Polyphonic Analog-to-MIDI Converter for Musical Applications

Senior Design 2

Final Document

University of Central Florida

Department of Engineering and Computer Science

Group 12

Andrew Obeso-Silva	Computer Engineering
Colin Smith	Electrical Engineering
Noah Watts	Computer Engineering
Sean Dillon	Electrical Engineering

Table of Contents

4. Related Standards and Constraints

4.1. Standards

4.1.1. ¼-Inch Audio Jack Port and Connector

4.1.2. XLR3 Audio Port and Connector

4.1.3. 48 Volt Phantom Power

4.1.4. Musical Instrument Digital Interface (MIDI)

4.1.4.1. Physical and Link Layers

4.1.4.2. Application Layer

4.1.5. Universal Serial Bus (USB)

4.1.5.1. Physical Layer

4.1.5.2. Link Layer

4.1.5.3. Network Layer

4.1.5.4. Transport Layer

4.1.5.5. Application Layer (Standard)

4.1.5.6. Application Layer (USB-MIDI)

4.1.6. Design impact of relevant standards

4.2. Realistic Design Constraints

4.2.1. Economic and Time constraints

4.2.2. Environmental, Social, and Political constraints

4.2.3. Ethical, Health, and Safety constraints

4.2.4. Manufacturability and Sustainability constraints

5. Hardware Design Details

5.1. Power Circuits

5.1.1. Power Switching Circuit

5.1.2. Complete Power Circuit

5.2. Analog Input and Output

5.3. Analog Preamplifier

5.4. Buffered Analog Signal Splitter

5.5. Adjustable Band-Pass Filters

5.6. MIDI Output

5.7. USB Controller Connections

5.8. Processor Connections

5.9. Device Casing and Other Hardware

5.10. Complete Integrated System

6. Software Design Details

6.1. ADC Driver and Sample Buffer Generation

6.2. FFT and Frequency Spectrum

6.3. Magnitude-Based Noise Filtering and Normalization

6.4. Q-Factor-Based Filtering

6.5. Note, Magnitude, and Effects Determination

6.6. MIDI Stream Generation

6.7. MIDI Driver

6.8. USB Driver

List of Tables

List of Figures

1. Project Summary

The Polyphonic Analog-to-MIDI Converter for Musical Applications is a device that will accept an analog signal from a quarter-inch or XLR3 tip-sleeve audio jack and convert it to a MIDI signal of up to six voices. It is designed to be used in both a music studio and live performance settings. MIDI technology is most often purely digital, with some digital MIDI controllers interfacing with a program or instrument that operates on the MIDI standard. While some products exist which that convert an analog music signal into a digital MIDI stream, they are usually monophonic (single-voiced). We aim to develop an analog-to-MIDI device that is capable of translating each string of a standard-tuning guitar simultaneously. While the main goal is for the device to work with a guitar, it will be designed to be able to work with any instrument capable of producing a consistent tone in the 12-tone equal-temperament tuning system.

While the MIDI stream is being created, the original instrument signal is simultaneously bypassed out of the device. This allows the device to be a single block in the instrument's signal chain and for the signal to pass through to other devices, amplifiers, or interfaces. This functionality opens a world of creative possibilities. The Analog-to-MIDI Converter can be used to transcribe the notes being played or control a MIDI device playing in parallel with an instrument while still allowing the instrument's voice to be heard. Designed with form, function, and compatibility in mind, the Analog-to-MIDI Converter will be lightweight and powered by +5V via USB or +9V via DC barrel jack. USB is ubiquitous in studio settings where musicians can connect to their desktop or laptop workstation and negative center DC barrel jack is a widely used power standard in music technology for performance. The most important challenges to overcome in this design are the analog to digital conversion, discrete Fourier transform (DFT), and filtering. The ADC must be fast enough to pick up all of the notes being played, the DFT must be accurate without slowing down the program too much, and the note detection must be able to identify all of the up to six voices for notes which may be only 5 Hz apart.

The prototype device presented at the Senior Design Showcase lacks many intended features, such as USB functionality, 9 volts power, and XLR3 input and output. However, the core functionality of polyphonic analog-to-MIDI conversion is implemented, and it is confirmed to work for a guitar as intended using external 5-volt and 3.3-volt power sources. This document includes the original intended design and feature set, details on the failure of the implementation of cut features, and new ideas and designs that avoid the failures we had.

2. Project Description

Many different technologies, both ancient and modern, are used in the music industry to create music. The instruments that musicians use can be either acoustic or electrical, and the electrical instruments can be either analog or digital. In the production studio, the sounds from all these different instruments are captured electronically for manipulation in a software application called a digital audio workstation (DAW). In music production, there are many ways to capture and incorporate analog and acoustic instruments into a DAW. Most of these solutions involve USB microphones or an audio interface that digitizes your analog signal for DAW use. Very few options exist that allow you to use your analog or

acoustic instruments in the same way that you might use a digital instrument. A digital instrument using MIDI protocol will have the notes and other musical information encoded into a message stream rather than a digital replication of an analog signal. The sounds of acoustic instruments are captured with a microphone that converts the sounds into an analog electrical signal. Analog electrical instruments also output an analog electrical signal. This signal is then digitized and recorded in a computer. Digital instruments send digital signals. They can send a digital representation of an analog signal, or they can send messages that can tell another system what sounds to produce. For example, a keyboard can send a message that tells the computer that a note was played. The computer can then output the corresponding sound. Unfortunately, only these digital instruments can send these encoded messages. Other instruments, like an electric guitar or a flute, cannot do this. What this project aims to achieve is to create a device that gives acoustic and analog electrical instruments the ability to send encoded, digital messages just like a digital instrument.

2.1. Motivation

Our motivation for this project is to apply what we have learned in our classes at the University of Central Florida. This will be one of our first experiences completing a project of this scale in a team, so we will also be building our teamwork and coordination skills to be applied in our careers moving forward. Furthermore, we all share an interest in music technology which drove us toward this project, and we came up with this idea since it can be very useful to both musicians and engineers. The ability for this device to take a microphone input and convert it to a MIDI stream opens a world of possibilities where any instrument can be used as a MIDI controller. This includes acoustic guitar, trumpet, flute, accordion, even voice. If it can be captured by a microphone or a pickup and produce a consistent tone, then it is compatible with one of our inputs.

Additionally, in a live music performance setting, musicians tend to either play an analog instrument or a digital instrument with very few options for blending the two. Percussionists have the option to attach digital “trigger” devices to their instruments, connecting them to a DAW and allowing them to use the drums to play a digital instrument. There are no clear equivalents to such a device for other instruments such as guitar, voice, or acoustic piano. This is likely because of the much higher processing power required to decompose an analog signal in real time with a DFT versus a binary on or off signal that can be used for percussion trigger devices.

Another useful application of this device would be note transcription. When musicians write down or transcribe their music compositions, they will typically use a digital instrument such as an electric keyboard or manually program notes into software. If the music is written for a different instrument than a piano, it may be very difficult to perform the piece on a keyboard to transcribe it and would require the musician knowing how to play the instrument. If the musician instead inputs notes manually into the software, they must choose the note length, the note, volume, and articulation of the note. This can be incredibly inconvenient and time consuming. With this device, you can instead play your analog or acoustic instrument of choice directly into the device and into your MIDI compatible transcription software. This will allow a musician to play the instrument that they are comfortable on and originally composed the music on while automatically transcribing their composition. We aim to blend forms of digital and analog music making with a Polyphonic Analog to MIDI Converter. Such a device will allow the user to transcribe

their notes into MIDI format or control a digital instrument in real-time using their analog or acoustic instrument of choice.

2.2. Objectives

The primary purpose of this device is to detect more than one note being currently played by an instrument in an analog audio signal and transmit that information through a MIDI stream. The device will ideally take analog input from a ¼-inch tip-sleeve mono audio jack and from an XLR3 connection. Most electronic devices with musical applications typically transmit analog audio signals through these connections. For example, electric guitars typically use a ¼-inch jack, and professional microphones typically use an XLR3 connection. There are other interfaces that could be implemented into the device, like USB, coaxial, and optical. However, the interfaces we have selected are the most common in music production applications, and these analog interfaces are simpler to implement than those three digital interfaces. It will have a switch to select the source of the input. Also, some microphones require a phantom power source from the input port that they connect to. The device must be able to supply enough power through the XLR3 port to power as many microphones as possible, and there will be a switch that toggles the availability of this power source. The device can output a MIDI stream through a MIDI port and through a USB port. These are the most popular interfaces through which MIDI streams are transmitted. Like the input ports, the device will have another switch to select the output port. The device will also have another ¼-inch tip-sleeve jack port and another XLR3 port to output an exact copy of the input. The purpose of this is to allow the signal to be passed through the device to other systems. This makes integration of this device into a system of other audio devices simpler and with less conflicts. There will also be a switch to toggle each of these outputs. We will need to buy ports for these interfaces and implement electrical circuitry that transmits the signals to and from these ports. We will need to use a dedicated integrated circuit for our USB implementation since it is a complex standard.

The device must be capable of digitizing audio signals within the range of frequencies that MIDI establishes as notes for processing. To prepare the analog signal for the analog-to-digital converter (ADC), we will have a preamp stage that amplifies the line-in signal to a range that is accepted by the ADC. The user can adjust the gain for this preamp using a potentiometer on the faceplate of the device in order to improve the accuracy of the device for whatever input source they are using and to prevent clipping. This preamp stage will also include a bandpass anti-aliasing filter to make the signal easier to work with once it is digitized. An ADC is then used to collect samples of the signal, but its sample rate must be high enough to digitize the signal accurately at the highest frequencies. The device must analyze a chunk of the audio signal and determine which notes are being played, if any. Musical notes are related to different sound frequencies, so the device must calculate the frequency content of the chunk of signal. A DFT is capable of converting a chunk of a digital audio signal into a spectrum of frequencies that make up the sound. The frequency spectrum lists the magnitudes of each multiple of a fundamental frequency, thus also giving information on the loudness of each note the device hears. However, the issue with this is that the spectrum has a linear frequency scale, while musical notes have a logarithmic frequency scale. The device needs enough frequency density in the spectrum to be able to accurately relate frequencies to notes. We will use a processor that is powerful enough to perform the DFT in real time on chunks of the audio signal. Also, there may be several notes being played at the same time, so the device must be capable of

detecting more than one note. This is called polyphony. As discussed in a later section, other analog to MIDI products exist, but they are exclusively monophonic, they can only convert one note at a time. The novel feature of our device will be its ability to convert multiple notes at once.

Using the frequency spectrum, the device must accurately determine which notes are being played and transmit the corresponding MIDI messages in time. The input will typically never be digitized into a mathematically perfect sine wave, even from analog synthesizer instruments, and especially not from acoustic or electric instruments like the guitar. Due to the complexity of typical input signals, the device must be able to distinguish between the overtones and the fundamental frequency of each note, and it must also filter out noise. Being able to filter out the noise, overtones, and lower resonant frequencies of each note is important in accurately determining the notes being played since only the fundamental frequencies of each note will be left over. After determining which notes are being played, it outputs that information to a MIDI output stream. The device must perform the translation from analog audio to MIDI within a very short amount of time. With low latency, the device becomes effective for use in live recordings and performances. Timing is critical in music production since the art itself relies on the arrangement of sounds in time. If the device outputs an instruction too late, then it will ruin the recording or performance because the effect or sound that the device was supposed to trigger would have happened at the wrong time. The processor and the DFT implementation will be an important factor in determining our latency because the DFT is the most time-consuming computation that the processor will perform.

There are other aspects that this device must achieve. It must be portable and small, since its potential applications include live performances, and the cost decreases with a smaller size. For example, if the device takes up too much space on a guitarist's pedal board, then it is useless because other tools will have priority over it. The device needs to be safe to handle and never get too hot to touch. This requires a smooth plastic case, suitable passive cooling systems, and low-power electronics. While we could implement an active cooling system with a fan and control circuit, this would make the device less portable, increase the size and cost, and create EMI. Reducing the amount of EMI that this device produces, and experiences is critical because analog audio signals are sensitive to interference. This interference creates noise in these signals when they are transmitted along unbalanced connections and along the PCB traces, and the noise will affect the accuracy of the note translation. The objectives that this product aims to achieve are the main goals. The project is successful if these goals are met.

There are also stretch goals for this project that we will attempt to implement but are not required for the final product. We would like to make the device capable of recognizing the instrument that produced the audio signal and detect and relay note effects such as vibrato, tremolo, pitch bend, note slur, etc. Doing this requires much more work on the algorithms that analyze the frequency spectrums and accurate detection of these effects without interfering in the detection of notes. We would also like to make the device configurable through the USB interface. The settings that would be changed are the equalizer, input gain, maximum polyphony, pitch range, and effects processing. The user would be capable of changing these settings anytime while the device is connected to a computer using a simple computer application. Finally, we would like to make the input band-pass filter and ADC sampling rate adjustable. This would allow optimized processing for low pitch ranges, since digitizing signals with only low frequencies does not require a

high sample and bandwidth. These goals make the device more versatile and more efficient. However, they are not required for the main functionality and are thus considered to be stretch goals.

2.3. Requirements Specification

The requirements specification defines the parameters for the device to be considered successful. These requirements are either self-imposed for the feature set of the device or required by the components needed to implement the functions of the device. For each requirement there are three value columns, minimum, maximum, and ideal. The minimum and maximum columns of the table are the minimum and maximum values that each characteristic of the device must have for our project to be considered successful for the purposes of Senior Design. This value is lower than the ideal value because we realize that we may be limited in the performance of the device by our budget, parts, and time. We want to create at least a working prototype of the device that can show that the ideal values are potentially possible by using more expensive parts. The ideal value column of the table shows the ideal required values we would like to have if we did not have any limitations. Some of the minimum and ideal values are the same due to being necessary for the device to be functional.

Requirement	Description	Unit	Min	Max	Ideal
Polyphony	Number of simultaneous notes converted		6		128
MIDI Stream Latency	Amount of time between signal reception and note output	ms		100	10
Device Temperature	Long-term temperature. Must not require active cooling.	℃		30	
Preamp Gain	Range of values for linear gain of adjustable preamp stage.	V/V	1.2	600	
Length	Length of device.	inch		12	
Width	Width of device.	inch		9	
Height	Height of device.	inch		6	
Weight	Device must be light-weight for portability.	lb		0.5	
¼” Audio Input Voltage Range	Voltage range for the ¼” audio input.	V	-2.5	+2.5	
XLR3 Audio Input Voltage Range	Voltage range for the XLR3 audio input.	V	-2.5	+2.5	

High-pitch Sample Rate	Sample rate for mid range to treble range voices.	kHz	2.637		
Low-pitch Sample Rate	Sample rate for low range to bass range voices.	Hz	390		
Input Impedance	A high input impedance is typical for musical devices.	kΩ	5		
Note Accuracy	The number of correct notes detected divided by the total number of notes detected.		0.8		1
Note Minimum	The lowest frequency that the device will be able to detect.	Hz	82.41		27.5

Table 2-1: Table of requirements.

We decided on 6 note polyphony so that the device can capture each voice in a standard, six-stringed guitar. We are not limited to just the six strings of a guitar, however. This six tone limit should also apply to how the device captures other instruments such as six musical tones of a piano or up to six singers singing into a microphone input. A number of the design choices we have made are based on this device being compatible with an electric guitar (6 note polyphony, quarter-inch bypass). A device such as the L.R. Baggs Para Acoustic D.I. is an interface device for use with acoustic instruments that can be used on its own or on a pedal board. Its dimensions are approximately 6” by 3.5” by 1.5”. These kinds of dimensions would be ideal, but we are allowing a slightly larger form factor for our minimum acceptable prototype. Delay tends to become noticeable at around 30ms [citation needed] , so we should aim to keep latency below that, ideally at less than 10ms, however for our minimum acceptable prototype we are aiming for a latency under 100ms as a proof of concept. Our goal is to convert six notes simultaneously so that this device will be fully compatible with a standard six string guitar. We want MIDI and USB output for compatibility with the widest range of devices. Simply having a MIDI output jack alone would be enough to use the device with other MIDI devices, such as digital synthesizers and audio interfaces, but including the USB port removes the need for an audio interface and enables connectivity to a computer. The device will be able to act as a USB compatible MIDI controller. We want the dimensions of the device to be small and the weight to be light so that it can be portable and useful for live performance applications. If we exceed the specified dimensions, we risk the box being too big to gig or inconvenient to place on a desk for studio use. We need our note recognition time and note translation time to be fast in order to reduce latency and make the device usable in a live performance setting. Even in a studio setting, it would be inconvenient to work with a device with too much latency. For both applications, low latency is preferable.

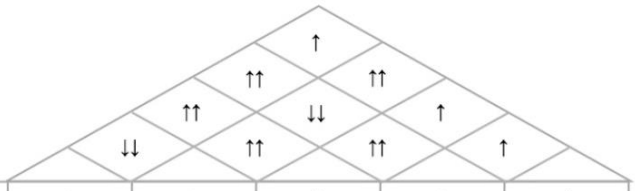
We need our components to not generate much heat so that we do not need to implement a fan which would add unwanted audible noise. This also applies to both live performance and studio recording settings. A fan would generate noise that the audience might hear at a live performance and that a microphone might pick up in a recording studio. We need to regulate the voltage of both the quarter-inch and XLR3 inputs to make both of them usable. Our implementation of our input jacks must comply with accepted standards for transmitting audio signals through them. We need our sample frequency to be sufficiently

high to capture the highest relevant frequencies of the input signal. Too low of a sampling frequency (below Nyquist rate), and we will not capture the signal correctly, potentially ending up with aliases that could cause problems in our analysis. Too high of a sampling frequency and we risk slowing down the DFT and introducing unwanted latency. We need a high input impedance to use electric instruments without introducing “tone suck” to the sound. Tone suck is an informal term used by musicians and music technology companies to explain the attenuation of high and mid frequencies in their signal as it runs through a long cable or other hardware. In this case, the tone suck can be caused by the loading effect which would diminish our signal by demanding too much power from it. [1] Finally, we need to keep the device under budget. It would be ideal to not spend more money than we need to on our project. That would be bad for the team because it would mean more money out of our pocket and it would be bad for any potential consumer because it would mean a higher price tag on the final product.

2.4. House of Quality Analysis

In the case of the Polyphonic Analog to MIDI Converter, many of the engineering requirements are directly related to the customer requirements. Both the engineers and the customers want low cost, low latency, and high accuracy. In addition to this, the engineers want high processing power and frequency bandwidth while the customer wants low power consumption and good portability.

Key:
↑↑ - Strong Positive Correlation
↑ - Positive Correlation
↓↓ - Strong Negative Correlation
↓ - Negative correlation
o - No correlation



Direction of Requirement	Engineering Requirements → / Customer Requirements - (What) ↓	Cost	Latency	Accuracy	Processing Power	Frequency Bandwidth
	Direction of improvement	↓	↓	↑	↑	↑
↓	Cost	↑↑	↓↓	↑↑	↑↑	↑
↓	Latency	↓↓	↑↑	↑↑	↓↓	↑↑
↑	Accuracy	↑↑	↑↑	↑↑	↑↑	↑
↑	Compatibility with MIDI devices	↑	↑	↓	↑	↑
↓	Power Usage	o	↓	↑↑	↑↑	↑
↑	Portability	↑	↑	↓	↓	↓
	Target	< $500	< 100 ms	Up to 6 notes at a time and up to 2 notes a second	25 MHz	82 Hz - 12.6KHz

Figure 2-1: House of quality

Development, manufacturing, and consumption all benefit from lower cost. The cheaper the parts, the easier it will be to acquire them for prototyping and testing and the cheaper it will be for the end customer. Unfortunately, almost every other requirement is at odds

with low cost, leaving power consumption as the only requirement that is independent of cost. Thankfully, cost is the simplest requirement to manage in the context of the senior design course; if something is not in the budget, simply increase the budget as we are not beholden to any sponsors or other constraints. We acknowledge this will not always be the case in projects moving forward, but for now we will take any advantage we can.

Possibly the most important relationship for the function of the device is that between latency and accuracy. They have a strong correlation with one another, as an increase in latency will allow room for an increase in accuracy and a decrease in latency will provide less time for an accurate algorithm. If latency was not an issue, we could implement the most accurate algorithm possible without worrying about how long it takes to execute, but because this device has to work in real-time for music performance many concessions may need to be made in the DFT, filtering, and note detection algorithms in order to manage latency.

One of the most important accessibility features of this device is its variety of inputs and outputs, including a MIDI out and a USB out. This is to make it compatible with as many other MIDI devices and platforms as possible though it may impact cost due to the additional hardware required, latency due to the additional step in the signal chain of the USB controller, and accuracy as a result of latency being affected. The additional hardware is likely to be a simple integrated circuit used to control the USB port, so it will not be very expensive or very time consuming to implement, so impact on cost should be minimal. Impact on latency should also be low as indicated by the single arrow of correlation in the house of quality table. With appropriate planning and part selection, MIDI compatibility should not be too difficult to achieve.

Low power consumption is an important feature for both usability and compatibility. We want to be able to build this device with no active cooling component. Adding a fan to the design can add audible noise which might be heard by those nearby in a performance setting or picked up by a room mic in a studio setting which is unacceptable in a music performance environment. The device will be able to accept a microphone input with 48V phantom power, meaning that a sensitive condenser microphone may be used as the input. If our device is creating audible noise that is being picked up by that microphone, then we have failed to deliver a useful product. Additionally, we are constrained on our power usage. In order to make it easy to power, we will try to implement two different powering options: DC 5V, 0.5A USB power and DC 9V power. USB is standard for use in a production setting where you will have a computer nearby and 9V will be useful in situations where you have the device on a pedal board with a typical 9V power supply and are cabling to a computer or other MIDI device that is not nearby. 9V DC power is a ubiquitous power supply voltage for music devices with current output ranging from 100mA to 1500mA. A 9V, 300mA power supply will supply an amount of power that is roughly equivalent to what the USB port will be able to supply, falling safely within the range of acceptable values for our target demographic. We will have to consider in our power budget the many parts used in our design and whether the amount of power they consume is worth the reduction in latency or increase in accuracy, processing power, or bandwidth that they provide.

Like power usage and cost, the portability requirement may negatively affect all our engineering requirements. If we find ourselves running low on space, we may need to design our PCB with more layers for more traces which costs more money. We may need

to use suboptimal parts if they take up less space on our PCB, which could impact our latency, processing power, accuracy, and bandwidth. We want a powerful microcontroller or microprocessor to carry out our fast Fourier transform and filtering algorithms to make them fast and consistent. More powerful parts may come at a higher cost, consume more power, and may even be physically larger on the PCB than other options but these will likely be the most impactful parts in the design on latency and accuracy so they should be prioritized over other parts affecting the cost, power, and portability constraints.

Finally, the frequency bandwidth requirement regards the bandwidth of the input to the device. Ideally, a device like this could accept input signals with frequencies across the range of human hearing. This is, however, not realistic for a project of this scale, so we are aiming for a bandwidth that covers the range of notes that is playable by a standard, six-string guitar: 82 Hz to 1.32 kHz. This range will cover many other instruments as well, encompassing most notes on the piano, most notes that are sung, etc. The most obvious constraints on input bandwidth is the ADC and the DFT. The ADC must have a sample rate of at least the Nyquist rate of the input (twice the highest frequency) in order to accurately capture it in digital form. However, the higher the sample rate, the more time-consuming it will be to perform the fast Fourier transform and filtering processes. We will have to find a sweet spot for the sample rate where it is high enough to capture what we need but low enough to not be too taxing on the microprocessor / microcontroller. We will also have to design clever fast Fourier transform and filtering algorithms that will allow us to take advantage of a high sample rate without compromising our real-time capabilities.

2.5. Device Overview

In summary, our final design will accept an analog input from ¼" or XLR3 input (one or the other, toggleable) which will be buffered, amplified, and filtered by our preamp stage. This prepared signal is then sent to the processor where it is converted to a digital signal and an FFT is performed to analyze the frequencies of the signal. This data is processed to find the root frequencies of the notes being played and up to six frequencies that cross the threshold for what the device recognizes to be a note. Finally, this signal is converted to MIDI and delivered to an external MIDI device or computer via a USB or MIDI connection.

Additional features include a potentiometer on the faceplate of the device to adjust input gain, allowing the user to control the level and prevent clipping of the signal. The anti-aliasing bandpass filter will be made adjustable via toggle switch for treble frequency, bass frequency, or full range to improve performance for specific instruments or voices. The analog signal output can also be toggled between a buffered output or an output that has been affected by the preamp stage. The device can be powered by both external 9V DC power supply and by 5V USB power. A bridge rectifier to protect against and even operate with reverse voltage from either power supply.

The prototype demonstrated in the Showcase does not feature the XLR3 input and output, the USB connectivity (including 5V from USB), and 9V power. 5V and 3.3V power are sourced from external jumper cables instead.

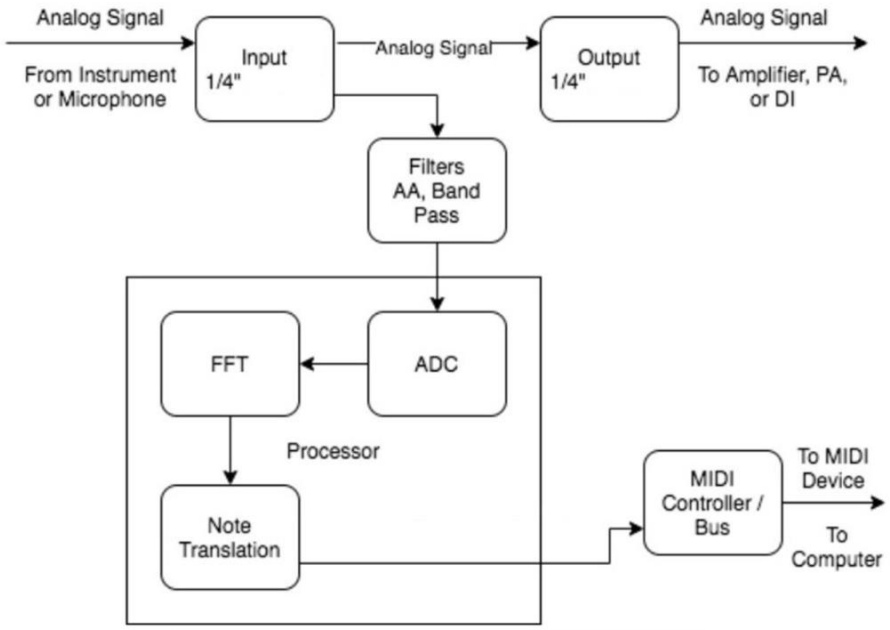


Figure 2-2: Overview block diagram.

3. Research Related to Project Definition

3.1. Existing Similar Projects and Products

3.1.1. Sonuus G2M V3

There are two current products made by Sonuus that deal with audio-to-MIDI conversion. These products are the G2M V3 and the i2M. Both products are very similar in their features but differ slightly in their implementations and hardware designs. The Sonuus G2M V3 is the third installment in the line of G2M models created by Sonuus. This device is a portable plug-and-play device that can connect to any MIDI device or computer. The G2M V3 is a major improvement from the previous versions of itself because it can work with any electric guitar along with bass, voice and wind instruments. The previous versions of this product only worked specifically for an electric guitar and there was a separate product (B2M) that worked specifically for a bass guitar. The device uses a single AA battery for power and has an optional 9V DC power adapter that can be used for pedal board integration. The device has a 6.35mm mono jack input to connect a guitar, bass or microphone and a MIDI output through a standard 5 pin MIDI socket. Some features of the G2M V3 include accurate pitch-bend determination, very low latency, built in precision tuner to guitar or bass tuning, and long battery life.

A limitation of the G2M V3 is that it can only translate monophonic notes. We would like to make a product that is like the G2M V3 but with polyphonic capabilities and some other changes in the hardware. We want our device to be powered through USB rather than by battery like the G2M. The polyphonic capabilities of our device would allow instruments like an electric guitar or an acoustic piano to play chords and harmonies that will be

converted into MIDI. We will have to develop an efficient algorithm for decomposing the instrument signal into individual notes in order to keep the latency low enough for use in a performance or production setting.

3.1.2. i2M musicport

The i2M musicport is a similar product created by Sonuus that has many of the same features as the G2M but with some changes in implementation and hardware. Like the G2M, the i2M musicport is a plug-and-play device that takes musical audio from guitar, bass, voice or wind instrument and converts it to MIDI. The main difference of the i2M musicport from the G2M is that it uses a USB interface for power and MIDI output. Since the i2M musicport is USB-powered instead of battery powered, it is much smaller than the G2M making it more lightweight and portable. The i2M musicport has a high impedance USB audio interface that prevents the degradation of the tone of the musical instruments being played which is a problem that can arise from using line inputs.

Like the G2M V3, the i2M musicport only supports monophonic MIDI conversion. We would like to make a device that is closer to the i2M musicport rather than the G2M. The device will be more like the i2M musicport because we would like our product to be USB powered as well but also having the option to be powered by a 9V DC barrel jack. The main difference between our device and the i2M musicport is that we would like our device to support polyphonic MIDI conversion. Other differences are that we want our device to have both an output for both USB and MIDI cable so that our device can work with as many MIDI devices as possible.

3.1.3. Beat Bars: A2M Converter

The A2M converter is a free software developed by Beat Bars that is used for real time Audio to MIDI conversion. This software can convert guitar real time audio to MIDI through a line input or other types of analog musical instruments through a microphone. This application works with any instrument that receives MIDI. This software is currently in beta so it is still being worked on and improved. Since this software is free it is a good option for people who may not be able to afford plug-in products such as the Sonuus devices or other costly Audio to MIDI converter softwares. This application is not a plug-in such as the Sonuus devices or the device that we are creating. One limitation of the A2M converter software includes the lack of polyphonic note detection. The device that we are developing should improve on this and should be able to recognize polyphony or multiple notes played at the same time and translate it to MIDI. Other known limitations of the A2M converter include lack of pitch bending feature and that only the first default audio-in channel of the audio interface is supported. Pitch bending could be a stretch goal for our design but is not currently one of the requirements for our device.

3.1.4. IntelliScore Polyphonic Audio to MIDI Converter

The IntelliScore Polyphonic Audio to MIDI converter is a software created by Innovative Music Systems Inc. The IntelliScore music recognition software can take prerecorded polyphonic music that is in WAV or MP3 format and convert it to a MIDI file that contains the notes played, chord names and overall key. There are many features that come along with the software such as multiple instrument recognition including drums, removal of

vocals from music files, and music transcription. The software makes it possible for the user to take polyphonic music from an MP3 and use it in MIDI. This software is one of the only polyphonic audio to MIDI converters that can be found when looking online. The IntelliScore Polyphonic Audio to MIDI converter can be purchased for about 100$.

This software has one main feature that we would also like to implement in our hardware device. This feature is the polyphonic audio to MIDI conversion. The main limitation of this product is that it is only able to convert prerecorded music audio to MIDI. This makes the product useless for live music transcriptions to MIDI. If someone wants to use the Intelliscore product to transcribe music that they are playing they would have to record the music and convert it to an MP3 or WAV file and then use the product to convert it to MIDI. We want our device to be able to convert live music into a MIDI file or stream which would cut out the middle of recording the music and converting to MP3 which would take time. With live music audio to MIDI conversion our device would be usable during a live performance which would be impossible using the IntelliScore product. Another difference between this product and our device is that our device will be a physical hardware plug-in device, while the IntelliScore product is software that must be downloaded on a computer which means it has very limited portability.

3.2. Relevant Technologies

3.2.1. Analog Signal Filtering

Filters for AC analog signals attenuate different frequency ranges found in the signal. In the input section of this device, we want to attenuate unnecessary high and low frequencies before converting the signal from analog to digital. This will help to clean up the signal and make it easier to process digitally. To filter just the high and low frequencies we will need some variation of a bandpass filter. There are active and passive filters, active having power consumption and typically an amplifier, while passive requires no power and can only have a voltage gain of less than 1. For this application we will investigate active filters as that will provide more versatility with the types of filters we can choose from as well as keep our signal voltage gain from dropping. Since we are processing audio, it is most ideal to try and maintain the most accurate phase and frequency response outside of the attenuated high and low frequencies. To do this we would prefer a sharp cutoff as well as a low Q factor to keep the frequency response as flat as possible. Another factor to keep in consideration is the power consumption since the filter will be active, although this will likely be negligible for a low voltage microphone or instrument input. Figure 3-4 shows different active filter types as described by a tutorial on globalspec.com.

Filter Characteristics

- **Bessel filter.** Bessel filters are active filters with a passband that maximizes the group delay at zero frequency, thus showing a constant group delay in the passband. Group delay is a measurement of the time it takes for a signal to move between two points in a network. A constant group delay in a filter's passband implies that for all signals with frequencies in the passband, the time delay will be the same. This fact is especially important in audio, video, and radar applications.
- **Butterworth filter.** Butterworth filters are designed so that the frequency response is flat in the passband.
- **Chebyshev filter.** Chebyshev filters feature a very steep roll-off, but have ripples in the passband.
- **Elliptic filter.** Elliptic filters (Cauer filters) exhibit equalized ripple in both the passband and the stopband.
- **Gaussian filter.** Gaussian filters produce no overshoot in response to an input step. They optimize the rise and fall times.
- **Legendre filter.** Legendre filters are designed to produce the maximum roll-off rate for a given order and a flat frequency response in the passband.

Figure 3-4: List of filter characteristics.[2]

While the Bessel filter has a more linear phase response, it has a very gentle cutoff which will not filter out as many frequencies as we would like. On the other hand, Butterworth filters have a very steep cutoff and a flat response in the frequencies that aren't attenuated. In our application it seems most ideal to give up a bit of the phase response and in turn have a better frequency response. This is because we are analyzing the frequencies using a relatively large set of samples with Fourier transform and so the phase should not be very important. Because of this, it would be best to use a variation of a Butterworth bandpass filter. Supported MIDI frequencies cut off at ~12,500Hz so our upper cutoff frequency should be about 15,000Hz to give a little room for roll off before the cutoff as well as a lower cutoff frequency of about 20Hz so that lower bass frequencies of 30-40Hz can be detected.

3.2.2. Analog Amplification & Mixing

We need amplification before our signal goes into the analog to digital converter to get the signal within the input specifications of the device. Amplification can also provide buffering and filtering applications that will be useful. For our purposes, the most practical way to amplify our signals is to use operational amplifiers. Operational amplifiers typically are a differential amplifier with feedback which will make it versatile and easy to configure for different purposes. Operational amplifiers also can come in IC packages with many amplifiers in one chip. This will be ideal as it will allow us to perform many functions with one IC in a relatively small surface area on our PCB.

With several op amp-based circuits, we can do filtering, amplification, buffering, and signal splitting [4]. Op amps are easily configured to form as a signal buffer by simply connecting the output to the negative terminal of the op amp input. This causes the voltage differential to be 0 so no amplification occurs. This allows us to isolate the input and output, so the load doesn't affect the input and gives us more ideal impedances to our circuit. This should be very useful for splitting and mixing signals. The implementation of this signal splitting is called a distribution amplifier. This means it recreates copies of the original input signals in the outputs [3]. Op amps also will be useful for our active analog filters. Even though our filters will only need to cut out frequencies and not amplify, the op amps will allow us to filter without hurting the circuit impedance by adding resistors and capacitors for filtering. Most importantly, op amps can function as our form of amplification of course. In its most simple form, an op amp can be configured to amplify by adding a voltage drop over the feedback loop which causes there to be a voltage differential in the inputs. The op amp by design then amplifies based on that voltage differential. This makes it very easy to control exactly how much the op amp amplifies the signal as well as what frequencies it amplifies. Using this, we may even be able to both amplify and perform our filtering in the same op amp circuits, although it may be better to isolate them.

Another use for our op amps would be as a comparator. We want to make sure we don't clip the input of our analog to digital converter by going past it's input voltage range. To prevent this, we can set a threshold in the analog input that if the input passes, it can shut off or reduce the input and then turn on an LED to notify the user that their input is clipping. This will be useful because it will protect the ADC as well as notify the user so they can adjust the gain on the preamp until it no longer clips. A comparator can be made in a few ways with an op amp. One way is to have the differential amplifier set up with an open loop, always making the gain maximum. This makes it so that when the input is greater

than the reference voltage, the output will be the difference between the input and reference multiplied “infinitely” until it hits Vcc+. For us this is 5 volts which will be perfect for powering an LED. Then when the input is below the set threshold, the output will multiply the same way as before but it will be negative so it will swing down to the low rail which we will be using ground for, making the LED not have any power. Simplified, the comparator basically acts as a switch allowing us to see if the voltage is above the clipping threshold we set, and then turns on an LED as a result.

3.2.3. Analog-to-Digital Conversion and Sampling

Analog to Digital conversion is the process of changing an analog signal into a digital signal. For our device we will be converting analog sound signals to digital signals that can be converted to MIDI. This process is usually done by with either a dedicated Analog to Digital conversion (ADC) chip but this process can be done with an integrated ADC one within a microcontroller or digital signal processor. A typical ADC samples an analog signal on either the falling or rising edge of a sample clock [5]. During every cycle, the ADC measures the analog signal voltage and converts it to an approximate digital value.

There are two main factors that contribute to the accuracy of the analog to digital conversion which are bit rate and sampling rate. Increasing the bit rate of an ADC improves the precision of the approximations in its digital output [5]. The sample rate is important due to the Nyquist sampling theorem. The Nyquist sampling theorem states that the sampling rate of the ADC must be at least twice as fast as the highest frequency component of the waveform being sampled. If the sampling rate is not at least equal to twice the highest frequency component, there can be inaccuracies in the sampling due to aliases. The minimum sampling rate that we will need to use for the analog to digital conversion for our device will depend on the frequency range of possible musical instruments that will utilize our device and the maximum musical frequency. The maximum note frequency used in MIDI is 12543.854 Hz. We want our device to be able to be used with multiple musical instruments including guitar, piano, voice and more. These instruments have varying frequency ranges. Table 3-1 shows the frequency ranges of some musical instruments and MIDI notes.

Musical Instrument	Minimum Frequency (Hz)	Maximum Frequency (Hz)
Guitar	82.41	1318.51
Piano	27.5	4186
Bass	41	262
Human voice	87	1047
Flute	262	1976
Trumpet	165	988
Clarinet	165	1568
MIDI	8.1758	12543.854

Table 3-1: List of frequency ranges for various instruments.

All of the common musical instruments listed in the chart have frequencies that are below the maximum frequency of the MIDI note range. This means that the ADC part of our device will need to be able to sample at 25087.708 Hz, which is double the max MIDI note frequency, to support all possible analog instruments; if there are musical instruments that play notes at a higher frequency than the highest MIDI note frequency it would be irrelevant for the uses of our device since MIDI would not have a note available to play in that frequency. For this project, we are focused on making this device work best with a guitar, so the desired sample rate will be at least 2.637 kHz.

3.2.4. Processors

Processors are electronic circuits that manipulate data from various sources. They are typically used to control other circuits or to perform calculations using binary data. There are many kinds of processors available, and they can be grouped into categories depending on their purpose and functionality. There are microprocessors, which are mostly focused on manipulating binary data and controlling external circuits and devices, and they are typically high-performance integrated circuits (ICs) with high power requirements. There are systems-on-chip (SoCs) and microcontrollers (MCUs), which are low-power alternatives to a microprocessor with more functionality and other systems included in the chip, like random-access memory (RAM) or a liquid-crystal display driver. There are also processors dedicated to specific tasks like graphics processors, digital signal processors (DSPs), and ASICs. There is also the field-programmable gate array (FPGA), which allows its physical circuitry to be programmed.

What we require for our device is a processor that is capable of performing a fast Fourier transform (FFT) and other digital filtering schemes within a limited amount of time, includes the peripherals we need within the chip, is easy to program and test, and consumes low power. The processor we choose needs to have internal RAM and writable program memory, an ADC, and a USB interface or support for an external one. The processor is also responsible for sending the MIDI stream serially through a USB or MIDI port. The MIDI standard for the physical interface requires a bit rate of 31.25 kHz, so the microcontroller must be capable of this bit rate while also performing other tasks. Since the MIDI physical and application layers are simple, a MIDI output can easily be implemented with just the microcontroller and some basic active and passive components. Since USB is a more complex standard, we would need either a dedicated IC that the processor supports that interfaces with USB or a processor that is capable of interfacing with USB directly.

From our experience with the MSP430 MCU, we know that MCUs can come with ADCs, RAM, program memory, and a serial communications system on-chip, so we will consider MCUs as a category of processors to select from. We will also consider using a DSP since it is specialized for the kind of work the device must do, and some DSPs have the peripherals we need included. We will not use a microprocessor since it would use too much power, and we will not use an FPGA since they are relatively expensive and require much more low-level programming. We will also not consider using an SoC because they typically have much more peripherals than we need, consume a medium amount of power, and are relatively more expensive than a microcontroller.

3.2.5. Digital Signal Processing

Digital signal processing is the act of performing operations on discretely quantized signals. This means that rather than a continuous waveform, the signals are turned into discrete voltage levels and processed using discrete logic rather than continuous such as calculus. Having a discrete signal allows you to represent it in binary which allows you to interpret signals with traditional computer architectures. These operations can be anything from interpreting the signal and encoding it into a file or data type or using it as input for machines or programs. The signals are often filtered or modified in some way and then output from there. Our applications will process the signal in multiple ways. These include filtering, FFT to analyze the signal in the frequency domain, amplitude filtering, selective frequency filtering etc.

The Fourier transformation is a function that converts a time domain signal to the frequency domain and there is a discrete version of it using summation rather than integration [6]. This can be implemented in many ways algorithmically, but we will focus on a fast Fourier transform algorithm to efficiently perform the transformation in real time. Filtering can be done directly by realizing a transfer function in the Z domain using delays and other functions. Signals can also be modified more simply in the frequency domain and then an inverse DFT can be performed to turn it back into a signal. In our case we are converting the signal simply to a MIDI signal so we can work with the signal in the frequency domain and change frequencies and magnitudes as needed in an array data structure.

Some important characteristics with digital signals are bit depth and sample rate. Sample rate refers to the rate at which discrete samples are taken from the analog signal [7]. The faster the sample rate the more accurate the interpreted signal, given a large enough bit depth. According to Nyquist sampling theorem, the sample rate must be at least twice as fast as a frequency to be able to mathematically recognize or interpret the digital waveform. This means that for us to recognize the highest frequency in MIDI protocol of about 12.5 kHz we need a sample rate of at least 25kHz. This ensures that we can actually recognize any frequencies up to the range MIDI protocol can represent. Bit depth on the other hand is the amount of voltage level quantization's [7]. For instance, if there was only a bit depth of 1, there could only be 2 volume levels, on and off. A higher bit depth means you have a larger dynamic range of the frequencies between soft and loud. When listening to audio it is important to have a good bit depth so that all the frequencies are represented accurately and reproduce the intended sounds with all their original complexity. Most listening applications try to maintain at least a bit depth of 16 which gives 2^{16} or 65,536 different volume levels for every frequency. In our case, we only need enough volume levels to be able to represent any notes in MIDI and possibly slightly more for filtering out low volume noise. MIDI has 128 different magnitude levels ranging from 0-127. This means that we should have a bit depth of at least 8 bytes per sample to be able to accurately represent the magnitude of notes in MIDI. Figures 3-5 and 3-6 show the effects of bit depth and sample rate on the digital representation of an analog signal.

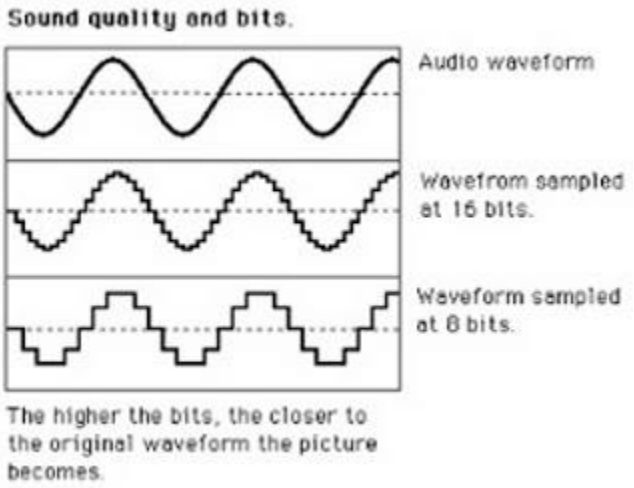


Figure 3-5: The effect of bit depth on sampling an analog signal.

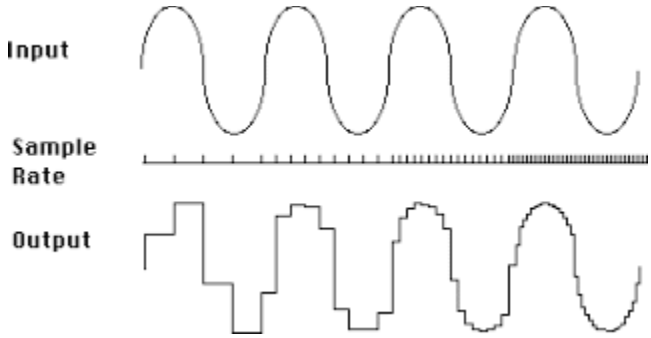


Figure 3-6: The effect of increasing sample rate on sampling an analog signal.

3.2.6. Pitch Detection

In music, monophony is a melody that is played by a single instrument or voice that is unaccompanied by any other chords or harmonies. An example of this is someone humming a tune with no other instrument being played. Polyphony is a melody that uses multiple voices or sounds. An example of this would be a choir singing multiple pitches at the same time. To achieve pitch detection, the fundamental frequency of the input must be found. This is what pitch detection algorithms are used for. Pitch detection algorithms can be done by using either time-domain or frequency domain techniques.

Time-Domain Methods

Analyzing the pitch of an input can be done by using time-domain algorithms. This method is done by changing the amplitude of the input in the time-domain and looking for repeating patterns in the waveforms to figure out its periodicity to find the fundamental frequency. The simplest of time-domain methods is the zero crossings approach. This method is done by counting the amount of times that a signal crosses the zero-level reference. Using the zero crossings method is easy and inexpensive but does not have great accuracy [8]. Noisy signals cause this method to have inaccurate results which means that it would not be good to use for polyphonic signals. Another time-domain method is the method of using

autocorrelation. This method tries to find similarities between the signal and a shifted version of the signal. Autocorrelation is defined by the equation in Figure 3-7.

$$y(n) = \sum_{k=1}^{M} u(k)u(k+n)$$

Figure 3-7: Autocorrelation definition.

This technique has a limited pitch range and works best at low to mid frequencies. This method can become very expensive due to the use of numerous add-multiply operations. Maximum likelihood is another method that can be used in the time-domain for pitch detection. Maximum likelihood is done by taking a signal and breaking it up into N segments that are of length τ. These segments are then added together. The segments add coherently when τ equals τ_0. Doing this gives the function shown in Figure 3-8. [8]

$$J(\tau) = (N+1)\sum_{t=0}^{b} r^2 + N\sum_{t=b+1}^{\tau} r^2(t,\tau)$$

Figure 3-8: Maximum likelihood function.

Maximizing this function gives τ_0 which can be used to find the fundamental frequency for pitch detection.[8]. The time domain methods shown can be successful in giving accurate pitch detection but for the purposes of our device we will need to use a method that utilizes the frequency domain. This is because the methods using the time domain will be too slow and expensive for making a device that can detect pitches in a live setting or polyphonic signal in general

Frequency Domain Methods

Methods using the frequency domain are more complex than the time domain but in general are more accurate and faster. All the methods that use frequency domain analysis to detect pitch use some version of the discrete Fourier transform (DFT). The DFT is a discrete version of the Fourier transform of a continuous time signal that uses a finite sum instead of an infinite integral which would be used in a continuous time Fourier transform. The definition of the DFT is shown in Figure 3-9.

$$X(\omega_k) \triangleq \sum_{n=0}^{N-1} x(t_n)e^{-j\omega_k t_n}, \qquad k = 0, 1, 2, \ldots, N-1,$$

Figure 3-9: Discrete Fourier transform.[9]

This equation requires no calculus in its computation unlike the continuous Fourier transform that uses an infinite integral. DFT is needed for digital signal processing because the spectra that are analyzed in digital signal processing are sampled which makes them discrete rather than continuous. For the purposes of pitch detection, the best

version of the DFT to use is the Fast Fourier Transform (FFT). The FFT is an efficient implementation of the Discrete Fourier Transform that will be used for the pitch detection algorithm for the device.

3.2.7. Fast Fourier Transform

A fast Fourier transform (FFT) is an algorithm that calculates the discrete Fourier transform (DFT) more quickly and efficiently. As mentioned before, the DFT transforms temporal data into frequency data in a discrete manner. This makes it easy to analyze the frequency makeup of the signal. The DFT can be calculated as it is defined, but its run-time would be $O(n^2)$ because of the summation of N terms and the use of this algorithm to calculate X for all k from 0 to N - 1. All known FFT algorithms have a run-time of O(nlogn), which is a considerable improvement.[10] The FFT algorithm we plan to implement is called the Cooley-Tukey FFT algorithm. It relies on N being a power of two and uses this property to reduce the number of complex additions and multiplications.[11] By expanding the sum, grouping terms with even or odd multiples of n in the exponent, factoring out like terms from each group, and repeating, the DFT will look something like Figure 3-10.

