MKR_Zero Audio Project

by John Talbert, April 2021

A versatile platform for exploring Audio Processing
With the MKR_Zero processor

Table of Contents

Arduino MKR_Zero

The MKR_Zero is a SAM D21-based 32-bit microcomputer board. It has several audio/ music features -- an on-board SD card reader with dedicated SPI interfaces (SPI1) that allows you to play MUSIC files with no extra hardware, and an I2S (Inter-IC Sound) serial bus interface, a standard interface for connecting digital audio devices. Two libraries have been built to take advantage of these two interfaces.

Arduino Sound library – a simple way to play and analyze audio data using Arduino on SAM D21-based boards.

I2S library – to use the I2S protocol on SAMD21-based boards.

The MKR_Zero has seven ADC Analog input pins which can be configured as 8, 10, or 12-bit. The sampling rate for these ADCs is set up for slow moving signals, but this can be changed to accomodate audio signals by hacking into the board's counter register settings for the ADCs (illustrated later in some of the sample programs).

Most previous Arduino boards had no DAC output pins. Developers resorted to filtering a PWM (Pulse Width Modulation) output to simulate a DAC. The MKR_Zero board, however, has one 10-bit Digital to Analog Converter ouput pin.

The ADC and DAC functions on the MKR_Zero are not ideal for audio signal processing applications but work fine as educational tools to explore simple audio digital processing application development. Perhaps future boards will have the higher sampling rates and 16-bit widths.

The supply voltage for the board is 5 volts, but the operating voltage is 3.3v
All pins must only use a voltage range of zero to 3.3volts. 5 volts applied to any of the pins will likely destroy the processor.

An I2C serial interface is available with its own 5 pin connector. An extra serial interface is provided in addition to the USB programming port. Its Tx and Rx pins are put to use

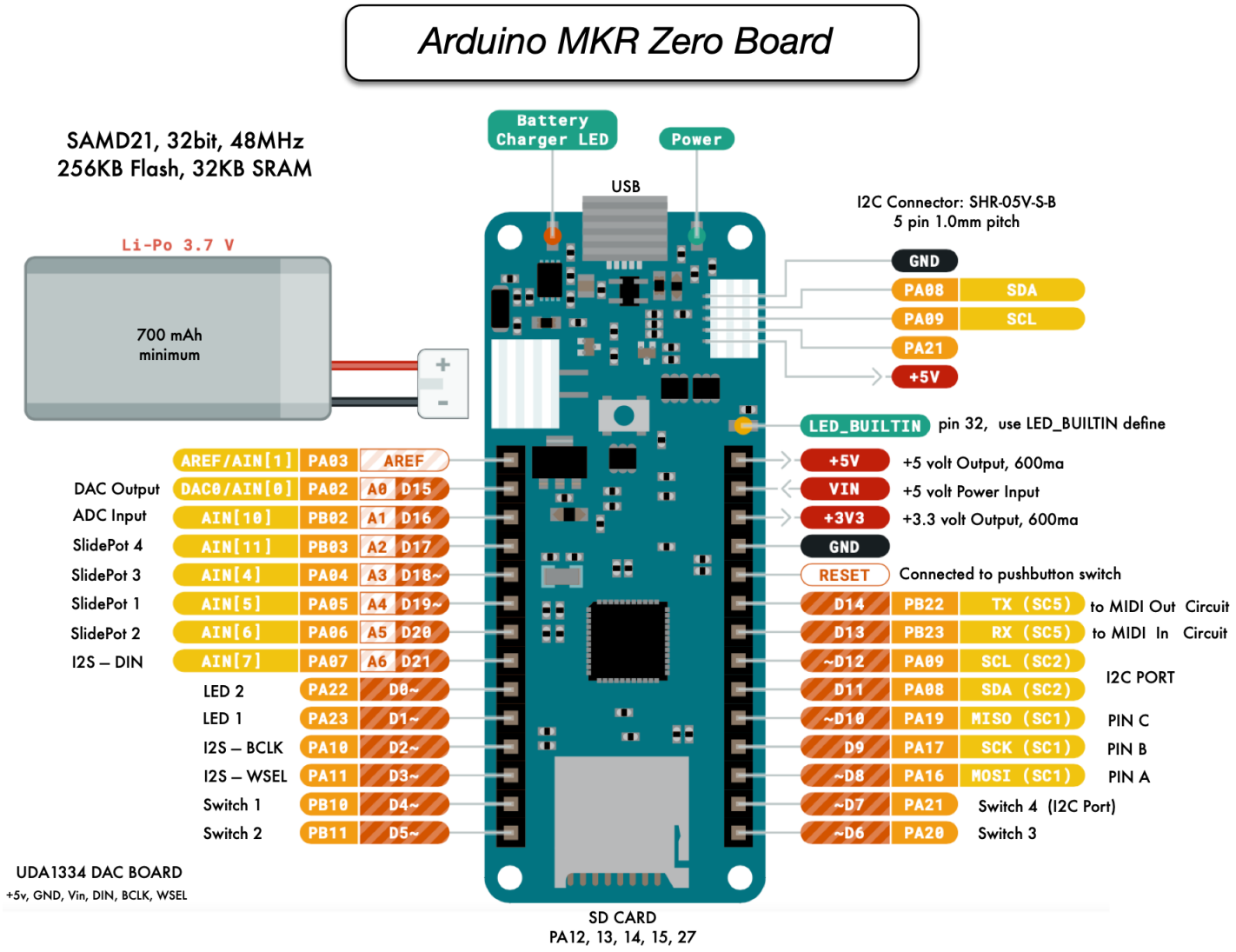
as standard MIDI input and output in this project.

The processor runs at a speedy 48MHz which is triple the speed of the original Uno board.

There is 256k of Flash memory for user programs, and 32k of SRAM memory for data storage (program variables). This is quite an improvement over the Uno's 30K of Flash memory and 2K of SRAM. The larger SRAM will come in handy when storing audio data for digital delays.

There is an optional connector for an external battery (3.7volt Lithium at 700 mAh minimum). In addition to the one ADC pin used for the audio input, 4 additional ADC pins are available for connection to slide pot controllers. Finally, miscellaneous switches, LEDs, and output signal generation pins can be connected to any of the 20 available digital I/O pins of the MKR_Zero.

The illustration below labels all the circuit connections made with the MKR_Zero pins for this project.



Audio Circuit

The overall purpose of this project is to create an versatile platform for exploring audio signal processing using the MKR_Zero processor. The planned signal chain is as follows: Audio from the outside world will be input to an MKR_Zero ADC pin. The ADC (Analog to Digital Converter) translates the audio signal into a stream of digital numbers which can be processed in some way in an Arduino sketch. The output data from the processing can then be sent to either the MKR_Zero DAC0 (Digital to Analog Converter) or to the higher fidelity UDA1334 DAC (using the I2S interface) resulting in a final analog output signal changed in some unique way from the original signal.

The source audio can come from any number of readily available audio devices -- electronic keyboard, guitar amp, mic preamp, computer, smart phone headphone out, com-

puter pad headphone out. A sinewave input is a useful signal to use when testing your signal processing application. Any readily available signal generator app on a smart phone or pad can output sinewaves at any frequency and amplitude.

The audio from all these devices is a bipolar signal that sit on zero volts and swings between both positive and negative voltages. The input to the ADC of the MKR_Zero, however, must be unipolar, swinging between zero and positive voltages only, and guaranteed to never go above 3.3 volts. Some circuitry is needed to accomodate these ADC input requirements.

For this project, special purpose LM358 opamps are used that can operate from a single power supply, in our case, +5 volts, with a full output range of zero to 5 volts.

Referring to the circuit diagram, the Audio Input jack is connected to an inverting opamp configuration (upper A) with a gain of negative one (- 100k/100k). The plus opamp input, which is normally connected to ground, in this case, is connected to a virtual ground of 1.6 volts, halfway between zero and 3.3 volts, as set by the 10k/4k7 voltage divider circuit. This sets the output signal at a bias voltage of 1.6 volts instead of the usual zero volts.

Before connecting to pin AD1, the op amp output goes through a 47k resistor and a diode tied to 3.3 volts. The diode prevents the ADC input from going above 3.3 volts.

A 300pf capacitor in parallel with a 100k resistor in the opamp feedback will drop the -1 gain for frequencies above about 5300Hz to minimize foldover frequencies, a problem with ADCs when the input signal has any frequencies above one-half the sampling frequency.

The MKR_Zero's DAC0 output can be mixed back in with the Audio Input. This creates a feedback loop useful in delay and reverb programs. The amount of feedback is controlled by a 100k pot that varies the gain of opamp B from zero to negative one. The same feedback could be accomplished in code but using circuitry allows for faster processing and sampling speeds.

Opamp A at the bottom of the circuit diagram is set up as a mixer in a non-inverting opamp configuration. A resistor mixer combines signals from 5 possible sources: ADC input, DAC0 output, UDA1334 DAC output , and any square or pulsewave signals created on the MKR_Zero pins A and B (D8, D9). Each of the 5 inputs go through 10K log pot volume controls.

The output of the opamp mixer stage is a zero to 5 volt signal. This unipolar property enables the use of a unique type of distortion circuit. Those familiar with transitor configurations may recognize this circuit as a common emitter transitor configuration with one strange difference, an audio signal is connected to the collector through 22k instead of the usual power supply voltage. The base input signal connected to the 100k resistor drives the distortion of the mixer signal. The base signal comes from either the DAC0 output or PinC (D10) as selected by a switch. The amount of distortion is controlled by

a volume control.

This simple transistor circuit has three modes of operation.

1. When the base input voltage is at zero volts, the transistor is said to be in cutoff which is an inactive state. Nothing happens to the mixed signal and it is allowed to advance unaffected to the unity gain buffer opamp B.

2. When the base input voltage is high enough, the transitor goes into saturation. This is an extremely active state which clamps the collector voltage to zero volts effectively shutting off the mixer signal.

3. A small range of input voltages between zero volts and the higher voltages causing saturation will put the transistor into its "active" state. In this state the mixer signal is allowed to pass on, but with attenuation. Higher input voltages result in a smaller mixer output approaching the highest attenuation at saturation, or zero output.

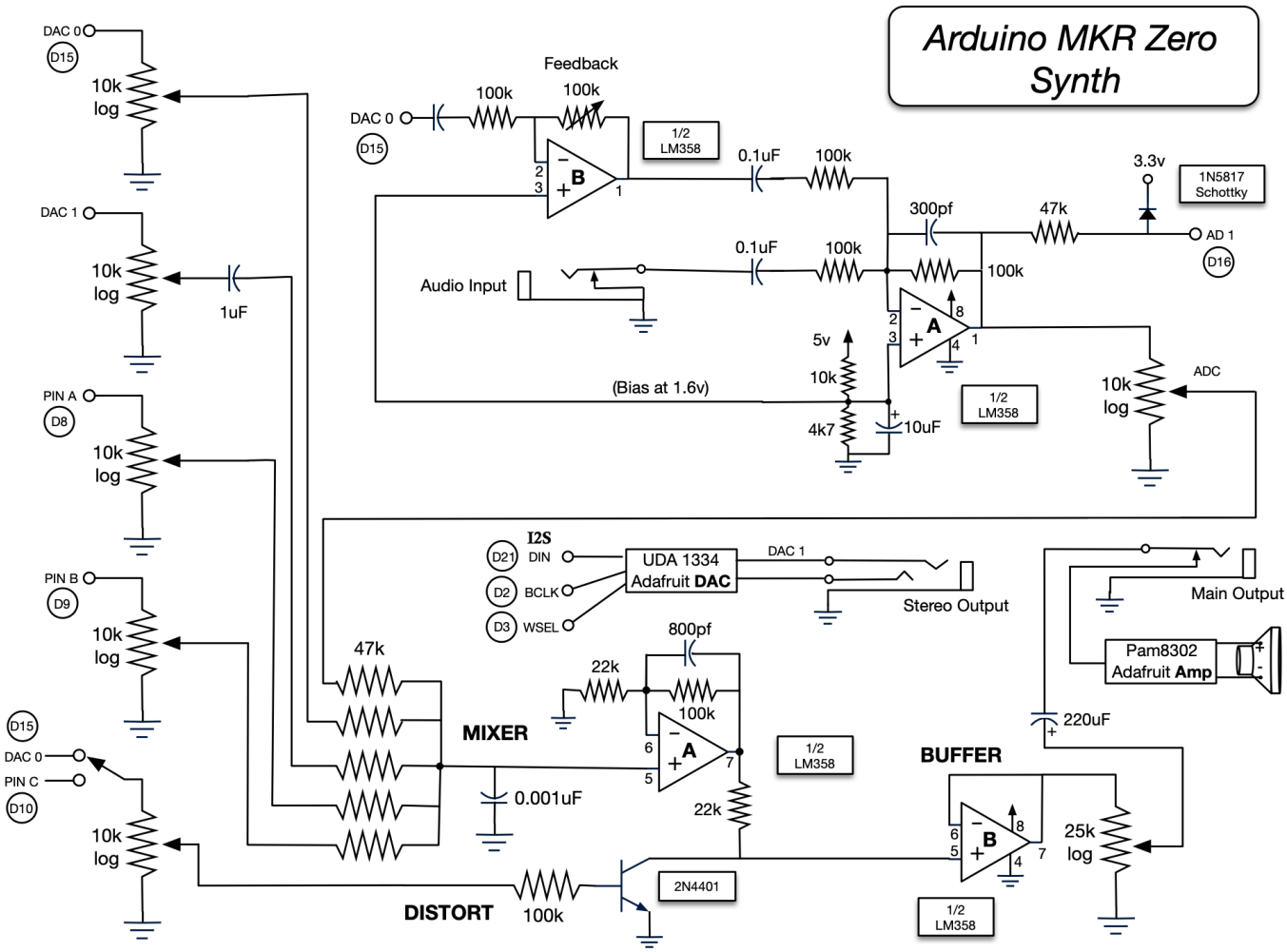
It must be noted that both the base input voltage and the collector mixer signal voltage affect the saturation and active states. It takes more input voltage to drive a higher voltage collector signal into saturation than it would for a smaller collector voltage.

To put this picture together, imagine a squarewave base input signal switching the transistor state between cutoff and saturation, or between cutoff and active, at a high frequency. The mixed signal output waveform will alternate between being unaffected and being reduced to zero or attenuated, all at the rate of the base squarewave. Turning down the volume control on the squarewave will reduce the attenuation on the affected parts of the mixed signal thus reducing the distortion. Turning the volume control all the way to zero will effectively turn off the distortion effect.

This distortion circuit has another fortunate property. When the mixed signal goes to zero, or silence, the transistor is put into its cutoff or inactive state resulting in zero output. The distortion driving signal will not feed through to the ouput when the mixed signal is silent.

The final opamp stage is a simple unity gain voltage follower. It acts as a buffer preventing the output volume control from interfering in the work of the distortion stage.

After the output volume control, the mixed output signal is sent to a jack and then onto the Pam8302 Adafruit Amplifier module which drives a small speaker in the enclosure. If any plug is inserted into the Main Output jack, the connection to the Amplifier is broken by means of a switch integrated in the jack.

