $1650 in quantities of 20 or more. If we were to turn this into a commercial product, a more cost-effective solution would be preferable. Our final parts cost was $2772.98.

## 4.2 Labor

We estimated that each of us spend approximately 170 hour doing research, working together in the lab, or working individually on this project. In addition, we estimate that David Switzer from the ECE machine shop spent about 40 hours fabricating the mechanical components of our project. We do not know David’s salary, so we will assume that the four of us each make $40 an hour. Using equation 4.1 to calculate the total cost, we get $55,000 of development costs as shown in

Table 4. This puts the total cost of our project at $57,772.98.

|  |  |  |
| --- | --- | --- |
|  |  | 4.1 |

Table 3, Parts Costs

|  |  |  |  |
| --- | --- | --- | --- |
| Part | Manufacturer | Quantity | Cost ($) |
| 200W Power supply | Kyosan | 1 | (est.) 115.00 |
| 6N138 opto-isolator | Fairchild Semiconductor | 1 | 1.00 |
| 9632 Single-Board RIO | National Instruments | 1 | 2200.00 |
| Guitar string |  | 1 | 6.00 |
| LD1117V33 3.3V Regulator | STMicroelectronics | 1 | 1.95 |
| Magnetic Pickup |  | 1 | 10.00 |
| MD10C motor controller | Cytron | 2 | 29.00 |
| Mechanical parts |  | 1 | (est.) 100.00 |
| Minertia F-Series DC motor with encoder | Yaskawa | 2 | (est.) 200.00 |
| PCBs |  | 3 | 90.00 |
| PCM5100a stereo audio DAC | Texas Instruments | 1 | 1.78 |
| Resistors, capacitors, diodes, cabling |  |  | (est.) 10.00 |
| SDS-50J DIN connector | CUI Inc. | 1 | 3.75 |
| TSSOP-20 to DIP-20 SMT Adapter |  | 1 | 4.50 |
| Total |  |  | 2772.98 |

Table 4, Labor Costs

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Hourly Rate | Hours Invested | Total (Hourly Rate × Hours Invested × 2.5) |
| Angad Bector | 40 | 170 | 17000 |
| Joel Spadin | 40 | 170 | 17000 |
| Ruichen Zhao | 40 | 170 | 17000 |
| David Switzer | 40 | 40 | 4000 |
| Total |  |  | 55000 |

# 5. Conclusion

In conclusion, we were very pleased with the result of our project, as it met or exceeded most of our expectations. The instrument responds quickly enough to real time input to be used in a performance, and is capable of playing repeated notes as quickly as 8 times per second. We were not able to implement all of the audio effects we originally wanted to use due to the limitations of our microprocessor and FPGA, and audio quality was somewhat subpar, but the audio effects we did implement gave the instrument an interesting tone and made it respond well to MIDI wind controllers.

## 5.1 Future work

After finishing our project, we noted a number of ways we could improve or expand upon it.

Firstly, we could use a humbucking pickup to reduce environmental noise and a pre-amplifier to increase the resolution of our input samples. We noticed that the audio output of our project was very quiet, so we could add a power amplifier to increase output volume.

Since the processor and FPGA we used were not designed to handle audio processing, the amount of processing we could do was very limited. We could use a dedicated DSP or a board with a faster processor instead to apply more complex audio effects and give our instrument a more interesting tone.

Our final design left all of our circuit boards exposed. To protect the boards from the environment, improve safety by keeping user away from high voltages, and improve the aesthetics of the instrument, we could design enclosures for the circuitry.

Another minor flaw in our instrument is that the motorized components make some mechanical noise. The pick makes a very audible click with each struck note. If we were to enclose the entire system inside a clear case to soundproof it, that would reduce the mechanical noise. Additionally, it would improve safety by preventing users from contacting the moving parts while still leaving them visible so that people can see the inner workings of the machine.

Finally, we could create an array of six motorized strings controlled from a single board. This would allow us to better emulate a slide guitar and give us the capability to play multiple notes at once.

## 5.2 Ethical considerations

The statements of the IEEE Code of Ethics that pertain to our project are as follows [11]:

1. “to accept responsibility in making decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment;”

The development of our project posed no public safety risks as we did not use any hazardous materials, but our final product contains moving parts and dangerous electrical currents. We insulated all high voltage cables to reduce risk of electric shock. Additionally, we have made sure that people are clear of the moving parts of the instrument before operating it.