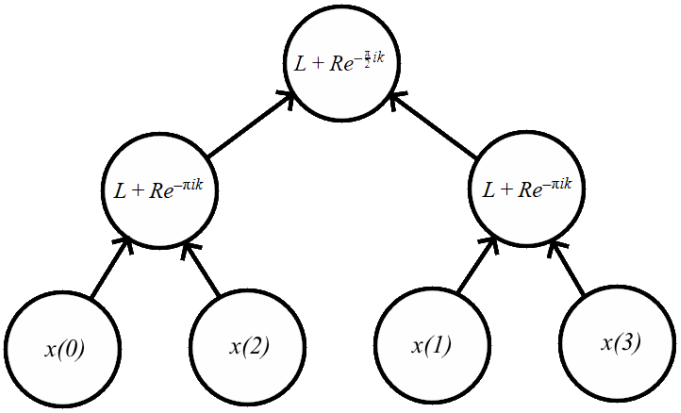


Figure 3-10: An example binary tree representing X(k) for N = 4.

This is a binary tree with $\log_2$ N levels, where, at each node, the node's left child is added to the product of the node's right child and a coefficient, where s is the current depth within the tree starting from 0 at the bottom, denoting the stage of the calculation. At the deepest level, the nodes are equal to values of x(n), where n corresponds to the directions taken to reach that node. A left turn represents a 0, and a right turn represents a 1, starting from the root node. The order is interpreted such that the last turn represents the most significant bit (ex. left-right-right = 110_2 = 6). This is also known as bit-reversed ordering, because the new order reflects the previous one, but with the bits of the indices reversed. After doing this, the run-time would still be $O(n^2)$ because, for any N, the total number of complex additions and multiplications has only decreased by 1. The number of operations can further be reduced by exploiting the periodic nature of the coefficient.[12] This coefficient is shown in Figure 3-11.

$$e^{-j\pi\frac{m}{2^s}}$$

Figure 3-11: The coefficient used in the FFT.

In the case of N = 4, the coefficient in the second level (s = 1) can take on two unique values: 1 when k is even, and -1 when k is odd, for any k. The coefficient in the first level (h = 0) can take on four unique values: 1, i, -1, and -i, for any k. Essentially, for each stage, each node in the level needs to be calculated only 2^s times to get X(k) for all k.[9] Figure 3-12 shows a graph of the entire FFT for N = 4.

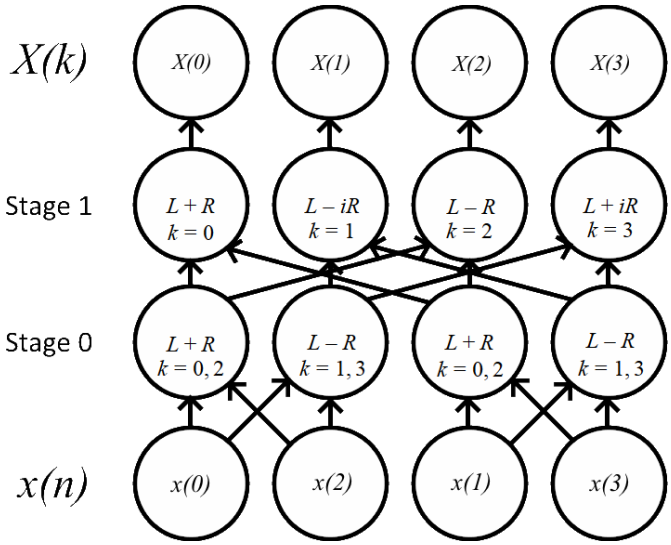


Figure 3-12: Example FFT graph for N = 4.

This is how the Cooley-Tukey FFT algorithm works. The reordering of the input data simplifies the implementation and allows the entire calculation to be split up into blocks of "butterfly" operations.[12] Figure 3-13 shows a visual representation of a butterfly operation. A butterfly operation takes a left input and a right input.

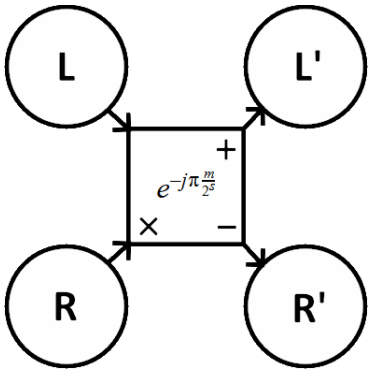


Figure 3-13: A visual representation of a butterfly operation.

It first multiplies the right input by the coefficient, then puts the sum of both inputs in the left output and the difference between both inputs in the right output. Each stage has $N / 2^{s+1}$ sets of 2^s unique butterfly operations, where uniqueness refers to the coefficient used in the operation.[12] It is important to note that the number of stages is equal to $\log_2 N$. Figure 3-14 shows a graph of the general form for an FFT.

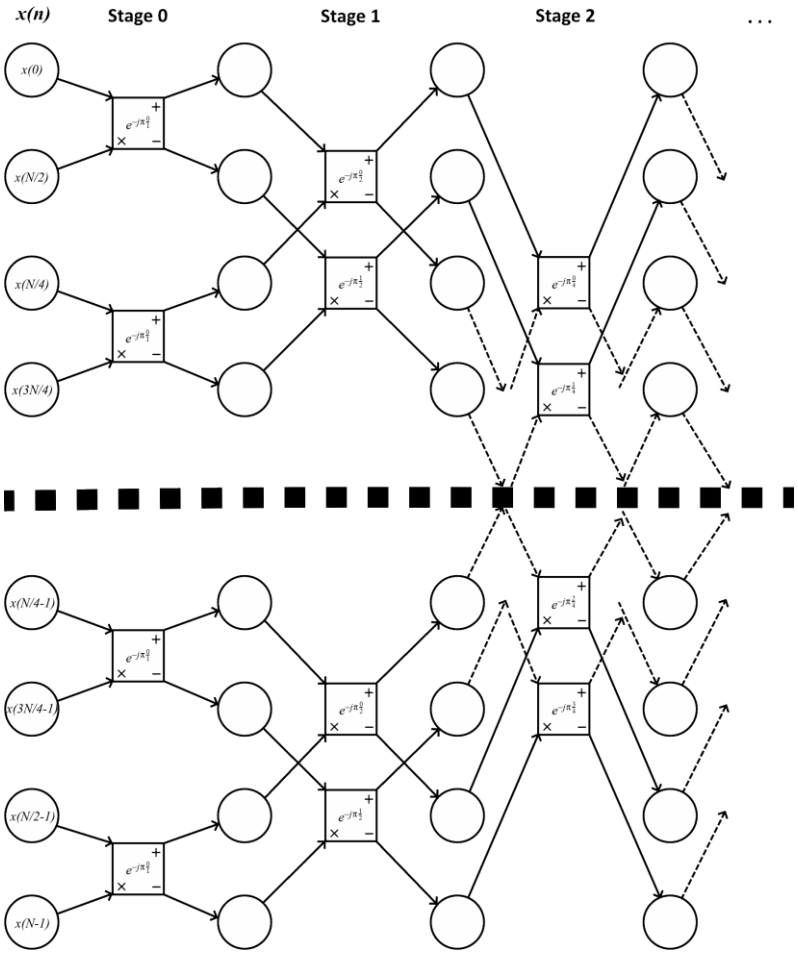


Figure 3-14: A visual representation of an FFT.

There is another optimization that can double the speed of our FFT implementation. Since our discrete signal only has real values, the summation can be split into two summations, one for cosine and another for sine.[13] This eliminates all complex math since the real and imaginary parts can be calculated separately without needing each other. To implement this optimization, each complex coefficient within every butterfly operation is replaced by a sine or cosine of the argument of the complex coefficient, depending on which summation is being processed.

3.2.8. Enclosure, Shielding, and signal Interference

Signal interference proves to be much more of a problem in analog circuits, especially audio, as compared to digital circuits. If noise causes a voltage level in a digital circuit to fluctuate, there are only 2 voltage levels so the interference would have to be very large to read a 0 as a 1 or vice versa. When processing analog signals and audio, noise can be picked up and amplified or distort a signal. In some audio applications like speaker systems or studio recording, noise can be a large issue. In our case, noise will not be nearly as important to avoid since most noise is in high frequency ranges beyond that we will be detecting, but a high frequency can be interpreted as a lower frequency harmonic by the analog to digital converter if the voltage is high enough. There can also be lower

frequency noise that could potentially be detected as a note or mess up the detection of notes that we need to avoid.

Noise and interference can come from many sources. Some of these include Electromagnetic waves like radio waves, digital signal lines like clocks, improper grounding which can cause ground loops, and power supply ripple [14]. Each of these can typically cause different types or amounts of interference but precautions should be taken to minimize all these sources of noise. Outside interference from EM waves can be minimized greatly through EMI-RFI shielding [15]. This simply involves physically encasing the device with an electrically conductive material to reflect and absorb EMI. For this to be fully effective, the conductive material must be grounded so that it can provide a path for the interference to go. Without this it could build a voltage on the conductive material and cause interference. We must ensure that the shielding has a very low impedance path to ground so no voltage can build up. This can be ensured by avoiding connectors with any measurable impedance and sticking with a direct connection to ground with low impedance such as a solder joint. The simplest ways to protect the device with shielding would be to use a conductive enclosure or to use a non-conductive enclosure with a conductive lining or exterior layer. This could be achieved with an aluminum enclosure or plastic enclosure lined with copper coated shielding tape for instance.

Other interference that can occur include capacitive and inductive coupling. Capacitive coupling is when there are fluctuations in the voltage on a line and it generates an electric field that can affect other lines. This can be largely avoided by shielding on the lines that are most likely to cause issues. Most of our lines will have low voltages and will not cause significant capacitive coupling but if we have any AC power input, it is best we isolate those from signal lines and shield them with conductive material to prevent them from causing interference. Inductive coupling occurs when there is a current in a line and the flow of charge generates a magnetic field. Fluctuations in the magnetic field from changes in the flow of charge can induce currents into other wires or lines. Magnetic fields can go through normal conductive shielding so other measures have to be taken to prevent inductive coupling. One of the simplest ways to reduce this is to separate any lines most likely to cause inductive coupling from other lines. This would also most likely be our power lines so we should attempt to isolate power from signal lines as much as possible in our design. Also, another way to reduce inductive coupling is to reduce loop area which can be done by twisting pairs of cables, like our power and ground. This reduces the magnetic field generated by the loop and as a result the inductive coupling.

Some noise can come from grounding issues. Ideally ground is a perfect reference voltage with no fluctuations but doesn't always present itself that way. If you ground from different paths to different points, there will always be different impedances and voltage levels along those paths, even if all the paths connect to ground. If the grounding scheme is made in a way that causes differences in voltages between ground nodes, there can be current between nodes on the ground line and noise in the circuit can occur. This is called a ground loop. One of the most common ways to attempt to reduce issues like this is a signal point grounding scheme. This is also sometimes called a star grounding scheme. This means that all paths to the ground go to a single physical point, forcing every line to end at nearly the same voltage [16]. The downside of this is that it can require a lot of extra wiring or pcb traces. Figure 3-15 shows two different grounding schemes.

Lastly, there can be interference between analog and digital lines. This occurs mostly through capacitive coupling as described above. This is a common issue with circuits that include both analog and digital signals such as ours. Analog signals are particularly prone to capacitive coupling from digital signals. This is because digital signals change voltage very fast with a high slope and that causes a stronger electric field to be generated. Our device will likely have many different digital signal lines throughout and only a few analog signal lines in the input section before digital conversion. This means we should attempt to isolate the analog signals from the digital signals as much as possible. One of the most common culprits of capacitive coupling from a digital signal is the clock signal. This can change very rapidly and cause a large amount of noise. One way to shield the analog signals in the PCB is to put a ground path or layer between the analog and digital paths for coupled or induced noise to travel through.

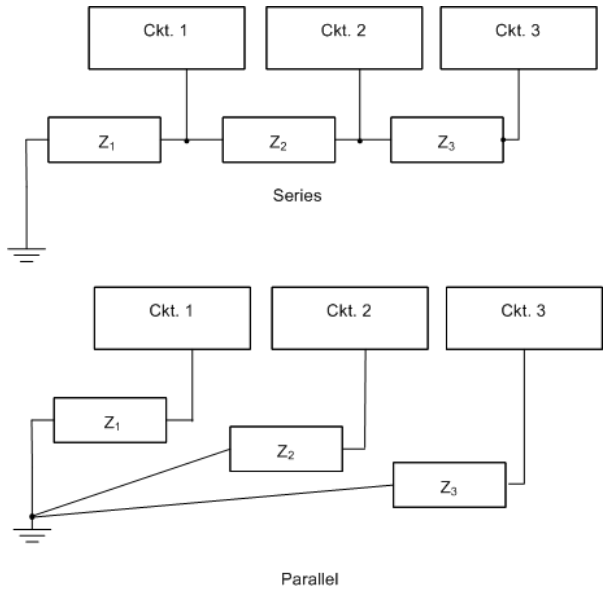


Figure 3-15: A diagram showing different grounding schemes.

3.2.9. DC-DC Conversion

To power our device, multiple DC-DC converters will need to be used. DC-DC conversion is a type of circuit that converts one DC voltage to a different one. Most electronic devices require DC-DC conversion because of different voltage requirements throughout their designs. There are three types of DC-DC converters which are Buck converters, Boost Converters and Boost-Buck Converters. Each of these DC-DC converters are a type of switch mode power supply circuit which is a circuit that uses semiconductor devices for switching methods to get a required output voltage rather than using a typical linear regulator. A Buck Converter or Step-Down Converter converts a DC voltage input to a lower DC voltage output. A Boost Converter or Step-up Converter converts a DC voltage input to a higher DC voltage output. The last type of DC-DC converter, the Boost-Buck converter is a circuit that can switch between being used as a boost or a buck converter. For our device we will need both boost and buck converters. A boost-buck converter will not be needed because there are no parts of our system that will need a voltage that varies.

Boost Converter

There is only one boost converter that will be needed for this device; it is a 5 Volts to 48 Volts DC-DC converter. This DC-DC converter is needed because we want our device to be able to supply 48 Volts of phantom power. This is a desirable feature of our device because 48 Volts phantom power is the most used power supply used for condenser microphones which would be used by users of our device to record voice or other instruments to transmit to MIDI. The 5 Volt input would be coming from either the USB which supplies 5 Volts regularly or from the 9 Volt DC barrel jack input that has been converted to 5 volts by the 9V to 5V DC-DC converter.

The boost converters that we use need to step up voltage while also stepping down current so that the circuit does not have a power output that is too high. Common configurations of boost converter circuits utilize different types of semiconductor devices such as diodes and transistors and at least one element is used for energy storage such as a capacitor or inductor [17]. Boost converter circuits usually work by switching the circuit on and off which changes the duty cycle of the circuit. The configuration of the circuit results in the steady state output voltage equation of Vout = 1/(1-D)*Vin, where D is the duty cycle which is the ratio of time that the circuit is on and the total time that the circuit is on or off [17]. This can be seen in the equation $D = t_{on}/(t_{on}+t_{off})$ [17]. This means that the output voltage can be regulated by controlling the duty cycle. Based on the equation the output voltage will always be greater than the input voltage which is what makes it a boost converter.

Buck Converter

We will need two buck converters for our device. One buck converter that will be needed for the device is a 9V to 5V DC-DC converter. This DC-DC converter is needed because we want our device to have the option to be powered by a 9V DC barrel jack. This is a desirable accessory for our device because a negative center DC barrel jack is a very common power standard used in music technology for performance so we want to provide this option of power supply to users of the device that want to use the device on stage. Since we also want to give the user the option to power the device through USB, we need to convert the 9 volts of input voltage from the DC barrel jack to 5 volts which is the voltage that would be coming from a computer if it was being powered by USB. The other buck converter that we will need is a 5V to 3.3V DC-DC converter. This DC-DC converter is needed to convert the 5 volts from the USB or 9 to 5 volt converter to 3.3 volts to power the microcontroller or processor we are using.

Buck converters use the same parts as boost converters but just configured in a different way to produce a different result. Like in the boost converter circuit, the buck converter circuit uses a semiconductor device like a BJT transistor or MOSFET as a switch and will usually also utilize diodes, capacitors and inductors [17]. The buck converter does not have the same problem of trying to keep the output current low since the output voltage should be lower than the input meaning that having a high power output should not be a problem. Like the boost converter circuits, buck converters usually work by switching the circuit on and off which changes the duty cycle of the circuit. The steady state output voltage of the buck converter is given by the equation, Vout = D*Vin, where D is the duty cycle [17]. This guarantees that the output voltage is less than the input voltage because the duty cycle must always be less than one.

Power Multiplexing

The device has two ways of supplying the main power needed for the device. These methods are either through 5V USB or 9V DC barrel jack which will be converted to 5 Volts through DC-DC conversion. There will need to be a logic circuit built to determine whether power will be supplied to the device from the 5V USB or 9V barrel jack. This part of the device will be needed to supply power to other parts of the device such as the MCU. Also, this part of the device will need to be able to supply 5 volts for the 5V to 48V DC-DC converter used for phantom power supply.

To make this logic circuit, we will need to make a type of switching circuit that can switch between which supply voltage is being used. When the 5V USB is plugged in, the 5V from the 9V barrel jack should be disconnected and when the 5V from the 9V barrel jack is connected the 5V USB should be disconnected. If both inputs are plugged in at the same time only the 5V USB part of the circuit should stay connected. We made this decision because using the 5V USB part of the circuit wastes less power than using the 9V barrel jack part of the circuit. There are multiple ways to implement this switching circuit. Two possible ways to build a switching circuit that will be explored are using discrete parts like diodes and MOSFETs or using an integrated circuit. Using discrete parts to build this switching circuit can become complicated and will increase the space needed for this part of the power design. Using an integrated switching circuit will make this process much simpler and will reduce the footprint of this part of the design. Using diodes for switching is very cheap and simple. An example of this type of circuit is shown in Figure 3-16. The drawback of using diodes to get this done is that there will be around a .6V drop in voltage from the diode which would lower the output voltage we want from this circuit. A Schottky diode can be used to reduce this voltage drop, but there would still be a voltage drop of around 300mV. We do not want this voltage drop so we will not be using diodes for the switching logic in our design.

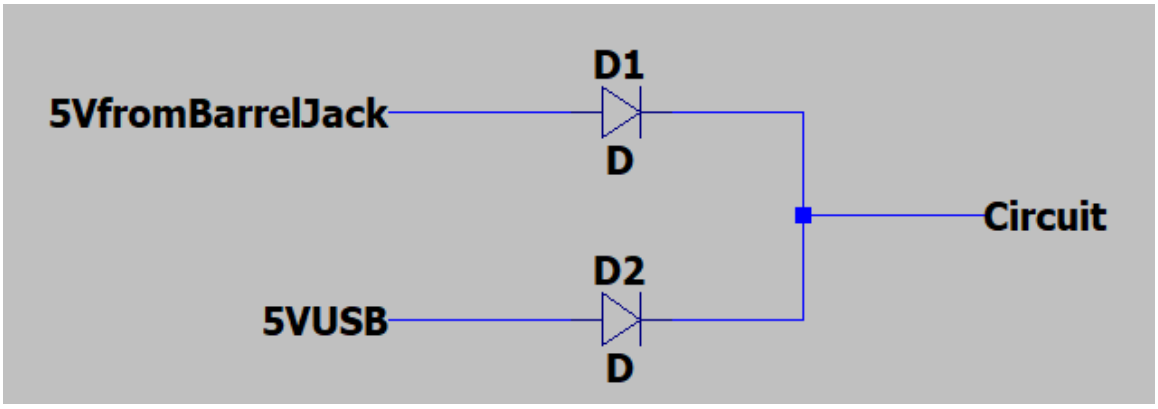


Figure 3-16. Simple diode switching circuit

Another option for the switching circuit would be to use MOSFETs. Using discrete MOSFETs for the switching circuits fix the dropout voltage problem but brings added complexity to the design. This would need a lot of space to design which could potentially increase our PCB cost. The best option for our switching circuit would be to use an IC. This would be more expensive than using Schottky diodes but would reduce the voltage drop significantly. The types of ICs that we would need to use for this application are called

ideal diodes or ORing MOSFET controllers. These ICs are designed to be able to control which input voltage is selected for use on a load and provide protection to power sources by minimizing reverse currents that go back into the supply. Using an ideal diode or Oring MOSFET controller instead of building a circuit with discrete MOSFET parts would significantly reduce the space used as well. When using these types of controllers there are two ways to control which voltage is chosen, highest voltage or highest priority. Since both input voltages we are using are the same at 5V we will look to make a circuit with one of these ICs that makes the highest priority the 9V jack voltage in the case that both inputs are plugged in.

It is possible for our device to be connected to a power source with the voltage reversed with ground connected to supply voltage and visa vice versa. This can happen if our negative pin barrel jack is connected to a 9V DC supply that is grounded on the shield or if the USB port of the device is connected to another USB device that is polarized in a nonstandard way. There are two ways in which we can protect our circuit from reverse voltage. The first protection technology is to use a single diode on each of our inputs to prevent current from flowing the wrong direction. It is a simple solution with the downside of the 0.7V drop across the diode. This cost is minimal as the 9V source already must be down converted to 5V and the 5V source can simply be amplified back up to 5V after the diode. Another downside is that the device will not work under reverse voltage conditions. Our second option does allow the device to work under reverse voltage conditions. It is the diode bridge rectifier. By putting a rectifier on each of our inputs, we can correct the polarity of our voltage source and continue operating. This comes at an increased cost since the voltage source sees two diodes in series with the rectifier, whether it is in reverse voltage or not, so there is a drop of approximately 1.3V. This, again, is a cost that can be compensated for by adjusting our 9V to 5V DC-DC converter to accept 7V to 8V and output 5V and using a noninverting amplifier to adjust our 5V source back up to 5V after the diodes. This option is better, as it allows the device to operate under reverse voltage conditions with similar drawbacks to the simple single protection diode. This scheme is illustrated in the LTSpice drawing in Figure 3-17.

In addition to reverse voltage, we must also be concerned with over voltage. It would be very easy for the user to accidentally plug in a 12V or 18V power supply into the 9Vin barrel jack of the device and it is also possible that the user may plug the USB port into a nonstandard USB power supply that runs on some voltage greater than 5V. Two Zener diodes can be used to protect against this over voltage condition. By connecting a 9V Zener diode across the 9V input and ground and a 5V Zener diode across the 5V input and ground, we can ensure that our device does not receive more voltage than expected. Further protection is required on each of the power rails. Decoupling capacitors will be placed on the power lines close to the pins of our ICs to minimize inductance of the line and to provide cleaner, less noisy power for our fast-switching IC signals.

3.3. Strategic Components and Parts Selection

This section is dedicated to explaining our methods for choosing the processor, op-amps, DC-DC converters, and USB controller that we had planned for the device to use. It includes explanations for each method and comparison tables. In general, we choose the parts that satisfy the bare minimum requirements while also allowing the ability to add additional features and improvements.

3.3.1. Processor Selection

The main aspects of the processor that are being considered in selection are: throughput, memory, ADC support, USB support, serial peripherals, power consumption, and programmability. We will choose from two classes of processor: an MCU and a DSP. We could use a microprocessor or system-on-chip as our processor; however, they typically are more feature-rich, more complex, and more expensive, and we do not require more features than what a simpler device provides. Since the processor is the core element of our device, it is important that we choose one that gives us room for additional features. Just using the bare minimum will hinder us from that and make it harder to implement required features and improve existing ones.

Since our choice of processor heavily depends on the FFT, we first need to figure out its parameters: the number of frequency indices, N, and the buffer frequency, f_0. The number of frequency indices determine how many different frequencies the FFT can distinguish, and the buffer frequency determines which frequencies correspond to each frequency index. The MIDI note scale is based on the 12-tone equal temperament scale, which is a logarithmic scale that assigns note names to specific frequencies. There are 128 possible notes in MIDI, but note information alone does not account for pitch effects like vibrato, slurring, and pitch bend. The hardware should be capable of detecting at least the 128 frequencies that correspond to notes, with the option of implementing the detection of pitch effects and the sending of MIDI messages with this information. Thus, we will use a modified logarithmic scale as shown in Figure 3-18 based on the MIDI scale with extra bins in between each note bin and at the beginning and end of the scale. Our scale has two adjustable parameters: the starting note, M, and the number of intervals between two note bins, K. The MIDI scale under this scale has M at C-1 and K at 1.

$$f_{log}(n) = 6.875 \times 2^{\frac{3+M}{12}+\frac{n\ -K\ /\ 2}{12K}}$$

Figure 3-18: The logarithmic frequency scale for note detection.

Ideally, we want the parameters of the scale to be specific values. To determine K, we must determine how many cents there should be in between bins. A cent is one hundredth of a semitone. The typical vibrato effect can bend the pitch of a note from around ±34 to ±123 cents, and the average adult can reliably hear differences between two pitches down to a 25-cent difference.[18] A reasonable pitch interval for pitch detection would then be 12.5 cents, since it is around half of 25 and 34, and it is one eighth of 100. With 128 notes and intervals of 12.5 cents, 1024 pitch bins will be in the logarithmic scale. A pitch interval of 12.5 cents corresponds to a K of 8. The M parameter depends on the physical limitations of our note detection system. Since detection of very low frequencies requires a high buffer period, the lowest frequency bins will be removed from the scale such that the lowest frequency is higher and all other bins above the cutoff remain the same. The M parameter is the number of semitones above C-1, so an M of 40 makes the first frequency bin 82.4 Hz, which is the note E2.

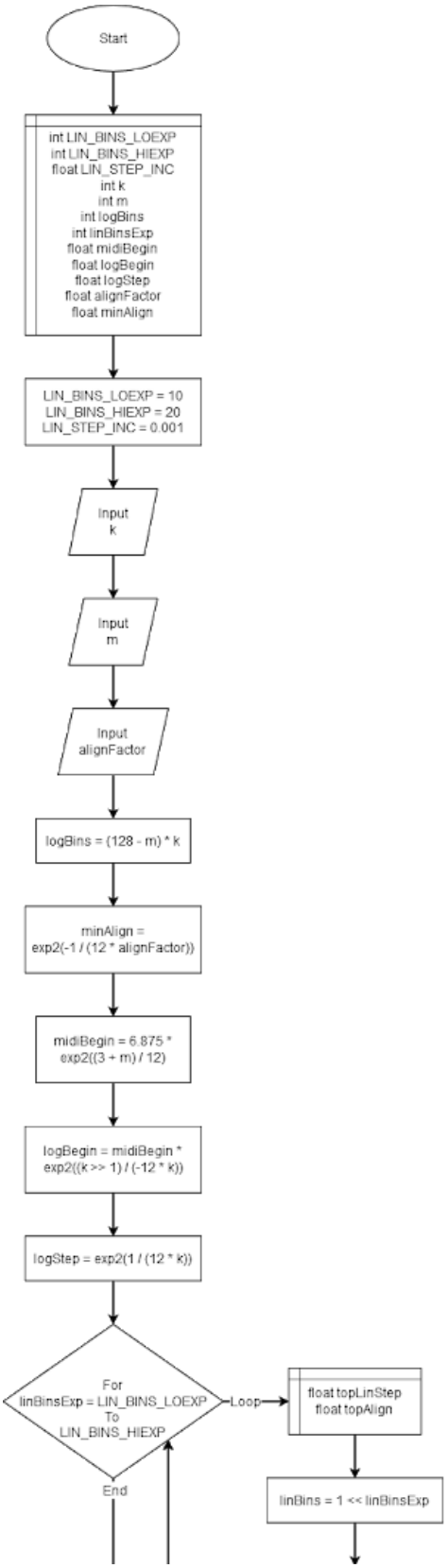
Since the FFT operates on a linear frequency scale, we must find a linear scale such that the difference between each linear bin and each logarithmic bin is small enough to not mismatch frequencies. The FFT linear scale as shown in Figure 3-19 starts at 0 and

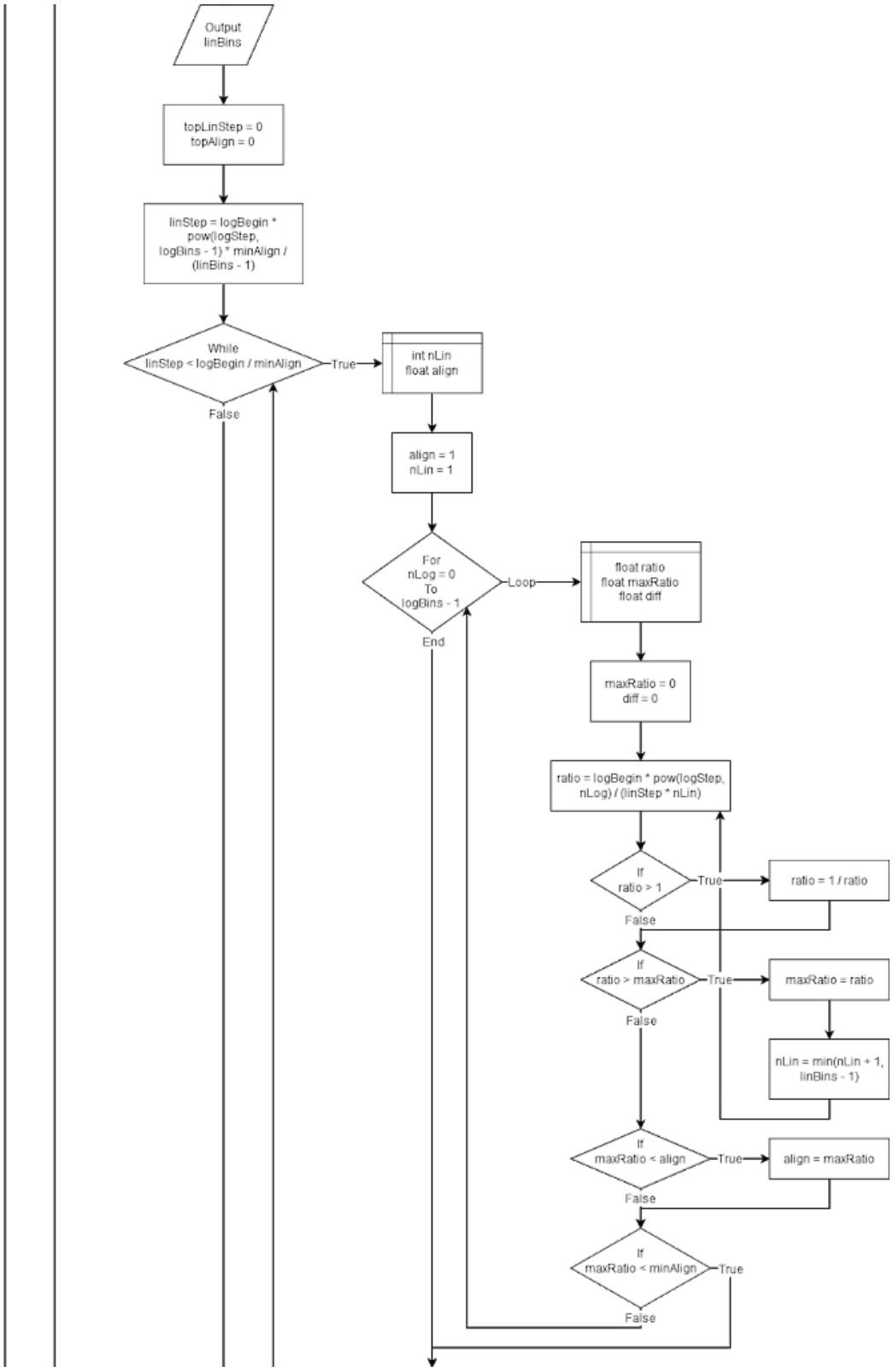
increases at a constant rate. This rate is determined by the buffer frequency, which also happens to be the frequency at n = 1. It also has an adjustable number of bins equal to a power of two. It is important to note that the number of bins in the linear scale is half the number of frequency indices produced by the FFT. This is due to the Nyquist limit. We define a measure called alignment for each logarithmic frequency bin. It is the ratio of the lowest frequency over the highest frequency of a pair of frequencies in different scales. The alignment that is assigned to each logarithmic bin is the greatest alignment it has when comparing it to the linear frequency bins.

$$f_{lin}(n) = n \times f_0$$

Figure 3-19: The linear frequency scale of an FFT.

The alignment between the linear and logarithmic scales is the lowest alignment for all the frequency bins. By varying the number of linear bins and the linear slope, we can find a set of linear frequency scales that are well aligned to the logarithmic scale we will use. We will vary the number of linear bins in between 2^{10} and 2^{20}, K in between 1 and 8, and the buffer frequency in between 0 and a semitone above the lowest logarithmic scale frequency. The goal is to find a linear scale that has a low bin count, has a high buffer frequency, and has an alignment greater than the minimum. This is done for various logarithmic frequency scales with different M and K. Figure 3-20 is a flowchart of the algorithm used to find these linear scales.


Start
int LIN_BINS_LOEXP
int LIN_BINS_HIEXP
float LIN_STEP_INC
int k
int m
int logBins
int linBinsExp
float midiBegin
float logBegin
float logStep
float alignFactor
float minAlign
LIN_BINS_LOEXP = 10
LIN_BINS_HIEXP = 20
LIN_STEP_INC = 0.001
Input
k
Input
m
Input
alignFactor
logBins = (128 - m) * k
minAlign =
exp2(-1 / (12 * alignFactor))
midiBegin = 6.875 *
exp2((3 + m) / 12)
logBegin = midiBegin *
exp2((k >> 1) / (-12 * k))
logStep = exp2(1 / (12 * k))
For
linBinsExp = LIN_BINS_LOEXP
To
LIN_BINS_HIEXP
Loop
End
float topLinStep
float topAlign
linBins = 1 << linBinsExp


Output
linBins
topLinStep = 0
topAlign = 0
linStep = logBegin *
pow(logStep,
logBins - 1) * minAlign /
(linBins - 1)
While
linStep < logBegin / minAlign
True
False
int nLin
float align
align = 1
nLin = 1
For
nLog = 0
To
logBins - 1
Loop
End
float ratio
float maxRatio
float diff
maxRatio = 0
diff = 0
ratio = logBegin * pow(logStep,
nLog) / (linStep * nLin)
If
ratio > 1
True
False
ratio = 1 / ratio
If
ratio > maxRatio
True
False
maxRatio = ratio
nLin = min(nLin + 1,
linBins - 1)
If
maxRatio < align
True
False
align = maxRatio
If
maxRatio < minAlign
True
False

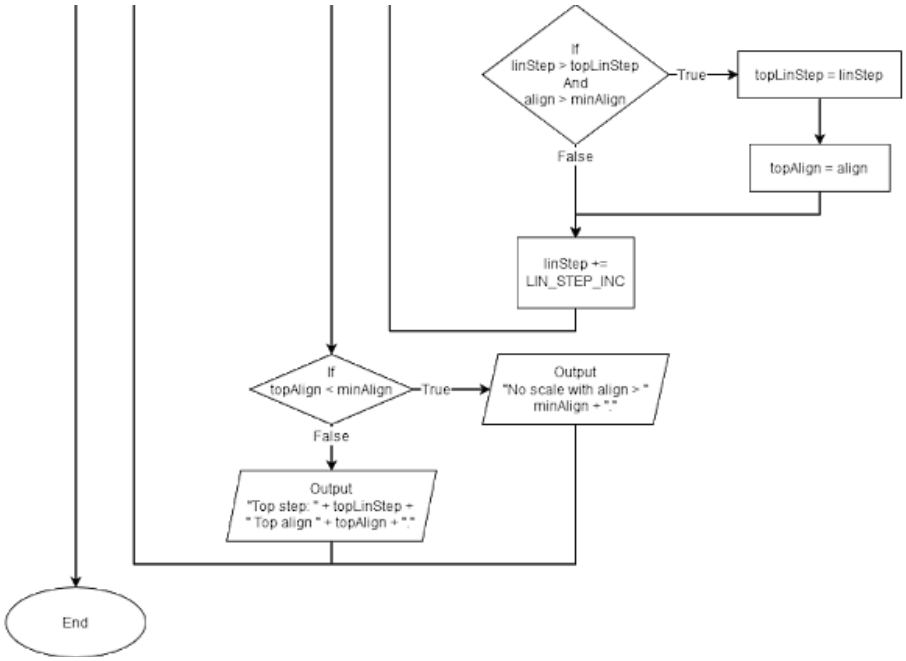


Figure 3-20: A flowchart of the algorithm used to determine the linear scales.

Tables 3-2, 3-3, and 3-4 show the results of the algorithm. The results for K above 2 are not shown, as the results are much worse than those for K = 2, whose results are already poor. The best option for minimizing latency would be to have the lowest note at E2, the number of frequency bins at 2048, and f_0 at 7.263 Hz. This corresponds to N being 4096 and the buffer period being 137 ms. The buffer period is the amount of time the processor must spend collecting samples. This buffer period is too high to keep the MIDI stream latency below 100 ms, but we can cut it in half by sampling only half the period and padding the other half with zeroes or a previous version of the signal. Thus, we would only spend 68.8 ms collecting samples. There are also other methods of reducing this number that will be discussed in the section on our FFT implementation. The product of f_0 and N determine the required sampling rate. In this case, it is 29.75 kHz. Any deviation from this sampling rate will require further testing.

Lowest Note (M)	Exponent of Minimum Bin Count	f_0 (Hz)	Alignment
E2 (40)	11	7.263	0.943875
C2 (36)	12	5.764	0.943902
A1 (33)	12	4.847	0.943909
E1 (28)	12	3.631	0.943928

C1 (24)	13	2.881	0.944153
A0 (21)	13	2.423	0.943977
F0 (17)	13	1.923	0.943997
C0 (12)	14	1.441	0.944182
C-1 (0)	15	0.720	0.944197

Table 3-2: Results of finding appropriate linear scales for K = 1 and a minimum alignment of 0.943874 (max. 1 semitone offset).

Lowest Note (M)	Exponent of Minimum Bin Count	f_0 (Hz)	Alignment
E2 (40)	12	3.427	0.943881
C2 (36)	13	2.719	0.944149
A1 (33)	13	2.286	0.944282
E1 (28)	13	1.713	0.943978
C1 (24)	14	1.360	0.944179
A0 (21)	14	1.144	0.943906
F0 (17)	14	0.908	0.943965
C0 (12)	15	0.679	0.944890
C-1 (0)	16	0.340	0.944897

Table 3-3: Results of finding appropriate linear scales for K = 2 and a minimum alignment of 0.943874 (max. 1 semitone offset).

Lowest Note (M)	Exponent of Minimum Bin Count	f_0 (Hz)	Alignment

E2 (40)	12	3.142	0.971794
C2 (36)	13	2.494	0.971791
A1 (33)	13	2.097	0.971895
E1 (28)	13	1.571	0.971907
C1 (24)	14	1.247	0.971826
A0 (21)	14	1.019	0.971608
F0 (17)	14	0.832	0.972197
C0 (12)	15	0.605	0.973053
C-1 (0)	16	0.302	0.974673

Table 3-4: Results of finding appropriate linear scales for K = 2 and a minimum alignment of 0.971532 (max. ½ semitone offset).

Throughput

The throughput is the most important characteristic of the processor that must be considered. There must be a maximum of 100 ms between when a sound is heard and when the related MIDI notes are output. If the processor spends 68.8 ms collecting samples, then it can only spend up to 31.2 ms processing the data. Thus, the processor should be able to perform a fast Fourier transform (FFT) with a set of samples within 25 ms. This gives enough time to perform the filtering methods, signal sampling, and data transfer in a short time. This requires high enough throughput, and can be achieved through high clock speeds, large word size, Single Instruction Multiple Data (SIMD) operations, and low clocks per instruction.

The maximum time-complexity of the FFT algorithm is O(N log N), where N is the number of samples. This means that it takes $N \log_2 N$ steps to perform the FFT. Each step will require a certain amount of specific instructions; by figuring out the instructions required for each step, the total number of each instruction can be calculated. Table 3-5 lists the instructions used in implementing the FFT and describes their function. We will define one step as half of a complex butterfly operation, since the number of those operations is half the time-complexity of the algorithm. Each complex butterfly operation consists of 1 complex multiplication, 1 complex addition, and 1 complex subtraction. However, as mentioned before, we can split all complex butterfly operations into two real butterfly operations, so one step is also one real butterfly operation. Within each real butterfly operation there is 1 real multiplication, 1 real addition, and 1 real subtraction.

Instruction	Description
`add x,y`	Adds two signed integers x and y and stores the sum in x.
`addi x,N`	Adds a constant to x and stores the sum in x.
`sub x,y`	Subtracts two signed integers x and y and stores the difference in x.
`or x,y`	ORs each bit of x with the corresponding bit of y and stores the result in x.
`srl x,N`	Shifts x's bits N times to the right without preserving the sign bit. The result is stored in x.
`sla x,N`	Shifts x's bits N times to the left while preserving the sign bit. The result is stored in x.
`mul x,y`	Multiplies two signed integers x and y, creating a product with twice the size of each factor. The top half is stored in x and the bottom half is stored in y.
`mov x,y`	Copies the value of y into x.
`lw x,A`	Copies the value at address A into x.
`sw x,A`	Copies the value of x into address A.
`cmp x,y`	Compares the values of x and y, setting flags.
`jeq A`	Checks if the last comparison was between two equal numbers and jumps to A if that is true.
`jmp A`	Jumps to address A.

Table 3-5: Table of instruction names and descriptions.

Since most microcontrollers do not support floating-point operations, these real operations will be implemented using fixed-point math. Real additions and subtractions can be done with the same integer instructions. However, a real multiplication requires more instructions. A real multiplication can be done with an integer formatted as a fixed-point fractional number, with N bits used as the fractional part. This formatting is referred to as QM.N format, where one bit is used as the sign bit, M bits are used for the integral part,

and N bits are used for the fractional part.[11] We will also need to consider the instructions required to iterate through parts of the entire calculation. As shown by Figure 3-14, there are 3 nested iterations: one iteration over stages, one iteration over butterfly operations with different coefficients, and another one over different sets of butterfly operations with the same coefficient. Thus, our implementation would also require three nested for-loops. Since the coefficients in every butterfly operation will not change, there is no need to calculate sines and cosines for each one, so they are treated as constants. Also, we assume that samples are already written to the buffer in bit-reversed order, so we would not need to reorder them. Table 3-6 shows the implementation of real multiplication and for-loops using instructions.

Statement	**Real Multiply**	**For-Loop**

Instruction Sequence	`mul x,y`
	`srl y,Q`
	`sla x,N-Q`
	`or x,y`
		`for:`
		`cmp x,y`
		`jeq endfor`
		`…`
		`addi x,1`
		`jmp for`
		`...`

Instruction Count	4	4
Notes	Assuming x and y are each N bits and the product is a 2N-bit integer, with x holding the higher half and y holding the lower half. Q is the number of bits in the fractional part.	This operation is implemented like this in most architectures, with slight potential differences in instructions, order, and number.

Table 3-6: A table showing the instructions for each operation.

We expect N/2 log N for-loop statements to be executed (log N for the outer stage loop, N/2 for both inner loops), and 2 load word, 2 move, and 2 store word instructions per pair of real butterfly operations (N log N). Also, we expect two extra load word instructions for each iteration over butterfly operations with different coefficients to load the sine and cosine coefficients (2(N - 1)). With knowledge of the instructions required for each operation and the number of operations required for one complete FFT, we can calculate the number of each instruction required to perform the entire FFT. Table 3-7 shows the results of this calculation. This analysis may be inaccurate for processors with SIMD instructions, since the processor may be able to achieve the results of multiple instructions with just one instruction, making the results of this method an overestimate. However, this method is still useful as it gives an upper bound for the execution time in this case. A DSP is designed to do these kinds of operations quickly and efficiently, however they come at a greater cost. We will discuss the details of the runtime of the FFT for each processor we consider.

Operation	Number of Operations	Instructions per Operation	Instruction	Number of Instructions
For-Loop	22528	4	`cmp`	22528
			`jeq`	22528
			`jmp`	22528
			`addi`	22528
Real Add	45056	1	`add`	45056
Real Subtract	45056	1	`sub`	45056
Real Multiply	45056	4	`mul`	45056
			`srl`	45056
			`sla`	45056
			`or`	45056
Load Word	53246	1	`lw`	53246
Store Word	45056	1	`sw`	45056
Move	45056	1	`mov`	45056
Total	**315392**			**503806**

Table 3-7: Operation and instruction tally for the FFT when N = 4096.

Memory

The ADC sample rate must be 29.75 kHz for the FFT to be accurate with the lowest sampling period. MIDI uses 7 bits for note velocity (equivalent to volume), so we only need to distinguish 128 different magnitudes on the frequency spectrum. We will only capture at least 8 bits per sample, but we will need 16 bits to store the real numbers used in the calculation. If N = 4096, and each sample uses 16 bits, we need 8 kB of memory to store this buffer. The size of RAM must accommodate at least two of these buffers. A faster or more complex implementation of the FFT may require more space. Also, the FFT needs a buffer large enough to store the twiddle factors: around 2 * 2 * 4096 / 2 bytes. We need a buffer to store the 8-bit magnitudes of each MIDI note and some spare stack space. The table below lists how much memory we will need. In total, we would use at least 30 kB of memory just for the FFT. To give room for additional or improved features, the total size

of RAM should be at least 64 kB. The constant coefficients (twiddle factors) can be stored in the processor's ROM. Some MCUs can use some of their program storage as writable non-volatile RAM. We will take this into account when determining the memory specifications of each processor. Note that this analysis assumes we will only do a plain FFT on the sample buffer. In fact, we use a more complex method that reduces memory requirements to 4 kB. This is explained in greater detail in section 6.2.

Purpose	Sample buffers (kB)	Twiddle Factors (kB)	Note Magnitudes (kB)	Stack space (kB)	Misc. (kB)	Total (kB)
Space Required	16	8	0.125	2	4	30.125

Table 3-8: A table showing the memory requirements of the processor.

Analog-to-Digital Converter

The processor we use must include an ADC that is capable of sampling all signals whose frequencies are within the MIDI range without aliasing. The maximum sampling rate is the most important aspect of the ADC, since it limits the bandwidth of signals that can be converted; it must be 29.75 kHz. An ADC that can also operate at a much lower sampling rate is a bonus since that allows us to work with low frequencies more efficiently. The number of values the ADC can detect is not very important, as the MIDI protocol uses 7 bits for note velocity, which relates to volume, and most MCUs and DSPs use at least 8 to 10 bits per sample. The ADC should have a high input voltage range, though, since this gives a good signal-to-noise ratio. Although we could choose to buy a processor and ADC separately, this would likely increase our costs. We would need more space on the PCB for both parts, and more complexities would be introduced into the design. Also, most MCUs and DSPs have an integrated ADC, so it is best to take advantage of that.

USB and MIDI

USB is a complex serial transmission interface that defines multiple protocol layers of the TCP/IP model, so it would be best to have either a separate IC to handle the interface or integrated capability within the processor itself. If the processor does have USB support, it must have USB 2.0 support with at least full-speed transmission capabilities to support the USB-MIDI specification. [18] This implies that the clock speed is greater than 12 MHz. For our MIDI implementation, the processor only needs general-purpose input/output (GPIO) pins and a UART. SPI support is not required if the processor includes a USB interface, since this would only be used to communicate with an external USB controller.

Programmability

The processor we choose must be easily programmable. There must be a software development application available for the processor that allows us to write C or C++ source code and allows us to use the special or required features of the processor. There must be libraries available for implementing the math-focused parts of our software on the processor. There must be cheap hardware available that can connect to a desktop computer and the processor on the PCB to program the processor. If no such hardware

exists, there must be enough documentation to be able to implement a programmer on an MSP430, and the programming protocol must be simple enough to implement. The best case would be for all the necessary development tools to be available at a low price, and a processor that would require our own implementation of its programmer will be low on the list of best choices.

Selection

When looking at the datasheets for processors, we are interested in parameters that correspond to our requirements; these are listed in the processor specification tables for eleven different processors. The clock frequency, instruction set architecture (ISA), availability of SIMD instructions, and inclusion of an FPU are related to the throughput of the processor. The cycles-per-instruction of the processor is also important, but there are more details for consideration, so the estimations of execution time will only be done for a small set of best candidates. The memory parameters for each processor must be reasonably high. While we do need at least 32 kB to implement the raw FFT, we plan on optimizing its runtime and memory requirements by at least one fourth, so a processor with less RAM is acceptable. Also, we are looking at whether the processor can read constant data from its ROM, whether it can write configuration data to its ROM, and whether it can use its ROM as RAM (quick reads and writes). If the ROM can be written to by the processor itself, then we do not need an external non-volatile memory for storing configuration data. Almost all processors in the tables that have ROMs are capable of this. If the ROM can reliably be used as RAM, then the actual RAM for that processor is the sum of the memory sizes of both the ROM and RAM. The ADC parameters are also included in the tables of processor specifications. All processors have ADCs that are suitable for our application. Most of them also support I2C, SPI, and UART. As for the programmability, our focus in the tables is the availability and cost of a programmer for each processor. The estimated maximum power requirements are also included with the V_{IN} and estimated maximum current fields. This is important if we want to keep the temperature low with passive cooling. All these parameters were collected from the datasheets and user manuals of each processor.

