Microcontroller Sound: Additional Projects
Joseph Kramer

These projects build on the introductory ones included in the book.

CIRCUIT 2: Pseudo-analog effects, tones without delay, and notes quantized to scales.

OVERVIEW

The following circuit expands on the digital oscillators described in the book by adding a second knob and a volume control (via a light-dependent resistor paired with an LED). By controlling the brightness of the LED using pulse width modulation (PWM), intermediate volume levels can be achieved. The basic principle is explored in the first sketch, then expanded in the second sketch using the tone() function. Arrays are introduced in the third sketch, allowing for the playback of pre-programmed melodies with dynamic volumes. Finally, a simple quantizer is created to constrain notes performed via knob turns to desired scales.

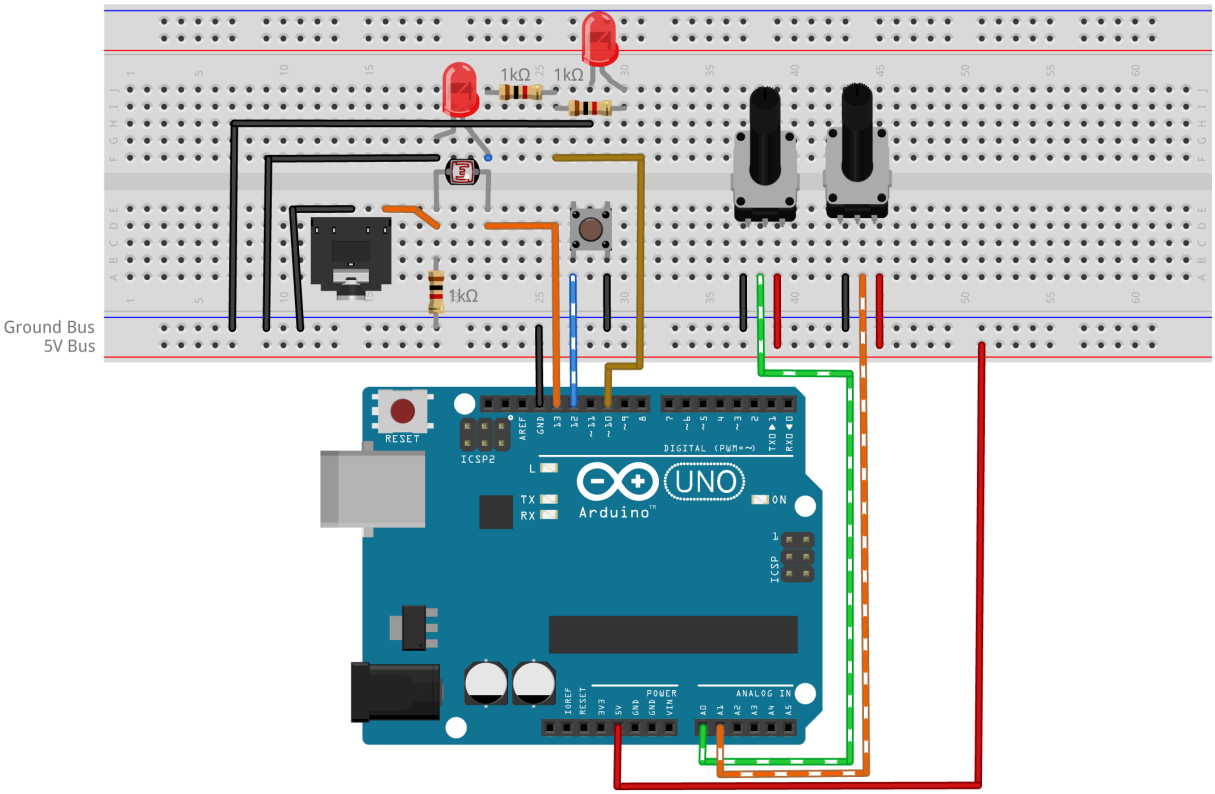

Figure 3: Fritzing breadboard image of digital oscillator with volume control circuit

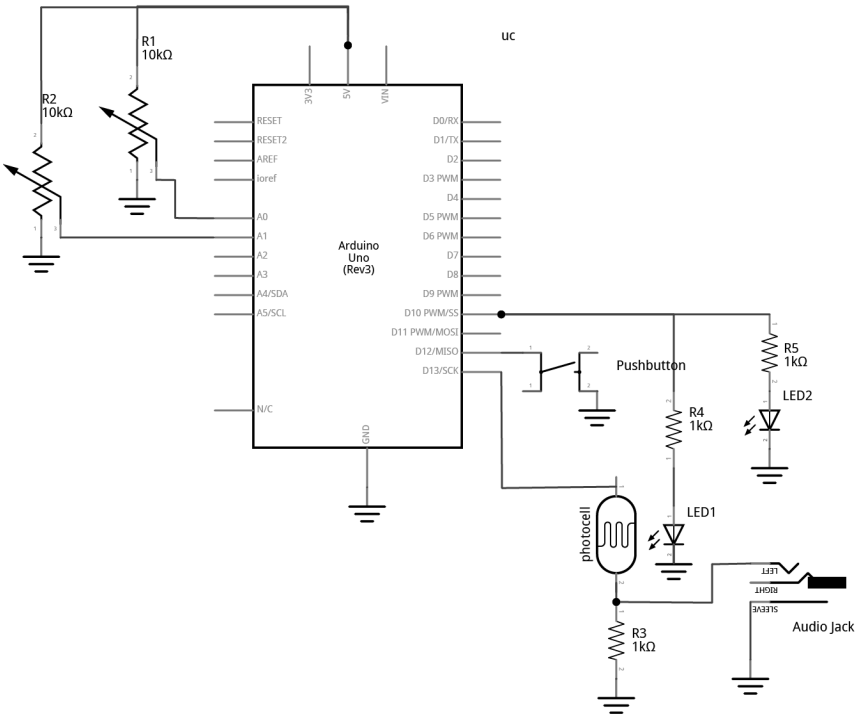

Figure 4: Fritzing schematic of digital oscillator with volume control circuit

You will need:

- An Arduino Uno.
- A computer running the Arduino IDE software.
- A USB cable that can connect from your computer to the Uno's USB type-B connector.
- A breadboard.
- Some solid hookup wire or premade wire jumpers.
- Assorted resistors (1kΩ – 100kΩ is a good range).
- A 10kΩ linear potentiometer.
- A pushbutton or toggle switch.
- Some photoresistors.
- Some LEDs (Light Emitting Diodes).
- An audio amplifier.
- Assorted jacks and plugs, to match your amplifier.
- Hand tools.

CONNECT THE HARDWARE

1. Open the Fritzing file named *C1_ButtonOSC.fzz* and navigate to Breadboard view
2. Insert all components into breadboard matching the image
3. Connect all wires on breadboard
4. Connect wires from Arduino to breadboard
5. Visually double-check all connections and wiring

6. Connect the Arduino Uno to your Computer

PROJECT ONE: Volume Control

Program the Arduino with the VolumeOSC sketch:

1. Open the Arduino sketch named *HEM_VolumeOSC.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

This sketch provides a method for creating analog-like effects using a technique called *Pulse Width Modulation (PWM)*. This is done with the analogWrite() function, but is (confusingly) only possible on the *digital* pins. This is because, technically, PWM is still a digital signal. PWM works by toggling a pin high and low at a fixed frequency while adjusting the percentage of time that the wave is HIGH versus the time it is LOW in order to deliver less current than if the pin were just held at a constant HIGH level. This allows the LED to be set to intermediate levels of brightness between all the way ON (analogWrite(pin, 255) - PWM signal is HIGH 100% of the time), to a medium brightness (analogWrite(pin, 128) - PWM signal is HIGH 50% of the time), to all the way off (anlaogWrite(pin, 0) - PWM signal is HIGH 0% of the time). Take note that not all of the digital pins can generate PWM. Only the six pins with the tilde (~) symbol next to them are capable of this function.

We are using PWM to control the brightness of two LEDs from one pin. One of the LEDs is used for visual feedback, while the other is paired with a photocell with the intent to control volume with light. The benefit of using PWM is that in addition to simple binary blinking, we are able to program the brightness level smoothly from fully bright to completely dark. When using this technique to control volume the effect can imperfect (you may hear some of the PWM frequency in your signal – this can be improved by using a different microcontroller with a higher PWM frequency), but with some finessing of the led-photocell pairings the technique opens up fruitful avenues of experimentation.

This sketch uses a button and two knobs to control the activation, pitch, and loudness of an oscillator. The knob on pin A0 controls the pitch by setting the period of the waveform via the delayMicroseconds() argument. (This method of sound generation is described in an earlier project.) When the button is pressed, the knob connected to pin A1 sets the brightness of the LED in the photocell/LED pair. This brightness controls the volume of the oscillator via the voltage divider created by the photocell and fixed resistor pairing. When the button is released, the LED turns off and the note fades out at the photocell's response speed. You may need to experiment with the value of the fixed resistor (1kΩ in the example) to get the best response from your photocell. (It may be useful to start with a potentiometer in place of the 1kΩ fixed resistor to dial in the value that works best for your photocell/LED pair. The technique for using a potentiometer to determine the value of a fixed resistor is described in the book.)

If your photocell causes a fade out due to a slow response time then you are likely to notice that the pitch of the oscillator changes when the button is released. This happens because the

tone is generated by pausing at points in the loop using *delays*. But delays are not the only things that the microcontroller has to do, and each instruction adds time between toggles. This means the frequency can be affected by how long it takes for the microcontroller to get through the other instructions in the loop. When the button is pressed, the *if* statement is engaged and more lines of code have to be executed by the microcontroller compared to when the button is not pressed. As a result, there is more time between the pin going LOW and the pin going HIGH again. This lowers the pitch of the microcontroller slightly when the button is down. One approach to stabilizing the frequency is to make sure that the *if* statement and the *else* statement take the same amount of time by moving some instructions out of that structure and into the main loop. But a better way to handle this issue would be to do away with the delay() method of tone generation altogether and instead make use the tone() function built into the Arduino library.

