Synthesizer Experiments with Teensy 3.2/4.1

Motivation
Around 1977 I built my first analogue synthesizer. It was the time when a Moog synth was so expensive that DIY was really interesting. Some time after this, keyboards with many possibilities got so cheap that building them seemed no more interesting. Nowadays the microcontrollers and SoCs are so evolved and so cheap that I became interested in renewing with an old passion.

My first goal was to build a MCO (MIDI controlled oscillator) as compared to the VCO (voltage controlled oscillator of the MOOG synthesizer). Maybe I could reuse some of the other componentss of my analogue synthesizer, like filters etc. ?

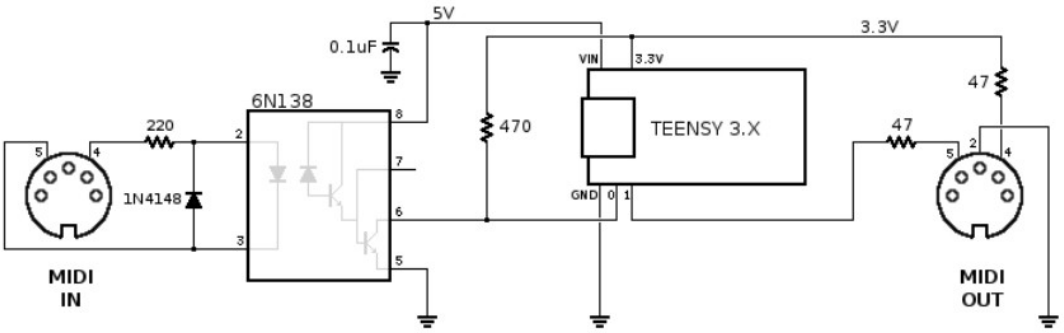
1 Hardware

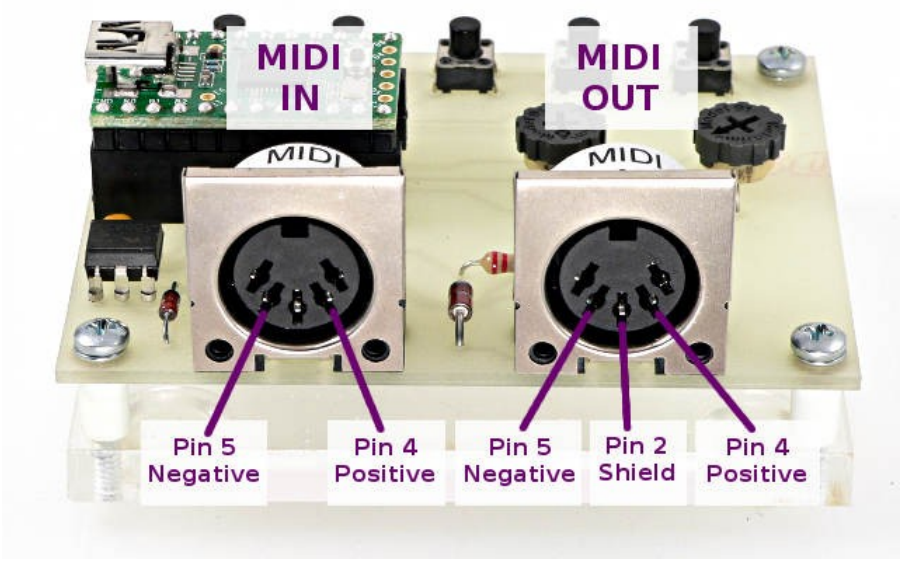
MIDI

(→ prjc.com)

MIDI uses the hardware serial ports to communicate with standard MIDI devices at 31250 baud.

You can use **Serial**.print() to observe what your program is doing, while using MIDI, or USB MIDI.





Which port for MIDI?

Using the Hardware Serial Ports (→ prjc.com)

On Teensy, **Serial** accesses the USB only.

USB and Serial1 (pins 0 & 1) are not shared on Teensy.

For hardware serial ports, **Serial1**, **Serial2**, **Serial3** must be used.

Serial Port	Signal	Teensy 1.0	Teensy 2.0	Teensy++ 1.0, 2.0	Teensy LC	Teensy 3.0, 3.1, 3.2	Teensy 3.5, 3.6	Teensy 4.0	Teensy 4.1
Serial1	Receive	2	7	2	0	0	0	0	0
	Transmit	3	8	3	1	1	1	1	1
Serial2	Receive	-	-	-	9	9	9	7	7
	Transmit	-	-	-	10	10	10	8	8
Serial3	Receive	-	-	-	7	7	7	15	15
	Transmit	-	-	-	8	8	8	14	14

I use Serial 1 on GP0 to receive MISDI messages (this is not compatible with the PJRC audio tutorial that has a button on GP0! But it is not a big deal to shift the buttons to 1,2,3).

Audio shield

The audio shield is stacked beneath the Teensy. It has a stereo high quality line output, and a headphone output. (The line output gives a much better signal than the headphone output).

For a discussion on the necessity of an audio shield, see Notes and Volts. You could also use the internal DAC to generate a mono signal.

Audio shield info:
https://www.pjrc.com/store/teensy3_audio.html

The audio chip, part number SGTL5000, connects to Teensy using 7 signals. The I2C pins SDA and SCL are used to control the chip and adjust parameters. Audio data uses I2S signals, TX (to headphones and/or line out) and RX (from line in or mic), and 3 clocks, LRCLK (44.1 kHz), BCLK (1.41 MHz) and MCLK (11.29 MHz). All 3 clocks are created by Teensy3. The SGTL5000 operates in "slave mode", where all its clock pins are inputs.

Function	Teensy 4.x Pins Rev D	Teensy 3.x Pins Rev C	Shareable
Audio Data	7, 8, 20, 21, 23	9, 11, 13, 22, 23	
Audio Control	18, 19	18, 19	SDA, SCL (other I2C chips)
Volume Pot	15 (A1)	15 (A1)	-
SD Card	10, 11, 12, 13	7, 10, 12, 14	MOSI, MISO, SCK (other SPI chips)

Memory Chip	6, 11, 12, 13	6, 7, 12, 14	MOSI, MISO, SCK (other SPI chips)

The Audio shield can be stacked on a Teensy 3.x. For the Teensy 4.x however, some signals have different pin correlation:

Audio Shield Signal	**Audio shield pin**	**Rev D (Teensy 4.x)**	**Rev C (Teensy 3.x)**	**Function**
MCLK	11	23 (MCLK1)	11 (MCLK)	Audio Master Clock, 11.29 MHz eventually add 100Ohm in line
BCLK	9	21 (BCLK1)	9 (BCLK)	Audio Bit Clock, 1.41 or 2.82 MHz
LRCLK	23	20 (LRCLK1)	23 (LRCLK)	Audio Left/Right Clock, 44.1 kHz
DIN	22	7 (OUT1A)	22 (OUT)	Audio Data from Teensy to Audio Shield
DOUT	13	8 (IN1)	13 (IN)	Audio Data from Audio Shield to Teensy
SCL	19	19	19	Control Clock (I2C)
SDA	18	18	18	Control Data (I2C)
SCK	14	13	14	Data Storage (SPI) Clock
MISO	12	12	12	Data Storage (SPI) from SD/MEM to Teensy
MOSI	7	11	7	Data Storage (SPI) from Teensy to SD/MEM
SDCS	10	10	10	Chip Select (SPI) for SD Card
MEMCS	6	6	6	Chip Select (SPI) for Memory Chip
Vol	15	15 / A1	15 / A1	Volume Thumbwheel (analog signal)

Additionally: GND, +3V3, +5, G

2 MIDI library, basic usage

Download: Included with the Teensyduino Installer
Latest Developments on Github

MIDI works with all Teensy models.

