

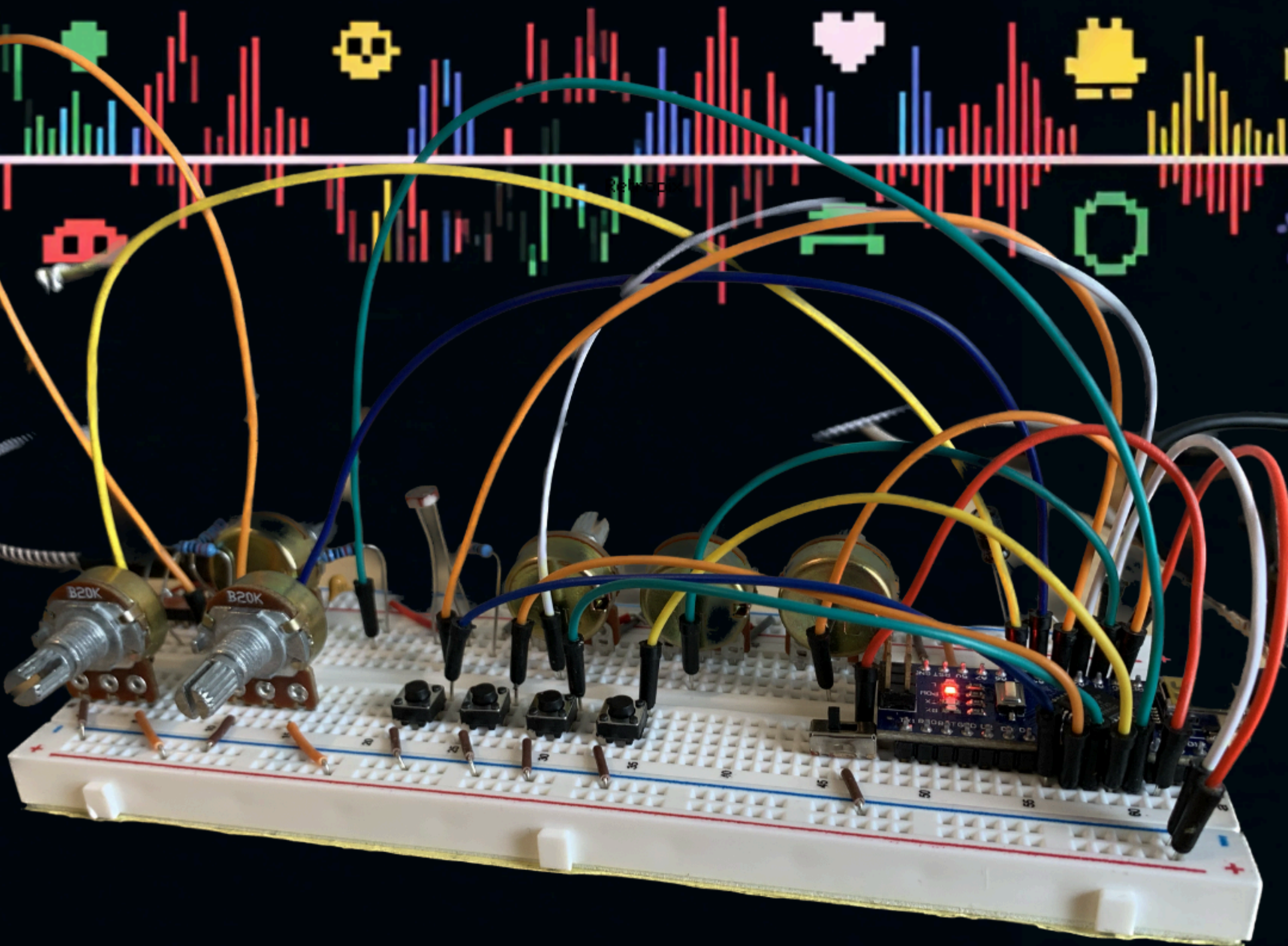
chipBoard DIY synth kit

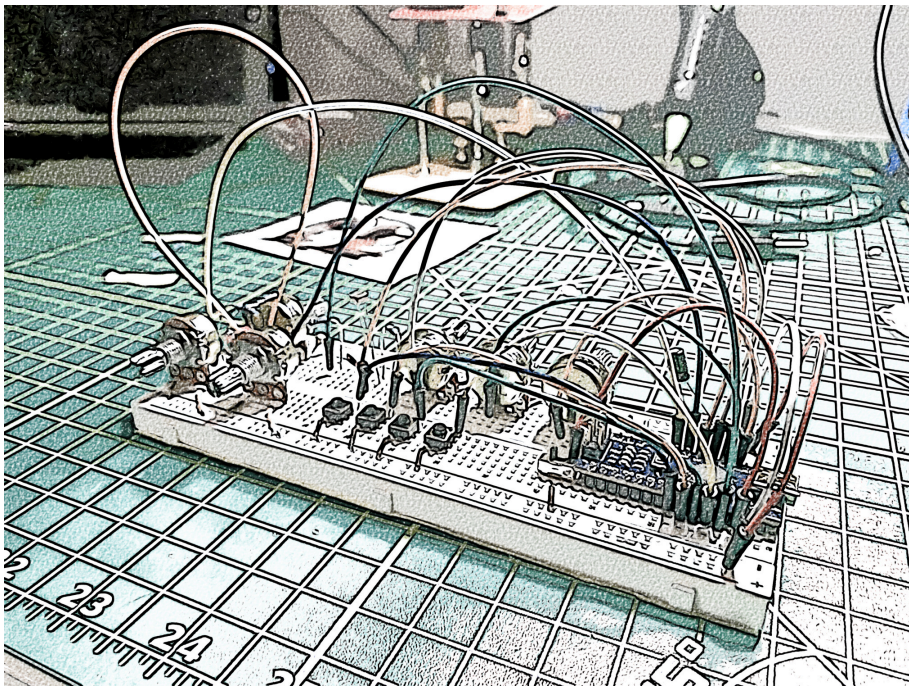
Build it yourself, and journey
into the wilds of 8bit, glitchy,
chiptune sounds!!

Order a Kit,

or sign up for a workshop at:
rhythmsciencesound.com

Learn the basics of building
electronics and digital
synthesis using Arduino,
and go on to make your
own projects.





RHYTHM SCIENCE SOUND



Welcome to Rhythm Science Sound “chipBoard” DIY 8-Bit Synth Kit,

a hands-on Introduction to digital synthesis and music making with Arduino!

This kit includes the components, instructions and Arduino code sketches for various projects designed to give a foundational knowledge of the Arduino platform and how we can use it to make electronic instruments and sound generators.

Your Arduino board is already loaded with the main chipBoard program so feel free to build and get playing.

Full code and build resources can be found at:

rhythmsciencesound.com/chipboard

Arduino Nano MicroController(MCU) Board:

At the heart of this kit is the Arduino Nano. Arduino are a family of development boards which include a microcontroller IC chip(integrated circuit) as well as pins which provide connections for adding analog and digital inputs/outputs(buttons, sensors, audio output, etc), as well as power supply components which allows us to supply power to supplemental circuitry. Arduinos also feature a USB connection which lets us easily upload code or “sketches” via the Arduino IDE software in order to program and control the functionality of our Arduino. Arduinos are available in several different development board sizes and configurations, all with different capabilities and specifications to fit a designer’s needs. The awesome thing about the Arduino platform is that it is open source and customizable, which means you can write custom program code for the arduino. You can also learn from, reuse and modify the huge library of code sketches and tutorials available online, which are shared by the worldwide community of Arduino creators.

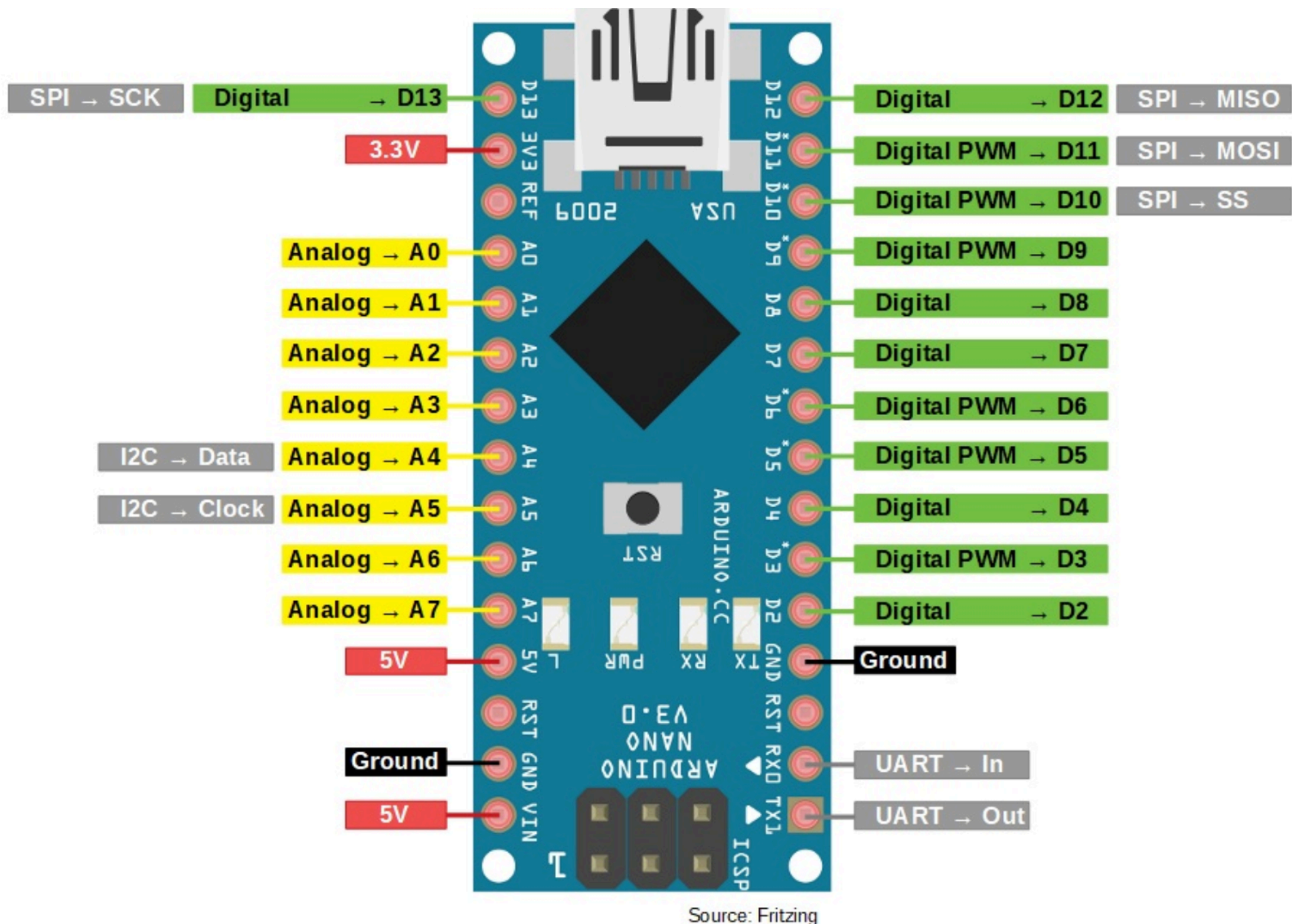
To build our synthesizer, we’ll be using the Arduino Nano which features:

- a USB connection for power and data transfer,

- 5v and GND pins for powering our breadboard circuit,

- 8 analog input pins A0-A7 for connecting analog sensors(potentiometers, LDRS, etc). We’ll be using A0-A5

- 11 Digital in/out pins D2-D13 for connecting buttons for input detection, or producing logic (HIGH/ LOW) outputs), and producing Pulse Width Modulation(useful for controlling LEDs, motors, and synthesizing tones!



1. **Power Pin (Vin, 3.3V, 5V, GND):** These pins are power pins

- Vin** is the input voltage of the board, and it is used when an external power source is used from 7V to 12V.
- 5V is the regulated power supply voltage of the nano board and it is used to give the supply to the board as well as components.
- 3V is the minimum voltage which is generated from the voltage regulator on the board.
- GND** is the ground pin of the board

2. **RST Pin (Reset):** This pin is used to reset the microcontroller. **Analog Pins (A0-A7):** These pins are used to calculate the analog voltage of the board within the range of 0V to 5V

3. **I/O Pins (Digital Pins from D0 – D13):** These pins are used as an i/p otherwise o/p pins. 0V & 5V

4. **Serial Pins (Tx, Rx):** These pins are used to transmit & receive TTL serial data.

5. **External Interrupts (2, 3):** These pins are used to activate an interrupt.

6. **PWM (3, 5, 6, 9, 11):** These pins are used to provide 8-bit of PWM output.

7. **SPI (10, 11, 12, & 13):** These pins are used for supporting SPI communication.

8. **Inbuilt LED (13):** This pin is used to activate the LED.

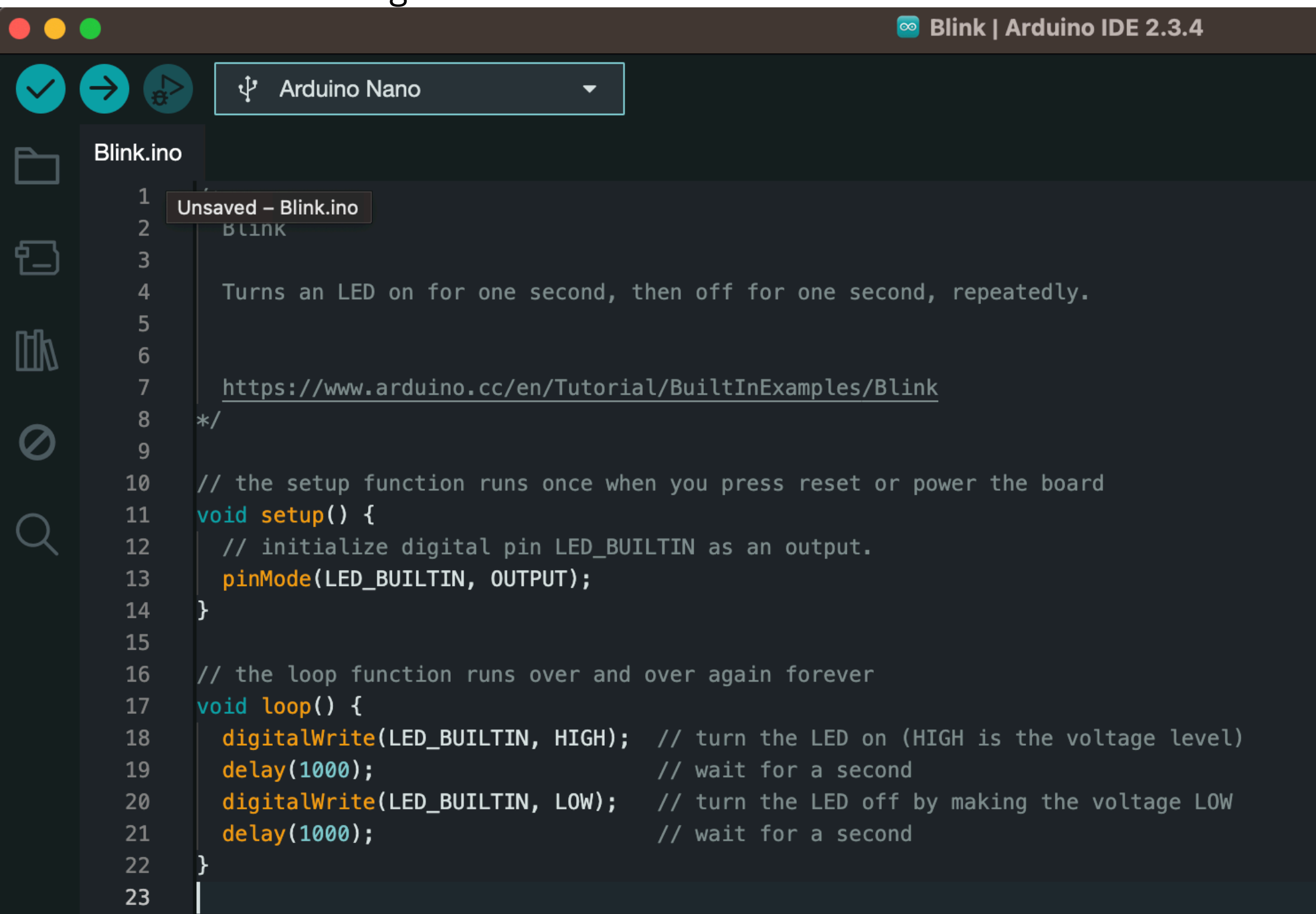
9. **IIC (A4, A5):** These pins are used for supporting TWI communication.

10. **AREF:** This pin is used to give reference voltage to the input voltage

Arduino IDE

Arduino provides an integrated development environment (IDE) which provides an abstracted text-based interface for coding your micro-controller. This environment is our workspace where we work with various code objects and libraries, write functions, and set rules and protocols for controlling and/or our Arduino's input and output pins. Ultimately, you'll learn the most by just diving in and gaining experience with a variety of code examples and the text usage and syntax will become more clear with each new project.

Here is a basic and classic example in which code is written to have an Arduino blink an Light Emitting Diode(LED). Here we can see some fundamental parts of an Arduino program or "sketch" where pins are set up in the setup() function and then in the loop() function are oscillated between HIGH and LOW voltage to turn on and off the LED.