1. “to be honest and realistic in stating claims or estimates based on available data;”

We recorded our experimental data accurately to the best of our ability for future reference. We did not modify or falsify data for any reason.

1. “to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;”

When we encountered areas outside of our expertise—especially when dealing with the potentially dangerous components of our project—we consulted with members of the ECE staff to ensure that our design was correct and that we created a safe product.

1. “to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;”

We checked each other’s work and provided feedback to correct errors and improve the project. We have credited any work that we used—either directly or after modification—in the design and construction of our project with the references section of this report.

# References

|  |  |
| --- | --- |
| [1] | National Instruments, "NI sbRIO-961x/963x/964x User Guide," [Online]. Available: http://www.ni.com/pdf/manuals/375052c.pdf. |
| [2] | Fairchild Semiconductor, "Single-Channel: 6N138, 6N139 Low Input Current High Gain Split Darlington Optocouplers," August 2008. [Online]. Available: http://www.fairchildsemi.com/ds/6N/6N138.pdf. |
| [3] | Texas Instruments, "112/106/100dB Audio Stereo DAC with 32-bit, 384kHz PCM Interface," September 2012. [Online]. Available: http://www.ti.com/lit/ds/slas859a/slas859a.pdf. |
| [4] | MIDI Manufacturers Association, "MIDI Electrical Specification & Proper Design of Joystick/MIDI Adapter," [Online]. Available: http://www.midi.org/techspecs/electrispec.php. |
| [5] | D. V. Brink, "David's MIDI Spec," [Online]. Available: http://www.srm.com/qtma/davidsmidispec.html. |
| [6] | National Instruments, "Quadrature Encoder Velocity and Acceleration Estimation with CompactRIO and LabVIEW FPGA," 3 February 2012. [Online]. Available: http://www.ni.com/white-paper/3921/en. |
| [7] | National Instruments, "Pulse Width Modulation (PWM) IP Core for LabVIEW FPGA," 6 September 2006. [Online]. Available: http://zone.ni.com/devzone/cda/epd/p/id/979. |
| [8] | F. R. Y. J. Y. E. C. Navarun Jagatpal, "Guitar Effects," 22 March 2007. [Online]. Available: http://www.cs.columbia.edu/~sedwards/classes/2007/4840/designs/GuitarEffects.pdf. |
| [9] | Bram, "Envelope Follower with Different Attack and Release," [Online]. Available: http://musicdsp.org/showArchiveComment.php?ArchiveID=136. |
| [10] | C. Loew, "Is it possible to build a low pass filter on LabVIEW FPGA?," 4 February 2004. [Online]. Available: http://forums.ni.com/t5/LabVIEW/Is-it-possible-to-build-a-low-pass-filter-on-LabVIEW-FPGA/m-p/118409#M71425. |
| [11] | IEEE, "IEEE Code of Ethics," [Online]. Available: http://www.ieee.org/about/corporate/governance/p7-8.html. |
| [12] | Cytron Technologies, "MD10C," February 2013. [Online]. Available: http://www.cytron.com.my/usr\_attachment/MD10C\_Rev2.0\_User's\_Manual.pdf. |