MIDI_CREATE_INSTANCE(HardwareSerial, **Serial1**, **MIDI**);

Define which serial port will be used. Multiple ports may be used if each has a unique name, rather than the default **MIDI**.

MIDI.begin(MIDI_CHANNEL_OMNI);

Initialize the MIDI library. The receive channel may be specified, or MIDI_CHANNEL_OMNI to receive all 16 channels. This only affects reception. You can send on any MIDI channel regardless of this setting.

MIDI.sendNoteOn(note, velocity, channel);
MIDI.sendNoteOff(note, velocity, channel);
MIDI.sendProgramChange(programNumber, channel);
MIDI.sendControlChange(controlNumber, controlValue, channel);
MIDI.sendPitchBend(value, channel);
MIDI.sendPolyPressure(note, pressure, channel);
MIDI.sendAfterTouch(pressure, channel);

Transmit basic MIDI messages. These allow you to easily send each MIDI message.

MIDI.read();

Receive a MIDI message. This returns true if a message has been received, or false if no new message has arrived. After returning true, the get functions can be used to obtain the MIDI message information.

MIDI.getType();

Returns the type of message received. The types are:
midi::NoteOff
midi::NoteOn
midi::AfterTouchPoly
midi::ControlChange
midi::ProgramChange
midi::AfterTouchChannel
midi::PitchBend
midi::SystemExclusive

MIDI.getData1();
MIDI.getData2();

These return the 2 data bytes of the received MIDI message.

Example 1: MIDI send

This simple examples sends a rapid sequence of notes on Tx1 (Pin1 on Teensy 3.x and 4.x). Not very exciting, but a simple and easy test.

```
//01_ex_MIDI_send.ino:

#include <MIDI.h>

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI);
const int channel = 1;

void setup() {
  MIDI.begin();
}

void loop() {
  int note;
  for (note=10; note <= 127; note++) {
    MIDI.sendNoteOn(note, 100, channel);
    delay(200);
    MIDI.sendNoteOff(note, 100, channel);
  }
  delay(2000);
}
```

Example 2: MIDI read

This example receives MIDI messages on Rx1 (Pin 0 on Teensy 3.x and 4.x) and displays them over USB serial. They can be seen on the Serial Monitor.

```
//02_ex_MIDI_receive.ino:

#include <MIDI.h>

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI);

void setup() {
  MIDI.begin(MIDI_CHANNEL_OMNI);
  Serial.begin(57600);
  Serial.println("MIDI Input Test");
}

unsigned long t=0;

void loop() {
  int type, note, velocity, channel, d1, d2;
      if (MIDI.read()) {                    // Is there a MIDI message incoming ?

          byte type = MIDI.getType();

          switch (type) {
            case midi::NoteOn:
                  note = MIDI.getData1();
                  velocity = MIDI.getData2();
                  channel = MIDI.getChannel();
                  if (velocity > 0) {
                    Serial.println(String("Note On:  ch=")
                      + channel + ", note=" + note + ", velocity=" + velocity);
                  } else {
                    Serial.println(String("Note Off: ch=") + channel + ", note=" + note);
                  }
                  break;
            case midi::NoteOff:
                  note = MIDI.getData1();
                  velocity = MIDI.getData2();
                  channel = MIDI.getChannel();
                  Serial.println(String("Note Off: ch=")
```

```
                    + channel + ", note=" + note + ", velocity=" + velocity);
              break;
          default:
              d1 = MIDI.getData1();
              d2 = MIDI.getData2();
              Serial.println(String("Message, type=") + type + ", data = " + d1 + " " + d2);
        }
        t = millis();
    }
    if (millis() - t > 10000) {
        t += 10000;
        Serial.println("(inactivity)");
    }
}
```

3 Audio: first sounds

Info:

https://www.pjrc.com/teensy/td_libs_Audio.html

Audio board:
https://www.pjrc.com/store/teensy3_audio.html

Tutorial:
https://github.com/PaulStoffregen/AudioWorkshop2015/raw/master/workshop.pdf
Connections:
https://www.pjrc.com/store/audio_tutorial_breadboard_setup.pdf

Hello world in audio

The following program beeps using a sine oscillator with 440Hz:

```
// 03a_Hardwaretest_oscillator.ino

#include <Audio.h>

AudioSynthWaveform    waveform1;
AudioOutputI2S        i2s1;
AudioConnection       patchCord1(waveform1, 0, i2s1, 0);
AudioConnection       patchCord2(waveform1, 0, i2s1, 1);
AudioControlSGTL5000  sgtl5000_1;

int count;

void setup() {

    waveform1.begin(WAVEFORM_SINE);
    delay(1000);

}


void loop() {
    waveform1.frequency(440);
    waveform1.amplitude(0.9);
    delay(250);
    waveform1.amplitude(0);
    delay(1750);
}
```

To use other waveforms, see the Audio design tool:

https://www.pjrc.com/teensy/gui/index.html?info=AudioSynthWaveform

The whole Audio lib documentation is found on the Audio design tool webpage!

Supported Waveforms:

- WAVEFORM_SINE
- WAVEFORM_SAWTOOTH
- WAVEFORM_BANDLIMIT_SAWTOOTH
- WAVEFORM_SAWTOOTH_REVERSE
- WAVEFORM_BANDLIMIT_SAWTOOTH_REVERSE
- WAVEFORM_SQUARE
- WAVEFORM_BANDLIMIT_SQUARE
- WAVEFORM_TRIANGLE
- WAVEFORM_TRIANGLE_VARIABLE
- WAVEFORM_ARBITRARY
- WAVEFORM_PULSE
- WAVEFORM_BANDLIMIT_PULSE
- WAVEFORM_SAMPLE_HOLD

4 MIDI to oscillator, naive and simple

Hardware: Teensy 3.2 /4.1 and audio shield stacked together (take care, the Audio shield is not directly compatible with Teensy 4.1, use an adaptor board in between).