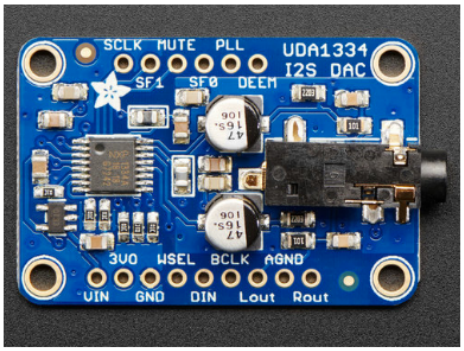

Arduino MKR Zero Synth
DAC 0
D15
10k log
DAC 1
1uF
PIN A
D8
PIN B
D9
PIN C
D10
Feedback
100k
1/2 LM358
0.1uF
3.3v
1N5817 Schottky
300pf
47k
AD 1
D16
Audio Input
5v
(Bias at 1.6v)
4k7
10uF
ADC
I2S
D21 DIN
D2 BCLK
D3 WSEL
UDA 1334 Adafruit DAC
Stereo Output
Main Output
Pam8302 Adafruit Amp
220uF
47k
800pf
22k
MIXER
0.001uF
BUFFER
25k log
2N4401
DISTORT

More Circuit Devices

Since the MKR_Zero's single DAC is only 10-bits wide a higher fidelity option was included in the project. The UDA1334 DAC module, sold by Adafruit, uses the I2S interface with three input controls - DIN, BCLK, AND WSEL. Its stereo output is labeled DAC1 and connected to a separate stereo mini jack.

Adafruit I2S Stereo Decoder - UDA1334A

Created by lady ada

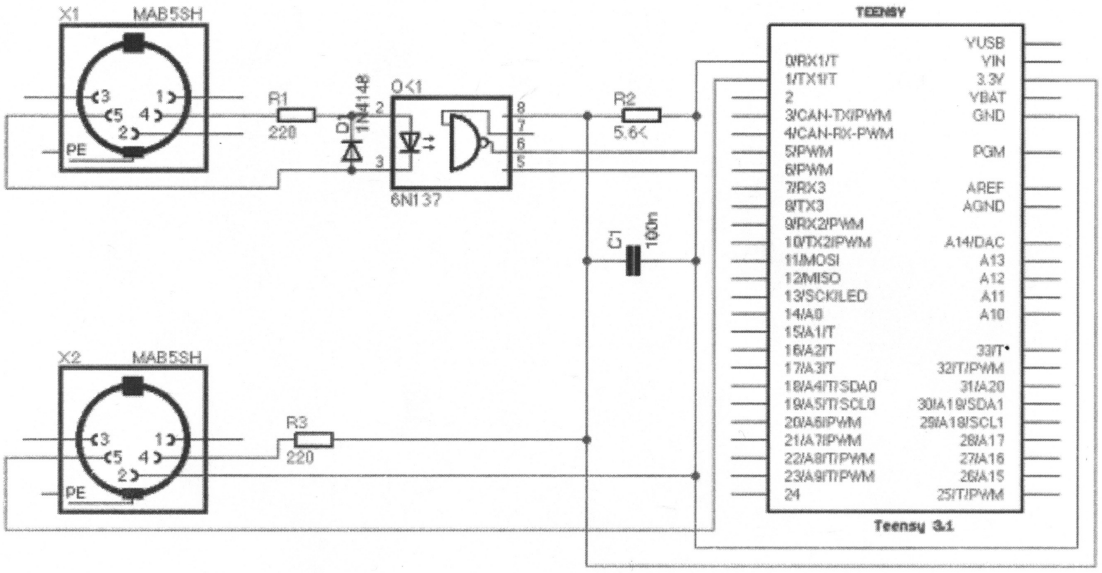


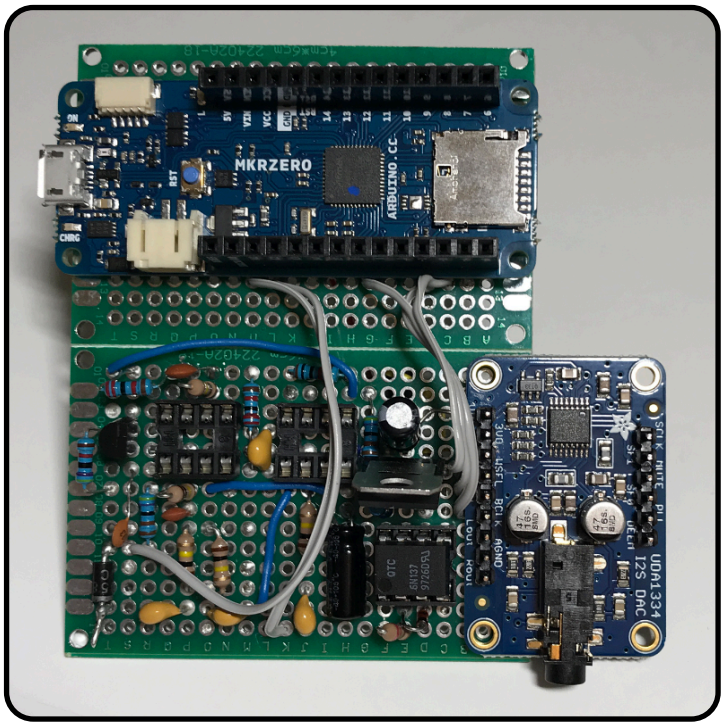
Last updated on 2020-06-15 04:21:25 PM EDT

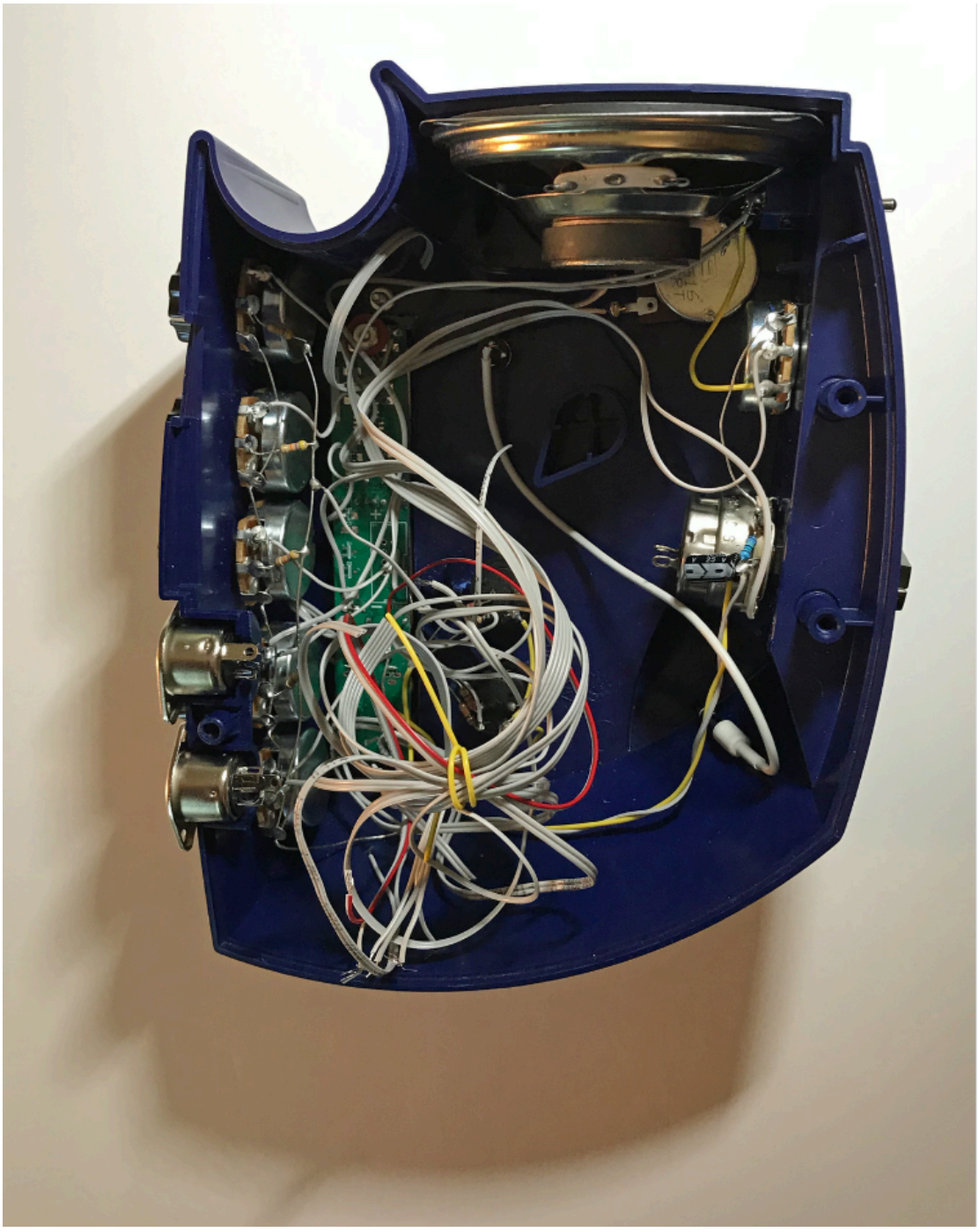
The MKR_Zero is programmed through a USB port which can be connected to a computer running the Arduino IDE application, available on the Arduino website. This same USB port can be programmed to act as a USB MIDI port using the MidiUSB library. This is an option if you want to program the MKR_Zero as a MIDI controller.

However, the MKR_Zero has an extra set of Tx and Rx serial data lines not involved with the USB port. These were wired up to provide a standard Midi IN and Midi OUT on two 5-pin DIN sockets. The minimal amount of circuitry needed for this is shown below (replace teensy chip with MKR_Zero pinouts).

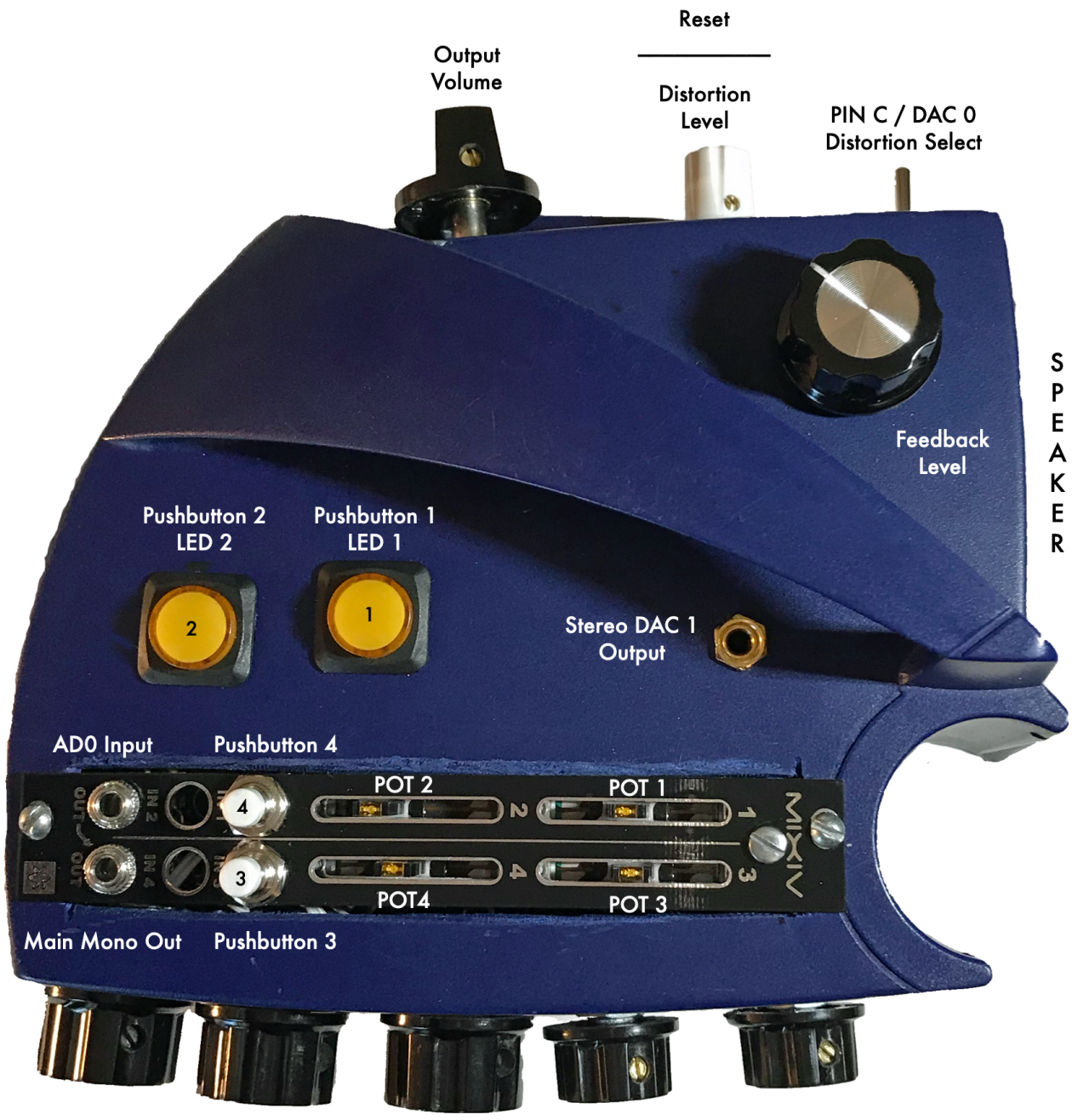
If you opt for the faster MIDI USB, you may experience problems uploading new programs from the Arduino IDE application. This is because the MidiUSB program is using the same USB port needed to upload a new program. A quick fix for this problem is to wait for the IDE to say "Uploading", then quickly double or triple click the MKR board reset button. This will stop the current program and put the microprocessor into bootloading mode. The timing is tricky and it may take a few attempts before it works. An external reset button has been mounted on the back of the project enclosure.


X1
MAB5SH
PE
R1
220
D1
1N4148
6N137
R2
5.6K
C1
100n
X2
MAB5SH
PE
R3
220
TEENSY
0/RX1/T
1/TX1/T
2
3/CAN-TX/PWM
4/CAN-RX-PWM
5/PWM
6/PWM
7/RX3
8/TX3
9/RX2/PWM
10/TX2/PWM
11/MOSI
12/MISO
13/SCK/LED
14/A0
15/A1/T
16/A2/T
17/A3/T
18/A4/T/SDA0
19/A5/T/SCL0
20/A6/PWM
21/A7/PWM
22/A8/T/PWM
23/A9/T/PWM
24
VUSB
VIN
3.3V
VBAT
GND
PGM
AREF
AGND
A14/DAC
A13
A12
A11
A10
33/T/PWM
32/T/PWM
31/A20
30/A19/SDA1
29/A18/SCL1
28/A17
27/A16
26/A15
25/T/PWM
Teensy 3.1


MKRZERO
ARDUINO.CC
UDA1334
I2S DAC



Arduino MKR Zero Synth



Sample Programs

Sensor Printout

This program prints out a running reading of the enclosure's four slide pots and four switches. The Arduino IDE's Monitor Function can be turned on from the Tools Menu. It uses the "Serial" library commands to print out the sensor values to the Monitor window.