# Appendix A Requirement and Verification Table

|  |  |  |
| --- | --- | --- |
| Requirement | Verification | Verified? |
| 1. Microcontroller initializes properly    1. Power supply should be between 17 and 30 volts    2. The board should power on    3. The board supplies 5 volts ±10% to digital logic circuits    4. The board is properly configured      * 1. Connection between a computer and the microcontroller can be made. LabVIEW is able to communicate with the microcontroller. | * 1. Verify power supply voltage with a voltmeter   2. The power (top-right) LED is lit   3. Verify 5V pin voltages with a voltmeter   4. The status (bottom-right) LED is not flashing. If it flashes, check page 22 of the user guide to determine the specific error and troubleshooting steps.   5. Connect the microcontroller to a computer with an Ethernet cable. Check that LabVIEW is able to establish a connection. | Y  Y  Y  Y  Y |
| 1. MIDI module and software function properly    1. Microcontroller must initialize properly    2. Power terminal of MIDI input module must be at 5 volts ±10%.    3. Circuit transfers MIDI input to 5V serial output. Logic high is > 3V. Logic low is  < 0.8V.    4. MIDI data received by the microcontroller is identical to that which was sent. Bit error rate is 1 in 10,000 or less.    5. MIDI data is read properly       1. Pitches are correct       2. Velocities are correct       3. Continuous controller (CC) values are correct       4. Pitch bends are correct | * 1. See requirement 1.   2. Verify voltage with a voltmeter   3. Connect a MIDI instrument and send MIDI data while measuring the serial output with an oscilloscope. Freeze the display and verify voltages.   4. Send MIDI a stream of 10,000 bits. Record the received data and compare. Repeat to improve confidence.   5. Connect a MIDI instrument      1. Play notes at different pitches and verify that received pitches match.      2. Play notes at different velocities and verify that received velocities match.      3. Send CC values of various values for various controllers. Verify that controller numbers and values match      4. Change the pitch bend and verify that received values match. | Y  Y  Y  N  Y  Y  Y  Y |
| 1. Motors function properly    1. Motor controllers are supplied 24 volts ±5%.    2. Motors work    3. Motor controllers work    4. Motor controllers drive motors    5. Motor speeds can be controlled with PWM signal    6. Microcontroller sends PWM signal correctly    7. Motors are supplied enough power under load.          1. Supply voltage does not drop       2. Supply currents do not exceed rated values for motors    8. Supply currents do not exceed rated values for motors when motors are stalled | * 1. Verify power supply voltage with a voltmeter   2. Connect motors to a 24V power supply with variable current capability. Increase current from 0 and verify that motors rotate. Do not exceed 1A. Reverse polarity and repeat.   3. Connect controller outputs to a multimeter. Press the test buttons on the controller. Verify that test LEDs light and the two buttons output positive and negative currents respectively.   4. Connect motors to controllers and press the test buttons on the motor controller. Verify that test LEDs light and motors turn in both directions.   5. Use an Arduino and the sample program provided by Cytron to send direction and PWM signals. Verify that the direction pin changes the direction of the motor and that the duty cycle of the PWM signal modulates the motor speed.   6. Repeat verification 3.e using the microcontroller. Use an oscilloscope to verify that the high voltage is > 3V.   7. Connect the motors to their loads. Run a test program on the microcontroller that runs the motors in both directions to simulate normal operation.      1. Verify that the input voltages to the controllers do not drop more than 5%.      2. Verify that the current into each motor does not exceed 1A   8. Prevent the motors from turning. Drive the motors and verify that the current into each motor does not exceed 1A. | Y  Y  Y  Y    Y  Y  Y  Y    N  N |
| 1. Motor control software functions properly    1. Microcontroller must initialize properly    2. Motors must function properly    3. Power to encoders must be 5 volts ±5%    4. Microcontroller should accurately read encoder position values      * 1. Control software is able to control picking motor properly   2. Control software is able to control pitch motor properly   3. Software properly converts pitches to pitch bar positions   4. Software coordinates picking and pitch motors | * 1. See requirement 1.   2. See requirement 3.   3. Verify voltages with a voltmeter   4. Disconnect power from the motors. Manually turn the motors arbitrary angles and verify that the encoder positions change properly.   5. Instruct the control software to pick once. Verify that the picking motor rotates enough to pick the string once and stops. Repeat and send commands in different rhythms. Verify that the string is hit with the proper rhythm.   6. Instruct the control software to move the pitch bar to a specific position. Verify that the bar reaches that position. Repeat for various positions.   7. Instruct the control software to move to a specific pitch. Wait for the pitch bar to stop, then strike the string. Use a tuner to verify the pitch.   8. Play a sequence of notes into the control software. Verify that the correct pitches and rhythms are reproduced. | Y  Y  Y  Y  Y  Y  Y  Y |
| 1. Pickup functions properly    1. Pickup converts string vibration to electric signal. Amplitude is < 200mV.    2. Signal can be read by microcontroller analog input    3. Signal does not saturate analog input    4. Resolution of analog input is high enough to capture small vibrations | * 1. Manually pluck the string and measure pickup output with an oscilloscope. Verify that a signal is received. Verify its amplitude.   2. Connect pickup to microcontroller manually pluck the string. Verify that microcontroller receives values.   3. Graph analog input values while plucking the string. Verify that waveforms do not reach the maximum numerical value.   4. Verify that microcontroller captures a signal when manually tapping on the string. | Y    Y    Y  Y |
| 1. Audio software functions properly    1. Microcontroller must initialize properly    2. Left-Justified generator converts samples to Left-Justified format    3. Software reads string signal and outputs audio    4. Software applies audio effects to signal    5. Audio effects can respond to MIDI values | * 1. See requirement 1.   2. Feed constant value audio samples into the generator. Use an oscilloscope on the microcontroller outputs to verify that the Left-Justified data matches specifications. Trigger on the LRCK channel to view a single audio frame at a time. Repeat for at least 20 different sample values.   3. Check requirements 5 and 7.b. Disable all audio effects. Connect speakers to the line out and pluck the string. Verify that it plays a sound.   4. Enable audio effects individually and pluck the string. Verify that each effect changes the tone of the sound properly.   5. Check requirement 2. Configure the aftertouch effect to respond to the modulation wheel CC. Vary the mod wheel value while playing and verify that the output volume changes. | Y  Y    Y  Y  Y |
| 1. DAC circuit functions properly    1. Microcontroller must initialize properly    2. Power terminal of DAC module must be at least 4.4 volts    3. Power after voltage regulator must be 3.0–3.6 volts    4. DAC reproduces audio from Left-Justified data | * 1. See requirement 1.   2. Verify voltage with a voltmeter   3. Verify voltage with a voltmeter      * 1. Check requirement 6.b. Generate Left-Justified signals from a sine wave. Sweep the waveform frequency from 100 Hz to 20 kHz while measuring the line out with an oscilloscope with FFT capability. Verify that the output frequency always matches the test signal frequency. | Y  Y  Y  Y |

