ProtoTone Synth

Author

Antoniu FLOREA | Github | YouTube | CB

Gallery

Introduction

ProtoTone Synth is a musical instrument based on the widely popular Stylophone. You can play musical notes by touching the copper pads with a stylus and manipulate them in real time.

You can do wavy pitch bends, add modulation or vibrato, switch up the octave and see the currently playing note on the screen.

I wanted to create something that has to do with music because this is what I am passionate about, and I never used a Stylophone, so it made perfect sense to just make my own. This will be a great demonstration of analogue signals, audio generation, digital control and communication with external devices.

At the end, it will be a fully functioning musical instrument, which will be possible to be customized even further by anyone willing.

General Description

Block Diagram

Modules

Based on the key pressed, the microcontroller will output a different musical note and it will be shown on the display. The musical note can be pitched up or down using the joystick, which is just a potentiometer that returns to the neutral position. With the twisting of the Modulation potentiometer, the note will begin to jitter consistenly based on the frequency set. The volume will also be able to be adjusted by twisting yet another potentiometer. At the end, we are left with the actions of the 4 buttons. Two are for choosing the octave, going both ways. Another is for holding the current note, letting you lift the stylus off the key and it would still play, and the last one is for cycling through waveforms. You can choose between square, saw and triangle.

The display will output the current note playing, the current octave for a short period of time when changing it in any way, and the current waveform when cycling.

UART is used for debugging via the computer terminal.

The speaker brings it all together by (hopefully) making the desired sounds.

Hardware Design

Components

Component	Quantity
ATmega328P Xplained Mini	1
Breadboard	1
Mini Breadboard	4
OLED I2C SSD1306	1
analogue Joystick	1
PAM8403 module	1
4 Ohm Speaker	1
Potentiometer 10k	2
Buttons	4
Resistors 10k 1%	13
Capacitors 100nF	8
Dupont wires	10
Hard wires for soldering	1 set
Copper tape	1 roll

Schematic

prototone_synth_schematic.pdf

The pin blocks 'Jxxx' represent the group of pins available on the ATmega 328P Xplained Mini. Instead of creating the footprint of the whole board, I chose to represent the important parts of it in a simplified way and label them accordingly.

Header	Description
J200	Port B pins and some Port C pins. Used for generating the audio signal through PWM. Also provides the SDA and SCL signals for the OLED screen.
J201	Port D pins used for simple digital functionalities, such as control buttons and a status LED.
J202	Power and ground connections for the external modules.
J203	ADC header. All analog inputs are connected here, including note selection, pitch bend, vibrato and volume.

The 'keys' are made out of a resistor ladder. Each pin you choose to read from will have a different voltage which will dictate the out tone. In reality, they will be connected to copper pads shaped as piano keys.

The screen is connected through it's 4 simple pins - SDA, SCL, GND and 5V.

Everything modulation is done via potentiometers. 2 are for the volume and vibrato rate, while the pitch adjustment is done though a joystick potentiometer (because it comes back to the center naturally) and we only read one axis from it.

The audio block is composed of the main amplifier, the filter, and the speaker. The audio comes in and enters the first RC filter, then into the PAM8403 circuit. It then hits the mono speaker - only the left channel is used.

Components and Pins

ATmega328P Xplained Mini

The brains of the project. Provides power and ground reference, reads analog and digital inputs and based on them creates the sound.

Xplained Mini Header	Pin	ATmega328P Pin	Project Function
J200	2	PB1 / OC1A	Audio PWM output
J200	7	GND	Ground
J200	9	PC4 / SDA	OLED I2C SDA
J200	10	PC5 / SCL	OLED I2C SCL
J201	1	PD0 / RXD	UART RX / serial debug
J201	2	PD1 / TXD	UART TX / serial debug
J201	3	PD2 / INT0	Octave down button
J201	4	PD3 / INT1	Octave up button
J201	5	PD4	Waveform select button
J201	6	PD5	Hold / sustain button
J201	7	PD6	Status LED
J202	5	5V	5 V supply for external modules

J202	6	GND	Ground
J202	7	GND	Ground
J203	1	PC0 / ADC0	Note selector / stylus ADC
J203	2	PC1 / ADC1	Pitch bend ADC
J203	3	PC2 / ADC2	Vibrato / modulation ADC
J203	4	PC3 / ADC3	Volume ADC

OLED Display

Shows useful information like the current key, volume, vibrato and pitch values.

