ESP32 4-Voice Synthesizer

with emulation of the AY Arcade Game Sound Chip

John Talbert - November 2021

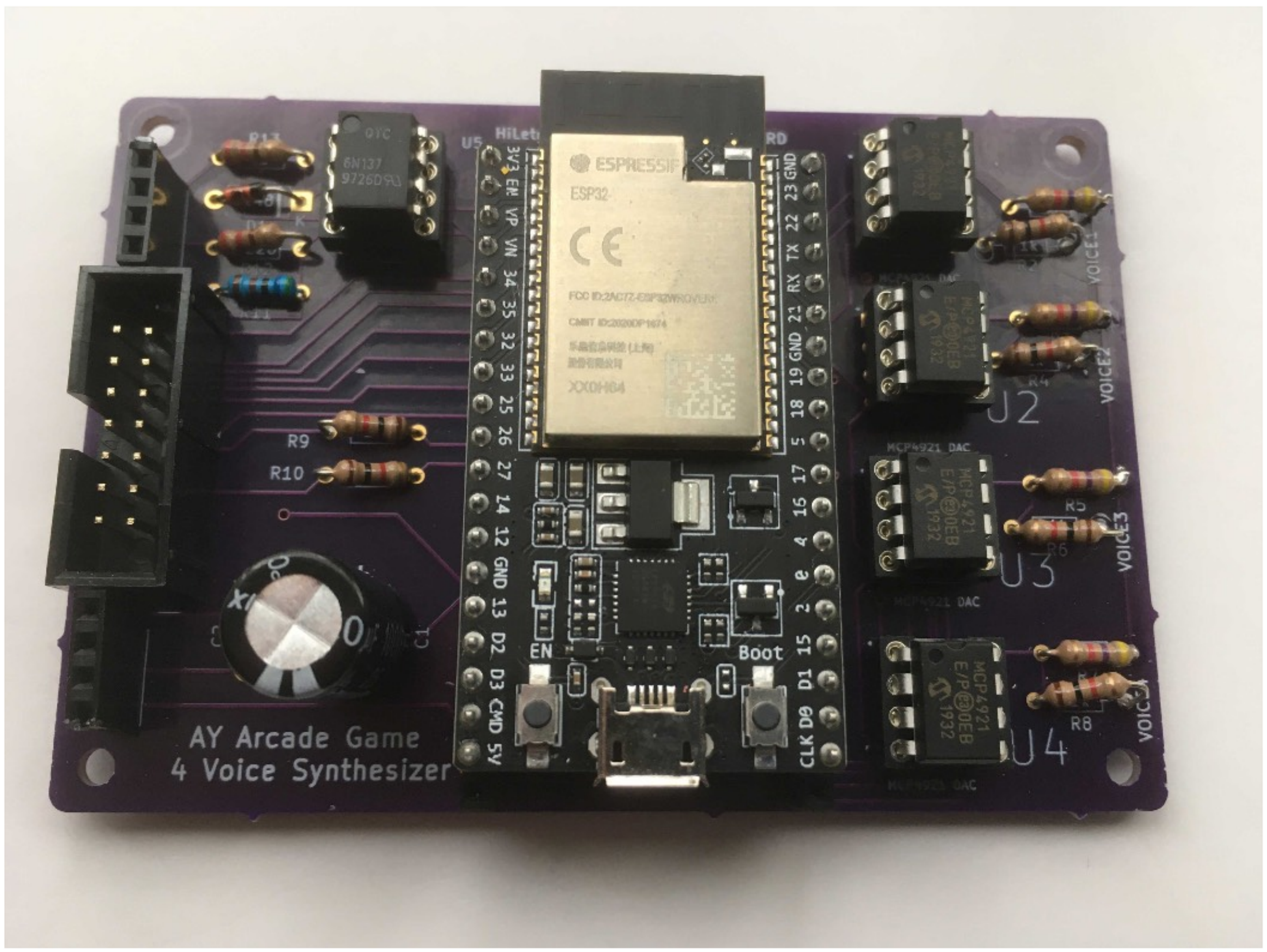


Table of Contents

the Synth	3
the Board	4
the Box	6
the Programming	16
Basic	18
LED PWM	23
MIDI Input and Output	28
DAC Waveforms	33
Touch Switch	37
PWM Noise Voice	40
Timer Interrupt Voices	46
Timer Voices with 2 Cores	52
Random Voices	58
SPIFFS File System	65
AY Arcade Playback	71