MIDI input over optocoupler on Rx1.
The following program reads a MIDI note from the MIDI input and sets the frequency of a sine wave oscillator accordingly:

```
/*
 * 04a_MIDI_2_OSC
 */
#include <MIDI.h>
#include <Audio.h>

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI);

AudioSynthWaveform    waveform1;
AudioOutputI2S        i2s1;
AudioConnection       patchCord1(waveform1, 0, i2s1, 0);
AudioConnection       patchCord2(waveform1, 0, i2s1, 1);
AudioControlSGTL5000  sgtl5000_1;

uint16_t frequency[128] PROGMEM = {8, 9, 9, 10, 10, 11, 12, 12, 13, 14, 15, 15, 16, 17, 18, 19, 21,
22, 23, 24, 26,  28, 29, 31, 33, 35, 37, 39, 41, 44, 46, 49, 52, 55, 58, 62, 65, 69, 73, 78, 82, 87,
92, 98, 104, 110, 117, 123, 131,  139, 147, 156, 165, 175, 185, 196, 208, 220, 233, 247, 262, 277,
294, 311, 330, 349, 370, 392, 415, 440, 466, 494, 523, 554,  587, 622, 659, 698, 740, 784, 831, 880,
932, 988, 1047, 1109, 1175, 1245, 1319, 1397, 1480, 1568, 1661, 1760, 1865, 1976,  2093, 2217, 2349,
2489, 2637, 2794, 2960, 3136, 3322, 3520, 3729, 3951, 4186, 4435, 4699, 4978, 5274, 5588, 5920,
5920,   6645, 7040, 7459, 7902, 8372, 8870, 9397, 9956, 10548, 11175, 11840, 12544};
//--------------------------------------
void setup() {

    MIDI.begin(MIDI_CHANNEL_OMNI);
    MIDI.setHandleNoteOff(note_off);
    MIDI.setHandleNoteOn(note_on);
    AudioMemory(10);
    sgtl5000_1.enable();
```

```
    sgtl5000_1.volume(0.9);
    waveform1.begin(WAVEFORM_SINE);
    delay(1000);
}
//-----------------------------------------

void note_on(byte channel, byte note, byte velocity){
     if (velocity > 0) {
            uint16_t f = frequency[note];
            waveform1.frequency(f);
            waveform1.amplitude(0.9);
      } else {
            // note off
            waveform1.amplitude(0);
      }
}


void note_off(byte channel, byte note, byte velocity){
      waveform1.amplitude(0);
}

//-----------------------------------------------
void loop(){
     MIDI.read();
}
```

The handlers note_on and note_off are automatically called when a NoteOn or a NoteOff event occurs.

This code inspired by the MIDI receive program allows you to play a simple melody, but the note handling is still very poor.

For example when you try to do a trill, it doesn’t work. Why?

You start with a NoteOn, push another note, this gives another NoteOn, and so on. The code expects that for every NoteOn there comes a NoteOff before playing a new note.

5 Using a note buffer

This video from Notes & Volts explains the necessity of using a note buffer:

https://www.youtube.com/watch?v=IoADj8dvTQc

Here is a simple program of a monophonic oscillator using the note buffer:

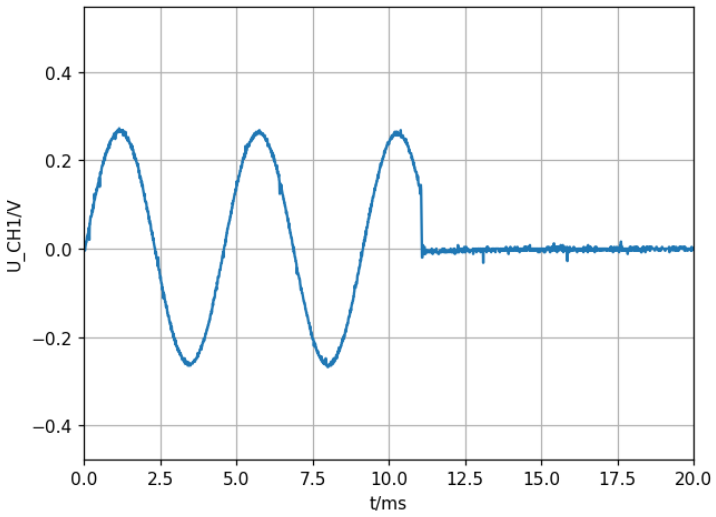
```
/*
 * 05a_MIDI_2_OSC
 * with key buffer   (idea: www.notesandvolts.com)
 */
#include <MIDI.h>
#include <Audio.h>

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI);

AudioSynthWaveform    waveform1;
AudioOutputI2S        i2s1;
AudioConnection       patchCord1(waveform1, 0, i2s1, 0);
AudioConnection       patchCord2(waveform1, 0, i2s1, 1);
AudioControlSGTL5000  sgtl5000_1;

//-------------------------------------
```

```
const byte BUFFER = 8; //Size of keyboard buffer
const float noteFreqs[128] = {8.176, 8.662, 9.177, 9.723, 10.301, 10.913, 11.562, 12.25, 12.978,
13.75, 14.568, 15.434, 16.352, 17.324, 18.354, 19.445, 20.602, 21.827, 23.125, 24.5, 25.957, 27.5,
29.135, 30.868, 32.703, 34.648, 36.708, 38.891, 41.203, 43.654, 46.249, 48.999, 51.913, 55, 58.27,
61.735, 65.406, 69.296, 73.416, 77.782, 82.407, 87.307, 92.499, 97.999, 103.826, 110, 116.541,
123.471, 130.813, 138.591, 146.832, 155.563, 164.814, 174.614, 184.997, 195.998, 207.652, 220,
233.082, 246.942, 261.626, 277.183, 293.665, 311.127, 329.628, 349.228, 369.994, 391.995, 415.305,
440, 466.164, 493.883, 523.251, 554.365, 587.33, 622.254, 659.255, 698.456, 739.989, 783.991,
830.609, 880, 932.328, 987.767, 1046.502, 1108.731, 1174.659, 1244.508, 1318.51, 1396.913, 1479.978,
1567.982, 1661.219, 1760, 1864.655, 1975.533, 2093.005, 2217.461, 2349.318, 2489.016, 2637.02,
2793.826, 2959.955, 3135.963, 3322.438, 3520, 3729.31, 3951.066, 4186.009, 4434.922, 4698.636,
4978.032, 5274.041, 5587.652, 5919.911, 6271.927, 6644.875, 7040, 7458.62, 7902.133, 8372.018,
8869.844, 9397.273, 9956.063, 10548.08, 11175.3, 11839.82, 12543.85};
//int octave = 0;
const float DIV127 = (1.0 / 127.0);


void setup() {

    MIDI.begin(MIDI_CHANNEL_OMNI);
    MIDI.setHandleNoteOff(note_off);
    MIDI.setHandleNoteOn(note_on);
    AudioMemory(10);
    sgtl5000_1.enable();
    sgtl5000_1.volume(0.6);
    waveform1.begin(WAVEFORM_SINE);
    delay(1000);
}
//-----------------------------------------
void note_on(byte channel, byte note, byte velocity){
     if ( note > 23 && note < 108 ) {
          keyBuff(note, velocity, true);
     }
}
//-----------------------------------------------

void note_off(byte channel, byte note, byte velocity){
      if ( note > 23 && note < 108 ) {
            keyBuff(note, velocity, false);
  }
}
//-----------------------------------------------

void keyBuff(byte note, byte velocity, bool playNote) {
  static byte buff[BUFFER];
  static byte buffSize = 0;

  // Add Note
  if (playNote == true && (buffSize < BUFFER) ) {
      oscPlay(note, velocity);
      buff[buffSize] = note;
      buffSize++;
      return;
  }

  // Remove Note
  else if (playNote == false && buffSize != 0) {
      for (byte found = 0; found < buffSize; found++) {
          if (buff[found] == note) {
              for (byte gap = found; gap < (buffSize - 1); gap++) {
                buff[gap] = buff[gap + 1];
            }
            buffSize--;
            buff[buffSize] = 255;
            if (buffSize != 0) {
                oscPlay(buff[buffSize - 1], velocity);
                return;
            }
            else {
                oscStop();
                return;
            }
          }
      }
  }
}
```

```
//-------------------------------------------------------
void oscPlay(byte note, byte velocity) {
    waveform1.frequency(noteFreqs[note]);
    //float velo = (globalVelocity * DIV127);
    float velo = (velocity * DIV127);
    waveform1.amplitude(velo);
}
//-------------------------------------------------------
void oscStop() {
    waveform1.amplitude(0.0);
}

//-----------------------------------------------
void loop(){
     MIDI.read();
}
```

For better accuracy, the note frequencies have been changed to float numbers.