PROJECT TWO: Volume Control with Improved Timing via tone()

Program the Arduino with the VolumeOSC_tone sketch:

1. Open the Arduino sketch named *HEM_VolumeOSC_tone.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

Using the tone() function eliminates the need to employ the cumbersome delay() function for our audio task. To start a note using tone() we will call the function and pass it two arguments: which pin, and what frequency. If we want stop the tone, we can call the complementary function noTone() which will take one argument to clarify which pin to silence. The tone() function can produce frequencies from about 31Hz to nearly 5kHz. The map() function is used here to select a range from 110Hz to 1760Hz. These numbers were chosen by ear and can be adjusted to taste. For more ways to use tone() see the built-in examples at **File>Examples>Digital>[tone…].**

Keeping the audio generation free of delays frees us to do useful things in the rest of the loop without affecting the pitch of the oscillator. Examples of useful things might include blinking lights, reading sensors, moving through sequences of notes, etc. This is demonstrated using this circuit and the tone() function to create a simple note sequencer in the next project.

PROJECT THREE: Stored Melodies with Dynamics

Program the Arduino with the ToneSEQ sketch:

1. Open the Arduino sketch named *HEM_ToneSEQ.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

What you should hear is a repeating sequence of notes that play at different volumes. Turning one knob will change how quickly the notes are played and the other knob will change the

number of notes in the sequence. Hold the button will cause the sketch to output an alternate sequence of notes.

This sketch makes use of *iteration* and *arrays*. We created this list of tones in a data structure called and *array*. The *array* is declared at the start of the code just like any other variable. Start by writing the data type, followed by a space and then a name of our choosing. This name is followed immediately by brackets enclosing a number that represents how many items will be in our array. If the number in the brackets is 16, then we would follow that with a list of 16 values, separated by commas, and enclosed in curly braces. End the line with a semicolon.

To access a value from the list, type the name of the array with the number of the item in the brackets. This is known as the *index* of the array. Note that counting starts at zero, so the index of the first value is 0, and the index of the last item is the number of items minus one. We created the variable "i" which we will use to select items from the list.

The variable "i" is used in this case to keep track of which number to pass the tone() function from a list of tones. We iterate, or count up through the list, by adding one to the variable "i" each time we go through the loop. The instruction i++; tells the program to add 1 to the current value of i. When we get to the last numbered item, we reset "i" to zero to start again at the beginning of the array.

```
if (i > stepNum) {
   i = 0;
  }
```

The list of volumes is stored in a simple array. This is a list of PWM values between 0 – 255 used to control the brightness of an LED. The list of tones, called "thisTone[][]" in the code, uses a two-dimensional array. This allows us to pick our notes from one of two lists in order to allow for alternate melodies. The array has two sets of brackets which can be thought of as representing rows and columns, respectively. A number in first bracket will select the row, or which list to play tones from. A number in the second bracket will select which tone to play from the selected list. The numbers in the lists represent the rough frequencies of different pitches. For a list of the pitches see the example files in Arduino at **File>Examples>02.Digital>toneMelody.** There is a second tab in that sketch window called pitches.h which lists available values. Alternatively, search the internet for the frequencies of musical notes.

To play the notes, call the tone() function and provide it with two arguments: which pin to play the tone on, and what frequency to play. Calling the following line of code will play 880 (note A5), the first tone from the first list:

```
tone(ledPin, thisTone[0][0]);
```

To play the tone 659 (E5), the 5th item on the 1st list, you would call:

```
tone(ledPin, thisTone[0][4]);
```

There are only two built-in sequences in this array, but you can experiment with adding more. To create a third list, simply change the number in the first bracket of the variable declaration at the top of the code from 2 to 3 and create a new list of numbers. Here is an example:

```
//create an array to store frequencies
int thisTone[3][8] = {
  {880, 41, 33, 494, 659, 1047, 62, 73},
  {41, 33, 33, 494, 247, 880, 880, 73},
  {659, 1047, 1047, 1047, 659, 1047, 1047, 1047},
};
```

This simple sequencer has many possible variations and could be used to code longer melodies or even entire songs. Another avenue of experimentation would be to lose the notes entirely and only keep the volume control. Connecting an audio jack to the photocell volume control instead of the internal oscillator would allow for the creation of a programmable audio slicer. Alternatively, the note arrays could be kept, but the stepping function removed and the code revised to play tones and volumes selected by knobs or some other novel controller. For example, a quantized Theremin-like instrument could be created using distance sensors to select pitches and volumes. The next example will walk through the creation of a simple tone quantizer.

PROJECT FOUR: Quantizer for Digital Oscillator

Program the Arduino with the ToneQUANTIZER sketch:

1. Open the Arduino sketch named *HEM_ToneQuantizer.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

What you should observe is that pressing the button causes the light to turn on and a note to be played. Releasing the button turns off the LED and silences the note. Turning the note knob (on pin A0) will cause the pitch of the note to change. Rotating the knob from one end to the other will play through the entire selected scale. Turning the second knob will change the type of scale that is selected. In this example, the scales are chromatic, major, natural minor, major pentatonic, and minor pentatonic. (Though many other scales are possible.)

A quantizer takes a set of incoming values and maps them to a desired set of outgoing notes. In this case, the incoming values are just a range of numbers that result from our program reading a knob (the 10-bit analogRead() reports a range of numbers between 0 – 1023). The desired outgoing notes of our quantizer will be selected from a list of frequencies belonging to the same musical scale. These frequencies will be passed to the tone() function. It is a bit like sliding one's finger across only the white keys of a piano to play C major. Only this implementation of a quantizer is able to select from a large number of different keys and scales.

To understand how this is accomplished, we will start by examining the tone() function. As discussed in previous examples, the tone() function takes two arguments: 1. which pin to play the tone on, and 2. the frequency of the tone to produce. The frequencies that the function can produce are limited to the relatively wide range of 31Hz (note B0) to 4,978Hz(D#8). When tone() was used in previous examples, a knob was read and the value (0 – 1023) was mapped to some desired range within that 32 – 4987 Hz limit. Consider the following code fragment:

```
int rawNote = analogRead(noteKnob);
int frequency = map(rawNote, 0, 1023, 31, 4978);
tone(tonePin, frequency);
```

With this example, turning the knob would result in a relatively smooth slide between pitches, with many octaves being moved through on the low range of the knob, and only about one high octave spread across the whole right-hand side of the knob. This is pretty classic oscillator behavior and certainly great fun, but it can be pretty difficult to play a specific note.

To spread the frequencies more evenly across the knob range, and only play notes that roughly correspond to notes in Western music, an array is defined. The first array, chromaticScale[89], is a large list of every possible note in 12-tone equal temperament that the tone() function is capable of producing. There are 89 frequencies in the list that correspond nearly, but not exactly, to the frequencies of the notes on a standard modern piano. (The range of a grand piano is very slightly different, and most notes in Western music have frequencies with decimal point components which tone() is not capable of producing.) If all we wanted was a chromatic scale, we could simply read an analog input and scale the result *from* the analogRead range of 0 – 1023 *to* the chromaticScale[89] range of 0 – 88 using the map() function. Using the output of that map function as the index of the chromaticScale[89] array would allow us to directly select from that list of notes. The code fragment below would quantize our oscillator to only play notes from the chromatic scale:

```
int rawNote = analogRead(noteKnob);
int pitchNum = map(rawNote, 0, 1023, 0, 88);
note = chromaticScale[pitchNum];
tone(tonePin, note);
```

It would be possible to just hard code arrays for every possible scale we wanted to play, but it would be impractical to create such lists for all desired scales. To accommodate a more flexible system that can be used to select different scale types (like major, minor, pentatonic, etc) and different tonics (for example A, Bb, F#, etc) this example implements a set of *masks*. These masks are arrays of index values for given scales, and offsets for given tonics and octaves. The numbers in the majorMask[ ] array, for example, contain the index values for the first octave of the B major scale if applied directly to the chromaticScale[ ] array. The tonicMask[ ] is a bit unnecessary as it is a simple sequence, but it would allow for more flexible access to the notes if a different system was desired.

Our first scale, the chromatic scale, is the easiest to understand as it just plays through each possible note, starting with the root and ending with the root an octave above. So to play one octave of a chromatic scale in the key of B, we just need to pass the chromaticScale[89] array an index value from 0 – 12. If we wanted to shift the root from B to C in order to play a C scale, we would just add one to each item in the index, causing the output to play notes 1 – 13. If we wanted to shift to a root of F, we would add 6 (notes 6 – 18). The code fragment below would play one octave of the F chromatic scale:

```
int root = 6;
int rawKnob = analogRead(noteKnob);
int degreeIndex = map(rawKnob, 0, 1023, 0, 12);  //this maps the knob so it only reads one octave of
pitchNum = degreeIndex + root;
note = chromaticScale[pitchNum];
tone(tonePin, note);
```

In this case, our knob gets mapped to a variable with a range of 0 – 12, and the instruction pitchNum = degreeIndex + root; just adds a 6 to each degreeIndex value. pitchNum is then used as the index of the chromatic scale array. This causes the notes that get played to be shifted up in the list by 6. This simple mechanism allows us to use the one large list of notes to generate a chromatic scale of any number of octaves starting at any note.

To get a different octave, we can add an octave mask. New octaves are 12 index values apart. So, if we want to start on the first octave, we add an offset to the index of zero. For the next octave, we add an offset of 12. Next, an offset of 24, and so on. We can add the octave offset as follows:

```
cost byte octaveMask[] = {0, 12, 24, 36, 48, 60};
int root = 6;
int octave = 2;
int rawKnob = analogRead(noteKnob);
int degreeIndex = map(rawKnob, 0, 1023, 0, 12);  //this maps the knob so it only reads one octave of
pitchNum = octaveMask[octave] + degreeIndex + root; //this gives the octave, scale degree and root
note = chromaticScale[pitchNum];
tone(tonePin, note);
```

Try different values of root and octave to hear different chromatic scales at different octaves. You can also try adding knobs to directly select the octave and the root by reading knobs and mapping their values to desirable ranges.

The second knob in this example, however, is used to select from a set of 5 different scale types. Each of these scale types has a unique mask that can be used to pick appropriate notes from the main chromatic scale array using the simple shifting and masking techniques described so far.