the Synth

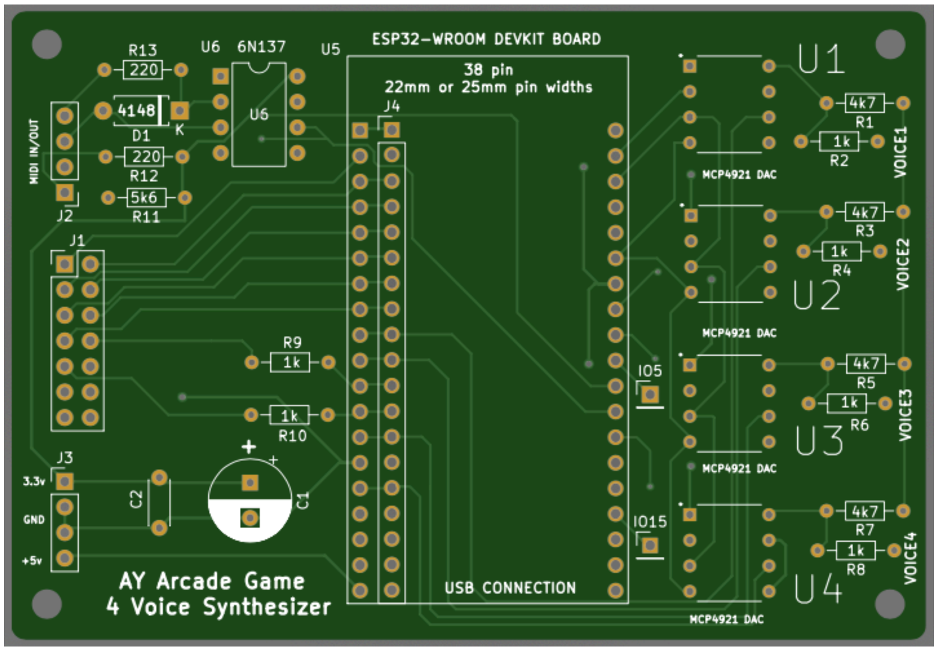
This is a four-voice programmable synthesizer built with the ESP32 DevKit. The four voices are built from signals generated on 4 ESP32 output pins connected to 4 multiplying MCP4921 DAC chips used for volume control. Two of the pins can be programmed as 8-bit DACs to produce any type of waveform. All four can be programmed to produce square waves and PWM pulse waves.   If the pulse width is fed random values the pulse wave becomes pitched noise.

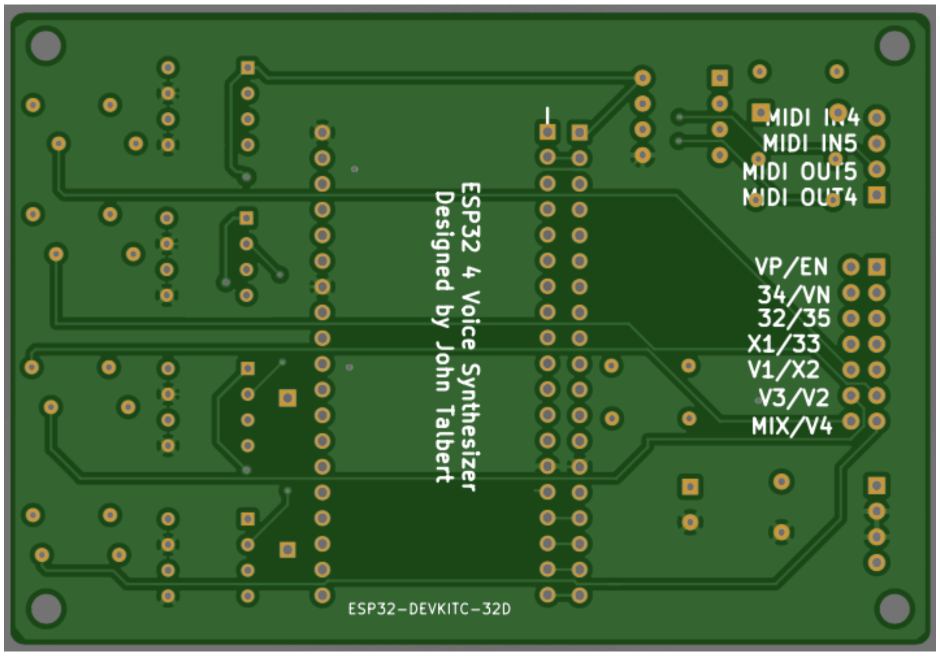
Two additional ESP32 pins are connected to onboard circuits for MIDI INPUT and OUTPUT.

Among its many applications, the Synth can be programmed to simulate an AY Arcade Game Sound chip from the 70's and play back old AY sound files still available on the web.

the Board

A PCB board was designed using the free, open source, KiCad app (www.kicad.org) and Dr. Peter Dalmaris' very useful book and tutorial "KiCad Like A Pro" (https://techexplorations.com).