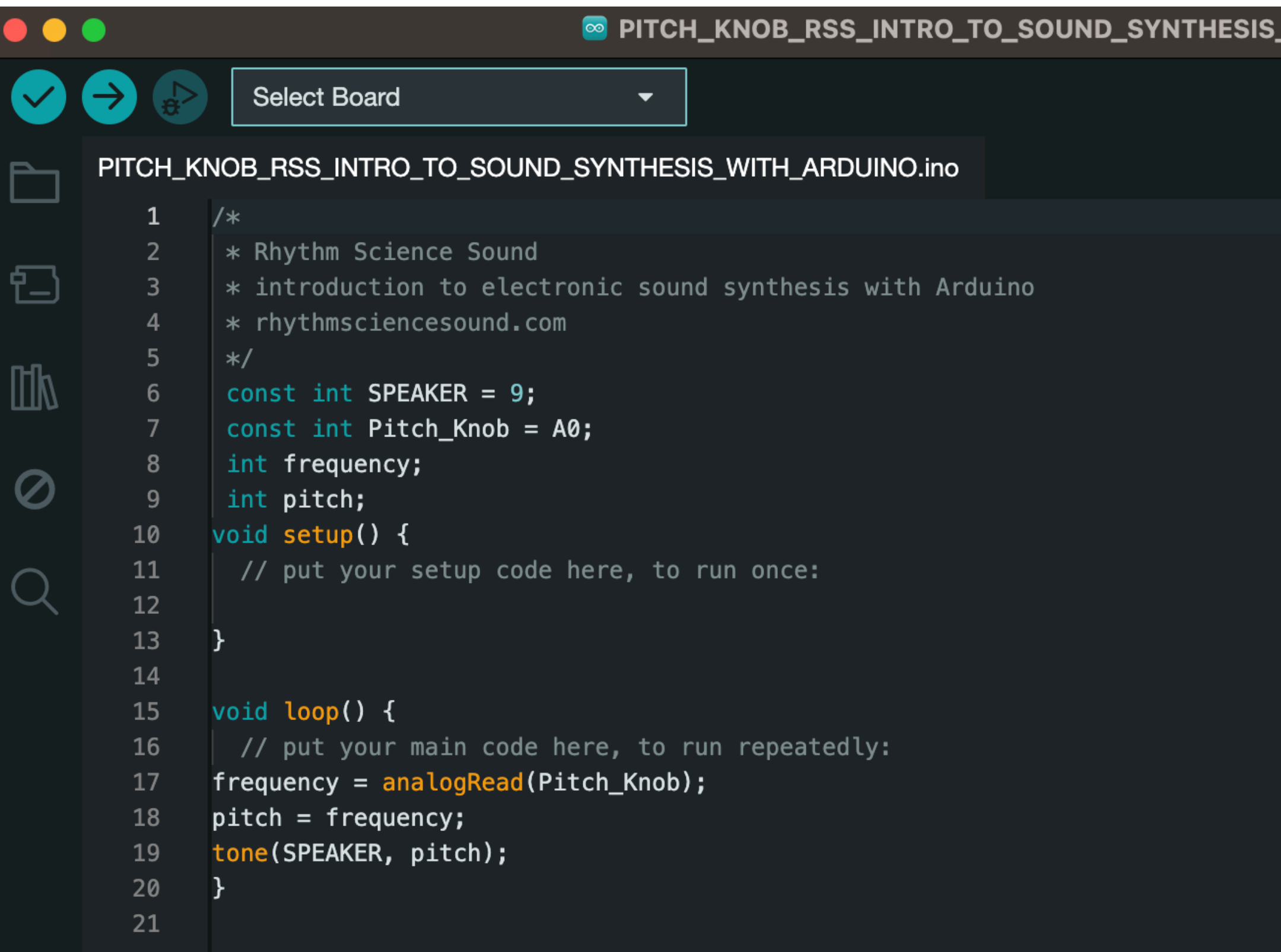
The screenshot shows the Arduino IDE 2.3.4 interface. The top bar indicates the current sketch is "Blink | Arduino IDE 2.3.4". Below the top bar, there are three icons: a checkmark, a right arrow, and a play button. A dropdown menu shows "Arduino Nano". The left sidebar contains icons for a folder, a file, a library, a search, and a magnifying glass. The main editor area displays the "Blink.ino" sketch, which is unsaved. The code is as follows:

```
1  // Blink
2
3
4  Turns an LED on for one second, then off for one second, repeatedly.
5
6
7  https://www.arduino.cc/en/Tutorial/BuiltInExamples/Blink
8  */
9
10 // the setup function runs once when you press reset or power the board
11 void setup() {
12   // initialize digital pin LED_BUILTIN as an output.
13   pinMode(LED_BUILTIN, OUTPUT);
14 }
15
16 // the loop function runs over and over again forever
17 void loop() {
18   digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
19   delay(1000); // wait for a second
20   digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
21   delay(1000); // wait for a second
22 }
23
```


Let's help the arduino sing

Lights are great, but we're here for the synth tones! We can start getting our to synthesize sound with the following simple bit of code for a voltage/pitch follower controlled by a single potentiometer providing a variable voltage to an Analog Input pin(A0 in this example), and outputting sound via the tone() function on digital pin 9(SPEAKER).

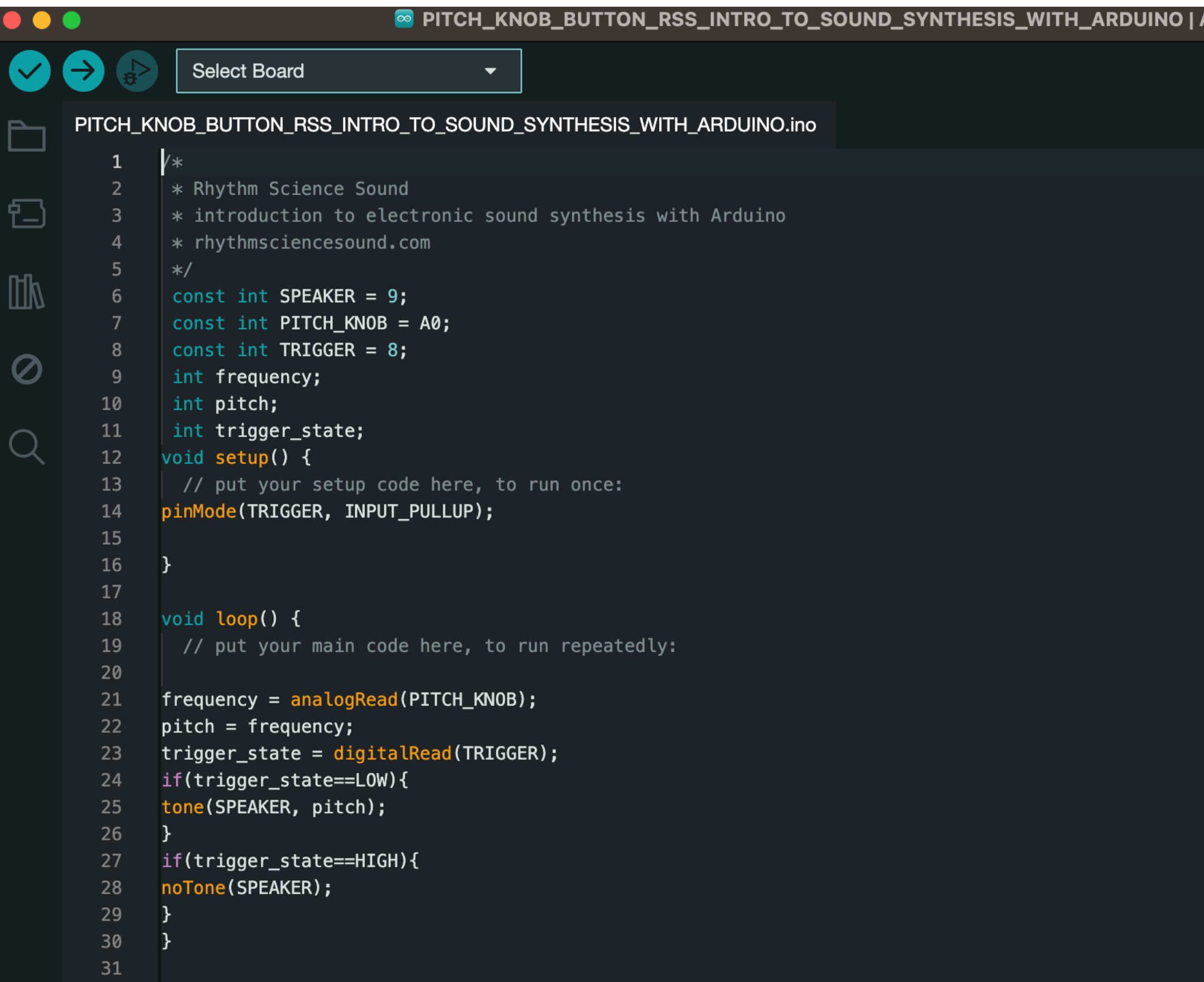
For this sketch, we see the variables for SPEAKER, Pitch_Knob, frequency, and pitch all declared before setup(). This sketch needs nothing set in setup(), and the loop() function simply reads the voltage from the Pitch_Knob into the frequency variable which then is passed to the tone() function

The image shows the Arduino IDE interface. At the top, the window title is "PITCH_KNOB_RSS_INTRO_TO_SOUND_SYNTHESIS_". Below the title bar, there are three colored window control buttons (red, yellow, green) and a "Select Board" dropdown menu. The main workspace displays a file named "PITCH_KNOB_RSS_INTRO_TO_SOUND_SYNTHESIS_WITH_ARDUINO.ino". The code is as follows:

```
1  /*
2   * Rhythm Science Sound
3   * introduction to electronic sound synthesis with Arduino
4   * rhythmsciencesound.com
5   */
6   const int SPEAKER = 9;
7   const int Pitch_Knob = A0;
8   int frequency;
9   int pitch;
10  void setup() {
11    // put your setup code here, to run once:
12
13  }
14
15  void loop() {
16    // put your main code here, to run repeatedly:
17    frequency = analogRead(Pitch_Knob);
18    pitch = frequency;
19    tone(SPEAKER, pitch);
20  }
21
```

Sweet sweet PWM waves....

Ok, but let's learn how to get a bit more control over our tone by adding in a push button which will let us trigger our tone on and off making it more playable. Here we can also see all 3 of our main code sections in action: The declarations and variables for our PITCH_KNOB, TRIGGER and SPEAKER at the top of our sketch, code in our setup() function setting our TRIGGER pin as an input, and our main loop() where we get readings from our analog and digital pins to control the tone output. Here we can also see some important functions on display such as: analogRead, digitalRead as well as the ever useful 'if' statement. Here we see the 'if' statement at work polling our TRIGGER pin to see 'if' a LOW signal is present(connection to GND 0v) and if so, then execute the tone() code. Another if statement is used to turn the tone off if a HIGH signal is present(the pin's default state).



```
PITCH_KNOB_BUTTON_RSS_INTRO_TO_SOUND_SYNTHESIS_WITH_ARDUINO | A

✓ → ⚙ Select Board

PITCH_KNOB_BUTTON_RSS_INTRO_TO_SOUND_SYNTHESIS_WITH_ARDUINO.ino

1  /*
2   * Rhythm Science Sound
3   * introduction to electronic sound synthesis with Arduino
4   * rhythmsciencesound.com
5   */
6   const int SPEAKER = 9;
7   const int PITCH_KNOB = A0;
8   const int TRIGGER = 8;
9   int frequency;
10  int pitch;
11  int trigger_state;
12  void setup() {
13    // put your setup code here, to run once:
14    pinMode(TRIGGER, INPUT_PULLUP);
15  }
16
17
18  void loop() {
19    // put your main code here, to run repeatedly:
20
21    frequency = analogRead(PITCH_KNOB);
22    pitch = frequency;
23    trigger_state = digitalRead(TRIGGER);
24    if(trigger_state==LOW){
25      tone(SPEAKER, pitch);
26    }
27    if(trigger_state==HIGH){
28      noTone(SPEAKER);
29    }
30  }
31
```


