Effects Joystick for LoFi Guitar

George Willingham
Boston University

Using an Arduino Uno for quality audio processing presents several immediate challenges: the analog inputs have only 10-bit ADC, there is only 2kB of SRAM for processing, and there is no onboard DAC for the output. Despite these limitations, people have succeeded in making decent sounding guitar pedals, amps, etc which do all their signal processing on an Arduino Uno. For my project, I plan to make an Arduino-based guitar effects module which will be controllable with a joystick.

PROJECT OVERVIEW

Before going into the details of the design, here is an overview of the project. The block diagram in Figure 1 depicts three stages of signal flow (including the joystick in the second stage). Each of the three stages can be broken down as follows.

Input Stage: The preamp block is the input stage which serves the purpose of preparing the signal induced on the guitar's pickups for the Arduino. The signal must be biased to center it between the rails of the Arduino and to take full advantage of the dynamic range available, a preamp step is needed. Filtering out high frequencies in this stage will also help prevent aliasing effects in the subsequent A/D conversion.

Processing Stage: After the input stage, the signal will pass through the Arduino's ADC and will be processed. I will write a C program for the Arduino which will modulate that signal based on an additional input from the joystick. Once modulated, the result will be output from the Arduino to an external DAC.

Output Stage: The output from the Arduino will be sent via SPI protocol to an external DAC. At the output of the DAC, I will use a low-pass filter to smoothen the signal and a blocking capacitor to remove a dc offset before finally sending it to a separate power amplifier and loudspeakers.

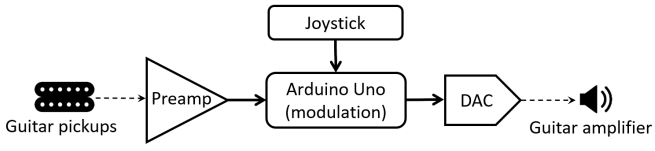


FIG. 1. Project Block Diagram

I. INPUT STAGE

To prepare the guitar's signal for the Arduino, the input stage circuit must perform three tasks: filtering, biasing, and preamplification. The filtering is for cutting high frequencies to prevent aliasing in the subsequent A/D conversion. The biasing is for centering the signal between the +5V and 0V rails of the Arduino. The preamplification is for boosting the signal to swing across the entire 5V dynamic range. The circuit designed for these purposes is shown in Figure 2. It can be broken into 3 parts:

1. Biasing Network

The biasing is taken care of with the voltage divider formed by R1 and R2. These resistors are set equal to center the signal at +2.5V. The capacitor C1 blocks any dc signal and more generally forms a high-pass filter with R2.

2. Tunable-Gain Amplifier

The biased signal is then fed to the non-inverting input of an op-amp for the preamplification step. A non-inverting configuration is chosen for its higher input impedance. To ensure that the dc bias is preserved across the op-amp, a capacitor (C2) is placed between the feedback path and ground, thus ensuring a dc gain of unity. To provide some manual control of the preamp gain, a rheostat (R5) is also included in the gain circuit.

3. Low-pass Filter

Finally, the biased and amplified signal is passed through a simple RC low-pass filter formed by R6 and C3. This cuts high frequencies and helps reduce aliasing effects in the subsequent A/D conversion.

FIG. 2. Input Stage

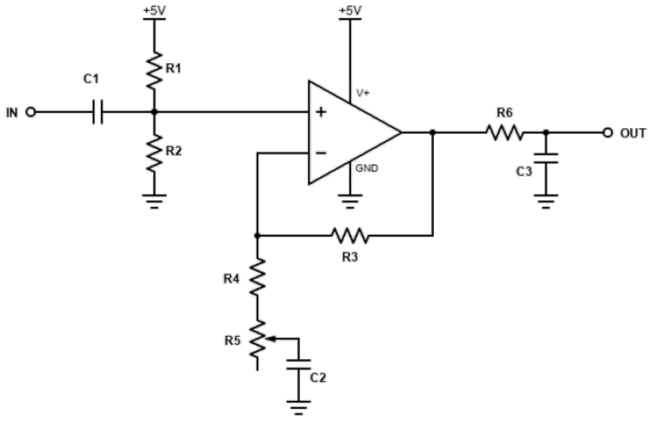


TABLE I. Input Stage Component Values

Resistors	(Ω)	Capacitors	(F)	Op-Amp
R1	100k	C1	1μ	LM358
R2	100k	C2	1μ	
R3	100k	C3	6.8n	
R4	10k			
R5	100k			
R6	1k			