The function "switchCombo" was included to link the two switch LEDs with their switch actions. An audio squarewave tone was also created on pinA using delay( ) functions.

At the top of the sketch is a section that defines all the constants and variables needed to use the mounted sliders, switches, LEDs, and voice pins. It is recommended that this section be copied to all your project sketches.

```
/*
Print Sensor Values. Test 3 Arduino Voices. LEDs tied to Switches.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//
// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
```

```
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;


#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED


unsigned int freq = 50;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                        SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  delay(1000);

Serial.begin(9600);

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

//noTone(VOICEPIN_A);

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  digitalWrite(VOICEPIN_A, LOW);
  delay(slider1);

 int x = switchCombo();

  loadSensors();
```

```
   Serial.print("s1 = ");
  Serial.print(slider1);
    Serial.print("  s2 = ");
  Serial.print(slider2);
    Serial.print("   s3 = ");
  Serial.print(slider3);
    Serial.print("   s4 = ");
  Serial.print(slider4);
    Serial.print("   s5 = ");
  Serial.print((50 + (slider3 >> 1)));

 Serial.print("     ");

  Serial.print("  switches ");
  Serial.print(switch1);
  Serial.print(switch2);
  Serial.print(switch3);
  Serial.println(switch4);


  //tone(VOICEPIN_A, freq);
  digitalWrite(VOICEPIN_A, HIGH);
  delay(slider1);

/*
freq = 0 + (slider1 >> 1);
//tone(VoicePinA, freq);

//int v = map(slider2, 0, 1024, 1, 255);

 digitalWrite(VoicePinA, HIGH);
 //  digitalWrite(VoicePinB, HIGH);

  delay(freq );
  digitalWrite(VoicePinA, LOW);
 // digitalWrite(VoicePinB, LOW);
  delay(10);
*/


} //End of Loop

//``````````````````````````````````````````````````````````````

void loadSensors(){      // load all current sensor values
      slider1 =   analogRead(SLIDER1) ;
      slider2 =   analogRead(SLIDER2) ;
      slider3 =   analogRead(SLIDER3) ;
      slider4 =   analogRead(SLIDER4) ;

      switch1 = digitalRead(SWITCH1);
      switch2 = digitalRead(SWITCH2);
      switch3 = digitalRead(SWITCH3);
      switch4 = digitalRead(SWITCH4);
}

int switchCombo(){
    int result = switch2 + (switch1 * 2);

    switch (result) {
      case 0:
              digitalWrite(LED2, LOW);
              digitalWrite(LED1, LOW);
              break;
```

```
      case 1:
              digitalWrite(LED2, HIGH);
              digitalWrite(LED1, LOW);
              break;
      case 2:
              digitalWrite(LED2, LOW);
              digitalWrite(LED1, HIGH);
              break;
      case 3:
              digitalWrite(LED2, HIGH);
              digitalWrite(LED1, HIGH);
              break;
    }
    return result;
}
```

Tone Generation

Three digital pins from the MKR_Zero are set aside for tone generation. Voice A and B (D8 and D9) are connected to the circuit mixer after volume controls. Voice C drives the distortion circuit described above.

The Arduino function tone( ) makes it easy to generation a squarewave tone on any digital pin. Its major limitation, however is that it can only be used on one voice at a time. Some other limitation to watch out for: the function will crash if called faster than once every 50 milliseconds, and the voice output clicks whenever the function is called.

A second method for generating a tone uses the delay( ) function in a loop. The voice pin is toggled each time through the loop. The value of the delay sets the frequency of the tone. Two different delay values, one for high and one for low, can create a pulse waveform instead of a squarewave. The main problem with this method is that the delay function stops all program activity for the duration of the delay.

This first sketch uses the tone( ) function for one of the voices and one delay( ) function for the other two voices. One voice pin is toggled every time through the main loop while the second voice pin is toggled every other time through the main loop, making it an octave lower. One slider sets the frequency value in the tone( ) function. A second slider sets the delay( ) value and thus the other voice frequencies.

```
/*

                Test 3 Arduino Voices

VoiceA on volume knob A uses the tone() function with Slider 1 controlling freq
VoiceB on volume knob B uses the delay() function with Slider 2 controlling freq
VoiceC on distortion knob uses the same delay() function as Voice B
```

```
Voice C is an octave higher in frequency than Voice B
Voice C is used to distort Voices A and B as controlled by the Distortion knob
    with the switch in the up position.

*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//
// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

unsigned int freq = 50;
unsigned int lastfreq = 50;
bool bb = 1;
bool cc = 1;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
```

```
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);


}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

 bb = !bb;  // toggle the boolean bb value
 if (bb){ cc = !cc; }  // toggle the boolean cc value at 1/2 the freq of bb

 digitalWrite(VOICEPIN_C, bb);
 digitalWrite(VOICEPIN_B, cc);

 //slider2 affects freq of Voices B&C
 delayMicroseconds(analogRead(SLIDER2) << 1 );

//  tone() is used to set up VoiceA at freq set by slider1.
//  there is a click whenever the freq value in tone() is changed
//  so new freq values are only loaded when slider1 changes more than 10

  freq = analogRead(SLIDER1) ;
  if ( abs(freq - lastfreq) > 10) {
        tone(VOICEPIN_A, 40 + freq);
        lastfreq = freq;
  }

} //End of Loop

//````````````````````````````````````````````````````````````````
```

Tone Generation with Loop Time

In this second script we use the tone( ) function for Voice3, the modulating voice, but dispense with the delay( ) function used in the previous sketch for the other two voices. The main loop time is very fast now, measured in microseconds, and is used to generate Voices 1 and 2.

The frequencies for Voices 1 and 2 can be set with two slide pots independently. Each voice is given a counter that decrements each time through the main loop, from a value set by its assigned slider. When the counter reaches zero, the voice pin is toggled and the slider value is reloaded. The main loop times are now very fast but not consistenly the same duration, resulting in some voice frequency jitters and phasing, mostly affecting the higher frequencies with the lower counter values.

The main loop time is now too fast for the tone( ) function, so Voice3 frequency changes from its assigned slider value are limited to taps on one of the pushbutton switches.

```
/*
       3-VOICE ARDUINO SYNTHESIZER

      3 squarewave tones produced from Arduino pins

      Slider3 - Sets Frequency of Tone3 with arduino tone(), used as the modulating voice
      Slider1 - Sets Frequency of Tone1
      Slider2 - Sets Frequency of Tone2

      Switch1 - turns off or on Tone1 slider adjustments
      Switch2 - turns off or on Tone2 slider adjustments
      Switch3 - Loads slider value for modulating voice

Arduino's tone() function can only be used to set up a squarewave on one output.
Arduino's tone() function also will stop working if loaded faster than every 50ms
Arduino's tone() function pops when loaded with the same freq value.

Two voices are created from a fast loop clock decrementing two freq values and toggling voice outputs
when they reach zero.  The freq values determine the pitch of the voices. Voice pitches are
also affected by any changes in the main program loop speed (other voices changing frequencies or
going in or out of IF statements).  Higher frequencies affected more than lower.  Changes in the loop
time cause phasing between the two voices.


*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//
//
// ANALOG INPUTS
//
const int Slider1 = A5;
const int Slider2 = A3;
const int Slider3 = A4;
const int Slider4 = A2;

int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
const int Switch1 = 5;
const int Switch2 = 4;
const int Switch3 = 6;
const int Switch4 = 7;

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

const int VoicePinA = 8;
const int VoicePinB = 9;
const int VoicePinC = 10;
```

```
const int LED1 = 1;
const int LED2 = 0;

unsigned int freq1 = 50;
unsigned int freq2 = 50;
unsigned int freq3 = 50;

unsigned int freq1save = 50;
unsigned int freq2save = 50;
unsigned int dummy = 50;

boolean lastSwitch = 0;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


void setup() {

pinMode(LED1, OUTPUT);    //turn on LED1 as power indicator
digitalWrite(LED1, HIGH);

pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);

pinMode(VoicePinA, OUTPUT);
digitalWrite(VoicePinA, LOW);
pinMode(VoicePinB, OUTPUT);
digitalWrite(VoicePinB, LOW);
pinMode(VoicePinC, OUTPUT);
digitalWrite(VoicePinC, LOW);

pinMode(Switch1, INPUT);  // Set up switch inputs with pullup resistor
pinMode(Switch2, INPUT);
pinMode(Switch3, INPUT);
pinMode(Switch4, INPUT);

} //End of Setup

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      MAIN LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {
//keep loop time as low as possible for better voice ranges, limit analogReads

     switch4 = digitalRead(Switch4);
//used to load slider3 setting for modulating voice frequency

    if ((switch4 == 0) && (lastSwitch == 1)){
 // loads only on high to low switch transition (edge)

        tone(VoicePinC, 50 + (analogRead(Slider3) << 2));
        lastSwitch = 0;
    }
    else if ((switch4 == 1) && (lastSwitch == 0)){
        lastSwitch = 1;
    }


    --freq1;                          //toggle voicePinA at end of freq1 countdown
    if (freq1 <= 0){
```

```
        digitalWrite(VoicePinA, !digitalRead(VoicePinA));
        freq1 = freq1save;

        if (digitalRead(Switch2)){    // turn off or on VoicePinA frequency adjustments, hold a pitch
          freq1save = 2 + (analogRead(Slider2) << 1);
        }
    }

     --freq2;                          //toggle voicePinB at end of freq2 countdown
    if (freq2 <= 0){
        digitalWrite(VoicePinB, !digitalRead(VoicePinB));
        freq2 = freq2save;

        if (digitalRead(Switch1)){   // turn off or on VoicePinB frequency adjustments
          freq2save = 2 + (analogRead(Slider1) << 1);
        }

    }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                     END OF MAIN LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Tone Generation with Timer

This next sketch illustrates how to set up an interrupt timer on an Atmel SAMD21 Arduino board (MKR Zero) with an ARM Cortex M0+ processor. Hidden from the Arduino IDE documentation for regular users is a CMSIS library that allows the programmer to manipulate SAMD21 registers directly. This is dangerous stuff prone to crashing and "bricking" the processor. Here it is used to set up timers and interrupts. This is not an easy thing to do and basically involves hacking into the internal workings of the SAMD21. Thanks is due to Michael Blank for delving into the ARM Cortex manuals to create his TIMER5 library on which this sketch is based.

The last sketch was beset with voice frequency fluctuations caused by a loop time that was not consistent. We can avoid using the loop time and instead set up an internal register decremented by a rock steady internal clock, our own internal timer. During the time the internal register decrements our own script can go about its business checking switches and loading slide values without affecting this internal timer. Once it reaches zero an interrupt ocurrs and all script business is temporarily halted so that a short interrupt routine can be performed. Within this interrupt routine the three voice counters are decremented and the voice pins are toggled when the counters reach zero, the same program scheme used in the previous sketch.

```
/*

 *  3-VOICE ARDUINO SYNTHESIZER USING TIMER INTERRUPTS
```

```
      3 squarewave tones produced from Arduino pins

      Slider3 - Sets Frequency of Tone3  used as the modulating voice
      Slider1 - Sets Frequency of Tone1
      Slider2 - Sets Frequency of Tone2

      Switch1 - turns off or on Tone1 slider adjustments
      Switch2 - turns off or on Tone2 slider adjustments
      Switch4 - Loads slider value for modulating voice

Voices are created from a master timer interrupt.

The interrupt function is run at regular intervals as set by a very high speed timer.
The timed interrupt function decrements freq values and toggles digital pin outputs
when they reach zero.  Freq values determine the pitch of the voices and are varied from sliders.

This sketch illustrates how to set up a timer on an Atmel SAMD21 based Arduino board (MKR Zero)
based on the ARM Cortex M0+.  The Arduino IDE includes the CMSIS
library for working with the SAMD21 registers directly.  Here it is used to set up timers and
interrupts.

Thanks to:
Timer5 library for Arduino Zero and MKR1000
  (only for SAMD arch.)
  Copyright (c) 2016 Michael Blank, OpenSX. All right reserved.

  based on the code of the AudioZero library by:
  Arturo Guadalupi <a.guadalupi@arduino.cc>
  Angelo Scialabba <a.scialabba@arduino.cc>
  Claudio Indellicati <c.indellicati@arduino.cc> <bitron.it@gmail.com>
*/

uint32_t sampleRate = 20000; //sample rate, determines how often TC5_Handler is called
//
// ANALOG INPUTS
//
#define SLIDER1 A5
#define SLIDER2 A3
#define SLIDER3 A4
#define SLIDER4 A2


int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5
#define SWITCH2 4
#define SWITCH3 6
#define SWITCH4 7

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8
#define VOICEPIN_B 9
#define VOICEPIN_C 10
```

```
#define LED1 1
#define LED2 0

volatile int freq1 = 50;
volatile int freq2 = 50;
volatile int freq3 = 50;

volatile int freq1save = 50;
volatile int freq2save = 50;
volatile int freq3save = 50;
unsigned int load = 50;

unsigned int swtch = 0;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

delay(1000);

pinMode(LED1, OUTPUT);    //turn on LED1 as power indicator
digitalWrite(LED1, HIGH);

pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT);  // Set up switch inputs with pullup resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

  tcConfigure(sampleRate); //configure the timer to run at <sampleRate>Hertz

  tcStartCounter(); //starts the timer

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  //tcDisable(); //This function can be used anywhere if you need to stop/pause the timer
  //tcReset(); //This function should be called everytime you stop the timer

 --load;  //Slider readings are limited since they cause noise in the higher voice frequencies.

 if (load <= 0){

  ++swtch;
  if (swtch >=4){ swtch = 1; }

      switch (swtch) {
  case 1:
    if (digitalRead(SWITCH1)){
        freq2save = 2 + (analogRead(SLIDER1) >> 1);
```

```
      }
    break;
  case 2:
    if (digitalRead(SWITCH2)){
        freq1save =  2 + (analogRead(SLIDER2) >> 2);
      }
    break;
  case 3:
    if (digitalRead(SWITCH4)){
        freq3save =  2 + (analogRead(SLIDER3) >> 1);
      }
    break;
}

     load = 300;
  }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//this function gets called by the interrupt at a high <sampleRate>Hertz
// "TC5_Handler" is in the CMSIS library, doesn't need to be declared as an interrupt handler.

void TC5_Handler (void) {

  --freq1;                          //toggle voicePinA at end of freq1 countdown
    if (freq1 <= 0){
        digitalWrite(VOICEPIN_A, !digitalRead(VOICEPIN_A));
        freq1 = freq1save;
    }

     --freq2;                          //toggle voicePinB at end of freq2 countdown
    if (freq2 <= 0){
        digitalWrite(VOICEPIN_B, !digitalRead(VOICEPIN_B));
        freq2 = freq2save;
    }

     --freq3;
      if (freq3 <= 0){
        digitalWrite(VOICEPIN_C, !digitalRead(VOICEPIN_C));
        freq3 = freq3save;
    }

  TC5->COUNT16.INTFLAG.bit.MC0 = 1;
  //Writing a 1 to INTFLAG.bit.MC0 clears the interrupt so that it will run again
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/*
 *  TIMER SPECIFIC FUNCTIONS FOLLOW
 *  you shouldn't change these unless you know what you're doing
 */

//Configures the TC to generate output events at the sample frequency.
//Configures the TC in Frequency Generation mode, with an event output once
//each time the audio sample frequency period expires.

 void tcConfigure(int sampleRate)
{
 // Enable GCLK for TCC2 and TC5 (timer counter input clock)
 //drive the timer from General Clock 0 (CPU clock) and enable the peripheral clock.