Beginning with the TI MSP430 MCUs, we considered the MSP430G2553 and the MSP430FR6989 at first because we had used them before, and we had programmers for those chips. However, the MSP430G2553 clearly lacks the memory requirements as it only has 512 bytes of RAM. For the MSP430FR6989, the memory size is also rather low, and each chip is $6.08. Since we are buying at least 2 of every IC, the total cost would be $18.24. Since there is nothing remarkable about this processor, it is not cost-effective. The MSP430FR5992 is an interesting one because it has a DSP coprocessor that can perform the FFT without using the CPU's resources. The programmer for this chip is also relatively cheap at $16.99. The only downside is that it has 8 kB, which gives us less room for performing the FFT. One of the cheapest MCUs that we found is the Atmel ATSAMD20E18. It has a much higher clock speed and more overall memory than the MSP430 MCUs. It surpasses the basic requirements, but unfortunately, the programmer for this MCU costs $69.00, which is high above our budget for the processor. There are three other processors whose programmers are very expensive: the TI TMS320C5533 DSP, the Cypress S6E1C32B0A MCU, and the SiLabs EFM32LG842F256G-F-QFP64R. These processors would be good choices, since all three have a high clock speed and implement a USB 2.0 interface, and the TI DSP has SIMD instructions for DSP tasks and plenty of memory.

We found another Cypress MCU, the CY9BF564K, but it is marked as obsolete by Cypress, and we cannot find enough documentation on how to develop for and program it. While it does have a high clock speed, 32 kB of RAM, and USB 2.0 support, we will not consider using it for that reason. Also, it uses up to 238 mW of power, which is at least an order of magnitude above most of the others. We considered another DSP: the AKM AK7755EN. The problem with it is that we cannot find enough information to know how to program and develop for it. It is also the second most expensive chip in the list. The last two MCUs, the ST STM32G071KB and the Microchip dsPIC33EP128MC202, have high clock speeds and an acceptable amount of memory. While the Microchip MCU has only 16 kB of RAM, it has specialized DSP instructions that can be used to perform the FFT much faster. Tables 3-9-A through 3-9-E list the relevant specifications and qualities of each processor that we have considered. Out of all these eleven processors, we have selected three best candidates for further throughput analysis: the TI MSP430FR6989, the ST STM32G071KB, and the Microchip dsPIC33EP128MC202.

Processor	**TI MSP430G2553**	**TI MSP430FR6989**	**TI MSP430FR5992**
Clk. Freq. (MHz)	16	16	16
V_{IN} (V)	3.3	3.3	3.3
Max. Current (mA)	4.3	2.68	3.00
ISA	TI MSP430	TI MSP430	TI MSP430X
SIMD Instructions	No	No	Yes
FPU	No	No	Unknown
Min. ADC Bit Depth	10	12	12
Sample Rate (ksps)	200	200	200
ADC Range (V)	0 - 3.3	0 - 3.3	0 - 3.3
Serial Interfaces	I2C, SPI, UART	I2C, SPI, UART	I2C, SPI, UART
ROM (kB)	16	128	128
RAM (kB)	0.5	2	8
Writable ROM	Yes	Yes	Yes
ROM as RAM	Yes	Yes	Yes
Debug Interface	JTAG	JTAG	JTAG
Unit Cost (USD)	1.77	6.08	4.10

Programmer	MSP-EXP430G2ET LaunchPad Development Kit	MSP-EXP430FR6989 LaunchPad Development Kit	MSP430FR5994 LaunchPad Development Kit
Prgm. Cost (USD)	0.00	0.00	16.99
Other Costs (USD)	0.00	0.00	0.00
Total Cost (USD)	5.31	18.24	29.29
Notes	Development kit already owned.	Development kit already owned. Includes hardware multiplier.	Includes DSP coprocessor with FFT support.

Table 3-9-A: Processor specifications for the MSP430 MCUs being considered.

Processor	**TI TMS320C5533**	**AKM AK7755EN**
Clk. Freq. (MHz)	100	18.6, 123
V_{IN} (V)	1.3, 3.3	1.2 & 3.3
Max. Current (mA)	16.9	68.5
ISA	?	?
SIMD Instructions	Yes	?
FPU	Unknown	Yes
Min. ADC Bit Depth	10 or None	24
Sample Rate (ksps)	62.5/64	96
ADC Range (V)	?	0 - 3.3
Serial Interfaces	I2C, SPI, UART, USB 2.0	I2C, SPI
ROM (kB)	128	0
RAM (kB)	320	63
Writable ROM	?	N/A
ROM as RAM	?	N/A
Debug Interface	JTAG	No

Unit Cost (USD)	3.95	5.64
Programmer	TMDX5535EZDSP	?
Prgm. Cost (USD)	249.00	?
Other Costs (USD)	0.00	?
Total Cost (USD)	260.85	16.92
Notes	Only available as BGA. Powering the chip is more complicated than other chips.	Clock frequency varies with sample rate. Includes amplifier & analog mixer. Evaluation kit only available through inquiry. Only available as QFN.

Table 3-9-B: The processor specifications for both DSPs being considered.

Processor	**Cypress CY9BF564K**	**Cypress S6E1C32B0A**
Clk. Freq. (MHz)	160	40.8
V_{IN} (V)	3.3 or 5	3.3
Max. Current (mA)	72	5.9
ISA	ARM Cortex-M4F	ARM Cortex-M0+
SIMD Instructions	Yes	No
FPU	Yes	No
Min. ADC Bit Depth	12	12
Sample Rate (ksps)	2000?	500?
ADC Range (V)	0 - 5	0 - 3.3
Serial Interfaces	I2C, SPI, UART, USB 2.0	I2C, SPI, UART, USB 2.0
ROM (kB)	288	128
RAM (kB)	32	16
Writable ROM	Yes	Yes
ROM as RAM	Partial	No
Debug Interface	SWJTAG	SWD

Unit Cost (USD)	1.42	0.85
Programmer	?	FM0-64L-S6E1C3 MCU Starter Kit
Prgm. Cost (USD)	?	49.00
Other Costs (USD)	?	?
Total Cost (USD)	4.26	51.55
Notes	Obsolete. Hard to find documentation. ROM code cannot run during writing.	Supported unlike previous Cypress chip.

Table 3-9-C: The processor specifications for the Cypress MCU's being considered.

Processor	**Atmel ATSAMD20E18**	**SiLabs EFM32LG842F256G-F**
Clk. Freq. (MHz)	48	48
V_{IN} (V)	3.3	3.3
Max. Current (mA)	6.16	10.8
ISA	ARM Cortex-M0+	ARM Cortex-M3
SIMD Instructions	No	No
FPU	No	No
Min. ADC Bit Depth	8	12
Sample Rate (ksps)	320	1000
ADC Range (V)	0 - 3.3	0 - 3.3?
Serial Interfaces	I2C, SPI, USART	I2C, UART, USB 2.0
ROM (kB)	256	256
RAM (kB)	32	32
Writable ROM	Yes	Yes
ROM as RAM	Partial	No
Debug Interface	SWD	SWD

Unit Cost (USD)	1.24	4.21
Programmer	SAM D20 Xplained Pro Evaluation Kit	EFM32LG-STK3600
Prgm. Cost (USD)	69.00	99.00
Other Costs (USD)	?	?
Total Cost (USD)	72.72	111.63
Notes	Can only write to ROM in chunks.	

Table 3-9-D: The processor specifications for other MCUs with expensive development kits.

Processor	**ST STM32G071KB**	**Microchip dsPIC33EP128MC202**
Clk. Freq. (MHz)	64	60
V_{IN} (V)	3.3	3.3
Max. Current (mA)	7.7	40
ISA	ARM Cortex-M0+	dsPIC33E
SIMD Instructions	No	No
FPU	No	No
Min. ADC Bit Depth	12	10
Sample Rate (ksps)	2000	1100
ADC Range (V)	0 - 3.3	0 - 3.3
Serial Interfaces	I2C, UART	I2C, SPI, UART
ROM (kB)	128	128
RAM (kB)	36	16
Writable ROM	?	Yes
ROM as RAM	?	No
Debug Interface	SWD	JTAG
Unit Cost (USD)	3.39	2.56

Programmer	ST-LINK/V2	TEMLP001 LProg Programmer
Prgm. Cost (USD)	22.61	20.00
Other Costs (USD)	?	?
Total Cost (USD)	32.78	27.68
Notes		Includes single-cycle multiply, multiply-accumulate, and multiply-subtract instructions.

Table 3-9-E: The processor specifications for other MCUs with cheaper development kits.

To estimate the throughput of each processor, we must determine the amount of time required to execute each kind of instruction. This is done by considering the clock speed and the number of cycles per instruction. Table 3-10 lists each processor and the number of clock cycles spent executing each instruction. This analysis does not completely consider any special instructions or hardware that may decrease the time of the FFT. There are important details that the table does not include that affect the throughput. The TI MCU's version of the addi instruction takes 4 clock cycles to execute, but it can be reduced to 1 by using the add instruction with the constant generator register as an argument. The MCU does not have a multiply instruction, but instead it has a hardware multiplier circuit that can be configured with two sw-like instructions, and it takes three more cycles until the multiplication is finished. This MCU can also perform multiply-accumulate operations with the multiplier using the same amount of time. The TI MCU does not have instructions to shift left or right the bits in a register in one cycle. However, each right shift by N in the FFT is paired with a left shift by 16 - N, so in total each pair of these instructions takes 16 cycles. The ST MCU has a high clock speed but requires wait states when accessing the ROM. However, the MCU has an instruction buffer and cache that can store instructions, thus keeping the effective clock speed at 64 MHz most of the time. The effective clock speed decreases after branching and fetching long instructions, however. Although the Microchip MCU has special instructions for DSP that are single-cycle, such as multiply-accumulate and multiply-subtract, they do not appear to be useful for an FFT implementation because the same number of instructions is required with or without using those special instructions.

	Individual Instruction Execution Cycles						
Processor	`cmp`	`jeq`	`jmp`	`add`	`addi`	`sub`	`mul`
MSP430FR5992	1	2	2	1	4	1	5
STM32G071KB[15]	1	1-2	2	1	1	1	1
dsPIC33EP128MC202	1	1 or 4	4	1	1	1	1

	srl	sla	or	lw	sw	mov	
MSP430FR5992	1-16	1-16	1	2	3	1	
STM32G071KB[15]	1	1	1	1 or 2	1 or 2	1	
dsPIC33EP128MC202	1	1	1	1 or 4	1 or 4	1	

Table 3-10: Comparison of individual instruction execution cycles for processors

Table 3-11 shows the estimated execution times for a 4096-point FFT based on the instructions used and the number of cycles per instruction. The MSP430FR5992 has the slowest execution time for the FFT due to its low clock speed and lack of a single-cycle bit shift instruction. The STM32G071KB and the dsPIC33EP128MC202 have similar execution times for each instruction. However, the ST MCU has a higher clock speed, so the execution time of the FFT is much quicker than the Microchip MCU.

Processor	Clock Speed (MHz)	Total Cycles Spent	FFT Execution Time (ms)
MSP430FR5992	16	1570812	98.2
STM32G071KB	64	624636	9.76
dsPIC33EP128MC202	40	669692	16.7

Table 3-11: Comparison of FFT Execution Times

An important feature that the TI MCU has is a DSP coprocessor that is designed to do an FFT quickly and efficiently. TI has published benchmarks of processors with this feature on performing an FFT. [16] By extrapolating their data, we estimate that the TI MCU would, using this feature, be able to perform a complex FFT with N = 2048 in 2.82 ms, which corresponds to a real FFT with N = 4096 in the same time. This relationship between the complex and real FFTs is explained in the section on designing the FFT algorithm. Although this coprocessor cannot do a complex FFT with N = 2048, we are able to reduce the N of our FFTs. This will also be explained later. Since the TI MCU is much faster at doing the FFT than the other processors, we will use the TI MSP430FR5992 in our project.

Final choice: TI MSP430FR5992

3.3.2. Operational Amplifiers Selection

We will be utilizing operational amplifiers in multiple parts of our project for buffering, amplification, and filtering. There are many different op amps to choose from and a very large number of specifications that define them. For our purposes, we want to choose the best for an audio application since these op amps will mostly be on the analog input

section of our device. Some important characteristics to look at when choosing our op amps will be gain, input and output impedance, noise/total harmonic distortion, input and output voltage ranges, and slew rate. Gain is always one of the most important factors of an amplifier but in this case most op amps will be able to have a range of gain well beyond what we need. We should be able to configure nearly any op amp to have a small to moderate amount of gain, so this characteristic won't be important to analyze in our selection. For voltage amplification, which is what we will be doing, it is ideal to have a very high input impedance and a very low output impedance. High input impedance minimizes any voltage loss while low output impedance will minimize any loading effects and guarantees the load gets most of the signal. Most op amps should have enough input and output impedance for our applications, but it is still an important factor that we will be comparing.

Probably the most important characteristic for us will be the noise and total harmonic distortion levels. These specifications describe how much the signal is distorted from the input to the output since all components are non-ideal and will change the signal to some extent. Since we are working with audio which is much more sensitive to distortion then say a binary digital signal, we want to minimize distortion and noise as much as possible. The output voltage for any modern op amp will be much higher than any voltage we will need to input to our analog to digital converter so this specification will not be the most important. The input voltage however may be important as we need to make sure that it can take input from most microphones and instruments without distorting or clipping. The slew rate of an op amp is the rate of change of the output voltage as caused by a change in the input voltage. For instance, if the input voltage changes by 2V in 1 ms and the voltage gain was 1, the slew rate would dictate how fast the output could mimic the input. If the slew rate was too low, it may take longer than 1 ms to change 2V in the output and distort the wave form. As such, it is best to have a slew rate as high as possible to minimize distortion, particularly in the high frequencies. Since we don't need frequencies beyond human hearing and possibly even instrumentation levels, it is possible that the slew rate won't be incredibly important, but it is still best to try and minimize the distortion of the signal. Lastly, price may be a factor in our selection depending on how expensive or affordable an op is. If one op amp is only slightly better specification wise but much more expensive, it may not be the best choice for our project. We will be comparing several dual op amps commonly used in analog audio and preamp circuits to make our selection. Most of these have a similar quad op amp equivalent that we may use instead to save PCB space if needed.

When comparing the input impedances, they should be sufficiently high on all the op amps but the TL072 and OPA2134 are exceptional and closer to the ideal impedance of infinity. Similarly, the output impedances on all the op amps are enough but exceptional on the OPA2134 and NE5532. The amount of voltage noise is especially high on the TL072 and it also has the most total harmonic distortion. The noise is low on the other three op amps but the THD is very low on the OPA2134 and NE5532. The input voltage range is relative to the supply voltage for all the op amps as the supply voltage affects the bias and thus the amount of voltage swing possible before clipping. For all the op amps the supply voltage range is above 18 volts which is way more than the peak voltage line level that we could see from an instrument or microphone of 1-2 volts. The slew rate is the best on OPA2134 and TL072 which will make the high frequency response better. Lastly, the price of all the op amps are relatively affordable except for the OPA2134.

	TI TL072	Burr Brown OPA2134	TI NE5532	TI LM833
Input Impedance (Ω | pF)	10^{12} | ?	10^{13} | 2	$3 \cdot 10^{4-5}$ | ?	$1.75 \cdot 10^{5}$ | 12
Output Impedance (Ω | pF)	~100 | ?	.01 | ?	.3 | ?	37 | ?
Voltage Noise $\frac{nV}{\sqrt{Hz}}$ (1 kHz)	18	8	5	4.5
THD (%)	.003	.00008	~.0001	.002
Input Voltage Range (V)	(V_{cc}-) - .3 to (V_{cc}+) + 36	(V_{cc}-) - .7 to (V_{cc} +) + .7	(V_{cc}-) to (V_{cc} +)	(V_{cc}-) to (V_{cc} +)
Slew Rate ($\frac{V}{\mu S}$)	13	20	9	7
Price ($) (As listed on DigiKey Electronics)	0.41	4.26	.48	1.02

Table 3-12: A list of op-amps and their properties.

Overall, the OPA2134 has the best specifications in every category we analyzed but the price is slightly prohibitive as we need multiple of these integrated circuits for all our buffering, splitting, and filtering purposes. The low output impedance, price, noise, and THD makes the NE5532 seem like the second-best option. The slew rate is on the lower side but since we are working with lower frequencies in musical notes (< 20 kHz), the slew rate is not the most important factor. [23]

Final choice: TI NE5532

3.3.3. DC-DC Converters

Our device is powered entirely by external DC power sources, connected by either a barrel jack for a 9V source or by USB for 5V. The device uses 5V power for the processor and other logic components, 3.3V to power the microcontroller and 48V for the microphone 48 volt phantom power supply. In order to get the voltage levels we need from both sources, we need a 9V to 5V DC-DC converter, a 5V to 48V DC-DC converter, and a 5V to 3.3V DC-DC converter

9V to 5V DC-DC converter

To build the 9-to-5 volt DC-DC converter needed for our device, we will be utilizing the TI WEBENCH power designer. This application uses TI DC-DC converter chips to create customized circuits based on the voltage and current requirements of the user. We looked at DC-DC converter circuits for 9 V input and 5 V output with a 500 mA max output current. When looking at the DC-DC converter chips that were suggested by TI we compared them using the following parameters: cost, BOM count, max output current, input voltage range, output voltage range, efficiency, footprint and frequency. The cost parameter is split into two categories: IC cost and BOM cost. The BOM cost refers to the total cost of all the parts that would be included in the bill of materials to build the circuit. The BOM Count is the amount of parts needed to build the circuit which usually includes resistors, capacitors, inductors and other materials along with the IC. An increase in the BOM count will usually result in an increase of the BOM cost for the circuit. The IC cost is just the cost of the IC chip without any other parts of the circuit which would be needed to make the DC-DC converter. Some ICs that were implemented in the designs that fulfilled our voltage and current requirements include the LMR50410X, TPS621351, TPS563231, and LMR14010A. A table with a comparison between the ICs is shown in Table 3-13.

	LMR50410X	TPS563231	TPS621351	LMR14010A
IC Cost	$0.45	$0.22	$0.73	$0.40
BOM Cost	$0.83	$0.97	$1.50	$0.92
BOM Count	9	11	8	8
Max Output current	1 A	3 A	4 A	1 A
Input Voltage Range	Min: 4 V Max: 36V	Min: 4.5 V Max: 17 V	Min: 3 V Max: 17 V	Min: 4 V Max: 40 V
Output Voltage Range	Min: 1 V Max: 28V	Min: 0.6 V Max: 7 V	Min: 0.8 V Max: 12 V	Min: 0.77 V Max: 38.4 V
Efficiency	93.8%	95.7%	92.4%	92.7%
Short Circuit Protection	Yes	Yes	Yes	Yes
Footprint	116 mm^2	167 mm^2	66 mm^2	194 mm^2
Frequency	700 kHz	696.12 kHz	2.44 MHz	700 kHz

Table 3-13: Comparison of 9V-to-5V DC-DC converters.

The desirable features that we are looking for in our 9 to 5 volt DC-DC converter include low cost, high efficiency and low footprint. All the ICs have very low BOM costs with the TPS621351 being the highest at $1.50. This makes this parameter not that important since none of these chips are too expensive. The input voltage ranges and output voltage ranges for all the ICs in the table are satisfactory for the device. The frequency of each IC is shown but should not be a factor in choosing which one to use. It should be noted that

all the DC-DC converter chips include short circuit protection which is helpful. In terms of efficiency, all the ICs have very high efficiencies with the TPS621351 having the lowest efficiency at 92.4% and the TPS563231 being the highest at 95.7%. This means that there is only a maximum difference of 3.3% in efficiency between all the circuits. In terms of footprint, all the footprints for each of the ICs are low with none of the footprints being smaller than 194 mm^2. One of the ICs had a significantly smaller footprint than the others. This IC was the TPS621351 which had a footprint of 66 mm^2. All the footprints should be more than small enough for use in our device. The choice for which IC to use for the 9 to 5 volt DC-DC converter came between the TPS563231 and the TPS621351. The TPS563249 has the highest efficiency but a larger footprint while the TPS621351 has the smallest footprint but a lower efficiency than the TPS563231. Since both footprints should be small enough for use in our device, we decided to use the TPS563231 since it had the best efficiency. The circuit design for the TPS563231 provided by WEBENCH is shown in Figure 3-21.

Final Choice: TPS563231

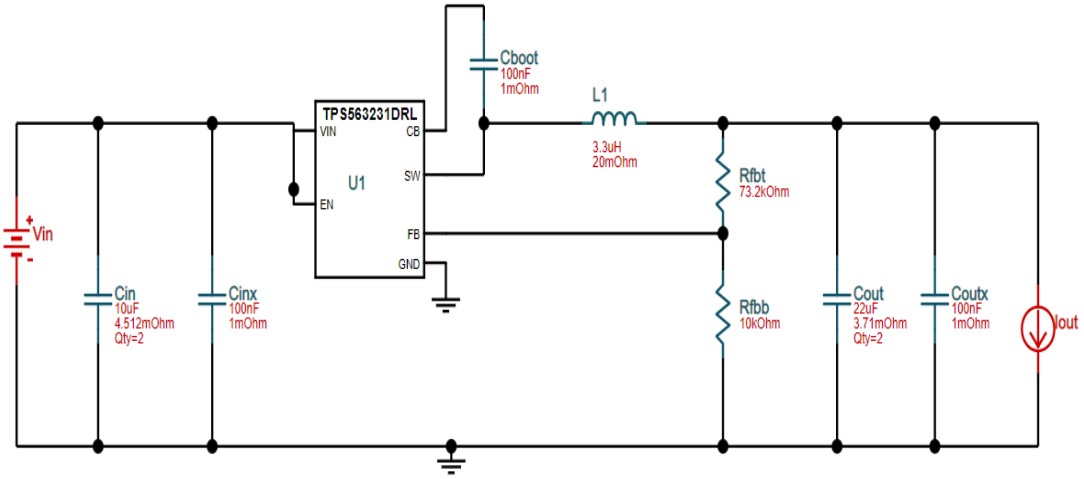


Figure 3-21. TPS563231 Circuit Design

5V to 3.3V DC-DC converter

The processor we're using requires 3.3 volts. This means we need a DC-DC converter on the 5V line to step it down to 3.3V to deliver to any pins on the processor that need it. Once again, we are using TI WEBENCH for this design. After inputting the power requirements of this converter, WEBENCH gave 396 potential circuits. Even after narrowing down the options to only those designs with BOM count of 12 components or less, cost of $4.30 or less, and efficiency of 88% or higher there are 102 options available. Valuing efficiency the most, we can sort the results by highest efficiency and take the top four options to compare. The schematics of each DC-DC converter are shown in figures 33A through 33D and figure 33T is a table comparing all the parameters of each circuit. All four options are from TI's TPS6282x line of chips, so they are very similar in performance and cost. Even expanding my search to the top eight results when sorting by efficiency, all the ICs used are from this line. TI seems to believe this is the correct chip for the job, so all that's left is to narrow it down to the specific model that is best for our device. Table 3-14 shows a comparison between the ICs that were analyzed.

	TPS62823	TPS62822	TPS62825	TPS62826
IC Cost	$0.45	$0.41	$0.41	$0.51
BOM Cost	$3.20	$3.10	$3.10	$3.20
BOM Count	8	8	8	8
Max Output current	3 A	2 A	2 A	3 A
Input Voltage Range	Min: 2.4 V Max: 5.5V	Min: 2.4 V Max: 5.5 V	Min: 2.4 V Max: 5.5 V	Min: 2.4 V Max: 5.5 V
Output Voltage Range	Min: 0.6 V Max: 4 V	Min: 0.6 V Max: 4 V	Min: 0.6 V Max: 4 V	Min: 0.6 V Max: 4 V
Efficiency	96.4%	95.9%	95.9%	95.9%
Short Circuit Protection	Yes	Yes	Yes	Yes
Footprint	113 mm^2	113 mm^2	111 mm^2	111 mm^2
Frequency	2.06 MHz	2.07 MHz	2.07 MHz	2.07 MHz

Table 3-14: Comparison of 5V to 3.3V DC-DC converters.

Being all from the same family of chips, there are few parameters where these ICs vary. TPS62823 is more efficient than the other options but only by 0.5%. With this being a theoretical, calculated efficiency rather than measured from a test board, we can ignore the difference in efficiency. TPS62825 and TPS62826 have a smaller footprint than the other two options, but only by 2 square millimeters. With so many similarities, it is difficult to find any substantial reason to choose one design over another, so we will simply use the cheapest one with the smallest footprint. This gives us the TPS62825 as our IC of choice. The circuit design for the TPS62825 provided by WEBENCH is shown in Figure 3-22.

Final Choice: TPS62825

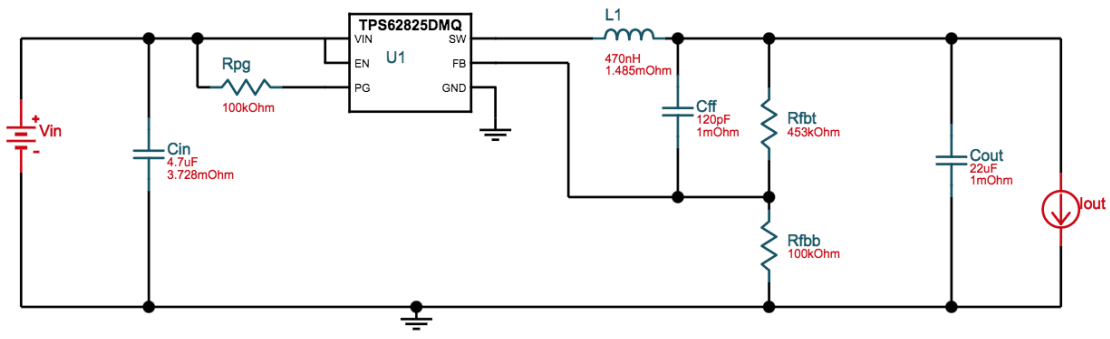


Figure 3-22: TPS62825 circuit design

5V to 48V DC-DC converter

We will also utilize the TI WEBENCH power designer online tool to design our 5 to 48V DC-DC converter. For this boost converter we require a 10 mA output current because 48V phantom power requires a low current. When looking at the DC-DC converter chips that were suggested by TI we compared them using the same parameters we used to compare for the buck converter. Some ICs that were implemented in the designs that fulfilled our voltage and current requirements include the LM3478, LM3488, TPS61390, LM2587-ADJ, and TPS40210.

The LM3478 and LM3488 have approximately all the same parameters, the only difference being slight change in frequencies. For comparison of the possible DC-DC converter chip we will use, we left out the LM3488 since there is virtually no difference between it and the LM3478 chip. A table with a comparison between the ICs can be seen below. The TPS40210 has an unspecified BOM Cost and Footprint because one of the parts used for the DC-DC converter circuit has an unavailable cost and footprint. Adding up the other quantities from the other parts on the BOM we could estimate these values to be at least 3.84$ and 249mm^2. A table with a comparison between the ICs is shown in Table 3-15.

	LM3478	TPS61390	LM2587-ADJ	TPS40210
IC Cost	$0.73	$1.32	$3.82	$0.66
BOM Cost	$1.77	$1.68	$5.42	>$3.84
BOM Count	16	21	9	21
Max Output current	20 A	0.04 A	5 A	20 A
Input Voltage Range	Min: 2.97 V Max: 40 V	Min: 2.5 V Max: 40 V	Min: 4 V Max: 40 V	Min: 4.5 V Max: 25 V
Output Voltage Range	Min:1.27 V Max: 300 V	Min:20 V Max:85 V	Min: 4 V Max: 60 V	Min: 5 V Max: 300 V
Efficiency	81.6%	70.4%	76.7%	78.6%
Short Circuit Protection	Yes	Yes	Yes	Yes
Footprint	470 mm^2	97 mm^2	674 mm^2	>249mm^2
Frequency	541.73 kHz	700 kHz	100 kHz	272.12 kHz

Table 3-15: Comparison of 5V-to-48V DC-DC converters.

For the purposes of powering our device we are looking for low cost, high efficiency and low footprint. The input voltage ranges and output voltage ranges for all the ICs are satisfactory for our device. While none of the BOM costs are out of our budget, the circuit with the LM2587-ADJ had the greatest cost by far while also having the largest footprint. This circuit also had the second lowest efficiency, making it undesirable for us to use. All the boost converters include short circuit protection like the buck converters looked at previously. This chip did have the lowest frequency, but the frequency of the chip should not matter much for the construction of our device. The uncertainty from not knowing the exact cost of the TPS40210 made us withdraw it from consideration considering it did not lead in any of the categories that we were using for comparison. This left us with the LM3488 and the TPS61390 for consideration for our 5-to-48 volt DC-DC converter. The advantage of the LM3488 is that it has the highest efficiency but with the drawback of a larger footprint. The advantage of the TPS61390 is that it has the smallest footprint of all the circuits. Both ICs have similar low costs. The problem with the TPS61390 is that it has the lowest efficiency in the table. We decided that the greater efficiency of the LM3478 outweighed its larger footprint for use in our device and that the low efficiency of the TPS61390 was not worth using for the smaller footprint. The circuit design provided by WEBENCH is shown in Figure 3-23.

Final Choice: LM3478

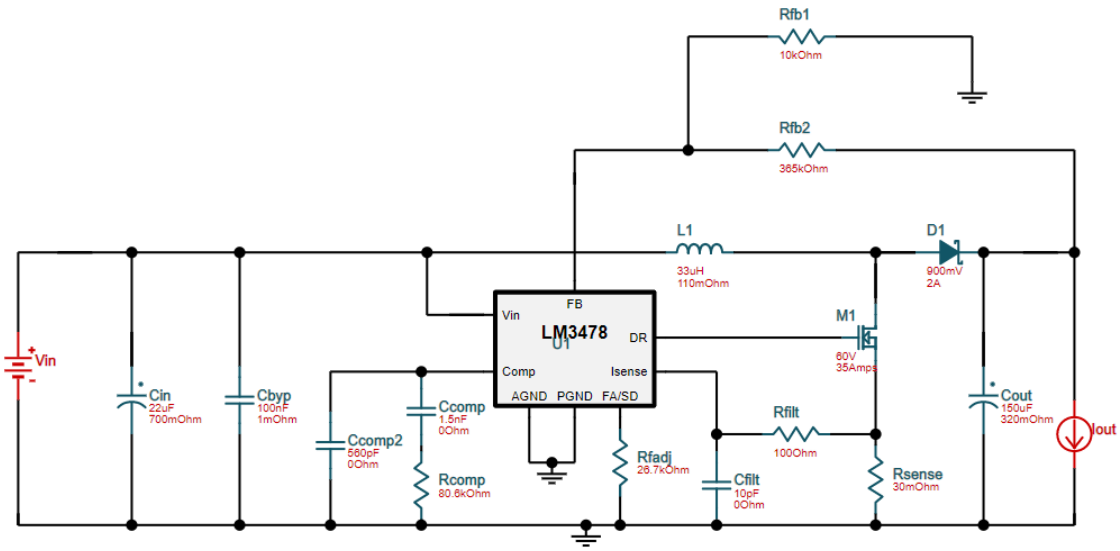


Figure 3-23. LM3478 Circuit Design

3.3.4. USB Controller

Initially, we had planned to incorporate USB support for MIDI to USB functionality. Unfortunately, due to time constraints we were unable to implement a USB driver that would allow this. The USB controller is, however, present in our schematic and board layout as it was cut late in development. Also, the specific MCU chosen in this section would not have worked with the current design, but its low-power counterpart, the PIC16LF1454, would work perfectly with some small adjustments.

We need to use an external USB controller because the processor we selected does not have USB support. While most USB controllers exist as an MCU that can communicate through USB, there are simpler systems called USB transceivers that allow an MCU to transmit data through a USB connection. We considered using a USB transceiver in the device, however upon further investigation we discovered that the MCU we chose would be unable to process USB data transmission and audio signal processing at the same time. Typically, the USB interface would not be utilized often because MIDI is a very low data rate protocol. However, when it is being utilized, the MCU will be stuck sending or receiving data through the USB interface instead of converting the audio signal into notes. Thus, we must use another MCU to handle USB data transmission. Table X shows a list of MCUs with USB support available. They all support USB 2.0 full-speed transmission and all the required transport-layer features to implement a USB-MIDI port. (See Section 4.1.5: Application Layer (USB-MIDI) for details on what features are required to implement a USB-MIDI device.) The Silicon Labs EFM8UB1 has the highest RAM space out of all these MCUs, but it also has the highest power usage. The Cypress CY7C64346 is the worst choice out of all since it has high power usage, the lowest clock rate, and the highest total cost. The Microchip PIC16F1454 is the best choice since it has the lowest power usage and the lowest total cost. Although it only has 1 kB of RAM, this is more than enough to store the MIDI data stream that the MCU will be producing.

USB Controller	**Cypress CY7C64346**	**SiLabs EFM8UB1**	**Microchip PIC16F1454**
V_{in} (V)	3.3 / 5	3.3 / 5	3.3 / 5
Max. Current (mA)	7.1	10.1	1.7
Pin Count	32	20	14
ISA	M8C	CIP-51 (8051)	PIC
Clock Rate (MHz)	24	48	48
ROM (kB)	32	8	8
RAM (kB)	1	2.25	1
Serial Interfaces	I2C, SPI	UART, I2C, SPI	UART, I2C, SPI
Cost (USD)	1.38	0.81	1.31
Programmer	PsoC MiniProg3	EFM8UB1-SLSTK2000A	Curiosity LPC Development Board
Prgm. Cost (USD)	91.76	29.99	26.99
Total Cost (USD)	98.90	32.42	30.92
Datasheet	https://www.cypress.com/file/138721/download	https://www.mouser.com/datasheet/2/368/efm8ub1-datasheet-	https://www.mouser.com/datasheet/2/268/40001639B-

		1666264.pdf	597000.pdf

Table X: List of USB MCUs.

Final choice: Microchip PIC16F1454

3.3.5. Input and Output Ports

There are six ports on the device for the analog inputs and outputs, MIDI, 9V power, and USB. There will be two ¼" jacks and two XLR3 jacks for instrument and microphone cables, a five-pin jack for MIDI signal, a barrel jack for the 9V power and a USB port for USB 2.0 data and 5V power.

Analog Input and Output Ports Selection

In order to make our prototype as easy to build as possible, we are deciding to attach the analog input and output ports to the case of the device and cable them to the board. We will solder wires to the terminals on the ports and use simple solder pads or through holes to solder the wires to the board. With this implementation in mind, we have a few options for these jacks. A simple search on Digikey yields the options for ¼" in Table 3-17.

Part Number	12A	112AX	4833.2230
Manufacturer	Switchcraft	Switchcraft	Schurter
Price per 1 part (DigiKey)	$3.18	$2.68	$2.05
Voltage Rating	Mono	Mono	Mono
Manufacturer Datasheet Link	http://www.switchcraft.com/Drawings/12A_CD.pdf	http://www.switchcraft.com/Drawings/110x-m110x_series_cd.pdf	https://us.schurter.com/bundles/sncescschurter/epim/_ProdPool_/newDS/en/typ_4833.2230.pdf

Table 3-17: ¼" Jack Comparison

These options are all mono audio jacks that can be mounted to the sides of the device and wired into the board. There are only a couple of things that distinguish them from one another. First is the form factor of the jack, with the 12A being small with three metal terminals and the 112AX and 4833.2230 having plastic cases with three and four metal terminals, respectively. The number of terminals is also a difference as the Schurter is the only jack with four terminals, two for tip and two for shield, whereas the Switchcraft jacks both have three terminals, two for tip and one for shield. Only two connections are necessary as these are all mono jacks. The terminals on the Switchcraft options are made with a hole in them for ease of soldering wire to them. The terminals on the Schurter jack are pins that are designed to be placed directly into through holes on the PCB which may

make soldering wires to those terminals harder. Considering all these factors, the best choice is the Switchcraft 112AX. The design of its terminals to facilitate easy wire soldering and its lower cost compared to the Switchcraft 12A make it the most attractive option.

Final Choice: Switchcraft 112AX

Initially, we had planned to incorporate XLR3-compatability into our design so that our device could accept microphone inputs and output a balanced XLR3 output for use with direct-in consoles and mixers. These features were cut in order to focus on the electric guitar as our primary input signal source for the design of the note detection algorithm and due to lack of time for implementing the hardware required to use XLR balanced audio signals.

A decision must also be made as to which XLR3 jacks we will use in our design. These will also be panel-mount connectors attached to the sides of the device and we will solder wires to connect them to pads or through holes on the PCB. A Digikey search for panel-mount XLR3 connectors yields several results. A selection of these results is shown in Table 3-18.

Part Number	IO-XLR3-F-BK-JL	IO-XLR3-F-EV	XLR331F77
Manufacturer	IO Audio Technologies	IO Audio Technologies	ITT Cannon, LLC
Price per 1 part (DigiKey)	$4.13	$2.19	$17.28
Voltage Rating	125VAC	50VAC	133VAC
Current Rating	15A	6A	3A
Manufacturer Datasheet	https://ioaudiotech.com/datasheet/IO-XLR3-X-BK-JL.pdf	https://ioaudiotech.com/datasheet/IO-XLR3-X-EV.pdf	https://ittcannon.com/Core/medialibrary/ITTCannon/website/Literature/Catalogs-Brochures/ITT-Cannon-AudioXL-Catalog.pdf

Table 3-18: XLR3 Port Selection

Most of the options available on Digikey are by IO Audio Technologies, but for the sake of completeness the ITT Cannon XLR331F77 will also be considered. The power ratings of all three options exceed values that we would require for the audio signals passing through them. The two IO Audio Technologies options in table 5.2.1b are from the same line of parts, the main difference being the intended implementation of the two parts. The “BK-JL” jack is almost twice the price of the “F-EV” jack, though it is the part that is intended to be used for our implementation of soldering wires to the terminals on the plug and connecting them to solder pads or through holes on the PCB. The ITT Cannon jack is also designed for this kind of implementation, however its price is over four times that of the

"BK-JL" IO Audio plug. Being that the IO-XLR3-F-BK-JL is reasonably priced and designed with our implementation in mind, it seems like the reasonable choice.

Final Choice: IO-XLR3-F-BK-JL

Digital and Power Jacks Selection

The three remaining ports are for a 9V DC barrel jack, a five pin MIDI port, and a USB port for programming the processor and sending MIDI signals. All three of these ports will be mounted directly onto the PCB and will be accessible through the back panel of the device.

Our USB port, used to provide 5V DC power to the device and transmit MIDI data, will comply with the USB 2.0 standard, but not the USB 3.0 standard. Because USB 3.0 ports are visibly different from USB 2.0 ports, we will have to select a USB 2.0 port to use in our design. In music technology, both USB A and USB B ports are common with micro A ports being used frequently to power devices and transmit data and B ports being used frequently in MIDI devices and USB microphones. A quick Digikey search for right angle SMT and through hole USB 2.0 connectors yields many results to choose from. These results were sorted by price and availability and three viable options have been selected, listed in Table 3-19.

Part Number	UJ2-BH-1-TH	USB-A-S-RA	UE27AC5410H
Manufacturer	CUI Devices	Adam Tech	Amphenol ICC
Price	$0.54	$0.55	$0.58
Voltage Rating	30VAC	30VAC	30VAC
Current Rating	1A	1.5A	1A / contact
Connector Type	USB-B	USB-A	USB-A
Datasheet	https://www.cuidevices.com/product/resource/uj2-bh-th.pdf	http://www.adam-tech.com/downloader.php?p=USB-A-S-RA.pdf	https://signin.amphenolcanada.com/ProductSearch/drawings/AC/UE27ACX4X0X.pdf

Table 3-19: USB Port Selection

All three of these options are solid as far as voltage and current ratings; adherence to the USB standard ensures this. They are also all right angle through-hole connectors, which is fine for our form factor requirements. The only meaningful differences are the fact that the UJ2-BH-1-TH is a B-type connector with the other two being A-type connectors and the small difference in price between the three. Not only is the UJ2-BH-1-TH the cheapest part, but it is a USB-B connector, which is the most common type for MIDI and other audio devices. For this reason, the UJ2-BH-1-TH is our USB connector of choice.

Final Choice: UJ2-BH-1-TH

The MIDI port is a circular, five-pin, female connector with the holes arranged in a semicircle. We found two suitable options from DigiKey and elsewhere. They are both board-mount and angled 90 degrees which is ideal. The options are listed in Table 3-20.

Part Number	SDS-50J	TEDIN-D501
Manufacturer	CUI Devices	Tayda Electronics
Price	$2.63	$0.32
Voltage Rating	100VAC	100VAC
Current Rating	2A	2A
Datasheet	https://www.cuidevices.com/product/resource/sds-j.pdf	https://www.taydaelectronics.com/datasheets/files/A-1010.pdf

Table 3-20: MIDI Port Selection

These two parts are identical, functionally. Even their PCB footprints are both approximately 15mm by 15mm. Their voltage and current ratings are identical. The only major difference is their price, where the CUI Devices SDS-50J is nearly nine times the price of the Tayda Electronics TEDIN-D501. Due to the price difference, we would have selected the TEDIN-D501 to be our MIDI port, but it was unavailable at the time of purchase, so we selected the SDS-50J instead.

Final Choice: SDS-50J

Finally, a barrel jack must be selected for our 9V power supply. For the most part, any barrel jack is suitable. It is a simple connector that is hard to get wrong. The form factor we are looking for is a right-angle connector that will mount directly onto the PCB. A quick DigiKey search yields over 200 options, all of which are basically the same. We have selected four options from four different suppliers to compare in table 5.2.2c. All these options surpass the voltage and current requirements that we have of 9V DC and 0.5 A max. They all have similar form factors, though the metal shield on the outside of the 54-00127 may be visible to the user from the panel on the side of the device which is a small concern.

Part Number	PJ-037A	EJ508A	RASM722X	54-00127
Manufacturer	CUI Devices	MPD (Memory Protection Devices)	Switchcraft Inc.	Tensility International Corp
Price	$0.58	$1.27	$1.77	$0.82
Voltage Rating	24VDC	12VDC	250VAC	48VDC

Current Rating	2.5A	5A	5A	6A
Datasheet	https://www.cuidevices.com/product/resource/pj-037a.pdf	https://www.memoryprotectiondevices.com/datasheets/EJ508A-datasheet.pdf	http://www.switchcraft.com/Specification.aspx?Parent=581	http://www.tensility.com/pdffiles/54-00127.pdf

Table 3-21: 9V DC Barrel Jack Selection

One possible user error to account for is that the user may plug in a power supply that is the incorrect voltage. If this happens, we want to be able to protect as much of the device as we can, including the power jack itself. We are using a 9V power supply in our design because it is by far the most common voltage level for power supplies used in music technology (i.e. guitar pedals or preamplifiers) but 12V and 18V supplies are also somewhat common to the point where power supplies made for music devices such as the MXR DC ISO-BRICK or the Voodoo Lab Pedal Power 2 have 18V jacks or toggle switches built in. For this reason, the EJ508A is not a suitable choice, as it is not rated to withstand voltages greater than 12V and may be damaged if the user accidentally plugs in an 18V supply.

The remaining two options are the PJ-037A and RASM722X. They are both good options in both form factor and ratings, so price will be the final determining factor. The PJ-037A is cheaper at $0.58, roughly a third of the price of the RASM722X. For this reason, the PJ-037A is our barrel jack of choice.

Final Choice: PJ-037A

3.3.6. Power Multiplexing ICs

Some potential ICs that we could use for power multiplexing are the LM74700, TPS2419, LTC4236, and the LTC4411. The LM74700 is the cheapest of all the ICs but after looking into its datasheet we found that this IC is more well suited for circuits with higher current and power needs. Both the LM74700 and LTC4236 are TI products and do not provide an easy way to simulate the designs without a CAD PSPICE simulator. The other two ICs, LTC4236 and LTC4411 are both produced by Linear Technologies which provides a free circuit simulator in LTSpice which has all the LT ICs programmed in already. The LTC4236 has the most expensive cost and looks to be the most complicated to design for because of its 28 pins. This many pins on the IC seems unnecessary for our power switching circuit. After looking at the parameters for each of these ICs, we concluded that using the LTC4411 would be the best for our power select circuit. The LTC4411 is on the more expensive side of the ICs but is the smallest in footprint. The LTC4411 significantly lowers the voltage drop to 28mV. It has a 2.6 to 5.5V operating range which is perfect for our 5V inputs.

	LM74700	TPS2419	LTC4236	LTC4411

IC cost	0.62$	1.02$	10.60$	4.14$
Vin	Min: 4.2V Max: 40V	Min: 0.8V Max: 16.5V	Min: 2.9V Max: 18V	Min: 2.6V Max: 5.5V
Iq (Typ)	0.3 mA	1.2 mA	2.7	10 uA
Iq (Max)	0.4 mA	7 mA	4 mA	N/A
Footprint	2.9 x 1.6 mm	19 mm^2	4.00 x 5.00 mm	1 mm
Features	Integrated FET, Analog Current Monitor, Adjustable Current Limit, ON/OFF Control	ON/OFF Control	Ideal Diode-OR and In-Rush Current Control for Redundant Supplies, Low Loss Replacement for Power Schottky Diodes.	Low loss replacement ORing Diodes, Small regulated Forward Voltage (28mV),

Table 3-22: Comparison of Power Switching ICs

Final Choice: LTC4411

4. Related Standards and Constraints

4.1. Standards

In order to create a product that is compatible with the greatest number of other MIDI devices and MIDI-compatible interfaces and computers, we will adhere to the standards shown in Figure 4-1.

Standard Name	**Description**
USB 2.0	Communication with computer, +5V power in.
MIDI 1.0	Defines physical layer and transmission protocol for musical data.
XLR3	Microphone in, signal out.
Phantom Power	Power for condenser microphones.
TS ¼" Connector	Tip + Sleeve connector for instrument cable.

Sleeve+ Tip/Center- 2.1mm Barrel Jack	+9V power jack.
Serial Wire Debug	Debugging interface for microcontrollers.

Table 4-1: Table of standards.

4.1.1. ¼-Inch Audio Jack Port and Connector

The ¼ jack connector is one of the most common connectors used on musical devices. This jack is a quarter inch or 6.3mm and is the connection usually used for electric guitars. A common application for our device will be converting analog audio from an electric guitar to a MIDI stream so it is necessary for our device to have a ¼ jack input. We will also have a ¼" output jack for the bypassed instrument signal, so that the instrument can still be connected to an amplifier, PA system, or direct input box. The ¼" jack we will be using will follow the tip-sleeve (TS) standard rather than the tip-ring-sleeve (TRS) or tip-ring-ring-sleeve (TRRS) standards. This is because TS is used for any normal guitar or most other electric instruments that produce a mono (not stereo) signal. TRS and TRRS are mainly used for headphones or mixers.

4.1.2. XLR3 Audio Port and Connector

The XLR3 connector is a 3-pin connector that is the most common style XLR connector used for audio. Most professional microphones use a XLR3 connector. This makes it important for us to implement a XLR3 input into our device because the device needs to be able to connect to good microphones to be able to pick up voice and other musical instruments that are not electric guitars. Since XLR3 is the one of the most common connectors used in audio, the device will also have a XLR3 output.

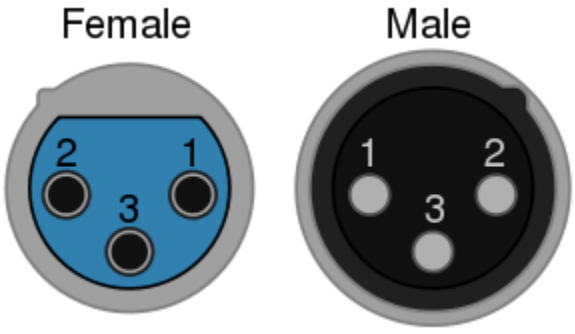


Figure 4-2: XLR 3 Connector Diagram. (Made by Omegatron under a Creative Commons License)

4.1.3. 48-Volt Phantom Power

Initially, we had planned to make this device compatible with condenser microphones by implementing XLR connectors and phantom power. Unfortunately, due to several factors we decided to focus mainly on the core functionality of the device and cut this along with several other peripheral features.

For the Polyphonic Analog to MIDI Converter to support all types of microphones, we need to incorporate 48 Volt phantom power into our design. Phantom power is a means of powering condenser microphones without the use of an external power supply. This lack of dedicated power supply unit is where the “phantom” name comes from. Instead, power is carried along the same lines as signal. The implementation of phantom power is straightforward. A simple voltage divider using 6.8k Ohm resistors delivers power from the 48 V DC supply to the microphone’s preamp via the signal pins of its XLR3 jack. When the mic signal + phantom power reaches the mixer input, coupling capacitors are used to remove the DC component. The remaining microphone signal is sent to an op-amp which combines the two differential signals into a single signal. This implementation is essentially an industry standard and a schematic can be seen in Figure 4-2. Our greatest concerns regarding the design of our phantom power circuit are the reliability of the step-up DC converter to turn 5 volts into 48 volts and the ability to maintain a steady 5mA to not deliver too much power and damage the microphone.