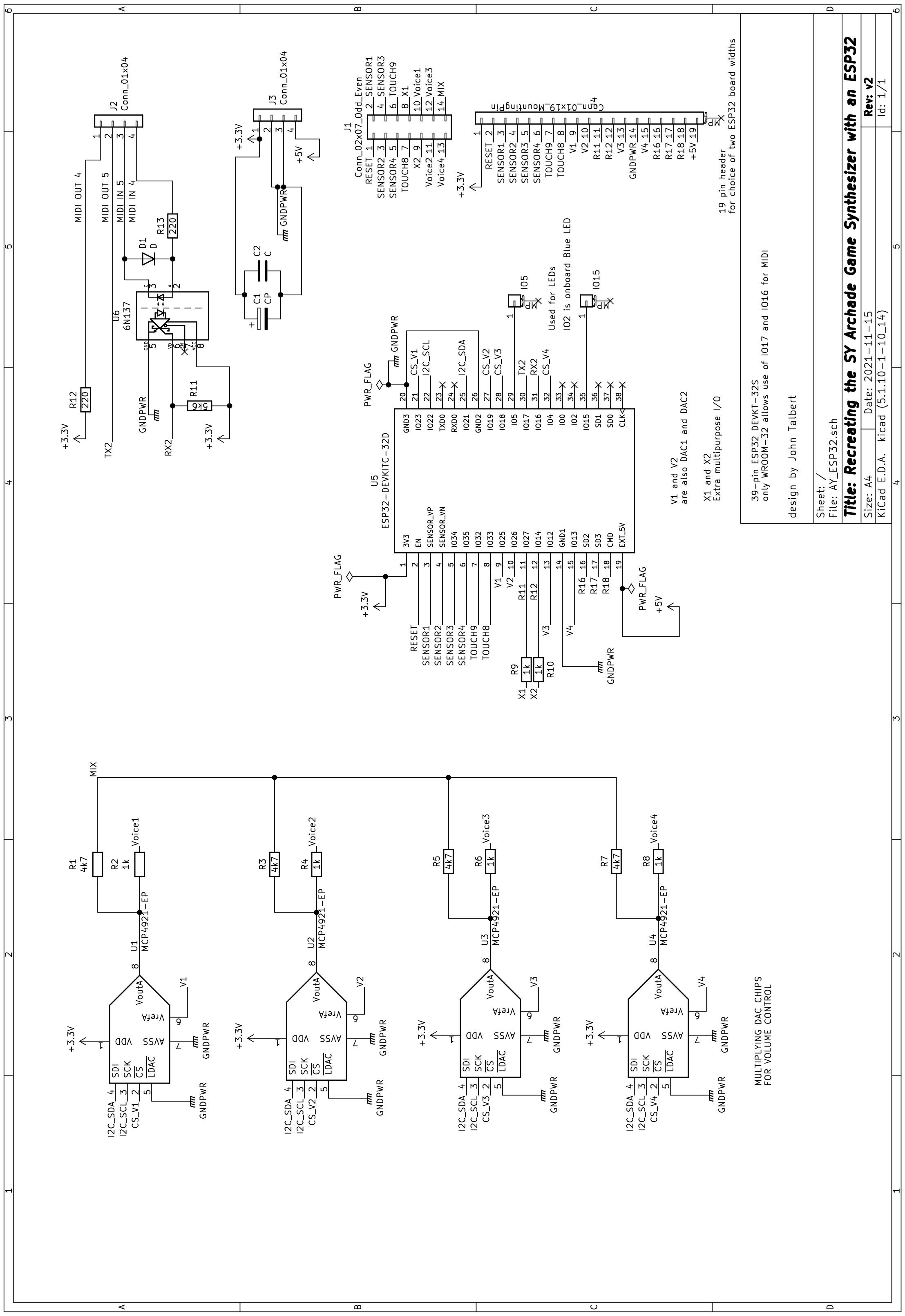


The central device on the PCB is the ESP32-DEVKITC-32D board. This is a 39 pin board with a USB connection to both program and power the ESP32 microprocessor. There are several versions of this board. The WROOM version has pins 30 and 31 (IO16 and 17) available as TX2 and RX2 for MIDI Input and Output.. The PCB board has an extra 19 pin header to accommodate two different versions of the DEVKIT with pin widths of 22mm or 25mm.

Four MCP4921-EP Multiplying DAC chips provide volume control. Any signal connected to the VrefA input of the Multiplying DAC chip can be volume controlled. The ESP32 pins chosen for this are IO25, IO26, IO12 and IO13 (pins 9, 10, 13, 15). All four pins can be programmed as digital outputs that produce square wave or pulse type signal outputs. IO25 and IO26 can also be programmed as 8-bit DACs and, as such, can output any waveform shape.

The volume of the connected signal on each of these four multiplying DAC chips is controlled through an I2C interface on the DAC chips, set up using ESP32 pins IO22 and IO21 (pins 22 and 25) programmed as I2C_SCL and I2C_SDA and connected to all four DAC chips. The individual DAC chips are enabled one at a time to recognized the common I2C signals through four ESP32 lines (LOW enabled) IO23, IO19, IO18, and IO4 (pins 23, 19, 18, and 32).

The 4 volume controlled outputs from these 4 DAC chips are connected to the J1 output pin header both individually through a 1k series resistor as Voice 1, 2, 3, and 4, and added together in a 4k7 resistor mixer as MIX.