        // GCLK1 is 32kHz while GCLK0 is 48MHz
```

```
 GCLK->CLKCTRL.reg = (uint16_t) ( GCLK_CLKCTRL_CLKEN |              // Enable clock
                                  GCLK_CLKCTRL_GEN_GCLK0 |          // Select GCLK0 (is 48MHz)
                                  GCLK_CLKCTRL_ID(GCM_TC4_TC5)) ;   // Feed GCLK3 to TC4 and TC5
 while (GCLK->STATUS.bit.SYNCBUSY);                                 // Wait for synchronization
 //accounts for the need to synchronize between different clocks in the chip before using the changed
value.



 tcReset();   //resets the timer TC5 settings.



 TC5->COUNT16.CTRLA.reg |= TC_CTRLA_MODE_COUNT16;    // Set TC5 Timer counter to be in 16 bit mode and
generate a frequency.
 TC5->COUNT16.CTRLA.reg |= TC_CTRLA_WAVEGEN_MFRQ;    // Set TC5 mode as match frequency
 TC5->COUNT16.CTRLA.reg |= TC_CTRLA_PRESCALER_DIV1 | // Set TC5 prescaler to 1 (can be any power of 2)
                           TC_CTRLA_ENABLE;          // Set TC5 clock enable
 TC5->COUNT16.CC[0].reg  = (uint16_t) (SystemCoreClock / sampleRate - 1);
 while (tcIsSyncing());                              // Wait for TC5 synchronization

 //set TC5 timer counter based off of the system clock (48MHZ) and the user defined sample rate or
waveform
 // higher sampleRate value results in more frequent interrupts.



 // Configure interrupt request
 NVIC_DisableIRQ(TC5_IRQn);
 NVIC_ClearPendingIRQ(TC5_IRQn);
 NVIC_SetPriority(TC5_IRQn, 0);
 NVIC_EnableIRQ(TC5_IRQn);

 // Enable the TC5 interrupt request
 TC5->COUNT16.INTENSET.bit.MC0 = 1;

 while (tcIsSyncing()); //wait until TC5 is done syncing
}



//Function that is used to check if TC5 is done syncing
//returns true when it is done syncing
bool tcIsSyncing()
{
  return TC5->COUNT16.STATUS.reg & TC_STATUS_SYNCBUSY;
}



//This function enables TC5 and waits for it to be ready
void tcStartCounter()
{
  TC5->COUNT16.CTRLA.reg |= TC_CTRLA_ENABLE; //set the CTRLA register
  while (tcIsSyncing()); //wait until snyc'd
```

```
}



//Reset TC5
void tcReset()
{
  TC5->COUNT16.CTRLA.reg = TC_CTRLA_SWRST;
  while (tcIsSyncing());
  while (TC5->COUNT16.CTRLA.bit.SWRST);
}



//disable TC5
void tcDisable()
{
  TC5->COUNT16.CTRLA.reg &= ~TC_CTRLA_ENABLE;
  while (tcIsSyncing());
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This Tone Generation script operates like the previous script except that it uses the Timer5 Library created by Michael Blank.

```
/*
       3-VOICE ARDUINO SYNTHESIZER

      Uses the library Timer5Lib to set up a timer interrupt routine

      3 squarewave tones produced from Arduino pins

      Slider3 - Sets Frequency of Tone3 used as the modulating voice
      Slider1 - Sets Frequency of Tone1
      Slider2 - Sets Frequency of Tone2

      Switch1 - turns off or on Tone1 slider adjustments
      Switch2 - turns off or on Tone2 slider adjustments


Three voices are created from a timer interrupt, decrementing 3 freq values and toggling voice outputs
when they reach zero.  Freq values determine the pitch of the voices. Voice pitches are
also affected by any changes in the main program loop speed (other voices changing frequencies or
loading slider and swtich values).  Higher frequencies affected more than lower. Changes in the loop
time causes phasing between the two voices.


*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//
//
// ANALOG INPUTS
//
```

```
const int Slider1 = A5;
const int Slider2 = A3;
const int Slider3 = A4;
const int Slider4 = A2;


int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
const int Switch1 = 5;
const int Switch2 = 4;
const int Switch3 = 6;
const int Switch4 = 7;

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

const int VoicePinA = 8;
const int VoicePinB = 9;
const int VoicePinC = 10;

const int LED1 = 1;
const int LED2 = 0;

volatile int freq1 = 50;
volatile int freq2 = 50;
volatile int freq3 = 50;

volatile int freq1save = 50;
volatile int freq2save = 50;
volatile int freq3save = 50;
unsigned int load = 50;

boolean lastSwitch = 0;

#include "Timer5.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


void setup() {

delay(1000);

pinMode(LED1, OUTPUT);    //turn on LED1 as power indicator
digitalWrite(LED1, HIGH);

pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VoicePinA, OUTPUT);
digitalWrite(VoicePinA, LOW);
pinMode(VoicePinB, OUTPUT);
digitalWrite(VoicePinB, LOW);
```

```
pinMode(VoicePinC, OUTPUT);
digitalWrite(VoicePinC, LOW);

pinMode(Switch1, INPUT);  // Set up switch inputs with pullup resistor
pinMode(Switch2, INPUT);
pinMode(Switch3, INPUT);
pinMode(Switch4, INPUT);

   // define frequency of interrupt
  MyTimer5.begin(15000);  // 200=for toggle every 5msec

    // define the interrupt callback function
    MyTimer5.attachInterrupt(timerInterrupt);

    // start the timer
    MyTimer5.start();
} //End of Setup

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       MAIN LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {  //keep loop time as low as possible for better voice ranges, limit analogReads

    --load;     //Do all the reads every 1000 times through the loop.  Causes some pops and fuzzyness
                //to the output tones, since it stretches one cycle time of the tones periodically
                //Switches used to stop and start analogReads - a hold frequency function.

    if (load <= 0){


      if (digitalRead(Switch1)){
        freq2save = 2 + (analogRead(Slider1) >> 2);
      }

      if (digitalRead(Switch2)){
        freq1save = 2 + (analogRead(Slider2) >> 2);
      }

        freq3save = 2 + (analogRead(Slider3) >> 1);

        load = 10000;
   }



}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                     END OF MAIN LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void timerInterrupt(void){

   --freq1;                           //toggle voicePinA at end of freq1 countdown
    if (freq1 <= 0){
        digitalWrite(VoicePinA, !digitalRead(VoicePinA));
        freq1 = freq1save;
    }

     --freq2;                          //toggle voicePinB at end of freq2 countdown
    if (freq2 <= 0){
        digitalWrite(VoicePinB, !digitalRead(VoicePinB));
        freq2 = freq2save;
    }
```

```
     --freq3;
      if (freq3 <= 0){
        digitalWrite(VoicePinC, !digitalRead(VoicePinC));
        freq3 = freq3save;
    }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

MIDI Output

A traditional MIDI Out 5-pin DIN socket is very easy to implement on the MKR_Zero. The Zero has an extra set of serial interface pins that can be used for MIDI. Connect the TX pin (D14) to pin 5 of the MIDI socket and connect pin 4 of the socket to a 220 ohm resistor and then to the 3.3 volt output pin of the Zero. In the sketch Setup, MIDI is initialized with the line "Serial1.begin(31250)" where 31250 is the MIDI baud rate. Then the command Serial1.write( ) is used to feed the MIDI output with the appropriate MIDI data bytes.

The first sketch here demonstrates a simple MIDI output application. The second sketch is a performance program that outputs clouds of random notes controlled by the sliders and switches. It demonstrates a way to share three sliders with 3 separate voices controlling a total of 9 parameters.

The MIDI Input circuit is a bit more involved as shown the previous circuit diagram. MIDI Input software is set up like an interrupt. The **Arduino Midi** Library uses something called 'Callbacks'. When a **Midi** event occurs, the Library will Call a function to handle it.

```
/*
     Simple MIDI output player for Serial1

  Generates a series of 12 MIDI notes.
  The melody is Steve Reich's "Piano phase"

  Uses Serial1 for MIDI, so will work on any board
  with 2 hardware serial ports: MKR boards, Leonardo, Micro, or Yún

  On the MKR Zero TX is D14, RX is D13

  Circuit:
    connect TX of Serial1 to pin5 of MIDI jack
    and connect pin4 of MIDI jack to a 220 ohm resistor to 3.3v power
*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//
// ANALOG INPUTS
```

```
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED


int bpm = 72;  // beats per minute
// duration of a beat in ms
float beatDuration = 60.0 / bpm * 1000;

// the melody sequence:
int melody[] = {64, 66, 71, 73, 74, 66, 64, 73, 71, 66, 74, 73};
// which note of the melody to play:
int noteCounter = 0;


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  delay(1000);

  // initialize MIDI serial:
  Serial1.begin(31250);  //MIDI rate

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, HIGH);

pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);
```

```
pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {
  // play a note from the melody:
  midiCommand(0x90, melody[noteCounter], 127);

//   Choose one of the follow note durations by uncommenting it:
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // all the notes in this are sixteenth notes,
  // which is 1/4 of a beat, so:
  //int noteDuration = beatDuration / 4;

  //or choose tempo set by slider1
    int noteDuration = map(analogRead(SLIDER1), 0, 1024, 50, 1000);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  // keep it on for the appropriate duration:
  delay(noteDuration);
  // turn the note off:
  midiCommand(0x90, melody[noteCounter], 0);
  // increment the note number for next time through the loop:
  noteCounter++;
  // keep the note in the range from 0 - 11 using modulo:
  noteCounter = noteCounter % 12;

}  //End of Main Loop

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                     Functions
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// send a 3-byte midi message

void midiCommand(byte cmd, byte data1, byte  data2) {
  Serial1.write(cmd);     // command byte (should be > 127)
  Serial1.write(data1);   // data byte 1 (should be < 128)
  Serial1.write(data2);   // data byte 2 (should be < 128)
}

void loadSensors(){      // load all current sensor values
      slider1 =   analogRead(SLIDER1) ;
      slider2 =   analogRead(SLIDER2) ;
      slider3 =   analogRead(SLIDER3) ;
      slider4 =   analogRead(SLIDER4) ;

      switch1 = digitalRead(SWITCH1);
      switch2 = digitalRead(SWITCH2);
      switch3 = digitalRead(SWITCH3);
      switch4 = digitalRead(SWITCH4);
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
/*
     MIDI 3-voice random notes

  For each of 3 voices, the sliders can change Frequency Base, Frequency Range,
  Note Amplitude and Note Duration.  Since there are only 4 sliders, Slider 4
  is used for Note Duration, and sliders 1, 2, and 3 are shared between the 3 voices
  to affect Base, Range, and Amplitude.

  Two toggle switches determine which of the 3 voices is affected by the first 3 sliders.

  A Slider position is divided into 5 ranges. Each of the 5 slider ranges is assigned an
  increment value of +2, +1, 0, -1, or -2.  Each time through the main program loop
  the Note parameter is incremented (or decremented) by the value assigned to that slider position.

  Sliders set in the middle have an increment value of zero, resulting in no
  change to any note parameters.

  Uses Serial1 for MIDI, so will work on any board
  with 2 hardware serial ports: MKR boards, Leonardo, Micro, or Yún

  On the MKR Zero TX is D14, RX is D13

  Circuit:
    connect TX of Serial1 to pin5 of MIDI jack
    and connect pin4 of MIDI jack to a 220 ohm resistor to 3.3v power
*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//
// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;
int swtchCombo = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED
```

```
//
//  3-VOICE VARIABLES
//

int dur = 200;
int durA = 0;
int durB = 0;
int durC = 0;

int baseA = 0;
int baseB = 0;
int baseC = 0;

int baseAincrement = 0;
int baseBincrement = 0;
int baseCincrement = 0;

int rangeA = 0;
int rangeB = 0;
int rangeC = 0;

int rangeAincrement = 0;
int rangeBincrement = 0;
int rangeCincrement = 0;

int freqA = 50;
int freqB = 75;
int freqC = 100;

int ampA = 0;
int ampB = 0;
int ampC = 0;

int ampAincrement = 0;
int ampBincrement = 0;
int ampCincrement = 0;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  delay(1000);

  // initialize MIDI serial:
  Serial1.begin(31250);

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, HIGH);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
```

```
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  delay(1);

  loadSensors();

  zeroIncrements();

  swtchCombo = switchCombo();

  switch (swtchCombo){  //sliders affect voice set by switches
    case 1:
            baseAincrement = sliderIncrement(slider1);
            rangeAincrement = sliderIncrement(slider2);
            ampAincrement = sliderIncrement(slider3);
    break;

    case 2:
            baseBincrement = sliderIncrement(slider1);
            rangeBincrement = sliderIncrement(slider2);
            ampBincrement = sliderIncrement(slider3);
    break;

    case 3:
            baseCincrement = sliderIncrement(slider1);
            rangeCincrement = sliderIncrement(slider2);
            ampCincrement = sliderIncrement(slider3);
    break;

    default:
    break;

  }


// ------------Voice A ---------------------

     if (durA > 0){  --durA;  }   //wait for a count of durA

     else{  // when envelope reaches zero, reset voice A with new frequency and envelope

            midiCommand(0x90, freqA, 0);  //turn off note

            baseA = baseA + baseAincrement;  //get new pitch base for voice A
            baseA = constrain(baseA, 20, 100);

            rangeA = rangeA + rangeAincrement; //get new pitch range for voice A
            rangeA = constrain(rangeA, 0, 50);

            freqA = freqA + random(rangeA);  //calculate new pitch for voice A
            freqA = constrain(freqA, 20, 100); //MIDI note constraints

            ampA = ampA + ampAincrement;  //get amp for voice A
            ampA = constrain(baseA, 0, 127); //MIDI note velocity constraints
```

```
            durA = random(200, 200 + slider4) ;  // get random duration

            midiCommand(0x90, freqA, ampA);  //turn on new note
   }



 // ------------Voice B ---------------------

     if (durB > 0){  --durB;  }   //wait for a count of durB

     else{  // when envelope reaches zero, reset voice A with new frequency and envelope

            midiCommand(0x90, freqB, 0);  //turn off note

            baseB = baseB + baseBincrement;  //get new pitch base for voice B
            baseB = constrain(baseB, 20, 100);

            rangeB = rangeB + rangeBincrement; //get new pitch range for voice B
            rangeB = constrain(rangeB, 0, 50);

            freqB = freqB + random(rangeB);  //calculate new pitch for voice B
            freqB = constrain(freqB, 20, 100); //MIDI note constraints

            ampB = ampB + ampBincrement;  //get amp for voice B
            ampB = constrain(baseB, 0, 127); //MIDI note velocity constraints

            durB = random(200, 200 + slider4) ;  // get random duration

            midiCommand(0x90, freqB, ampB);  //turn on new note
   }


// ------------Voice C ---------------------

     if (durC > 0){  --durC;  }   //wait for a count of durC

     else{  // when envelope reaches zero, reset voice A with new frequency and envelope

            midiCommand(0x90, freqC, 0);  //turn off note

            baseC = baseC + baseCincrement;  //get new pitch base for voice C
            baseC = constrain(baseC, 20, 100);

            rangeC = rangeC + rangeCincrement; //get new pitch range for voice C
            rangeC = constrain(rangeC, 0, 50);

            freqC = freqC + random(rangeC);  //calculate new pitch for voice C
            freqC = constrain(freqC, 20, 100); //MIDI note constraints

            ampC = ampC + ampCincrement;  //get amp for voice A
            ampC = constrain(baseC, 0, 127); //MIDI note velocity constraints

            durC = random(200, 200 + slider4) ;  // get random duration

            midiCommand(0x90, freqC, ampC);  //turn on new note
   }

 // ------------Switch Slows everything to almost a standstill----------------------

                     if (switch3 == 1){ delay(250); }
} // End of Loop

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
//                    Functions
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