# Appendix B Mechanical and Circuit Design

These pictures and diagrams show the circuitry, PCB design and implementation for each circuit in our project as well as the electrical interconnections between modules. Additionally, they show the overall hardware design of the project.

Figure B-1, MIDI input circuit



Figure B-2, MIDI circuit PCB layout

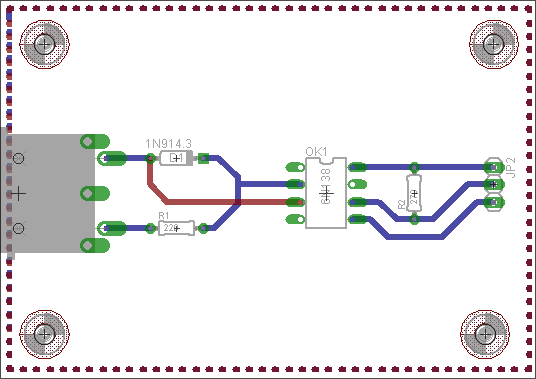


Figure B-3, Picture of MIDI circuit

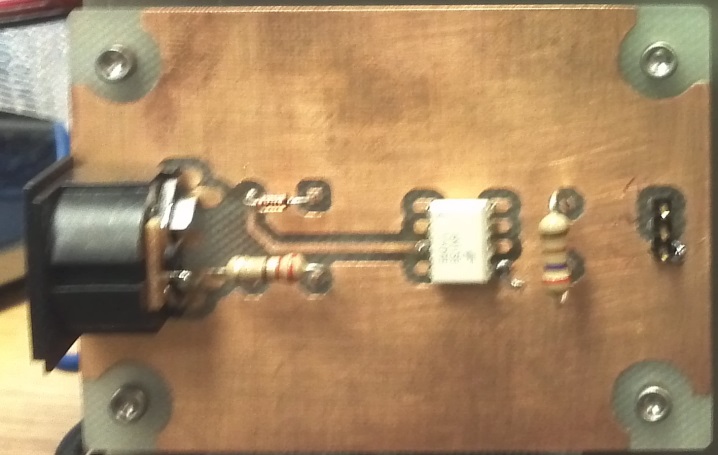


Figure B-4, Motor control circuit



Figure B-5, Analog input grounding circuit



Figure B-6, Digital to analog converter circuit