Module Pin	Connected To	Function
GND	GND	Ground
VCC	+5V	Power supply
SDA	J200 pin 9 / PC4 / SDA	I2C data line
SCL	J200 pin 10 / PC5 / SCL	I2C clock line

Potentiometers

Control volume and vibrato levels.

Component	Pin	Connected To	Function
Vibrato potentiometer	Side pin 1	GND	Voltage divider low side
Vibrato potentiometer	Middle pin	J203 pin 3 / PC2 / ADC2	Vibrato / modulation analog input
Vibrato potentiometer	Side pin 2	+5V	Voltage divider high side
Volume potentiometer	Side pin 1	GND	Voltage divider low side
Volume potentiometer	Middle pin	J203 pin 4 / PC3 / ADC3	Volume analog input
Volume potentiometer	Side pin 2	+5V	Voltage divider high side

Joystick

Controls the pitch shift up and down.

Joystick Pin	Connected To	Function
GND	GND	Ground
VCC	+5V	Power supply
VRx	J203 pin 2 / PC1 / ADC1	Pitch bend analog input
VRy	Not connected	Unused axis
SW	Not connected	Unused joystick button

PAM8403 Audio Amplifier

Amplifies the audio signal cleanly.

PAM8403 Pin	Connected To	Function
+5V / +	+5V	Amplifier power supply
GND / -	GND	Ground
LIN / L	AUDIO_FILTERED	Left audio input from PWM RC filter
RIN / R	Not connected	Unused right audio input
L+	Speaker +	Left speaker positive output
L-	Speaker -	Left speaker negative output
R+	Not connected	Unused right speaker positive output
R-	Not connected	Unused right speaker negative output
SW	+5V through 10k resistor	Amplifier enable / switch

Quick Demonstration

Here (redirect to YouTube)

This video demonstrates the ability to read, generate and alter the musical note selected of the board using the amplifier, speaker and screen to confirm it.

The buttons are not implemented - as of 15th of May.

Final Wiring

First look

Side view

Resistor ladder

Bad solder job

Software Design

Introduction

Writing software for the ProtoTone was not easy, given the audio nature of the project. I encountered many bugs, choppy note playback due to full RAM, random sounds because of the ADC fluctuations, non-stopping notes because of bad conditions, just to list some. I hope I can demystify the black box that I created in this chapter.

For development I used VSCode, Platform IO, Arduino and AVR. The full code is available on my Github.

Libraries Used

Library	Role in the project
Arduino.h	Provides the Arduino core functions used in the code, such as pinMode(), digitalWrite(), analogRead(), delay(), delayMicroseconds(), millis() and Serial.
Wire.h	Provides I2C communication. It is used for communication between the ATmega328P and the OLED display.
Adafruit_GFX.h	Provides general graphics and text drawing functions for the display.
Adafruit_SSD1306.h	Provides the driver for the SSD1306 OLED display. It initializes the screen and sends the framebuffer data to the display.
avr/interrupt.h	Provides AVR interrupt support, including the ISR() macro used for the Timer2 audio interrupt.
util/atomic.h	Provides ATOMIC_BLOCK(), used to update shared audio variables safely between loop() and the Timer2 interrupt.
math.h	Provides mathematical functions such as sinf() and powf(), used for vibrato, pitch bend and octave frequency calculations.