However, there is a klicking noise , especially at NoteOff.
The oscilloscope shows why:



6 Adding a second oscillator for richer sound

The sine wave sound is a bit sterile (whilest sawtooth and rect are aggressive without filtering). It might be made more interesting by adding a second (eventually slightly detuned) oscillator.

The audio definitions

```
AudioSynthWaveform    waveform1;
AudioOutputI2S        i2s1;
AudioConnection       patchCord1(waveform1, 0, i2s1, 0);
AudioConnection       patchCord2(waveform1, 0, i2s1, 1);
AudioControlSGTL5000  sgtl5000_1;
```

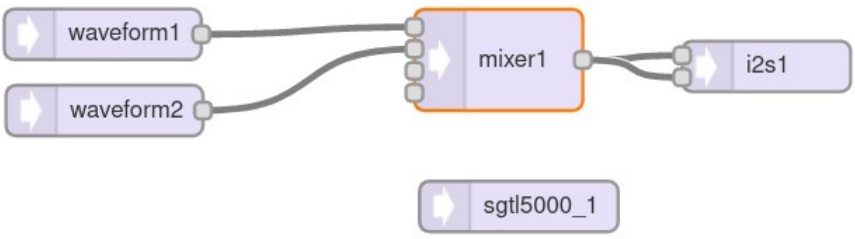
can be imported to the Audio design tool

https://www.pjrc.com/teensy/gui/index.html

if we want to add new components.

By the way, it is not a good idea to delete the xy infos of the blocks in the code, as I had done, as this leads to a cluster where everything is stacked on top of each other in the same place. But as the schematic was not very complicated I could separate the blocks by using the mouse.

I added one more oscillator and a mixer:



The result can now be exported and pasted to the Arduino program instead of the old definitions.

In the setup function we have to define 2 waveforms:

```
void setup() {
    MIDI.begin(MIDI_CHANNEL_OMNI);
    MIDI.setHandleNoteOff(note_off);
    MIDI.setHandleNoteOn(note_on);
    AudioMemory(10);
    sgtl5000_1.enable();
    sgtl5000_1.volume(0.6);
    waveform1.begin(WAVEFORM_SINE);
    waveform2.begin(WAVEFORM_SINE);
    mixer1.gain(0, 0.5);
    mixer1.gain(1, 0.5);
    delay(1000);
}
```

I reduced the mixer gain to 0.5 to avoid distortions (though these might even make the sound more interesting.)

And in the oscPlay and oscStop functions we have to do something with both waveforms.

For example detune one of the oscillators by 1-10Hz:

```
waveform2.frequency(noteFreqs[note] + 5);
```

This gives a sound that has the beat of the detuning frequency.

Another idea is to tune the second oscillator one octave higher by multiplying its frequency by a factor 2:

```
void oscPlay(byte note, byte velocity) {
    waveform1.frequency(noteFreqs[note]);
    waveform2.frequency(noteFreqs[note] *2 + 5);
    float velo = (velocity * DIV127);
    waveform1.amplitude(velo);
    waveform2.amplitude(velo);
}
//-------------------------------------------------------
void oscStop() {
```

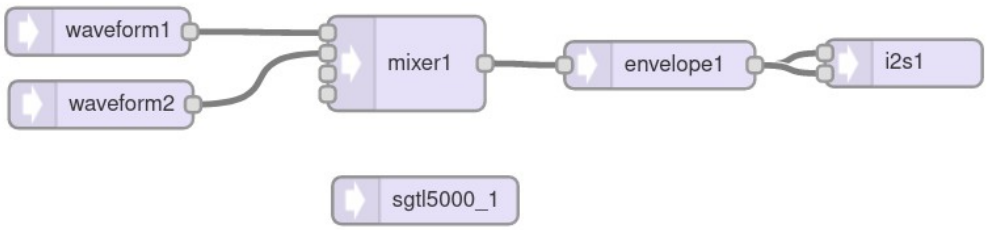
```
    waveform1.amplitude(0.0);
    waveform2.amplitude(0.0);
}
```

That gives us a much richer sound.

Later of course the parameters will be set by controls (potentiometers and switches).

7 ADSR: Attack – Delay - Sustain – Release

My motivation to include an ADSR at this stage of the project already was that I hoped to avoid the clicking noises at the end of a note by smoothing this phase with the ADSR.



The exported code is put at the beginning of the sketch.

In the setup function, we put some default values:

```
    envelope1.attack(0);
    envelope1.decay(0);
    envelope1.sustain(1);
    envelope1.release(500);
```

An additional

```
    envelope1.releaseNoteOn(5);
```

is said to reduce the transition klicks further, see

https://forum.pjrc.com/archive/index.php/t-61599.html

Now the oscPlay and oscStop functions must be modified to include an envlope function:

```
void oscPlay(byte note) {
    waveform1.frequency(noteFreqs[note]);
    waveform2.frequency(noteFreqs[note] *2 + 5);
    float velo = (globalVelocity * DIV127);
    waveform1.amplitude(velo);
    waveform2.amplitude(velo);
    envelope1.noteOn();
}

void oscStop() {
    envelope1.noteOff();
}
```

At this stage, the ADSR uses fix parameters, later I want to set the values interactively with potentiometers.

Anyway, the sound is better with this modification.

8 MIDI panic ?

Mostly the system is running well, but from time to time a note got stuck and the oscillator kept on going without any key being pushed. This even happened at powerup sometimes.

I did my best to debug the problem, but seemingly the best was not enough: sometimes the phenomenon occurs, with no appearent cause.

I had three ideas to make things better:

- In the keybuffer function I put some prints to see what exactly happens
- At startup, the serial input buffer is flushed to avoid reading false bytes
- A panic button is added that stops the oscillator, prints information about the buffer and clears it.

The key buffer with debug prints:

```
void keyBuff(byte note,  bool playNote) {
  // playNote: True -> play note and add it to buffer
  //           False -> stop note and remove it from buffer


  // Add Note to buffer and play it:
  if (playNote == true && (buffSize < BUFFER) ) {
      oscPlay(note);
      buff[buffSize] = note;
      buffSize++;
      Serial.print( " P");
      Serial.print(note);
      return;
  }

  // Remove Note
  else if (playNote == false && buffSize != 0) {

      // search through the buffer for note:
      for (byte found = 0; found < buffSize; found++) {

          // if the note is found:
          if (buff[found] == note) {

                // close the gap and reduce the buffer size:
                for (byte gap = found; gap < (buffSize - 1); gap++) {
                    buff[gap] = buff[gap + 1];
                }
                buffSize--;

                buff[buffSize] = 255;         // why this ??

                // play the last note in the buffer, if it is not empty:
                if (buffSize != 0) {
                      oscPlay(buff[buffSize - 1]);
                      Serial.print( " p");
```

```
                        Serial.print(buff[buffSize - 1]);
                        return;
                    }

                else {
                    oscStop();

                    Serial.print(" S B:");
                    Serial.println(buffSize);
                    return;
                  }
                }
            }
      }
}
```

It is interesting to watch what happens when some notes are played. For example this

```
+48 P48 +50 P50 +52 P52 -52 p50 -50 p48 -48 S B:0
```

is playing C(48), then D(50) and 52(E) while holding the C key down, then releasing all. At the end, the buffer is empty (B:0).