// send a 3-byte midi message
void midiCommand(byte cmd, byte data1, byte  data2) {
  Serial1.write(cmd);     // command byte (should be > 127)
  Serial1.write(data1);   // data byte 1 (should be < 128)
  Serial1.write(data2);   // data byte 2 (should be < 128)
}

void loadSensors(){      // load all current sensor values
      slider1 =   analogRead(SLIDER1) ;
      slider2 =   analogRead(SLIDER2) ;
      slider3 =   analogRead(SLIDER3) ;
      slider4 =   analogRead(SLIDER4) ;

      switch1 = digitalRead(SWITCH1);
      switch2 = digitalRead(SWITCH2);
      switch3 = digitalRead(SWITCH3);
      switch4 = digitalRead(SWITCH4);
}

int switchCombo(){   // set LEDs to reflect switch state, switch down = LED on
    int result = switch2 + (switch1 * 2);

    switch (result) {
      case 0:
              digitalWrite(LED2, LOW);
              digitalWrite(LED1, LOW);
              break;
      case 1:
              digitalWrite(LED2, HIGH);
              digitalWrite(LED1, LOW);
              break;
      case 2:
              digitalWrite(LED2, LOW);
              digitalWrite(LED1, HIGH);
              break;
      case 3:
              digitalWrite(LED2, HIGH);
              digitalWrite(LED1, HIGH);
              break;
    }
    return result;
}

int sliderIncrement(int slider){  //slider position determines incremental changes
    int result;

    if((slider <= 600) && (slider > 400)){ result = 0; } //do nothing middle state
    else if((slider > 800)){ result = 2; }  // double increase
    else if((slider <= 200)){ result = -2; }  // double decrease
    else if((slider <= 400) && (slider > 200)){ result = -1; }  //single decrease
    else { result = 1; }  //single increase  (range between 600 and 800)

    return result;
}

void zeroIncrements(){   //set to no incremental changes in freq. base, range and amp

            baseAincrement = 0;
            rangeAincrement = 0;
            ampAincrement = 0;
```

```
            baseBincrement = 0;
            rangeBincrement = 0;
            ampBincrement = 0;

            baseCincrement = 0;
            rangeCincrement = 0;
            ampCincrement = 0;
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Signal Processing with ADC to DAC0

The MKR_Zero has several 12-bit Analog to digital converters and one 10-bit Digital to Analog Converter (DAC). In this project pin A1 is used as the ADC. Special opamp circuitry restricts the audio input signal to a range of zero to 3.3 volts before being sent to pin A1. Once converted to a stream of digital data the programmer can manipulate it in any number of ways before sending it back out to DAC0 (pin A0).

The first sketch simply sends the output of the ADC back out to the DAC without any processing. The main loop has only these two lines:

```
sample = analogRead(A1);
analogWrite(A0, sample);
```

As an option, on either side of these two lines, are statements using the Arduino micros( ) function designed to calculate and print the time it takes to perform an ADC to DAC sample conversion. The Arduino ADCs are set up to deal with fairly slow moving data. The time it normally takes to perform the two ADC/DAC operations turned out to be 838 microseconds which is a sample rate of 1193Hz. This low sampling rate is unusable for audio signals.

The fix for this low sampling rate problem is to hack into the ADC clocking registers and speed up the conversion. This involves searching through the CMSIS library manuals for the ARM Cortex M0+ ADC operations, or you can do a web search for code written by good people who have solved this problem.

```
            ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
            while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

        //Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
            ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1
```

After inserting this code into the script's Setup Section, the sampling time dropped to around 24 microseconds which is a sampling rate of 42kHz. The GCLK divisor was dropped to 32 instead of the normal 512, and the Sampling Time Length was dropped

from 63 to 1.

```
/*
         Test DAC0 and ADC1

Send output of ADC1 directly to DAC0.

*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//
// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

int sample;
long timeIn;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

//    Serial.begin(500000);  //used only for printing samplerate times

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);
```

```
pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

// Speed up the ADC Sampling Rate

ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                 ADC_CTRLB_RESSEL_10BIT;       //Set ADC resolution to 10 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync
//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1

analogReadResolution(10);
analogWriteResolution(10);

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

//        timeIn = micros();

sample = analogRead(A1);
analogWrite(A0, sample);

//        timeIn = micros() - timeIn;
//        Serial.println(timeIn);  //print sample rate time in microseconds

} //End of Loop

//``````````````````````````````````````````````````````````````````
```

ADC to DAC0 - Modulation

Now we can insert some digital signal processing between the ADC and DAC operations. The next script multiplies the input signal with a calculated triangle waveform to create a ring modulation output. Slider values are used to adjust the modulation frequency and depth.

Note that the ADC sample values are unipolar. Any signal processing calculations are best done on bipolar signals. The ADC values are converted to bipolar with the "sample minus midPoint" calculation. The signal now swings between plus and minus values while sitting on zero volts. The multiply is then performed with a bipolar triangle wave.

Before being sent to the DAC, the modulated signal is turned back into a unipolar signal.

```
/*

                ADC1 to DAC0 with triangle modulation

Send output of ADC1 directly to DAC0, one sample each time through the main loop
Sampling frequency determined by loop time.

When Pushbutton1 is down, Slider1 changes frequency of modulation
When Pushbutton2 is down, Slider2 changes depth of modulation (and also frequency)

Thanks to "Arduino Music and Audio Projects" by Mike Cook
*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//
// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right


int slider1 = 0;
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch


boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;


#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

int sample;
float mod = 0.5;
float increment = 0.0035 ;
float depthHI = 0.80;
float depthLO = 0.02;
int sampleIn, sampleOut;
int midPoint = 516;
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  delay(1000);

 //  Serial.begin(9600);  //turn on to see mod Waveform

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV512 |  //Divde 48Khz GCLK by 512 for ADC
                 ADC_CTRLB_RESSEL_10BIT;       //Set ADC resolution to 10 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1

analogReadResolution(10);
analogWriteResolution(10);

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                     Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

 if (digitalRead(SWITCH1)){
 float incr = (2.0 + analogRead(SLIDER1))/10000.0; //mod frequency
 if (increment < 0){increment = -incr;} else{ increment = incr;}
 }

 if (digitalRead(SWITCH2)){
 depthLO = analogRead(SLIDER2)/2030.0;
 depthHI = 1.0 - depthLO;
 }

//make triangle wave for modulation. Ramp up, Ramp down
  mod += increment;
  if(mod > depthHI || mod < depthLO){
    increment = -increment;
    mod += increment;
  }
```

```
 //Serial.println(mod, 4); //use Tools/Serial Plotter, turn on Serial in Setup

sampleIn = (analogRead(A1)) - midPoint; //Read ADC input
sampleOut = (float)sampleIn * mod;      // Modulation multiply
analogWrite(A0, sampleOut + midPoint); // Write to DAC0 output

// toggle PA22 (LED2) each sample to check sample rate
// digitalWrite(LED2, !digitalRead(LED2));

} //End of Loop

//``````````````````````````````````````````````````````````
```

ADC to DAC0 - Delay Line

The next script creates a delay. The input samples are loaded into a circular array. "Circular" just means that the pointers into the array are reset to location zero after incrementing past the top array element. There are two pointers, an input pointer that points to the newest sample loaded, and the output pointer that points to the oldest sample loaded. The ADC fills the array with samples at the input pointer location incrementing the pointer after each load. The DAC is fed with samples addressed by the output pointer, incrementing the pointer after each load. At each sample time one new sample is loaded and one old sample is removed from the array. The delay time in seconds between the input signal and the output signal is then the array size divided by the sampling rate.

Feedback is a very useful effect in digital delays where part of the delayed output is added to the input signal resulting in repeating delay signals. That could be done just as easily in the script but, in this project, an opamp mixer circuit does it for you with a pot controlling the amount of feedback.

Arduino Processors have two types of memory. Flash memory holds the program and SRAM holds all the variables used in the program, which includes our delay array. The MKR_Zero has 32k of SRAM which is rather limited (Arduino Due has 96k of SRAM). The array elements are defined in the program as "short integers" which are 2 bytes each. Setting an array size of 10k will use up 20k of that 32k of SRAM. You need to leave space for other program variables, otherwise the program is likely to crash. A 10k array is playing it safe and it still gives us several seconds of delay time.

A software reverb is a simple variation of this delay script. Create several more output pointers at several places within the array and then add these extra samples of different delay times together to create a reverb effect.

The sampling rate of these signal processing programs is determined by the main loop

time. The function micros( ) can be used to calculate the loop time in microseconds and print it out using the IDE Print Monitor Tool. The sampling rate is then just the inverse of the loop time (1/loop-time).

```
    timeIn = micros();
 ---main loop processing---
timeIn = micros() - timeIn;
  Serial.println(timeIn);
```

This running printout of the loop times can also reveal how steady the loop time is and how it changes with any slider or switches functions within the processing program. Be aware that these loop-time lines are for testing only, they will adversely affect the sound output. During normal operation they should be commented out.

If for some reason the sampling rate fluctuates, the same lines could also be used lock in a steady lower sampling rate.

```
while( (micro( ) - timeIn) < 1/samplerate ) { }
```

The processing programs shown here have fairly good sample rates for audio. We found the simple ADC to DAC program to have a rate of 42kHz. The standard rate for audio CDs is only a bit higher than this at 44.1kHz. Adding any processing between the ADC and DAC, of course, will slow everything down. The delay program was found to have a sampling rate of 35.7kHz, and the next pitch program has a rate of 29.4kHz. These rates were also fairly steady. The loop times only fluctuated by one microsecond.

Lower sampling rates become a problem when there are frequencies in the audio input signal that are higher than one-half the sampling frequency (the Nyquist Rate). These higher frequencies will "foldover" and show up as "phantom" pitches at (freq - 1/2 sampling-rate). An easy way to hear this happen is with the following setup: Input a high frequency sinewave with the simple ADC to DAC script loaded. Add the following line to the main loop.

```
delayMicroseconds(analogRead(SLIDER1));
```

This will drastically increase the loop time, lowering the sample rate, when Slider1 is raised. Also try a square wave which has lots of higher partials.

```
/*

                ADC1 to DAC0 with buffer delay

Send samples from a live ADC input to an array buffer.
Output older samples from the array to DAC0.
```

```
Switch2 changes the delay by changing the output pointer's offset.
Switch1 drastically changes the delay time by adding delayMicroseconds(Slider1)
to the loop time lowering the sampling rate.

The MKRZero has only 32K bytes of SRAM memory used to store program variables.
The buffer array is limited to less than this size (10k x 2 bytes).  The array is circular,
an input pointer and an output pointer are incremented after every sample operation and
wrapped back to point to zero on reaching the top of the array.

The op amp circuit used for this project mixes the input signal with the DAC output
creating a feedback loop.


*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#define BUFFER_SIZE 10000  //10k x 2 bytes = 20k bytes, limited by the 32k SRAM size
short buffer1[BUFFER_SIZE]; // the size of a short is 2 bytes (16 bits)

// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

// Initialize the buffer contents to all zero
  for (int i=0; i<BUFFER_SIZE; i++){
    buffer1[i] = 0;
  }

   Serial.begin(9600);  //used only for printing samplerate times

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
```

```
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                 ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1

analogReadResolution(12);  //sample manipulations are done at higher resolution
analogWriteResolution(10); //highest resolution of MKRZero DAC

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

static signed int bufferIn = BUFFER_SIZE - 1; //input pointer set to array top
static unsigned int bufferOffset = BUFFER_SIZE - 1;   //output offset from input, sets maximum delay
static signed int bufferOut = BUFFER_SIZE - bufferOffset; //output pointer
static signed int sample;
int timeIn;

while(1){  // inner loop for higher sample rate

 timeIn = micros();

//lowers the sampling rate to increase delay, results in foldover frequencies with larger delays
  if (digitalRead(SWITCH2)){ delayMicroseconds(analogRead(SLIDER1) >> 3); }

 //set up two different delay times
  if (digitalRead(SWITCH1)){
    bufferOffset = BUFFER_SIZE - 1;  // maximum delay
  }
    else {bufferOffset = BUFFER_SIZE >> 4; // 1/4 maximum delay
  }

  sample = analogRead(A1) - 2048;  //make audio bipolar, plus and minus with 12 bit resolution
  //any multiply or divide operations must be done on bipolar signals with zero offsets

  //do any operations on the samples here

  buffer1[bufferIn] = sample;  //save sample in delay array

  sample = buffer1[bufferOut] + 2048;    //get output sample, make unipolar
  analogWrite(A0, sample >> 2);   //send sample to DAC0, 12 to 10 bit resolution change

// update the two circular array pointers, input and output
```

```
 bufferIn++;
  if (bufferIn >= BUFFER_SIZE){ bufferIn = 0; }

  bufferOut++;
  if (bufferOut >= BUFFER_SIZE) {
      bufferOut = bufferIn - bufferOffset ;
      if (bufferOut < 0){ bufferOut += BUFFER_SIZE;}
  }

//timeIn = micros() - timeIn;
//Serial.println(timeIn);  //print sample rate time in microseconds

} //end of while
} //End of loop


//``````````````````````````````````````````````````````````
```

ADC to DAC0 - Pitch

The final program in this section uses the delay construct to change the output pitch. The pitch can be lowered an octave by outputting each sample twice to the DAC. The pitch can be raised an octave by only outputting every other sample in the sample stream.

Both operations are very effective in changing the pitch but rather noisey since they cause the output pointer to crash into the input pointer. In the original delay program the output and input pointers keep the same offset between themselves. They step in tandom, after each sample is loaded one sample is unloaded. The octave lower operation makes the movement of the output pointer too slow, and the octave higher operation makes the movement of the output point too fast. When the input and output pointers crash the output waveform breaks up because it must skip ahead or back in time to keep up with the input stream. How often this happens and when is set by the delay time so that adjusting the delay time can help somewhat in controlling how much noise is generated.