Using the example of a C major scale to illustrate how this works may help circumvent too much discussion about music theory (though a thorough understanding of that subject may

help with comprehension and expanding the available scale types). The C major scale starts on the note C and only includes the white keys. The index values of the white keys starting on C are 1, 3, 5, 6, 8, 10, 12, and 13 brings us back to C. (figure 2)

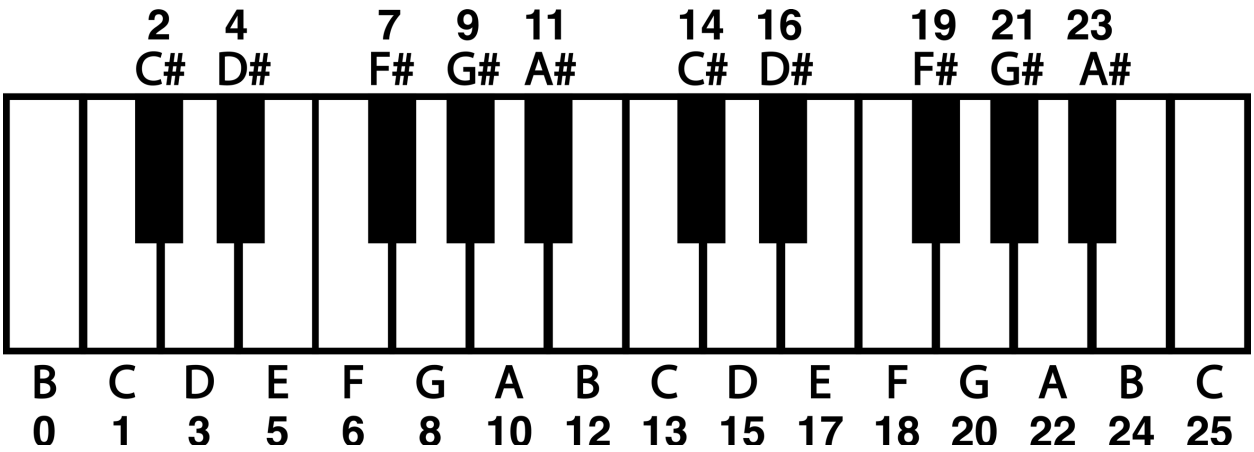

Figure 5: Piano keyboard with note names and index values

To make a mask for C Major, we would simply use those numbers as the index of our chromatic scale array. To use a knob to play through the first octave of C Major, we could use the following code fragment:

```
const byte octaveMask[] = {0, 12, 24, 36, 48, 60};
const byte cMajorMask[8] = {1, 3, 5, 6, 8, 10, 12, 13};

int octave = 2;
int rawKnob = analogRead(noteKnob);
int i = map(rawKnob, 0, 1023, 0, 7);  //note, only 8 notes in one octave of a major scale
int degreeIndex = cMajorMask[i];
pitchNum = octaveMask[octave] + degreeIndex; //this gives the octave, scale degree and root
note = chromaticScale[pitchNum];
tone(tonePin, note);
```

It would again be impractical to make a mask for every possible major scale at every tonic. So, we need to adjust this mask to make it usable for major scales starting on any note. To do that, we need to make the mask start at zero. That way the scale with a root of C is accessed by using the degree index plus a root offset of 1. So, if we subtract one from each item in the array, we get a major scale mask that starts at zero and can be offset to start at any note by changing the root value, just like the chromatic scale mask in the previous illustration. The resulting fragment will play a C major scale at the third octave. Change the root and octave to hear other major scales.

```
const byte octaveMask[] = {0, 12, 24, 36, 48, 60};
const byte majorMask[8] = {0, 2, 4, 5, 7, 9, 11, 12};
```

```
int root = 1;
int octave = 2;
int rawKnob = analogRead(noteKnob);
int i = map(rawKnob, 0, 1023, 0, 7);  //note, only 8 notes in one octave of a major scale
int degreeIndex = majorMask[i];
pitchNum = octaveMask[octave] + degreeIndex + root; //this gives the octave, scale degree and root
note = chromaticScale[pitchNum];
tone(tonePin, note);
```

The rest of this code establishes arrays for alternative scales and uses if statements to select from among the scales and respond to button presses. For more info on how if statements work, see earlier examples or check out the documentation built into Arduino's IDE and on the web.

This quantizer is a useful proof of concept and can be included in other projects to make flexible and performable interfaces. Quantizers can be particularly useful when connecting the microcontroller hardware to other digital instruments using MIDI (Musical Instrument Digital Interface). MIDI divides the frequency space into MIDI note numbers, which can be packed up in array and called using the same basic mechanism described here.

CIRCUIT 3: Four Digital Outputs for Rhythm Generation and Noisy Swarms

OVERVIEW

The following circuit creates four individual patterns of pulses visualized by four LEDs. Different patterns can be selected for the four channels using a potentiometer. The speed of the blinking is controlled by a second potentiometer and the length of the individual blinks are controlled by a third. These four channels are attached to four 1/8" output jacks that can be connected to other circuits to be used as gate signals (0v - +5V range) or connected to a mixer and used as linked polyphonic audio channels.

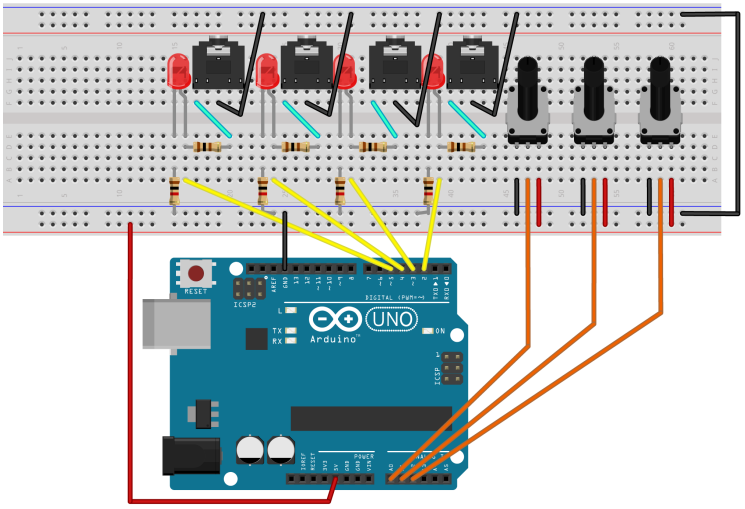

Figure 6: Fritzing breadboard image of rhythm generation circuit

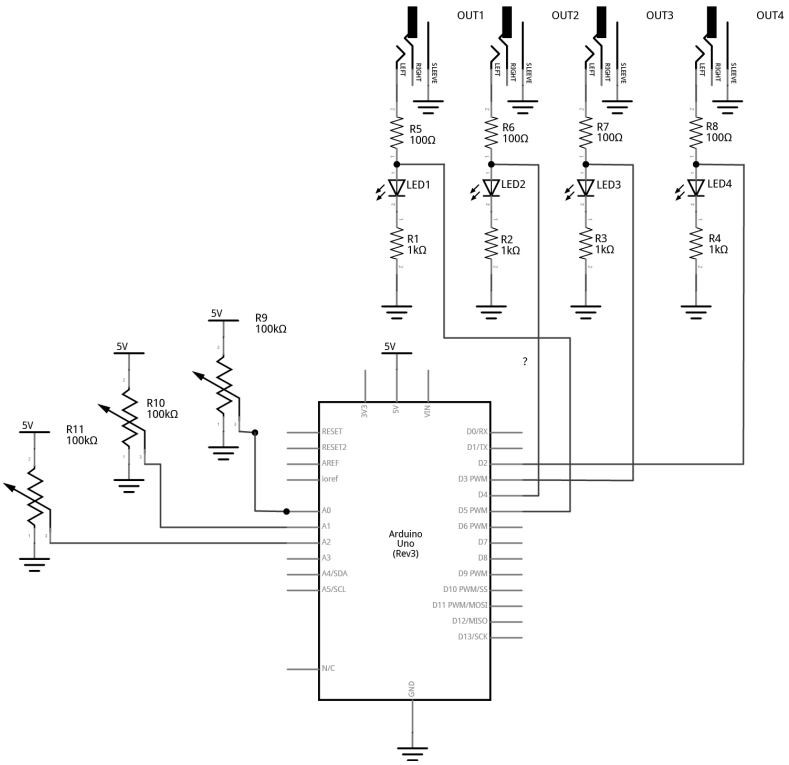

Figure 7: Fritzing schematic of rhythm generation circuit

You will need:

- An Arduino Uno.
- A computer running the Arduino IDE software.
- A USB cable that can connect from your computer to the Uno's USB type-B connector.
- A breadboard.
- Some solid hookup wire or premade wire jumpers.
- At least 4 100Ω resistors and at least 4 1kΩ resistors
- At least 4 LEDs (Light Emitting Diodes).
- An audio amplifier.
- Some kind of circuit that can respond to 5-volt gates/triggers.
- Assorted jacks and plugs, to match your amplifier.
- Hand tools.

CONNECT THE HARDWARE

1. Open Fritzing File named *C3_QuadQATES.fzz* and navigate to Breadboard view
2. Insert all components into breadboard matching image
3. Connect all wires on breadboard
4. Connect wires from Arduino to breadboard
5. Visually Double-check all connections and wiring

6. Connect the Arduino Uno to your Computer

PROJECT FIVE: Quad Gate Generator

Program the Arduino with the QuadGATES sketch:

1. Open the Arduino sketch named *QuadGATES.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

In this circuit, each of the four signal outputs are toggled HIGH and LOW by pre-programmed patterns (as visualized by the 4 LEDs). In the example, each of the four channels has 5 selectable patterns of 16 beats each. (There could be any number of patterns or beats, though.) There are three knobs that control the following: the speed of stepping, which pattern is selected for all channels, and how long each blink lasts in each step (When the blink time is all the way up, adjacent blinks tie together for longer notes).

The code starts with variable declarations as described in the comments within the code itself.

The setup() function repeats the instruction pinMode(pin, OUTPUT); four times to set each of the pins for the four channels to function as outputs.

The loop() is relatively short with the first three instructions reading the three knobs and storing those values in variables.

```
//Read Knobs and store raw values
int clockRaw = analogRead(clockKnob);
int patternRaw = analogRead(patternKnob);
int lengthRaw = analogRead(lengthKnob);
```

Then three more lines map those three variables to desired ranges for the speed, pattern number, and pulse length parameters.

```
//Scale Raw values to desired ranges using the map() function
stepTime = map(clockRaw, 0, 1023, 1000, 30); //set a clock from 1000ms between beats to 30ms
patternNum = map(patternRaw, 0, 1010, 0, PATTERN_COUNT - 1); //select from patterns numbered 0 - 4
trigLength = map(lengthRaw, 0, 1023, 5, stepTime); //select trig lengths from 5ms to a full beat length
```

Note that the stepTime variable is mapped to a range of 1000ms to 30ms. These values were chosen to taste. Experiment with by changing the last two numbers of the map() function to find a desirable range. Making the numbers larger will allow for slower tempo settings. Setting both numbers shorter will create noisy oscillator effects if channel outputs are connected to your mixer and amplified speaker.