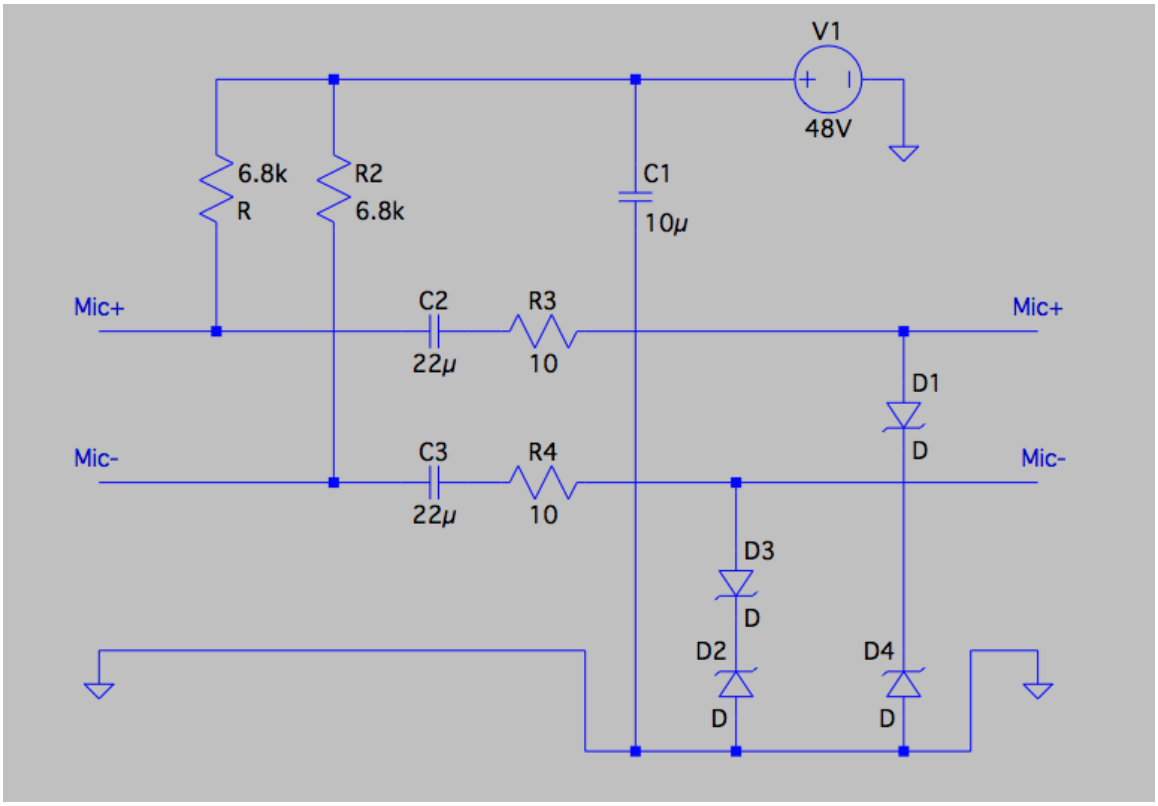


Figure 4-2: Phantom Power standard circuit.

4.1.4. Musical Instrument Digital Interface (MIDI)

Physical and Link Layers

MIDI is a very simple interface to implement with just a digital output pin and some passive and active components. MIDI uses asynchronous data transmission. MIDI is transmitted as bytes at 31.25 kilobits per second. The interface uses a start bit, 8 data bits and a stop

bit. This means for each serial byte there are a total of 10 bits that are sent for a period of 320 microseconds. MIDI has this low data transfer rate because MIDI usually only needs to send basic instructions of which MIDI notes to play and changing its timing, velocity etc.

Application Layer

There are only 16 messages (based on the status byte) defined in MIDI. They are grouped into five categories: Channel Voice, Channel Mode, System Real-Time, System Common, and System Exclusive. The last three sets of messages generally do not apply to this device because they are used by sequencers and/or devices that have custom messages. Every message begins with a status byte and ends with at least zero data bytes, except for System-Exclusive messages, which append an End-of-Exclusive status byte. The values of every status and data byte are 7-bit values. However, the most significant bit of each status byte is always 1, while for each data byte it is always 0, giving a total of 8 bits and differentiating the purpose of each byte. A status byte contains a 3-bit code specifying the type of message and a 4-bit number used to address the channel that the message applies to. This does not apply to the System messages, which all have the code Fh, since the 4-bit number is used instead to specify the functionality of the message. There are 7 different Channel messages. They all have a maximum of two data bytes. Table 4-2 shows their formats.

Name	Code	Data 0	Data 1
Note Off	8h	Note number	Note velocity
Note On	9h	Note number	Note velocity
Polyphonic Key Pressure	Ah	Note number	Note pressure
Controller Change	Bh	Controller ID	Controller value
Program Change	Ch	Program number	N/A
Channel Key Pressure	Dh	Channel pressure	N/A
Pitch Bend	Eh	Pitch change (LSB)	Pitch change (MSB)

Table 4-2: All Channel messages

Control messages are sent by the MIDI controller to adjust the various parameters that affect the notes being played by the controlled instrument. These control signals are for varying the effects that are applied to the note, adjusting volume, and ending notes among other things. Control messages consist of a status byte and two data bytes which are the type and value (0 to 127) of the control adjustment. Table 4-3 enumerates the IDs of all control types and describes their function. [25]

ID	Name	Function
00h	Bank Select	Used for switching between instrument sets,

		expanding the range of the Program Change message
01h	Modulation Wheel	Expression value used to apply some effect to the sound with controller
02h	Breath Controller	Expression value used for Breath Controller peripheral devices to play MIDI as a wind instrument, can also be used by other controllers
04h	Foot Controller	Expression value used for foot pedal controller, can send continuous stream of signals
05h	Portamento Time	Controls speed at which one note slides into another
06h	Data Entry	Controls NRPN or RPN parameter values
07h	Channel Volume	Controls volume
08h	Channel Balance	Left and right balance for stereo
0Ah	Channel Pan	Left and right balance for mono
0Bh	Expression	This is a percentage of volume
0Ch	Effect Control 1	Controls a parameter of some effect
0Dh	Effect Control 2	Controls a parameter of some effect
10h-13h	General-Purpose	
20h-3Fh	Controllers' LSB	For controllers 00h-1Fh. Expands their precision to 16-bits.
40h	Damper Pedal	Sustain toggle
41h	Portamento Switch	Portamento toggle
42h	Sostenuto	Sostenuto toggle
43h	Soft Pedal	Soft/Hard note toggle
44h	Legato footswitch	Legato toggle
45h	Hold 2	Used to hold notes
46h	Sound Variation	Controls how sound is made

47h	Harmonic Intensity	Shaping of voltage controlled filter
48h	Release Time	Control release of voltage controlled amplifier
49h	Attack Time	Controls time it takes for sound to reach maximum amplitude
4Ah	Brightness	Cutoff frequency of voltage controlled filter
4Bh-4Fh	Other Sound Controllers	
50h-53h	General-Purpose	
54h	Portamento Control	Controls amount of portamento
5Bh	External Effects Depth	Controls some effect, usually reverb
5Ch	Tremolo Depth	Controls tremolo or other effect
5Dh	Chorus Depth	Controls chorus or other effect
5Eh	Detune Depth	Controls detune or other effect
5Fh	Phaser Depth	Controls phaser or other effect
60h	Data Increment	Increments data for RPN and NRPN
61h	Data Decrement	Decrements data for RPN and NRPN
62h	Nonstandard Parameter Number (LSB)	Sets NRPN parameter
63h	Nonstandard Parameter Number (MSB)	Sets NRPN parameter
64h	Standard Parameter Number (LSB)	Sets RPN parameter
65h	Standard Parameter Number (MSB)	Sets RPN parameter
78h	Panic	Mutes all notes regardless of timing or sustain
79h	Reset All	Resets all controllers to default
7Ah	Keyboard Connect	Toggles connection of keyboard or workstation
7Bh	All Notes Off	Mutes all notes but not those affected by sustain

7Ch	Omni-Mode Off	Set omni-mode off
7Dh	Omni-Mode On	Set omni-mode on
7Eh	Monophonic Mode	One note at a time
7Fh	Polyphonic Mode	Several notes at once

Table 4-3: Enumeration of controllers

4.1.5. Universal Serial Bus (USB)

USB is a standard that is used very commonly in various electronic devices and applications. Its main purpose is to standardize the way separate devices connect to and communicate with a computer. Since we will be using USB as a MIDI stream output and as a configuration input, it is very important to understand the USB standard and the protocols it defines. All information about USB in this section is taken directly from the document on the USB 2.0 specification, called “Universal Serial Bus Specification”, revision 2.0. Only information relevant to this project is included. USB defines protocols for every layer of the 5-layer TCP/IP networking stack, from the physical layer to the application layer. The topic of USB is split by layer. In summary, USB is a serial bus architecture with one bus master and several slaves arranged in a tiered star topology. The bus master is the computer, and it initiates all transactions. The bus slaves cannot manipulate the bus unless requested to by the computer. A bus slave can behave as a hub, which connects the device it serves to all other devices connected to it. Hubs make up the structure of the bus system.

Physical Layer

The physical layer describes the physical hardware on which data is sent between two different systems. USB defines the physical layer of the connection. It includes specifications for ports, connectors, cables, and wires. For USB 2.0, there are two different attributes for ports, type and size. There are two different types of ports: A and B. Devices with A ports can have devices under them in the hierarchy; this applies to computers and USB hubs. Devices with B ports connect to a device above them in the hierarchy; this applies to peripheral devices like storage drives and keyboards. This port type system ensures that devices are connected in the correct orientation. There are three different sizes of ports: normal, mini, and micro. The only purpose of the different sizes is for fitting in different size constraints, except for the mini and micro ports, which also support USB On-the-Go (OTG) functionality. Each port has four to five pins, with four of them being common to all ports. The VBUS pin is used to supply 5 volts power to the peripheral device. There is a limit to how much power a peripheral device can use: 100 mA for low-power devices, 500 mA for high-power devices. The amount of power supplied by the computer is negotiated after the device is connected; by default, it is low.

USB 2.0 defines three speeds for data transmission across the connection: low-speed, full-speed, and high-speed. Table USB shows the data transfer rates of the different USB speeds. When devices are connected to a port that supports a different speed, the highest common speed among both is used. The speed also affects the operation of other layers

of the USB specification. While full-speed and high-speed devices may use a separate cable to connect their USB ports, a low-speed device must either hardwire its own cable and connector within the device or use a custom cable with a custom device-side connector that meets the USB specifications for a low-speed cable. This device will be using full-speed communication, so this restriction does not apply to the device.

Link Layer

The link layer is responsible for transmitting binary data between two ends of a physical link. USB also defines the link layer protocol; it specifies the data rate, voltage ranges, and bit encodings on the physical wires. The discussion on the link layer will focus on that for full speed devices since our device will operate at full speed. Full-speed connections have a clock cycle speed of 12 MHz, thus one cycle lasts 83.3 ns. Since there is no clock signal transmitted through the connection, this is the maximum rate at which the wire states can change. The jitter time is the maximum deviation in the length of a multiple of cycles and is measured in between state transitions. For full-speed connections, the jitter time between consecutive differential data transitions (from J to K or K to J) must be within ± 2.0 ns and within ± 1.0 ns for paired differential data transitions (JK to JK or KJ to KJ). To mark the device as a full-speed device, the D+ line has a pull-up resistor of 1.5 kΩ connected to 3.3 V sourced from VBUS.

Each line can have a high-voltage and low-voltage state. The high-voltage state is 3.3 V and the low-voltage state is 0 V. Together they have 4 possible states: Differential 1, Differential 0, Single-ended 0 (SE0), and Single-ended 1 (SE1). The SE1 state is an invalid state. The actual data and bus status states are defined using these wire states. The two data states are the J and K state, where the J state represents a logic 1. These states are used to send raw binary data across the link. The J state for full-speed devices is a Differential 1. The K state for full-speed devices is a Differential 0. The bus status states are the Idle, Start-of-Packet (SOP), End-of-Packet (EOP), Reset, Suspend, and Resume states. The Idle state is when neither line is being driven, which is equivalent to the J state because of the pull-up resistor. The SOP state is a transition from the Idle state to the K state, and it is used to mark the start of a packet transfer. The EOP state is when the wires are at the SE0 state for about two cycles followed by the J state for one cycle. This state is used to mark the end of a packet transfer. The SE0 for the EOP state must last for between 160 ns and 175 ns for full-speed connections, but it may be as short as 82 ns. The device must wait for between 2 and 6.5 cycles before manipulating the connection after the EOP to allow the host's bus drivers to be turned off. The Reset state is when the wires are at the SE0 state for at least 10 ms, but the device can treat the SE0 state as a Reset state if it lasts for at least 2.5 µs. This state tells the device that the connection state is reset and that it should prepare to receive information about the host and introduce itself to it. The Suspend state is an Idle state that lasts for more than 3 ms. This state suspends the device, which limits the maximum amount of current draw from VBUS to 500 uA. The Resume state is any state that is not an Idle state, which is typically the K state. This applies only when the device is suspended, and it is used to bring the device out of suspension. This state must last for at least 20 ms and must end with the EOP state. Table 4-4 lists these states and their properties.

Name	Definition

Line States	
High	~3.3 V
Low	0 V
Wire-Pair States	
Differential 0	D+: High & D-: Low
Differential 1	D+: Low & D-: High
SE0	D+: Low & D-: Low
SE1	D+: High & D-: High
Data Logic States	
J	Differential 1
K	Differential 0
Bus Status States	
Idle	J state & wires not explicitly driven.
SOP	Transition from J to K state.
EOP	SE0 for ~2 cycles, J state for 1 cycle.
Reset	SE0 for at least 2.5 µs.
Suspend	Idle state for more than 3 ms.
Resume	Transition from Suspend state to other state.

Table 4-4: Different states of the full-speed USB connection at different levels.

Binary data is transmitted in sets called packets; they are transmitted only in between the SOP and EOP states using J and K states. Bits are encoded using the NRZI format, where a 1 is represented by no change in state and a 0 is represented by a change in state. A 0 is added after six consecutive 1s before the following bits are sent. This is to maintain synchronization even when several 1s are being transmitted. Thus, when receiving data, a 0 after six 1s should be ignored. If seven 1s are received, then the packet is corrupt and must be discarded. Each bit in a byte is sent with the least-significant bit first and the most-significant bit last. Each multi-byte datum is sent in little-endian order. Each packet is preceded by an 8-bit SYNC pattern. The SOP state is the first bit transmitted of the SYNC pattern. For full-speed connections, the pattern is three pairs of K and J states with 2 K states at the end (i.e. 10000001).

Network Layer

The network layer is responsible for the proper transmission of data across several different network nodes. Essentially, it establishes a system for sending and receiving data to and from the correct destination and source through multiple different devices. Since our device communicates only with the host computer, the network layer implementation is very simple. However, the host has multiple devices connected to it, and it must be able to send packets to the correct device. When a device is first connected to a USB port, its address is the default address of 0. The host assigns the device an address when the host discovers it. The device should only respond to requests whose address matches its own. In the hierarchy of USB devices, the host is always at the top. Below it are the USB hubs and peripheral devices. Each node of the hierarchy can connect to a hub or device below it, except for peripheral devices, which can only connect to a higher-level node. This hierarchy is limited to seven levels, including the host's level. Each node has its own unique address.

Transport Layer

The transport layer manages data transmission between connected devices. It provides the backbone for application-specific data transfer by addressing the correct application that data must be transmitted to or from and organizing that data transfer. To ensure that data is transferred to the correct application, USB uses a system of endpoints. Endpoints split data that is transferred through USB into logical data streams. Each full-speed USB device has a set of input and output endpoints and a maximum of 16 of each. Thus, an input transaction and an output transaction on endpoints with the same endpoint number are transactions on two different endpoints. Each endpoint has an associated number and specific properties that describe its functionality. Some properties of an endpoint are the required bandwidth, maximum latency, error-handling requirements, maximum packet size, and transaction type. By default, there is always a pair of input and output endpoints with endpoint number 0. This is called the Default Control Pipe, and it is primarily used for USB-defined Control transactions.

To organize data transfer between the host and the device, USB defines sixteen packets that may be transmitted between them. They are split into four categories: Token, Data, Handshake, and Special. Token packets mark the purpose of a transaction and the device being accessed. Data packets carry only binary data. Handshake packets inform the host or device of the result of a transaction. Special packets have special purposes. Table X shows the packet IDs (PIDs) of each packet and its function. The functions are only described in relation to the device.

Category	Name	PID	Function
Token	OUT	1h	Tells the device that it must receive a data packet.
	IN	9h	Tells the device that it must send a data packet.
	SOF	5h	Marks the start of a frame with a new frame number.
	SETUP	Dh	Used to set an endpoint's synchronization bits.

Data	DATA0	3h	Contains binary data. Even PID version.
	DATA1	Bh	Contains binary data. Odd PID version.
	DATA2	7h	Only used in high-speed connections.
	MDATA	Fh	Only used in high-speed connections.
Handshake	ACK	2h	The received data packet did not have errors.
	NAK	Ah	The device cannot send or receive data.
	STALL	Eh	The endpoint is in an error or halted state.
	NYET	6h	Only used in high-speed connections.
Special	These packets are only used in high-speed connections or between a host and a hub.		

Table 4-5: List of packets and their functions.

There are four Token packets: OUT, IN, SETUP, and SOF. The OUT, IN, and SETUP packets are used to send or receive data to or from the device. All three of these packets have four fields: the PID, the device address, the device endpoint, and the CRC-5 of the address and endpoint fields. All data transactions begin with one of these packets. The OUT packet is always followed by any data packet sent by the host, while the SETUP packet is always followed by a DATA0 packet specifically. The IN packet is always followed by either a data, a NAK, or a STALL packet sent by the device, or no packet if the IN packet contains errors. Table X shows the format of these packets.

Field	PID	Address	Endpoint	CRC5
Size (b)	8	7	4	5
Notes	The CRC5 field contains the CRC-5 of the Address and Endpoint fields.			

Table 4-6: OUT, IN, and SETUP packet format.

The Start-of-Frame (SOF) packet marks the start of a time frame. For full-speed devices, a frame is defined to be 1 ms. Thus, the host will send an SOF packet every millisecond to establish time frames. This will also keep the device from going into the Suspend state. No reply is expected for this packet. Table 4-7 shows the format of the SOF packet. The Data packets all have three fields: the PID, the data payload, and the CRC-16 of the payload. Only the DATA0 and DATA1 packets are used in full-speed communication. The size of the payload is not fixed, but the maximum size for full speed is 1023 bytes. Any of the Data packets must only be sent by the host after an OUT or SETUP packet, and they must only be sent by the device after an IN packet. Data packets may be followed by an

ACK, NAK, or STALL packet sent by the recipient of the transaction, or no packet at all if the Data packet or the OUT packet that preceded it contains errors. However, if the Data packet was preceded by a SETUP packet, the device must reply with an ACK packet, except when the endpoint is not a control endpoint, in which case it does not reply. Table 4-8 shows the format of the Data packets used in full-speed connections.

Field	PID	Frame Number	CRC5
Size (b)	8	11	5
Notes	The CRC5 field contains the CRC-5 of the Frame Number field.		

Table 4-7: SOF packet format.

Field	PID	Payload	CRC16
Size (b)	8	0-1023	16
Notes	The CRC16 field contains the CRC-16 of the Payload field.		

Table 4-8: DATA0 and DATA1 packet format.

There are three Handshake packets used in full-speed connections: ACK, NAK, and STALL. All these packets only have one field: the PID. The ACK packet is used to tell the sender of the Data packet that it was received without errors. The NAK packet is used to tell the sender of the Data or IN packet that the receiver cannot receive data or that it has no data to send, respectively. The NAK packet cannot be used in a transaction that began with a SETUP packet. The STALL packet behaves in the same way as the NAK packet, except it is used when the device is in an error state. This packet informs the host that it must reconfigure the device. Table 4-9 shows the format of all Handshake packets. Together, these packets make up three packet sequences: Output, Input, Setup, and SOF. Certain USB transactions may use several of these sequences for one transaction. Only the host may begin a transaction through USB; the device cannot.

Field	PID
Size (b)	8
Notes	

Table 4-9: ACK, NAK, STALL, and NYET packet format.

There are four types of USB transactions: Bulk, Control, Interrupt, and Isochronous. Bulk transactions are used to transmit large amounts of data without errors. They do not require a maximum latency, and they do not happen on a periodic basis. They consist of only one Output or Input packet sequence for each transaction, except in the case of errors. When transmission errors are detected (because the device did not send a handshake), the transaction is retried. The result of a Bulk transaction is indicated by the handshake that is given by the device or host. A Bulk transaction from the host to the device uses the Output packet sequence. If the handshake is an ACK, then the transaction was successful and the next one may begin. If the handshake is a STALL, then the transaction was not successful because the endpoint is halted, and no further transactions should take place. If the handshake is a NAK or if there is no handshake, then there was an error in the device or in the transmission, respectively, so the host should try the transaction again. A Bulk transaction from the device to the host uses the Input packet sequence. If the handshake is an ACK, then the transaction was successful. If the handshake is a NAK, then the device has no data to send or is not ready to send data. If the handshake is a STALL, then the transaction was not successful because the endpoint is halted, and no further transactions should take place. If there is no handshake either from the host or from the device, then the transaction was not successful and should be retried by the host. Bulk transactions can use either of the two Data packets. The use of these packets alternates between DATA0 and DATA1 for each Bulk transaction on the same endpoint. The order is reset when the endpoint receives a configuration event, so the next Bulk transaction after that must use the DATA0 packet.

Control transactions are used to send commands or transmit configuration or status data. These transactions are not periodic and make sure that data is transmitted without errors. Control transactions have three stages: a setup stage, a data stage, and a status stage. There are three kinds of control transactions: Read, Write, and Dataless. A Control Read transaction consists of one Setup packet sequence for the setup stage, at least one Output packet sequence for the data stage, and one Input packet sequence for the status stage. Both DATA0 and DATA1 packets are used in Control transactions, and they alternate just like those in Bulk transactions. Thus, the first Data packet sequence uses the DATA1 packet, the next Data packet sequence uses the DATA0 packet, and so on. However, in the status stage, the Input packet sequence always uses the DATA1 packet. A Control Write transaction behaves in the exact same way as the Control Read transaction, except the Output and Input packet sequences are swapped in all stages of the transaction. A Dataless Control transaction has no data stage; it consists of only one Setup packet sequence for the setup stage and one Input packet sequence for the status stage. Like the other kinds of Control transactions, the Input packet sequence of the status stage always uses the DATA1 packet. The device may send a STALL packet during the data or status stage of the transaction, in which case it must send STALL packets for all following transactions except the beginning of a Control transaction, upon receiving a SETUP packet. After receiving the SETUP packet, the device is expected to be released from the error state it was in, and should operate normally without sending any STALL packets, unless it enters another error state. The result of the Control transaction is given by the handshake used by the device in the status stage. For Write and Dataless Control transactions, if the transaction is complete, the device sends a DATA1 packet during the Input packet sequence. If the transaction failed, the device sends a STALL packet during the Input packet sequence. If the device is still busy processing the transaction, it sends a NAK packet during the Input packet sequence. For a Read Control transaction, the behavior is

the same, except the case where the transaction is complete. In this case, the device sends an ACK packet during the Output packet sequence. Since a Control transaction can use multiple Input or Output packet sequences in the data stage, there needs to be a method of determining which sequence is the last one. The last sequence is the one where the size of the payload of the Data packet is not the maximum size. If the last sequence does have this property, then another sequence must be sent with the size of the payload being 0.

Interrupt transactions are used to send or receive data that must be handled immediately. They are not periodic, and they behave exactly like bulk transactions. Isochronous transactions are used to send or receive data quickly and in a periodic fashion. They behave like bulk transactions, except no handshakes are transmitted. Neither the Interrupt nor the Isochronous transactions are used in the USB-MIDI specification.

Application Layer (Standard)

The application layer implements required protocols for a certain function. USB also defines an extensible protocol for the application layer, and it also defines protocols for common applications. It defines a set of standard Device Requests for reading and writing device characteristics and settings. Standard Device Requests are sent through the Default Control Pipe, which always exists. They are transported using Control transactions. After the Reset state is released on the bus, the device is in the Default state. It cannot do anything and must first be configured by the host before it becomes useful. The host then makes a Device Request asking for the device descriptor and uses this information to determine the maximum size of the Data packet payload that the device supports. Then, the host assigns the device an address using the SET_ADDRESS Device Request. Then the host makes several other Device Requests to read other descriptors from the device. Finally, it sends a SET_CONFIGURATION Device Request to tell the device that it is configured and is now functional. If the device does not support a certain Device Request, then it must signal an error through the methods described earlier for Control transactions.

Device Requests must be processed within a short amount of time. For standard Device Requests the device must complete the status stage of the transaction within 50 ms of the last packet sequence. The device must send Input packet sequences within 500 ms of the last packet sequence for device-to-host Device Requests, and the device must be able to process all packets received within the time it claims to be able to process them for host-to-device requests. Table X shows the format of a standard Device Request. A standard Device Request has five fields: bmRequestType, bRequest, wValue, wIndex, and wLength. The bmRequestType field lists the characteristics of the request: the data transfer direction, the specification level of the request, and the target of the request. The direction establishes whether it involves a Read or Write Control transaction. The specification level marks who defines this request; it could be defined as a standard request, a device class-specific request, or a custom request. The target can be the device itself, one of its logical interfaces, one of its endpoints, or something else. The bRequest field contains the ID of the request. This determines the function of the request. The wValue and wIndex fields are generic containers for data that may be used for certain requests. Finally, the wLength field determines how many bytes of data must be transmitted by the host or device in the data stage of the Control transaction.

Offset	Field	Size (B)	Function
00h	bmRequestType	1	Bit 7: Data transfer direction ● 0: Host-to-Device ● 1: Device-to-Host Bits 5-6: Request type ● 0: Standard ● 1: Class ● 2: Vendor ● 3: Reserved Bits 4-0: Recipient ● 0: Device ● 1: Interface ● 2: Endpoint ● 3: Other ● 4-31: Reserved
01h	bRequest	1	ID of the request.
02h	wValue	2	Request value parameter.
04h	wIndex	2	Request index parameter.
06h	wLength	2	Number of bytes to transfer in the data stage.

Table 4-10: Format for standard Device Requests.

USB defines 11 standard Device Requests. Most of them must be supported by all devices. If the host sends a Request that the device does not support, then it must return a STALL packet as described previously for Control transactions. Table X shows the request IDs for the standard USB Device Requests. The GET_STATUS request is used to get the status of a device, interface, or endpoint. This request is specifically a device-to-host request, and it can target the device, an interface, or an endpoint. The wValue field is always 0. If the target is the device, the wIndex field must be 0. Otherwise, the wIndex field is the interface or endpoint number associated with the interface or endpoint being targeted. This request requires that two bytes of data be sent from the device to the host. The format of this data depends on the target. If the target is the device, then the data is a bit field where bit 0 is set to 1 if the device is self-powered and bit 1 is set to 1 if the device is enabled to be able to wake up the host while suspended. All other bits are set to 0. If the target is an interface, then all bits in the data payload are 0. If the target is an endpoint, then the data is a bit field where bit 0 is set to 1 if the endpoint is halted. When the endpoint is halted, no data may be transferred through it, and all transactions will result in a STALL packet being sent by the device. The host can halt and unhalt the endpoint.

Request Name	ID	Function
GET_STATUS	0h	Gets the status of the recipient.
CLEAR_FEATURE	1h	Disables a specific feature.

SET_FEATURE	3h	Enables a specific feature.
SET_ADDRESS	5h	Sets the address of the device.
GET_DESCRIPTOR	6h	Gets a specific descriptor if available.
SET_DESCRIPTOR	7h	Optional. Sets the values of a new or existing descriptor.
GET_CONFIGURATION	8h	Gets the configuration value of the device.
SET_CONFIGURATION	9h	Sets the configuration value of the device.
GET_INTERFACE	Ah	Gets the current setting of a specific interface.
SET_INTERFACE	Bh	Sets an alternative setting for a specific interface.
SYNCH_FRAME	Ch	Sets and gets the frame number of an endpoint.

Table 4-11: Enumeration of IDs for standard Device Requests.

The CLEAR_FEATURE request disables a specific feature of the device, an interface, or an endpoint. This request is specifically a host-to-device request, and it can target the device, an interface, or an endpoint. The wValue field contains the feature ID that the host wants to disable. The wIndex field is 0 if the target is the device. Otherwise, this field contains the interface or endpoint number associated with the interface or endpoint being targeted. There is no data payload that is transferred for this request. The feature ID can be one of several values. Table 4-12 enumerates the available standard features. The ENDPOINT_HALT feature indicates whether an endpoint is halted. Disabling this feature unhalts the endpoint. The DEVICE_REMOTE_WAKEUP feature indicates whether the device can wake-up the host while the connection is in the Suspend state. Disabling this feature tells the device that it must not wake-up the host. The TEST_MODE feature indicates whether the device is in testing mode. This feature is not valid for full-speed devices. The SET_FEATURE request does the opposite of the CLEAR_FEATURE request and has almost the same format, except that the MSB of the wIndex field can contain a test mode ID. This field will only contain a test mode ID when the wValue field is set to the TEST_MODE feature ID.

Feature	ID	Target
ENDPOINT_HALT	0	Endpoint
DEVICE_REMOTE_WAKEUP	1	Device
TEST_MODE	2	Device

Table 4-12: Enumeration of standard features.

The SET_ADDRESS request sets the address of the device. This request is a host-to-device request that targets the device. The wValue field contains the new device address. The

wIndex field is always set to 0. There is no data payload that is transferred for this request. By default, the address of a device is 0. The new address goes into effect after the end of the status stage of this request.

The GET_DESCRIPTOR request gets a descriptor from the device. This request is a device-to-host request, and it can only target the device. The wValue field contains the descriptor type ID in the MSB and the descriptor index in the LSB. The descriptor index is only valid for Configuration and String descriptors since there can be several of them. Otherwise, the descriptor index is set to 0. The wIndex field contains the language ID of the string for String descriptors, otherwise it is set to 0. The data that is sent by the device is the entire descriptor data structure that the host is requesting. Descriptors are data structures that describe either a device, an interface, a string, or something else. These descriptors tell the host what protocols it must enable to communicate with the device and use its functionality. Table 4-13 enumerates the relevant descriptors that are valid for this device. The SET_DESCRIPTOR request is optional and sets the contents of a descriptor on the device. This request is a host-to-device request, and it can only target the device. This request is not used in the USB-MIDI specification.

Name	ID
Device	01h
Configuration	02h
String	03h
Interface	04h
Endpoint	05h

Table 4-13: Enumeration of descriptor IDs.

The GET_CONFIGURATION request gets the current configuration ID of the device. This request is a device-to-host request, and it can only target the device. The wValue and wIndex fields are unused, so they are set to 0. The data that must be transferred is a single byte with the device's current configuration ID. This request is used to determine the device's behavior. A configuration has interfaces and endpoints associated with it, and different configurations may have different interfaces and endpoints. The SET_CONFIGURATION request sets the current configuration ID of the device. It is a host-to-device request that only targets the device. The wValue field contains the new configuration ID, and the wIndex field contains only 0. There is no other data transmitted with this request. This request is used to set the device's behavior to a specific configuration described by the associated Configuration descriptor.

The GET_INTERFACE request gets the current alternative setting of the specified interface. This is a device-to-host request that only targets an interface. The wValue field is 0, and the wIndex field contains the interface number associated with the targeted interface. The data that must be transferred is a single byte with the current alternative setting of the interface. A single interface may have several different Interface descriptors associated

with it; these are called alternative settings. Alternative settings are distinguished by an ID, and several can be associated to one interface. The SET_INTERFACE request sets the current alternative setting of the specified interface. This is a host-to-device request that only targets an interface. The wValue field contains the alternative setting ID of the requested alternative setting, and the wIndex field contains the interface number associated with the targeted interface. There is no other data that is transferred with this request. Setting the alternative setting of an interface changes its properties described by an Interface descriptor to those of another related Interface descriptor. These Interface descriptors are associated with one interface, but are distinguished by the alternative setting ID.

The SYNCH_FRAME request sets the frame number for an endpoint and gets the frame number for that endpoint. This is a device-to-host request that targets only an endpoint. The wValue field is 0, and the wIndex field contains the endpoint number associated with the targeted endpoint. The data that is transferred from the device to the host is a 16-bit number containing the current frame number of the endpoint.

The descriptors that are transferred in the GET_DESCRIPTOR request each have a specified format and purpose. The Device descriptor contains information about the device including the level of USB support, the device's functionality, the manufacturer of the device, and the total number of configurations that it supports. Table 4-14 shows the format of the Device descriptor. The bDeviceClass, bDeviceSubClass, and bDeviceProtocol fields describe the functionality of the device. These are filled with values defined by USB for specific functions. For USB-MIDI, these fields are all set to 0. The idVendor field contains an ID associated with a manufacturer. This ID is assigned by the USB Implementers' Forum (USB-IF).

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes. (12h)
01h	bDescriptorType	1	The descriptor type ID. (1)
02h	bcdUSB	2	The supported version of USB. (0200h)
04h	bDeviceClass	1	The device class code.
05h	bDeviceSubClass	1	The device subclass code.
06h	bDeviceProtocol	1	The device protocol code.
07h	bMaxPacketSize0	1	Maximum size in bytes of a packet for the Default Control Pipe.
08h	idVendor	2	Device vendor ID.
0Ah	idProduct	2	Device product ID.
0Ch	bcdDevice	2	Device release number in BCD format.

0Eh	iManufacturer	1	Index of the String descriptor with the name of the manufacturer.
0Fh	iProduct	1	Index of the String descriptor with the name of the product.
10h	iSerialNumber	1	Index of the String descriptor with the device's serial number.
11h	bNumConfigurations	1	Number of possible configurations.

Table 4-14: Format for a standard Device descriptor.

The Configuration descriptor contains information about the properties of a specific configuration of the device. Table 4-15 shows the format of the Configuration descriptor. The wTotalLength field contains the total amount of data in bytes that is sent along with this Configuration descriptor, including the descriptor itself. When a Configuration descriptor is transferred, other descriptors are included in the transfer. A Configuration descriptor is always followed by at least one Interface descriptor if the device exposes any interface. However, the value of the bNumInterfaces field only reflects the actual number of interfaces, and not the number of Interface descriptors that follow the Configuration descriptor. This is because several Interface descriptors may be associated with one interface, with the number of Interface descriptors reflecting the number of alternative settings of that interface. The bConfigurationValue field must be greater than 1 since the value 0 is used for unconfigured devices.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes. (09h)
01h	bDescriptorType	1	The descriptor type ID. (2)
02h	wTotalLength	2	Total size of data that is given by this configuration, including the sizes of itself and all interface, endpoint, class-specific, and custom descriptors.
04h	bNumInterfaces	1	Number of interfaces for this configuration.
05h	bConfigurationValue	1	ID for this configuration. Used for setting the configuration.
06h	iConfiguration	1	Index of the String descriptor with the name of the configuration.
07h	bmAttributes	1	A bitmap with configuration characteristics. Bit 7: (must be set to 1)

			Bit 6: Power source ● 0: Bus-powered ● 1: Self-powered Bit 5: Remote wake-up support ● 0: Not supported ● 1: Supported Bits 4-0: (must be set to 0)
08h	bMaxPower	1	The maximum amount of power in units of 2 mA used by the device in this configuration.

Table 4-15: Format for a standard Configuration descriptor.

There are two formats for the String descriptor; the String descriptor at index 0 has the String Descriptor Zero format, while all other String descriptors have the same format. String Descriptor Zero contains information about the languages that the device supports. This allows the device to supply different strings with the same index for different languages. Table X shows the format of String Descriptor Zero. The language IDs are defined by the USB-IF. If the device does not have any string descriptors, then no array of language IDs must be included in the descriptor. The normal String descriptor contains only an array of Unicode characters.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes.
01h	bDescriptorType	1	The descriptor type ID. (3)
02h	wLANGID[N]	2*N	An array of supported language IDs.

Table 4-16: Format for String Descriptor Zero.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes.
01h	bDescriptorType	1	The descriptor type ID. (3)
02h	bString[N]	N	Unicode encoded string.

Table 4-17: Format for a standard String descriptor.

The Interface descriptor contains information about an alternative setting of an interface. If the interface only has one alternative setting, then the interface only has one Interface descriptor associated with it. Table 4-18 shows the format of the Interface descriptor. Interface descriptors cannot be specifically requested, they must only be transferred during a request for a Configuration descriptor and sent after it. The function of the

bInterfaceClass, bInterfaceSubClass, and bInterfaceProtocol fields are like those of the Device descriptor, but these fields may contain different ranges of values depending on the functionality of the device and the specification of such codes by the USB-IF. For a USB-MIDI device, the bInterfaceClass field is set to 01h for the Audio Device Class. The values of the other fields depend on the interface. The Interface descriptor is always directly followed by the Endpoint descriptors associated with it.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes. (09h)
01h	bDescriptorType	1	The descriptor type ID. (4)
02h	bInterfaceNumber	1	The number associated with this interface within the scope of the configuration.
03h	bAlternateSetting	1	ID of this alternative setting for the interface associated with bInterfaceNumber.
04h	bNumEndpoints	1	Number of endpoints used by this interface excluding the Default Control Pipe.
05h	bInterfaceClass	1	Interface class code.
06h	bInterfaceSubClass	1	Interface subclass code.
07h	bInterfaceProtocol	1	Interface protocol code.
08h	iInterface	1	Index of the String descriptor with the name of this interface.

Table 4-18: Format for a standard Interface descriptor.

The Endpoint descriptor contains information about an endpoint that is used by an interface. Table 4-19 shows the format of the Endpoint descriptor. Endpoint descriptors cannot be specifically requested, they must only be transferred during a request for a Configuration descriptor and sent directly following the Interface descriptor with which it is associated. The Endpoint descriptor cannot be used to describe the Default Control Pipe, and endpoints cannot overlap through different interfaces unless the endpoint is used by different alternative settings of the same interface. However, they can overlap across different configurations. The bInterval field is not valid for endpoints that use Bulk and Control transactions, so it must be set to 0.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes. (09h)

01h	bDescriptorType	1	The descriptor type ID. (5)
02h	bEndpointAddress	1	The address of this endpoint. Bit 7: Direction ● 0: Host-to-device ● 1: Device-to-host Bits 6-4: (must be set to 0) Bits 3-0: Endpoint number
03h	bmAttributes	1	A bitmap of the endpoint's attributes. Bits 7-6: (must be set to 0) Bits 5-4: Isochronous usage Bits 3-2: Isochronous synchronization Bits 1-0: Transaction type ● 00: Control ● 01: Isochronous ● 10: Bulk ● 11: Interrupt
04h	wMaxPacketSize	2	Maximum packet size that this endpoint supports. Bits 15-13: (must be set to 0) Bits 12-11: High-speed only Bits 10-0: Maximum packet size in bytes
06h	bInterval	1	Interval in frames that the host should attempt transactions.

Table 4-19: Format for a standard Endpoint descriptor.

Most of these requests are valid any time after the device has a unique address assigned to it. Only the GET_DESCRIPTOR and SET_ADDRESS requests are valid if the device does not have its address set. The GET_INTERFACE and SET_INTERFACE requests are only valid when the device is configured. GET_DESCRIPTOR requests may ask for any descriptor at any time, except for Interface and Endpoint descriptors. Once the device is configured, it should expect to perform other requests and transactions associated with its functionality.

Application Layer (USB-MIDI)

There is a standard for sending a MIDI stream through USB called USB-MIDI. It uses a MIDIStreaming interface, which is a subclass of the Audio interface class. In order to use the MIDIStreaming interface, the device must also support the AudioControl interface. A USB-MIDI device is organized into several of Elements and Jacks. An Element is a source or sink of a MIDI data stream. It can have at least one input pin and at least one output pin. Each pin carries one MIDI data stream. A Jack connects an Element to the USB host. The Element could be one internal to the device or it could be an external device that connects to the USB device. A MIDI IN Jack moves a MIDI data stream from the host or an external device to the USB device, while a MIDI OUT Jack moves a MIDI data stream from the USB device to the host or an external device. A MIDI IN Jack can only have one

output pin, while a MIDI OUT Jack can have several input pins. Pins are internal connections in the USB device between different Elements and Jacks. Both Elements and Jacks are called Entities.

In order to support a MIDIStreaming interface, the device must support the AudioControl interface. This interface allows the host to control the audio functionality of the device and receive the device's audio-related statuses. The AudioControl interface uses the standard Interface descriptor and several other class-specific descriptors to describe itself. For the standard AudioControl Interface descriptor, the `bNumEndpoints` field must be set to 0, the `bInterfaceClass` field must be set to 01h (Audio), the `bInterfaceSubClass` field must be set to 01h (AudioControl), and the `bInterfaceProtocol` field must be set to 0. The AudioControl uses class-specific descriptors to provide more information about the audio functionality of the device. All these descriptors are appended to the standard AudioControl Interface descriptor. Together they make up the class specific AudioControl Interface descriptor. Since these descriptors are class-specific, they must use special class-specific descriptor type IDs. Table 4-20 lists these type IDs; their functionality is like the standard versions of these descriptor type IDs. These descriptors also use special descriptor subtype IDs to specify the AudioControl-specific object that the descriptor describes. These subtype IDs are listed in Table 4-21. The only relevant AudioControl descriptor is the Header descriptor, since the other descriptors describe objects that only apply to audio stream processing and not MIDI data streams. The AudioControl Header descriptor is shown in Table 4-22. This descriptor contains extra general information related to the audio device and introduces the descriptors that follow it. The AudioControl interface does not have any endpoints associated with it because it uses the Default Control Pipe. Thus, it does not use any class-specific Endpoint descriptors.

Name	ID
CS_Undefined	20h
CS_Device	21h
CS_Configuration	22h
CS_String	23h
CS_Interface	24h
CS_Endpoint	25h

Table 4-20: Enumeration of class-specific descriptor type IDs.

Name	ID
AC_Descriptor_Undefined	00h
Header	01h
Input_Terminal	02h

Output_Terminal	03h
Mixer_Unit	04h
Selector_Unit	05h
Feature_Unit	06h
Processing_Unit	07h
Extension_Unit	08h

Table 4-21: Enumeration of AudioControl descriptor subtype IDs.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes.
01h	bDescriptorType	1	The descriptor type ID. (24h)
02h	bDescriptorSubtype	1	The descriptor subtype ID. (01h)
03h	bcdADC	2	Version of the supported Audio Device Class Specification. (0100h)
05h	wTotalLength	2	Total number of bytes sent with this descriptor, including itself and all other related descriptors.
07h	bInCollection	1	Number of AudioStreaming and MIDIStreaming interfaces that belong to this AudioControl interface.
08h	baInterfaceNr[N]	N	Array of interface numbers of the AudioStreaming or MIDIStreaming interfaces associated with this AudioControl interface.

Table 4-22: Format for an AudioControl Header descriptor.

The MIDIStreaming interface is described with a combination of the standard Interface descriptor and several class-specific descriptors. The standard MIDIStreaming Interface descriptor has the `bInterfaceClass` field set to 1 (Audio), the `bInterfaceSubClass` field set to 3 (MIDIStreaming), and the `bInterfaceProtocol` field set to 0. The `bNumEndpoints` field should be set to at least 1 since it needs at least 1 endpoint to transfer data. The MIDIStreaming interface also has special descriptors for it. All these descriptors are appended to the standard MIDIStreaming Interface descriptor. Together they make up the class specific MIDIStreaming Interface descriptor. These descriptors also use special class-specific descriptor type IDs and MIDIStreaming descriptor subtype IDs. These subtype IDs are listed in Table 4-23. The format of the class specific

MIDIStreaming Header descriptor is shown in Table 4-24. This descriptor contains extra general information related to the MIDIStreaming interface and introduces the descriptors that follow it.

Name	ID
MS_Descriptor_Undefined	00h
MS_Header	01h
MIDI_IN_Jack	02h
MIDI_OUT_Jack	03h
Element	04h

Table 4-23: Enumeration of MIDIStreaming descriptor subtype IDs.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes. (07h)
01h	bDescriptorType	1	The descriptor type ID. (24h)
02h	bDescriptorSubtype	1	The descriptor subtype ID. (01h)
03h	bcdADC	2	Version of the supported MIDIStreaming SubClass Specification. (0100h)
05h	wTotalLength	2	Total number of bytes sent with this descriptor, including itself and all other related descriptors.

Table 4-24: Format for a MIDIStreaming Header descriptor.

The other MIDIStreaming descriptors that are appended to the standard MIDIStreaming Interface descriptor are the MIDI IN Jack, MIDI OUT Jack, and Element descriptors. Only the MIDI OUT Jack and Element descriptors are relevant for this device. Table 4-25 shows the format for a MIDIStreaming MIDI OUT Jack descriptor. This descriptor describes the properties of a MIDI OUT Jack, including its ID, input sources, and type. The type of Jack can either be embedded or external; this specifies whether the Jack connects to an internal MIDI Element within the USB device or connects to an external one beyond the USB device, respectively.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes.
01h	bDescriptorType	1	The descriptor type ID. (24h)

02h	bDescriptorSubtype	1	The descriptor subtype ID. (03h)
03h	bJackType	1	Type ID of the Jack. Can be one of: ● 1h: Embedded ● 2h: External
04h	bJackID	1	Entity ID of this Jack.
05h	bNrInputPins	1	Number N of input pins that this Jack has.
06h	baSourceIDPin	2*N	An array of words where the LSB contains the Entity ID of the Entity that this Jack connects to and the MSB contains the output pin number on that Entity to which this input pin is connected.
...	iJack	1	Index of the String descriptor with the name of this Jack.

Table 4-25: Format for a MIDIStreaming MIDI OUT Jack descriptor.

The MIDIStreaming Element descriptor describes the properties of a MIDI Element within the scope of the device. It can be an internal or external source or sink of a MIDI data stream. It also lists the capabilities of the Element. These are shown in Table 4-26. For this device, only the GM1 capability is relevant because GM1 defines the musical note scale that we are using. The entire table is shown for reference only, in case the device is extended to support a new feature that uses any of these capabilities. The properties that the Element descriptor lists are the Entity ID, the number of input and output pins, the MIDI stream input sources, the audio stream inputs and outputs, and the Element's capabilities. Table 4-27 shows the format of the Element descriptor.

Name	Bit Index	Meaning
CUSTOM	0	Custom capabilities.
MIDI_CLOCK	1	MIDI CLOCK messages support.
MTC	2	Synchronization features support.
MMC	3	MMC messages support.
GM1	4	General MIDI System Level 1 compatibility.
GM2	5	General MIDI System Level 2 compatibility.
GS	6	Roland GS format compatibility.
XG	7	Yamaha XG format compatibility.

EFX	8	USB-controlled audio effects processor included.
MIDI_PATCH_BAY	9	Internal MIDI patcher or router provided.
DLS1	10	DownLoadable Sounds Standard Level 1 compatibility.
DLS2	11	DownLoadable Sounds Standard Level 2 compatibility.

Table 4-26: Bitmap of Element capabilities and their meanings.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes.
01h	bDescriptorType	1	The descriptor type ID. (24h)
02h	bDescriptorSubtype	1	The descriptor subtype ID. (04h)
03h	bElementID	1	Entity ID of this Element.
04h	bNrInputPins	1	Number N of input pins that this Element has.
05h	baSourceIDPin	2*N	An array of words where the LSB contains the Entity ID of the Entity that this Element connects to and the MSB contains the output pin number on that Entity to which this input pin is connected.
...	bNrOutputPins	1	Number of output pins that this Element has.
...	bInTerminalLink	1	The Terminal ID of the Input Terminal to which this Element is connected.
...	bOutTerminalLink	1	The Terminal ID of the Output Terminal to which this Element is connected.
...	bElCapsSize	1	Size M in bytes of the bmElementCaps field.
...	bmElementCaps	M	A bitmap that contains information on the capabilities of the Element. See Table x for the format.
...	iElement	1	The index of the String descriptor with the name of this Element.

Table 4-27: Format for a MIDIStreaming Element descriptor.

Since a MIDIStreaming interface can have at least one endpoint associated with it for transferring MIDI data streams, it must append at least one Endpoint descriptor with its associated MIDIStreaming Interface descriptor. The MIDIStreaming Endpoint descriptor uses the standard Endpoint descriptor appended with the class specific MIDIStreaming Bulk Data Endpoint descriptor. The MIDIStreaming Bulk Data endpoint uses bulk transactions, so the associated Endpoint descriptor must describe it as such. Two bytes are also appended to the end of the standard Endpoint descriptor; these are set to 0. The format of the class specific MIDIStreaming Bulk Data Endpoint descriptor is shown in Table 4-28. All it does is associate this endpoint to several MIDI Jacks. There is another class specific MIDIStreaming Endpoint descriptor, called the MIDIStreaming Bulk Transfer Endpoint descriptor, but it is primarily used for devices that support DLS. Since our device does not use DLS, this descriptor is irrelevant. The descriptor subtype of these class specific MIDIStream Endpoint descriptors is always 1.