Thanks to "Arduino Music and Audio Projects" by Mike Cook for all the program ideas in this section.

```
/*

                ADC1 to DAC0 with delay and pitch change

Send samples from a live ADC input to an array buffer.
Output older samples from the array to DAC0.
```

```
Switch1 sets output pitch an octave lower. Switch2 set output pitch an octave higher.
Switch3 reads Slider3 with a delay value to lower sampling rate.
Switch4 reads Slider1 with a buffer size to help clean up the pitch change.

Thanks to "Arduino Music and Audio Projects" by Mike Cook
*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#define BUFFER_SIZE 10000  //10k x 2 bytes = 20k bytes, limited by the 32k SRAM size
short buffer1[BUFFER_SIZE]; // the size of a short is 2 bytes (16 bits)

// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

// Initialize the buffer contents to all zero
  for (int i=0; i<BUFFER_SIZE; i++){
    buffer1[i] = 0;
  }

   Serial.begin(9600);  //used only for printing samplerate times

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
```

```
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                 ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1

analogReadResolution(12);  //sample manipulations are done at higher resolution
analogWriteResolution(10); //highest resolution of MKRZero DAC

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                     Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

static signed int bufferIn = BUFFER_SIZE - 1; //input pointer set to array top
static unsigned int bufferOffset = BUFFER_SIZE - 1;   //output offset from input, sets maximum delay
static signed int bufferOut = BUFFER_SIZE - bufferOffset; //output pointer
static unsigned int buffersize = BUFFER_SIZE - 1;
static signed int sample;
static signed int delayMicros = 1;
long timeIn;



while(1){  // inner loop for higher sample rate

 timeIn = micros();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Two ways to change the delay time ~~~~~~~~~~~~~~~~~~~~~~~~~~

//lowers the sampling rate to increase delay, results in foldover frequencies with larger delays

  if (digitalRead(SWITCH3)){ delayMicros = analogRead(SLIDER3) >> 3; }
  delayMicroseconds(delayMicros);

//set up different buffer sizes less than max of BUFFER_SIZE

 if (digitalRead(SWITCH4)){
  buffersize =  analogRead(SLIDER1) << 1;
  bufferOffset = buffersize - 1;
  }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


  sample = analogRead(A1) - 2048;  //make audio bipolar, plus and minus with 12 bit resolution
  //any multiply or divide operations must be done on bipolar signals with zero offsets

  //do any operations on the samples here

  buffer1[bufferIn] = sample;  //save sample in delay array

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Frequency Shift an Octave Lower ~~~~~~~~~~~~~~~~~~~~~~~~~~~

  if(digitalRead(SWITCH1)){  //save sample in delay array a second time for octave lower output
            bufferIn++;
            if (bufferIn >= buffersize){ bufferIn = 0; }
            buffer1[bufferIn] = sample;
  }  //End of SWITCH
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


  sample = buffer1[bufferOut] + 2048;    //get output sample, make unipolar
  analogWrite(A0, sample >> 2);   //send sample to DAC0, 12 to 10 bit resolution change

// update the two circular array pointers, input and output

 bufferIn++;
  if (bufferIn >= buffersize){
    bufferOut = bufferIn - bufferOffset ;  //line needed when randomly changing bufferOffset
    bufferIn = 0;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Frequency Shift an Octave Higher ~~~~~~~~~~~~~~~~~~~~~~~~~~~

  if (digitalRead(SWITCH2)){ bufferOut += 2; } //bufferOut reads every other sample for octave higher
  else { bufferOut ++ ; }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (bufferOut >= (buffersize)) {
      bufferOut = bufferIn - bufferOffset ;
      if (bufferOut <= 0){ bufferOut += buffersize;}
  }


//timeIn = micros() - timeIn;
//Serial.println(timeIn);  //print sample rate time in microseconds

} //end of while
} //End of loop


//``````````````````````````````````````````````````````````
```

Upgrading to the UDA1334 DAC

It is fairly easy to upgrade the MKR_Zero's 10-bit wide DAC with an external 16-bit stereo DAC board that uses the I2S serial interface and operates at high fidelity audio sample rates like the standard 44.1Khz. Adafruit sells the UDA1334 DAC for just $7. This board is supported by three I2S libraries for the Arduino IDE: The Arduino I2S.h library, the Arduino AudioSound.h library, and the Adafruit_ZeroI2S.h library.

This DAC excels in producing hi-fidelity stereo audio from the MKR_Zero's SD card reader (using the AudioSound library), or from sketch built waveforms. The next two

sketches play back a sinewave, one using the Arduino A2S library and the other using the Adafruit_ZeroI2S library. Both libraries initialize the UDA1334 DAC with a user defined sampling rate and a bit width. The Adafruit library seems to only work with a bit width of 32.

```
//
// Connect an I2S DAC or amp (like the UDA1334A) to the Arduino Zero
// and play back simple sine.  Using I2S.h library.

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "I2S.h"

#define AMPLITUDE     ((1<<15)-1)  //32, 24, 16, 8
#define WAV_SIZE      1024
const int sampleRate = 32000; //sample rate for I2S.h

// Define the frequency of music notes
#define C4_HZ      261.63
#define D4_HZ      293.66

// Store basic waveform in memory.
short sine[WAV_SIZE]     = {0};

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void generateSine(short amplitude, short* buffer, short length) {
  for (int i=0; i<length; ++i) {
    buffer[i] = short(float(amplitude)*sin(2.0*PI*(1.0/length)*i));
  }
} //End generateSine
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  // Configure serial port.

//              Serial.begin(500000);
//              while (!Serial) { delay(10); }  //waits for you to open Serial Monitor
//              Serial.println("Zero I2S Audio Tone Generator");

   // start I2S at the sample rate with 16-bit wide sample
  if (!I2S.begin(I2S_PHILIPS_MODE, sampleRate, 16)) {
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }

  // Generate waveforms.
  generateSine(AMPLITUDE, sine, WAV_SIZE);

}   //End setup

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  uint16_t pos = 0;  //integer position in wave table
  float posx;        //float calculated position in wave table
  short sample = 0;  //samples from wave table
```

```
  // set output frequency by accessing wavetable at different intervals or "delta"
  float delta = (C4_HZ * WAV_SIZE)/float(sampleRate); //wavetable sample interval

  int timeIn; //for finding sample rate

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

while(1){

//                                timeIn = micros(); //for printing sample rate time

      posx += delta;
      if (posx > WAV_SIZE){ posx = posx - WAV_SIZE; }
      pos = uint32_t(posx);

      sample = sine[pos];
//                        Serial.println(sample); //Use Serial Plotter to see wave
        I2S.write(sample);
        I2S.write(sample);

//                        timeIn = micros() - timeIn;
//                        Serial.println(timeIn);  //print sample rate time in microseco

  }   //End while
}   //End loop

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// Arduino Zero / Feather M0 I2S audio tone generation example.
// Author: Tony DiCola
//
// Connect an I2S DAC or amp (like the UDA1334A) to the Arduino Zero
// and play back simple sine.  Using Adafruit_ZeroI2S library.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "Adafruit_ZeroI2S.h"

#define SAMPLERATE_HZ 44100// 22.7usec --> 44100, works also at 48000, 88200 (not 96000)
#define AMPLITUDE     ((1<<30)-1)  //32, 24, 16, 8
#define WAV_SIZE      512

// Define the frequency of music notes
#define C4_HZ      261.63
#define D4_HZ      293.66

// Store basic waveform in memory.
   int32_t sine[WAV_SIZE]     = {0};

// Create I2S audio transmitter object.
Adafruit_ZeroI2S i2s;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void generateSine(int32_t amplitude, int32_t* buffer, uint16_t length) {
  for (int i=0; i<length; ++i) {
    buffer[i] = int32_t(float(amplitude)*sin(2.0*PI*(1.0/length)*i));
  }
} //End generateSine
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  // Configure serial port.

//              Serial.begin(500000);
//              while (!Serial) { delay(10); }  //waits for you to open Serial Monitor
//              Serial.println("Zero I2S Audio Tone Generator");

  // Initialize the I2S transmitter.
    if (!i2s.begin(I2S_32_BIT, SAMPLERATE_HZ)) {  //only works with 32_BIT ??
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }
  i2s.enableTx();

  // Generate waveforms.
  generateSine(AMPLITUDE, sine, WAV_SIZE);

}   //End setup

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  uint16_t pos = 0;  //integer position in wave table
  float posx; //float calculated position in wave table
  int32_t sample = 0;

  // set output frequency by accessing wavetable at different intervals or "delta"
  float delta = (C4_HZ * WAV_SIZE)/float(SAMPLERATE_HZ); //wavetable sample interval

  int timeIn; //for finding sample rate

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

while(1){

//                    timeIn = micros(); //for printing sample rate time

      posx += delta;
      if (posx > WAV_SIZE){ posx = posx - WAV_SIZE; }
      pos = uint32_t(posx);

      sample = sine[pos];
//                    Serial.println(sample); //Use Serial Plotter to see wave
      i2s.write(sample, sample);

//                    timeIn = micros() - timeIn;
//                    Serial.println(timeIn);  //print sample rate time in microsec

  }   //End while
}   //End loop

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Signal Processing - ADC to UDA1334

External hi-fidelity ADC breakout boards like the UDA1334 DAC are hard, if not impossible, to find. Paring the MKR_Zero's ADC with the UDA1334 DAC requires some compromises. The ADC's 12-bit output must be multiplied by 16 to bring it up to the DAC's 16-bit levels. This raises the noise floor of the ADC by quite a bit. I found that raising the SAMPLEN value in the ADC speedup routines from 1 to 16 helps quiet the noise a bit but that also necessitates a much lower DAC sampling rate.

The next script uses the Arduino I2S library to simply send the output from the MKR_Zero ADC to the UDA1334 DAC. With zero audio input you can experiment with DAC sample rates and ADC PRESCALER_DIVxx and SAMPLEN() values to find the lowest noise floor.

```
// Connect an I2S DAC or amp (like the UDA1334A) to the Arduino Zero
// and play back from ADC A1.  Using I2S.h library.

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "I2S.h"
const int sampleRate = 16000; //sample rate for I2S.h

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

// Configure serial port.
//                        Serial.begin(500000);
 //                       while (!Serial) { delay(10); } //waits for you to open Serial Monitor
 //                       Serial.println("Zero I2S Audio Tone Generator");

// start I2S at the sample rate with 16-bits per sample
  if (!I2S.begin(I2S_PHILIPS_MODE, sampleRate, 16)) {
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }

// set up speed of ADCs

  ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                   ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
  while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
  ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(16);  //Set Sampling Time Length to 16

}   //End setup

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  signed int sample = 0;
  int timeIn; //for finding sample rate
```

```
while(1){
//                       timeIn = micros(); //for printing sample rate time

  sample = (analogRead(A1) - 2048) << 4;  //make bipolar, bring amp up from 12-bit to 16-bit

//                       Serial.println(sample);
  I2S.write(sample); // write twice for left and right outputs
  I2S.write(sample);
//                        timeIn = micros() - timeIn;
//                        Serial.println(timeIn);  //print sample rate time in microseconds

  }  //End while
}   //End loop

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

ADC to UDA1334 - Delay Line & Pitch

The previous Delay and Pitch Change scripts used with the DAC0 are easily revised to used the UDA1334 DAC instead as shown here. SAMPLEN() was changed from 1 to 16 to lower the noise floor, and the sampling rate for the UDA1334 was lowered to 10,000Hz to accomodate the longer ADC conversion time and the delay processing.

```
/*

                ADC1 to DAC0 with buffer delay

Send samples from a live ADC input to an array buffer.
Output older samples from the array to DAC1334 using the I2S library.

If Switch1 is up the maximum delay is set up.
If Switch1 is down the delay is set by Slider1.

The MKRZero has only 32K bytes of SRAM memory used to store program variables.
The buffer array is limited to less than this size (12k x 2 bytes).  The array is circular,
an input pointer and an output pointer are incremented after every sample operation and
wrapped back to point to zero on reaching the top of the array.

Thanks to "Arduino Music and Audio Projects" by Mike Cook
*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "I2S.h"
const int sampleRate = 10000; //sample rate for I2S.h

#define BUFFER_SIZE 8200  //10k x 2 bytes = 20k bytes, limited by the 32k SRAM size
short buffer1[BUFFER_SIZE]; // the size of a short is 2 bytes (16 bits)

// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
```

```
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

// Initialize the buffer contents to all zero
  for (int i=0; i<BUFFER_SIZE; i++){
    buffer1[i] = 0;
  }

//   Serial.begin(9600);  //used only for printing samplerate times

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

// Configure serial port.
//                       Serial.begin(500000);
 //                      while (!Serial) { delay(10); } //waits for you to open Serial Monitor
 //                      Serial.println("Zero I2S Audio Tone Generator");

// start I2S at the sample rate with 16-bits per sample
  if (!I2S.begin(I2S_PHILIPS_MODE, sampleRate, 16)) {
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }
```

```
// set up speed of ADCs

ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                 ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(16);  //Set Sampling Time Length to 16

analogReadResolution(12);  //sample manipulations are done at higher resolution
analogWriteResolution(10); //highest resolution of MKRZero DAC

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

static signed int bufferIn = BUFFER_SIZE - 1; //input pointer set to array top
static unsigned int bufferOffset = BUFFER_SIZE - 1;   //output offset from input, sets maximum delay
static signed int bufferOut = BUFFER_SIZE - bufferOffset; //output pointer
static signed int sample;
int timeIn;

while(1){  // inner loop for higher sample rate

 timeIn = micros();

 //set up different delay times
  if (digitalRead(SWITCH1)){
    bufferOffset = BUFFER_SIZE - (analogRead(SLIDER1) << 1); // variable delay
  }
    else {
    bufferOffset = BUFFER_SIZE - 1;  // maximum delay
  }

  sample = analogRead(A1) - 2048;  //make audio bipolar, plus and minus with 12 bit resolution
  //any multiply or divide operations must be done on bipolar signals with zero offsets
  //do any operations on the samples here

  buffer1[bufferIn] = sample;  //save sample in delay array

  sample = sample + buffer1[bufferOut] << 3 ; //add original plus delayed

//                       Serial.println(sample);

  I2S.write(sample); // write twice for left and right outputs
  I2S.write(sample);


// update the two circular array pointers, input and output

 bufferIn++;
  if (bufferIn >= BUFFER_SIZE){ bufferIn = 0; }

  bufferOut++;
  if (bufferOut >= BUFFER_SIZE) {
      bufferOut = bufferIn - bufferOffset ;
      if (bufferOut < 0){ bufferOut += BUFFER_SIZE;}
  }
```

```
//timeIn = micros() - timeIn;
//Serial.println(timeIn);  //print sample rate time in microseconds

} //end of while
} //End of loop


//``````````````````````````````````````````````````````````


/*

                ADC1 to 1334DAC with delay and pitch change

Send samples from a live ADC input to an array buffer.
Output older samples from the array to DAC1334 using the I2S library.

Switch1 sets output pitch an octave lower. Switch2 set output pitch an octave higher.
Switch3 reads Slider3 with a delay value to lower sampling rate.
Switch4 reads Slider1 with a buffer size to help clean up the pitch change.

The op amp circuit used for this project can mix the input signal with the DAC output, and
it can also mix the DAC signal with the ADC input to create feedback
thus the software only needs to create a delay signal.2

Thanks to "Arduino Music and Audio Projects" by Mike Cook
*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "I2S.h"
const int sampleRate = 10000; //sample rate for I2S.h

#define BUFFER_SIZE 10000  //10k x 2 bytes = 20k bytes, limited by the 32k SRAM size
short buffer1[BUFFER_SIZE]; // the size of a short is 2 bytes (16 bits)

// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

//
//DIGIITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)
```

```
#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

// Initialize the buffer contents to all zero
  for (int i=0; i<BUFFER_SIZE; i++){
    buffer1[i] = 0;
  }

   Serial.begin(9600);  //used only for printing samplerate times

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

// Configure serial port.
//                      Serial.begin(500000);
 //                     while (!Serial) { delay(10); } //waits for you to open Serial Monitor
 //                     Serial.println("Zero I2S Audio Tone Generator");

// start I2S at the sample rate with 16-bits per sample
  if (!I2S.begin(I2S_PHILIPS_MODE, sampleRate, 16)) {
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }

// set up speed of ADCs

ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                 ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1

analogReadResolution(12);  //sample manipulations are done at higher resolution
analogWriteResolution(10); //highest resolution of MKRZero DAC

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {
```

```
static signed int bufferIn = BUFFER_SIZE - 1; //input pointer set to array top
static unsigned int bufferOffset = BUFFER_SIZE - 1;   //output offset from input, sets maximum delay
static signed int bufferOut = BUFFER_SIZE - bufferOffset; //output pointer
static unsigned int buffersize = BUFFER_SIZE - 1;
static signed int sample;
static signed int delayMicros = 1;
int timeIn;



while(1){  // inner loop for higher sample rate

// timeIn = micros();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~change the delay time ~~~~~~~~~~~~~~~~~~~~~~~~~~

//set up different buffer sizes less than max of BUFFER_SIZE

 if (digitalRead(SWITCH4)){
  buffersize =  analogRead(SLIDER1) << 1;
  bufferOffset = buffersize - 1;
  }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


  sample = analogRead(A1) - 2048;  //make audio bipolar, plus and minus with 12 bit resolution
  //any multiply or divide operations must be done on bipolar signals with zero offsets

  //do any operations on the samples here

  buffer1[bufferIn] = sample;  //save sample in delay array

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Frequency Shift an Octave Lower ~~~~~~~~~~~~~~~~~~~~~~~~~~

  if(digitalRead(SWITCH1)){  //save sample in delay array a second time for octave lower output
            bufferIn++;
            if (bufferIn >= buffersize){ bufferIn = 0; }
            buffer1[bufferIn] = sample;
  }  //End of SWITCH
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


  sample = buffer1[bufferOut] << 4;    //get output sample. 12-bit to 16-bit

  //                       Serial.println(sample);
  I2S.write(sample); // write twice for left and right outputs
  I2S.write(sample);


// update the two circular array pointers, input and output

 bufferIn++;
  if (bufferIn >= buffersize){
    bufferIn = 0;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Frequency Shift an Octave Higher ~~~~~~~~~~~~~~~~~~~~~~~~~~

  if (digitalRead(SWITCH2)){ bufferOut += 2; } //bufferOut reads every other sample for octave higher
  else { bufferOut ++ ; }
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (bufferOut >= (buffersize)) {
      bufferOut = bufferIn - bufferOffset ;
      if (bufferOut <= 0){ bufferOut += buffersize;}
  }


//timeIn = micros() - timeIn;
//Serial.println(timeIn);  //print sample rate time in microseconds

} //end of while
} //End of loop


//``````````````````````````````````````````````````````````
```