Title: Recreating the SY Archade Game Synthesizer with an ESP32
Sheet: /
File: AY_ESP32.sch
Size: A4
Date: 2021-11-15
Rev: v2
KiCad E.D.A. kicad (5.1.10-1-10_14)
Id: 1/1
design by John Talbert
39-pin ESP32 DEVKT-32S
only WROOM-32 allows use of IO17 and IO16 for MIDI
V1 and V2
are also DAC1 and DAC2
X1 and X2
Extra multipurpose I/O
Used for LEDs
IO2 is onboard Blue LED
19 pin header
for choice of two ESP32 board widths
MULTIPLYING DAC CHIPS
FOR VOLUME CONTROL
U5
ESP32-DEVKITC-32D
U6
6N137
J1
Conn_02x07_Odd_Even
J2
Conn_01x04
J3
Conn_01x04
J4
Conn_01x19_MountingPin
U1
U2
U3
U4
MCP4921-EP
MIDI OUT 4
MIDI OUT 5
MIDI IN 5
MIDI IN 4
MIX
PWR_FLAG
GNDPWR
+3.3V
+5V

MCP4921/4922

12-Bit DAC with SPI™ Interface

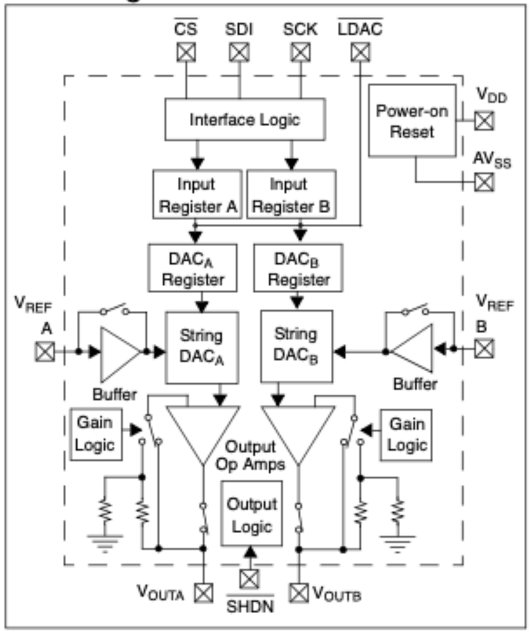
Features

- 12-Bit Resolution
- ±0.2 LSB DNL (typ)
- ±2 LSB INL (typ)
- Single or Dual Channel
- Rail-to-Rail Output
- SPI™ Interface with 20 MHz Clock Support
- Simultaneous Latching of the Dual DACs w/$\overline{\text{LDAC}}$
- Fast Settling Time of 4.5 μs
- Selectable Unity or 2x Gain Output
- 450 kHz Multiplier Mode
- External V_{REF} Input
- 2.7V to 5.5V Single-Supply Operation
- Extended Temperature Range: -40°C to +125°C

Applications

- Set Point or Offset Trimming
- Sensor Calibration
- Digitally-Controlled Multiplier/Divider
- Portable Instrumentation (Battery-Powered)
- Motor Feedback Loop Control

Block Diagram



Description

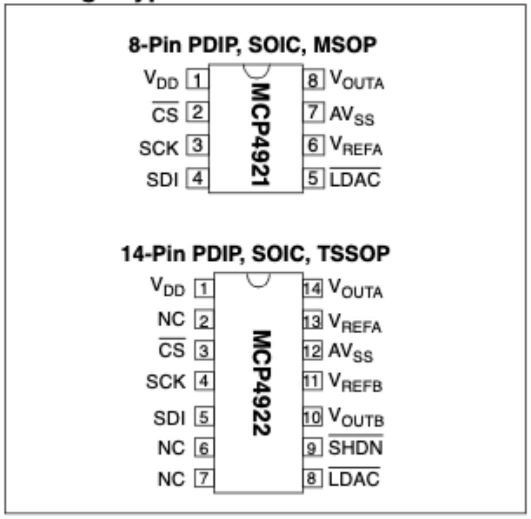
The Microchip Technology Inc. MCP492X are 2.7 – 5.5V, low-power, low DNL, 12-Bit Digital-to-Analog Converters (DACs) with optional 2x buffered output and SPI interface.

The MCP492X are DACs that provide high accuracy and low noise performance for industrial applications where calibration or compensation of signals (such as temperature, pressure and humidity) are required.

The MCP492X are available in the extended temperature range and PDIP, SOIC, MSOP and TSSOP packages.

The MCP492X devices utilize a resistive string architecture, with its inherent advantages of low DNL error, low ratio metric temperature coefficient and fast settling time. These devices are specified over the extended temperature range. The MCP492X include double-buffered inputs, allowing simultaneous updates using the $\overline{\text{LDAC}}$ pin. These devices also incorporate a Power-On Reset (POR) circuit to ensure reliable power-up.

Package Types

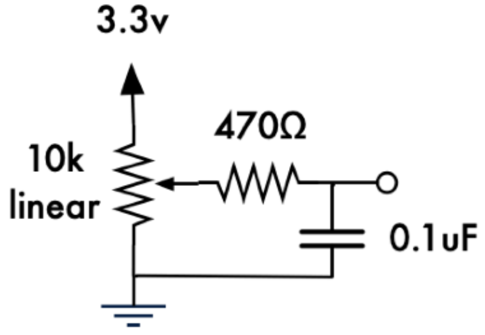


Two ESP32 pins are used to create a MIDI INPUT and MIDI OUTPUT. IO17 (pin 30) is TX2 used for MIDI Output, and IO16 (pin 31) is RX2 used for MIDI Input. Only a 6N137 opto-isolator chip and a few resistors are needed to transform TX2 and RX2 into MIDI In and MIDI Out, all of which is placed in the upper corner of the PCB board along with a 4-pin J2 header. J2 has two connections each for standard 5-pin DIN MIDI IN and MIDI OUT sockets.