Included in the chipBoard code resources are code examples which also show how to use tools like arrays which are special variables which store an index of values such as a range of notes/pitches for our tone() to chose from. This array list all notes chromatically, but one could choose to limit the array to notes of a particular scale, or notes of a melody, etc.

```
// Provide a list of the notes we want to be able play
int notes[] = {
  NOTE_B0,
  NOTE_C1, NOTE_CS1, NOTE_D1, NOTE_DS1, NOTE_E1, NOTE_F1, NOTE_FS1, NOTE_G1, NOTE_GS1, NOTE_A1, NOTE_AS1,
  NOTE_C2, NOTE_CS2, NOTE_D2, NOTE_DS2, NOTE_E2, NOTE_F2, NOTE_FS2, NOTE_G2, NOTE_GS2, NOTE_A2, NOTE_AS2,
  NOTE_C3, NOTE_CS3, NOTE_D3, NOTE_DS3, NOTE_E3, NOTE_F3, NOTE_FS3, NOTE_G3, NOTE_GS3, NOTE_A3, NOTE_AS3,
  NOTE_C4, NOTE_CS4, NOTE_D4, NOTE_DS4, NOTE_E4, NOTE_F4, NOTE_FS4, NOTE_G4, NOTE_GS4, NOTE_A4, NOTE_AS4,
  NOTE_C5, NOTE_CS5, NOTE_D5, NOTE_DS5, NOTE_E5, NOTE_F5, NOTE_FS5, NOTE_G5, NOTE_GS5, NOTE_A5, NOTE_AS5,
  NOTE_C6, NOTE_CS6, NOTE_D6, NOTE_DS6, NOTE_E6, NOTE_F6, NOTE_FS6, NOTE_G6, NOTE_GS6, NOTE_A6, NOTE_AS6,
  NOTE_C7, NOTE_CS7, NOTE_D7, NOTE_DS7, NOTE_E7, NOTE_F7, NOTE_FS7, NOTE_G7, NOTE_GS7, NOTE_A7, NOTE_AS7,
  NOTE_C8, NOTE_CS8, NOTE_D8, NOTE_DS8
};
```

There are also examples which show handling for multiple buttons and potentiometers to allow for more notes/keys.

```
if(trigger1_state==LOW){
  playingNote = pitch1;
}
if(trigger2_state==LOW){
  playingNote = pitch2;
}
if(trigger3_state==LOW){
  playingNote = pitch3;
}
if(trigger4_state==LOW){
  playingNote = pitch4;
}
if(trigger5_state==LOW) {
  playingNote = pitch1+12;
}
if(trigger1_state==LOW || trigger2_state==LOW || trigger3_state==LOW || trigger4_state==LOW ){
  tone(SPEAKER, notes[playingNote]);
}
else{
  noTone(SPEAKER);
}
```

But the main chipBoard sketch is where we can make some really wild sound!

Our main code project takes the ideas we've used so far for simple control over tones via knobs and buttons, and adds a timed note **sequencer**.

With the addition of a bit of code we can set up a **clock** which starts a **counter** in milliseconds(ms). Then, we can set a duration in our code controllable by a potentiometer connected to Analog pin A4, to control our note trigger timing. This will play and loop a 4 note sequence which is tunable by our potentiometers connected to Analog pins A0-A3. The 4 buttons shift the sequence by an amount set in the sketch. There is a potentiometer wired to Analog pin A5 to provide control over the macro pitch, shifting the notes of the entire sequence up or down.

With the tempo set at slow speeds, the buttons serve as a keyboard playing individual notes. With tempo set at higher speeds you get more rhythmic sequences, chord like bursts of notes, and multi-oscillating madness!

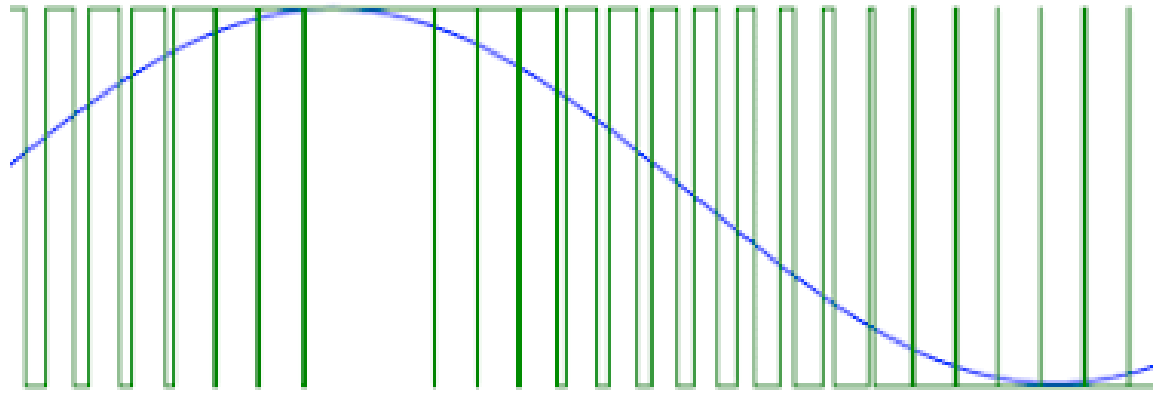
The chipBoard is monophonic(only playing a single note at a time). Playing multiple notes at once on the chipBoard causes the oscillators to combine in software; multiplying, distorting and glitching the output. While this could be dealt with('cleaned up') in code, we actually think this is kind of a cool feature and part of what makes the chipBoard(and many simple diy coding/synth projects) so sonically interesting....the way the sound behaves unpredictably past the edges of the parameters....into the wilds of digital synthesis.

We hope that in addition to having fun making bleeps and bloops, you'll experiment further with these code examples and make some other cool instruments. Experiment with the clocking code, limit the playable notes to scales, introduce new multi-button press functionalities.

All the code resources for the chipBoard kit are available on our website.
rhythmsciencesound.com/chipboard

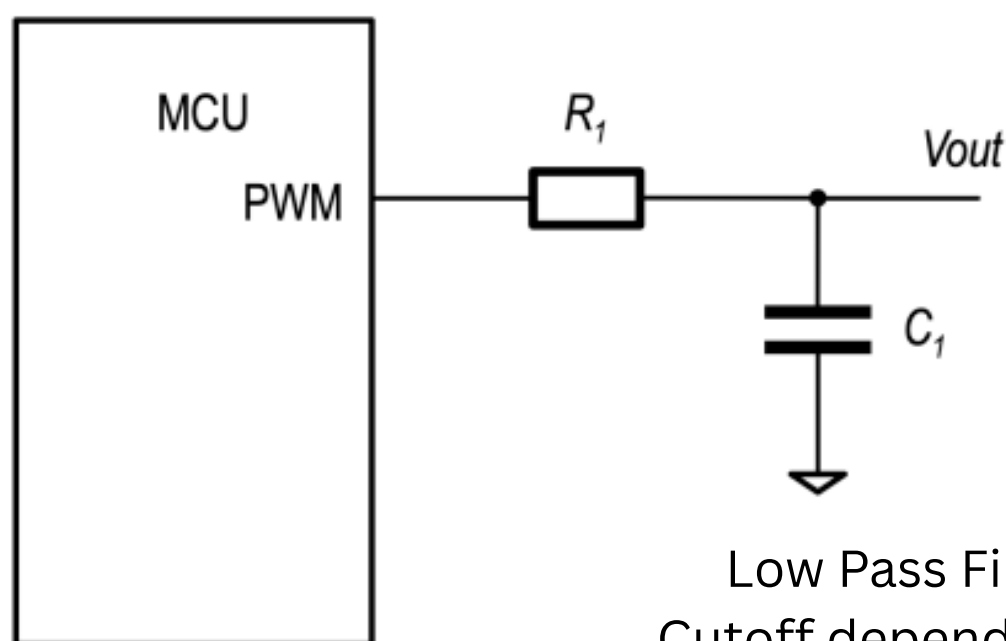
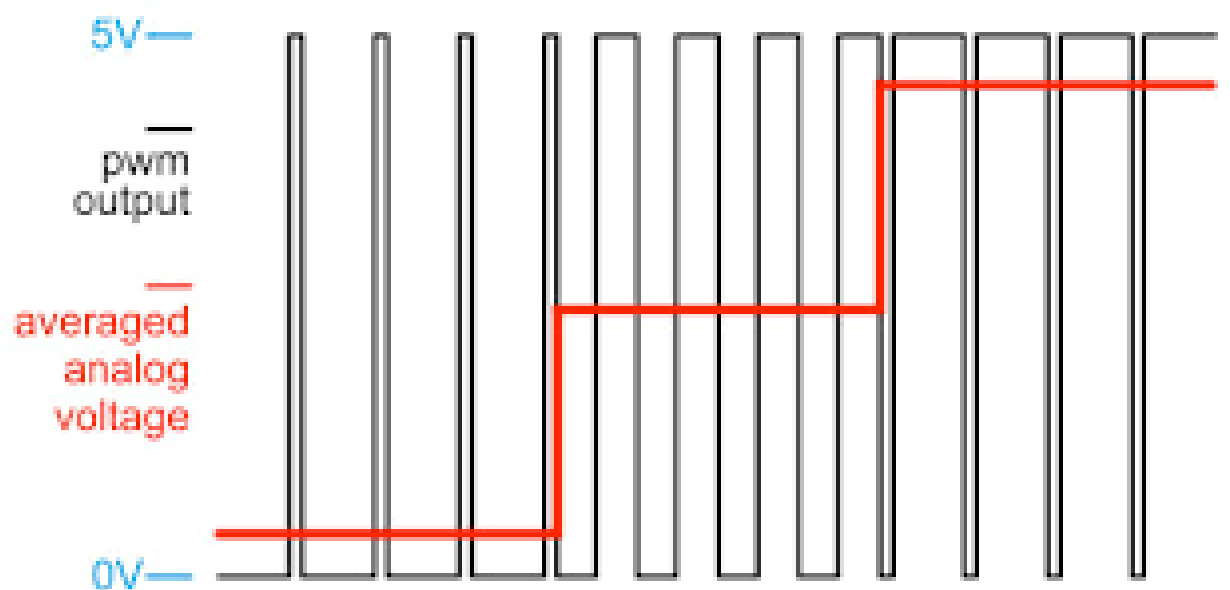
And when you make something cool, or if you have suggestions on how to make this kit a better learning experience, please reach out!