ADC to UDA1334 - Fuzz Distortion

One classic signal processing effect is the Fuzz Box which distorts a signal by clipping its peaks. The next sketch takes a value from Slider1 and clips the tops and bottoms of the ADC input signal. Any samples above that Slider1 value, or below negative that value are ignored and replaced by the set Slider1 value. This is called "Hard Clipping".

A "Soft Clip", with less distortion, can be achieved by replacing the original clipped samples with the Slider1 value plus a fraction of the sample value so that the peaks of the ADC input signal are "squished" instead of clipped. The "Squish" or fraction value is derived from Slider2. This is also known as a Compressor effect.

These sample calculations for the "fuzz" take some time to perform. When the sample rate set for the UDA3114 DAC is too high to accomodate these calculation times, the sound output will have high pitched tones or pops and clicks. When this happens, incrementally lower the sample rate (found at the start of the sketch) until these sound artifacts disappear. 8kHz was found to work for this program.

To judge the worth of this sample rate consider the Nyquist Frequency at half the sampling rate. This is the highest frequency the conversion process can pass. 4kHz is pretty bad considering the range of human hearing is 20Hz to 20,000Hz. For more realistic markers, my aging ears can just barely hear 12,000Hz, and the top key of a grand piano is 4186Hz.

The Arduino IDE has a Plotter Function under the Tool Menu that will allow you to see the output waveform. Just input a low frequency sinewave and watch how the program clips or squishes the peaks of the waveform as Sliders 1 and 2 are manipulated. To set this up, uncomment the "Serial.begin( )" line under "Setup" and the

"Serial.println(sample)" line in the Main Loop, and reload the sketch. The sound output will be seriously degraded by the extra print function, but you will be able to watch the program do its work.

```
/*

                ADC1 to DAC0 with hard and soft clipping

Mathmatically distort samples from a live ADC input before
sending out to DAC1334 using the Adafruit I2S Library.

If switch1 is down, hard clip the signal at levels set by slider1.
If switch2 is down, soft clip the signal, slider2 set level the clipping starts,
                    slider4 sets amount of clip.

*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "I2S.h"  //library for 1334 DAC board
#define SAMPLE_RATE 8000

// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

int slider1 = 0;  // ADC 12 bit  zero to 4096
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {
```

```
//   Serial.begin(500000);  //used only for testing

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);

// start I2S at the sample rate with 16-bits per sample
  if (!I2S.begin(I2S_PHILIPS_MODE, SAMPLE_RATE, 16)) {
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }

// set up speed of ADCs

ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                 ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(16);  //Set Sampling Time Length to 16

analogReadResolution(12);  //sample manipulations are done at higher resolution
analogWriteResolution(10); //highest resolution of MKRZero DAC


}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                     Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

//static signed int sample;
int16_t sample ;
static int squish = 7;  //squish factor
int timeIn;


while(1){  // inner loop for higher sample rate

  sample = analogRead(A1) - 2048;  //make audio bipolar, +/- 2048, with 12 bit resolution
  //any multiply or divide operations must be done on bipolar signals with zero offsets



  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Hard Fuzz ~~~~~~~~~~~~~~~~~~~~~~~~~~

  if(digitalRead(SWITCH1)){  //simple fuzz, clip off top and bottom of waveform
```

```
         slider1 = (analogRead(SLIDER1) >> 2) + 20;   // slider, 20 to 1024
           if (sample > slider1){ sample = slider1; } //clip off top of waveform
           else if (sample < -slider1){ sample = -slider1; }  //chop off bottom of waveform
           // else middle range of waveform unaltered

  }  //End of SWITCH
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Soft Fuzz ~~~~~~~~~~~~~~~~~~~~~~~~~~

  if(digitalRead(SWITCH2)){  //softer clip, squish top and bottom of waveform

    squish =  (analogRead(SLIDER4) >> 6) + 5;
    slider2 = (analogRead(SLIDER2) >> 2) + 20;   // slider, 20 to 1024

           if (sample > slider2){
            sample = slider2 + ((sample - slider2) / squish); //squish top of waveform
            }
           else if (sample < -slider2){
            sample = -slider2 - ((-slider2 - sample) / squish); //squish bottom of waveform
            }
           // else middle range of waveform unaltered

  }  //End of SWITCH
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 sample = sample << 4; // 12-bit to 16-bit

 I2S.write(sample);  // write twice for left and right outputs
 I2S.write(sample);

// Serial.println(sample);  //use Serial Plotter to see output waveform, use low freq inputs

} //end of while
} //End of loop


//````````````````````````````````````````````````````````````
```

ADC to UDA1334 - Transform Function

The MKR_Zero ADC is 12 bits wide which means it can have samples values that range from zero to 4095. Those samples correspond to voltages from zero volts to a maximum of 3.3 volts. This is a unipolar signal, or a signal that only has positive values and voltages. Most audio signals, however, are bipolar. A bipolar signal sits on a voltage bias of zero volts and swings between both positive and negative voltages. To make the ADC digital samples bipolar we subtract one half of 4096 from the ADC sample values, resulting in a new bipolar range of samples from negative 2048 to positive 2048. The UDA3114 DAC accepts only bipolar digital inputs. However, the UDA3114 DAC has a

minimum range of 16 bits. The ADC signal must be multiplied by 16 to get a 16-bit range of negative 32,768 to positive 32,768. If the ADC is not multiplied by 16, the DAC output will have much less volume. On the other hand, multiplying by 16 also raises the inherent noise floor of the ADC by a factor of 16.

Any manipulation of the digital signal can be performed on the 12-bit ADC numbers, before they are raised in amplitude by a factor of 16. In fact, for most cases, we can further narrow our range to only the positive values, zero to 2048. In the next script a function will be devised for the zero to +2048 range of input sample values. The same calculations can then be applied to the negative range of zero to -2048 samples values.

An equation will be derived from the straight line "transform" function shown on the graph in the figure below. The x-axis is ADC input sample values from zero to 2047. The y-axis is DAC output sample values from zero to 2047. The function shown is an improved version of the "Fuzz" calculation in the last sketch.

For low sample values from zero to a "threshold" the straight line function is very simple: x = y, or input sample value is equal to the output sample value - no change from input to output. A threshold value is set by Slider 1 ("a" on the graph) and is the start of a second straight line function. The end of this second straight line, at sample input = 2047, is set by Slider 2. Let's explore some specific possibilities for this second straight line function.

If Slider 2 sets a value of y=2047 at x=2047, then this second straight line is just a continuation of the slope of the first line setting x = y for all possible input sample values. The rather boring result is that there is no change from the input to the output.

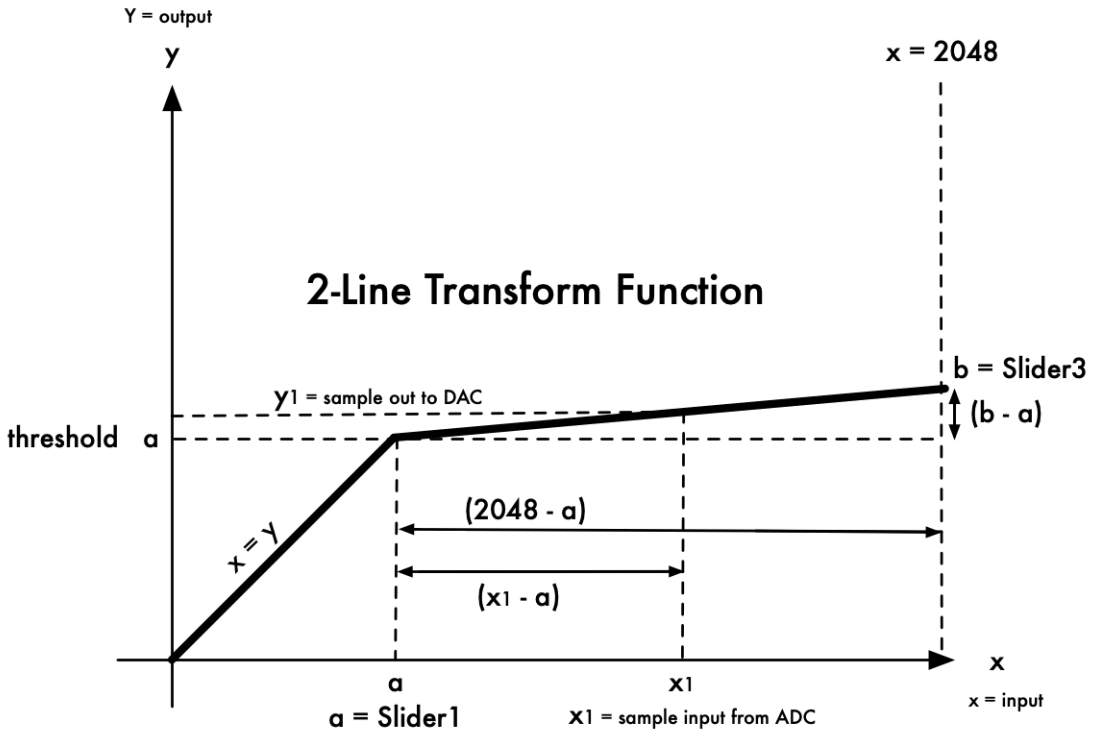
If Slider 2 sets a value that is equal to Slider 1, the threshold, then the second line lies flat at that same threshold value for all input values from threshold to 2047. Any input values coming in above the threshold will be ignored and set to equal to that one threshold value resulting in a clipped waveform, which is our hard clip Fuzz effect from the last sketch.

If Slider 2 sets an end value between the Slider 1 value and 2047. The output signal peaks above the threshold will be a "squished" version of the input. The amount of squish can vary between the two examples described above, from zero squish (x = y, when Slider2 = 2047) to complete clipping (x = threshold, when Slider2 = Slider1). This becomes a type of audio "compressor".

A somewhat unusual case happens when Slider 2 sets an end point below the Slider 1 threshold value resulting in a downward sloping line. In this case the input signal peaks are compressed, or "squished" as described above, but they are also inverted from the

original peaks - an upward hump now becomes a downward hump.

I encourage you to input a low frequency sinewave, uncomment the script's print functions, turn on the Arduino Plotter function, and watch the results described above actually happening on the input waveform.



$$y1 = a + \frac{(x1 - a)(b - a)}{(2048 - a)}$$

In the sketch below the calculations shown in the graph are implemented. For input values below the Slider 1 threshold the output value is unchanged. For input values above the threshold, the equation shown is used to calculate the output sample value. The same calculations are made separately for the negative going parts of the input signal.