Note that only the WROOM ESP32 makes IO17 and IO16 available for TX2 and RX2. The WROVER does not.

The J1 14-pin header is provided to carry the four voice outputs and the MIX output. This J1 header also carries other ESP32 connections, described as follows.

There are 4 sensor input pins on J1 from the ESP32 — Sensor_VP, Sensor_VN, IO32 and IO35 (pins 3, 4, 5, and 6). These can be connected to potentiometers wired between Ground and 3.3volts. It is good practice to put a 470 ohm resistor in series with the potentiometer wiper before connecting it to the ESP32 input pin. This will limit the current if the input pin is mistakenly programmed as an output.



Two more pins are provided on J1 for switches — IO32 and IO33 (pins7 and 8). An internal pull-up resistor can be programmed for the switch action. One side of the switch is connected to the ESP32 pin while the other side is connected to ground through a 270 ohm resistor, again, to protect against an input pin mistakenly programmed as an output and shorting to ground.

These two switch pins can also be programmed as Touch sensors T9 and T8. For Touch sensor applications the pins are simply connected to any metallic object such as a screw insulated from the metal chassis.

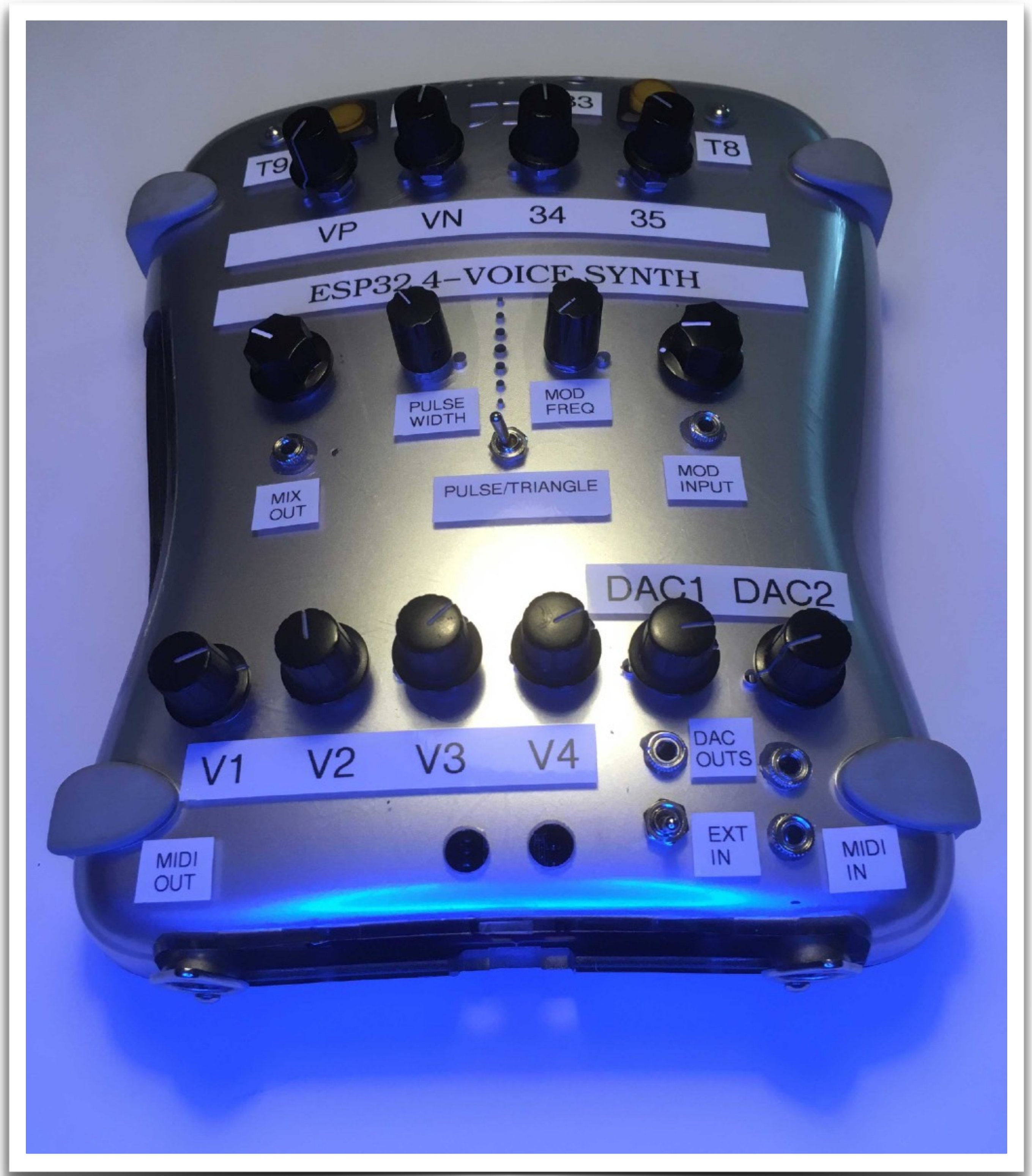
Finally, J1 has two extra pin connections labeled X1 and X2 connected to IO27 and IO14 (pins 11 and 12) through series 1k resistors. These two connections can be configured as two extra digital voices, sensor inputs, or whatever the user determines.