CUSTOM FUNCTION 1: stepper()

The first major new concept in this code appears as a short instruction near the middle of the loop():

```
stepper(stepTime);
```

stepper() is not a built-in function of the Arduino language. This is a *custom function* that is created in this sketch. Custom functions are used relatively frequently and can be a convenient way to batch chunks of code into reusable pieces. I recommend heading over to the Arduino website's reference on custom functions to look at their structure as you explore this next bit of description.

The stepper() function definition happens after the end of the loop function. It appears as follows:

```
//master timer function keeps track of which beat to play and keeps a running timer
void stepper(unsigned long stepDur) {
  static unsigned long previousTime = 0;
  counter = millis() - previousTime;  //counter counts up from 0 to whatever the step time has been set to

  if (counter >= stepDur) { //if the counter reaches the stepTime, do the following:
    beat++;           //increment the beat by one
    previousTime = millis(); //update previousTime to right now
    if (beat > 15) {      //if the beat reaches 16, start over at 0
      beat = 0;
    }
  }
}
```

This function starts with a data type declaration (just like variable declarations). This one is declared "void". That means that whatever this function does, it is not going to report any new number for our sketch to use. Then we get to choose a name, in this case, "stepper".

Now, our function can have as many arguments as we want. These are parameters that we might want to be able to change while our loop is running. Just like tone() has an argument for the pin, and an argument for the frequency, we gave stepper() an argument for the step time (kind of like the delay time in the earlier Blink sketch, but better). By defining this parameter, we are enabling the stepping speed to be changed dynamically by sending it the value of a knob or some other input sensor.

Since we are going to use this to keep track of time, we declare the argument's data type as an unsigned long, and choose the name "stepDur", for step duration. Note, when we create arguments for custom functions, they must be declared just like variables, but they have to be unique to the function. We can't use a variable we already declared and pass that value. We have to make something new just for this function. To use this argument, we will add stepDur to an instruction later in the body of the function.

Next an open curly brace starts the body of the function. The first thing that happens in the function is the declaration of a variable called previousTime.

```
static unsigned long previousTime = 0;
```

We will use this variable to keep track of what the time was when we last took a step. This is critical to creating a time-base for our project without using delay(). This variable is an unsigned long because we are keeping track of the number of milliseconds since the project was powered on. This number will get quite large, and unsigned longs can store numbers just past four million. One thing that is new and noteworthy about this variable declaration is that it is declared as *static*. This is important, because this function gets activated (or called) each time the stepper() appears in the loop (including every time the loop repeats, which could be thousands of time per second). Each time a variable declaration happens, even declarations in custom functions, the variable gets set to the number it is listed as being equal to - in this case 0. However, we don't want to reset to 0 every time the function is called, so the word static tells the function not to reinitialize, or reset, the value of this variable every time the function is called.

In the next line the variable "counter" is used to store the number of milliseconds since the last step. It is a global variable, meaning it was declared at the beginning of the code, outside of any functions. This allows it to be used in any function in the body of the sketch. We update it here, and we check it in other functions.

```
counter = millis() - previousTime;
```

millis() is a function that returns the number of milliseconds since the program began. So millis() can be thought of as the current clock time. In the above line of code, the current clock – previousTime (which was set to zero at the start), will equal the current clock and it will continue to increase the number of counter until something causes previousTime to be updated.

The rest of the function happens in the following if statement:

```
if (counter >= stepDur) { //if the counter reaches the stepTime, do the following:
  beat++;           //increment the beat by one
  previousTime = millis(); //update previousTime to right now
  if (beat > 15) {      //if the beat reaches 16, start over at 0
    beat = 0;
  }
}
```

Here is where we use the stepDur argument. This statement checks to see if the counter (the running tally of the number of milliseconds since the last step) is greater than or equal to the stepDur argument that we passed to the function. If so, another global variable called "beat"

increments up by one (beat++;). Then our static variable previousTime gets set equal to the current program time in the line:

```
previousTime = millis(); //update previousTime to right now
```

This is how the counter will keep track of the time since the last step. Consider the following case: The device has just powered on and the program just started. millis() is counting up from zero. If our stepDur is 1000, then once the counter gets up to 1000, the beat variable will increase by one and then the above line of code will set previousTime equal to 1000. Since millis() just keeps counting up forever, the next time the function gets called it will be at some value greater than 1000, let's say 1002. Recall:

```
counter = millis() - previousTime;
```

If millis() is at 1002 and previousTime is at 1000, then the counter is now back down at 2 and ready to count back up to stepDur before resetting again. Counter is always the difference between the time now and the time of the last step. This is how we keep track of time without using the delay() function blocking all our code.

The last thing in our stepper function is inside of the previous if statement:

```
if (beat > 15) {     //if the beat reaches 16, start over at 0
  beat = 0;
}
```

The beat variable is going to be used to step through an array of 16 beats. Index values of 0 – 15 will address each beat in a pattern. This line of code says that if the beat is greater than 15, start over at 0. If patterns had 32 beats, this would be modified to read (beat > 31).

CUSTOM FUNCTION 2: trigger()

After the definition of the stepper function, a new custom function called trigger() is defined. This tells our program what to do at each of the four outputs.

```
//Custom Function to produce outputs of different lengths
//The switch…case structure looks for which Output number ("out")
//then assigns a pattern number(patternNum) from a 2D array
//then keeps track of how long to keep the Output HIGH by comparing the trigLength ("dur") to the
//current time of the counter - which is updated each time through the loop by the "stepper()" function
void trigger(int out, int patternNum, int dur) {
  switch (out) {
    case 1: //if the first argument ("out") is "1", do the following:
      //check to see if the counter is less than the note length
      //AND (&&) that the pattern indicates a "1". If so, write the output HIGH to 5V
      if (counter < dur && gateOnePatterns[patternNum][beat] == 1) {
        digitalWrite(out1, HIGH);
      }
```

```
  else if (gateOnePatterns[patternNum][beat] == 0) {
    digitalWrite(out1, LOW);
  }

  //if the above cases are not true, and the trigLength is equal to the step length,
  //don't turn the gate off at the end of the step. This will create a tie between adjacent "1"s
  //in a pattern when the length knob is turned all the way to maximum
  else if (stepTime == trigLength) {
    //if the gate duration is all the way up, don't toggle the gate LOW
  }

  //in any other case, write the gate LOW once the counter passes dur
  else {
    digitalWrite(out1, LOW);
  }
  break;

case 2:
  if (counter < dur && gateTwoPatterns[patternNum][beat] == 1) {
    digitalWrite(out2, HIGH);
  }

  else if (gateTwoPatterns[patternNum][beat] == 0) {
    digitalWrite(out2, LOW);
  }

  else if (stepTime == trigLength) {
    //if the gate duration is all the way up, don't toggle the gate LOW
  }
  else {
    digitalWrite(out2, LOW);
  }
  break;

case 3:
  if (counter < dur && gateThreePatterns[patternNum][beat] == 1) {
    digitalWrite(out3, HIGH);
  }

  else if (gateThreePatterns[patternNum][beat] == 0) {
    digitalWrite(out3, LOW);
  }

  else if (stepTime == trigLength) {
    //if the gate duration is all the way up, don't toggle the gate LOW
  }
  else {
    digitalWrite(out3, LOW);
  }
  break;

case 4:
  if (counter < dur && gateFourPatterns[patternNum][beat] == 1) {
```

```
      digitalWrite(out4, HIGH);
    }

    else if (gateFourPatterns[patternNum][beat] == 0) {
      digitalWrite(out4, LOW);
    }

    else if (stepTime == trigLength) {
      //if the gate duration is all the way up, don't toggle the gate LOW
    }
    else {
      digitalWrite(out4, LOW);
    }
    break;

  default:
    break;
 }
}
```

The trigger() function is declared void because it doesn't report any number back to the sketch. The three arguments are integers named "out", "patternNum", and "dur". An open curly brace indicates the start of the body of the function.

```
void trigger(int out, int patternNum, int dur) {
```

The out argument will be a number that specifies which output channel the function should be controlling. patternNum tells the function which of the channel's 5 patterns to read from. And dur will tell the function how long to keep the channel's output HIGH when the pattern array shows a 1.

The function uses a *switch…case* control structure to decide which portion of the code to execute. The first argument, out, is used to control which case gets activated within the switch structure. Since each of the four cases in the switch are identical (except for which channel they act on), we will just examine what happens in the first case. The main loop of the sketch ends with four calls to the trigger() function:

```
//write outputs high and low with custom function: trigger()
 trigger(1, patternNum, trigLength);
 trigger(2, patternNum, trigLength);
 trigger(3, patternNum, trigLength);
 trigger(4, patternNum, trigLength);
```

Since each of these passes a different number to the first argument, each will activate a different case in the switch structure. We will examine the first call:

```
trigger(1, patternNum, trigLength);
```

In this call, out is equal to 1, which will activate the first, and only the first, case of our switch. If out were equal to 2, it would activate the second case in our switch. Out = 3 would activate the third case, and out = 4 would activate the fourth case.

```
switch (out) {
   case 1: //if the first argument ("out") is "1", do the following:
     //check to see if the counter is less than the note length
     //AND (&&) that the pattern indicates a "1". If so, write the output HIGH to 5V
     if (counter < dur && gateOnePatterns[patternNum][beat] == 1) {
       digitalWrite(out1, HIGH);
     }

     else if (gateOnePatterns[patternNum][beat] == 0) {
       digitalWrite(out1, LOW);
     }

     //if the above cases are not true, and the trigLength is equal to the step length,
     //don't turn the gate off at the end of the step. This will create a tie between adjacent "1"s
     //in a pattern when the length knob is turned all the way to maximum
     else if (stepTime == trigLength) {
       //if the gate duration is all the way up, don't toggle the gate LOW
     }

     //in any other case, write the gate LOW once the counter passes dur
     else {
       digitalWrite(out1, LOW);
     }
     break;
```

Case 1: starts with an if statement control structure. When activated, the body of this if statement toggles output 1 HIGH. This will only execute if the conditional statement in the parenthesis is true. In this case, the "AND" boolean operator, "&&", indicates that the two conditions must *both* be true in order for this statement to execute and the output to be toggled HIGH.

First, "counter < dur" indicates that the counter must be less than the dur variable (which is the argument we pass to the function to tell how long to keep an output HIGH). This is how we are able to set the length of the output pulses. If the counter is greater than the desired pulse duration, the output will not be toggled HIGH).