How does our Arduino make sound?



Our chipBoard synth produces sound through a technique of Pulse Width Modulation(PWM). PWM is achieved by oscillating between a high and low voltage for varying lengths of time(width of the pulse) which creates an average analog voltage. By accurately controlling the pulse width we can approximate an analog synth waveform through digital means.

This is great, but PWM can sound a bit harsh, especially in the upper frequencies. We further condition the signal by putting this PWM signal through a low pass filter comprised of a resistor and capacitor(connected to GND).

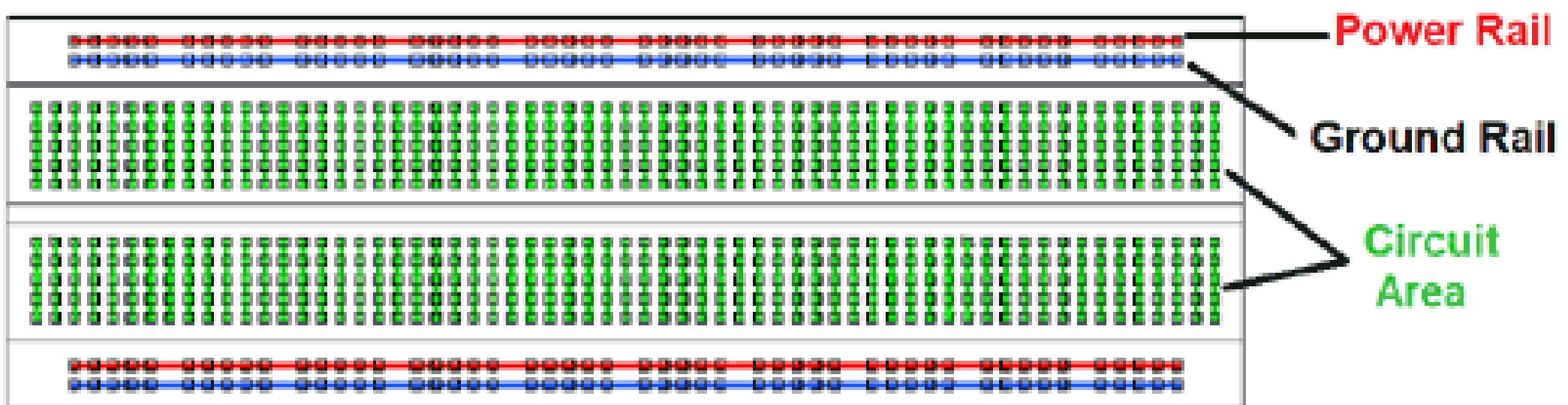


Low Pass Filter circuit.
Cutoff dependent on values
of resistor and capacitor

The breadboard

Breadboards are a convenient tool for prototyping and building electronic circuits. They provide power rail distribution and a regular grid system for connecting components with leads (the metal legs) or pins such as Integrated Circuit chips, resistors, capacitors, potentiometers, buttons, LEDs, etc.

Note that the circuit area is divided down the middle creating two unconnected halves of the design area. Each side features its own power rail for + voltage and gnd



For our chipBoard synth we'll be getting power from the Arduino Nano itself, which draws 5 volts of power through the USB connection and makes it available to other circuitry via its 5V and Gnd pins. We'll wire these to the Power and Ground rails of our breadboard to provide 5v and 0v Gnd connections to any components that require them.

*Fun note: breadboards are so named because in the early days of electronics, engineers literally wired up circuits using metal nails, wire and wooden breadboards from their kitchens as a base to hold their prototyped circuit together.

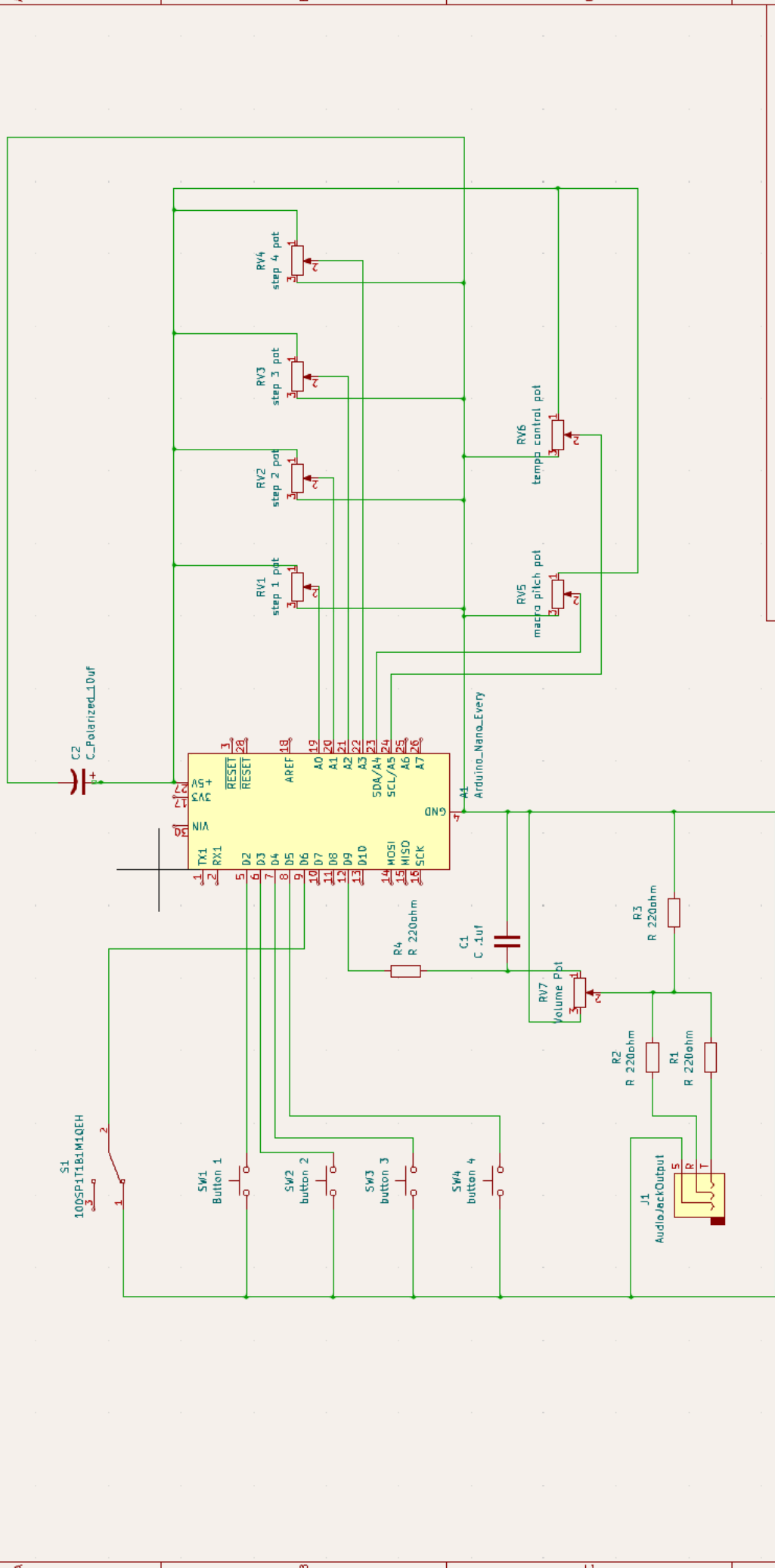
The “ChipBoard” Arduino Synthesizer Kit

Bill of Materials(BOM)

More info at rhythmsciencesound.com



PART NAME	QUANTITY NEEDED	NOTE
Arduino Nano Microcontroller	1	The “Chip” in the kit where we upload our code and where we connect our sensors like buttons, switches, potentiometers, etc.
Breadboard prototyping surface	1	The “Board” of our kit which lets us connect all our components together into a project that is literally greater than the sum of its parts!
Variable Resistor Potentiometer	7	20kohms. Set up to send variable voltage to Analog Input pins
Light Dependent Resistor(LDR) - (Optional)	2	Set up to send a variable voltage to Analog Input pins dependent on light source
Push Button Tactile Switch	4	Connect to digital IO pins for momentary switch
Toggle Switch	1	Connect to Digital IO pin for latched switch
Polarized Capacitor	1	10uf Placed in circuit across power supply + and - . Used to stabilize power supply(decoupling capacitor)
Ceramic disc Capacitor	1	.1uf Used in the part of the circuit for our audio signal output.
Resistor	6	220ohm 4 resistors used in the part of the circuit for our audio signal output. 2 100k ohm resistors for use with LDRs
Audio Jack	1	3.5mm headphone/aux jack
Jumper wires long	3	Used to make long connections on your breadboard
Jumper wires medium	3	Used to make medium connections on your breadboard
Jumper wires short	9	Used to make short connections on your breadboard
Pre-cut hookup wires	23-25	Used to make very short connections on your breadboard. Useful for connecting components to power supply + and - rails.