The same debugging technique allowed me to see that sometimes at the beginning a note was added that I never played. This could come from a MIDI buffer that was not totally empty.
So I put a clearing function into the code:

```
 void setup() {
    ...
    clearMIDIinputBuffer();
    MIDI.begin(MIDI_CHANNEL_OMNI);
    MIDI.setHandleNoteOff(note_off);
    MIDI.setHandleNoteOn(note_on);
    Serial.println("READY!");
}

void clearMIDIinputBuffer() {
          // clear MIDI input buffer
          uint32_t m = micros();
          do {
                  if (Serial1.read() >= 0) {
                  m = micros();
                  }
          } while (micros() - m < 10000);
}
```

The third change was to introduce a **panic button** (on GP2) to stop the oscillator:

```
const byte MIDI_panic = 2;

void setup() {
    pinMode(MIDI_panic, INPUT_PULLUP);
    ...
}
//-----------------------------------------------
void MIDIpanic(){
    Serial.println("STOP");
    delay(20);
    oscStop();

    Serial.println("BUFFER:");
    for ( int i = 0; i <= buffSize; ++i ) {
        Serial.print(buff[i]);
        Serial.print("    ");
    }
    Serial.print("Buffer size: ");
    Serial.println(buffSize);
```

```
    buffSize = 0;
   delay(50);
}
//-----------------------------------------------
void loop(){
     MIDI.read();
     if (digitalRead(MIDI_panic) == 0) {
          MIDIpanic();
     }
}
```

Now I could stop any crazy oscillation.

```
/*
 * 06_MIDI_2_OSC_ADSR
 */
#include <MIDI.h>
#include <Audio.h>
#include "frequencies.h"

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI);

// GUItool: begin automatically generated code
AudioSynthWaveform       waveform1;      //xy=386.16668701171875,344
AudioSynthWaveform       waveform2;      //xy=388.16668701171875,390.16668701171875
AudioMixer4              mixer1;         //xy=582.1666717529297,364.1666717529297
AudioEffectEnvelope      envelope1;      //xy=745.1666717529297,365.1666717529297
AudioOutputI2S           i2s1;           //xy=901.1666870117188,373
AudioConnection          patchCord1(waveform1, 0, mixer1, 0);
AudioConnection          patchCord2(waveform2, 0, mixer1, 1);
AudioConnection          patchCord3(mixer1, envelope1);
AudioConnection          patchCord4(envelope1, 0, i2s1, 0);
AudioConnection          patchCord5(envelope1, 0, i2s1, 1);
AudioControlSGTL5000     sgtl5000_1;     //xy=594.1666870117188,450.9999694824219
// GUItool: end automatically generated code


//---------------------------------------
// keyboard buffer:
const byte BUFFER = 8;
byte buff[BUFFER];
byte buffSize = 0;
byte globalNote = 0;
byte globalVelocity = 0;


const float DIV127 = (1.0 / 127.0);

const byte MIDI_panic = 2;

void setup() {
    pinMode(MIDI_panic, INPUT_PULLUP);

    Serial.begin(115200);

    AudioMemory(10);
    sgtl5000_1.enable();
    sgtl5000_1.volume(0.6);
    waveform1.begin(WAVEFORM_SINE);
    waveform2.begin(WAVEFORM_SINE);
    mixer1.gain(0, 0.5);
    mixer1.gain(1, 0.5);
    envelope1.attack(10);
    envelope1.decay(10);
    envelope1.sustain(200);
    envelope1.release(500);
    envelope1.releaseNoteOn(5);
    delay(100);
    clearMIDIinputBuffer();
    MIDI.begin(MIDI_CHANNEL_OMNI);
    MIDI.setHandleNoteOff(note_off);
    MIDI.setHandleNoteOn(note_on);

    Serial.println("READY!");
}
//-----------------------------------------------
```

```
void MIDIpanic(){
    Serial.println("STOP");
    delay(20);
    oscStop();

    Serial.println("BUFFER:");
    for ( int i = 0; i <= buffSize; ++i ) {
        Serial.print(buff[i]);
        Serial.print("    ");
    }
    Serial.print("Buffer size: ");
    Serial.println(buffSize);
    buffSize = 0;
   delay(50);


}
//-------------------------------------------------
void loop(){
     MIDI.read();
     if (digitalRead(MIDI_panic) == 0) {
          MIDIpanic();
     }
}

//-------------------------------------------------------
void clearMIDIinputBuffer() {
  // clear MIDI input buffer
  uint32_t m = micros();
  do {
  if (Serial1.read() >= 0) {
  m = micros();
  }
  } while (micros() - m < 10000);
}
//----------------------------------------
void note_on(byte channel, byte note, byte velocity){
     if ( note > 23 && note < 127 ) {
          Serial.print(" +");
          Serial.print(note);
          globalNote = note;
          globalVelocity = velocity;
          keyBuff(note,  true);

     }
     else { Serial.println("*"); }
}
//-----------------------------------------------

void note_off(byte channel, byte note, byte velocity){
      if ( note > 23 && note < 127 ) {
            Serial.print(" -");
            Serial.print(note);
            keyBuff(note, false);

      }
      else { Serial.println("_"); }
}
//-----------------------------------------------

void clear_keyBuff(){

   for ( int i = 0; i < BUFFER; ++i ) {
        buff[i] = 0;
   }
   buffSize = 0;
   //oscStop();
}


void keyBuff(byte note,  bool playNote) {
  // playNote: True -> play note and add it to buffer
  //           False -> stop note and remove it from buffer


  // Add Note to buffer and play it:
  if (playNote == true && (buffSize < BUFFER) ) {
```

```
      oscPlay(note);
      buff[buffSize] = note;
      buffSize++;
      Serial.print( " P");
      Serial.print(note);
      return;
  }

  // Remove Note
  else if (playNote == false && buffSize != 0) {

      // search through the buffer for note:
      for (byte found = 0; found < buffSize; found++) {

          // if the note is found:
          if (buff[found] == note) {

                // close the gap and reduce the buffer size:
                for (byte gap = found; gap < (buffSize - 1); gap++) {
                    buff[gap] = buff[gap + 1];
                }
                buffSize--;

                buff[buffSize] = 255;         // why this ??

                // play the last note in the buffer, if it is not empty:
                if (buffSize != 0) {
                      oscPlay(buff[buffSize - 1]);
                      Serial.print( " p");
                      Serial.print(buff[buffSize - 1]);
                      return;
                }

          else {
                oscStop();

                Serial.print(" S B:");
                Serial.println(buffSize);
                return;
            }
          }
      }
  }

}

//-------------------------------------------------------
void oscPlay(byte note) {
    waveform1.frequency(noteFreqs[note]);
    waveform2.frequency(noteFreqs[note] *2 + 5);
    float velo = (globalVelocity * DIV127);
    waveform1.amplitude(velo);
    waveform2.amplitude(velo);
    envelope1.noteOn();
}
//-------------------------------------------------------
void oscStop() {
    envelope1.noteOff();
}
```

9 Polyphony ? First ideas

The first MOOG synthesizer (and my analogue synthesizer) could only play one key at a time, like what we have realized until now.