Offset	Field	Size (B)	Function
00h	bLength	1	Size of the descriptor in bytes.
01h	bDescriptorType	1	The descriptor type ID. (25h)
02h	bDescriptorSubtype	1	The descriptor subtype ID. (01h)
03h	bNumEmbMIDIJack	1	Number N of embedded MIDI Jacks associated with this endpoint.
04h	baAssocJackID	N	An array of bytes where each byte contains the Entity ID of a Jack that is associated with this endpoint.

Table 4-28: Format for a MIDIStreaming Bulk Data Endpoint descriptor.

The AudioControl interface also defines class-specific requests. However, all of these are only relevant for audio stream processing and not for MIDI streams. The MIDIStreaming interface also defines class-specific requests. However, there is only one standard request, and it is only relevant to the MIDIStreaming Bulk Transfer endpoint. As mentioned before this feature does not apply to this device. Thus, none of these class-specific requests need to be supported.

USB-MIDI has a special feature called virtual cables. This feature allows the transport of 16 logical MIDI streams through one Bulk endpoint. Each cable number corresponds to the index of the MIDI Jack associated with the endpoint, as listed in the MIDIStreaming Bulk Data Endpoint descriptor. In order to transfer a MIDI stream from the device to the host, the host must initiate a Bulk transaction through the MIDIStreaming Bulk Data endpoint. The data that is transferred is a sequence of 4-byte MIDIStreaming packets that wrap around three bytes of a MIDI stream. Typically, the three bytes correspond to one MIDI message. However, for System Exclusive messages, every three bytes of the message is contained within a MIDIStreaming packet, since those messages have

arbitrary length. The format of the MIDIStreaming packet is shown in Table 4-29. The Code Index number (CIN) describes the function of the message. For CINs 8h through Eh, these bits correspond to the message code in the status byte of the message. CINs 2h through 7h are used for System messages. CIN Fh is used to send a single byte in the packet. Table 4-30 lists the available CINs and their function. Support for the Running Status feature of MIDI is unspecified.

Byte 0	Byte 1	Byte 2	Byte 3
Bits 0-3: Cable number Bits 4-7: CIN	MIDI byte 0. Typically the status byte for every message other than System Exclusive messages.	MIDI byte 1. Typically the first data byte. If the message does not have one then this field is set to 0.	MIDI byte 2. Typically the second data byte. If the message does not have one then this field is set to 0.

Table 4-29: Format for a MIDIStreaming MIDI data packet.

CIN	Payload Size (B)	Function
2h	2	2-byte System Common message.
3h	3	3-byte System Common message.
4h	3	Beginning or middle part of a System Exclusive message.
5h	1	1-byte System Common message or last byte of a System Exclusive message.
6h	2	Last 2 bytes of a System Exclusive message.
7h	3	Last 3 bytes of a System Exclusive message.
8h	3	Note Off message.
9h	3	Note On message.
Ah	3	Polyphonic Key Pressure message.
Bh	3	Controller Change
Ch	2	Program Change
Dh	2	Channel Key Pressure
Eh	3	Pitch Bend
Fh	1	Single byte.

Table 4-30: Enumeration of CIN functions.

4.1.6. Design Impact of Relevant Standards

The standards that we have chosen to adhere to are all for the benefit of making the device more accessible to the widest range of applications with the greatest variety of other devices. By outputting our MIDI signal through both a MIDI port and a USB port, we can interface with 1) other MIDI devices such as digital synthesizers and digital pianos via MIDI cable and 2) computers and digital audio workstations via USB. These two standards give us all the variety we need in our outputs to interface with any device that a potential user could want to interface with.

Similarly, the choice to use quarter-inch tip-sleeve and XLR3 with phantom power is one with consideration for accessibility. The two types of devices that we foresee a user connecting to the input of this device are either an electric instrument, such as electric guitar or electric piano, or a microphone. Most electric instruments use a mono quarter-inch tip-sleeve audio cable to transmit their signal. There are also quarter-inch cable standards of tip-ring-sleeve (TRS) and tip-ring-ring-sleeve (TRRS) for balanced signals or stereo audio and stereo audio with a separate channel for microphone input. These options seem unnecessary as they are typically used for devices such as headphones, speakers, and mixers. Including balanced TRS or TRRS compatibility will not make our input more accessible because the input devices one would use with this device do not use that standard.

The XLR3 jack is the most common standard for microphone inputs or for “direct out” or “line out” outputs on amplifiers. In order to capture vocals or acoustic instruments, the device should be able to accept a microphone input. There are, however, many different types of microphones such as dynamic, ribbon, or condenser. Most types of microphone will work on their own just by plugging them in, but most condenser microphones need to be supplied with 48V phantom power. By implementing a 48V phantom power toggle switch, we can support any kind of XLR3-compatible microphone the user has available. After XLR3, the next most common microphone standard is USB, to be plugged directly into a computer. We have decided not to support these kinds of microphones.

4.2. Realistic Design Constraints

There are numerous constraints that need to be considered when designing and building our device. These include but are not limited to economic, time, environmental, social, political, ethical, safety, manufacturability, and sustainability constraints.

4.2.1. Economic and Time Constraints

The economic constraints that are considered when designing our device is that we do not have a sponsor for our project. Most of the group are low-budget college students that would not like to spend a great amount of money on building the device. Due to this, we set a maximum budget on building the device to 450$. Due to the nature of the device, this is a realistic budget that should allow us to build a functionable design that meets our requirements specifications. This maximum budget is also desirable because it limits the

maximum cost of the device for each group member to 112.50$ each. This economic constraint does limit us in the quality of functionality of our device. To keep our budget low, we may eliminate potential features that we may have wanted in our device initially. This means that we must take pricing into account when choosing our microcontroller, power supply, PCB, and all other parts of the device.

Component/Device	Budget	Component/Device	Budget
PCB	200$	Case	30$
Microphone	Free	Passive components	30$
Power Supply	Free	DC-DC converters	50$
Analog-to-Digital Converter	10$	Input/output interfaces	30$
Processor	20$	Voltage regulators and active components	50$
USB cable	2$	48 V Phantom Power Output	25$
MIDI cable	3$	**Total**	**450$**

Table 4-31: Expected budget breakdown

There are multiple time constraints that are considered when designing and building our device. One time constraint is that some of the members of the group working on the device have jobs and other classes/projects that will keep them busy through senior design. To mitigate this constraint, we make sure to meet at times weekly where each member is not busy. This constraint can mean that some members of the group may not be able to work on the project for as much time as they would like. Another time constraint is the amount of time it may take for us to receive parts that we order for our device. There could be delays in the shipping time for the parts we select. If a part that we order is not available in the time that we need it, then we can mitigate this problem by exploring different suppliers and options for the part needed.

4.2.2. Environmental, Social, and Political Constraints

The main environmental constraint that needs to be considered for our senior design project is the COVID-19 pandemic. Due to COVID-19, we are not meeting in person and we are all meeting online to design the device. We also do not have the access to the Senior Design lab or other on-campus amenities during Senior Design 1. Some members of the group have moved back to their original residence and are not in Orlando for Senior Design 1. This makes it difficult to plan and work together on the project. The COVID-19 pandemic could also influence the manufacturing of parts that we need for our device. COVID-19 started in China and has had a detrimental effect on products coming out of China. China is one of the main manufacturers of electronic parts so we may have to

consider parts from other manufacturers. Without access to the Senior Design Lab it will make it difficult to test and prototype circuits needed for our device. An oscilloscope testing kit has been sent to one of our group members which will allow us to test circuits we will need for the device. Social constraints also played a role in the design of the device. The social constraints that we considered when designing our device mostly deal with devices being used for musical purposes. We want the device to be compatible with as many MIDI devices as possible. For this reason, we needed to implement both XLR3 inputs and outputs and ¼-inch jack inputs and outputs. Both connector types are commonly used with MIDI devices, guitars and microphones which can be used with the device.

4.2.3. Ethical, Health, and Safety Constraints

Usually when dealing with devices that record audio, they are not allowed to continuously record without the user's permission. This is a privacy concern that many developers of anything that can record audio have to deal with. Our device needs to have a microphone attached to the device that is provided by the user, so the device user does not need to worry about the device secretly recording audio. All MIDI transcriptions that are made by the user will be used only by the user and there will be no way for the developers to be able to take any recording from the device. We want our device to be completely safe for all people to use, so we have implemented some health and safety constraints for the device. We need to make sure our device is built fundamentally sound so that it is safe to use without any malfunctions that could harm the user. This is done by testing all the circuits that will be used for power before we implement them into the device. As an additional safety measure, we must make sure to design our device such that it does not produce any unintended interference. Almost all consumer electronics sold in the United States is subject to part 15 of the FCC CFR title 47. Subpart B of this section explains the rules that apply to consumer electronics which may act as "unintentional radiators" of radio wave noise. We don't want any part of our device from the case to the PCB to be acting as an antenna and spreading radio interference in a way that conflicts with these rules. [30]

4.2.4. Manufacturability and Sustainability Constraints

Manufacturability constraints need to be considered when choosing parts for the construction of the device. We will have to research standard techniques for PCB design, stackup, material (rigid, flex, rigidized flex, etc.), and other board parameters before we can have our PCB manufactured. Multiple factors go into the decision making process of which manufacturers we will use to get our parts for the device. Some of these factors include cost, availability, and quality. Single unit and bulk pricing both need to be considered when buying parts for the device. If we were to put the device on the market we would buy in bulk for the parts we use to maximize profits. In terms of availability, most of the parts should be easily attainable for the construction of the device. To lower the cost of the device we are using some parts that certain members of the group already own. Another constraint that we could face in availability in manufacturing of parts is that many manufacturing plants may not be working now due to the coronavirus pandemic. This means we may have to look at more potential manufacturers than we may have had to before the pandemic. It also means that there could be delays in the manufacturing in the parts we need. These are circumstances that we will need to consider when purchasing parts for the device.

In terms of sustainability, we want our product to last for a long time without any malfunctions. One sustainability constraint that we have placed on ourselves for the device is for the device to have a low heat output. We do not want our device to require active cooling because this can interfere with quality of the MIDI translation done by the device. We also do not want our device to overheat and potentially harm the device or user. There should be little to no electromagnetic noise created by the device, to protect the functionality of the device. In the event of a part failing in the design, we would want to be able to diagnose the problem and repair the device. In order to do this, we must design our PCB with test points that will allow us to take measurements that yield meaningful data regarding the function and wellbeing of specific parts. We will also have to consider component availability or scarcity when selecting our parts in order to ensure that the board will be repairable in the future.

5. Hardware Design Details

The hardware design can be broken down into three sections: the power section, the analog audio circuitry section, and the note transcription section. Each section consists of several smaller blocks that will perform the various tasks needed to operate the device. The power section consists of two voltage protection circuits and two voltage converters which will provide power to the device at 5 volts and 48 volts. Power will be drawn from either an external 9V supply or from 5V USB power if a USB connection is present. 9V power will be regulated and stepped down to 5V and regulated to power the logic circuitry in the note transcription section. 5V power will also be stepped up to 48V for the optional phantom power that can be toggled on and delivered to a microphone plugged into the XLR3 jack. The analog audio circuitry section handles all the inputs and outputs for the instrument signal. There are two inputs and two outputs, one for ¼" tip and shield cable and one for XLR3. There will also be a block that handles the selection, routing, and mixing of the input signal to match whatever feature set is implemented in the final product, be it a toggle between the two inputs, simultaneous input from both jacks, or some other implementation. Finally, there will be a preamplifier stage that prepares our instrument signal for the note transcription section.

Managing power constraints is an important aspect of any electrical design. We need a good estimate of how much power is going to be required by each of our hardware blocks to ensure that our 9V barrel jack / 5V USB power scheme is viable. In order to track this, we will use two tools: a power diagram and a power budget spreadsheet. The power diagram is a flow chart, showing the sources (9V DC barrel jack and 5V DC USB port) and every component they are responsible for powering. In the case of this device, both the 5V USB port and the 9V barrel jack will have to be capable of powering the entire device. Once we have more information about the specific components being used in the design, we will be able to fill in the power requirements of each block (currently denoted by "?W"). Our current estimate is that 2W will be required for our design to function. This is a very rough estimate based on our microprocessor options drawing somewhere in the ballpark of 200mA current from our 5V source. Doubling that to 400mA gives us some idea of how much power will be drawn by the entire device, giving us the 2W estimate.

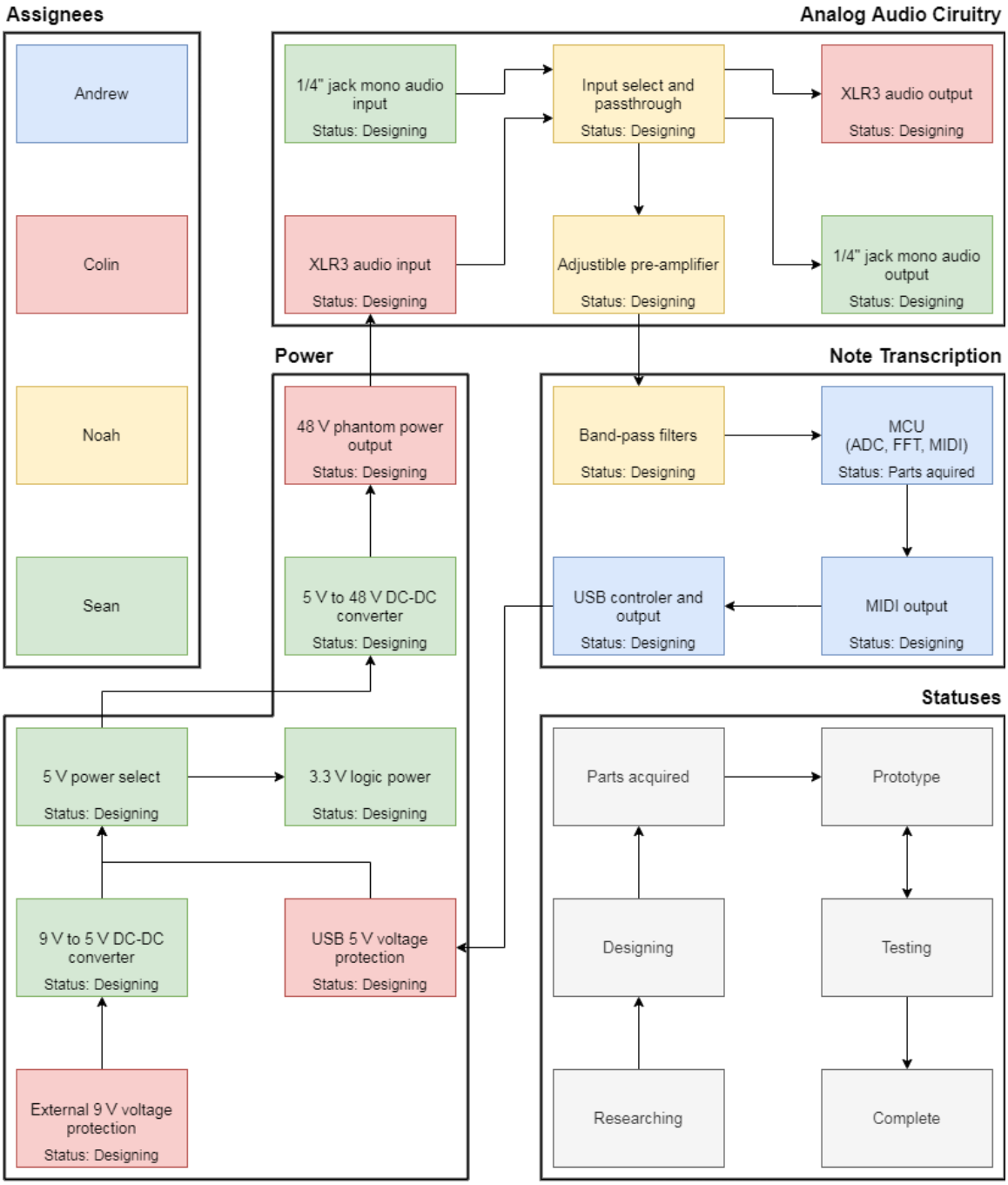


Figure 5-1: The hardware block diagram.

In addition to the power diagram, we have the power budget, which is a part-by-part analysis of how much power is being consumed by what and how much power must be delivered by each source. Again, this is a tool that will become more useful as we are able to populate it with the specific parts we are using in our design. For now, it is filled with the names of the hardware blocks. When it is completely filled out it will be a list of every electrical component on our bill of materials, showing the number of each part used in the device, the current drawn by each part, and the power consumed by each part. We then

can estimate the efficiency of these parts and of the power supply to get our final number for the power required.

9V Supply				
Part Identifier	Part Name	Supply Current Per Part (A)	Supply Voltage (V)	Power (W)
U?	9V to 5V Converter	55mA	9V	0.5W

Table 5-1: 9V Power budget.

5V Supply				
Part Identifier	Part Name	Supply Current Per Part (A)	Supply Voltage (V)	Power (W)
U?	Preamplifier	6mA	5V	30mW
D?	Preamplifier “Clipping” Diode	6mA	5V	30mW
U?	MIDI Controller	5mA	5V	25mW
U?	USB Controller	5uA	5V	25uW
U?	5V to 48V Converter	60mA	5V	0.3W
U?	5V to 3.3V Converter	2.1mA	5V	10.5mW

Table 5-2: 5V Power budget.

3.3V Supply				
Part Identifier	Part Name	Supply Current Per Part (A)	Supply Voltage (V)	Power (W)
U?	MSP430FR5994	3mA	3.3V	10mW

Table 5-3: 3.3V Power budget.

48V Supply				
Part Identifier	Part Name	Supply Current Per Part (A)	Supply Voltage (V)	Power (W)
J?	XLR3 Input Jack	50mA	48V	0.24W

Table 5-4: 48V Power budget.

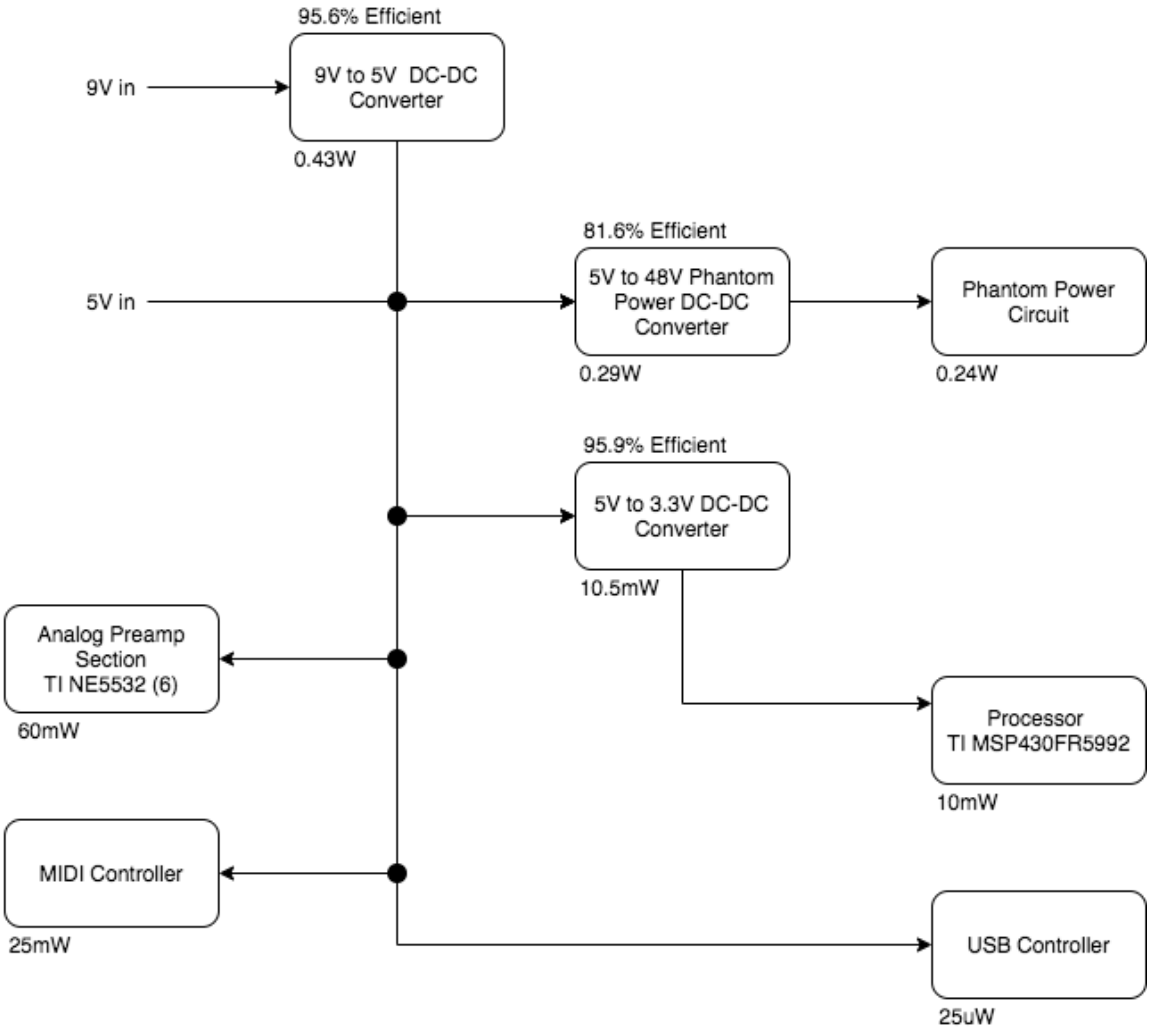


Figure 5-2: Power Diagram

5.1. Power Circuits

Power Switching Circuit

By using two LTC4411s we can prioritize the USB input voltage over the input voltage from the DC barrel jack. Figure 5-3 shows an example of a circuit that uses 2 LTC4411s that prioritizes the USB voltage and can switch between both voltages depending on which is plugged in. Figure 5-4 shows the voltages from both inputs as well as the voltage at the output. The circuit was simulated for three conditions which are all shown on the graph. These conditions are 5V USB ON Jack 5V OFF, USB 5V OFF and Jack 5V OFF, and both OFF. As seen in Figure 5-4 there is a constant Vout slightly below 5V if one of the power sources is turned on. The small voltage drop should be negligible for the purposes of powering the device. Figure 5-5 shows the currents from both power sources during the simulation. This shows that when both power sources are turned on, only the 5V from USB is delivering current.

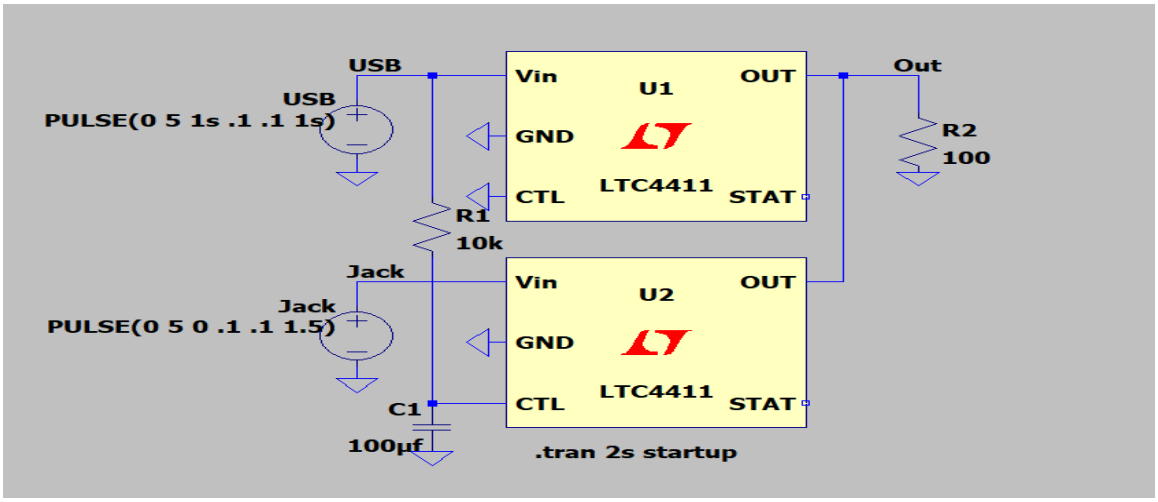


Figure 5-3: Power Switching Circuit

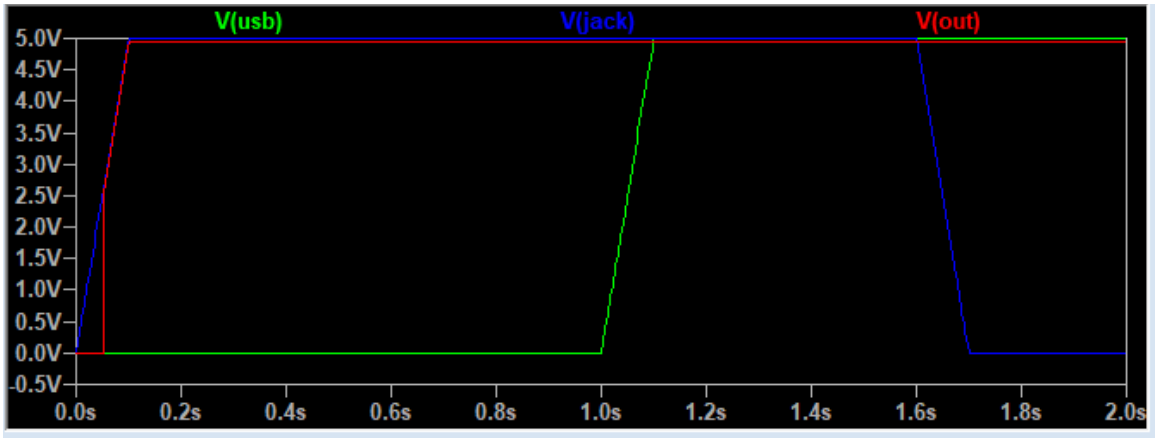


Figure 5-4. Voltage graph of inputs and output

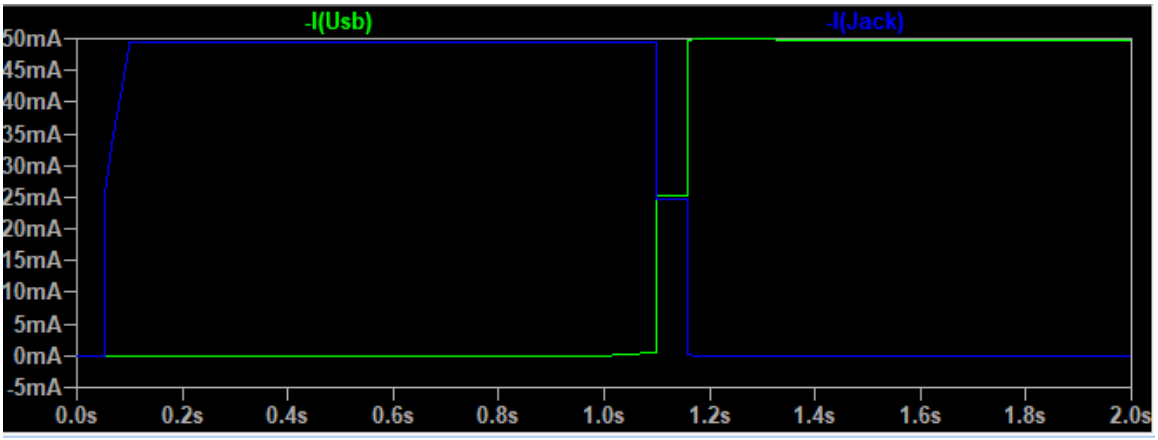


Figure 5-5. Input Current from Power Supplies

A simpler design would be to follow the example one in the chip's datasheet. It is simpler and cheaper. Figure X is a schematic of the design, and Figure X is a simulation showing the functionality. This design has the same functionality as the previous design, and it also outputs a logic signal depending on the current source of the 5V power rail. For this design to work, the 5V power output from the 9V to 5V converter must be at least 5.75V in this simulation. However, this higher voltage heavily depends on the voltage drop of the Schottky diode, which depends on the load on the 5V power rail. A 50-ohm load, as used in the simulation, will cause the voltage drop to be greater than if the load were a 500 ohm one. The output voltage of the 9V to 5V converter can be adjusted by tuning the RFBT1 resistor; increasing the resistance will increase the voltage of the output. Coordinated selection of the diode and resistance value is very important.

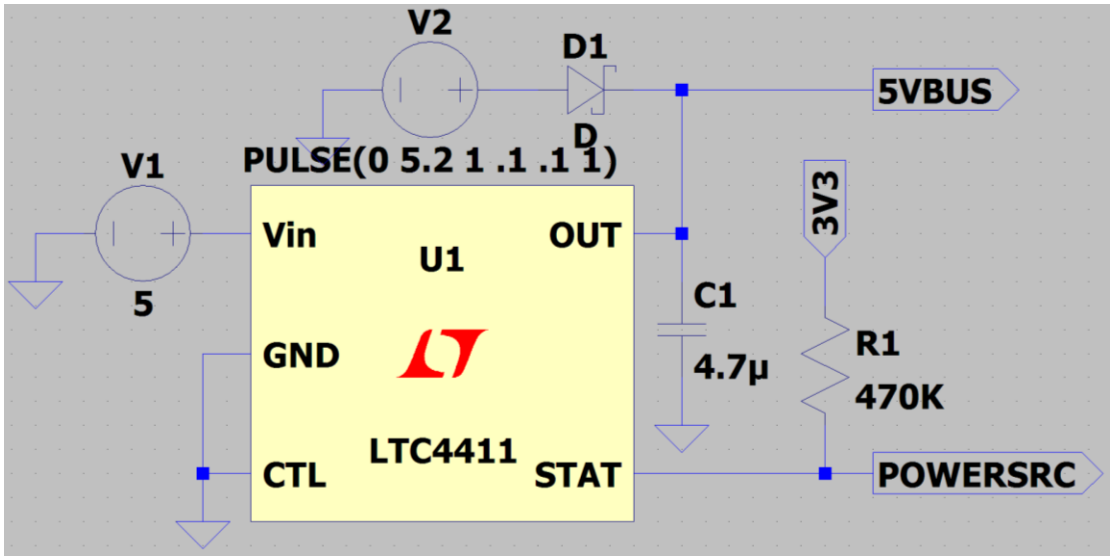


Figure X: Schematic of new power switching circuit.

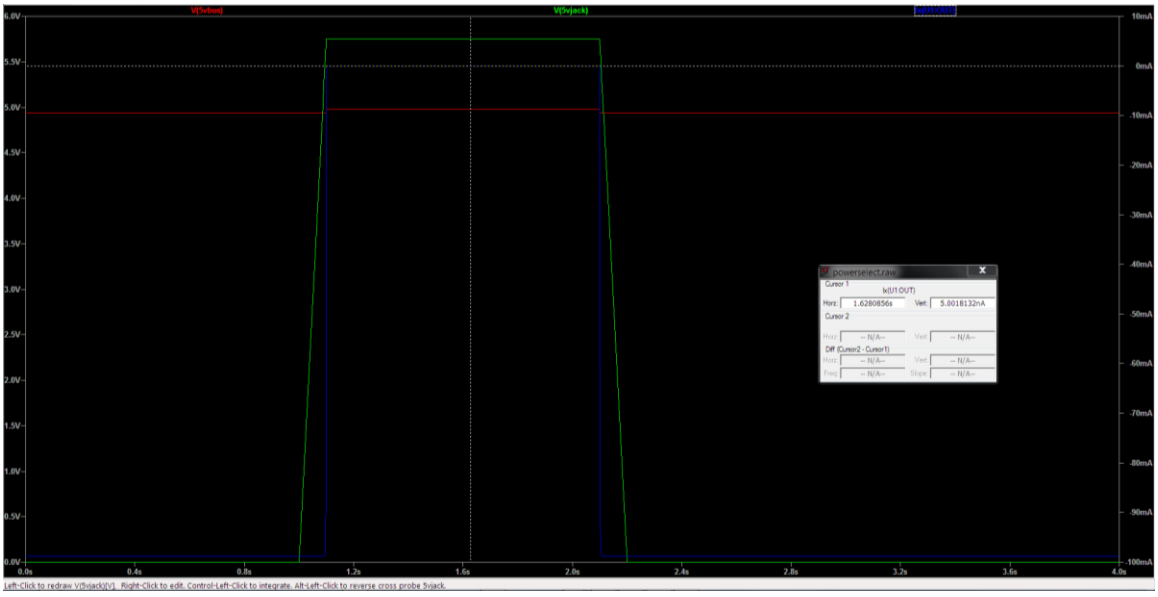


Figure X: Simulation results. 5VBUS (red), 5VJACK (green), and OUT current (blue) are shown.

Power Circuit

Since we have multiple DC-DC converter circuits throughout our power circuit, we have split it up into 3 schematics. The first part of the power circuit is shown in Figure 5-5. This schematic shows the power source inputs to the power switching section of our circuit. This schematic can also be divided into 3 sections. The first section on the top of the schematic is the two power sources from the USB connector and the 9V DC Barrel Jack connector. The USB connector has 4 pins, but for this schematic we only need to use 2 which are the VBUS and GND pins which go to the power switching section and ground respectively. The other two pins are data pins which will need to be connected to the USB Transceiver. The 9V DC Barrel Jack has two pins that are used as well. As seen in the schematic one of the pins goes to ground and the other goes that is used for voltage. The next section of the schematic is the 9V to 5V DC-DC converter that we designed using the TI Webench tool. This can be seen in the right side of the schematic. The 9V to 5V DC-DC converter takes the 9V from the pin from the DC Barrel jack and outputs 5Vto the power switching section. The last section of the first schematic is the power switching section which can be seen on the left of the schematic. This uses two power switching ICs and to switch between 5V from USB and 5V from the 9V to 5V DC-DC converter output. This section outputs approximately 5V to the net labeled 5VBUS which will be used to power most of our device. The 5VBUS net will be used to power the other subsections of our power circuit and be used as the Vcc for any other parts of our design.

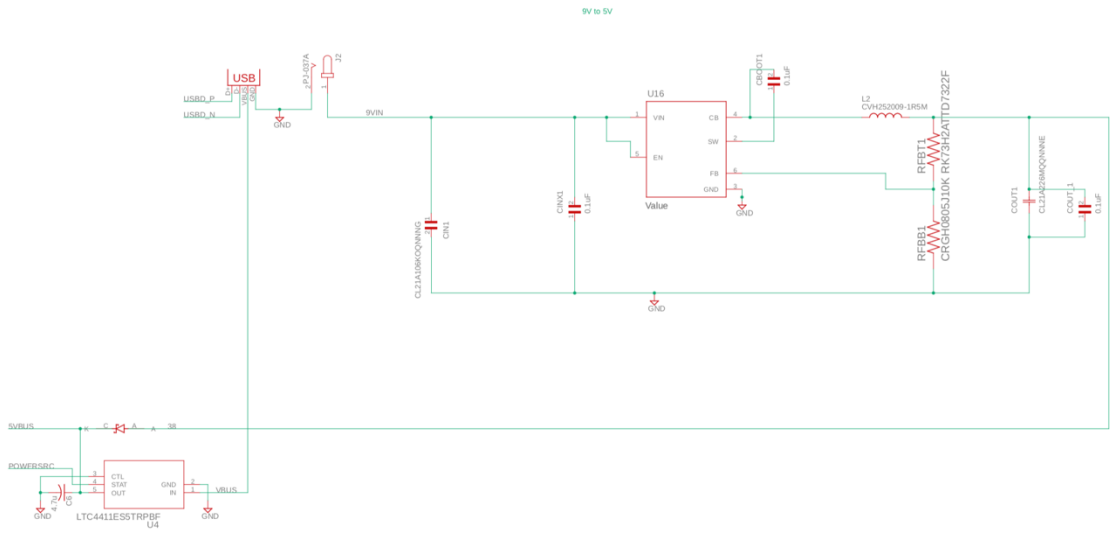


Figure 5-6. 1st subsystem of power circuit

The 2nd subsystem of the power circuit is shown in Figure 5-7. This schematic is the 5 to 3.3V DC-DC converter that we designed using the TI Webench tool. This part of the circuit takes 5V from the 5VBUS net as an input and outputs 3.3V that is used to power the microcontroller that we selected for the device. The final subsystem of the power circuit is shown in Figure 5-8. This schematic is the 5V to 48V DC-DC converter that we designed using the TI Webench tool. This part of the circuit takes 5V from the 5VBUS net as an input and outputs 48V that is used for the phantom power needed to power the condenser microphones that would be used with the device.

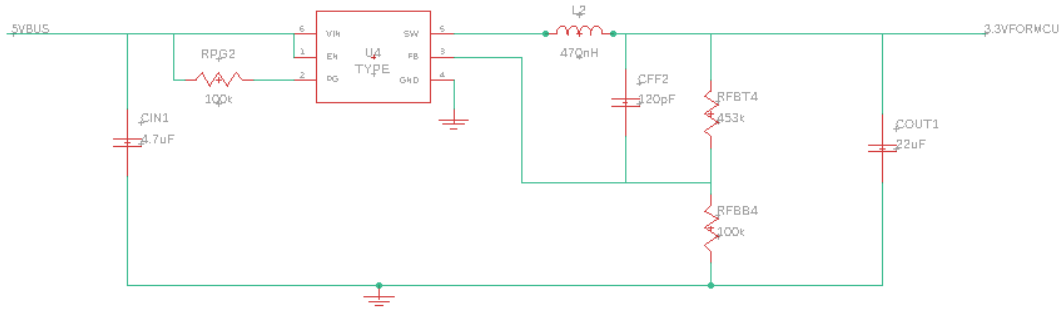


Figure 5-7. 2nd subsystem of power circuit

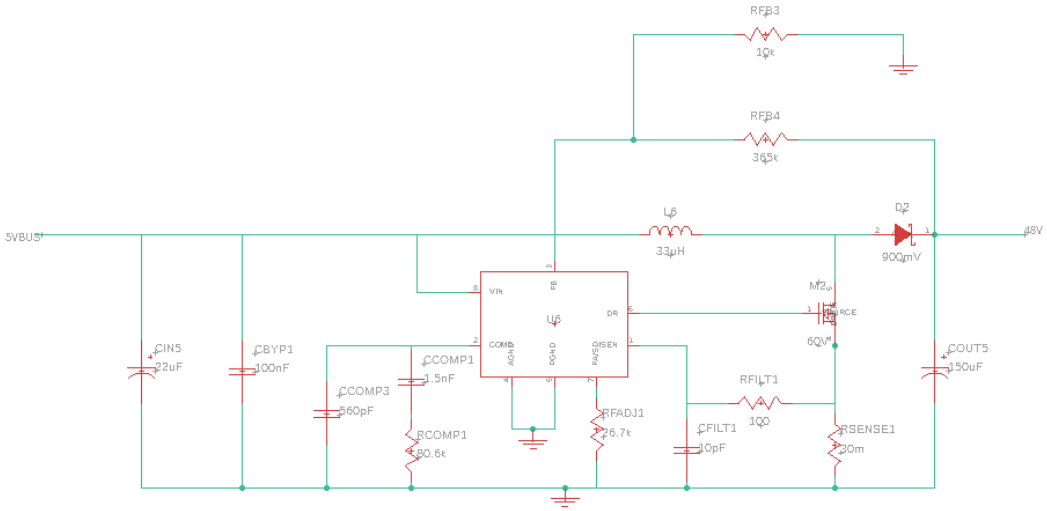


Figure 5-8. 3rd subsystem of power circuit

5.2. Analog Preamplifier

The first section of the device is the analog input section. When the analog signal from either ¼" line in or XLR3 microphone is input into the device, the signal must be prepared and handled in various ways before it reaches our analog to digital converter. The main functions of the preamp section are the microphone input and phantom power circuit, the ¼" line input, an amplifier, several buffers to split the signal into different outputs, and an active bandpass filter. All these sections comprise the entirety of the analog small signals in this device.

The first stage of the preamp is the input of either the ¼" line in or XLR3 microphone input. These are toggleable by a switch on the front panel of the device. When selected, the ¼" input feeds directly into the first amplifier stage while the XLR3 input first goes through a phantom power circuit. The phantom power circuit feeds power at 48 volts through the

microphone to power active microphones like condenser type microphones. The phantom power circuit is standard in the industry and involves feeding the DC power to differential microphone lines which is then taken out on the receiving end. The differential lines are then combined with an op amp in differential mode which causes any received noise during transmission to phase cancel since both lines will receive approximately the same noise but are subtracted from each other at the end. The 48V phantom power is also toggleable by a switch so that passive microphones will be usable as well. See Figure 5-8 to see the phase cancelation of line noise in the microphone lines. The red signal is the positive input V/2 with the noise graphed on top of it and the blue signal is the negative input -V/2 with the noise graphed on top as well. The green signal represents the noise. The first signal (V/2+Noise) is subtracted by (-V/2+Noise) which gives the input signal of V without the noise gained in the microphone cable. The final signal V is represented by the purple sine wave.

$$\left(\frac{V}{2} + Noise\right) - \left(\frac{-V}{2} + Noise\right) = V$$

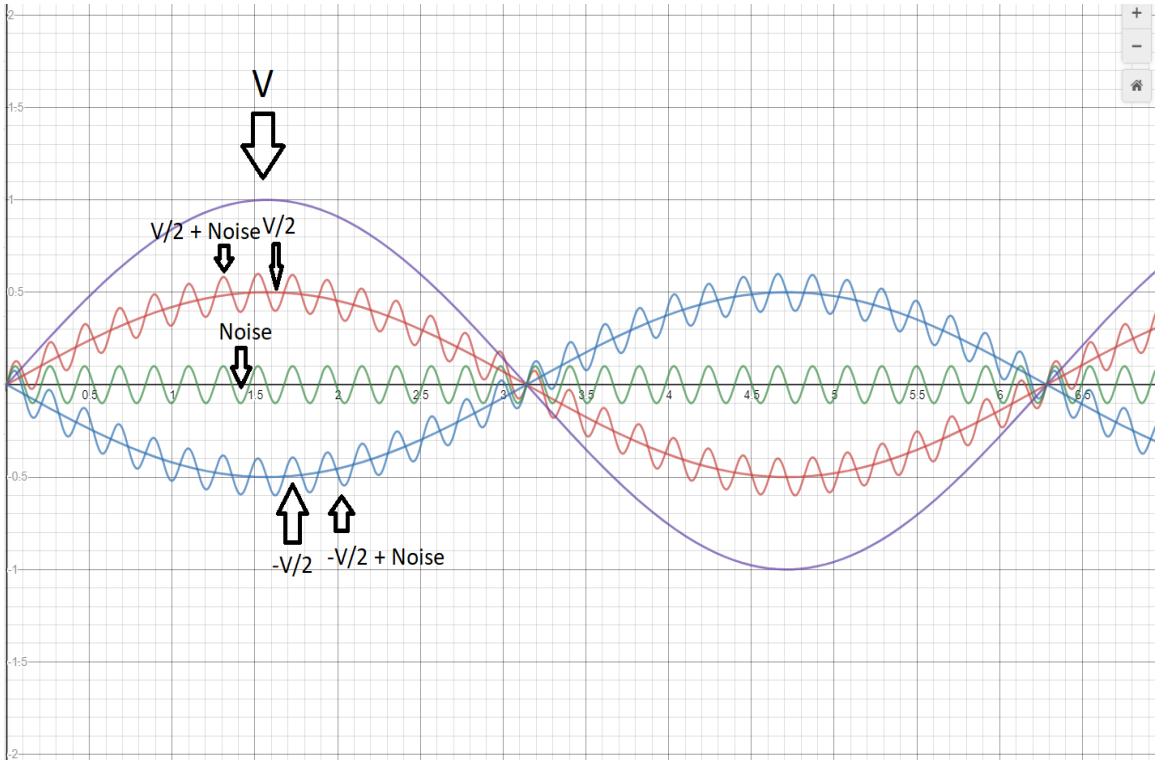


Figure 5-9: Phase cancelation of line noise in the microphone lines

The next stage of this circuit involves splitting the signal into 3. This is done so that the analog input signal can pass through the device unaltered through both an XLR3 output and ¼" line output. This allows you to simultaneously use your analog signal at the same time as the MIDI signal the device creates and outputs generates. The signal splitting is simply achieved by splitting the line through 2 different op amp buffers while the 3rd goes to the next stage of the circuit. These buffers are simply made by connecting the negative feedback to the output causing the differential across the input to be 0, making the gain of

the amplifier 0. This allows us to help isolate the electrical lines from each other so that loads will cause minimal effects on each other. The two lines that go out to either the XLR3 output or ¼" output finally go through a coupling capacitor to get rid of the Vcc/2 DC offset before going through the pass-through output.

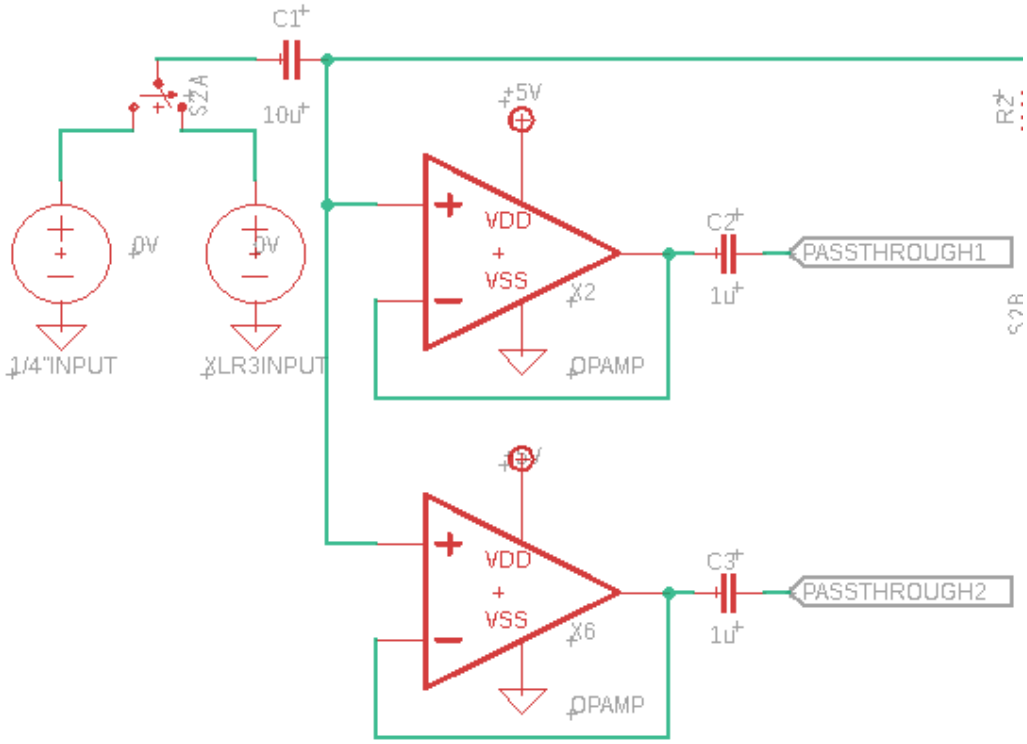


Figure 5-10: Preamp Stage 1

These two different togglable inputs then feed into a non-inverting amplifier. The op amps in this circuit use a single voltage input of 5 volts and a ground reference. This means we give the first amplifier a virtual ground of Vcc/2 so the rest of the circuit has a DC offset of 2.5 volts before the DC offset is decoupled by a capacitor in the output. This is so that we can use a single supply voltage and still give the signal headspace for voltage swing between 0-5 volts. Otherwise, we would have to use a dual rail power supply which is not preferable. This first amplifier then sets the gain level of the input signal. This is adjustable by a potentiometer to "dial" in the gain with your particular instrument or signal, and it also switches ranges of gain when switching between XLR3 or ¼" inputs. This is because the level of microphone outputs is significantly less voltage than line level instrument outputs such as an electric guitar or other instruments with pickups. This is done by wiring the potentiometer to function as a variable resistor in the negative feedback loop by wiring the middle and a side pin in series with a resistor. This makes it so that when the potentiometer is rotated fully clockwise, the resistance is the resistance of the potentiometer and resistor in series. When rotated fully counterclockwise, the resistance is that of just the resistor. This makes the gain the highest when rotated clockwise and lowest when rotated counterclockwise. The range of gain is adjusted by adding in another resistor in the

feedback loop when the switch to select between ¼" and XLR3 is toggled. This combined controls the proper gain levels needed for the analog to digital converter.