Labs Used

Topic	How it is used in the project
GPIO	Used for reading the control buttons: octave down, octave up, waveform select and hold.
ADC	Used for reading all analog inputs: note selector ladder, pitch bend joystick, vibrato potentiometer and volume potentiometer.
Timers	Timer1 is used for generating the high-frequency PWM audio signal. Timer2 is used as the audio sample timer at 16 kHz.
PWM	Used to generate the audio output signal on PB1 / OC1A. The duty cycle is updated continuously to create square, saw and triangle waveforms.
Interrupts	Timer2 Compare Match interrupt is used to generate audio samples periodically.
I2C / TWI	Used for communication between the ATmega328P and the SSD1306 OLED display.
UART / Serial	Used for debug messages, such as initialization status, button presses and detected notes.
External components / interfacing	Used for connecting the OLED display, PAM8403 audio amplifier, speaker, potentiometers, joystick and stylus note keyboard.
Analog signal processing	Used through the resistor ladder for note detection and through RC filtering on the PWM audio signal before the amplifier.

So I used everything from the labs except SPI.

Code

This section aims to explain in detail the main loop of the code.

First, we take the converted analog value of the current note (that's only if the stylus is currently

touching a note), pitch, vibrato and volume.

```
notesAdc = smoothAnalogRead(NOTES_PIN);
pitchAdc = smoothAnalogRead(PITCH_PIN);
vibratoAdc = smoothAnalogRead(VIBRATO_PIN);
volumeAdc = smoothAnalogRead(VOLUME_PIN);
```

The function used takes the average of multiple consecutive inputs to smooth it.

```
// Get median of reads
uint16_t smoothAnalogRead(uint8_t pin) {
    uint32_t sum = 0;

    // Add up 12 consecutive reads
    for (uint8_t i = 0; i < 12; i++) {
        sum += analogRead(pin);
        delayMicroseconds(120);
    }

    // Get median
    return sum / 12;
}
```

Then we read the stabilized note. That's a note that the board is sure is touched with the intent of it being played, not random noise of an ADC misfire.

```
int detectedNote = detectNoteStable(notesAdc);
```

The steps to detect a stable note go like this:

1. Use static variables to remember the last state

```
    // Current stabilized note
    static int lockedNote = -1;

    // How many consecutive reads are in the air
    static uint8_t airCounter = 0;

    // Potential new note
    static int candidateNote = -1;

    // How many times in a row we-ve seen the candidate
    static uint8_t candidateCounter = 0;
```

2. Check if stylus is in the air and if it is, turn off playback after multiple air reads

```
    // ADC below threshold = air
    // If stylus doesn't make good contact for AIR_RELEASE_COUNT reads the
sound doesn't stop
    if (adc < AIR_THRESHOLD) {
```

```
        if (airCounter < AIR_RELEASE_COUNT) {
            airCounter++;
            return lockedNote;
        }

        // Reset
        lockedNote = -1;
        candidateNote = -1;
        candidateCounter = 0;
        return -1;
    }
```

3. If stylus is not in the air, look for the nearest note

```
int nearestNote = detectNearestNote(adc);
```

```
// Clamp read adc value to nearest note
int detectNearestNote(uint16_t adc) {
    int bestIndex = -1;

    // Maximum difference
    int bestDiff = 1024;

    for (int i = 0; i < 12; i++) {
        int diff = abs((int)adc - noteAdcExpected[i]);

        if (diff < bestDiff) {
            bestDiff = diff;
            bestIndex = i;
        }
    }

    if (bestDiff > 90) {
        return -1;
    }

    return bestIndex;
}
```

4. If we don't currently have a locked note, look for one by constantly checking who the candidate is and how many times we've seen hiom in a row. After enough times, he becomes the current note.

```
    if (lockedNote < 0) {
        // If we've seen this candidate we increment the counter
        if (nearestNote == candidateNote) {
            candidateCounter++;
        // If not, this is the candidate now
        } else {
            candidateNote = nearestNote;
            candidateCounter = 1;
```

```
        }

        // If we've seen the candidate enough times in a row, it becomes our
locked note
        if (candidateCounter >= NOTE_CONFIRM_COUNT) {
            lockedNote = candidateNote;
            candidateNote = -1;
            candidateCounter = 0;
        }

        return lockedNote;
    }
```