Second, "gateOnePatterns[patternNum][beat] == 1" looks at the array for channel one to see if the current beat is a one. Consider the gateOnePatterns array:

```
bool gateOnePatterns[][16] = {
  {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
  {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1},
  {1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0},
  {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
};
```

Our conditional statement looks for a one in the current beat of the current pattern. patternNum is an argument that gets passed into the function from the main loop. The variable is set in the loop by the following instruction:

```
patternNum = map(patternRaw, 0, 1023, 0, PATTERN_COUNT - 1); //select from patterns numbered 0 - 4
```

This variable will be mapped from a knob reading range of 0 -1023 to the index range of our array. Since there are five patterns, we select from each of the rows by passing a number between 0 – 4 to the first bracket of the array. If patternNum == 2, we are looking at the third row of values, for example.

The variable "beat" in the second bracket position of the array tells us which item in the row we want to examine. "beat" gets set by the stepper() function. As an example, if patterNum is equal to 2 and beat is equal to 9, we are looking for the 10th item in the 3rd row, which is a 0 (in bold below). In this case, the channel would not be toggled high and the relevant line of code will be executed.

```
bool gateOnePatterns[][16] = {
  {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
  {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1},
  {1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0},
  {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
};
```

The next part of case 1 is a an *else if* statement. This is just like an if statement, except that it must follow another if statement and will *only* be evaluated if the previous if statement conditions *are not* met. This else if statement toggles the gate low if the step is a zero. In our previous example, the beat was a 0, so this is the statement that would be activated and the conditional statements that follow would be ignored. A beat of 0 will always toggle the output LOW.

```
else if (gateOnePatterns[patternNum][beat] == 0) {
  digitalWrite(out1, LOW);
}
```

Next, we have another else if statement. This one would only be evaluated if the current beat was a 1 and the counter was less than the dur variable. It states that if the stepTime is equal to the trigger length, don't do anything. This is really just here to block the final else statement that follows. This will have the effect of tying together multiple adjacent 1s by keeping the output from being toggled low at the end of every step.

```
//if the above cases are not true, and the trigLength is equal to the step length,
//don't turn the gate off at the end of the step. This will create a tie between adjacent "1"s
```

```
//in a pattern when the length knob is turned all the way to maximum
else if (stepTime == trigLength) {
  //if the gate duration is all the way up, don't toggle the gate LOW
}
```

The final part of the first case is the else statement and case break.

```
//in any other case, write the gate LOW once the counter passes dur
else {
  digitalWrite(out1, LOW);
}
break;
```

This else statement says that if none of the above are true, then toggle Output 1 LOW. Specifically, this is useful in cases where the pattern indicates a one, but the counter has become greater than the step duration. This will set the output low where it will stay until next time a 1 shows up in the pattern.

The break command tells the program not to look at any other cases and to leave the switch and do whatever comes after it. In this case, there is nothing else in the function, so the program leaves this function and heads back into the loop to execute the next instruction.

The final three instructions in the loop are just calling the trigger() function again, but each time it is looking at the next pattern array and toggling the next output channel. Since each of these trigger() calls are being passed the same variables for pattern number and the trigger length, they will seem strongly correlated and patterns may start to feel recognizable. More complex behavior can be generated by mixing and matching from different pattern numbers for different channels, and by changing how long triggers last for different channels. Simply read more knobs and map them to individual variables to be used as arguments for each of the different trigger() calls. For example, the six analog inputs of the Arduino could be set up as follows:

Knob on A0 will still control the stepping speed.
Knobs on A1 – A3 can be used to select different patterns for channels 1, 2, and the 3 and 4 channels can change together based on the A3 knob.
Knobs A4 and A5 can each select different trigger lengths and be used to alter channels 1+3 and 2+4 respectively.

Then the calls to trigger might look something like the following:

```
trigger(1, patternNum1, trigLength1);
trigger(2, patternNum2, trigLength2);
trigger(3, patternNum3, trigLength1);
trigger(4, patternNum3, trigLength2);
```

Using a microcontroller with more analog inputs, like the Arduino Mega, would allow unique knob control for each parameter. It is also worth mentioning that choosing 4 output channels was a relatively arbitrary decision. It is certainly possible to have more or fewer. Experiment to find the most suitable arrangement for your application.

If you don't have a modular synthesizer lying around, consider connecting these outputs to inputs of some of the circuits from the book. For example, this project pairs well with the 4093 NAND-based oscillators from an earlier chapter in the book. Connecting this project's outputs to one of each of the four oscillator inputs of the 4093 can create a dynamic polyphonic tone generator. In fact, many of the CMOS projects from the book will work great with Arduino provided you power the chips from the 5-volt pin of the UNO. If you have 9V CMOS synths, you can *probably* control them safely with the output of this circuit, but you wouldn't want to send anything back to the Uno from your synth – even by accident. Beware, applying more than 5 volts to any of the Uno's analog or digital pins risks damage to your system.

CIRCUIT 4: Basic Step Sequencing Using Delay

OVERVIEW

The following circuit uses multiple digital output pins to send manually-controlled voltages to an external voltage-controlled oscillator. This project is broken up so that functions are added over the course of several examples. This first example shows the basic mechanism for stepping through the outputs, activating one output at a time. Additional examples will modify this code to include a built-in oscillator and then a speed control knob.

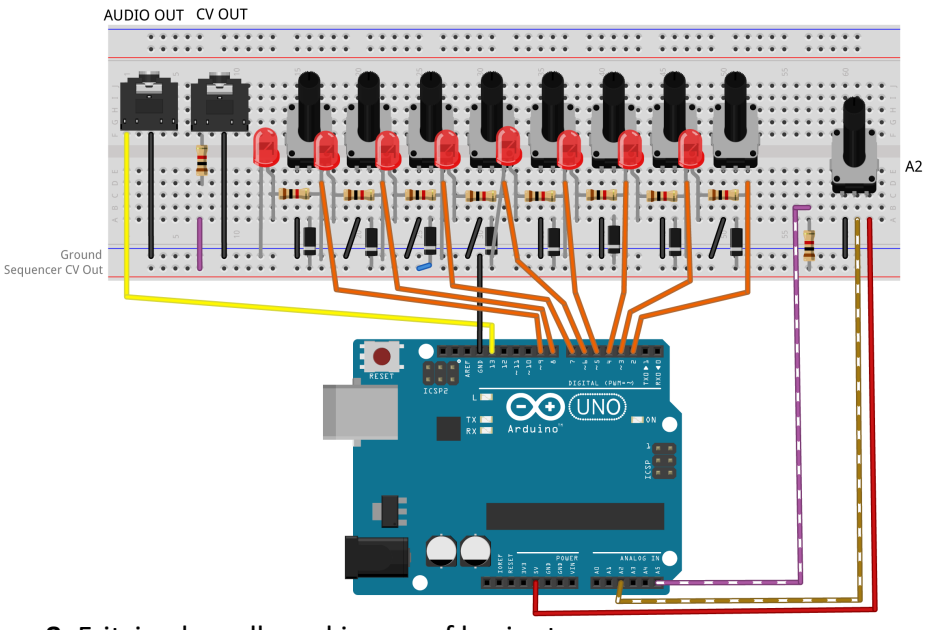

Figure 8: Fritzing breadboard image of basic step sequencer

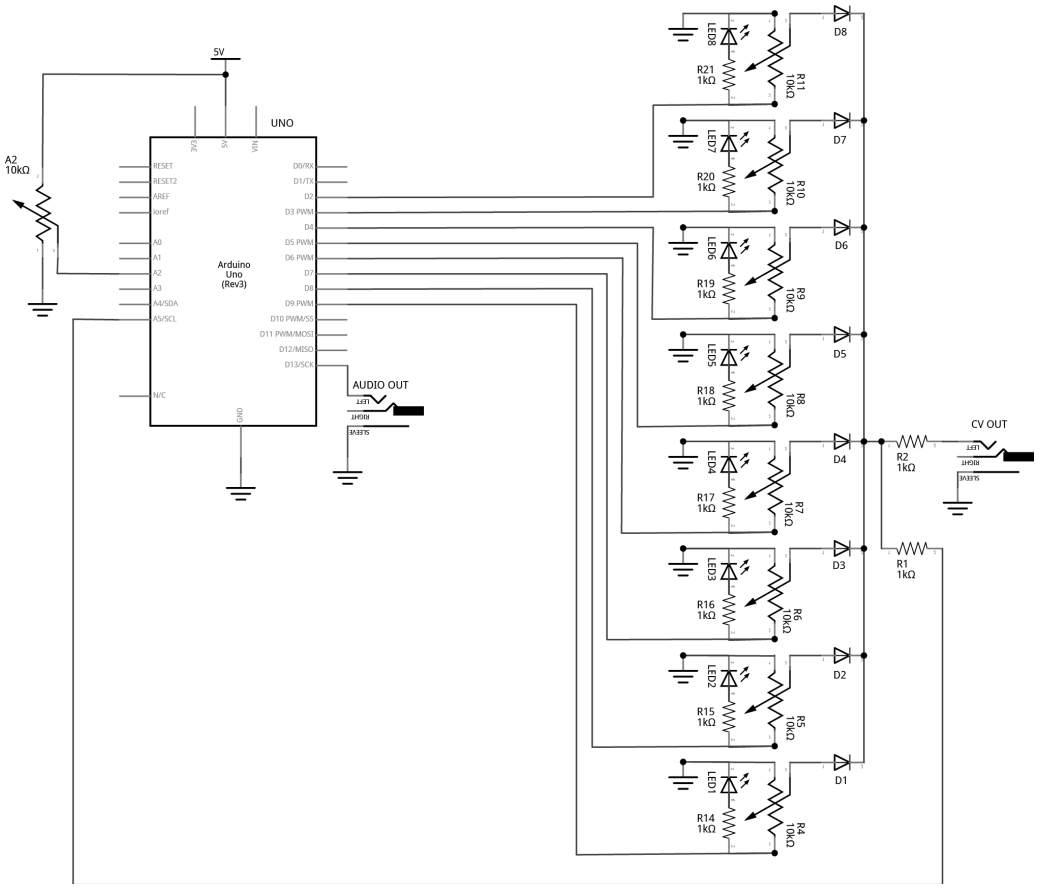

Figure 9: Fritzing schematic of basic step sequencer

You will need:

- An Arduino Uno.
- A computer running the Arduino IDE software.
- A USB cable that can connect from your computer to the Uno's USB type-B connector.
- A breadboard.
- Some solid hookup wire or premade wire jumpers.
- 10 1kΩ resistors.
- 9 10kΩ linear potentiometers.
- 8 LEDs (Light Emitting Diodes).
- 8 small signal diodes, such as 1N914.
- An audio amplifier.
- 2 jacks and plugs, to match your amplifier.
- Hand tools.