The gain levels that we need to achieve are based on both the input range of the ADC and the input level of the microphone or instrument. Microphone voltage levels typically can range from 5 - 50 mV peak to peak and line levels (the ¼" input) are typically .3 - 2 volts peak to peak. We want to try to maximize the voltage going into the range of the ADC. This is because it will minimize the noise to signal ratio with the louder we make our signal. The ADC input range is 0 - 3v peak to peak so we want to come as close to the edges of this range as we can without risking clipping. For the line in input, at the top of its range we have ~ 2V PP so we need a gain of 1.5 to get to 3V pp. Then at the bottom of its range we have .3 volts, so we need a factor of 10 to get to 3V PP. This means for the line in input, we need a configuration of the potentiometer and resistors to give a variable gain of 1.5 - 10 V/V. For the microphone (XLR3) input, at the top of his range we have 50 mV which requires a voltage gain of 60 to get to 3 V. At the bottom of its range, we have 5 mV which requires a gain of 600 to get 3 V peak to peak. This overall means we need a range of gain of 60 - 600 V/V for the microphone input. One thing to keep in mind is that we should give a little "wiggle room" with the gain range so that if we have a voltage at the top of the range, we can lower it a little to ensure it doesn't clip the ADC input. Let us choose a factor of ⅙ to lower the minimum gain by. This gives a range of gain of 1.25 - 10 for the ¼" line in input and 50 - 600 for the XLR3 microphone input. The voltage gain of a non-inverting amplifier is $A_v = 1 + \frac{R_A}{R_B}$ and we will use this to calculate what resistor and potentiometer values to use. Let's set R_B to 10k to find R_A for both the microphone and line-in inputs. The gain of the last stage of the amplifier will be ~1.25 to set the minimum gain so our equation is now $A_v = 1.25(1 + \frac{R_A}{10k})$.

For the low end of the gain for the line-in: $R_A = 0\ \Omega, R_B = 10\ k\Omega$

$$1.25 = 1.25(1 + \frac{0}{10k})$$

For the high end of the gain for the line-in: $R_A = 70\ k\Omega, R_B = 10\ k\Omega$

$$10 = 1.25(1 + \frac{70k}{10k})$$

For the low end of the gain for the microphone input: $R_A = 390\ k\Omega,\ R_B = 10\ k\Omega$

$$50 = 1.25(1 + \frac{390k}{10k})$$

For the high end of the gain for the microphone: $R_A = 4.79\ M\Omega,\ R_B = 10\ k\Omega$

$$600 = 1.25(1 + \frac{4.79M}{10k})$$

With R_B set to 10k, there is the issue that the line in needs a range of ~70 k$\Omega$and microphone in needs a range of ~ 4.4 MΩ. This is not possible if we are keeping things simple and using the potentiometer as a variable resistor. What we can do to bring these ranges to the same scale is to change $R_B$for each of the inputs to compensate for the different resistance ranges. 500k is a readily available potentiometer value so lets first scale the microphone values to fit this. We can build a system where we have $R_A$low as the offset of the gain as the static resistor that is added and then the 500k potentiometer added for the maximum gain.

$$R_{A\,High} = 500k + R_{A\,Low}, 600 = 1.25(1 + \frac{500k + R_{A\,Low}}{R_B}), 50 = 1.25(1 + \frac{R_{A\,Low}}{R_B})$$

This gives us $R_{A\,Low} = 44{,}318\,\Omega, R_B = 1{,}136\,\Omega$

Now lets scale the line in values to the 500k potentiometer as well, this time where the lowest resistance value will be 0 for a gain of 0 (1.25 after the final stage)::

$$R_{A\,High} = 500k, 10 = 1.25(1 + \frac{500k}{R_B}), 1.25 = 1.25(1 + \frac{0}{R_B})$$

This gives us $R_B = 1 = 77{,}429\,\Omega$

Now with these new values, this should correctly scale the gain levels and ranges for each of the inputs on a simple 3 pole 3 throw switch. When this switch is flipped it will add the resistance offset for the microphone input and change R_B to change the gain scaling. It will also switch which input is being used.

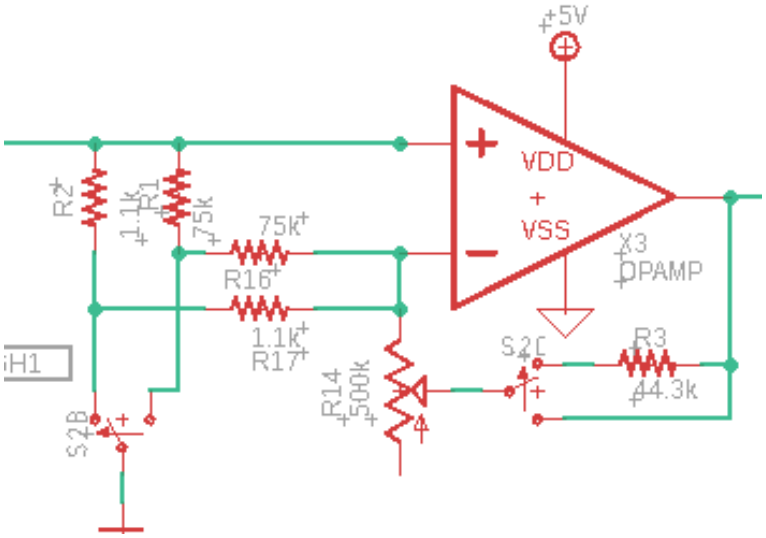


Figure 5-11. - Preamp Stage 2

When considering the gain, we also must consider how the user can know if their gain level is clipping or not so they can adjust with the potentiometer as necessary. This can be solved by adding a circuit in the preamp that detects the voltage levels before the ADC.

When the voltage reaches a threshold of over ~2.8 V then it will switch on an LED which will be visible on the outside of the enclosure. This can be implemented with an op amp configured to be a comparator utilizing the open loop. This means that when the input passes 2.8 volts, the output of the op amp will be 5 volts and under the output will be ground. This makes it so that it can power the LED when it is above 2.8 Volts and ground it out when below. Since op amps cause minimal loading effect and next to no current draw, we can have the signal line split directly off into this circuit before the ADC. We can make our reference voltage simply by making a voltage divider from our 5-volt power rail. Lastly, there is a current limiting resistor going into the LED from the driving op amp to set the amount of current draw from the LED. This resistor will be much less than what we expect to make up for a loss of brightness. This is needed because with an AC signal, our LED will be on only during the time if it is clipping. If we increase the brightness, then it will appear to be on the whole time at a normal brightness by increasing the average power. There is also a buffer in this stage to isolate the next stage as well as improve any impedance issues caused by the previous stage.

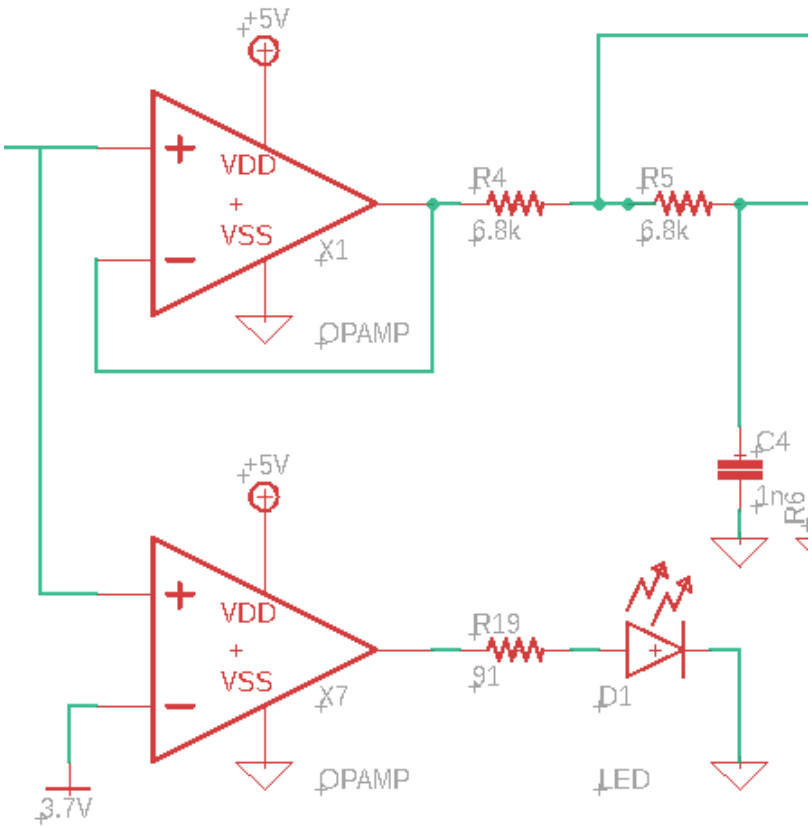


Figure 5-12. - Preamp stage 3

The last stage before the analog input signal reaches the analog to digital converter is an active bandpass filter. This filter is to reduce unwanted frequencies that are outside of musical range so that we have less noise when processing the signal digitally. The frequencies we chose as the corner points of the band pass are ~20 Hz for the low end and ~16 kHz for the high end. This is because the lowest note on a standard bass guitar is ~40 Hz and a corner frequency will remove low resonance without altering the gain of 40 Hz much at all. The highest note recognizable by MIDI protocol is ~15 kHz so we chose

~16 kHz as the upper corner frequency for the same reasons. Ideally this bandpass will filter 100% of frequencies out of the range that we need so we need a steep drop off on our filter. To do this we chose an active 3-stage Butterworth filter for both the high pass and low pass parts of the bandpass. This gives a very flat bandpass which is closer to our ideal response. This is because an ideal 3rd order filter gives an 80dB per decade roll off rate and Butterworth filters approximate this roll off well. The simplest way to calculate the cutoff points of a multistage filter is to use the same resistor and capacitor values for each stage. This gives a corner frequency of $f_C = \frac{1}{2\pi RC}$ for each pass.

For the high pass of ~20 Hz: $R = 75\ k\Omega,\ C = 100\ nC$

$$21.22\ Hz = \frac{1}{2\pi(75k)(100n)}$$

For the low pass of ~16 kHz: $R = 10\ k\Omega,\ C = 1\ nC$

$$15{,}915.49\ Hz = \frac{1}{2\pi(10k)(1n)}$$

One issue we found when simulating the preamp is that the slew rate of the op amp caused decay of higher frequencies when at very high gain levels. This means when using the microphone input, we would be losing some magnitude on the high end of frequencies we want to detect. To negate this slightly, we decided to change the low pass to be a little gentler to counteract the effect of the op amp slew rate. We decided on 6.8k for the resistance.

For the low pass of ~23 kHz: $R = 6.8\ k\Omega,\ C = 1\ nC$

$$23{,}405.14\ Hz = \frac{1}{2\pi(6.8k)(1n)}$$

This active bandpass filter stage also has a bit of voltage gain. Since we determined the minimum voltage gain for all input types to be 1.25 into the ADC, we want to make the last stage of the bandpass amplify the signal by that much. The gain of a non-inverting op amp (which the last stage is), is easily calculated as $A_v = 1 + \frac{R_A}{R_B}$ where R_A is the resistor in the negative feedback loop and R_B is the resistor connecting from the negative feedback loop to the ground.

For a voltage gain of ~1.25: $R_A = 2.4\ k\Omega,\ R_B = 10\ k\Omega$

$$1.24 = 1 + \frac{2.4k}{10k}$$

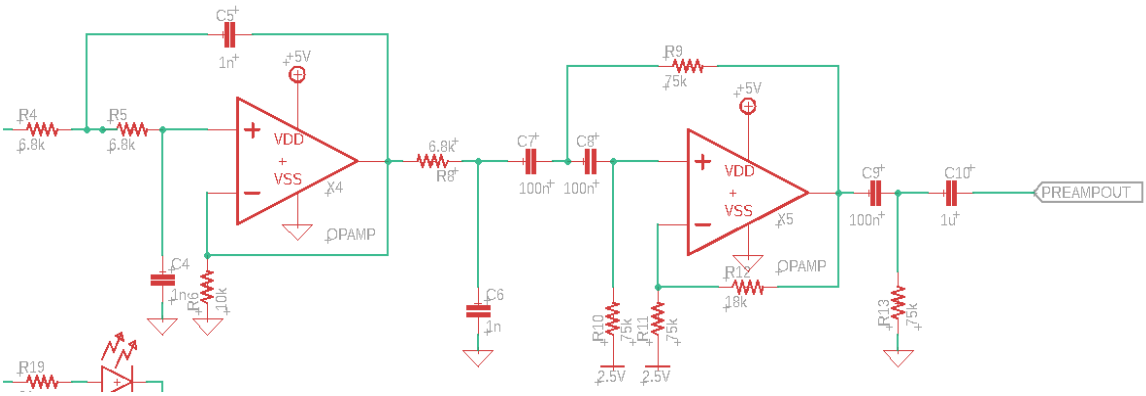


Figure 5-13. - Preamp stage 4

One last thing needs to be done before the signal is passed on to the analog to digital converter, is setting the dc offset of the signal. We must make our signals have a DC offset at the center of the ADC voltage range since it only takes positive signals. The DC bias must be between the max ADC voltage of 3.3 and minimum of 0 volts. This means our ideal DC offset is 1.65 volts. This is done by adding a virtual ground reference point to the signal by adding a pull up resistor, which will bias the signal to 1.2 volts. This reference point is given by the ADC to ensure a consistent bias.

We simulated a frequency sweep of the designed circuit before the gain design and low pass adjustments to see the frequency response and it follows our desired cutoff points of nearly exactly 20Hz and 16kHz. It also appears to give a very stable passband gain and phase response. In the simulation in figure X, the red line is the phase and gain response in the frequency domain of the input. The green lines are the gain and phase response of the final output to the ADC in the frequency domain. Not shown are the outputs to the passthrough outputs which are nearly flat in response but have a slight roll off on very low frequencies, mostly below 5 Hz.

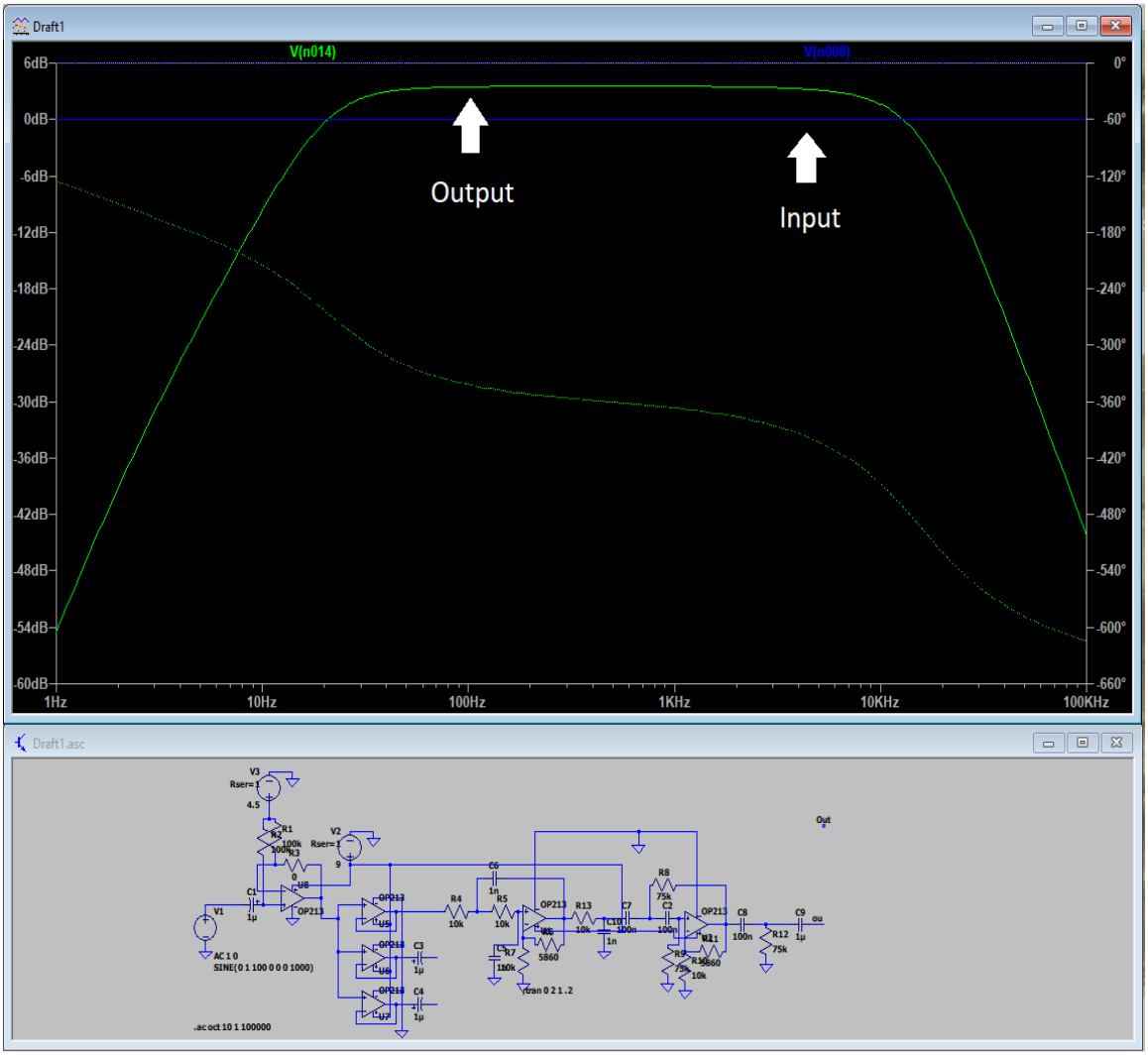


Figure 5-14: Simulation of analog preamp section.

5.3. MIDI Output

Our processor will communicate with MIDI-controllable instruments through a 5-pin MIDI port. This port is soldered directly onto the PCB and will be accessible by the user through the backplate of the device's case. This port is connected to a UART port on the processor and the UART out signal is buffered and sent through a 220 Ohm resistor on its way to the port. This UART connection is the only connection that the MIDI port needs to make with the processor. The full MIDI output hardware diagram is shown in Figure 5-15

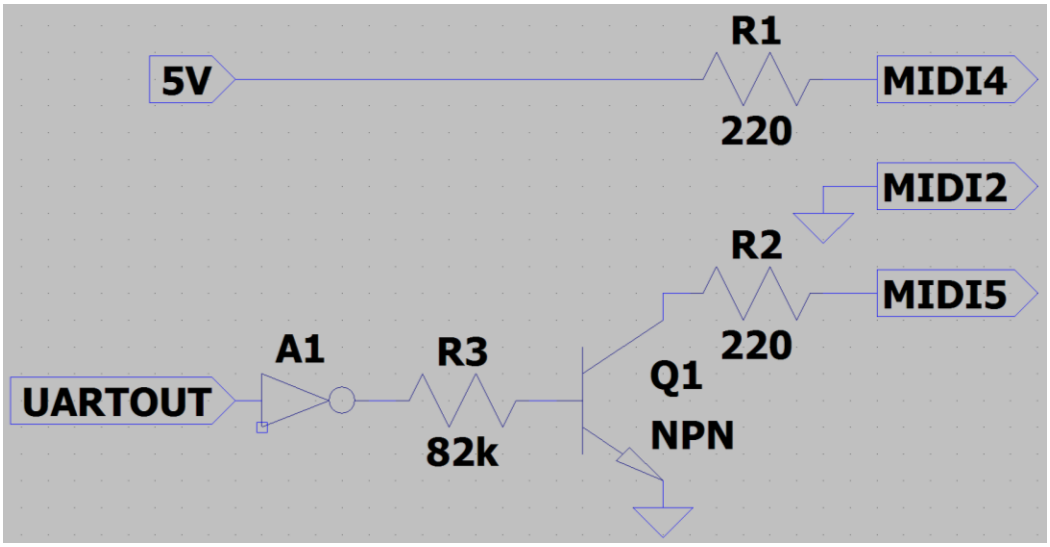


Figure 5-15: MIDI Output Schematic

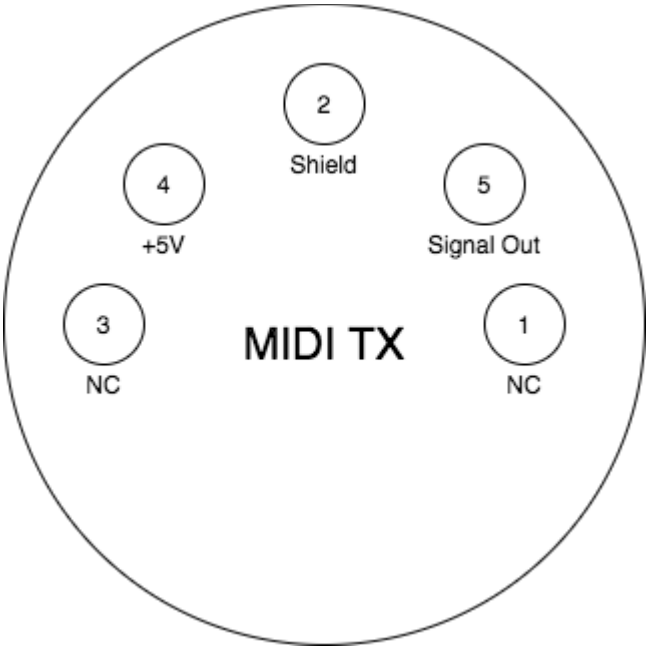


Figure 5-16: MIDI Transmitter Diagram and Pin Assignment

5.4. USB Controller Connections

The USB controller has 8 pins that are used. Four of these pins are connected directly to the pins on the USB port. Two of these pins are connected directly to a power source. The rest are connected to the MCU either directly or through a pair of shift registers. This depends on whether the results of testing show that the MCU can drive the transceiver at 12 MHz. Table 5-5 shows the functions and the connections associated with each pin. Unlisted pins are to remain unconnected. The names in the connection column are names of nets that will be used in the schematic for the digital circuits.

Functions	Purposes	Nets	Pins
VSS	Ground Logic Low	GND	14
VDD	Power Logic High	5VUSBCTLIN	1
VUSB3V3	3.3 V Output	3V3USBCTLOUT	11
USB D+	USB Data + Wire	USBDP	13
USB D-	USB Data - Wire	USBDM	12
Digital I/O	Send & Receive Signals	POWERSRC	2, 3, 5, 6
		USBCONN	
		USBSUSP	
SPI Clock	Receive MIDI Data	SPICLK	10
SPI MOSI		SPIMOSI	9
SPI MISO		SPIMISO	8
SPI Slave Select		GND	7
VPP	Programming Voltage	VPPUSBCTL	4
ICSP Clock	Programming Clock	ICSPCLK	9
ICSP Data	Programming Data	ICSPDATA	10

Table 5-5: List of connections for the USB transceiver.

5.5. Processor Connections

The MCU has in total 80 pins, of which few need to be used. There are four power pins, each of which need to be connected to 3.3 V. We only need to use a maximum of 8 I/O pins, 1 ADC input pin, 1 voltage reference pin, 1 UART transmitter pin, 1 set of SPI pins, and the programming and debugging pins. The I/O pins are used to send or receive signals from other systems within the device (such as the power multiplexing circuits). The ADC input pin receives the buffered and amplified audio signal. The voltage reference pin outputs a 1.2 V reference that is used by the analog preamplifier. The UART transmitter pin is used to output serial data that is fed back into the MCU, inverted, then sent out through the MIDI output port. All unused data pins are to be left unconnected. Table 5-6 shows the functions and the connections associated with the pins that we will use. Pins that are not listed are not used and must remain unconnected. The POWERSRC net is used

by the power multiplexing circuit to report whether the barrel jack power input or the USB connection is the source of power. The USBCONN net is used by the USB controller to notify the MCU that the USB port is connected, and that MIDI data can be sent through it. The USBSUSP net is used by the USB controller to notify the MCU that the USB connection is suspended and that it must decrease its power consumption. The MIDIDATA net carries the inverted UART output signal (UARTDATA). Every pin is listed in order of the function name given to each pin in the datasheet. The digital I/O pins are arranged vertically in sets of up to 8 pins, where each set is one physical 8-bit port. The SPI pins are arranged horizontally such that each vertical set of 3 pins is one SPI port.

Functions	Purposes	Nets	Pins
VSS	Ground Logic Low	GND	60, 19, 39
VCC	Power Logic High	3.3VFORMCU	61, 20, 40
UART Transmitter	Send Thru MIDI	UARTDATA	41, 35, 65, 8
SPI Clock	Send Thru USB	SPICLK	18, 64, 67, 10, 43, 55, 25, 71
SPI MOSI		SPIMOSI	41, 35, 65, 8, 51, 53, 13, 69
SPI MISO		SPIMISO	42, 36, 66, 9, 52, 54, 14, 70
Digital I/O	Receive Signals	POWERSRC	1, 2, 3, 16, 17, 18, 51, 52 41, 42, 43, 63, 64, 35, 36, 62 4, 5, 6, 7, 47, 48, 49, 50 31, 32, 33, 34, 57, 58, 59, 12 53, 54, 55, 56, 65, 66, 67, 68 8, 9, 10, 11, 69, 70, 71, 72 13, 14, 25, 26, 27, 28, 29, 30 15, 44, 45, 46
		USBCONN	
		USBSUSP	
		UARTDATA	
	Send Signals	MIDIDATA	
Analog I/O	Analog Audio	PREAMPOUT	1, 2, 3, 16, 17, 18, 63, 64, 31, 32, 33, 34, 4, 5, 6, 7, 27, 28, 29, 30
AVSS	Analog Low	GND	79
AVCC	Analog High	3.3VFORMCU	80
Internal Reference Output	1.2 V Reference	1V2REF	2
Test SBW Debug Clock	Debugging Programming	SBWDCLK	37
Reset	Reset	SBWDDATA	38

SBW Debug Data	Debugging Programming		

Table 5-6: Notable pins functions and connections

5.6. Complete Integrated System

Over the course of this project, all four group members contributed schematics for their hardware designs which were combined into a single EAGLE schematic document. The hardware blocks that make up the Polyphonic Analog to MIDI Converter were sorted into four pages: Connectors, Power, Preamp, and MCU. Though, the connectors page ended up rather sparse, as most connectors ended up on either the Preamp or MCU pages for convenience. The schematics are shown in the following figures.

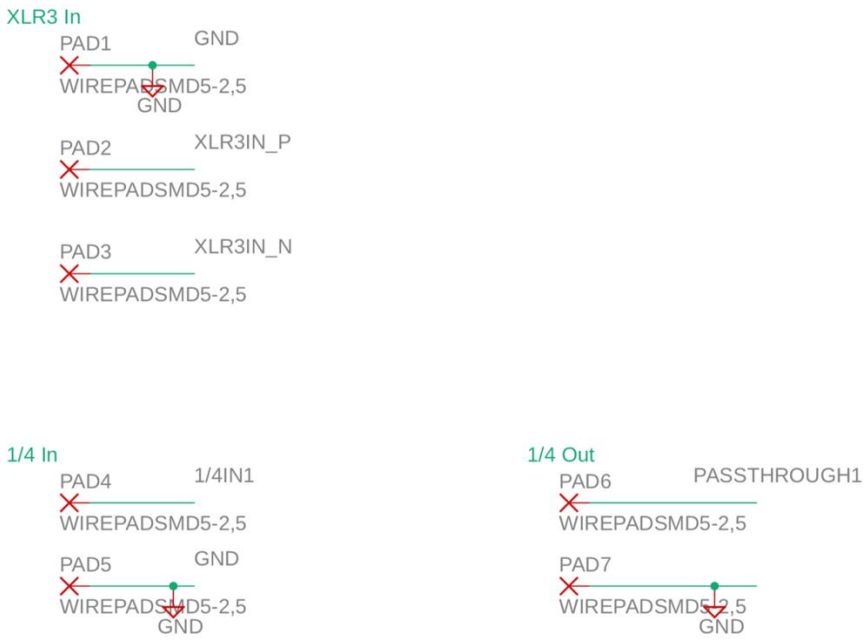


Figure 9-1: The connectors page shows the wire pads that are used to attach the ¼" jacks to the board. Three XLR pads are also shown but were not used in the final design.

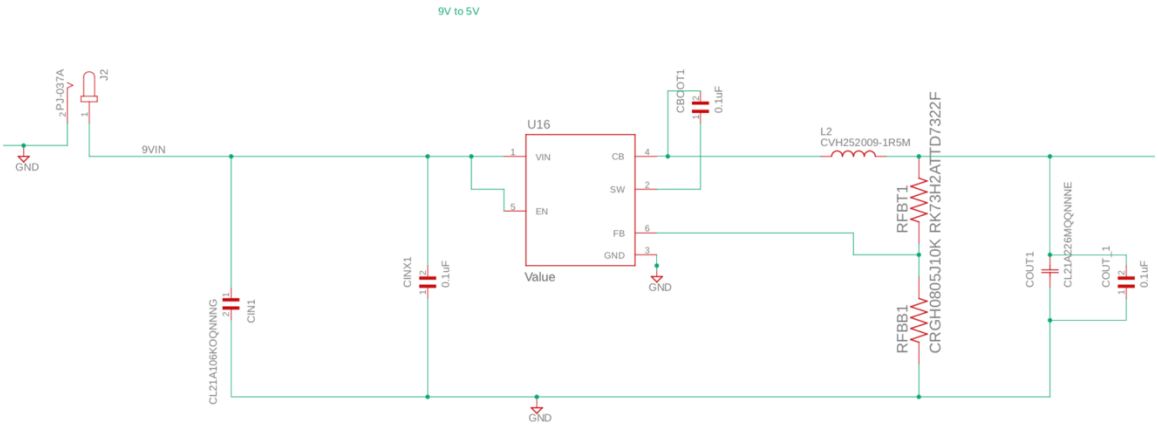


Figure 9-2: The 9V to 5V power converter on the power page, based around the TPS563231DRLR chip.

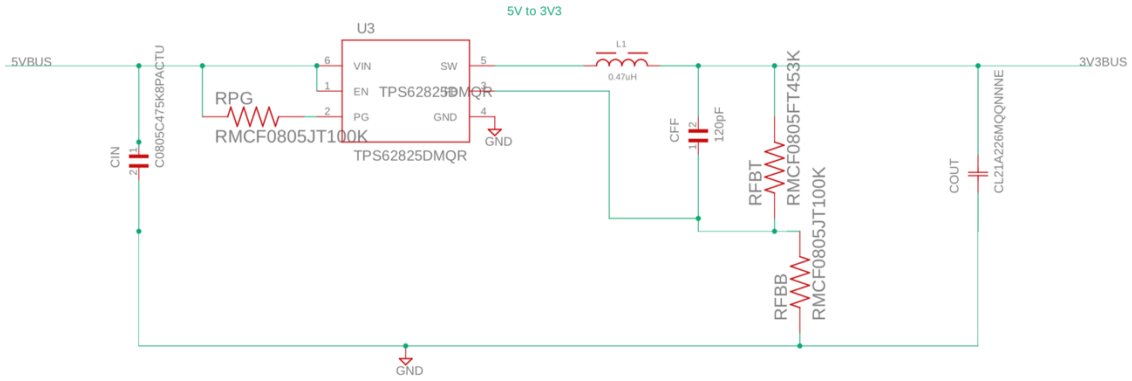


Figure 9-3: The 5V to 3V3 converter on the power page, based around the TPS62825DMQR chip.

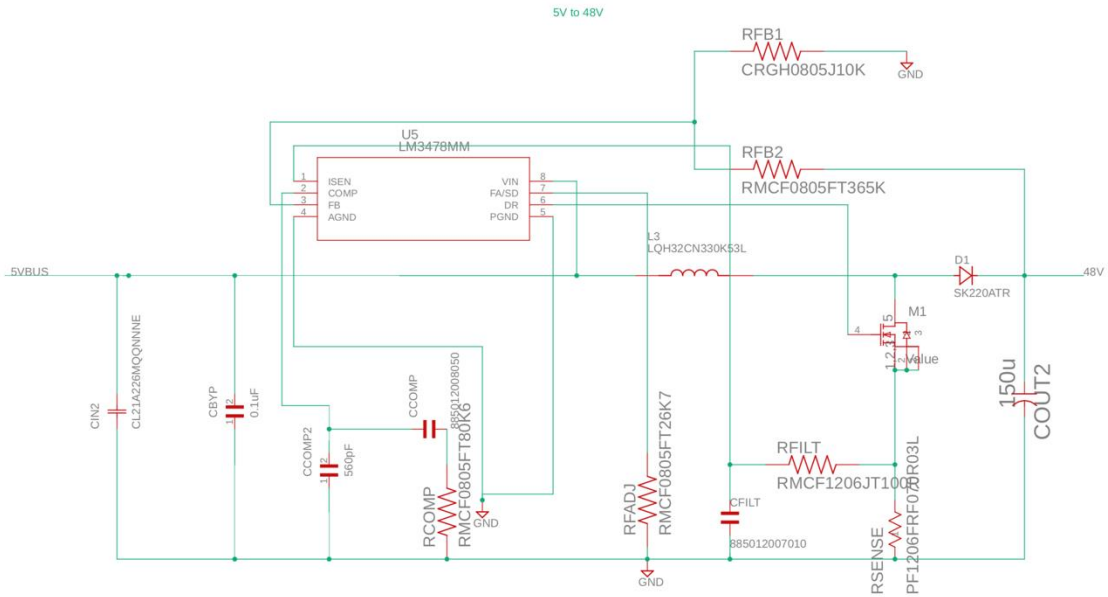


Figure 9-4: The 5V to 48V converter on the power page for phantom power, based on the LM3478MM chip.

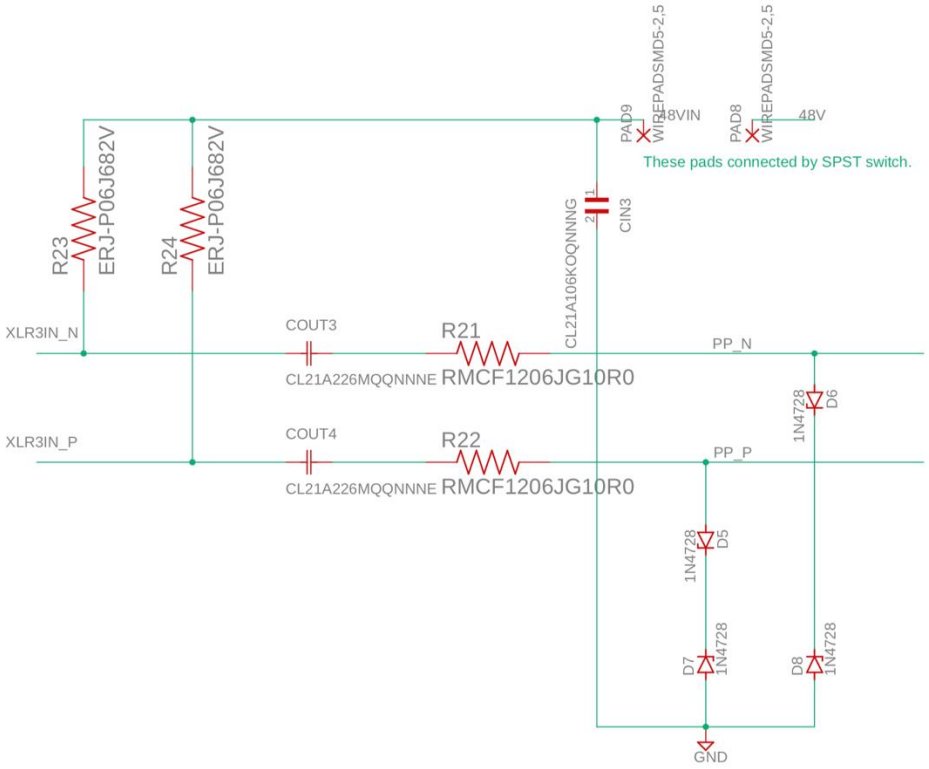


Figure 9-5: The phantom power circuit on the power page, to deliver 48V power to the XLR lines for condenser microphones.

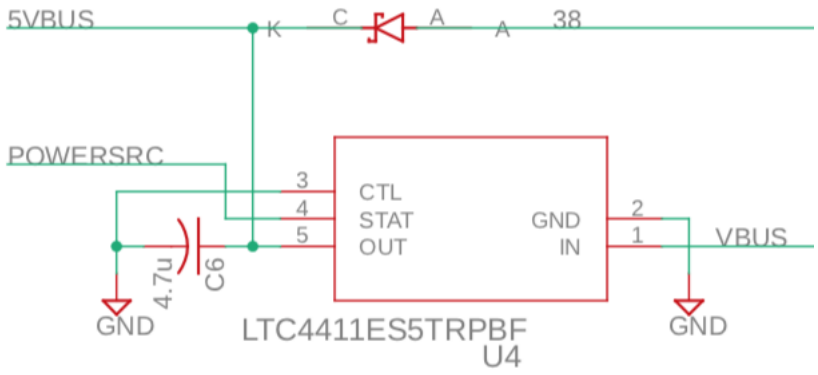


Figure 9-6: The power select circuit on the power page, connected to the USB 5Vin and to the output of the 9V to 5V circuit. Based on the LTC4411ES5TRPBF chip.

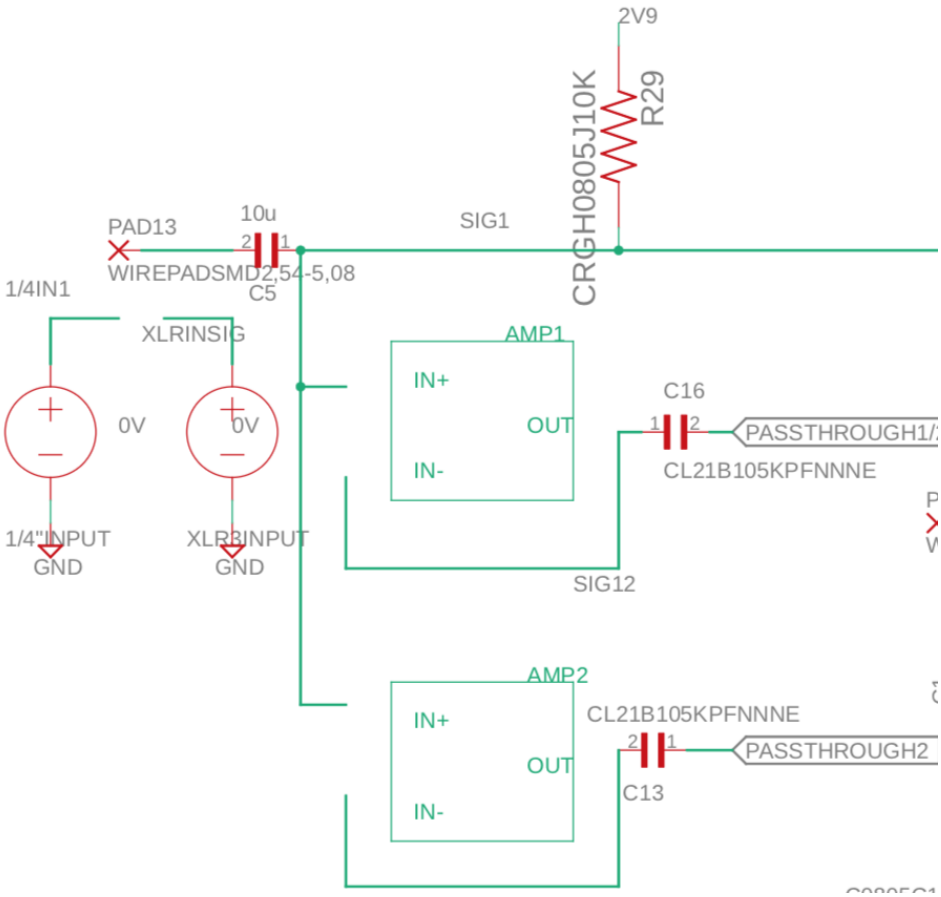


Figure 9-7: Input of the preamp section and op-amp buffer passthroughs. Originally two inputs were going to be selectable for ¼" and XLR.

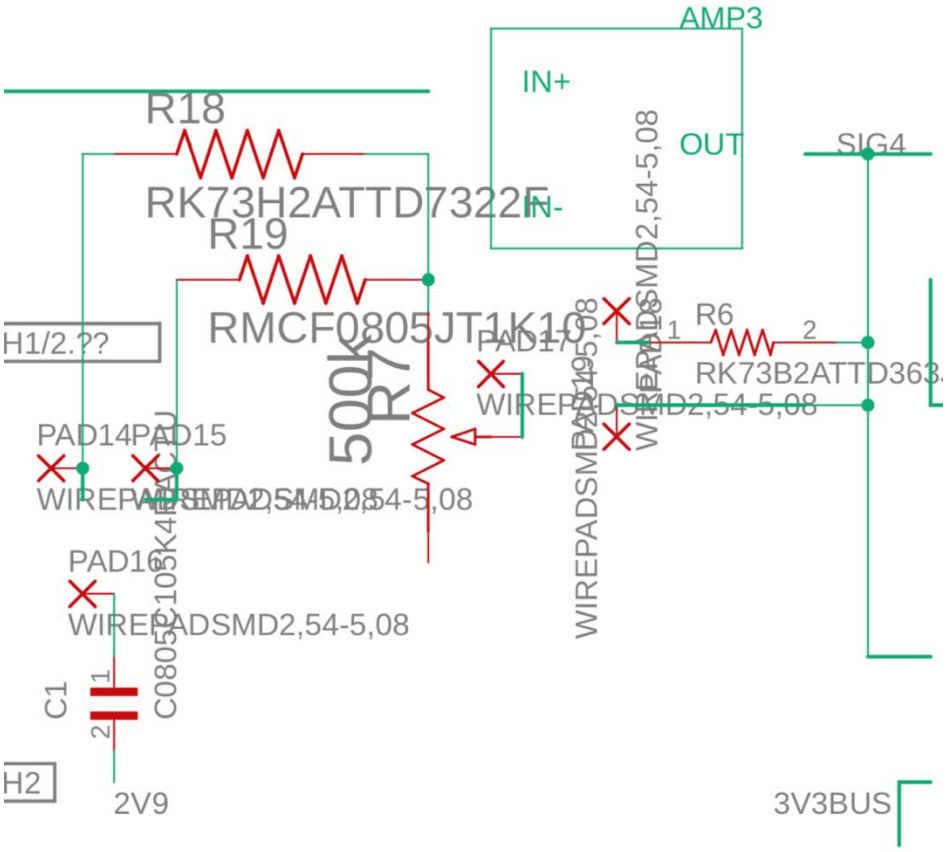


Figure 9-8: Noninverting amplifier in the preamp section.

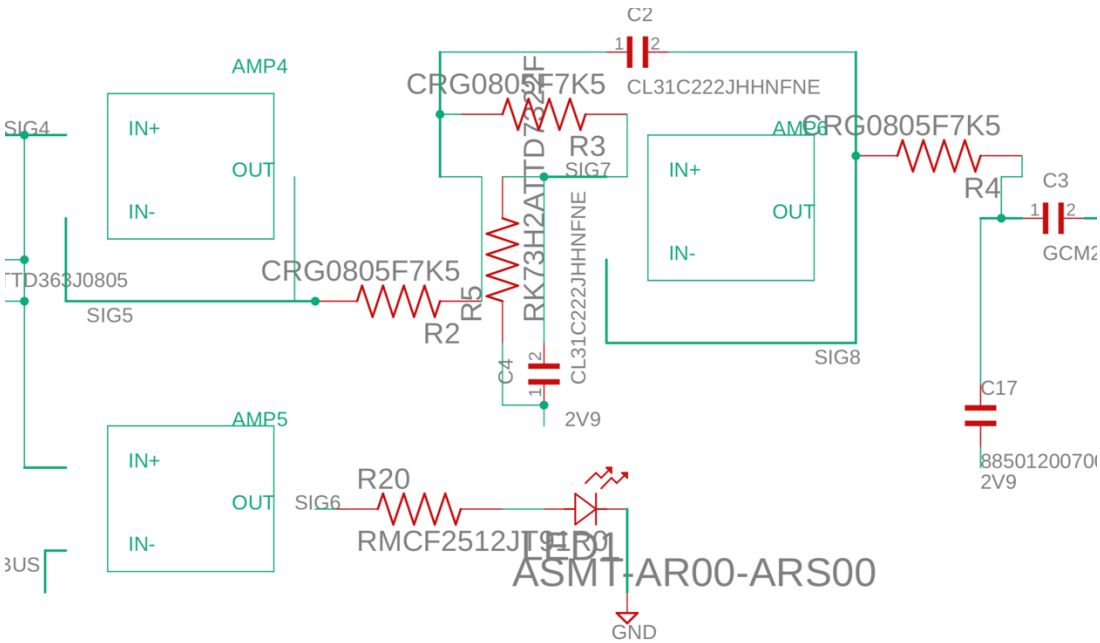


Figure 9-9: Buffer, Clipping-detect LED circuit, and first stage of multi-stage active bandpass filter in the preamp section.

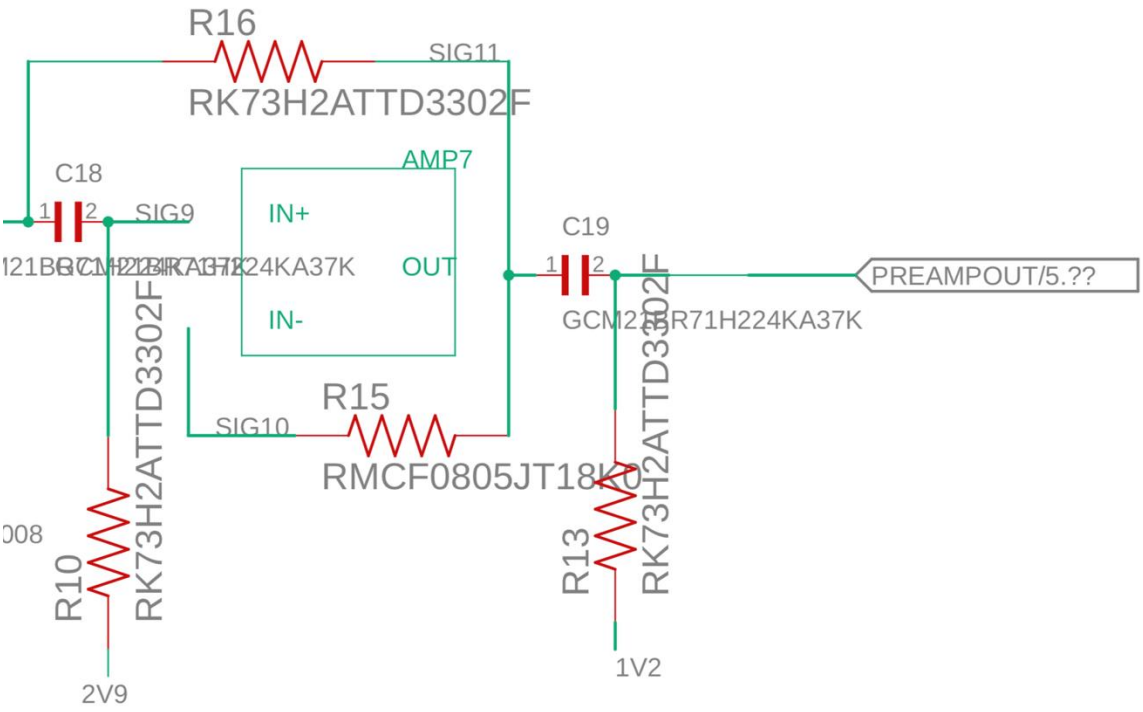


Figure 9-10: Second stage and output of active band-pass filter in the preamp section.

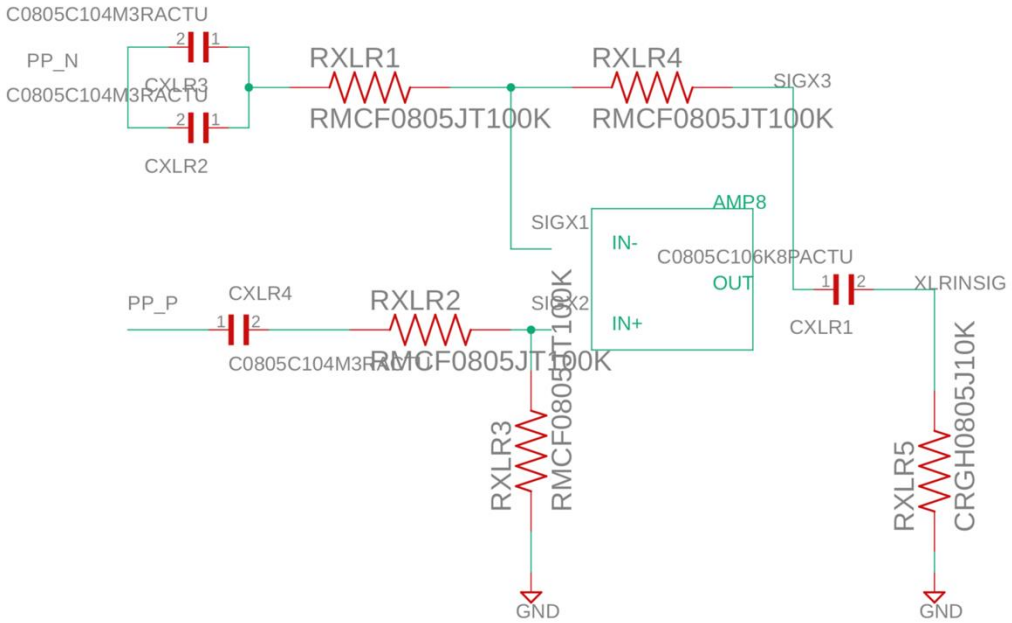


Figure 9-11: XLR input circuit in the preamp section. This differential amplifier takes the balanced signal of the XLR input and outputs a single signal.