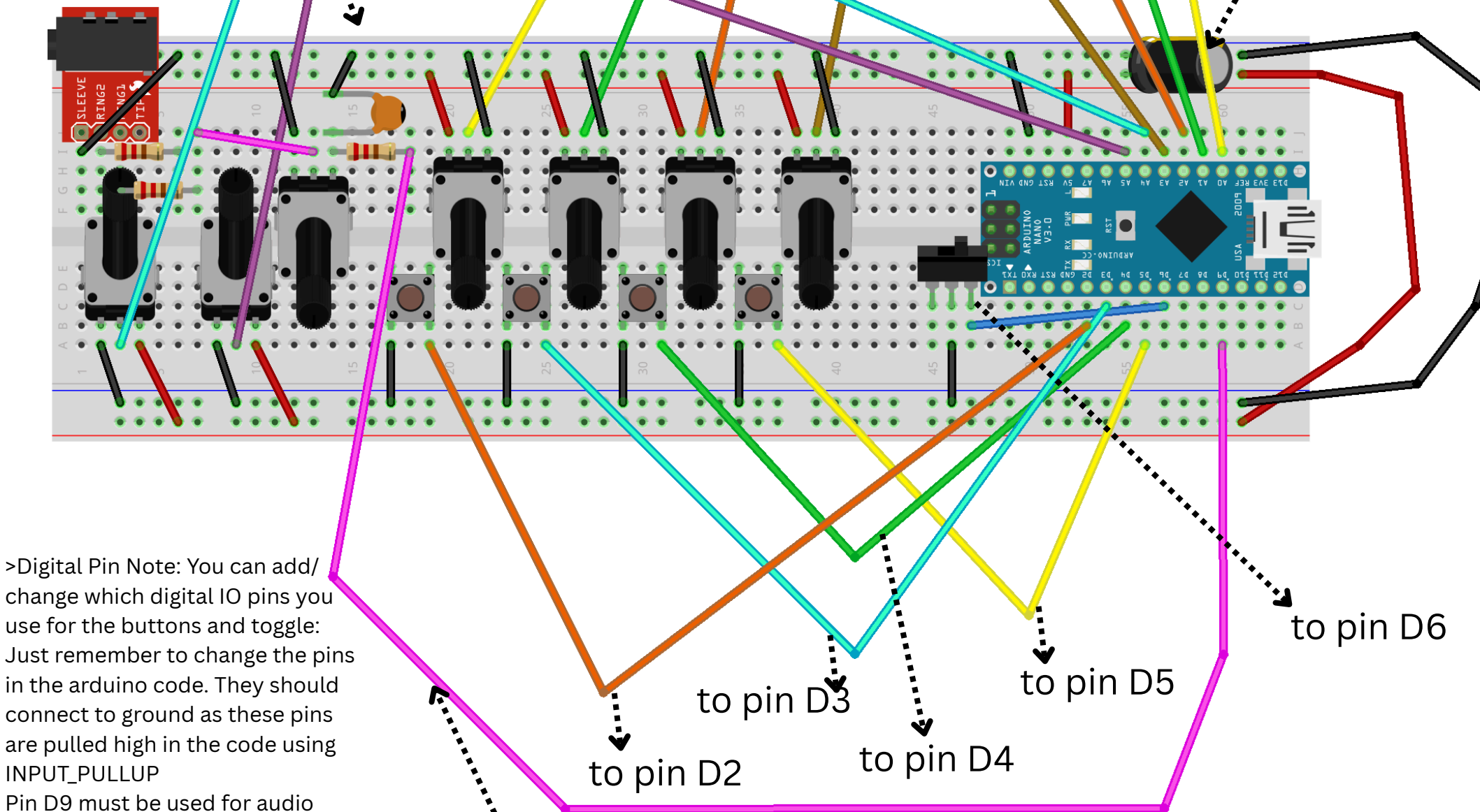


Sheet: /		File: RSS chipBoard DIY Arduino Synth Kit kicad_sch	
Size: A4		Date:	
KICad E.D.A.		Kicad 7.0.10	
Title: RSS chipBoard DIY Arduino Synth Schematic		Rev:	
Id: 1/1			

Resistor/Capacitor
filter circuit

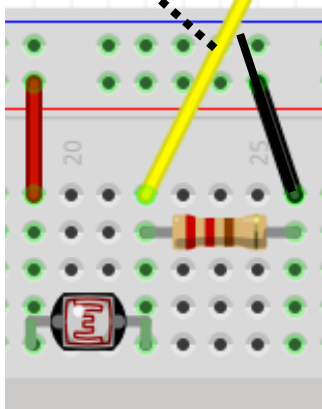
>Audio Jack wiring:
We can split our
single audio signal
using two resistors
connected to RING1
and RING2 of our
audio jack, SLEEVE
is connected to Ground,
and TIP is left
unconnected.

>Analog Pin Note: Analog input
pins(A0-A5) should be
connected to potentiometers(or
LDRs) which serve as a voltage
divider between our power
supply rails(5v + and Ground).
Feel free to change the order of
analog inputs and/or to swap the
right and left pins of the
potentiometers connections to
5v and Gnd to get reverse knob
behavior



>Digital Pin Note: You can add/
change which digital IO pins you
use for the buttons and toggle:
Just remember to change the pins
in the arduino code. They should
connect to ground as these pins
are pulled high in the code using
INPUT_PULLUP
Pin D9 must be used for audio
output.

to Analog Pin

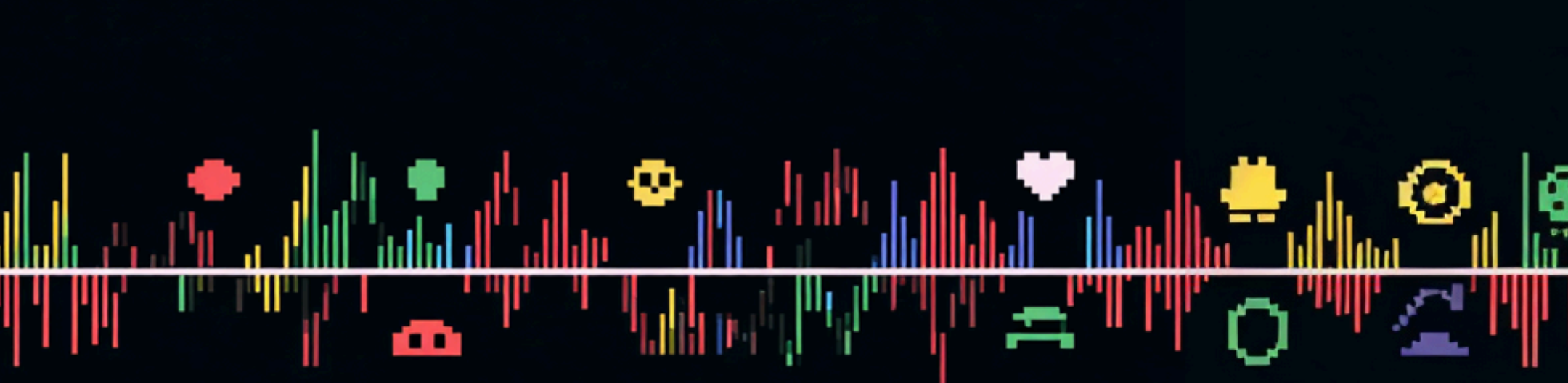


Light Dependent Resistors(LDRs)

You can switch out any of the Potentiometers providing
voltage control to the Analog Input pins(A0-A5) with Light
Dependent Resistors(LDRs) by wiring one lead of the LDR
to +5v, and the other lead connected to 0v GND through a
resistor(100k recommended). The Analog pin of your
choice can then be connected to the junction of the 2nd
lead and resistor. This will give you fun and dynamic light
control over this parameter in the code!

>Connections to the +(positive
voltage) are shown with red wire:
Connections to the -(ground, 0
volts,) are shown with black wire:

>The decoupling capacitor is
polarized and should be mounted
with the Anode(longer leg)
connected to + power rail, and the
Cathode(shorter leg, marked with
gray stripe) should connect to
ground. This stabilizes our power
supply.



chipBoard Main Code Breakdown

The Big Picture

This sketch turns your Arduino into a 4-step sequencer with pitch control, tempo adjustment, and button-triggered note shifting. Think of it as a mini step sequencer that loops through 4 notes, with each note's pitch controlled by its own knob, all running on an adjustable clock.

Pin Declarations (The Hardware Map)

cpp



```
const int SPEAKER = 9;
const int PITCH_KNOB1 = A0;
const int PITCH_KNOB2 = A1;
const int PITCH_KNOB3 = A2;
const int PITCH_KNOB4 = A3;
const int TEMPO_KNOB = A4;
const int MACRO_PITCH_KNOB = A5;
const int TRIGGER1 = 2;
const int TRIGGER2 = 3;
const int TRIGGER3 = 4;
const int TRIGGER4 = 5;
const int TRIGGER5 = 6;
const int LEDCLOCK = 13;
```

What's happening: We're giving friendly names to all our pins. The speaker outputs sound on pin 9. Four pitch knobs (A0-A3) control individual step pitches. The tempo knob (A4) controls sequence speed. The macro pitch knob (A5) shifts the entire sequence up or down. Five trigger buttons (pins 2-6) shift the playing notes, and pin 13 drives a clock LED.

State Variables (Keeping Track of Things)

cpp



```
int trigger1_state;
int trigger2_state;
int trigger3_state;
int trigger4_state;
int trigger5_state;
int frequency1;
int frequency2;
int frequency3;
int frequency4;
int pitch1;
int pitch2;
int pitch3;
int pitch4;
```

What's happening: These variables store the current state of each button (pressed or not) and pitch values. Note that `frequency1` through `frequency4` are declared but never actually used in the code—they're leftover variables from an earlier version!