It would be nice to be able to play chords or (at least) two voices together.

How complicated is this?

To see what has to be done I wrote some Python code (not really playing any notes, but showing what happens at note on and note off, and easier to debug and giving a better overview):

```
osc_notes = [48, 50, 52, 56, 0, 0, 23]

def is_playing(note, osc_notes):
    result = False
    for n in osc_notes:
        if n == note:
            result = True
            break
    return result



def get_free_osc(osc_notes):
    i=0
    for note in osc_notes:
       if note ==0:
           break
       i += 1
    return i

def add_note(note, osc_notes):
    if not (is_playing(note, osc_notes)):
        i = get_free_osc(osc_notes)
        osc_notes[i] = note
    return osc_notes


def remove_note(note, osc_notes):
    i = 0
    for n in osc_notes:
       if n == note:
           osc_notes[i] = 0
           stopped = i
           break
       i+= 1

    return osc_notes, i


print("Original notes:")
print(osc_notes)
print()

note = 63
r = is_playing(note, osc_notes)
print("Is note ", note, " playing?:", r)


i = get_free_osc(osc_notes)
print("Next free OSC:", i)

new_notes = add_note(note, osc_notes)
print("Add a note ", note, " to free OSC:")
print(new_notes)
print()

new_notes, stopped = remove_note(note, osc_notes)
print("Remove a note: ", note)
print(new_notes)
print("Stopped: ", stopped)
```

An array stores the notes that are playing. (This could, on the Teensy, correspond to an array of oscillators. The important functions are **add_note** and **remove_note**.

Running this example code gives:

```
Original notes:
[48, 50, 52, 56, 0, 0, 23]

Is note  63  playing?: False
```

```
Next free OSC: 4
Add a note  63  to free OSC:
[48, 50, 52, 56, 63, 0, 23]

Remove a note:  63
[48, 50, 52, 56, 0, 0, 23]
Stopped:  4
```

So it seems to work.

A first difficulty in porting this to Teensy C++ is the need for arrays of audio objects.

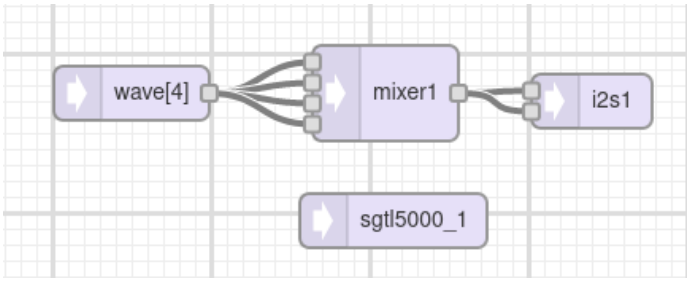
When doing a research on internet I found several solutions using a really complicated diagram in the audio design GUI an d a lot of if – else in the code. This was not the elegant solution I was looking for.

I did some tests and noticed that I can define all audio objects as arrays, except the AudioConnections. But there is a way to define them dynamically at runtime, as Jannik Manicken shows here:

https://forum.pjrc.com/printthread.php?t=68944&pp=25&page=1

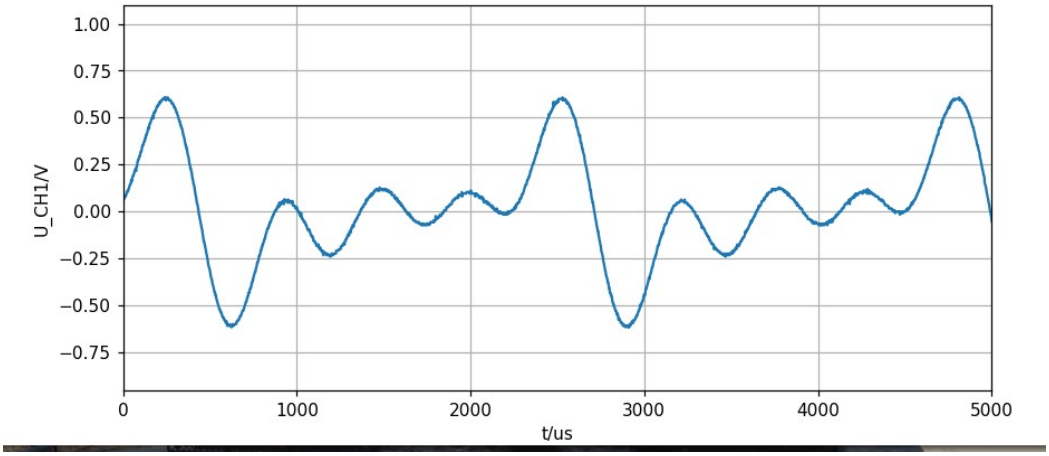
10 Using arrays of audio objects

This sample code does a Fourier synthesis, adding 4 sine waves to make a complex signal:




```
#include <Audio.h>

AudioSynthWaveform    wave[4];
AudioOutputI2S        i2s1;
AudioMixer4           mixer1;
AudioConnection       patchCord1(mixer1, 0, i2s1, 0);
AudioConnection       patchCord2(mixer1, 0, i2s1, 1);
AudioControlSGTL5000  sgtl5000_1;

AudioConnection  *patchCord[10];

//--------------------------------------
void setup() {

    AudioMemory(10);
    sgtl5000_1.enable();
    sgtl5000_1.volume(0.9);

    for(int i=0; i<=3; i++){
        patchCord[i] = new AudioConnection(wave[i], 0, mixer1, i);
        wave[i].begin(WAVEFORM_SINE);
        wave[i].frequency((i+1) * 440);
        wave[i].amplitude(0.3);
    }
}
//--------------------------------------
void loop(){
```

}



The oscillators are defined as array:

```
AudioSynthWaveform    wave[4];
```

In the for loop, the waveforms, frequencies and and amplitudes are defined, and the Audioconnections are created dynamically:

```
for(int i=0; i<=3; i++){
        patchCord[i] = new AudioConnection(wave[i], 0, mixer1, i);
        wave[i].begin(WAVEFORM_SINE);
        wave[i].frequency((i+1) * 440);
        wave[i].amplitude(0.3);
    }
```

Before this can be done, the array of Audioconnections must be prepared:

```
AudioConnection  *patchCord[10];
```

(Notice the * operator!)