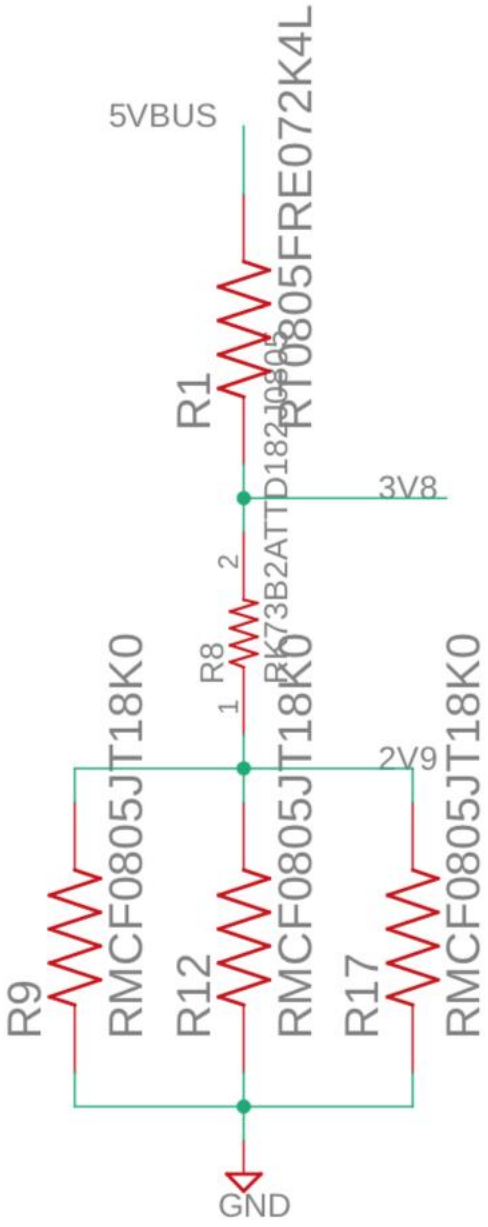


Figure 9-12: Voltage divider in the preamp section. Generates the 2.9V and 3.8V used for references in the preamp section.

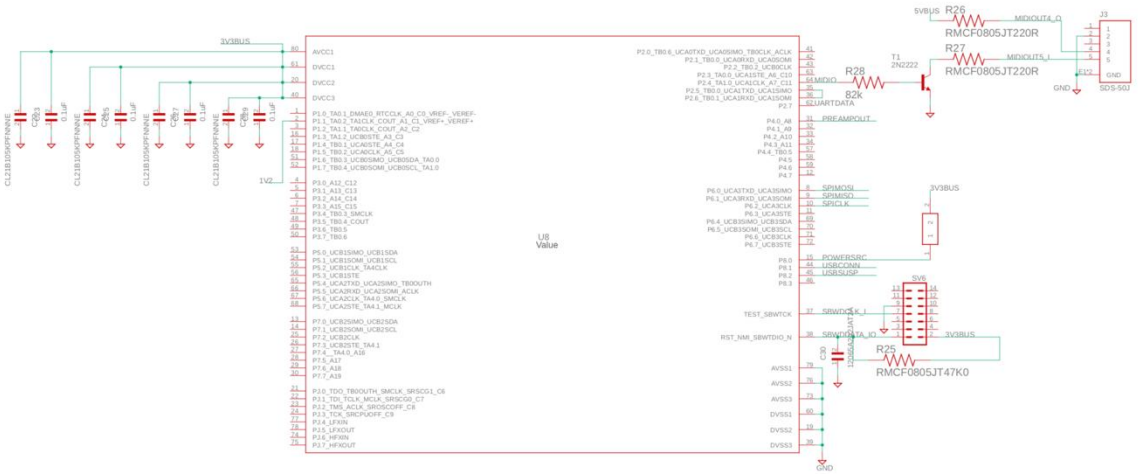


Figure 9-13: MCU and supporting circuitry, including power capacitors, JTAG programming pin header, and MIDI output transistor circuit.

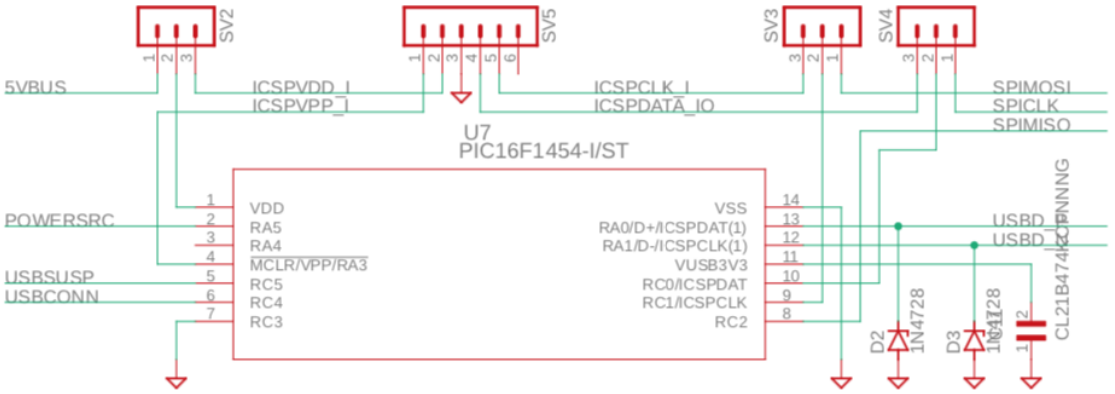


Figure 9-14: USB controller and pin headers for programming on MCU sheet.

Some of the schematics that were imported to EAGLE came from LT Spice and TI Webench had issues. Net ties that were present in Webench disappeared in EAGLE and placeholder parts from LT Spice had to be replaced. On top of that, revisions had to be imported from LT Spice throughout the entire length of the project as collaborators sent in revised schematics. This led to some issues with the schematic, most of which were discovered and corrected but a few remained in the final version. This includes the misplaced net tie in Figure 9-2 which should have had the inductor L2 attached to pin 2 of the chip U16 or the lack of a 3.8V reference in the preamp circuit. These mistakes were corrected using wires to connect parts to different nets.

The design that we came up with for this board contains hardware that was not used in the final presentation of the device, namely the XLR input circuit, USB controller, and the power section. The parts listed in our BOM in table 9.1 reflect this fully featured version of the device.

Part	Value	Device	Package	Description
C1	C0805C105K4RACTU	C0805C105K4RACTU	CAPC2012X88N	SMD 0805 CERAMIC X7R -55^+125 1 UF +/-10? 16V Check prices
C2	CL31C222JHHNFNE	CL31C222JHHNFNE	1206	CL31C222JHHNFNE
C3	GCM21BR71H224KA37K	GCM21BR71H224KA37K	CAPC2012X140N	Check prices
C4	CL31C222JHHNFNE	CL31C222JHHNFNE	1206	CL31C222JHHNFNE
C5	10u	CL21A106KOQNNNG	CAPC2012X140N	Check prices
C6	4.7u	C-USC0805	C0805	CAPACITOR, American symbol
C11	CL21B474KOFNNNG	CL21B474KOFNNNG	CAPC2012X135N	
C13	CL21B105KPFNNNE	CL21B105KPFNNNE	C0805K	
C16	CL21B105KPFNNNE	CL21B105KPFNNNE	C0805K	
C17	885012007008	885012007008	WCAP-CSGP_885012007008	WCAP-CSGP Ceramic Capacitors
C18	GCM21BR71H224KA37K	GCM21BR71H224KA37K	CAPC2012X140N	Check prices
C19	GCM21BR71H224KA37K	GCM21BR71H224KA37K	CAPC2012X140N	Check prices
C22	CL21B105KPFNNNE	CL21B105KPFNNNE	C0805K	
C23	0.1uF	CC0805JRX7R7BB104	C0805K	
C24	CL21B105KPFNNNE	CL21B105KPFNNNE	C0805K	
C25	0.1uF	CC0805JRX7R7BB104	C0805K	
C26	CL21B105KPFNNNE	CL21B105KPFNNNE	C0805K	
C27	0.1uF	CC0805JRX7R7BB104	C0805K	
C28	CL21B105KPFNNNE	CL21B105KPFNNNE	C0805K	
C29	0.1uF	CC0805JRX7R7BB104	C0805K	
C30	12065A202JAT2A	12065A202JAT2A	CAPC3216X127N	Cap Ceramic 0.002uF 50V C0G 5% SMD 1206 125???□□ T/R Check prices
CBOOT1	0.1uF	CC0805JRX7R7BB104	C0805K	
CBYP	0.1uF	CC0805JRX7R7BB104	C0805K	
CCOMP	885012008050	885012008050	WCAP-CSGP_885012008050	WCAP-CSGP Ceramic Capacitors
CCOMP2	560pF	CL21C561JBANNNC	C0805K	
CFF	120pF	CC0805JRNPO9BN121	C0805K	
CFILT	885012007010	885012007010	WCAP-CSGP_885012007010	WCAP-CSGP Ceramic Capacitors
CIN	C0805C475K8PACTU	C0805C475K8PACTU	CAPC2012X88N	Cap Ceramic 4.7uF 10V X5R 10% SMD 0805 85?C Plastic T/R Check prices
CIN1	CL21A106KOQNNNG	CL21A106KOQNNNG	CAPC2012X140N	Check prices
CIN2	CL21A226MQQNNNE	CL21A226MQQNNNE	CAP_CL21_SAM	
CIN3	CL21A106KOQNNNG	CL21A106KOQNNNG	CAPC2012X140N	Check prices
CINX1	0.1uF	CC0805JRX7R7BB104	C0805K	
COUT	CL21A226MQQNNNE	CL21A226MQQNNNE	CAP_CL21_SAM	

COUT1	CL21A226MQQNNNE	CL21A226MQQNNNE	CAP_CL21_SAM	
COUT2	150u	UUD1H151MNL1GS	U2-R_NCH	
COUT3	CL21A226MQQNNNE	CL21A226MQQNNNE	CAP_CL21_SAM	
COUT4	CL21A226MQQNNNE	CL21A226MQQNNNE	CAP_CL21_SAM	
COUT_1	0.1uF	CC0805JRX7R7BB104	C0805K	
CXLR1	C0805C106K8PACTU	C0805C106K8PACTU	CAPC2012X88N	
CXLR2	C0805C104M3RACTU	C0805C104M3RACTU	CAPC2012X88N	.1UF 25V 20% 080
CXLR3	C0805C104M3RACTU	C0805C104M3RACTU	CAPC2012X88N	.1UF 25V 20% 080
CXLR4	C0805C104M3RACTU	C0805C104M3RACTU	CAPC2012X88N	.1UF 25V 20% 080
D1	SK220ATR	SK220ATR	DIO_ES1J	
D2	1N4728	1N4728	DO41Z10	Z DIODE
D3	1N4728	1N4728	DO41Z10	Z DIODE
D5	1N4728	1N4728	DO41Z10	Z DIODE
D6	1N4728	1N4728	DO41Z10	Z DIODE
D7	1N4728	1N4728	DO41Z10	Z DIODE
D8	1N4728	1N4728	DO41Z10	Z DIODE
D9	SCHOTTKY	SCHOTTKY	DO-214AC	DIODE
J2	PJ-037A	PJ-037A	CUI_PJ-037A	2.0 mm Center Pin, 2.5 A, Right Angle, Through Hole, Dc Power Jack Connector Buy Part
J3	SDS-50J	SDS-50J	CUI_SDS-50J	Check prices
L1	0.47uH	WE-HCI_7050_744314047	WE-HCI_7050	WE-HCI SMD Flat Wire High Current Inductor
L2	CVH252009-1R5M	CVH252009-1R5M	INDC2520X100N	INDUCTOR, 1.5UH, 1.5A, +20%, 50MHZ Check prices
L3	LQH32CN330K53L	LQH32CN330K53L	IND_LQH32CN330K53L	Wire Wound Ferrite Inductor for Power Lines For Automotive Check prices
LED1	ASMT-AR00-ARS00	ASMT-AR00-ARS00	LED_ARS00	
M1	Value	CSD18543Q3A	DNH0008A	
PAD1	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD2	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD3	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD4	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD5	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD6	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD7	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD8	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD9	WIREPADSMD5-2,5	WIREPADSMD5-2,5	5-2,5	Wire PAD connect wire on PCB
PAD13	WIREPADSMD2,54-5,08	WIREPADSMD2,54-5,08	SMD2,54-5,08	Wire PAD connect wire on PCB

PAD14	WIREPADSMD2,54-5,08	WIREPADSMD2,54-5,08	SMD2,54-5,08	Wire PAD connect wire on PCB
PAD15	WIREPADSMD2,54-5,08	WIREPADSMD2,54-5,08	SMD2,54-5,08	Wire PAD connect wire on PCB
PAD16	WIREPADSMD2,54-5,08	WIREPADSMD2,54-5,08	SMD2,54-5,08	Wire PAD connect wire on PCB
PAD17	WIREPADSMD2,54-5,08	WIREPADSMD2,54-5,08	SMD2,54-5,08	Wire PAD connect wire on PCB
PAD18	WIREPADSMD2,54-5,08	WIREPADSMD2,54-5,08	SMD2,54-5,08	Wire PAD connect wire on PCB
PAD19	WIREPADSMD2,54-5,08	WIREPADSMD2,54-5,08	SMD2,54-5,08	Wire PAD connect wire on PCB
R1	RT0805FRE072K4L	RT0805FRE072K4L	RESC2012X60N	Res Thin Film 0805 2.4K Ohm 1% 1/8W ±50ppm/°C Molded SMD SMD Paper T/R Check prices
R2	CRG0805F7K5	CRG0805F7K5	RESC2012X65N	CRG0805 1% 7K5
R3	CRG0805F7K5	CRG0805F7K5	RESC2012X65N	CRG0805 1% 7K5
R4	CRG0805F7K5	CRG0805F7K5	RESC2012X65N	CRG0805 1% 7K5
R5	RK73H2ATTD7322F	RK73H2ATTD7322F	RESC2012X60N	Res Thick Film 0805 73.2K Ohm 1% 0.25W(1/4W) ±100ppm/°C Pad SMD Automotive T/R Check prices
R6	RK73B2ATTD363J0805	RK73B2ATTD363J0805	R0805	Res Thick Film 0805 36K Ohm 5% 0.25W(1/4W) ?200ppm/?C SMD Automotive T/R Check prices
R7	500k	3314J-1-103E	POT_3314J	
R8	RK73B2ATTD182J0805	RK73B2ATTD182J0805	R0805	Res Thick Film 0805 1.8K Ohm 5% 0.25W(1/4W) ?200ppm/?C SMD Automotive T/R Check prices
R9	RMCF0805JT18K0	RMCF0805JT18K0	RESC2012X65N	Check prices
R10	RK73H2ATTD3302F	RK73H2ATTD3302F	RESC2012X60N	33 kOhms ±1% 0.25W, 1/4W Chip Resistor 0805 (2012 Metric) Automotive AEC-Q200, Moisture Resistant Thick Film Check prices
R12	RMCF0805JT18K0	RMCF0805JT18K0	RESC2012X65N	Check prices
R13	RK73H2ATTD3302F	RK73H2ATTD3302F	RESC2012X60N	33 kOhms ±1% 0.25W, 1/4W Chip Resistor 0805 (2012 Metric) Automotive AEC-Q200, Moisture Resistant Thick Film Check prices
R15	RMCF0805JT18K0	RMCF0805JT18K0	RESC2012X65N	Check prices
R16	RK73H2ATTD3302F	RK73H2ATTD3302F	RESC2012X60N	33 kOhms ±1% 0.25W, 1/4W Chip Resistor 0805 (2012 Metric) Automotive AEC-Q200, Moisture Resistant Thick Film Check prices
R17	RMCF0805JT18K0	RMCF0805JT18K0	RESC2012X65N	Check prices
R18	RK73H2ATTD7322F	RK73H2ATTD7322F	RESC2012X60N	Res Thick Film 0805 73.2K Ohm 1% 0.25W(1/4W) ±100ppm/°C Pad SMD Automotive T/R Check prices
R19	RMCF0805JT1K10	RMCF0805JT1K10	RESC2012X65N	Check prices
R20	RMCF2512JT91R0	RMCF2512JT91R0	RESC6332X70N	
R21	RMCF1206JG10R0	RMCF1206JG10R0	RESC3216X70N	Check prices
R22	RMCF1206JG10R0	RMCF1206JG10R0	RESC3216X70N	Check prices
R23	ERJ-P06J682V	ERJ-P06J682V	RESC2012X70N	6.8K OHM 5% 1/2W Check prices

R24	ERJ-P06J682V	ERJ-P06J682V	RESC2012X70N	6.8K OHM 5% 1/2W Check prices
R25	RMCF0805JT47K0	RMCF0805JT47K0	RESC2012X65N	Check prices
R26	RMCF0805JT220R	RMCF0805JT220R	RESC2012X65N	Check prices
R27	RMCF0805JT220R	RMCF0805JT220R	RESC2012X65N	Check prices
R28	82k	RMCF0805JT82K0	RESC2012X65N	
R29	CRGH0805J10K	CRGH0805J10K	RESC2012X65N	Res Thick Film 0805 10K Ohm 5% 1/3W ±100ppm/°C Molded SMD SMD T/R Check prices
RCOMP	RMCF0805FT80K6	RMCF0805FT80K6	RESC2012X65N	Check prices
RFADJ	RMCF0805FT26K7	RMCF0805FT26K7	RESC2012X65N	Check prices
RFB1	CRGH0805J10K	CRGH0805J10K	RESC2012X65N	Res Thick Film 0805 10K Ohm 5% 1/3W ±100ppm/°C Molded SMD SMD T/R Check prices
RFB2	RMCF0805FT365K	RMCF0805FT365K	RESC2012X65N	Check prices
RFBB	RMCF0805JT100K	RMCF0805JT100K	RESC2012X65N	Check prices
RFBB1	CRGH0805J10K	CRGH0805J10K	RESC2012X65N	Res Thick Film 0805 10K Ohm 5% 1/3W ±100ppm/°C Molded SMD SMD T/R Check prices
RFBT	RMCF0805FT453K	RMCF0805FT453K	RESC2012X65N	Check prices
RFBT1	RK73H2ATTD7322F	RK73H2ATTD7322F	RESC2012X60N	Res Thick Film 0805 73.2K Ohm 1% 0.25W(1/4W) ±100ppm/°C Pad SMD Automotive T/R Check prices
RFILT	RMCF1206JT100R	RMCF1206JT100R	RESC3216X70N	Check prices
RPG	RMCF0805JT100K	RMCF0805JT100K	RESC2012X65N	Check prices
RPU2	RESC2010X45N	RESC2010X45N	RESC2010X45N	
RSENSE	PF1206FRF070R03L	PF1206FRF070R03L	YAG_PF1206_YAG	
RXLR1	RMCF0805JT100K	RMCF0805JT100K	RESC2012X65N	Check prices
RXLR2	RMCF0805JT100K	RMCF0805JT100K	RESC2012X65N	Check prices
RXLR3	RMCF0805JT100K	RMCF0805JT100K	RESC2012X65N	Check prices
RXLR4	RMCF0805JT100K	RMCF0805JT100K	RESC2012X65N	Check prices
RXLR5	CRGH0805J10K	CRGH0805J10K	RESC2012X65N	Res Thick Film 0805 10K Ohm 5% 1/3W ±100ppm/°C Molded SMD SMD T/R Check prices
SV2		MA03-1	MA03-1	PIN HEADER
SV3		MA03-1	MA03-1	PIN HEADER
SV4		MA03-1	MA03-1	PIN HEADER
SV5		MA06-1	MA06-1	PIN HEADER
SV6		MA07-2	MA07-2	PIN HEADER
T1	2N2222	2N2222	TO18	NPN TRANSISTOR
U$2	1003P3T1B1M1QE	1003P3T1B1M1QE	NOFP	
U3	TPS62825DMQR	TPS62825DMQR	DMQ0006A_TEX	
U4	LTC4411ES5TRPBF	LTC4411ES5TRPBF	S_5_ADI	

U5	LM3478MM	LM3478MM	MUA08A	
U6	NE5532D	NE5532D	SOIC127P599X175-8N	Check prices
U7	PIC16F1454-I/ST	PIC16F1454-I/ST	TSSOP14_MC_MCH	
U8	Value	MSP430FR5992IPNR	PN0080A_N	
U9	NE5532D	NE5532D	SOIC127P599X175-8N	Check prices
U10	NE5532D	NE5532D	SOIC127P599X175-8N	Check prices
U11	NE5532D	NE5532D	SOIC127P599X175-8N	Check prices
U16	Value	TPS563231DRLR	DRL0006A	
X2	USB-B-H	USB-B-H	USB-B-H	USB Connectors

Table 9.1: PAMC Bill of Materials

5.7. PCB Design

The parts we are using on our PCB are small and simple to connect for the most part. Our processor only has 64 pins and only a fraction of those pins are being used, so routing traces away from the processor will be relatively simple. We should only need one or two signal layers to lay out all of the parts and traces that we need, so a standard four layer stackup will be perfect for our design. Four layers will also provide us with a total thickness of about 1.6mm. This is good for sturdiness as the board will have several connectors mounted to it that the user will be plugging into and unplugging from regularly. The proposed stackup is shown in Figure 5-16.

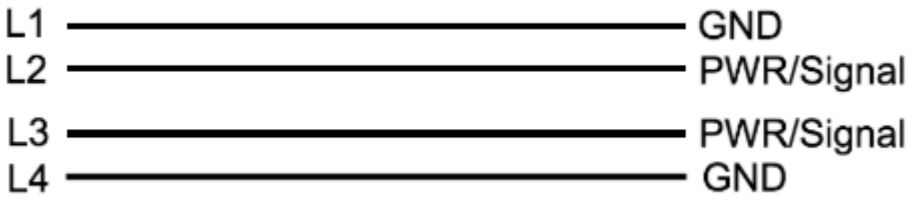


Figure 5-17: Four Layer PCB Stackup

There are some important advantages to designing the board stackup like this, with ground planes on the top and bottom of the board and signal and power layers sandwiched inside. Crosstalk is a serious issue for ICs operating at a megahertz frequency. It is recommended to reduce cross coupling to approximately 10% that signal traces on the surface of a PCB are spaced apart by twice the distance from the nearest ground plane. By making the traces internal and surrounded by the PCB's dielectric material, the traces need only be spaced apart by a distance equivalent to the distance from the nearest ground plane. Additionally, having two signal and power layers one right after the other is not a big issue for this stackup because in a standard four layer PCB, the dielectric core between layers one and two is thicker than the other two dielectric layers. Another advantage to this design over a stackup where the signal layers are on the top and bottom surfaces of the board is that it is easier for a trace to change layers. There is no need to drop vias awkwardly through ground and power planes when the signal layers are adjacent.

The PCB design was relatively straightforward once the schematic was created. EAGLE allows the user to generate a PCB file from a schematic that contains all parts of the schematic. Each hardware block was laid out individually and then combined on the PCB. The individual hardware blocks are: The preamp, MCU, MIDI output circuit, phantom power circuit, 9V to 5V converter, 5V to 3V3 converter, 5V to 48V converter, power select circuit, and USB controller. In addition to these blocks, there were three PCB-mounted connectors and seven wire pads to be placed. The board was two layers and as much routing as possible was done on the top surface of the board to make the ground as consistent as possible on the bottom surface. Unfortunately, some bottom layer routing was required for complicated hardware blocks such as the USB controller and the preamp, but the grounding on the board ended up being enough regardless.

We ordered our PCB from 4PCB in Colorado. They offered a two-layer board express option that we took advantage of and we ordered four copies to ensure that we would not run out due to hardware failures or other unfortunate circumstances. We were behind schedule for ordering the PCB, so we could not afford to take the chance of needing to reorder. Once we had the boards, two local companies assisted us in assembly. Our first board was assembled with the help of Quality Manufacturing Services in Oviedo. They offer free services to UCF students to mount difficult parts onto the PCB. Our second board was assembled in part by the manufacturing team at Astronics Test Systems in Central Florida Research Park who we have connections to. The assistance we received from both companies was key to our success in PCB assembly and integration, as we lack the technical skill and equipment to solder QFN packages and fine-pitch ICs like our 80-pin MSP430.

6. Software Design Details

The software design can be broken down into 4 main components. These components are the input section, the Fourier transform section, filtering, and frequency analysis section, and finally the output section. Each of these sections will rely on different technologies but all serve the purpose of digital signal processing. This processing starts with the digitally converted input signal from our microphone or instrument and ends with a MIDI signal in the output along a MIDI cable or USB cable using MIDI protocol. The input section involves processing the input from the analog to digital converter and storing the input as samples in memory. The actual implementation of this will vary depending on our final hardware choices. If we have a discrete analog to digital converter, we will have to write a low-level driver to decode the protocol of the analog to digital converter and make the data usable by the microcontroller. If the analog to digital converter is built into the microcontroller then we might not have to write a low-level driver as it may include functionality to interpret the input on the chip. Either way we will have to write a high-level driver to be able to save data from the analog to digital converter in a usable memory space as a usable data type. Ideally our processor will have enough memory to store all of our samples without needing external memory as that will add added complexity. The samples will be saved for the next Fourier transform simultaneously as the current set of samples is being processed. This will require twice the memory but will allow us to continuously sample the input and not have to sample, process, sample, process etc. This could make it miss new notes or when a note ends etc. while the block of samples is being

processed. The samples must be saved in blocks at least large enough to recognize our lowest required notes.

The goal of the Fourier transform section is to convert the digital signal from the time domain to the frequency domain. This is so that we can process the frequencies and determine the notes being played. The Fourier transform section first adds zeros into the set of samples to pad the signal and provide a higher resolution in the frequency domain. This will allow us to more accurately identify the bandwidth of peaks and the magnitudes of peaks in the frequency domain. The implementation of the Fourier transform algorithm is a fast Fourier transform similar to the Cooley-Tukey Fast Fourier transform. This will be done using the most efficient means possible to improve calculation speeds, so using bitwise manipulations for instance rather than higher level abstractions that require more data and time. This will result in an array of frequencies and their magnitudes. In the next section we take our input that is now in the frequency domain and analyze it to determine what notes are being played by the input instrument. The first and most easy thing to do is to filter any frequencies below a low magnitude threshold. This will get rid of any frequencies that have a magnitude so low they are irrelevant so they do not affect our analysis in any way.

After this the magnitudes will be normalized to accurately represent the input magnitudes of the frequencies. After this we filter out frequency bands that have a large Q factor with no largely defined peak. With an instrument, the played note will have a very defined frequency without a large amount of fluctuation so the peaks will have a very small Q factor. So any frequency bands with a low Q factor will be very unlikely to be a note and instead be noise or unwanted resonance. Finally, we should be left with any defined frequency peaks and we can determine the musical note(s) being played. These up to six output notes will be constantly kept in an easily accessible space in memory like a register for instance so they can be consistently be output as a stream to our MIDI and usb outputs. Finally, we have our output section of the software. In this section we need to take the notes we determined that were being played and output them via a MIDI protocol through both a MIDI and a USB output. To do this we need to write a driver to convert the data to the MIDI protocol and possibly for the USB as well depending on the functions of our USB controller.

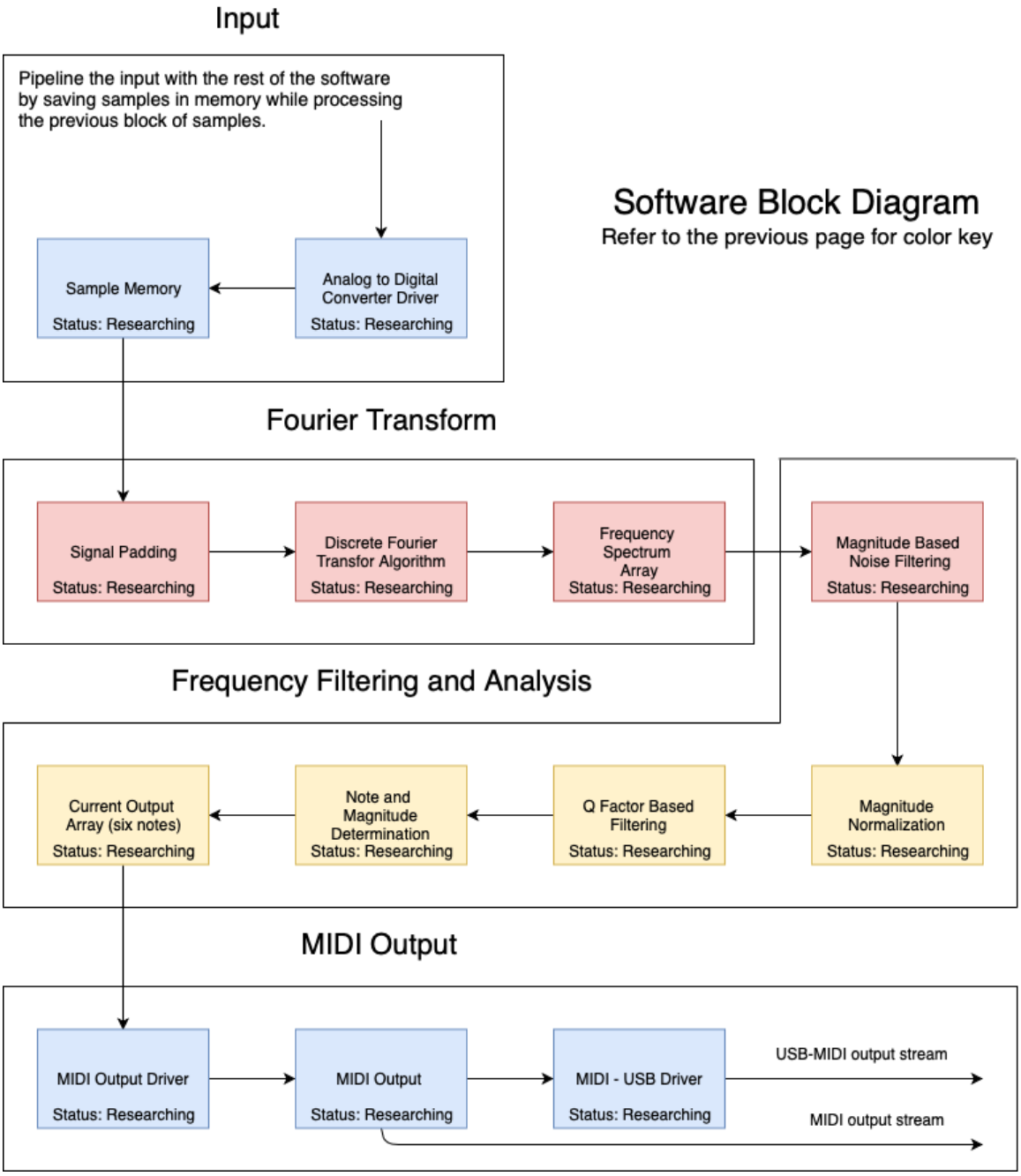


Figure 6.1: Software block diagram.

6.1. ADC Driver and Sample Buffer Generation

The ADC must be set to collect samples at a fixed rate and notify the program when a sample buffer is filled. The ADC is configured at the start of the program to repeatedly collect 12-bit samples at roughly 12.8 kHz. The sample rate is determined by a timer that increments every nanosecond, so the actual sample rate will be slightly off. When the timer reaches its maximum value, it will toggle the state of an output signal. At every rising

edge of the timer, the ADC will take a sample, and at every falling edge, it will convert the analog sample into a digital one. Whenever the ADC has a sample ready, it will call a function that puts it into the sample buffer. This same function is responsible for swapping buffers when the current one is full and notifying the main thread that a new sample buffer is ready for processing. Also, each sample is shifted and scaled to accommodate the mismatch between the preamp output voltage range and center and what the MCU expects.

6.2. Frequency Spectra Generation

It is possible to use an FFT algorithm for complex inputs to obtain a transform for real-valued inputs. The result of doing this is a transform with double the points of the underlying complex transform.[31] The MCU's DSP coprocessor is designed to only perform complex FFTs, so we will use this trick to efficiently perform the FFT. TI provides a library that implements the FFT in this manner. Only a couple of lines of code within that function were required to make it output frequency bin N/2 instead of frequency bin 0. Another thing to notice is that a normal FFT across the entire frequency range that we are working with is very inefficient because there is plenty of unneeded density at the high frequencies. A more efficient method would be to split the FFT into two parts, an FFT with half the bandwidth and half the points, and another FFT with half the bandwidth and a small number of points that covers the high frequencies. This can be repeated for every octave, giving a total of 10 FFTs. If we use 64 points for every octave, then we have a much faster algorithm, like a 640-point FFT in the worst-case scenario.

To be able to do this, the signal buffer must be split into 10 buffers, with each successive buffer holding every other sample of the previous buffer. Also, the digital signal must pass through a low-pass filter before being added to the other buffers. This eliminates aliases that will arise due to undersampling. Using the algorithm introduced in Section 3.3.1, we can determine the number of points and the buffer frequency for the FFT in order to capture one octave range of notes. For this test, we set M = 116 (G#8), K = 1, and the minimum alignment to ⅓ of a semitone. The result is that the minimum number of frequency spectrum bins we can use is 32, and the highest buffer frequency is 502.045 Hz. This means that a 64-point FFT is required to capture one octave range of notes. The resulting sampling frequency required to fill the buffer is 32.13 kHz. However, it is important to note that only 12 bins are used in this spectrum; in fact, the higher bins contain higher notes above G9. At the highest octave, we can use these bins to contain notes, so it would be best to rebase the buffer frequency of the FFT so that the higher bins are useful for the highest octave range. By shifting the first note in the octave range from G#8 to F8, the higher bins in the highest octave range will contain notes up to G9.

The final version of the Frequency Spectrum Generator (FSG) uses three FFT stages that work similarly as mentioned before. The top stage uses a 512-point FFT on the original sample buffer as received from the ADC. This stage is responsible for capturing the mid to high frequencies. The stage below that uses a 64-point FFT on a filtered and downsampled version of the original sample buffer. The filter used is an elliptic low-pass IIR filter with a cutoff frequency below 1/16th the original sampling frequency; 16 is the downsample factor, and the sampling frequency for this stage is 800 Hz. The final stage uses a 64-point FFT on a filtered and downsampled version of the previous stage's sample buffer. The same kind of filter is used, but the cutoff frequency is instead below ½ the sampling frequency of the previous stage; 2 is the downsample factor, and the sampling

frequency for this stage is 400 Hz. After downsampling, the resulting samples are boosted (multiplied by 2).

After the FFTs are done, the magnitudes are calculated for each bin and stored separately into spectra buffers. These buffers are then passed to the note detection algorithm. Figure X shows a flowchart of a previous version of FSG. Although it is outdated, its functionality is very similar to that of the final version.

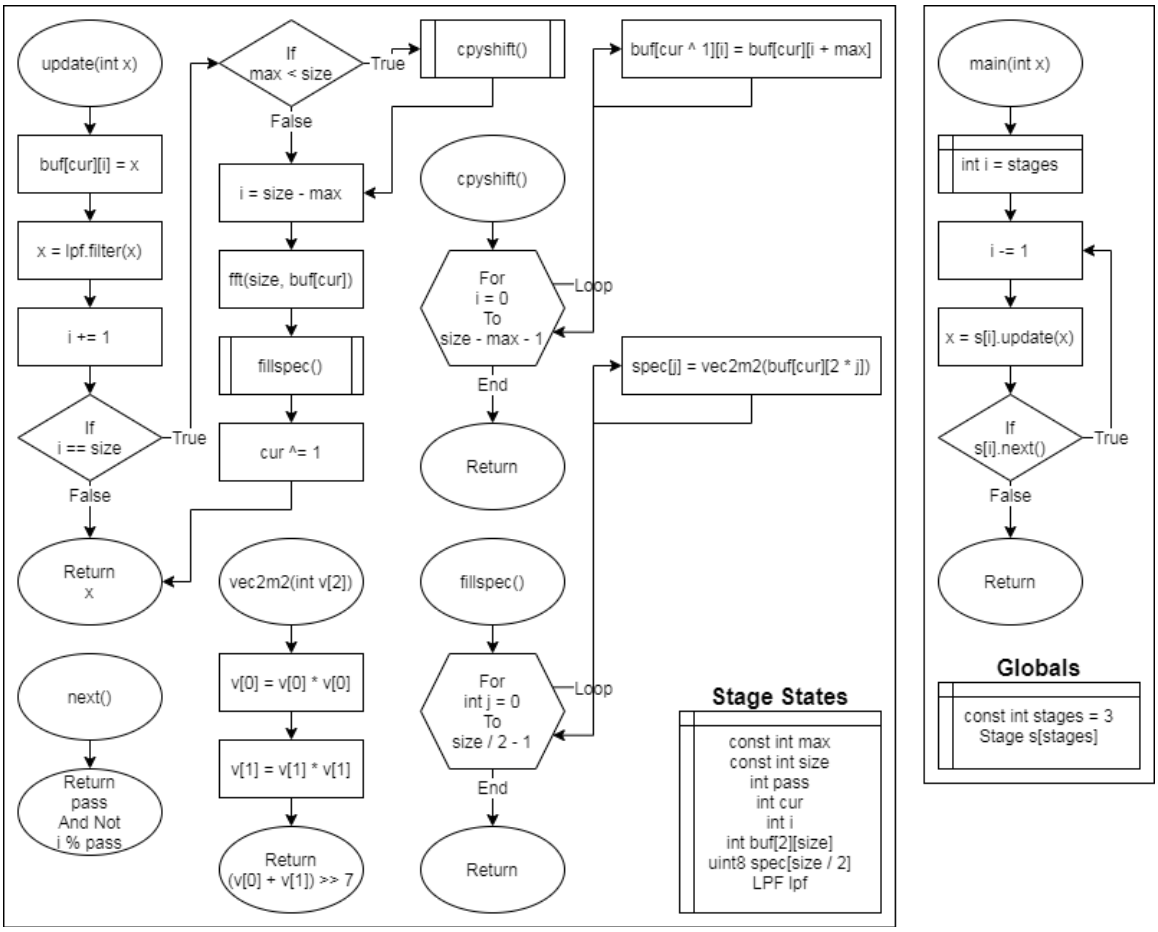


Figure 6-6: Flowchart of a previous version of the FSG.

6.3. Note, Magnitude, and Effects Determination

The note determination algorithm using the frequency spectrum given by the FFT is nearly entirely composed of filtering out unwanted frequencies. By the end of this algorithm, there will ideally only be frequency peaks remaining that represent the notes input by the user via their instrument. The inputs of this algorithm are the frequency spectrum array which is an array of magnitudes correlating to frequencies in order from our lowest frequency to our highest frequency detected by our FFT. The space between these magnitude values is all the same in Hz so there does not need to be another array for the correlating frequencies of the magnitudes. Instead, the index of the value in the magnitude array and the space between the frequency can be multiplied and added to the offset of the first

frequency, to give the frequency of that specific magnitude bin. This saves room in memory and possibly can save time required to access memory every time we need this information. Other values used in the algorithm are a slope value used to determine an approximate Q factor threshold as well as a magnitude threshold to filter out low magnitude noise and harmonics. We also will have an array referring to either the indices or memory locations of the magnitude bins that correlate to the actual frequencies of the 12-tone music system, along with a few temporary arrays for calculations. With this relatively small amount of information on top of the frequency spectrum we will be able to process and determine the notes, relying mostly on our algorithm.

The algorithm goes through several different stages before the final notes can be determined. The first stage filters out all frequencies with low magnitudes that are irrelevant and clearly not a played note. The next stage normalizes the magnitude of the transform to properly represent the unit sine wave and scale the input magnitudes to more properly represent the magnitude of the notes being played. Then the next stage goes through each detected peak and estimates the Q factor by finding the average slope of the peak. If the peak has a low Q factor, then it is not a clearly defined note and is removed. Then we have another stage that removes harmonics of fundamental frequencies. Finally, only the notes being played should be remaining and the defined semi-tones closest to the peaks' frequencies are output to the MIDI drivers.

The first stage is by far the simplest in both theory and implementation. All that needs to be done is loop through each of the indices in the magnitude array and compare the value to that of the magnitude threshold we set. If the value of the magnitude is less than that of the threshold, remove the value by setting it to zero. This threshold value is not something that we can calculate easily but rather something that will require testing to determine. Theoretically, this will remove any magnitudes and frequencies that are a result of low magnitude resonance, noise, and harmonics. This should help clean up the signal and leave only significant frequencies such as large resonance as a result of an instruments design such as an acoustic guitar, or some harmonics which will have much higher peaks than noise.

The second stage is like the first stage in that it is linear, and in fact it is technically done at the same time while looping through the indices of the magnitude array. This stage is also simple and all that needs to be done is multiply (or divide) each magnitude by a factor after the previous stage. This factor is determined based on the sample rate and number of samples. This is necessary because a normal discrete Fourier transform does not account for magnitude normalization and gives magnitudes not representing the magnitudes of the input waves. For instance, if a discrete Fourier transform had for instance 8 cycles of a wave in one sample buffer, then it would add the magnitude of that wave 8 times over, greatly exaggerating it. After this stage is complete, we now have no low magnitude noise and an accurate representation of the remaining magnitudes.

Figure 6-2 shows a discrete Fourier transform of the function:

$$X(t) = 10cos(2\pi 100t) + 20cos(2\pi 200t) + 30cos(2\pi 300t)$$

Using a sample rate of 800 Hz, 800 samples, and 4000 zeroes padded and not normalized.

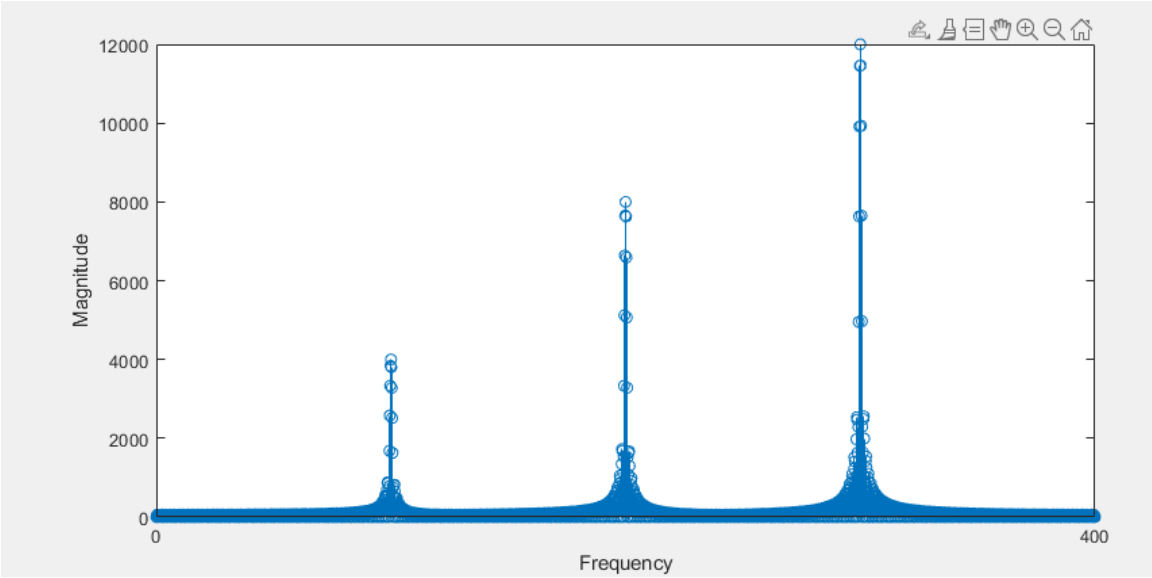


Figure 6-2: DFT example 1

As you can see, the magnitudes are far off from the input magnitudes of 10, 20 and 30. Figure 6-3. shows the normalized version of the same discrete Fourier transform and input function.

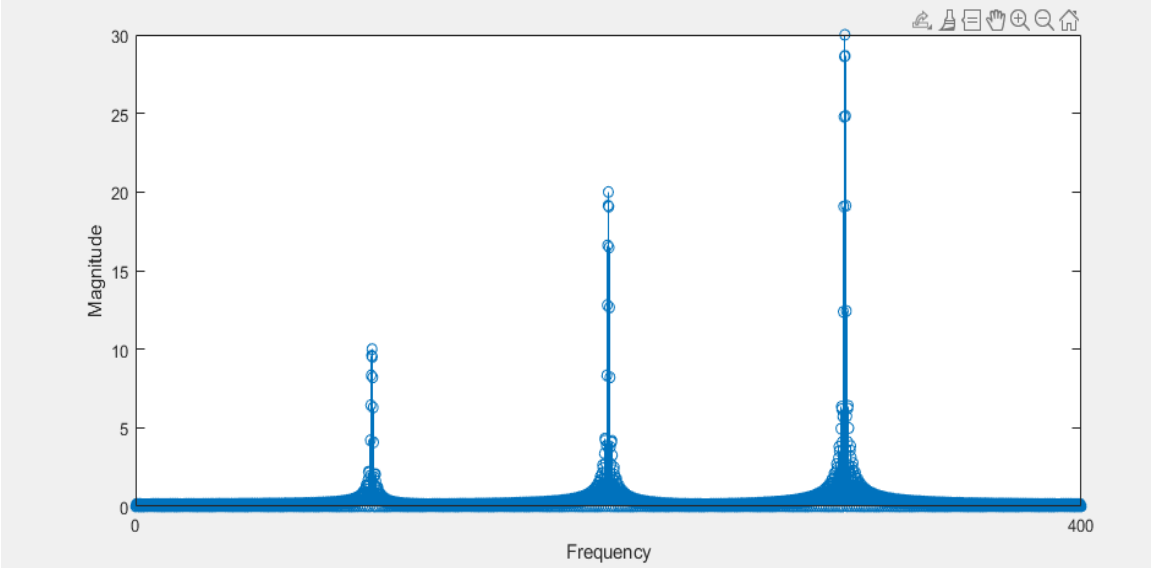


Figure 6-3: Normalized version of DFT example.

The third is the Q factor based filtering. Depending on the instrument and input method, there may be many different peaks that are large in magnitude that aren't notes played by the user on their instrument. This can come from resonance, or very loud noise. Resonance and overtones like these are usually what defines an instruments timbre and sound characteristics. These are not only not necessary to convert the signal to MIDI

protocol, but detrimental since we don't want to recognize anything but the fundamental frequency of the note played. These peaks of frequencies will typically have a much smaller Q factor than the fundamental frequency. This is something that we will take advantage of so that we can remove these leaving us with only the fundamental frequencies. To do this we loop through every semitone in our detectable range of frequencies. In this loop, we find the peak in the range of halfway between the semitone before and after the current one. Then we find the average slope based on several bins before and after the peak and compare it to our threshold. If the slope is too low to be considered a note, then the peak and closest bins have their magnitude set to zero. In Figure 6-4. it can be seen that there are several wide peaks other than the obvious peak at about 300 Hz. The 300 Hz is the fundamental frequency of a plucked guitar note and the low Q factor peaks are from resonance of the guitar. These are exactly the type of peaks that this stage of the algorithm is going to filter out.