5. So we already have a note but another one has come up. First we calculate how close the raw ADC value is to the nearest note and to the current locked note.

```
    int nearestDiff = abs((int)adc - noteAdcExpected[nearestNote]);
    int currentDiff = abs((int)adc - noteAdcExpected[lockedNote]);
```

```
    // Hysteresis: we switch only if the new note is SWITCH_MARGIN closer to
the adc value to prevent false switches
    if (nearestNote != lockedNote && nearestDiff + SWITCH_MARGIN <
currentDiff) {
        if (nearestNote == candidateNote) {
            candidateCounter++;
        } else {
            candidateNote = nearestNote;
            candidateCounter = 1;
        }

        if (candidateCounter >= NOTE_CONFIRM_COUNT) {
            lockedNote = candidateNote;
            candidateNote = -1;
            candidateCounter = 0;
        }
    } else {
        // No clear switch, destroy candidate
        candidateNote = -1;
        candidateCounter = 0;
    }
```

The top 2 values are used to prevent false switches. We have a safe SWITCH_MARGIN and we only switch to the new note if the ADC is that close to the new note.

Imagine we have to decide between to neighbour notes, E and F. E is at 478 while F is at 413.

If the ADC value fluctuates at the center: 446 → E, 444 → F, 447 → E, 443 → F, the output jumps between them. With the hysteresis step, we make sure the ADC value is much closer to the intended new note before we switch, preventing false switches.

The rest is updating the candidate or locking the candidate in.

6. If we are pressing on the same locked note, keep returning it.

Moving forward, after detecting the stable note, set the current note for playback, if HOLD is enabled we set the held note as well. On the other hand, if no stable note is found and HOLD is enabled and we have a held note set then set it as the current note for playback.

```
    // If we got a stable note, set current note
    if (detectedNote >= 0) {
        currentNote = detectedNote;

        // Set held note if hold is enabled
        if (holdEnabled) {
            heldNote = detectedNote;
        }
    } else {
        // If no note is detected and hold is enabled and we have a held
note set, set current note
        if (holdEnabled && heldNote >= 0) {
            currentNote = heldNote;
        } else {
            // No note :(
            currentNote = -1;
        }
    }
```

After we have the note I check the buttons

```
    // Go down an octave
    if (pressedOnce(BTN_OCT_DOWN)) {
        if (octaveOffset > -1) octaveOffset--;
    }

    // Go up an octave
    if (pressedOnce(BTN_OCT_UP)) {
        if (octaveOffset < 1) octaveOffset++;
    }

    // Cycle through waveforms
    if (pressedOnce(BTN_WAVEFORM)) {
        waveform = (waveform + 1) % 3;
    }

    // Toggle hold
    if (pressedOnce(BTN_HOLD)) {
        holdEnabled = !holdEnabled;

        // Clear held note
        if (!holdEnabled) {
            heldNote = -1;
        }
```

```
    }
```

Now it's time for the final processing of the note.

We need the frequency of it to alter it. Here they are, just need to select the right one

```
// Notes frequencies
const float noteFreqs[12] = {
    261.63, 277.18, 293.66, 311.13,
    329.63, 349.23, 369.99, 392.00,
    415.30, 440.00, 466.16, 493.88
};
```

We select it based on the current note:

```
        // Get target frequency for the current note
        float baseFreq = noteFreqs[currentNote];
```

Now it's time to manipulate it.

Based on the octave, we half it, or double it:

```
        // Double or half the frequency based on the octave offset (-1, 0 or
1)
        baseFreq *= powf(2.0f, octaveOffset);
```

The joystick I use as a pitch shifter goes between 0 and 1023 after being DAC'd. That means that when it's idle, it shows 512 digitally. I center it in 0 to be more clear.

PITCH_DEADZONE is used to ignore small fluctuations in the joystick movement which can be unintentional, like the ADC fluctuating.