Since this is for an audio application, the frequency range of interest is 20Hz - 20kHz. With this in mind, the resistor/capacitor values should be chosen with the following design considerations:

- To act as an anti-aliasing filter, frequencies higher than 20kHz should be filtered out
- With my typical guitar signal at 1Vpp, frequencies in the audible range should have a flat gain of around 5. The variability of that gain as controlled by the pot should be limited to similarly reasonable values (say 1 to 10)

The values chosen are shown in Table I. A listening test confirmed that the output sounds loyal to original guitar signal.

II. PROCESSING STAGE

After the input stage, the signal will be input to the Arduino through one of its analog input pins. The signal is then sampled and quantized by a 10-bit ADC onboard the Arduino before going on to be modulated. After the modulation, the signal will be output to a DAC via SPI protocol.

A. Audio with Timer Interrupts

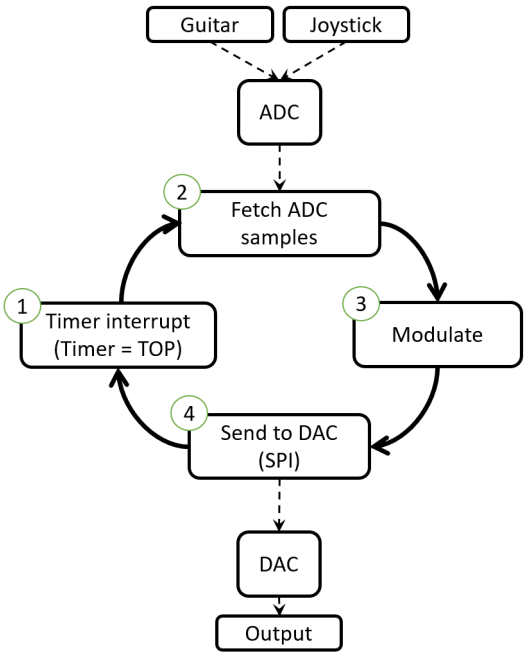
In digital audio, it is standard to sample at 44.1kHz and with 16-bit resolution. This is a tall order for the Arduino, but it is possible to get reasonably close. The microcontroller onboard the Arduino is an ATmega328P which has a system clock with $f_{\text{clk}_{sys}} = 16\text{MHz}$. On that chip there are also three timers; one of which is 16-bit (Timer1). The idea is to use this timer to trigger an interrupt every time it reaches its TOP value. Then in the interrupt routine, a sample is taken from the ADC, modulated, and sent to the DAC. The full process is illustrated in Figure 3.

To get a decent sound, the timer and the ADC have to be sufficiently fast. The Timer1 frequency f_{timer} is determined by a programmable prescalar which divides the system clock. Since the interrupt frequency (i.e. sampling/output frequency) is related to the timer's frequency and TOP value through $f_{\text{sample}} = \frac{f_{\text{timer}}}{\text{TOP}+1}$, we can write

$$f_{\text{sample}} = \frac{f_{\text{clk}_{sys}}}{\text{prescalar} \times (\text{TOP} + 1)}$$

By default the prescalar is set to 64 and the TOP value is $2^8 - 1$ which gives an f_{sample} of about 1kHz. Changing the prescalar and TOP value, this can be bumped up to $\boldsymbol{f_{\text{sample}} = 31.25\text{kHz}}$. It could be higher, but there will need to be enough time to modulate each sample before getting a new one. The modulation code must be kept short and sweet!