Outside of the J1 connector, IO5 and IO15 are provided individual pads for optional uses such as LED indicator lights. Note that some DEVKIT boards set up a blue LED on IO2 (pin 34).

That puts to use all the available I/O pins on the ESP32. It is recommended that you don't use pins SD2, SD3, CMS, IO0, IO2, SD1, SD0 and CLK (pins 16, 17, 18, 33, 34, 36, 37, 38) as those pins have internal uses such as flash memory and startup.

One more 4-pin header, J3, is provided on the PCB board to carry power connections 3.3 volts, 5 volts, and Ground. These can be used to power external devices such as Potentiometers, Switches, LEDs, and other sensors.

the Box


T9
T8
VP VN 34 35
ESP32 4-VOICE SYNTH
PULSE WIDTH
MOD FREQ
MIX OUT
PULSE/TRIANGLE
MOD INPUT
DAC1 DAC2
V1 V2 V3 V4
DAC OUTS
EXT IN
MIDI OUT
MIDI IN

Here is the AY ESP32 Synthesizer built into a chassis. All 4 voices from the PCB are used along with the two extra X1 and X2 pins for a total of six. Volume controls are provided for all six voices at the bottom of the box. A switch is put on one of the ESP32 DAC voices to allow a choice between the DAC and any external signal to be added to the mix. Output jacks are provided for each of the ESP32 DAC signal pins (before volume control).

A special second PCB board was designed to mix the 6 unipolar signals and to add optional modulation. The final modulated mix is provided on a mini-jack at the center of the box. A modulator input jack allows a choice between an external modulator signal (such as one of the ESP32 DAC outputs) and an internal modulator (normalled on the jack switch). The internally generated modulator signal can be switched between pulse and triangle type waveforms with controls on its pulse width and frequency. A control for modulation amount determines how much, if any, modulation is applied to the final mix.

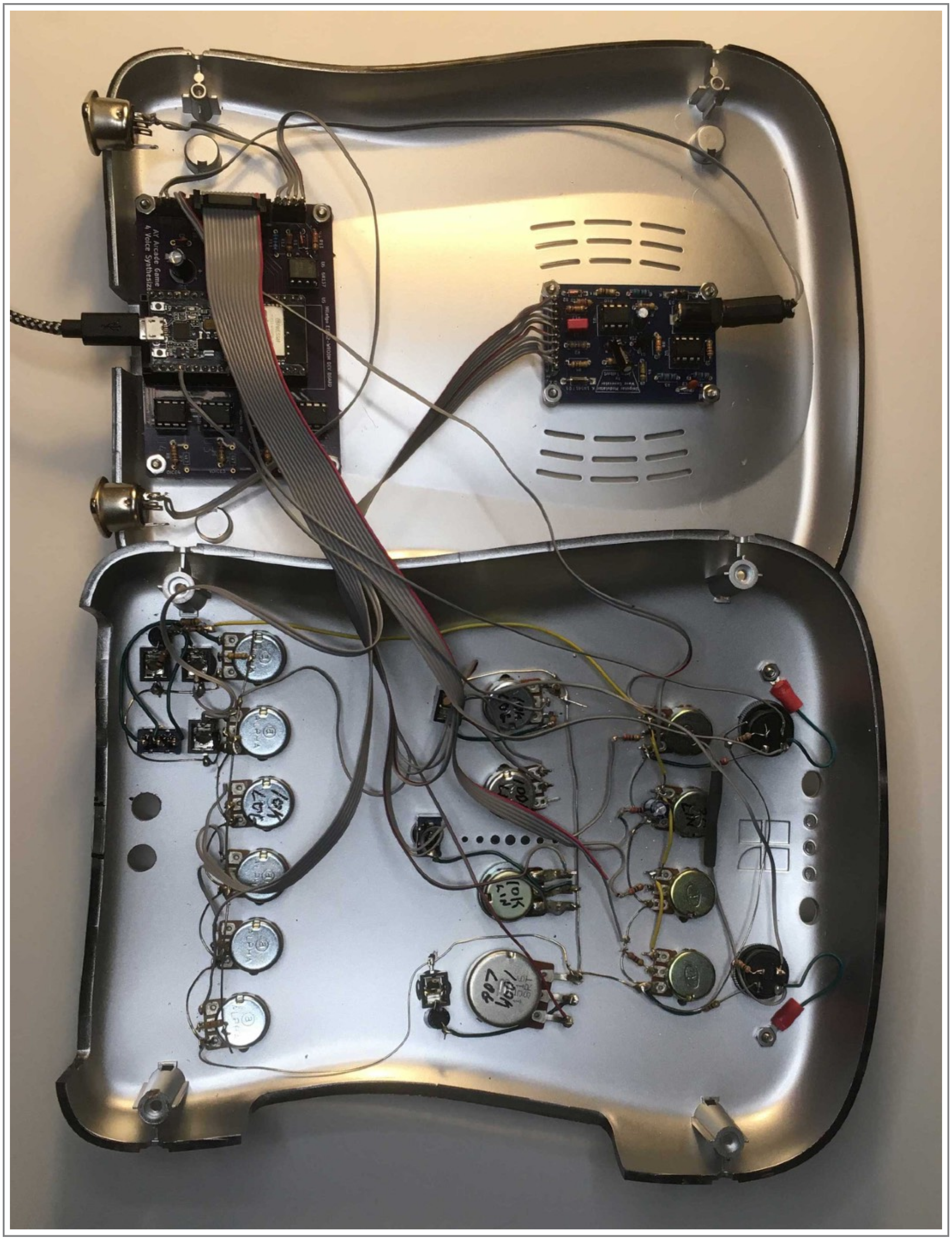
At the top of the box are the ESP32 sensor controls — four potentiometer control voltages and two pushbutton switches. It was easy to add Touch controllers to the two switches by just connecting the switch contacts to screws mounted next to each switch. These controls can be programmed to control the playback of the 6 voices in various ways.