The Note Library (Musical DNA)

cpp

```
#define NOTE_B0  31
#define NOTE_C1  33
// ... (all the note definitions)
#define NOTE_DS8 4978

int notes[] = {
    NOTE_B0,
    NOTE_C1, NOTE_CS1, NOTE_D1, ...
};
```

What's happening: The `#define` statements create constants for musical note frequencies in Hertz. The `notes[]` array organizes all these frequencies into a list we can reference by index number. Index 0 is NOTE_B0 (31 Hz), index 1 is NOTE_C1 (33 Hz), and so on up through NOTE_DS8 (4978 Hz). This gives us about 8 octaves of chromatic notes to play with!

Timing Variables (The Clock Mechanism)

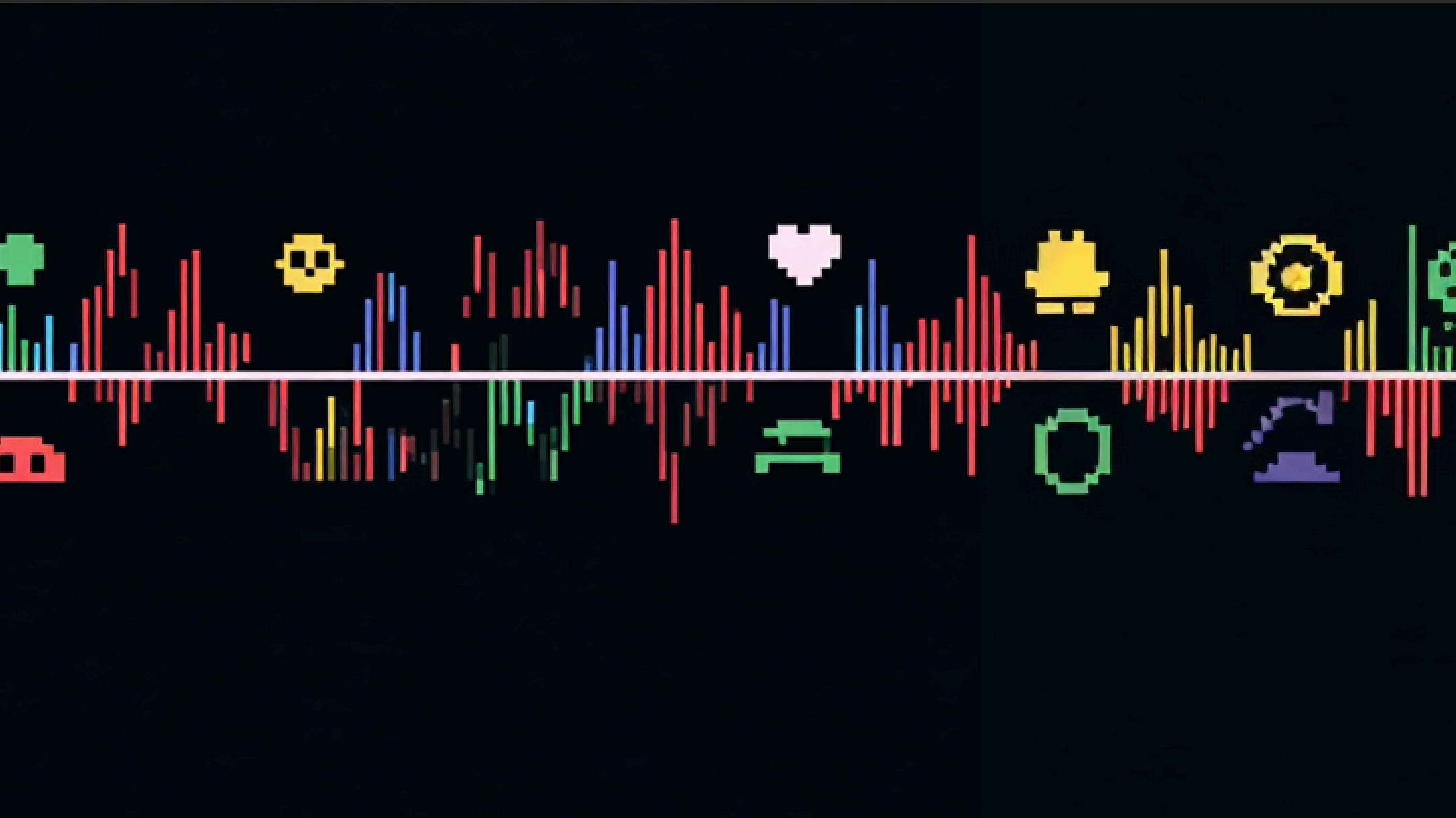
```
cpp

int numNotes;
int playingNote;
unsigned long startMillis;
unsigned long currentMillis;
const unsigned long period = 0;
```

What's happening:

- `numNotes` will store how many notes are in our array
- `playingNote` tracks which step (0-3) in the sequence we're currently on
- `startMillis` remembers when we last advanced a step
- `currentMillis` checks what time it is now
- `period` is declared here but gets recalculated in the loop based on the tempo knob

The `unsigned long` data type can hold really big numbers (up to 4,294,967,295), which is perfect for millisecond counting that never stops!



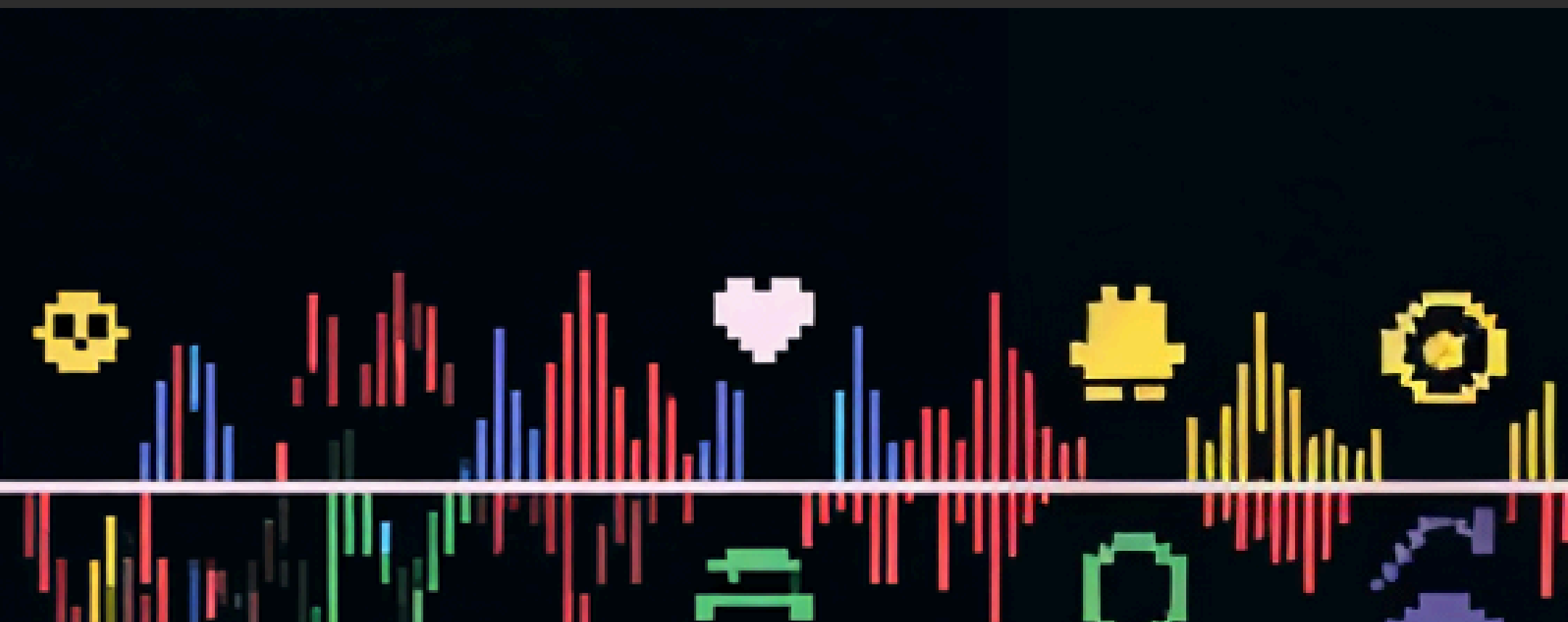
Setup Function (The Bootup Routine)

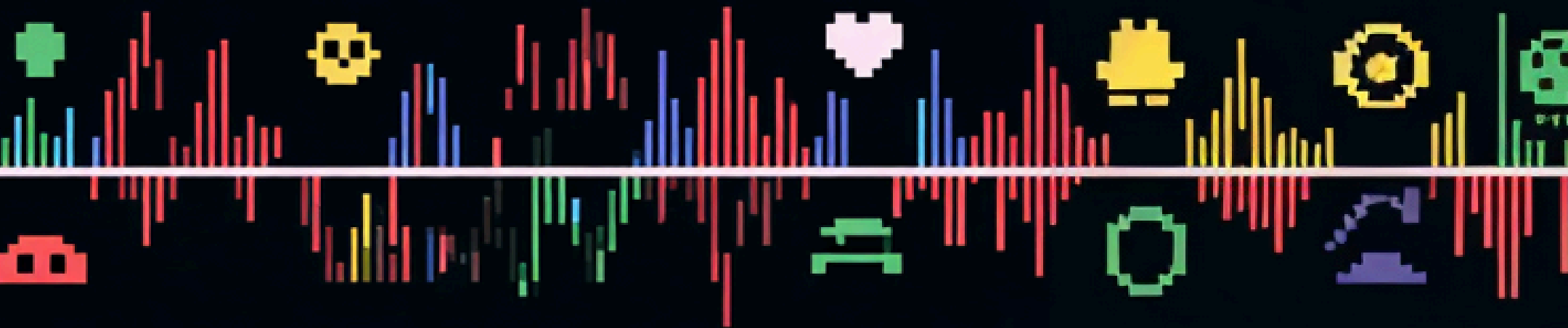
cpp

```
void setup() {  
  pinMode(TRIGGER1, INPUT_PULLUP);  
  pinMode(TRIGGER2, INPUT_PULLUP);  
  pinMode(TRIGGER3, INPUT_PULLUP);  
  pinMode(TRIGGER4, INPUT_PULLUP);  
  pinMode(TRIGGER5, INPUT_PULLUP);  
  pinMode(LEDLOCK, OUTPUT);  
  
  numNotes = sizeof(notes)/sizeof(notes[0]);  
  playingNote=0;  
  startMillis = millis();  
}
```

What's happening:

- `INPUT_PULLUP` mode means the buttons read HIGH when not pressed, and LOW when pressed (the internal pullup resistor keeps the pin at 5V until the button grounds it)
- The LED clock pin is set as an OUTPUT so we can blink it
- `sizeof(notes)/sizeof(notes[0])` is a clever trick: it divides the total byte size of the array by the size of one element, giving us the count of elements (about 89 notes!)
- We start at step 0 of our sequence
- `millis()` returns the number of milliseconds since the Arduino powered on—this starts our clock





Main Loop (Where the Magic Happens)

Reading the Controls

cpp

```
digitalWrite(LEDCLK, LOW);

int MacroPitchPot = analogRead(MACRO_PITCH_KNOB);
int MacroPitch= map(MacroPitchPot, 0, 1023, 0, notes-1);
```

What's happening: First we turn off the clock LED. Then we read the macro pitch knob (A5), which returns a value from 0-1023. The `map()` function converts this to a range from 0 to (number of notes - 1), so we can shift the entire sequence by that many semitones.