Figure B-7, Digital to analog converter PCB layout

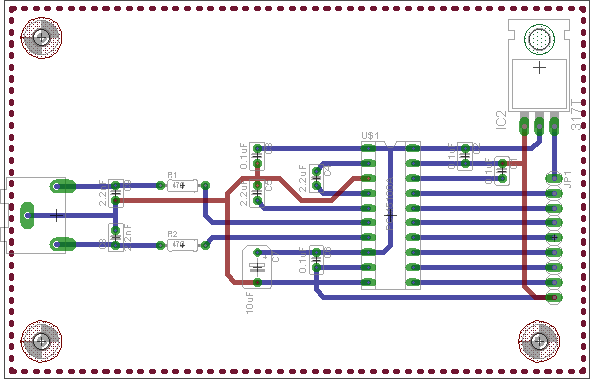


Figure B-8, Picture of digital to analog converter PCB

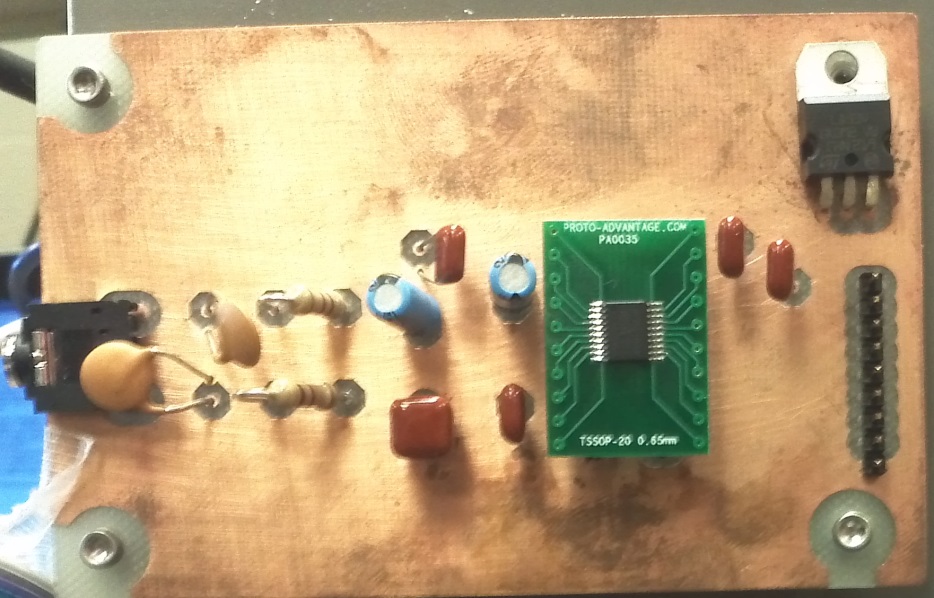


Figure B-9, sbRIO connections



Figure B-10, Component connections



Figure B-11, Mechanical system