11 Using Jannik Svensson's Audio GUI tool

There is a phantastic tool going beyond Paul Stoffregens Audio tool:

https://manicken.github.io

I made a break and tried to understand how to use this tool. Not so easy as it has many undocumented features. The author helped me a lot to get on:

https://forum.pjrc.com/threads/69109-Audio-Lib-Manicken-design-tool?p=296954#post296954

Here is my humble contribution to the documentation:

https://github.com/jean-claudeF/TeensySynth/blob/main/MannickenAudioTool.pdf

12 A simple polyphonic synthesizer with Jannik's tool

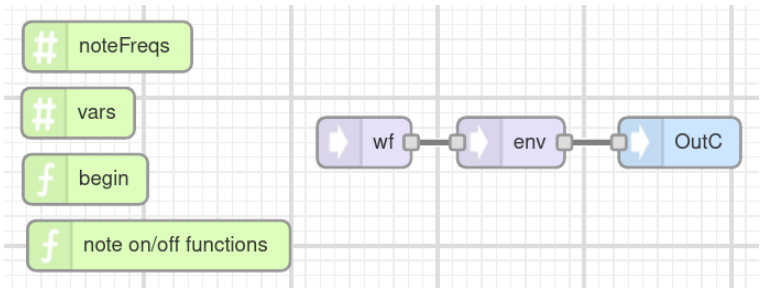
On my demand, Jannik was so friendly as to contribute a downsized synthesizer project for the tool, which helps a lot to understand and can be used as a basis for own projects

Here is a link to the JSON file that can be imported to the Tool:

`https://raw.githubusercontent.com/manicken/manicken.github.io/master/examples/SimplePolySynth.json`

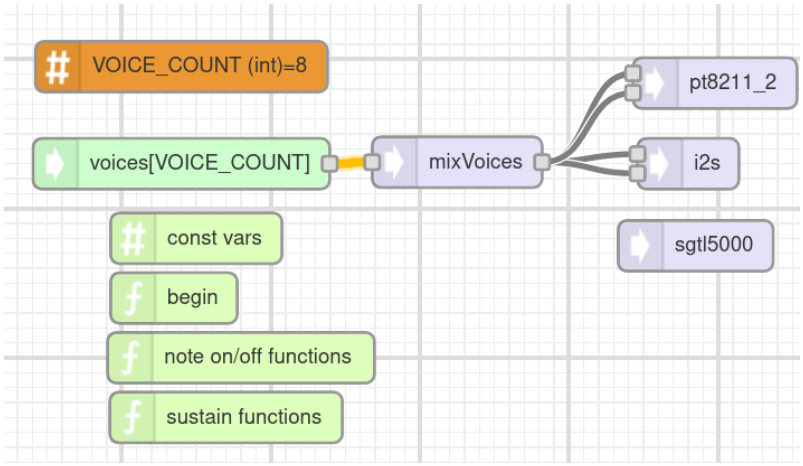
I will only show the most important parts of it:

Voice:



Main:

Synth:



The **Voice** tab takes care of ONE oscillator at the lowest level, it contains essentially an oscillator wf, an envelope env function and note on / off functions.

```
//-----------------------------------------------
    void begin()  {
        wf.begin(1.0, 110, WAVEFORM_SINE);
    }
    //-----------------------------------------------
    void noteOn(byte Note, byte velocity) {
        float newAmp = 0.0f;
        if (Note >= sizeof(noteFreqs)) return;

        note = Note;
        isNoteOn = 1;

        newAmp = (float)velocity*(1.0f / 127.0f);
        wf.frequency(noteFreqs[note]);
```

```
        wf.amplitude(newAmp);
        env.noteOn();
    }
    //-----------------------------------------------
    void noteOff()  {
        isNoteOn = 0;
        if (!isSustain)
            env.noteOff();
    }
    //-----------------------------------------------
    bool isNotPlaying()  {
        if (!env.isActive())
            return true;
        else
            return false;
    }
```

The Synth tab is one level higher, it checks for Note ON if a note is already playing, if there is a free “slot” to play a new note and for Note OFF it turns down the corresponding oscillator.

The difference here is that there is an array of oscillators that must be taken care of.

```
void noteOn(byte note, byte velocity)
    {
        // first checks if this note is already playing
        // if that is the case then it "reuses" this "slot"
        // this makes sure that not all "slots" are filled
        // with the same playing note

        for (int i = 0; i < VOICE_COUNT; i++)
        {
            // first check if the note was played recently
            if (voices[i].note == note)
            {
                voices[i].noteOn(note, velocity);
                return;
            }
        }
        // then if the note has not allready been played
        // // second see if there is any free "spot"
        for (int i = 0; i < VOICE_COUNT; i++)
        {
            if (voices[i].isNotPlaying())   {
                voices[i].noteOn(note, velocity);
                return;
            }
        }
    }

    void noteOff(byte note)
    {
        for (int i = 0; i < VOICE_COUNT; i++)        {
            if (voices[i].note == note)   {
                voices[i].noteOff();
                return;
            }
        }
    }
```

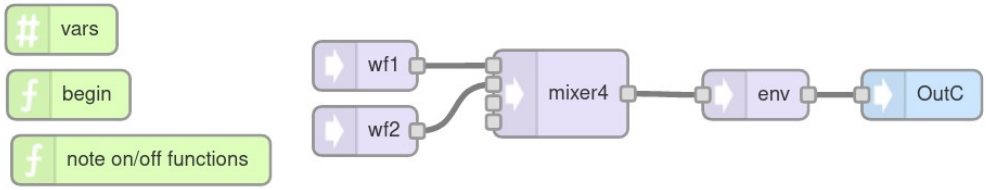
The main tab contains declarations and, as the most important part, the loop function that only checks for MIDI events. The declared MIDI handlers are automatically called if a MIDI event is happening.

13 Going further from the SimplePolySynth

Jannik's PolySyntn example was exactly what I needed to go on, from a simple design to more complex ones.

First thing of course is to add a second, slightly detuned oscillator:

In the Voice tab I have now:



Of course the begin function must initialize 2 waveforms now:

```
void begin()
{
    wf1.begin(1.0, 110, WAVEFORM_SINE);
    wf2.begin(1.0, 110, WAVEFORM_SINE);
}
```

In the noteOn function we must take care of two waveforms also:

```
void noteOn(byte Note, byte velocity)
{
    float newAmp = 0.0f;
    if (Note >= sizeof(noteFreqs)) return;

    note = Note;
    isNoteOn = 1;

    newAmp = (float)velocity*(1.0f / 127.0f)/2;

    wf1.frequency(noteFreqs[note]);
    wf2.frequency(noteFreqs[note]+2);

    wf1.amplitude(newAmp);
    wf2.amplitude(newAmp);

    env.noteOn();
}
```

Here I have added a 2Hz frequency offset to wf2 that gives a floating celestial sound.

The amplitude NewAmp has a division factor of 2, to avoid distortion (though this also may sound interesting).