After that, the number of bend semitones is mapped from (-512, 512) to (-12, +12) to achive a full octave worth of bending.

```
        // Center pitch in 0
        int pitchCentered = (int)pitchAdc - 512;
        float bendSemitones = 0.0f;

        if (abs(pitchCentered) > PITCH_DEADZONE) {
            bendSemitones = (pitchCentered / 512.0f) * 12.0f;
        }
```

Time to apply vibrato. We will generate a sinusoidal oscillation for it based on the following formula:

```
y = sin(2 * PI * f * t) * A
A = amplitude
f = frecuency
t = time

float vibrato = sinf(2.0f * PI * vibratoRate * t) * vibratoDepth;
```

```
amplitude = vibratoDepth
frequency = vibratoRate
period = 1 / vibratoRate
```

For depth, we map the DAC'd value from (0, 1023) to (0, 0.5).

```
float vibratoDepth = (vibratoAdc / 1023.0f) * 0.5f;
```

Vibrato rate, I set it to 5Hz.

```
float vibratoRate = 5.0f;
```

Get time in seconds

```
float t = millis() / 1000.0f;
```

Final vibrato value based on formula:

```
float vibrato = sinf(2.0f * PI * vibratoRate * t) * vibratoDepth;
```

Now to add them all up to get the final frequency.

If we need 12 semitones to double the frequency, then each semitone multiplies the frequency with a factor `r`. So `r^12 = 2`. That means that `r = 2^(1/12)`. For n semitones, that's `r = 2^(n/12)`.

We add vibrato and bend together because they are both expressed in semitones. the `n` from above.

So final frequency is:

```
finalFreq = baseFreq * powf(2.0f, (bendSemitones + vibrato) / 12.0f);
```

The next part is important. We will update the variables that are used for actually producing the sound.

Because they are used in the interrupt routine, it's important to disable interruptions while we do the update. Here, we set the incrementation that should be used to navigate the waveform (I will explain this below) and the volume. If there's no active note, no progression is made. which means silence.

```
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
        // Set if note is active
        noteActive = active;

        // Set increment in waveform
        phaseInc = active ? freqToPhaseInc(finalFreq) : 0;

        // Set volume
        audioVolume = active ? vol : 0;

        // If not active, set default
        if (!active) {
```

```
            OCR1A = 128;
        }
    }
```

Sound Generation after We Have the Target Frequency

Alright, we read analog inputs, detected the note, applied transformations and now we have the final frequency which we want to be played in a specific moment. How is it actually done?

Phase Incrementation

Let's calculate `phaseInc` which tells us how fast we are moving through the waveform. This is what actually dictates what note we hear. The bigger `phaseInc`, the higher pitched the output. The smaller it is, the lower the pitch.

```
phaseInc = active ? freqToPhaseInc(finalFreq) : 0;
```

```
uint32_t freqToPhaseInc(float freq) {
    return (uint32_t)((freq * 4294967296.0f) / SAMPLE_RATE);
}
```

```
                                    ^
                                    |
                                  2^32
```

We have another variable, which is a 32 bit number `phaseAcc`. This one holds the position (total of `phaseInc`'s) in the whole waveform.

We want `finalFreq` cycles of the waveform per second, one cycle is 2^32 steps (because we chose `phaseAcc` as 32 bits) so in one second we need to go `finalFreq * 2^32` steps. Sample rate is 16kHz, so we make 16000 adjustments per second. To get the final `phasInc`, we divide by that. That's where the formula comes from.

Sample Timer

For the sampling of audio, I used `Timer2`.

```
// Set Timer2 to generate an interruption at 16kHz
void setupSampleTimer() {
    // Reset Timer2
    TCCR2A = 0;
    TCCR2B = 0;
    TCNT2 = 0;
```

```
    // Value at which we compare match
    OCR2A = 124;

    // Set timer to CTC
    TCCR2A |= (1 << WGM21);

    // Set prescaler at 8 so ISR runs at 16kHz
    TCCR2B |= (1 << CS21);

    // Activate interruption on compare match
    TIMSK2 |= (1 << OCIE2A);
}
```

We run it in Clear Timer on Compare mode to generate an interruption that actually generates the waveform.

We are aiming for a 16kHz sample rate to have good sound fidelity, but not too much to overwhelm the microcontroller.