Figure B-12, Picture of guitar instrument

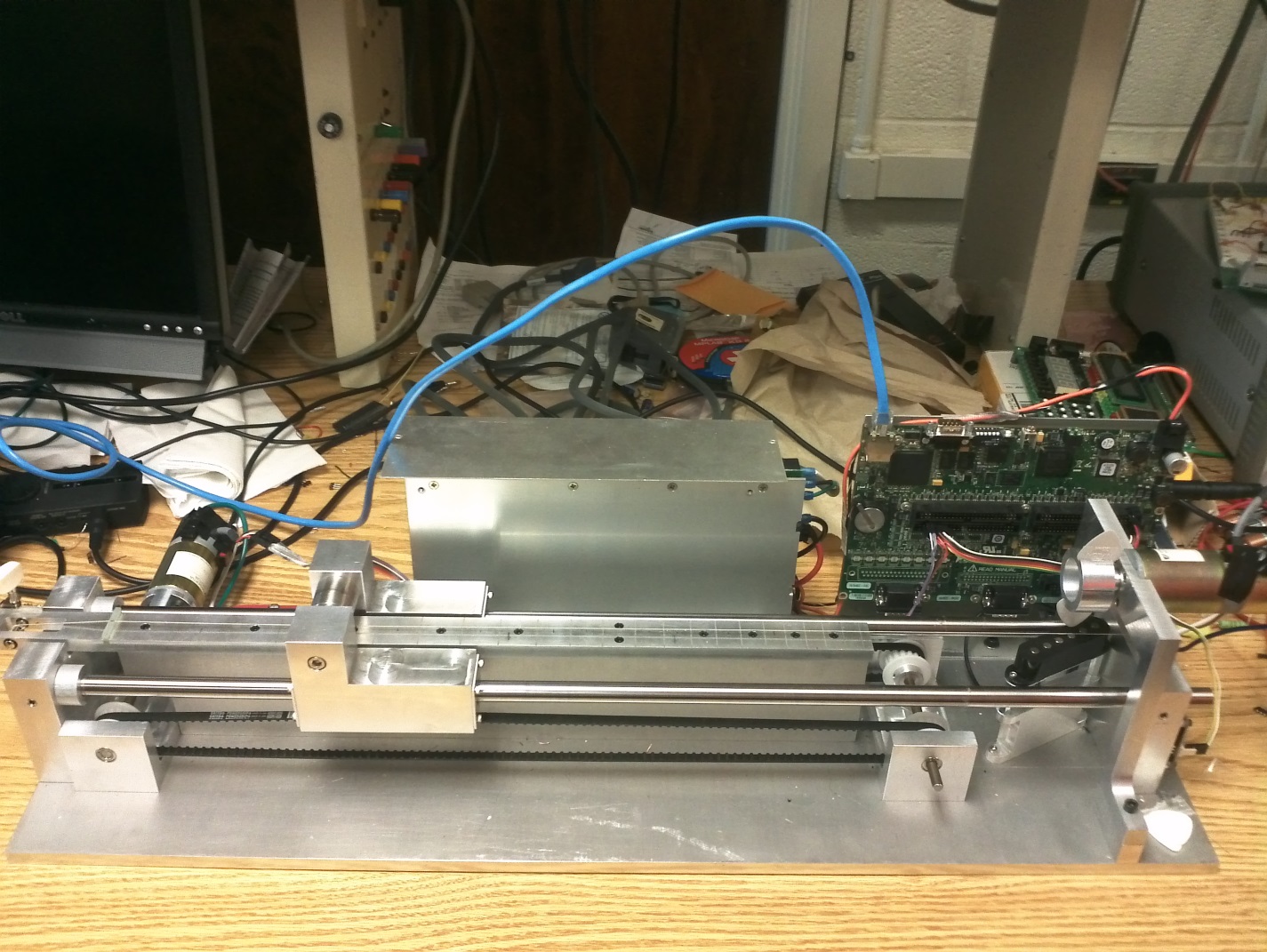


Figure B-13, Picture of motor, encoder, and motor controller



Figure B-14, Picture of pick wheel in normal position

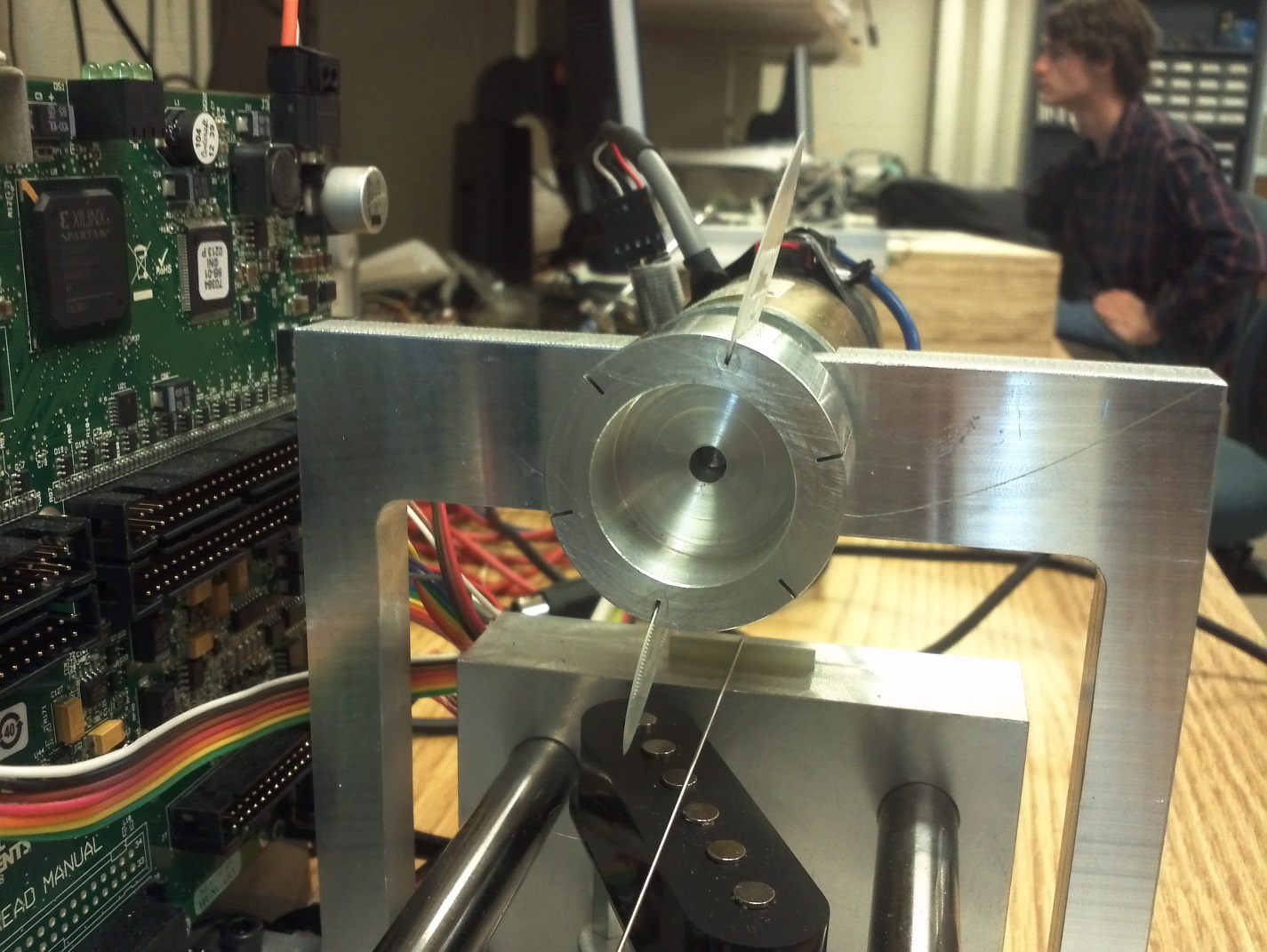
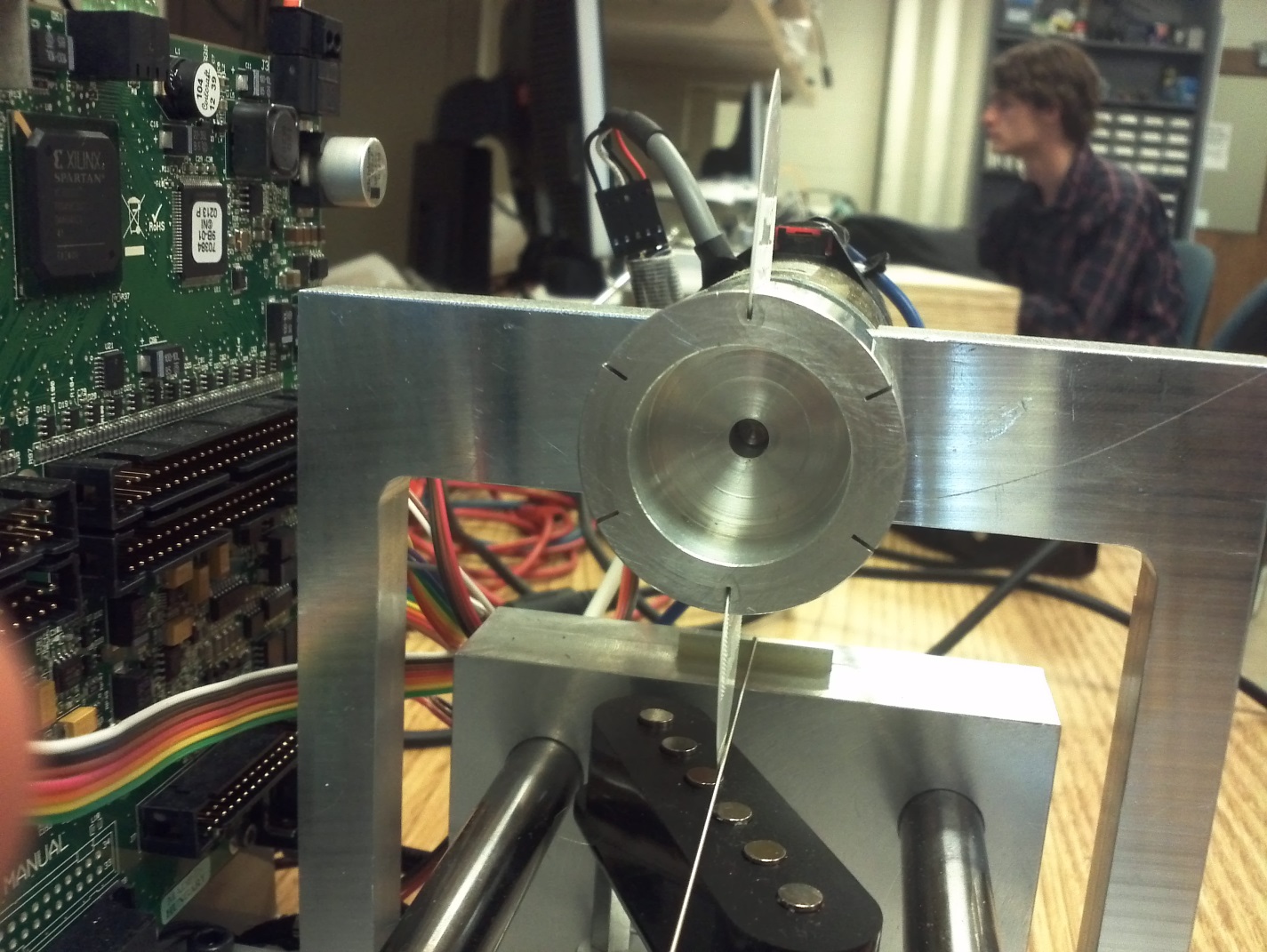