CONNECT THE HARDWARE

1. Open Fritzing File named *C4_SEQwithDELAY.fzz* and navigate to Breadboard view
2. Insert all components into breadboard matching image
3. Connect all wires on both breadboards

4. Connect wires from Arduino to breadboards
5. Visually Double-check all connections and wiring
6. Connect the Arduino Uno to your Computer

PROJECT SIX_A: Basic Step Sequencing Using Delay

PROGRAM THE ARDUINO with the Programmable_Sequencer_A sketch:

1. Open the Arduino sketch named *HEM_Programmable_Sequencer_A.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

This code functions in a very similar way to the ToneSEQ sketch in that it uses iteration to count through an array, so that at each new step of the array a new note can be played. (Though this first example will only produce notes if connected to an external voltage-controlled oscillator (VCO) via the CV output jack). One major difference between the ToneSEQ example and the project here is that this array does not include a list of notes, it instead includes a list of digital pins that are each toggled HIGH then LOW in turn. These pins are connected through resistors to LEDs that will light to indicate the current active step. The pins are also connected through potentiometers (knobs) that are manually adjusted so that each step will produce a different voltage (between 0 and about 5 volts). When a step is active, the potentiometer will have 0 volts on one end and 5 volts on the other. The knob can select any voltage in between 0 and 5 volts. When a step is inactive, it will have 0 volts on both ends, so the knob stays at zero volts regardless of its position. Each potentiometer's wiper (this is the center leg – it acts as a kind of output in our scenario) is tied together (each goes through a diode first so that they will not interfere with each other) and that junction is connected to the CV output jack and the Uno's own analog input pin #5.

The code starts by making sure that the step number it is going to turn on is within the range of the array:

```
if (stepNum > 7) {
  stepNum = 0;  //"0" is the first note, "1" is the second, and so on
}
```

Since the code starts with stepNum = 0, this if statement is not activated the first several times through the loop.

The code turns selected steps on (HIGH) and off (LOW) in the middle of the loop:

```
//this is where the step gets turned on, the code paused, and then the step turned off again
digitalWrite(steps[stepNum], HIGH); //toggle selected pin (stepNum[0] = 9) to 5v (step 1 turns on)
delay(stepTime);              //pause code for pre-determined number of milliseconds
digitalWrite(steps[stepNum], LOW);  //toggle selected pin to 0v (step 1 turns off)
```

Each time through the loop, the digitalWrite() function toggles a pin HIGH based on what the stepNum index variable is. When the program starts, stepNum is 0, so steps[0] = 9. This will cause pin 9 to be toggled HIGH. Then the program waits for a certain number of milliseconds as defined by the variable, stepTime (change this at the beginning of the code to try different speeds). Once the delay has ended, the digitalWrite() function is used to toggle the pin LOW again.

Finally, once the pin has been toggled HIGH and toggled LOW again, the variable stepNum gets incremented up by one:

```
stepNum ++;  //increment stepNum up by one
```

That means that stepNum will equal 1 now. The next time through the loop, steps[1] will address pin 8. Once all 8 pins have been toggled, stepNum++ will set stepNum equal to 8. The next time through the loop, the if statement at the beginning will see that 8 is greater than 7 and reset stepNum to 0 before the next step starts.

You can observe that this code is functioning by watching the pattern of LED lights. To hear anything though, the CV output of this circuit would need to be connected to an external oscillator. The next code example adds an internal oscillator to the project so tones can be produced at the audio output jack. We are going to use the tone() function to play a frequency based on what we read on analog pin 5. For a challenge, try to add this function yourself before moving forward. (Since the next example is just modifying the current code, the additions and deletions have been highlighted to make them easier to locate.)

PROJECT SIX_B: Add Internal Oscillator to Basic Sequencer

PROGRAM THE ARDUINO with the Programmable_Sequencer_B sketch:

1. Open the Arduino sketch named *HEM_Programmable_Sequencer_B.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

A new variable, tonePin, was created to represent the pin that our audio output jack will be connected to. This is the pin that the tone() function (described in previous examples) will produce its tone on. Additionally, a variable called note was declared so that we can scale our raw knob reading to a range of notes to provide to the tone() function.

The important new addition to the loop is as follows:

```
//get value of CV from pin A5
//Note: the (void) and the delay are there to deal with ghost voltages
//resulting from multiplexing of the ADC and the unbuffered circuitry.
//This problem could also be solved with additional hardware
(void)analogRead(CVPin);
delay(8);
```

```
note = map(analogRead(CVPin), 0, 1023, 35, 1200);     //get frequency value by reading the A5 pin
tone(tonePin, note);              //write the note to the audio out
delay(stepTime);              //pause code for pre-determined number of milliseconds
noTone(tonePin);
digitalWrite(steps[stepNum], LOW);  //toggle selected pin to 0v (step 1 turns off)
```

This is where we read the voltage of the current step, map it, and apply it to the frequency argument of the tone() function.

There is a little bit of funny business here due to the way the hardware is set up. If the analog pin is read directly it may carry over some voltage from the previous step's reading. This ghost voltage would cause bad tracking of our steps. As mentioned in the comments, changing the circuit would solve this problem and make the (void)analogRead(CVPin); and delay(8); unnecessary. But every attempt was made to keep the hardware simple and affordable and two lines of code adequately addressed the problem and didn't cost a penny.

Take note that another consequence of attempting to keep the hardware as minimal as possible is that the audio output and the CV output affect each other. If, while listening to the internal oscillator, you plug the CV into a v/oct input, you will notice a change in pitch of your internal oscillator's notes. Buffering the voltage from the step knobs using an op amp could provide a simple solution to both the ghost voltage problem and the loading at the analogRead pin when the CV out is connected. This is certainly worth considering for anyone looking to make improvements. However, our current technique is great for quickly generating audio and CV from a relatively simple set of components.

Finally, the delay pauses the program, then noTone() to stops the current note and digitalWrite() toggles the step LOW.

The final addition we will make to this sequencer design is to add knob to control the speed of the stepping. Consider trying it for yourself before uploading the next example. The unused knob is attached to pin A2.

PROJECT SIX_C: Add Speed Control Knob to Basic Sequencer

PROGRAM THE ARDUINO with the Programmable_Sequencer_C sketch:

1. Open the Arduino sketch named *HEM_Programmable_Sequencer_C.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?

The knob attached to A2 is now controlling the speed of the sequencer. The value read on pin A2 is stored in the variable stepTime. This variable is mapped to the new range of 300 to 1 (this range was set by ear and can be adjusted to taste) using the map() function, and then stored in the variable mappedSpeed. This new variable is then used in the delay function that sets the speed of the sequencer.

CONCLUSION

This 8-step sequencer is a relatively simple design and demonstrates the mechanics of the concept of generating analog outputs by combining digital signals and potentiometers. This example is severely limited by the use of the delay, though. The power of the microcontroller allows for many more features and options that can make our little beep machine into a performable instrument. In the final project of this series, the 8-Step Sequencer will be re-written to run without using a delay for its timing, allowing for a set of new features and enhancements to be explored.

CIRCUIT 5: 8-Step Performance Sequencer

OVERVIEW

The hardware for this project just expands what was already connected in the previous circuit by adding 3 additional knobs, 2 buttons, one switch, one additional digital output jack and one digital input jack with a transistor and two resistors to protect the microcontroller pin from unknown voltages.

The example sketches start by redesigning the stepping mechanics of the basic sequencer from the last project to function without using delay. Then, features are added a little bit at a time through a sequence of example sketches with the intention of showing how smaller techniques can be combined into larger, more complex projects. Each new sketch builds on the last and the location of code changes are indicated by the following: “//!!!”. While working through the examples, use the “find” feature in the Arduino IDE to look for the 3 exclamation points. They will show the changes and the nearby comments will explain the techniques.