Based on the formula:

```
f_ISR = F_CPU / (prescaler * (OCR2A + 1))
```

With the CPU running at 16MHz, we can set the prescaler to 8 and `OCR2A = 124` to achive the desired `f_ISR = 16.000Hz`

Generating the Waveform

The interruption is happening now 16k times per second. The routine will calculate where it currently is in the selected waveform and set `OCR1A`, the PWM duty cycle, to the right value.

```
// Interrupt routine that generates the sample
ISR(TIMER2_COMPA_vect) {
    // If no note is active or volume is 0 we set duty cycle to constant
    if (!noteActive || audioVolume == 0) {
        OCR1A = 128;
        return;
    }

    // Go further in waveform by phaseInc determined by current note
    phaseAcc += phaseInc;

    // phaseAcc is on 32 bits for accuracy, while our waveform is on 8, so
we take the 8 most significant bits
    uint8_t p = phaseAcc >> 24;

    // Raw value of the wave
    uint8_t raw;
```

```
    switch (waveform) {
        // Square wave: first half is up, second is down
        case 0:
            raw = (p < 128) ? 255 : 0;
            break;

        // Saw wave: goes up linearly then jumps to 0
        case 1:
            raw = p;
            break;

        // Triangle wave: goes up linearly to max then descends linearly to
0
        case 2:
            raw = (p < 128) ? (p * 2) : (255 - ((p - 128) * 2));
            break;

        default:
            raw = 128;
            break;
    }

    // Transpose signal from 0:255 ro -128:127 for volume to scale the
amplitude around the center, not to drag everything towards 0
    int16_t centered = (int16_t)raw - 128;
    int16_t scaled = (centered * audioVolume) / 255;

    // Set signal back to original interval
    OCR1A = (uint8_t)(scaled + 128);
}
```

See comments.

Transforming the Sample into Sound

For the last step, we are using Timer1 in Fast PWM 8-bit mode. The configuration is the following:

```
// Setup Timer1 to generate audio on AUDIO_PIN
void setupAudioPWM() {
    // Switch pin to output
    pinMode(AUDIO_PIN, OUTPUT);

    // Reset Timer1
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1 = 0;
```

```
    // Set Timer1 non-inverting PWM on OC1A
    TCCR1A |= (1 << COM1A1);

    // Set Fast PWM 8-bit
    TCCR1A |= (1 << WGM10);
    TCCR1B |= (1 << WGM12);

    // Select prescaler 1 => f_PWM = F_CPU (prescaler * 256) = 62.5kHz
    TCCR1B |= (1 << CS10);

    // Set a default duty cycle
    OCR1A = 128;
}
```

So, Timer1 doesn't 'create' sound. Timer2 changes OCR1A 16.000 times per second, which creates the sound we hear.

Optimizations

- Used timers instead of software polling which is costly
- ISR instruction is short and doesn't do expensive calculations
- Sample rate is a compromise between speed and fidelity
- Every variable type is exactly what it needs to be, no heap wasted
- Display not updated every loop

Known Bugs

- Keys don't register immediately
- Note keeps ringing a bit after lifting stylus
- ADC fluctuations in volume, vibrato and pitch mess constantly with the out tone
- Pitch is kind of choppy and hard to control
- A lot of noise while on standby

Big Final

I am very proud to present you the full ProtoTone synth. It was a difficult project and the first one I made out of real hardware by myself. Every functionality I planned on is here, almost no compromises, except the noisy audio which is hard to fix with cheap components on breadboards :)

Watch the full demo here!

Case design credits go to my wife

Bibliography

- Lab 3 PM OCW
- Secrets of Arduino PWM - https://www.righto.com/2009/07/secrets-of-arduino-pwm.html
- Phase Accumulators - https://www.digikey.com/en/articles/the-basics-of-direct-digital-synthesizers-ddss
- Simple MIDI synthesiser using an AVR ATmega328 (2-operator FM) - https://www.youtube.com/watch?v=Zq1o3Phj4Wo
- Post-Box Synthesizer - https://www.instructables.com/Post-Box-Synthesizer/