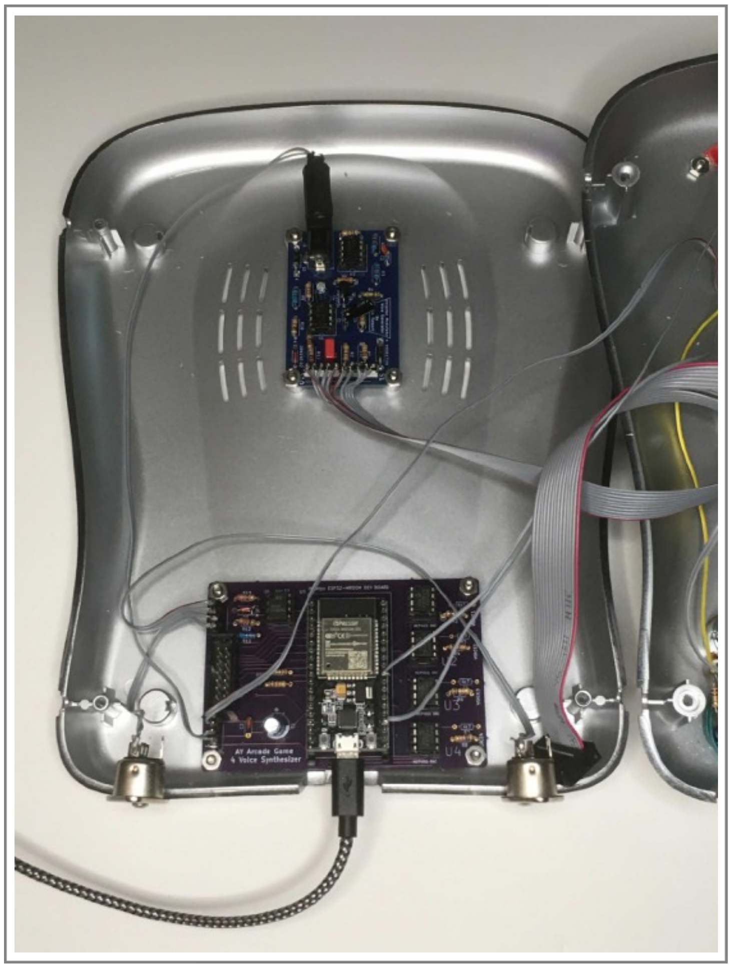
Each switch also incorporates an LED which were connected to ESP32 pins IO5 and IO15 along with a 1000 ohm current limiting resistor to ground.