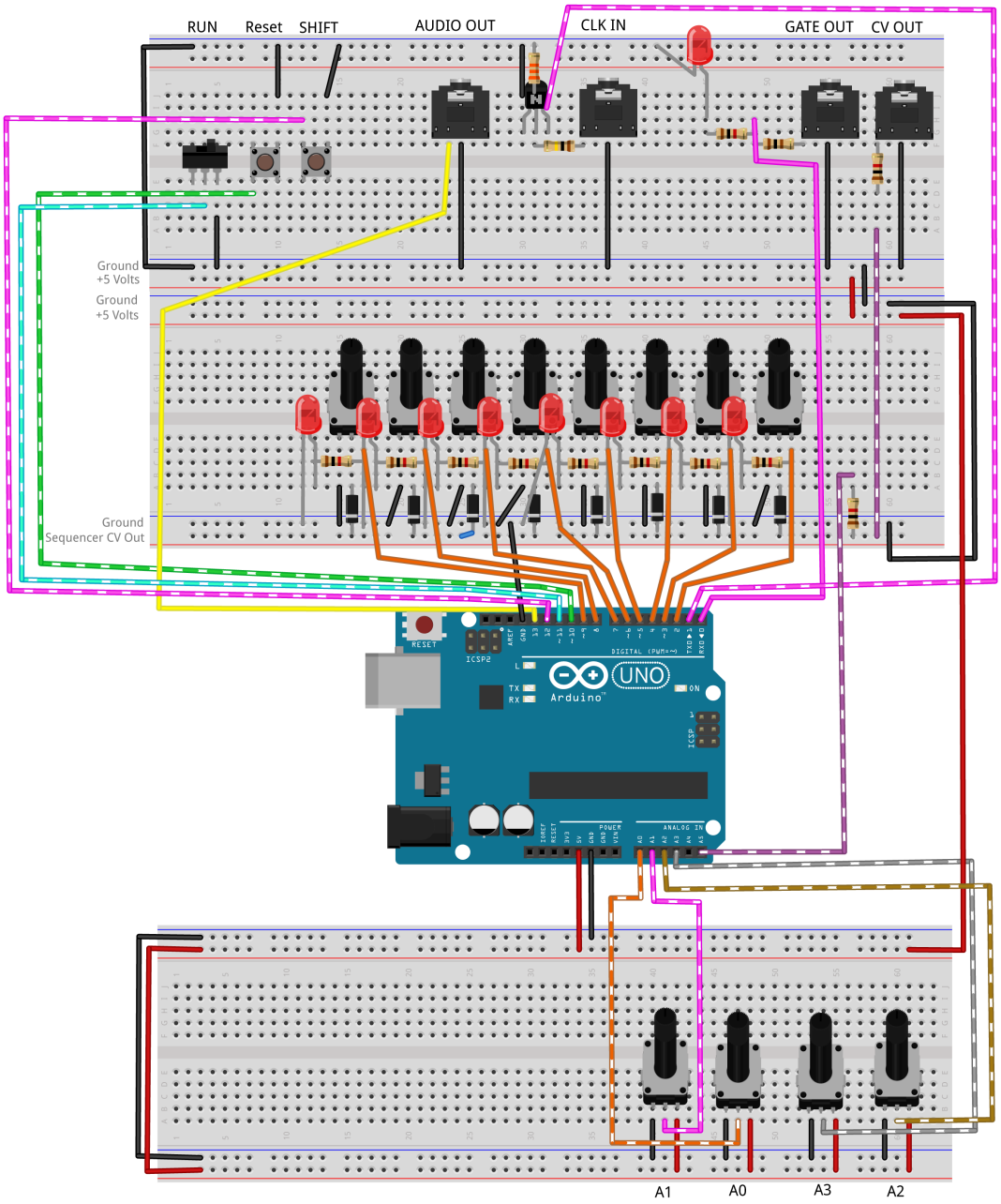

Figure 10: Fritzing breadboard image of performance sequencer circuit

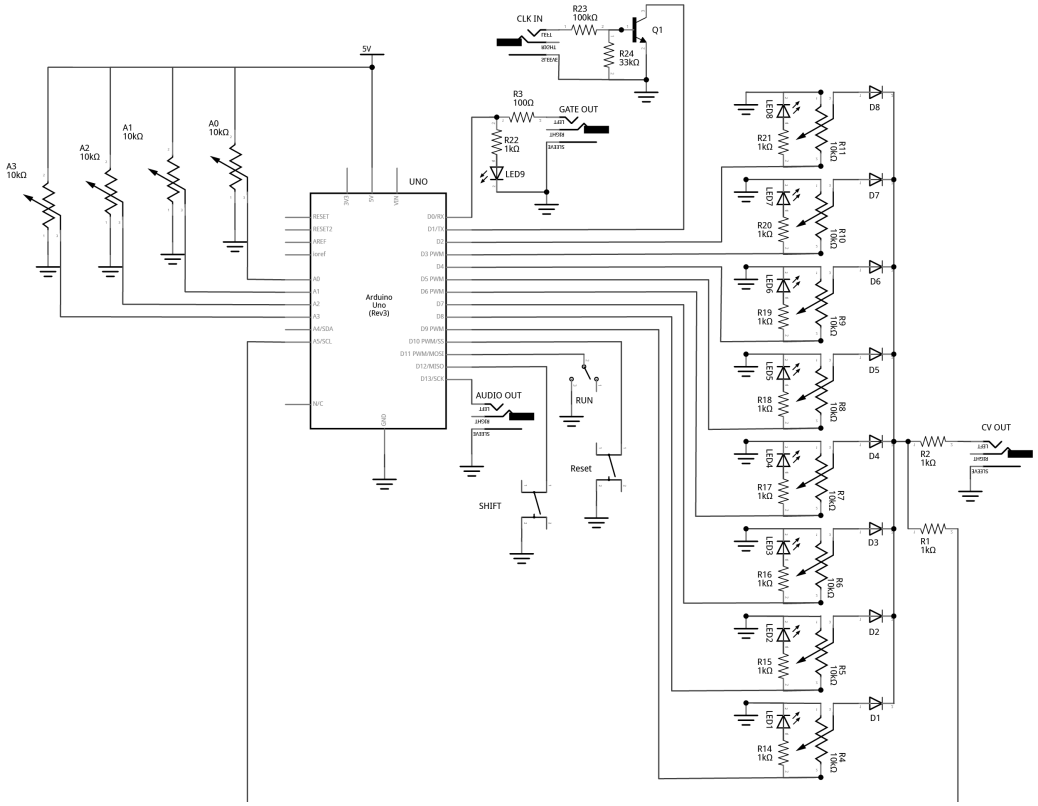

Figure 11: Fritzing schematic of performance sequencer circuit

You will need:

- An Arduino Uno.
- A computer running the Arduino IDE software.
- A USB cable that can connect from your computer to the Uno's USB type-B connector.
- Several breadboards.
- Some solid hookup wire or premade wire jumpers.
- 1 x 100kΩ, 1 x 33kΩ, 11 x 1kΩ, and 1 x 100Ω resistors (it is okay to use values that are within about 10 – 15% of these values.
- 12 10kΩ linear potentiometers.
- 9 LEDs (Light Emitting Diodes).
- 8 small signal diodes, such as 1N914.
- 1 toggle switch.
- 2 pushbuttons.
- An audio amplifier.
- 4 jacks and plugs, to match your amplifier and external modules.
- Hand tools.

CONNECT THE HARDWARE

1. Open Fritzing File C5_StepSEQ.fzz and navigate to Breadboard view
2. Insert all components into breadboard matching image

3. Connect all wires on both breadboards
4. Connect wires from Arduino to breadboards
5. Visually Double-check all connections and wiring
6. Connect the Arduino Uno to your Computer

Hint: Open the downloaded project sketches and look for the "!!!" in the comments to quickly find changes as the program grows through subsequent examples.

PROJECT SEVEN_A: Step Sequencer: Controller (no delay)
PROGRAM THE ARDUINO with the A_SEQ_StepController sketch:

1. Open the Arduino sketch named *A_SEQ_StepController.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S HAPPENING?
This code uses a custom function to keep track of time. It takes two arguments, one for how long each step should last, and one for how long a note should stay on during each step. The mechanics are very similar to the stepper() function from the Quad Gate Generator example. It uses the millis() function to check the current time against the last time a step was taken. This function is a little bit different in that it sets flags so that the rest of the program can check to see if it is time to start a step or if it is time to stop a step. Most of the time we are neither starting a step nor stopping a step.

PROJECT SEVEN_B: Step Sequencer: Internal Oscillator
PROGRAM THE ARDUINO with the B_SEQ_tone sketch:

1. Open the Arduino sketch named *B_SEQ_tone.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
The major changes in this example involve the creation of a bunch of variables to handle connecting a reading the CV voltage and some for mapping the raw voltage to a range of frequencies that are useful in the tone() function. Then noTone() is called if the stepStop flag is true. tone() is called if the stepStart flag is true. If neither flag is true, the loop just keeps updating the stepController() function until something changes.

PROJECT SEVEN_C: Step Sequencer: Speed
PROGRAM THE ARDUINO with the C_SEQ_SpeedControl sketch:

1. Open the Arduino sketch named *C_SEQ_SpeedControl.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?

In this example, the value read from a knob attached to A2 is passed into both arguments of the stepController() function. This will use the same value to control the speed of the stepping and the length of the steps. Experiment with using a second knob for the second argument. Inside the stepController(), the analog pin is read and then mapped to values that are appropriate for the two parameters.

```
int speedValRaw = analogRead(speedIn); //!!! read the speed knob and store the raw value
int stepInterval = map(speedValRaw, 0, 1023, 300, 10); //!!! map the value to new speeds (chosen by ear)
```

Note above, the stepInterval (the speed of stepping) is mapped to my taste. Change the last two numbers in the map function to experiment with different speed control.

```
int durValRaw = analogRead(durIn);  //!!! read the duration knob and store the raw value
int stepDuration = map(durValRaw, 0, 1023, 1, stepInterval - 1); //!!! map the duration knob to a new range from very short to an entire step - 1ms
```

Also note that the durations are automatically scaled based on the step interval and are able to be set from as little as 1 ms to as great as an entire step interval minus one millisecond. This is useful for keeping the start and stop flags unambiguous.

The final thing to mention is that this code stops the sequencer when the knob is all the way down. This is used as a pause function for now. Later this knob position will be where we add a mode to read an external clock signal.

PROJECT SEVEN_D: Step Sequencer: Gate Output

PROGRAM THE ARDUINO with the D_SEQ_GateOutput sketch:

1. Open the Arduino sketch named *D_SEQ_GateOutput.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?

When connecting the CV to an external oscillator, having the LEDs turn off partway through the step length will cause the oscillator to play two notes for every step: one that you intended, and the other corresponding to zero volts. To fix this, the steps themselves will now be kept high for an entire step length, but a gate signal will be written that corresponds to the step duration control. The internal oscillator will continue to function as before, with notes only playing for the selected note duration. Now the gate output and it's corresponding LED will mirror this duration, allowing the oscillator to be tuned each step, and the Gate to control the volume through a VCA or other voltage-controlled dynamic module. (Consider rolling your own quick and dirty volume control with the photocell/LED pairing approach described earlier).

PROJECT SEVEN_E: Step Sequencer: Alternate Step Modes

PROGRAM THE ARDUINO with the E_SEQ_SteppingModes sketch:

1. Open the Arduino sketch named *E_SEQ_SteppingModes.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
In this example, several new options have been added for how the sequencer moves through its eight possible steps. There are five defined, but any number less than 1024 would be possible. The modes are selected by a knob attached to pin A3.

The 5 Modes

Repeat Mode: The sequencer just repeats its current step forever. This can be useful for tuning each step.
Forward Mode: The sequencer advances left to right as normal.
Reverse Mode: The sequencer advances right to left.
Random Mode: The sequencer selects steps randomly at each step interval.
Double Mode: The sequencer advances by two from left to right. It plays every other note.

PROJECT SEVEN_F: Step Sequencer: Run/Pause Switch
PROGRAM THE ARDUINO with the F_SEQ_RunSwitch sketch:
1. Open the Arduino sketch named *F_SEQ_RunSwitch.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
This example adds a switch that must be engaged in order for the sequencer to advance in its current step mode. The switch is set up and read just like a button. An if statement encloses almost all of the loop. If the switch is not engaged, there is no need to do all the stuff in the loop, just turn off the internal oscillator. If the switch is engaged, run the program like normal.

PROJECT SEVEN_G: Step Sequencer: Reset Button
PROGRAM THE ARDUINO with the G_SEQ_ResetButton sketch:
1. Open the Arduino sketch named *G_SEQ_ResetButton.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
This sketch adds a "Reset button". The Reset button will have different behaviors in each of the operating modes. This means there are a lot of new lines of code for a relatively simple-seeming addition. The behavior we have programmed for the Reset button in each mode is as follows:

Mode 1: In step repeat mode, the reset button actually causes the step to advance at each step start. This is a kind of "manual run" button.
Mode 2: In Forward Mode, reset sends the step back to step 1 at the next step start.
Mode 3: In Reverse Mode, it will send the step back to step 8.