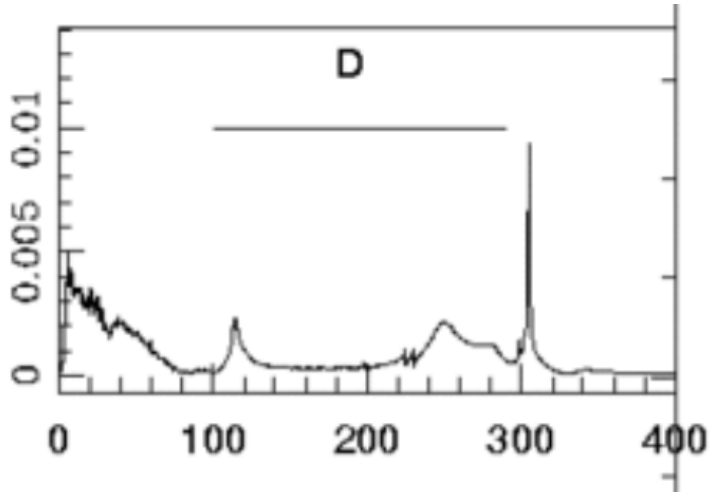


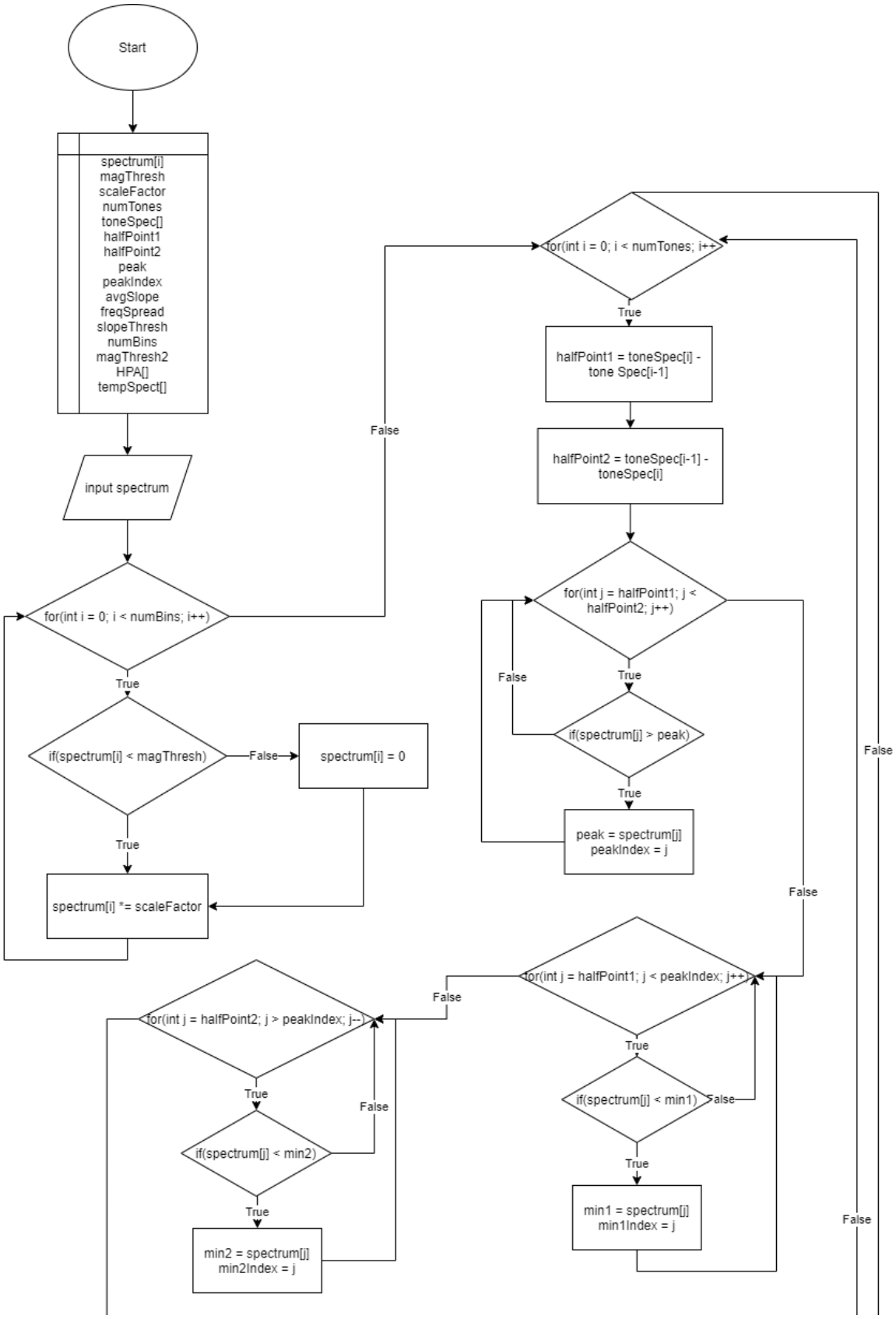
Figure 6-4: Q Factor based filtering example.

The fourth stage is the most difficult and one of the most important. This stage deals with determining the fundamental frequencies given a frequency spectrum with many harmonics. This is very important because basically every acoustic instrument will create harmonics of the fundamental frequency and often these will have a very sharp peak and often very high magnitude on a frequency spectrum. Often these harmonics can even have a higher peak than the fundamental frequency itself. This means that we must remove these harmonics otherwise they will be detected as notes by our algorithm. Harmonics are always multiples of the fundamental frequency, making them easy to find but are still difficult to remove. This is especially because different simultaneous notes might have overlapping harmonics and can cause issues in the magnitudes of some of the harmonics. There are some very complicated methods of trying to remove harmonics including analysis in different spectrums such as the cepstrum analysis which includes multiple layers of scale changes and transforms. The issues with this are that it would take way too much processing to be able to feasibly do it in real time. Instead, we will have to compromise and use a simpler, but less accurate method. This method is called the harmonic product spectrum. This transform/method utilizes the fact that harmonics are multiples of the fundamental frequency.

It works by dividing the frequency spectrum by ½, ⅓, ¼ and so on. This will cause the first, second, third, and so on harmonics to move down to another multiple of the fundamental. For instance, if you divide the first harmonic, which is the twice the fundamental, by two, you get the fundamental. Then you take these divided frequency spectrums and multiply them all. This will cause higher harmonics to be completely canceled by 0's since say 5 kHz would be moved down to 1 kHz and if there is no 25 kHz to "replace" the 5 kHz it will leave a magnitude of zero in its place. This will cancel out a lot of higher harmonics and leave the fundamental with the highest peak in the new spectrum. This works because the fundamental frequency is a common denominator of the harmonics and dividing the harmonics leaves you with the fundamental. In implementation, there will be the original frequency spectrum, a temporary copy of it, and the new spectrum to be used as the harmonic product spectrum. The temporary spectrum will be populated by iterating through the original spectrum and dividing each frequency by ½ to start with and filling the closest frequency bin to ½ of the frequency divided from. The harmonic product spectrum, which starts as a copy of the original spectrum, is then multiplied through iteration by the temporary divided spectrum. This is then repeated for dividing by ⅓, ¼, and so on until the 5th harmonic. Any point past then is unnecessary as the magnitude will most likely be negligible. After all of this, we now have a finished harmonic product spectrum in which we can do our final analysis on.

The very last stage is where the algorithm filters the harmonic product spectrum by magnitude and results in the final determined notes. The remaining peaks which are the actual played notes at this point, and then assigns them to their closest semi-tones. This is simply done by checking the semitone bin lower than the peak and seeing if it is closer than the semitone bin above the peak. This is done up to six times to choose the six maximum notes played at a time. If there are more than 6 notes detected, the 6 of highest magnitude are converted to MIDI

The implementation of these stages and the algorithm can be seen in fig. 6-4. In implementation there are many loops since many different bins of data must be dealt with. Fortunately, nearly all these computations are in linear time, so it will have a small amount of latency as compared to the FFT which will be the majority of our latency.


Start
spectrum[i]
magThresh
scaleFactor
numTones
toneSpec[]
halfPoint1
halfPoint2
peak
peakIndex
avgSlope
freqSpread
slopeThresh
numBins
magThresh2
HPA[]
tempSpect[]
input spectrum
for(int i = 0; i < numBins; i++)
True
False
if(spectrum[i] < magThresh)
False
spectrum[i] = 0
True
spectrum[i] *= scaleFactor
for(int i = 0; i < numTones; i++
True
False
halfPoint1 = toneSpec[i] - tone Spec[i-1]
halfPoint2 = toneSpec[i-1] - toneSpec[i]
for(int j = halfPoint1; j < halfPoint2; j++)
False
True
if(spectrum[j] > peak)
True
peak = spectrum[j]
peakIndex = j
False
for(int j = halfPoint1; j < peakIndex; j++)
True
if(spectrum[j] < min1)
False
True
min1 = spectrum[j]
min1Index = j
False
for(int j = halfPoint2; j > peakIndex; j--)
True
if(spectrum[j] < min2)
False
True
min2 = spectrum[j]
min2Index = j
False

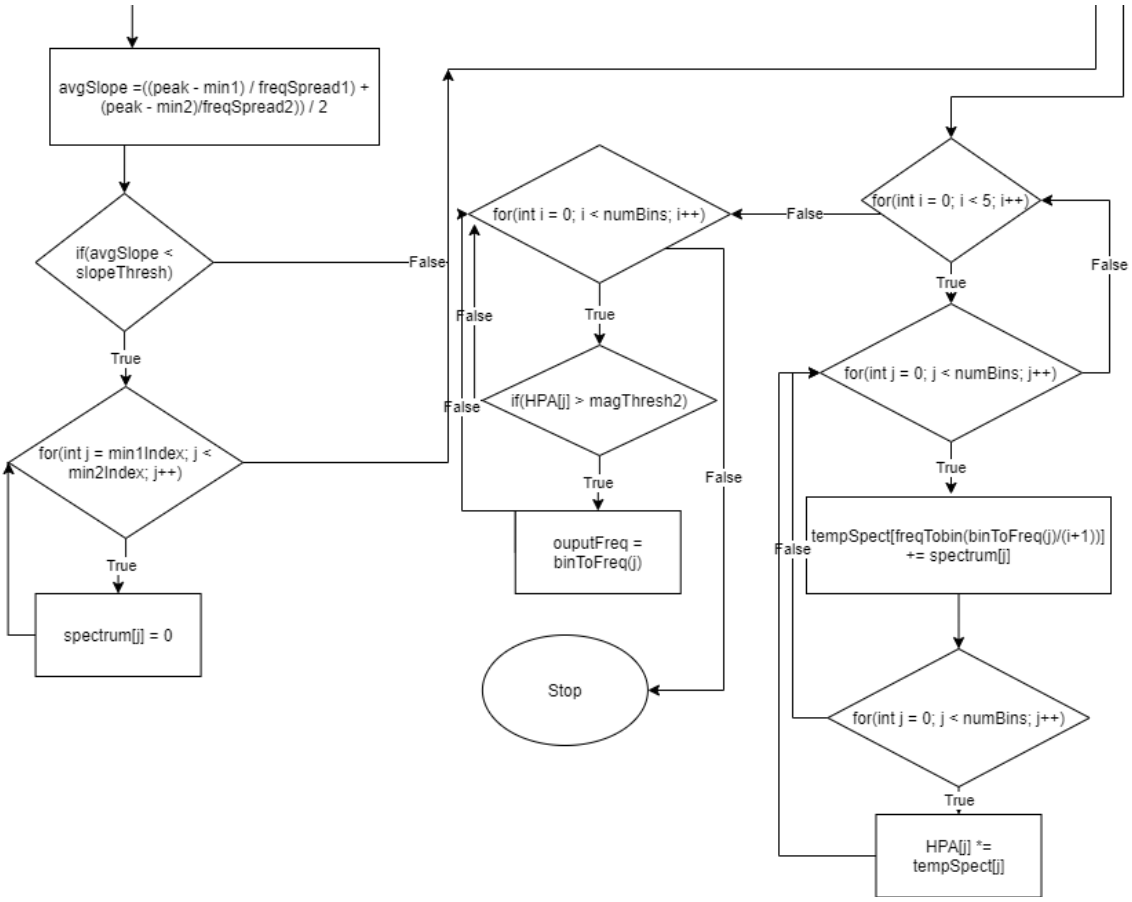


Figure 6-5: Note detection Algorithm Flowchart.

6.4. MIDI Stream Generation

The generation of a MIDI stream from the note detection algorithm outputs is simple. The generator remembers the previous state of each note and compares that to the note detection algorithm output. If there is a change from off to on, a Note On message with a velocity of 127 is generated for that note, turning it on. If there is a change from on to off, a Note On message with a velocity of 0 is generated for that note, turning it off. Figure x is a flowchart showing the process of generating data for the MIDI stream.

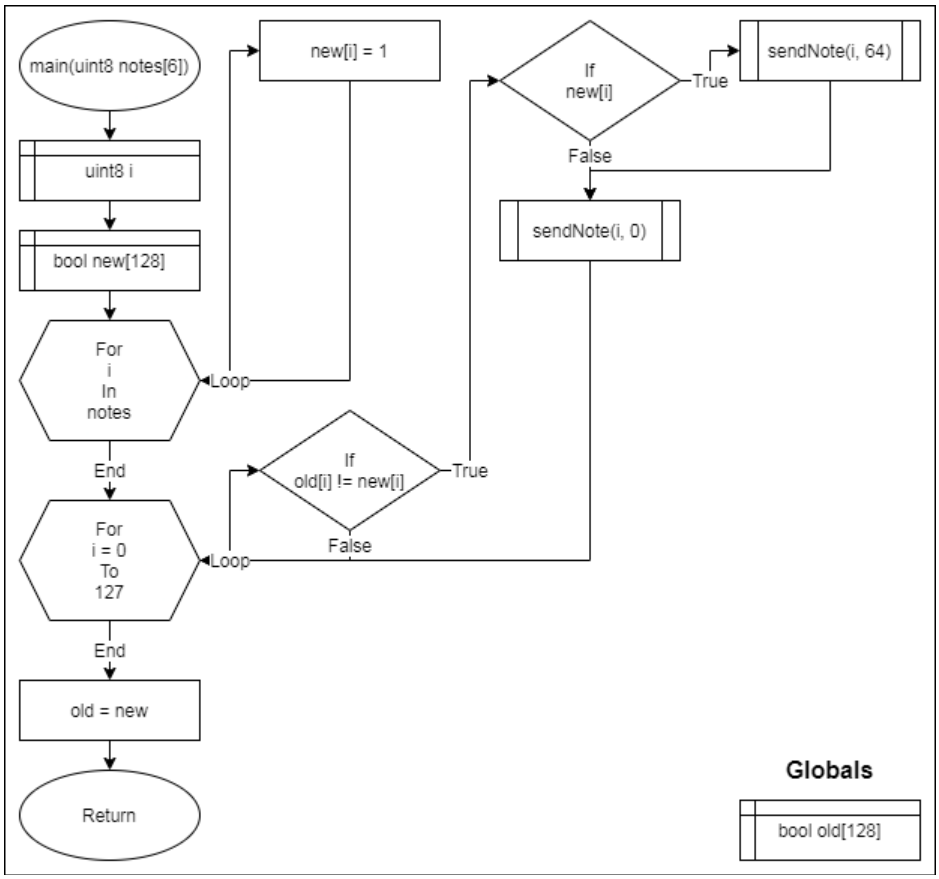


Figure 6-7: Flowchart of the MIDI stream generator.

6.5. MIDI Driver

The MIDI driver is responsible for sending MIDI messages through the physical MIDI port. An inverted UART data stream must be sent to the NPN transistor that drives the output lines. The UART data stream can be generated with the built-in peripheral of the MCU; it is configured to have a baud rate of 31.25 kHz, sourced from a 1 MHz clock. It is very important that it is sourced from this clock because it keeps the baud rate as accurate as possible; any errors from using a different clock will cause incorrect data to be received by any external device connected to the MIDI output port. The UART will attempt to send anything within its output buffer if it is not empty; filling the buffer when it is empty will cause the UART to start sending data. The buffer is 64 bytes long, which accommodates the maximum number of generated bytes possible from the MIDI stream generator, 24.

To invert the UART data stream, the UART output is fed back into a digital input pin. The interrupt handler for this pin will call a function that inverts the output of an output pin that is connected to the base of the transistor. Since the input pin will only send an interrupt on either a rising edge or a falling edge, the setting for this behavior is toggled after each interrupt. The initial states of the input and output pins are 1 and 0, respectively. Also, the input pin is initialized to send an interrupt at the falling edge.

6.6. USB Driver

Unfortunately, due to time constraints and mistakes in the design, the USB driver was never implemented. However, the intended behavior of the MCU would have been to send MIDI messages through SPI as well as UART at a rate of 1 Mbps. The USB controller would receive these MIDI messages and store them to send later in a Bulk transfer to the USB host. It would only send 0s through SPI. The other USB controller functionality remain undefined and unimplemented.

7. Prototype Testing and Building

In this section we list the procedures we did for building prototypes of each part of the device. We also list test procedures to verify the functionality of each hardware and software block.

7.1. Prototype Building

This subsection covers the construction and development of different pieces of the Analog-to-MIDI converter.

7.1.1. Power Circuits

Since the power ICs are very small surface-mount chips, we could not easily set up a breadboard prototype to test them and their circuits. We had to wait until we got the PCB to test these circuits. Although we were able to simulate the power circuits with good results, almost all of them failed when they were assembled on the PCB, mostly due to an unnoticed mistake in the 9V to 5V converter circuit. The only circuit that appeared to work was the 5V power select circuit, which was able to supply 5V from USB.

7.1.2. Preamp Prototype

The preamp prototype was built before our bill of materials for the board was finalized. We did this to ensure that the preamp was functional before we finalized the PCB design and BOM. This preamp prototype also served for us to be able to test our MCU and note detection code with the MCU development board. Since the preamp prototype was built before our BOM and parts order were made, we used a breadboard and through hole components that we already had. Since we were using a limited supply of parts to prototype the preamp, we subbed parts and values very similar equivalents. For example, the op amps used TL072 instead of the NE5532 that we selected for use on the final board. While both op amps do have subtle differences, the overall functionality as far as buffering, filtering, and amplification were nearly the same so the TL072 op amps worked well in our prototype. Similarly, we had to substitute for many of the resistor and capacitor values with series and parallel combinations that gave us close approximations. While these substitutions were not exact, they only shifted filtering by a few Hz or gain by a small percentage which was not important as to testing the overall functionality of the preamp. This is because the preamp does not aim to have exact gain values rather than a range of values for the user to be able configure in conjunction with the LED clipping indicator.

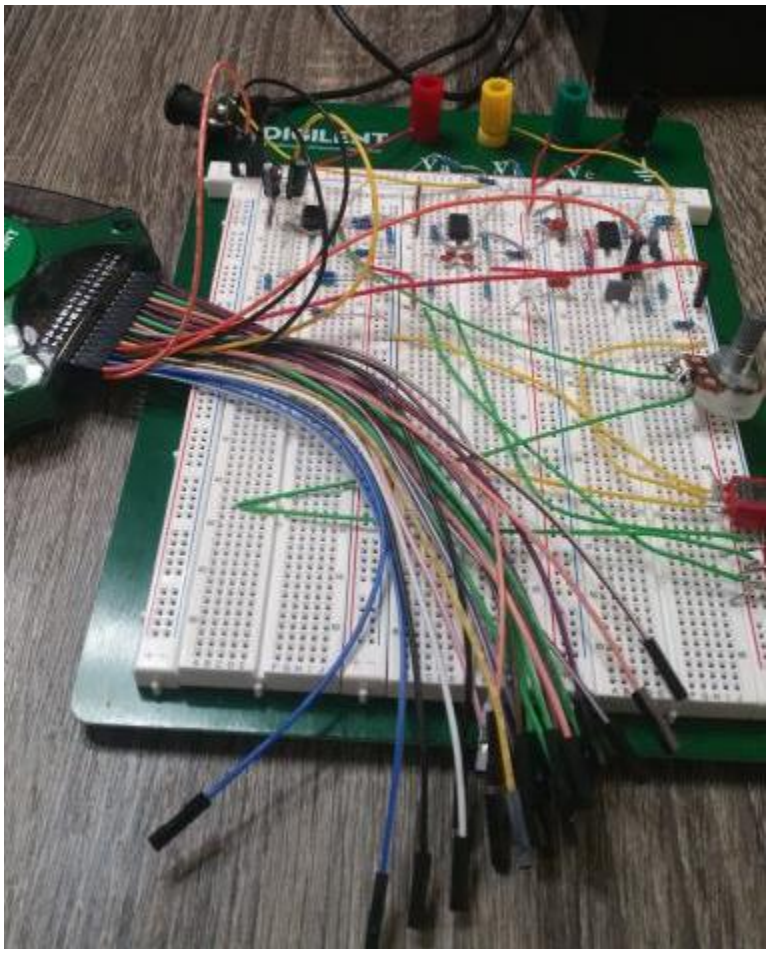


Figure 7.1: Preamp Prototype

We for testing of the prototype preamp, we used the Digilent Analog Discovery 2 kit which was sent to us, along with an external 9-volt power supply. We used the function generator on the Analog Discovery 2 to apply an input signal to the preamp and the oscilloscope on the same device to measure the output of the device. We also used the power supply on the Analog Discovery 2 for the bias voltage at the end of the preamp circuit.

When initially testing we struggled getting any signal at all, but we were able to find the source of the issue by testing the signal at different points throughout the circuit and found some incorrect connections on the bread board. After debugging the initial issues, we found that we were just getting a DC voltage that seemed to be clamped to about 7.8 volts, 1.2 volts off the power rail we were using, and the LED clipping indicator was constantly going off. After some research we realized two issues with our initial circuit design, first that the op amps we chose and the TL072 we used to prototype can't output entirely rail to rail. This means they could only go about 1.5-1.2 volts from the voltage rails powering the op amp. This means that we had to had to change the voltage range and bias a bit to compensate. This gives us about a 2-volt maximum peak to peak voltage given voltage rails of 5 volts and ground on the final board. After testing and trying different bias values we found 2.9 Volts to work well since there was a bit of a voltage drop on the bias resistors and that we found we were clipping the bottom side of the signal more often than the top.

After debugging the first issue we still found that we were getting the 7.8 volts output on the prototype board. After a bit of trouble shooting, we realized that this was because we didn't include a DC coupling capacitor on the connection between the negative op amp input and virtual ground on the voltage amplifier section. This was causing the DC bias of the signal to also get multiplied in the voltage gain section, giving us the maximum output value of the op amp, which was about Vcc+ - 1.2. After this fix, the preamp seemed to be working as intended so we tested it by making sure the whole range of input voltages worked in conjunction with the XLR/1/4" input switch. Below are figures showing the functioning LED clipping indicator as well as a sample of output.

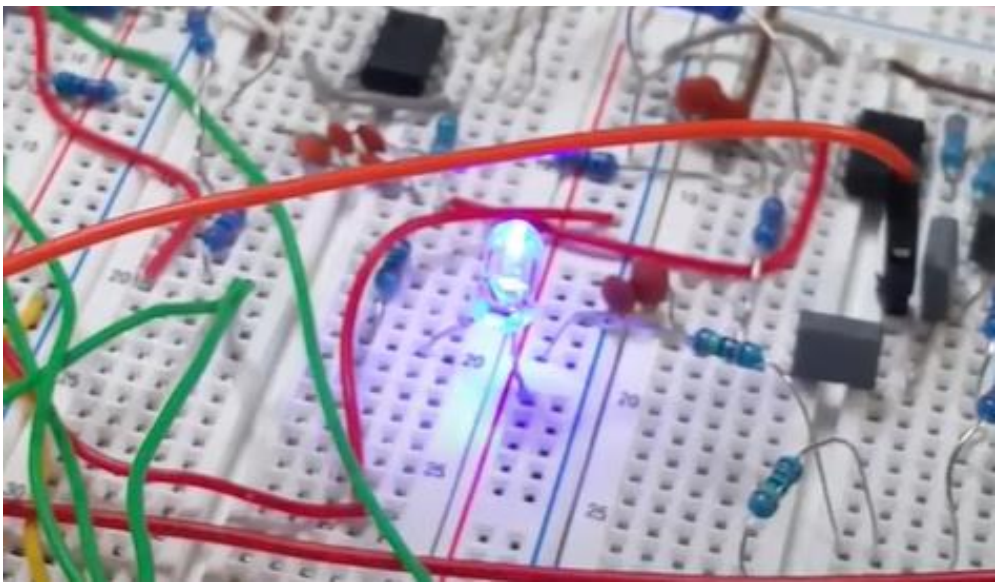


Figure 7.2: Active Clipping indicator LED on the Preamp Prototype

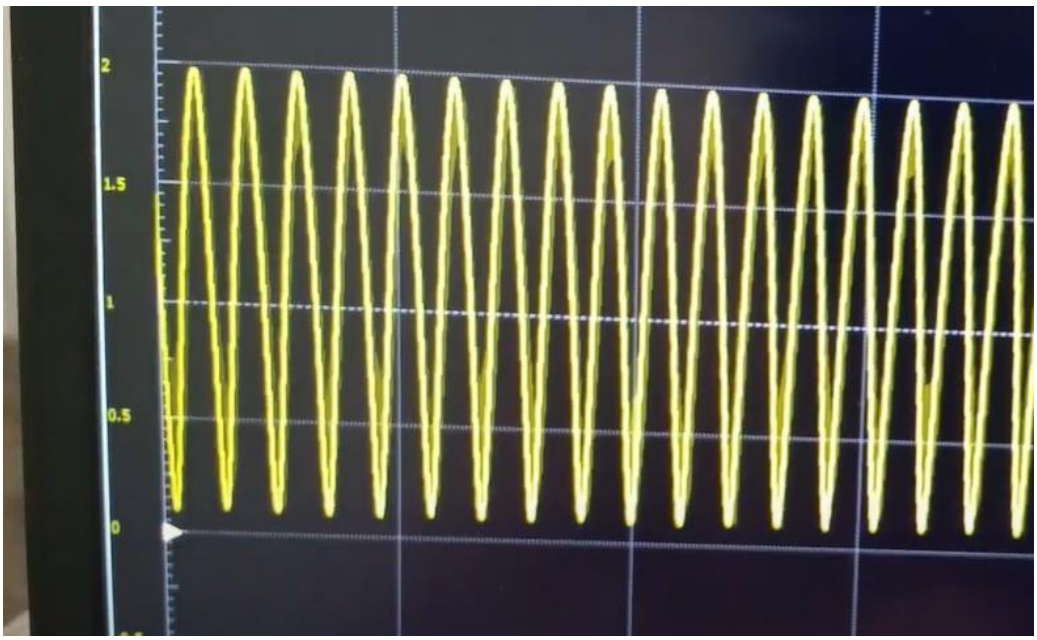


Figure 7.3 Oscilloscope reading of output on the Preamp Prototype with a 300mV peak input

This now improved version of the preamp prototype was then used in conjunction with our MCU development board to test input signals from a guitar for ADC, FFT, and note detection prototyping.

7.1.3. Microcontroller Circuit and Digital Port Controllers

The MCU circuit and USB controller circuit were designed with reference to their respective datasheets. Each IC for these circuits came in a small surface mount package, making it difficult to use a breadboard to prototype the connections. However, for the MCU circuit, we were able to use the MSP-EXP430FR5994 LaunchPad evaluation board as a stand-in for the MCU IC. The MIDI output circuit was easy to prototype using discrete through-hole parts. After assembling it according to the schematic, the circuit was confirmed to work very well.

7.1.4. MCU Peripheral Drivers, FSG, and MIDI Stream Generation

We were also able to use the evaluation board to write and test the code for the MCU. This board was also used to program the live MCU on the PCB. The code was originally written without configurability in mind for I/O ports and peripheral settings. This made the code harder to modify when prototyping and testing. Over time, the configurability of the code was greatly improved. Roughly each block in the software block diagram was given its own source file, and functionality for each block was implemented in isolation from other blocks. This means that the functionality of one block is not very dependent on that of another block, which leads to less bugs and easier programming and debugging. The only cases where functionality is dependent between several blocks are between the frequency spectrum array and the note detection algorithm, and the note detection algorithm and the MIDI stream generation.

Code for debugging the system was written to help speed up development and confirm that everything was working properly. It can be enabled or disabled and configured if necessary. The debugging code helps test the ADC sample rate, FSG latency, FSG correctness, total latency, and MIDI output correctness. Writing code that configures the peripherals was straightforward after reading about them in the user manual and using the debugging code to gauge their functions.

7.1.5. Note Detection Algorithm Prototype

The first note detection prototype first started out as a proof of concept in the C programming language. We wanted to make sure that our Harmonic Product spectrum and other parts of the note detection algorithm worked as intended. To do this we wrote code that would perform the algorithm we had designed but using a lot of hard coded input, lookup tables for frequencies, and useful operations and data types like divisions and floating point for instance. This allowed us to program it in a comfortable environment before we tried to optimize code for the MSP430FR5992. There were also some differences in the algorithm like the use of a single 2048 frequency array rather than the multiple tiered arrays we ended up deciding on.

We tested this initial prototype code using MATLAB to perform the FFT and then plugged in the data directly to the code. We read the output simply by printing out the detected note frequencies and magnitudes. These methods worked well to prove that our algorithm ideas worked well but the code would have to be entirely rewritten to work on the MSP430 and meet our latency requirements. Below are figures to display MATLAB output we were using for the initial note detection as well as results for a C major acoustic guitar chord.

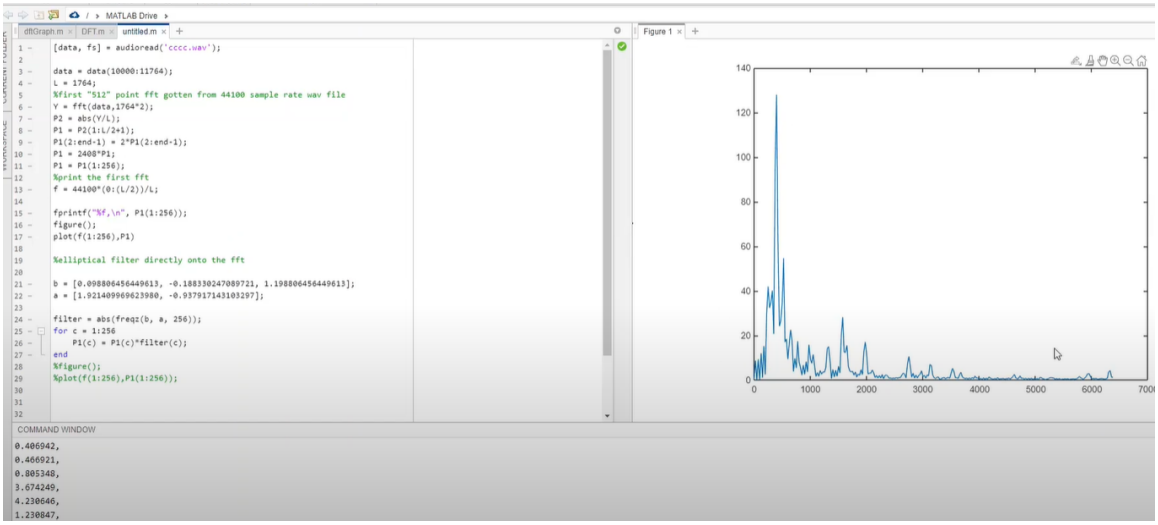


Figure 7.4: Matlab Output for use in note detection prototype

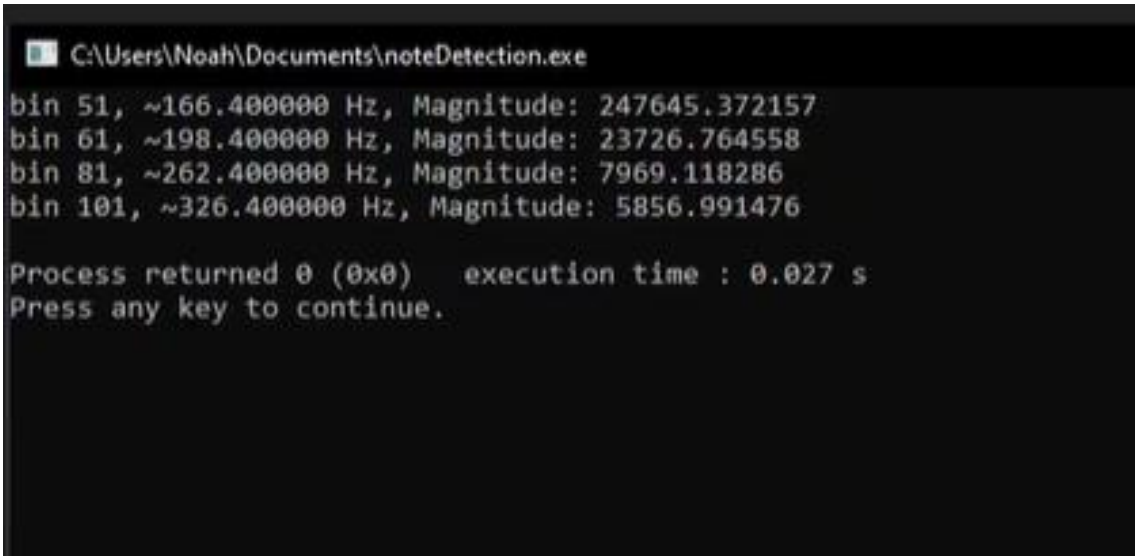


Figure 7.5 Note detection Output for C major chord

Following our initial note detection prototype, we moved on to making a highly optimized version that could run on our microcontroller. Our microcontroller uses C code, so we were able to use the same development environment to continue prototyping the code. We started with making a new function to aggregate the input from the 3-tiered FFT system we decided to use. After this, we focused on completely reworking the harmonic product spectrum code from the ground up. We didn't want the MCU to have to divide and find the

closest bins over and over so instead we wrote a script to find where our FFT bin placements divide into for the HPS and wrote a long and tedious series of loops to hard code in these bin placements for multiplication. This was very tedious to do but allowed us to avoid the time and memory usage associated with having a lookup table for bin placements and significantly reduce latency of calculating them every iteration of the algorithm. After much testing, this approach seemed to work well but we were getting an uneven frequency response across the range of the harmonic product spectrum. Part of this was because we were placing the bins in the closest division even when the placement didn't line up perfectly. To fix this we used right shifts to split proportions of a harmonic bin into a fundamental bin based on how close it was when between two bins. This gave a much more linear response across the range of harmonics and fundamentals rather than favoring certain divisions that resulted in many harmonic bins being multiplied into the same fundamental bin fully.

At this point we were using sample data gathered by our prototype preamp and MCU development board. This allowed us to get input to work with that should be nearly the same as our finalized device. We found that we still had an uneven frequency response in our harmonic product spectrum. We found that in part this was due to the several different bin spacings all aggregated into one spectrum. This meant that at some points more bins got multiplied into fundamental bins than other points but at a consistent rate due to our bin spacing. We were able to remedy this by running test input with a flat frequency response and adjusting the frequency response until it was even across the whole spectrum. At this point our response was as flat as it should be, but we were noticing a trend, the harmonic content of guitar notes was different, making the harmonic product spectrum magnitudes inconsistent. Particularly, the higher the note, the lower the harmonic content. To fix this issue, we analyzed the harmonic content of many single guitar notes through our input, and then for each harmonic on each fundamental bin compensated so that every bin should have an equal harmonic content when played on guitar. This was tedious to do manually but gave us very good results and, in the future, could be automated to work with harmonic responses of other instruments as well.

One of the last issues when prototyping the note detection code was that the range of input magnitudes varied greatly, resulting in an exponentially large variance in output magnitudes. We decided to see if there was a correlation between input RMS and this variance and found that there was. So, to solve this, we plotted the input RMS of many different samples of guitar notes and chords against the estimated threshold needed to properly detect those notes and chords. This gave us a clear exponential graph that we fit an exponential curve to. We took this exponential curve and turned it into a set of discrete threshold values to choose based on the input RMS of the signal. This effectively gave us an output detection threshold that scaled with the input RMS, making our device work with a range of input magnitudes. This was very important to the devices overall function because notes and chords can very different RMS values, not to mention notes can fluctuate a lot in volume over their lifetime. Similarly, the musician can play notes with many different magnitudes. Had we not come up with this solution our device would hardly work in a usable way.

7.2. Prototype Testing

Each test listed here should be done in order.

7.2.1. Continuity Test

This test checks for continuity between power rails and ground. This test should be performed many times throughout the assembly process.

Equipment:
- Digital multimeter with continuity check setting

Step	Description	Pass Condition
1	Set the multimeter to check continuity.	-
2	Fix the negative terminal of the multimeter to ground. Consult the schematic for probe points for ground.	-
3	Using the positive terminal, probe each power rail (9V, 5V, 3.3V, 48V). Consult the schematic for suitable probe points.	-
4	If at any point the multimeter beeps, that means that there is a short between the probed power rail and ground. The last circuit or component added to the board should be checked for validity and proper assembly.	No beep

Table 7.1: Procedure for the Continuity Test

7.2.2. Voltage Test

This test checks for the correct voltage on each power rail, 9V, 5V, 3.3V, and 48V. This test should be performed many times throughout the assembly process. Failure at any step suggests serious mistakes in the schematic or assembly, and digital components will have likely been damaged.

Equipment:
- Digital Multimeter (DMM)
- 9V power supply with barrel jack cable termination
- 5V USB power Supply
- USB cable

Step	Description	Pass Condition
1	Plug the 9V power supply into the barrel jack of the MIDI Translation Device.	-
2	Set the DMM to read voltage. Touch the leads to the GND and 9Vin test points and check the reading.	9V
3	Touch the DMM leads to the GND and +9V test points and check the reading.	6.7V – 9V

4	Touch the DMM leads to the GND and Vcc test points and check the reading.	5V
5	Touch the DMM leads to the GND and 3.3V test points and check the reading.	3.3V
6	Touch the DMM leads to the GND and 48V test points and check the reading.	48V
7	Unplug the 9V power supply from the device. Plug in the 5V USB power supply to the USB port of the device.	-
8	Touch the DMM leads to the GND and 5Vin test points and check the reading.	5V
9	Touch the DMM leads to the GND and Vcc test points and check the reading.	5V
10	Touch the DMM leads to the GND and 3.3V test points and check the reading.	3.3V
11	Touch the DMM leads to the GND and 48V test points and check the reading.	48V
12	Unplug the 5V power supply from the USB port of the device and turn off the DMM.	-

Table 7.2: Procedure for the Power Test

7.2.3. Preamp Oscilloscope Test

This test is designed to check the functionality, voltage gain levels, and bandwidth of the preamp section.

Equipment:

- Function generator
- Oscilloscope
- 9V or 5V Power source

Step	Description	Pass Condition
1	Connect power source directly if testing preamp prototype or connect power source to the MIDI-to-Analog converter.	-
2	Connect the function generator input to the input of the preamp and ground to the ground line.	-
3	Connect the Oscilloscope probe to the output of the preamp section and the ground to the ground line	-

4	Set switch to ¼” input mode and potentiometer to minimum gain.	-
5	Apply a 300mV peak to peak 1kHz sin wave with the function generator and read the oscilloscope output.	300 mV pp sin wave biased to 1.1V
6	Turn the gain potentiometer to the maximum.	LED indicator is shining
7	Turn the gain potentiometer down until the LED indicator just barely stops shining and read the oscilloscope output.	~2V pp sin wave centered at 1-1.1V
8	Repeat steps 6-7 using a 1V pp 1kHz sin wave and a 2V pp 1kHz sin wave.	Steps 6-7 pass conditions

Table 7.3: Preamp Oscilloscope Test

7.2.4. MIDI Output Test

This test checks for valid MIDI output on the MIDI port. Any failures in this test suggest that improvements are needed in the MIDI port driver code or the MIDI output circuit.

Equipment:
- Digital logic analyzer
- Electric guitar
- ¼” Cable
- MIDI cable
- 9V power supply with barrel jack cable termination or 5V USB power supply

Step	Description	Pass Condition
1	Connect the power source and electric guitar to the device.	-
2	Connect the MIDI cable to the output port of the device and the input port of an external device.	-
3	Use the digital logic analyzer to probe the collector of the transistor.	-
4	Play a single note on the electric guitar and record the digital logic at the collector.	-
5	Check if the bit sequence matches the intended bit sequence of the MIDI message.	Correct MIDI message sent

6	Check if the period of each bit is consistently 32 microseconds.	Bit period of 32 µs

Table 7.4: Procedure for the MIDI Output Test

7.2.5. Guitar Note Detection Test

Equipment:
- Electric guitar
- ¼" Cable
- MIDI-to-USB cable

Step	Description	Pass Condition
1	Set up the MIDI Translation Device so that you can see its output.	-
2	Attach the electric guitar to the ¼" input jack of the MIDI Translation Device, making sure to send the signal through the tip of the cable and attach the sleeve to ground.	-
3	Play an E2 on the guitar. Observe the output from the MIDI Translation Device.	Output of E2
4	Play a G3 on the guitar. Observe the output from the MIDI Translation Device.	Output of G3
5	Play an A4 on the guitar. Observe the output from the MIDI Translation Device.	Output of A4
6	Play a C5 on the guitar. Observe the output from the MIDI Translation Device.	Output of C5
7	Play an E6 on the guitar. Observe the output from the MIDI Translation Device.	Output of E6
8	Disconnect the function generator from the ¼" input jack of the MIDI Translation Device.	-

Table 7.5 Procedure for the Guitar Note Detection Test

8. Administrative Content

During Senior Design 1, most of the milestones have to do with documentation deadlines rather than actual testing, implementation, etc. In Senior Design 2 we will be implementing and testing our design. See Tables 8.1 and 8.2 below for details.

Senior Design 1

No.	Task	Deadline	Status
1	Pick Project Idea, Assign roles	5/22/2020	Completed
2	Initial Project Documentation- Divide and Conquer	5/29/2020	Completed
3	Updated Divide and Conquer document	6/5/2020	Completed
4	60-page draft	7/3/2020	Completed
5	100-page draft	7/17/2020	Completed
6	120-page Final Document	7/28/2020	In Progress
7	Breadboard testing	7/28/2020	Not Started
8	Begin ordering parts	7/28/2020	Not Started

Table 8.1: Senior Design 1 Milestones

Senior Design 2

No.	Task	Deadline	Status
1	Implemented Note Detection & Test Software	8/18/2020	Not Started
2	Finish first draft of drivers	8/18/2020	Not Started
3	Testing Parts	8/25/2020	Not Started
4	Possible Redesign	9/15/2020	Not Started
5	Finalized Design	10/6/2020	Not Started
6	Final Prototype working	11/17/2020	Not Started
7	SD Showcase	TBA	Not Started

Table 8.2: Senior Design 2 Milestones

9. User Instructions

The instructions in this section refer to the version of the device used in the final presentation and showcase demonstration. Any features that are not present in this version of the device, such as USB, XLR, or on-board power, will be omitted from these instructions.

Getting Started

In order to use the Polyphonic Analog to MIDI Converter (PAMC), you will need: A 5V power supply, a 3.3V power supply, a MIDI to USB cable, one or two ¼" instrument cables, and an electric guitar. Please attach the 5V power supply to pin 1 of header SV2, attach 3.3V to pin 2 of header SV6, and attach ground to pin 9 of header SV6. Once these connections are made, the device should start up. Connect the MIDI end of the MIDI to USB cable to the MIDI out port on the device and plug the USB end into your computer. Your digital audio workstation (DAW) should now recognize that a MIDI device is available to use. Finally, plug your guitar into the input of the device using a ¼" instrument cable. Optional: Use a second ¼" cable to connect the passthrough port of the device to an amplifier and hear your guitar signal at the same time as the MIDI output of the device.

Using the Device

Simply open a MIDI-controllable instrument in your DAW and begin playing notes on your guitar. The PAMC should convert your signal into MIDI for the DAW to read. If nothing is happening, try unplugging the power from the PAMC and plugging it back in to restart the device. If you are getting more than one MIDI note for each single note you play, try using the potentiometer to adjust the gain until you are getting one MIDI note for each one note played.

Appendix A: Glossary of Music Terminology

Digital Audio Workstation (DAW)
: A computer program or digital device used to record and edit music.

12-Tone Equal Temperament
: A musical pitch system that divides octaves into twelve pitches that are equally spaced on a logarithmic scale

Pitch
: Frequency of a note in Hertz, usually notated with a letter and octave number (ex. A2 or C3)

Octave
: Range of pitches from one note to the next note of the same name or from a note of pitch n Hertz to 2n Hertz (ex. From A2 to A3 or 110 Hz to 220 Hz)

Portamento
: Smooth glide transition from one note to another

Sustain
: Holds musical note until sustain is released

Sostenuto
: Sustain that only affects notes played at the time that sustain is activated

Legato
: Smooth, even, connected note style

Attack
: Beginning of a note, time it takes for a note to reach maximum amplitude

Decay
: Ending of a note, time it takes for a note to soften and end after it is released

Reverb
: Reverberation echo effect

Tremolo
: Trembling effect, adjusts amplitude of signal up and down

Chorus
: Effect where an additional note is played at approximately the same pitch and with approximately the same timbre

Phaser
: Effect where a filter is applied to produce peaks at different frequencies of the signal

Detune
: Effect that puts a note out of pitch

Omni-Mode
: Mode where MIDI device is listening for incoming signals on any MIDI channel

Appendix B: Bibliography

[1] Jack Deville, “Buffers, impedance and other internet lore”, 26 Sep 2012. URL: https://www.mrblackpedals.com/blogs/straight-jive/6629774-buffers-impedance-and-other-internet-lore

[2] IEEE GlobalSpec, “Active Band Pass Filters Information”, 2020. URL: https://www.globalspec.com/learnmore/semiconductors/analog_mixed_signals/amplifier_linear_devices/active_bandpass_filters

[3] Larry Davis, “Distribution Amplifier”, 7 Mar 2012. URL: http://www.interfacebus.com/distribution-amplifier.html

[4] Wikipedia, “Operational amplifier”, 23 Jul 2020. URL: https://en.wikipedia.org/wiki/Operational_amplifier

[5] ElProCus, “Know All about Analog to Digital ADC Converters.”, 8 Oct. 2014, URL: https://www.elprocus.com/analog-to-digital-adc-converter/

[6] Matthew R Finch, “Constructing, Manipulating, Classifying and Generating Audio with Digital Signal Processing and Machine Learning”, 17 Apr 2020. URL: https://towardsdatascience.com/constructing-manipulating-classifying-and-generating-audio-with-digital-signal-processing-and-2c5a252dbab9

[7] Unknown, “Digital Sound”, 11 May 2016. URL: http://monicandreatic.blogspot.com/2016/05/digital-sound.html

[8] Stanford, “PITCH DETECTION METHODS REVIEW.” Date unknown, URL: https://ccrma.stanford.edu/~pdelac/154/m154paper.htm

[9] Smith, J.O. *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications, Second Edition,* online book, 2007 edition, URL: https://ccrma.stanford.edu/~jos/mdft/ ,

[10] Wikipedia, “Fast Fourier transform”, 19 Jul 2020. URL: https://en.wikipedia.org/wiki/Fast_Fourier_transform

[11] Wikipedia, “Cooley-Tukey FFT algorithm”, 9 Jul 2020. URL: https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm

[12] Sergey Chernenko, “Fast Fourier Transform - FFT”, 2020. URL: http://www.librow.com/articles/article-10

[13] Embedded.com, “Develop FFT apps on low-power MCUs”, 19 Oct 2005. URL: https://www.embedded.com/develop-fft-apps-on-low-power-mcus/

[14] Dataforth, “Protecting Signal Lines Against Electromagnetic Interference”, 2020. URL: https://www.dataforth.com/protecting-signal-lines-against-electromagnetic-interference.aspx

[15] Polycase, “Does Your Enclosure Need EMI-RFI Shielding?”, 18 Sep 2018. URL: https://www.polycase.com/techtalk/electronics-tips/does-your-enclosure-need-emi-rfi-shielding.html

[16] Industrial-Electronics.com, “Grounding to Reduce EMI”, 19 Aug 2008. URL: https://www.industrial-electronics.com/measurement-testing-com/EMI-Grounding.html

[17] *Basic Electronics Tutorials* “Switch Mode Power Supply Basics and Switching Regulators.” , 2 Mar. 2018, URL: https://www.electronics-tutorials.ws/power/switch-mode-power-supply.html

[18] Wikipedia, “Cent (music)”, 1 May 2020. URL: https://en.wikipedia.org/wiki/Cent_(music)

[19] ARM, *Cortex-M0+ Technical Reference Manual*, 16 Dec 2012. URL: https://static.docs.arm.com/ddi0484/c/DDI0484C_cortex_m0p_r0p1_trm.pdf

[20] Peacock, Craig. “USB Protocols.” *USB in a NutShell - Chapter 3 - USB Protocols*, Apr. 2018, URL: https://beyondlogic.org/usbnutshell/usb3.shtml.

[21] Evan Wakefield, *Benchmarking the Signal Processing Capabilities of the Low-Energy Accelerator on MSP430 MCUs*, Nov 2016. URL: https://www.ti.com/lit/an/slaa698b/slaa698b.pdf?ts=1595120336339

[22] Jim Karki, *Understanding Operational Amplifier Specifications, Feb 2020. URL:* https://www.ti.com/lit/an/sloa011a/sloa011a.pdf

[23] Robert Keim, “Understanding Operational Amplifier Slew Rate”, 28 Feb 2020. URL: https://www.allaboutcircuits.com/technical-articles/understanding-operational-amplifier-slew-rate

[24] Dr. Sebastian Anthony Birch, *Technical Introduction to MIDI*, URL: http://personal.kent.edu/~sbirch/Music_Production/MP-II/MIDI/an_introduction_to_midi_contents.htm

[25] NickFever. “MIDI CC List.” *NickFever*, 2019, nickfever.com/music/midi-cc-list. URL: https://nickfever.com/music/midi-cc-list

[26] “USB 2.0 Specification.” *USB*, 21 Dec. 2018, URL: www.usb.org/document-library/usb-20-specification.

[27] “Defined Class Codes.” *USB*, URL: https://www.usb.org/defined-class-codes

[28] "Universal Serial Bus Device Class Definition for Audio Devices" *USB*, 18 Mar, 1998, URL: https://www.usb.org/sites/default/files/audio10.pdf

[29] "Universal Serial Bus Device Class Definition for MIDI Devices" *USB*, 19 Nov 1999, URL: https://www.usb.org/sites/default/files/midi10.pdf

[30] "Title 47 CFR Part 15." *Wikipedia*, Wikimedia Foundation, 25 June 2020 URL: https://en.wikipedia.org/wiki/Title_47_CFR_Part_15

[31] Texas Instruments, "Efficient_Computation_of_Real_Input". URL: https://processors.wiki.ti.com/images/tmp/f1258160414-1106104018.html