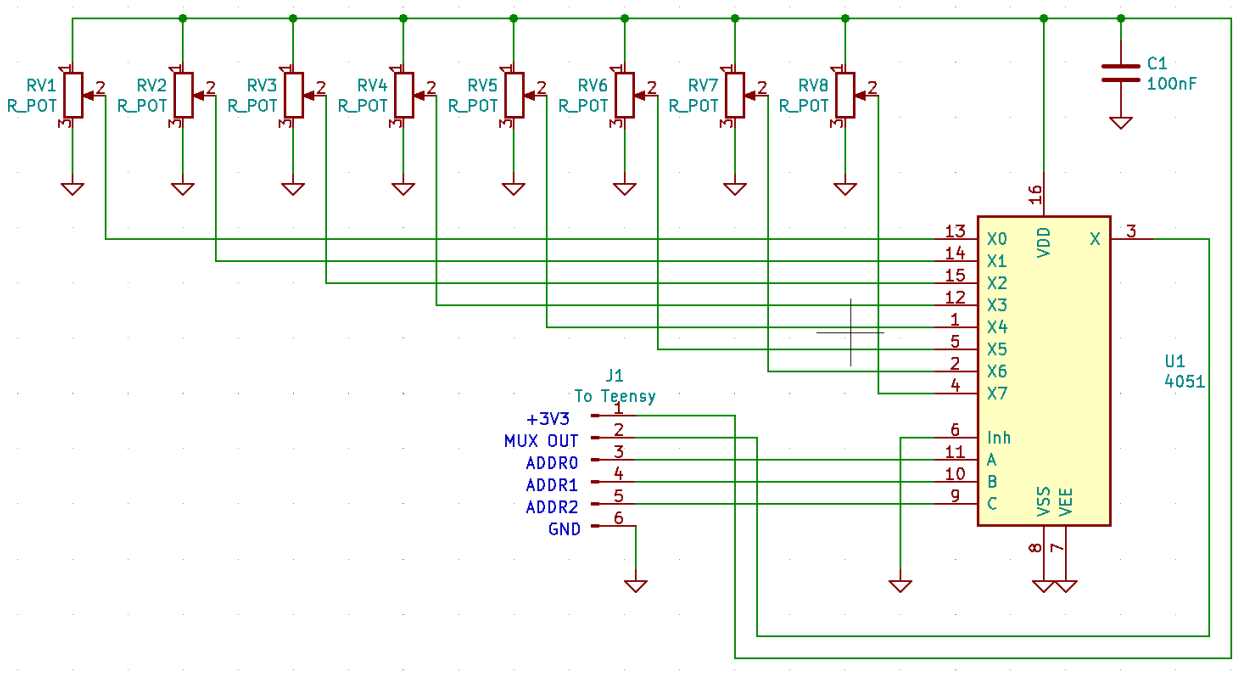
I exported this to Class Based ZIP, copied the file Voice.h and pasted it to the Arduino code.
(There is a way to automatically push it to the IDE, but until now I have not got that to work)

14 Providing control with potentiometers

There are now already several parameters to be controlled manually, and there will be more.

As the Teensy has many, but not an enormous amount of pins, I decided at this moment to attach 8 potentiometers through a 4051 analog multiplexer. Later probably there will be more.

(And at this moment, while writing, I realize the importance of the INHIBIT pin that I will use later to select between several MUXes. For a first test it is connected to GND.)



Fortunately the 4051 operates at 3.3V supply voltage, and fortunately, if no negative analogue signal is used, VEE can be connected to GND instead of a negative supply voltage.

The 3 address pins are connected to GP33,34,35 and the MUX OUT is read by analog input A17.

I use a class for the MUX object, so that later it is easy to use more than one 4051:

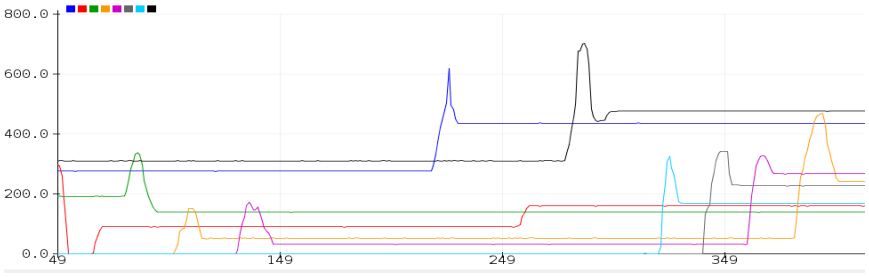
```
//mux4051.h

class Mux4051 {
public:
    word values[8];
    int A0pin;
    int A1pin;
    int A2pin;
    int outpin;
    //-------------------------------------------------------
    void begin(int addr0, int addr1, int addr2, int out){
          A0pin = addr0;
          A1pin = addr1;
          A2pin = addr2;
          outpin = out;
          pinMode(A0pin, OUTPUT);
          pinMode(A1pin, OUTPUT);
          pinMode(A2pin, OUTPUT);

    }
    //-------------------------------------------------------
    void setChannel( byte ch ){
```

```
        byte a0 = ch & 1;
        byte a1 = (ch & 2) >> 1;
        byte a2 = (ch & 4) >>2;
        digitalWrite(A0pin, a0);
        digitalWrite(A1pin, a1);
        digitalWrite(A2pin, a2);
        delayMicroseconds(20);
    }
    //-----------------------------------------------------
    word readChannel(byte ch){
        setChannel(ch);
        word value = analogRead(outpin);
        return value;
    }
    //-----------------------------------------------------
    void readAllChannels(word values[], int nbvalues){
        for (byte i=0; i<nbvalues; i++){
            setChannel(i);
            values[i] = readChannel(i);
            //Serial.println(values[i]);
        }
    }
};
```

A first test program is used to read the values of the potentiometers:

```
/*  muxtest.ino
 *  Read 8 potentiometers through a 4051 multiplexer
 */

word values[8];     //0...1023

//----------------------------------------
#include "mux4051.h"
//----------------------------------------

Mux4051 mymux;

void setup()
{

       mymux.begin(33, 34, 35, A17);
       Serial.begin(115200);
       Serial.println("HELLO");
  }

//-----------------------------------------------------
void loop()
{

      word v = mymux.readChannel(1);
      //Serial.println(v);

      mymux.readAllChannels(values, 8);
      for(byte i=0; i<8; i++){
          Serial.print(values[i]);
          Serial.print('\t');

      }
      Serial.println();
      delay(100);
 }
```

For controlling the circuit, Arduino's Serial Plotter is useful:



15 Using Pot 0 to control osc2 detuning

The first potentiometer pot 0 shall control the detuning of the scond oscillator.
There are two ways to do it:

- additive: add Δf to the base frequency.
  This would be suitable to produce a beat in the range of 1Hz for all tones
- multiplicative: muliply the frequency by a factor.
  This could be interesting to produce overtones, constant musical intervals like octaves, fiths, etc.

Both methods need to read the pot value and to take influence on the voice frequency.

I started with the additive variant.

Main tab (SimplePolySynth_02.ino):

```
//…
word muxvalues[8];     //0...1023
#include "mux4051.h"
Mux4051 mymux;

void setup()
{
   mymux.begin(33, 34, 35, A17);
  //...


void loop(){

    MIDI.read();

    mymux.readAllChannels(muxvalues, 8);
    float f_off = muxvalues[0];
    synth.setNoteOffset(f_off/50);
    blinkLedTask();
}
```

I declare a Mux4051 object and an array muxvalues to hold the pot values.
In the setup function, the mux is started with the pins 33, 34, 35 for the addresses and A17 for the analog input.

The loop function reads the all pots and uses pot0 (after scaling it to 0...20Hz) as note offset that is passed to the synth.setNoteOffset function.

Synth.h:

```
class Synth
{
public:
    //...

    void setNoteOffset(float f_offset){
          for (int i=0; i<VOICE_COUNT; i++){
              voices[i].setNoteOffset(f_offset);
          }
    }
```

```
};
```

The Synth class gets a new function setNoteOffset which sets the offset for all voices.

Finally in the Voice.h tab there is a change in the noteOn function:

```
class Voice
{
public:
    // ...

    void noteOn(byte Note, byte velocity)
    {
        // ...

        wf1.frequency(noteFreqs[note]);
        wf2.frequency(noteFreqs[note] + noteOffset);

        wf1.amplitude(newAmp);
        wf2.amplitude(newAmp);

        env.noteOn();
    }
```