Mode 4: In Random Mode, it will hold at the current step until the button is released.
Mode 5: In Double Mode, reset sends the step back to step 1 at the next step start.

PROJECT SEVEN_H: Step Sequencer: Scale Quantizer for Internal Oscillator
PROGRAM THE ARDUINO with the H_SEQ_Quantizer sketch:

1. Open the Arduino sketch named *H_SEQ_Quantizer.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
This example adds a performable note quantizer. The quantizer is created in a custom function. Nearly all of the changes to the sequencer happen in this function definition. The logic of the quantizer is very similar to that used in the earlier quantizer example, but it has been altered here a little bit to better fit the context of a step sequencer. Recall that a quantizer takes some incoming stream of values and constrains its outputs to some predetermined values. This quantizer reads the voltage on the CV pin and causes the internal oscillator to only output notes in a desired scale. The scale can be chosen using a knob attached to pin A1. A knob attached to A0 will cause the tones that are played to shift up in the scale by a selected degree. This will allow all the knobs still play tones in the same scale, but they can be modulated up and down. See the earlier Note Quantizer project for a more detailed explanation of how the quantizer functions.

Note, this only affects the internal oscillator. The CV output voltages are set directly by the knobs and are not processed by the microcontroller and so they cannot be quantized in this way.

PROJECT SEVEN_I: Step Sequencer: Silence Steps
PROGRAM THE ARDUINO with the I_SEQ_SilenceSteps sketch:

1. Open the Arduino sketch named *I_SEQ_SilenceSteps.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
This example shows how to add some logic to silence any step with a knob that is turned all the way down. It can be used to introduce rests into patterns of notes. There is just a little bit of conditional logic added toward the end of the loop to make this work.

This will cause the internal oscillator to go quiet. It will not affect the CV output, but it will cause the Gate Output to stay LOW on these steps. If using the gate to control the volume of your VCO through a VCA, this can affectively silence your external tone as well.

PROJECT SEVEN_J: Step Sequencer: Shift Button_Octave Knob
PROGRAM THE ARDUINO with the J_SEQ_ShiftOctave sketch:

1. Open the Arduino sketch named *J_SEQ_ShiftOctave.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
All the knobs in the design are now in use, but there are still many parameters that it would be nice to have real-time control over. While we could use a microcontroller that has more analog inputs, or employ an additional chip called a *multiplexor*, both of those solutions require additional hardware. Instead we will use the second button, which I'm referring to as the "Shift Button", to activate alternative, *shifted* modes for some of the knobs.

These new lines use the scale knob to update two different parameters in the Quantizer function. When the button is pressed, the variable for the octave is updated allowing the base octave of the quantizer to be changed by turning the scale knob. When the button is not pressed, the octave variable will stay at its most recent value and the knob will update the scale type as usual.

PROJECT SEVEN_K: Step Sequencer: Shift Button_Note Duration Knob
PROGRAM THE ARDUINO with the K_SEQ_ShiftDuration sketch:
1. Open the Arduino sketch named *K_SEQ_ShiftDuration.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
In this update, the shift button is used to activate an alternative function for the mode selection knob in addition to the alternate mode selection from the previous example. While the button is pressed, the mode selection knob will be used to update a variable controlling the note length. This will allow the mode knob to set the duration of the steps. When the shift button is not being pressed, the mode knob will update the current stepping mode as usual and the note length parameter will stay wherever it was when the shift button was released.

PROJECT SEVEN_L: Step Sequencer: Change Detection for Shifted Values
PROGRAM THE ARDUINO with the L_SEQ_ChangeDetection sketch:
1. Open the Arduino sketch named *L_SEQ_ChangeDetection.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?
This example addresses a problem with the shift button logic of the previous sketches. In the previous sketches, when the Shift Button is pressed, the shifted variable is immediately updated the knob's current position. After the knob has been moved and the new parameter value chosen, the Shift Button is released. When this happens, the un-shifted parameter is immediately updated to the current knob position. This problem is noticeable when shift only changes one parameter, but the problem becomes quickly unmanageable when shift is used for more than one knob. In the previous example, every time shift is pressed, both knobs are read

and the values are immediately applied to their variables. Playing the sequencer will reveal why this is such an issue: every time you shift to change a parameter, the parameter of the second knob will change as well if the knob is in a different position than the last time shift was pressed.

To solve this problem, a change detection algorithm is developed to keep the knobs from updating after shift has been pressed or released until the program can be sure that the knob was intentionally turned. This means that once shift is pressed, the shifted variable will be locked until the knob is wiggled. Once the knob is wiggled, it is updated just like normal until the state of the shift button changes again.

The changeRead() function locks out updates of the knobs' variables when the Shift Button first changes state. (changing state just means going from being un-pressed to being pressed, or vice-versa). Then it quickly takes several readings of the knob and stores the differences between successive readings. If the total changes by more than a few digits, we can assume that the knob was intentionally moved. Then changeRead reports the knob has been moved and the chosen variable can now be freely updated until the Shift Button changes state again. The function returns a 1 to report that shifted parameter can be changed. It returns a 2 to indicate that the unshifted parameter can be changed.

This change detection is implemented as a custom function. Each knob has its own changeRead() function defined. If more knobs are assigned shifted parameters, additional changeRead() functions will need to be created for each of the knobs by simply duplicating the entire function definition and giving it a unique name.

PROJECT SEVEN_M: Step Sequencer: Trigger Input for External Clocking

PROGRAM THE ARDUINO with the M_SEQ_TriggerIn sketch:

1. Open the Arduino sketch named *M_SEQ_TriggerIn.ino*
2. Click the upload button near the top left of the sketch window

WHAT'S NEW?

This example introduces our last feature: external clocking. This happens in the custom stepController() function whenever the speed knob is turned down to zero. Turning the speed knob all the way down disengages the internal clock and sets the sequencer in external clocking mode.

We read a voltage level on the trigger input pin and each time it changes state from HIGH to LOW (this indicates a the presence of a new signal) then the startStep flag gets set to *true* and our sequencer takes a step, playing a note and setting the Gate Out HIGH. If there is no voltage detected on the pin, the stopStep flag gets set to true and notes stop and the Gate Out goes LOW.

The last little bit of new hardware is introduced in this technique: an NPN transistor (2n2222 or 3904 work well) is used to pull our INPUT_PULLUP trigger pin LOW whenever a voltage over about a volt is present on the base of the transistor via the trigger input jack. This wouldn't be totally necessary if we know that our input clock signal is coming from a 5-volt device, like another Arduino Uno. In that case, we could read the voltage directly on the pin. But the addition of the transistor and its two resistors means that unknown voltage levels from external devices are not applied directly to the microcontroller but are instead applied to the transistor. This offers a degree of protection and allows for the higher voltage levels of common modular synthesizer formats (like Eurorack) to be used safely.

PROJECT SEVEN CONCLUSIONS

PROGRAM THE ARDUINO with the N_SEQ_Final sketch:

1. Open the Arduino sketch named *N_SEQ_Final.ino*
2. Click the upload button near the top left of the sketch window

This last example is the exact same as the previous example, except the "!!!" markers have been removed.

This project is meant to demonstrate the process of increasing the complexity of code by adding features one at a time. The sequence of examples is designed to illustrate some techniques for integrating existing code into a project, and to encounter some common problems and possible solutions along the way. There are many more features that could be added and substitutions that could be made with the existing controls. Likewise, the hardware has been kept minimal for cost and the software somewhat inelegant for clarity. As a result, there are many opportunities to improve the design and function of both.

It is also worth mentioning that since this is an Arduino project, the code could easily be migrated to alternative hardware to explore additional/alternative features. For example, the Arduino Mega has many more inputs and outputs, so a longer sequence length would be easily achievable, and many more parameters could be placed under real-time control via the additional analog input pins. Alternatively, using something like the Teesny 3.2 or 3.6 would allow the use of the Teensy Audio System design tool and Audio library. This would enable a relatively easy expansion of the internal synthesis engine allowing this sequencer to play all sorts of different wave shapes, physical models, or even samples through programmable volume envelopes, filters, distortion, and more. Additionally, the Teensy would allow USB MIDI notes to be generated by the quantizer (with a little modification to the function). Your sequencer hardware could then control plugins and other instruments on your production computer using the same USB cable you used to program it.

C: Additional Arduino Audio Learning Resources

Once you become familiar with the projects in this chapter and supplement, the next step is to try some custom libraries. Libraries are like expansion packs for the Arduino software language:

they bundle new functions geared toward specific tasks. The free and open-source Mozzi Library is a good starting point for synthesis and sampling. It has some powerful features that work on a wide variety of boards including the Uno, many example sketches and tutorials, and a forum to find help. Similarly, the Teensy 3.x family of boards have an expressive audio library and are powerful, small, and relatively user-friendly.

To go beyond the use of other designers' libraries and write your own audio programs in Arduino, I recommend the DSP tutorials of Amanda Ghassaei, published on the Instructables website and linked from her personal site http://www.amandaghassaei.com/projects/arduinodsp/. Additionally, Brent Edstrom's *Arduino For Musicians: A Complete Guide for Arduino and Teensy Microcontrollers* charts a clear path from basic Arduino programming concepts to building custom classes and devices for digital signal processing.

D: Alternative Hardware Options

In addition to the expanded functionality possible with more complex code, deeper levels of sophistication are achievable by applying additional hardware. (Conversely, more minimalist outcomes are also possible if not all the features of the Uno are necessary.) And while the spirt of hacking certainly supports rolling your own solution with custom circuitry, there are many available alternatives to the Uno board that can provide the benefit of working hardware and significant educational support. The Arduino brand offers smaller board sizes for breadboard compatibility and project flexibility, larger sizes with more inputs and outputs, boards with faster processing, lower power, wireless capabilities, built-in sensors, etc. Additionally, many third-party companies produce Arduino-compatible boards that can be programmed from within the Arduino IDE with little to no modification of existing code. Some features that are particularly useful for audio purposes include native USB MIDI functionality and built in digital to analog converters (DACs) that are capable of producing output waveforms with complexity far beyond that of the digital rectangles of the Uno. And while the power of microcontrollers grows every year, there are still case where multiple streams of high quality, real-time audio processes become necessary. For the next step up in embedded digital audio, it may be reasonable to consider non-Arduino solutions like Axoloti, or single-chip computers solutions like the Bela Audio platform (built on the Beaglebone) or audio software running on a Raspberry Pi.