FIG. 3. Interrupt-Based Audio Processing



With each timer interrupt requesting a sample, the ADC needs to be able to meet that demand. The ADC on the ATmega328P has its own clock which has a frequency determined by a prescalar just like the timer. From the datasheet, one ADC conversion takes 13 clock cycles. As a consequence, the ADC cannot take samples any faster than

$$\text{max ADC sampling freq.} = \frac{f_{\text{clk}_{sys}}}{13 \times \text{prescalar}_{\text{ADC}}}$$

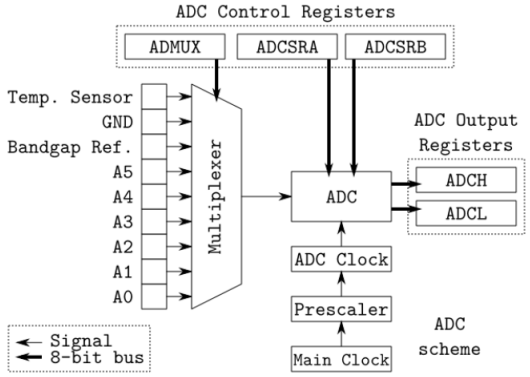
By default, the prescalar is set to 128 which leaves a maximum sampling frequency of around 9.6kHz — too small for good audio! But that value can be changed. With prescalar=32, the maximum sampling frequency is 38.5kHz which would work, but since the joystick also needs to be sampled, it will be preferable to push the limit and set prescalar=16 which gives a maximum sampling frequency of 77kHz. This is elaborated on in the next section.

B. The Joystick: Multiplexed Analog Inputs

A challenge in this project is figuring out how to incorporate the joystick input into the timer interrupt algorithm just described. The output of the joystick must go into an analog input pin of the Arduino just like the guitar signal. But as can be seen in Figure 4, all of the analog inputs are sent to a single multiplexer which feeds the ADC. So the multiplexer must constantly switch between these inputs to read them and as a consequence, time that would ideally be spent handling audio will have to be sacrificed to sample the joystick.

With the consistent timer interrupts, it seems like the best solution is to sample both the audio and the joystick every interrupt. This way the audio sampling frequency remains 31.25kHz.

FIG. 4. ATmega328P ADC block diagram



III. OUTPUT STAGE

With the digital signal being output from the Arduino, it can now be sent to the external DAC (MCP4921) for conversion. As described, this is done through SPI protocol. All that remains is to put one last low-pass filter to smooth out the signal and put a blocking capacitor to remove the dc bias. The output stage circuit is shown in Figure 5 and its component values are shown in Table II.

FIG. 5. Output Stage

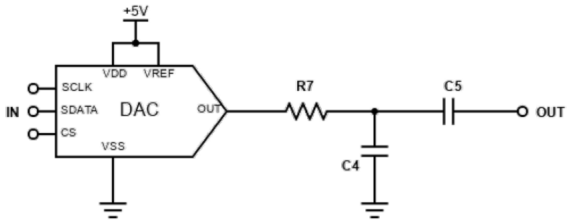


TABLE II. Output Stage Component Values

Resistors (Ω)		Capacitors (F)		DAC
R7	1k	C4	6.8n	MCP4921
		C5	4.7μ	

TABLE III. Complete Parts List

Resistors	(Ω)	Capacitors	(F)	Op Amp	DAC	Connectors	Microcontroller	Other
R1	100k	C1	1μ	LM358	MCP4921	2× 1/4" audio jacks (female)	Arduino Uno	Joystick Module
R2	100k	C2	1μ					
R3	100k	C3	6.8n					
R4	10k	C4	6.8n					
R5 (pot)	100k	C5	4.7μ					
R6	1k							
R7	1k							

TABLE IV. Implementation Plan

Week	Plan
4/4 - 4/10	Build input stage prototype and test/troubleshoot. Also focus on how viable the overall software structure is. Get working output to DAC at desired frequency using just a test signal. Switch to dual PWM output if need be.
4/11 - 4/17	Write code for ADC sampling. Connect input stage and joystick to Arduino and begin testing guitar input.
4/18 - 4/24	Write code for modulation. General testing/troubleshooting
4/25 - 5/1	Finishing touches. Solder everything together if possible.

- Creative Technologies — Arduino Guitar Amp
- Electrosmash — pedalSHIELD UNO Arduino Guitar Pedal
- ATmega328P datasheet
- Open Music Labs — ATmega ADC
- Instructables — Arduino Audio Input
- Instructables — Arduino Audio Output
- QEEWiki — Analog Inputs (Analog to Digital Converter)
- QEEWiki — Timers on the ATmega 168/328
- AVR — ATmega Interrupts
- Oscar Liang — Arduino Timer and Interrupt Tutorial
- Glenn Sweeney Tutorials — Interrupt Driven Analog Conversion with ATmega328P