These calculations are pretty heavy duty and take up quite a lot of processing time especially since they are floating point calculations. In addition, the script employs a third Slider to pan between the processed signal and the unprocessed signal, which is another floating point calculation. This required a lowering of the sampling rate to10kHz for the UDA3114 DAC in addition to lowering the ADC register SAMPLEN

value back to 1.

```
//                     Transform Function
// Connect an I2S DAC or amp (like the UDA1334A) to the Arduino Zero
// and play back from ADC A1 with signal processing using the I2S.h library.
/*
An input/output transfer function is set up for both the positive and negative going parts of the
signal.

Each possible input sample value, -2048 to 0 to +2048, from the 12-bit A1 ADC  is translated into
a different output sample value.  This can be accomplished with a 4096 cell table, or calculated with a
function, as done here.

The input/output function is made up of two straight line functions.The first is simple, input equals
output -- no change to input values zero to a value determined by Slider1.  The second straight
line starts at the Slider1 value and ends at a Slider2 value and results in a distortion
of the tops and bottoms of the input waveform.

These straight line functions are mirrored in the negative input values zero to -2048.

Slider 1 and 3 values are applied only when Switch1 is down.

Slider2 pans between the input waveform and the distorted output waveform.
The SampleRate can be raised to 16000 if this slow pan calculation is commented out.

*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "I2S.h"
const int sampleRate = 10000; //sample rate for I2S.h

signed int a = 0;
signed int b = 0;
signed int sample = 0;
signed int samplex = 0;
float pan = 0;

// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

//
//DIGITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
```

```
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

// Configure serial port.
//                        Serial.begin(500000);

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, HIGH);

pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);


// start I2S at the sample rate with 16-bits per sample
  if (!I2S.begin(I2S_PHILIPS_MODE, sampleRate, 16)) {
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }

// set up speed of ADCs

  ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
  while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
  ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1

}   //End setup

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  samplex = analogRead(A1) - 2048;
  sample = samplex;  //keep samplex as the original input signal


  if (digitalRead(SWITCH1)){
  a = analogRead(SLIDER1)  >> 2;
  b = analogRead(SLIDER3) >> 2 ;
  }

  if (sample >= 0) {  //for positive samples

      if (sample > a){                    // if sample < a, leave it unchanged
        sample = a + (  ((b - a) * (sample - a)) / (2048 - a)  ) ;  //transfer function
```

```
      }


  } //End positive samples


  else {  //for negative samples

       sample = -sample;
       if (sample > a){                    // if sample < a, leave it unchanged
        sample = a + (  ((b - a) * (sample - a)) / (2048 - a)  ) ;  //transfer function
      }
      sample = -sample;

  }  //End negative samples


  pan = analogRead(SLIDER2) ;
  pan = pan / 4095;  //pan is a float between 0 and 1

sample = (pan * sample) + ((1 - pan) * samplex);  //pan between input and distorted version

//  Connect a sinewave to the A1 input and use the Serial Plotter to see the output waveform
//                           Serial.println(sample);

 sample = sample << 4;  //12-bit to 16-bit size


  I2S.write(sample);  // write twice for left and right outputs
  I2S.write(sample);

}   //End loop

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

ADC to UDA1334 - Transform Array

The simplicity and versatility of using an array to transform an audio input signal into something completely different is hard to pass up. For each possible ADC sample input value from 0 to 2047 an output DAC sample value can be specified. This is easily accomplished in software by a 2048 element array. The address of the array element, from 0 to 2047, comes from sample input value and the addressed array element value specifies the output value which can be anything from 0 to 2047.

The array function is built for the positive half of the ADC bipolar signal. Only a few extra lines of code are needed to apply the same function to the negative half of the input signal.

Once the array is built, the input/output program is basically just three lines of code

```
sample = analogRead(A1) - 2048;  // read from ADC and make bipolar
     sample = tfunction[sample];  // apply transfer function
             I2S.write(sample);  // write to the DAC
```

One example of transform functions was illustrated in the input/output graph from the previous sketch. The question now becomes how to draw any input/output graph you desire and then get that function into a 2048 element array.

One possibility would be to couple the MKR_Zero with a higher level program such as Max/MSP or Processing running on another computer, where you could use its graphic capabilities to actually draw a function using a mouse or graphic pad, translate the drawing into an array, and send it to the MKR_Zero over a serial USB line.

The sketch below uses a simpler method, setting up a routine that records the motion of a slider to create the function array. The routine uses a switch_case control structure to set up a series of events that are initiated by pressing one of the box pushbuttons. First, LED1 blinks three times at a rate of about one blink per second. The user will use this as a countdown before the actual recording. Next, LED2 will turn on for a second count of three LED1 blinks during which time the position of Slider3 is recorded 2048 times over about three seconds. At the end of this count both LEDs turn off, the recording is finished, and the newly created array is now used in the main loop.

The user is encouraged to read up on audio effects such as fuzz distortion, compression, sustain, limiter, expander, gate. Find explanations that include input/output graphs. With some practice with the Slider Record routine you should be able to recreate any of these effects and many more.

The sketch below also uses a slider to pan between the processed and unprocessed signal. That may or may not make up for not having Attack and Release functions, usually included with the above effects.

```
/*

                ADC1 to DAC0 with with Transform Array

Distort live ADC input with a transform array before sending out to DAC1334 using I2S library.
The input signal is treated as bipolar with values -2048, through zero, to +2048
The same transfer array is applied to both positive and negative swings of the input signal.

A 2048 element array is created.  Each possible ADC input value from 0 to 2048 acts as the array index.
The array elements act as alternative waveform values for each ADC waveform input value.

The array transform function is created manually by moving Slider3 through a count of 3 seconds.
Led1 gives a countdown of 3 one second blips . Led2 then turns on for a second count of 3 Led1 blips
during which 4096 values are recorded from the manual movement of Slider3.
```

```
Pushbutton3 is used to start the function record countdown process.

Then Output DAC Sample = array[ ADC Input Sample ]

Slider1 pans between the input waveform and the distorted output waveform.
The SampleRate can be raised to 16000 if this slow pan calculation is commented out.

Switches 1 & 2 turn on the Serial.print function for testing with the Serial Plotter

*/
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "I2S.h"  //library for 1334 DAC board
#define sampleRate 10000
#define TDELAY 1300


// ANALOG INPUTS
//
#define SLIDER1 A5  //top left
#define SLIDER2 A3  //bottom left
#define SLIDER3 A4  //top right
#define SLIDER4 A2  //bottom right

int slider1 = 0;  // ADC 12 bit  zero to 4096
int slider2 = 0;
int slider3 = 0;
int slider4 = 0;

//
//DIGITAL SWITCHES
//
#define SWITCH1 5  //top toggle switch
#define SWITCH2 4  //bottom toggle switch
#define SWITCH3 6  //right pushbutton switch
#define SWITCH4 7  //left pushbuttun switch

boolean switch1 = 0;
boolean switch2 = 0;
boolean switch3 = 0;
boolean switch4 = 0;

#define VOICEPIN_A 8   //pot 4 on box right side
#define VOICEPIN_B 9   //pot 5 on box right side
#define VOICEPIN_C 10  //Modulating Voice (white knob on box left side, and switch up)

#define LED1 1  //top toggle switch's LED
#define LED2 0  //bottom toggle switch's LED

unsigned int t = 0;
unsigned int x = 0;
boolean countdown = 0;
boolean countrecord = 0;
boolean drawfunction = 0;
int blink3[6] = {171, 683, 853, 1365, 1536, 2047};
int tcase = 7;
int tfunction[2048];

signed int sample = 0;
signed int samplex = 0;
float pan = 0;
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       SETUP()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

   Serial.begin(500000);  //used only for testing

pinMode(LED1, OUTPUT);
digitalWrite(LED1, HIGH);
pinMode(LED2, OUTPUT);
digitalWrite(LED2, LOW);
delay(250);
digitalWrite(LED1, LOW);


pinMode(VOICEPIN_A, OUTPUT);
digitalWrite(VOICEPIN_A, LOW);
pinMode(VOICEPIN_B, OUTPUT);
digitalWrite(VOICEPIN_B, LOW);
pinMode(VOICEPIN_C, OUTPUT);
digitalWrite(VOICEPIN_C, LOW);

pinMode(SWITCH1, INPUT); //Switch inputs have external 10k pulldown resistor
pinMode(SWITCH2, INPUT);
pinMode(SWITCH3, INPUT);
pinMode(SWITCH4, INPUT);


// set up speed of ADCs
ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV32 |  //Divde 48Khz GCLK by 32 for ADC
                 ADC_CTRLB_RESSEL_12BIT;       //Set ADC resolution to 12 bits
while(ADC->STATUS.bit.SYNCBUSY);              // Wait for these changes to sync

//Sampling Time Length SAMPLEN (normally 63) allows time for ADC capacitor to charge
ADC->SAMPCTRL.reg = ADC_SAMPCTRL_SAMPLEN(1);  //Set Sampling Time Length to 1

analogReadResolution(12);  //sample manipulations are done at higher resolution
analogWriteResolution(10); //highest resolution of MKRZero DAC



// start I2S at the sample rate with 16-bits per sample
  if (!I2S.begin(I2S_PHILIPS_MODE, sampleRate, 16)) {
    Serial.println("Failed to initialize I2S");
    while (1); //do nothing if failed
  }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                       Main LOOP
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                      Creating Transform Curve
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 switch3 = digitalRead(SWITCH3);
```

```
 if (switch3 && !countdown){    //Start countdown and countrecord sequence
      countdown = 1;
      t = 0;
      tcase = 1;
      digitalWrite(LED1, HIGH);
 }

switch (tcase) {  //countdown and countrecord states

  case 1:   //count1 led on
    if( t > blink3[0] ){
      //Serial.print(tcase); Serial.print("\t");  Serial.println(t);
      digitalWrite(LED1, LOW);
      tcase = 2;
    }
    break;
  case 2:   //count1 led off
    if( t > blink3[1] ){
      //Serial.print(tcase); Serial.print("\t");  Serial.println(t);
      digitalWrite(LED1, HIGH);
      tcase = 3;
    }
    break;
  case 3:   //count2 led on
    if( t > blink3[2] ){
      //Serial.print(tcase); Serial.print("\t");  Serial.println(t);
      digitalWrite(LED1, LOW);
      tcase = 4;
    }
      break;
  case 4:   //count2 led off
    if( t > blink3[3] ){
      //Serial.print(tcase); Serial.print("\t");  Serial.println(t);
      digitalWrite(LED1, HIGH);
      tcase = 5;
    }
      break;
  case 5:   //count3 led on
    if( t > blink3[4] ){
      //Serial.print(tcase); Serial.print("\t");  Serial.println(t);
      digitalWrite(LED1, LOW);
      tcase = 6;
    }
     break;
  case 6:   //finish
    if( t > blink3[5] ){   //at end of countdown or countrecord, finish up
      //Serial.print(tcase); Serial.print("\t");  Serial.println(t);

      if (countdown){    //end the countdown and start the record
        countdown = 0;
        countrecord = 1;
        digitalWrite(LED1, HIGH);
        digitalWrite(LED2, HIGH);
        t = 0;
        tcase = 1;
     }

     else if (countrecord){    //end the record
        countrecord = 0;
        countdown = 0;
        digitalWrite(LED2, LOW);
        digitalWrite(LED1, LOW);
        tcase = 7;
     }
```

```
    }
     break;
  case 7:  //do nothing till another start countdown
     break;
  default:
     break;
}  //End of case


 if(countrecord){  //load Transfer function from Slider3 over a count of 3 led blips
  slider3 = analogRead(SLIDER3) ;
  tfunction[t] = slider3 >> 1 ;
 }

 if(countrecord || countdown) {   //increment index in tfunction[] and wait about a second
    ++t;
    delayMicroseconds(TDELAY);
 }


 switch1 = digitalRead(SWITCH1);  //use Serial Plotter to see transfer function in tfunction
 if(switch1){
    for (int i = 0; i <= 2047; i++) {
      Serial.println(tfunction[i]);
      Serial.println("Min:0, Max:2047");
      delay(1);
    }
  }  //End Switch1

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//             Sampling Transform
//  ___________________________________________________________________________


  samplex = analogRead(A1) - 2048;
  sample = samplex;  //keep samplex as the original input signal

  //for positive samples
  if (sample >= 0) {  sample = tfunction[sample];  }  //apply transfer function

  //for negative samples
  else {
       sample = -sample;
       sample = tfunction[sample];  //apply transfer function
       sample = -sample;
  }

  pan = analogRead(SLIDER1) ;
  pan = pan / 4095;  //pan is a float between 0 and 1

sample = (pan * sample) + ((1 - pan) * samplex);  //pan between input and distorted version

// Connect a sinewave to the A1 input and use the Serial Plotter to see the output waveform
                switch2 = digitalRead(SWITCH2);
                if (switch2){ Serial.println(sample); }

 sample = sample << 4;  //12-bit to 16-bit size


  I2S.write(sample);  // write twice for left and right outputs
  I2S.write(sample);

} //End of loop
```

//``

Closing Thoughts

The Arduino MKR_Zero is a big improvement in processing power over the original Arduino Uno. The clock speed is increased from 16MHz to 48MHz, Program Memory goes from 32kB to 256kB, Data Memory goes from 2kB to 32kB, and the Uno's ATmega328P 8-bit processor is replaced with a SAMD21 Cortex-M0+ 32-bit processor in the MKR_Zero. Any audio processing code could make good use of this extra processing power.

The MKR_Zero's main claim to fame in the audio realm is its ability to play back audio files from an included SD card through its new 10-bit DAC. An example sketch of this can be found at https://www.arduino.cc/en/Tutorial/SimpleAudioPlayerZero. For this sketch they describe building an 8-bit mono wav audio test file. That could probably be improved upon with a 10-bit file instead.

In a world where 16-bit stereo audio DACs are standard, a 10-bit DAC will not sound very high fidelity. To improve upon the sound output, the 10-bit internal DAC could be replaced with a better external DAC that uses the MKR_Zero's I2S interface and the Arduino AudioSound library (https://www.arduino.cc/en/Reference/ArduinoSound).

One example sketch from the Arduino AudioSound library is "WavePlayback" which plays back a signed 16 bit stereo wav file at 44100Hz stored on an SD card. That sketch claims to use a MAX08357I2S Amp Breakout board with an improved DAC and speaker amp. The same sketch should work with this project's Adafruit UDA3114 breakout board.

At 12-bits the ADCs on the MKR_Zero are also under-whelming for audio applications. In addition, the internal ADC registers must be hacked to bring the conversion speed anywhere close to useful audio conversion rates. ADC breakout boards with an I2S interface do not seem to be as available as the DAC breakout boards.

Though the processor improvements gained with the Arduino MKR_Zero don't quite

come up to the specs necessary for high fidelity audio processing, they at least make it possible, for the first time, to experiment with audio processing code as demonstrated in this paper.

The Next Step

As technology rolls onward there are already signal processing platform alternatives to the Arduino MKR_Zero.

If you need more speed and storage space consider using the ESP32 board. It has 520kB of Program Memory, 520kB of Data Memory, and runs at a speedy 80 to 240MHz. Sadly there are no improvements to the internal ADCs and DACs. The two internal DACs are only 8-bit, but the board has two I2S interfaces to use for better, external DAC/ADC boards.

The ESP32-A1S is a new chip version by AI-Thinker which includes an integrated AC101 Codec with stereo 24 bit ADCs and DACs running at fast sample rates and controlled through an I2S interface.

The Chinese company SEEED sells the ESP32-Audio-Kit, a small board that incorporates the ESP32-A1S. They have also sold other ESP32 audio boards such as the TTGO-TAudio and the LilyGO. These boards are not as user friendly as the Arduinos and the software libraries tend to be exclusively oriented towards voice recognition "SIRI" type applications or SD card playback.

The company Espressif.com makes audio development boards. Its ESP32-LyraT uses the ESP32-WROVER-B along with an ES8388 audio codec chip by Everest Semiconductor.

Audio Codec chips that include both ADC and DAC with I2S interfaces seem to be the way to go. The AC101 by AI-Thinker is one. The AK4556 by Asahi Kasei Microdevices is another. I've yet to see an external codec breakout board with user friendly software libraries for anything except the Raspberry Pi.

Several small companies have sprung up providing complete audio signal processing solutions with both processor hardware and huge extensive libraries:

Electro-Smith.com	DAISY	uses the AD4556 codec
Bela.io	BELA	uses the BeagleBones Black Micro
Blokas.io	PiSound	uses the Raspberry Pi
Deeptronic.com	BlackStomp	uses the ESP32-A1S