Pitch Calculation

cpp

```
int potReading = analogRead (PITCH_KNOB1+playingNote);
int globalPitch = playingNote + macroPitch;
int pitch = map(potReading, 0, 1023, 0, globalPitch);
```

What's happening: This is super clever! Since `PITCH_KNOB1` is A0, adding `playingNote` (0-3) gives us A0, A1, A2, or A3—automatically reading the correct knob for the current step! The global pitch combines the step number with the macro shift. Then we map the knob reading to a pitch index from 0 to that global pitch value.

Result: Each step's knob controls how high (within the available range) that note plays.



Reading All the Buttons

cpp

```
trigger1_state = digitalRead(TRIGGER1);  
trigger2_state = digitalRead(TRIGGER2);  
trigger3_state = digitalRead(TRIGGER3);  
trigger4_state = digitalRead(TRIGGER4);  
trigger5_state = digitalRead(TRIGGER5);
```

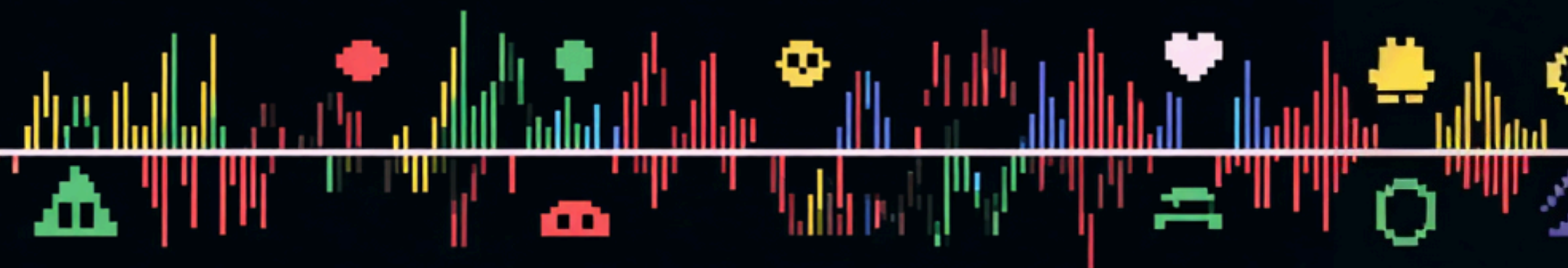
What's happening: We check all five buttons to see if they're pressed (LOW) or not (HIGH). This happens every loop cycle—hundreds of times per second!

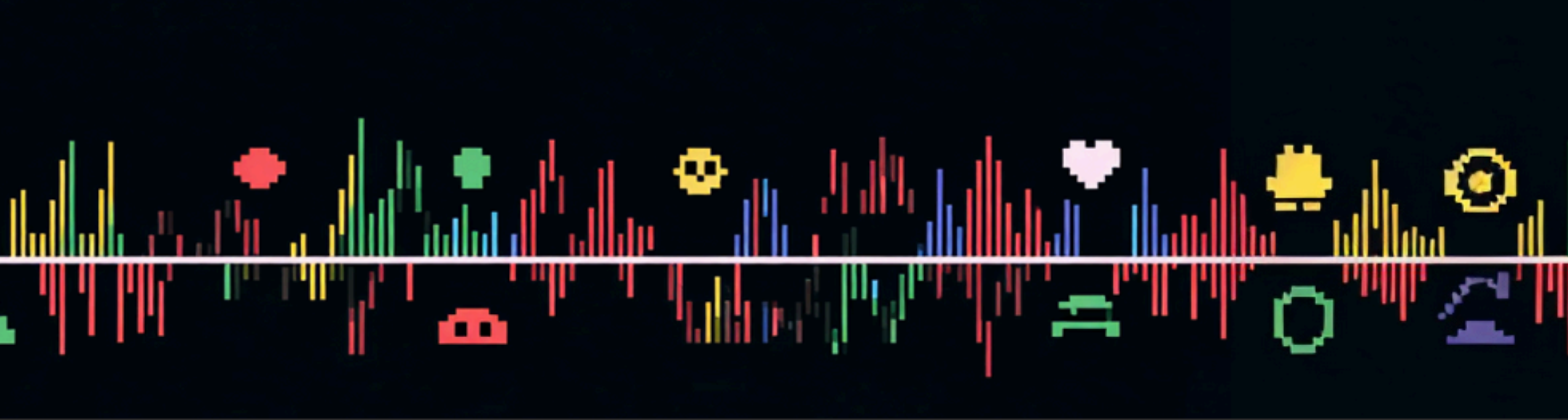
Sequence Reset

cpp

```
if (playingNote >= 4) {  
    playingNote = 0;  
    Serial.print("\n");  
}
```

What's happening: If we've advanced past step 3 (to step 4 or higher), we loop back to step 0. This keeps our sequence at 4 steps.





Button Triggers (Note Shifting)

cpp

```
if (trigger1_state == LOW ) {  
    pitch = pitch;  
    tone (SPEAKER, notes[pitch]);  
}  
if(trigger2_state == LOW){  
    pitch = pitch+1;  
    tone (SPEAKER, notes[pitch]);  
}  
if(trigger3_state == LOW){  
    pitch = pitch+2;  
    tone (SPEAKER, notes[pitch]);  
}  
if(trigger4_state == LOW){  
    pitch = pitch+3;  
    tone (SPEAKER, notes[pitch]);  
}  
if(trigger5_state == LOW){  
    pitch = pitch+4;  
    tone (SPEAKER, notes[pitch]);  
}
```

What's happening: When a button is pressed, we shift the pitch by a certain number of semitones (0, 1, 2, 3, or 4) and play that note from our array using `tone()`. The `tone()` function generates a square wave at the specified frequency on the speaker pin.

The Glitch Feature: If you press multiple buttons at once, the code will call `tone()` multiple times in rapid succession. Since the Arduino can only play one tone at a time, they interrupt each other, creating those wild distorted sounds we mentioned—the oscillators "combining in software"!

Silence Detection (All Buttons Released)

```
cpp

if(trigger1_state==HIGH && trigger2_state==HIGH && trigger3_state==HIGH && trigger4_state==HIGH && trigger5_state==HIGH) {
    playingNote = 0;
    noTone(SPEAKER);
}
```

What's happening: If ALL five buttons are released (all HIGH), we stop the sound with `noTone()` and reset the sequence to step 0. This is your silence/reset state.

The Clock (Sequence Advancement)

```
cpp

int bpm = analogRead (TEMPO_KNOB);
int period = map(bpm, 0, 1023, 1000, 5);
currentMillis = millis();
if (currentMillis - startMillis >= period) {
    playingNote++;
    digitalWrite(LED_CLOCK, HIGH);
    startMillis = currentMillis;
}
```

What's happening: This is the sequencer heart!

1. Read the tempo knob (A4)
2. Map it to a period from 1000ms (slow—1 beat per second) down to 5ms (crazy fast—200 beats per second)
3. Check the current time
4. If enough time has passed since the last step advance, move to the next note, flash the LED, and reset the timer

Why milliseconds? The `millis()` function is non-blocking—unlike `delay()`, it lets your code keep running while tracking time. This is crucial for responsive button presses!

At slow tempos, you have time to press buttons and play individual notes. At fast tempos, the sequence races through, and button presses create chord bursts since you're triggering multiple rapid notes.

The Wild Edges Explained

The chipBoard's "glitchy" behavior comes from:

1. **Multiple simultaneous tones:** Each `tone()` call interrupts the previous one, creating digital distortion
2. **Fast tempo + button presses:** The sequence advances while you're adding pitch shifts, creating chaotic rhythmic patterns
3. **Pitch calculations near array boundaries:** If `pitch` gets calculated higher than `numNotes-1`, you'll read past the array into random memory—pure digital chaos!

These aren't bugs... they're features that make the chipBoard sonically wild and unpredictable! They also show the creative/sonic value for experimenting and testing tools and rules beyond their limits to see occurs...it makes for fun experimentation and good learning!

Experiments to Try

- **Add scales:** Instead of using the full chromatic `notes[]` array, create smaller arrays with just pentatonic or blues scale notes
- **Variable sequence length:** Make the sequence 8 or 16 steps instead of 4
- **Probability triggers:** Use `random()` to sometimes skip notes or add rests
- **Reverse mode:** Make the sequence play backwards when a button is held
- **Tempo sync:** Make the clock LED blink patterns that subdivide the beat

Happy bleeping and blooping! 🎵