Figure B-15, Picture of pick wheel in damping position



# Appendix C Software Diagrams

These flowcharts describe the algorithm used to interpret MIDI data. Figure C-1 shows the main input loop, while Figure C-2 shows the subroutine which processes data bytes according to the message type given by the most recent status byte.

Figure C-1, MIDI message interpretation algorithm



Figure C-2, Process data byte subroutine



# Appendix D Design Data

Table D-1 shows the table used to convert pitches to slide positions. Table D-2 gives the PID gains for the two modes of pick control and two modes of pitch control, which we determined by manually tuning the control loops. Figure D-1 gives the frequency response of the lowpass filter used to reduce noise. The frequency axis is normalized and corresponds to the range between 0 and 22.05 kHz.

Table D-1, Note number to slide position conversion

|  |  |  |
| --- | --- | --- |
| Pitch | Note Number | Slide Position (cm) |
| E♭3 | 51 | 0 |
| E3 | 52 | 3.5 |
| F3 | 53 | 6.6 |
| G♭3 | 54 | 9.6 |
| G3 | 55 | 12.5 |
| A♭3 | 56 | 15.2 |
| A3 | 57 | 17.7 |
| B♭3 | 58 | 20.0 |
| B3 | 59 | 22.4 |
| C4 | 60 | 24.4 |
| D♭4 | 61 | 26.5 |
| D4 | 62 | 28.35 |
| E♭4 | 63 | 30.1 |
| E4 | 64 | 31.8 |
| F4 | 65 | 33.4 |
| G♭4 | 66 | 35.1 |
| G4 | 67 | 36.5 |
| A♭4 | 68 | 37.6 |
| A4 | 69 | 39.0 |
| B♭4 | 70 | 40.1 |
| B4 | 71 | 41.5 |

Table D-2, PID gains

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Pick String | Dampen String | Fast Pitch | Smooth Pitch |
| P | 0.01 | 0.01 | 0.08 | 0.01 |
| I | 0.002 | 0.001 | 0.0014 | 0.004 |
| D | 0.0004 | 0.004 | 0.00048 | 0.00012 |

Figure D-1, FPGA lowpass filter frequency response