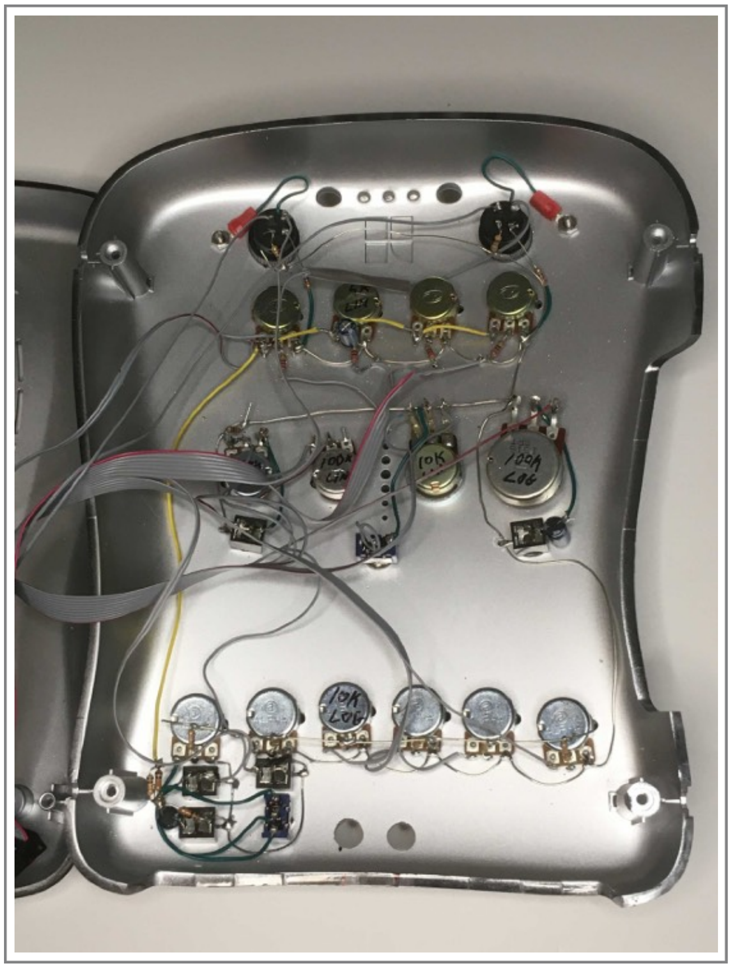
An opening at the front of the box allows access to the ESP32 board's USB cable connection. The USB cable provides power for the entire box and is used for programming the ESP32. Holes were drilled to allow access to a Reset Switch and a Boot Switch on either side of the USB jack. The eraser end of a pencil can be used to press either switch on the DEVKIT board which is sometimes needed while programming the micro.

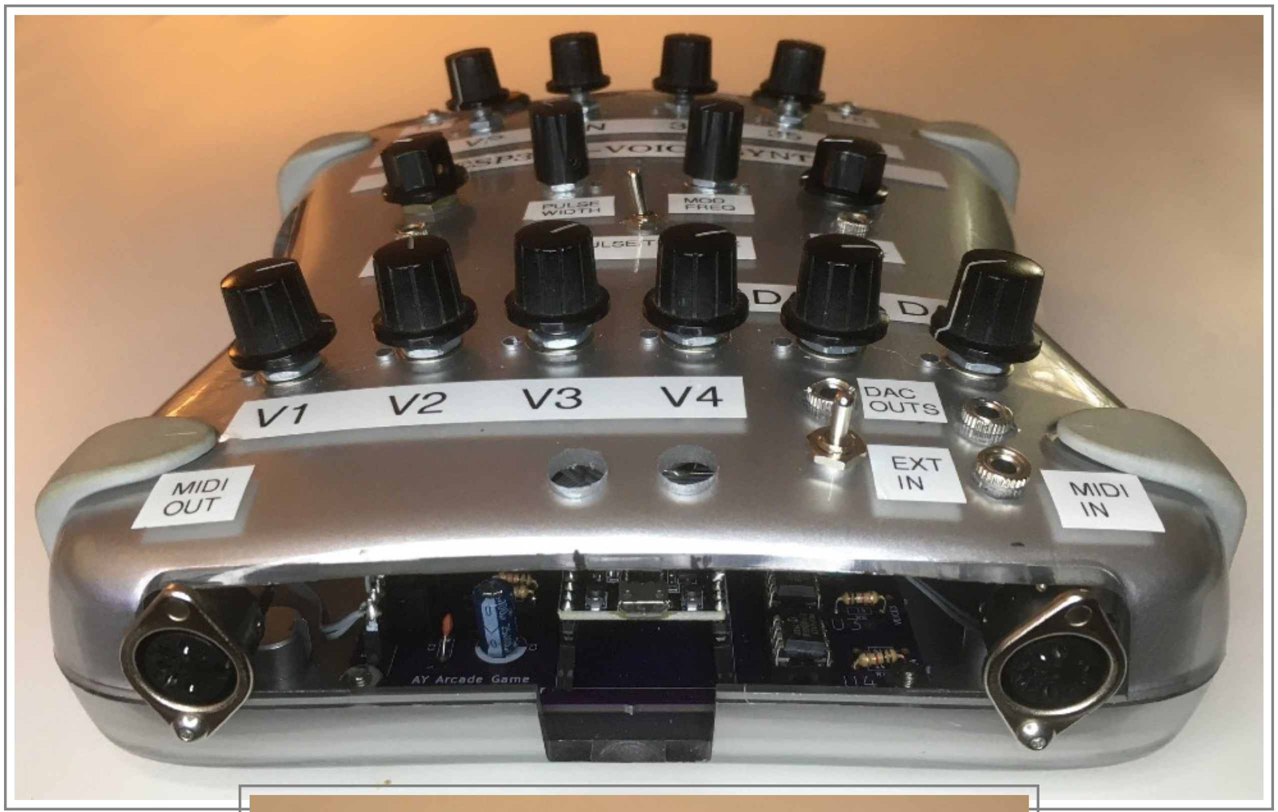
Also mounted at either end of the front opening are the MIDI INPUT and MIDI OUTPUT 5-pin DIN jacks.

Quite a few useful features were built into this box, but a much simpler version is also possible. It would only require the main PCB ESP32 DEVKIT with access to its USB jack, a signal output jack connected to ground and the MIX pin, and a few sensor controllers like a pot and switch.

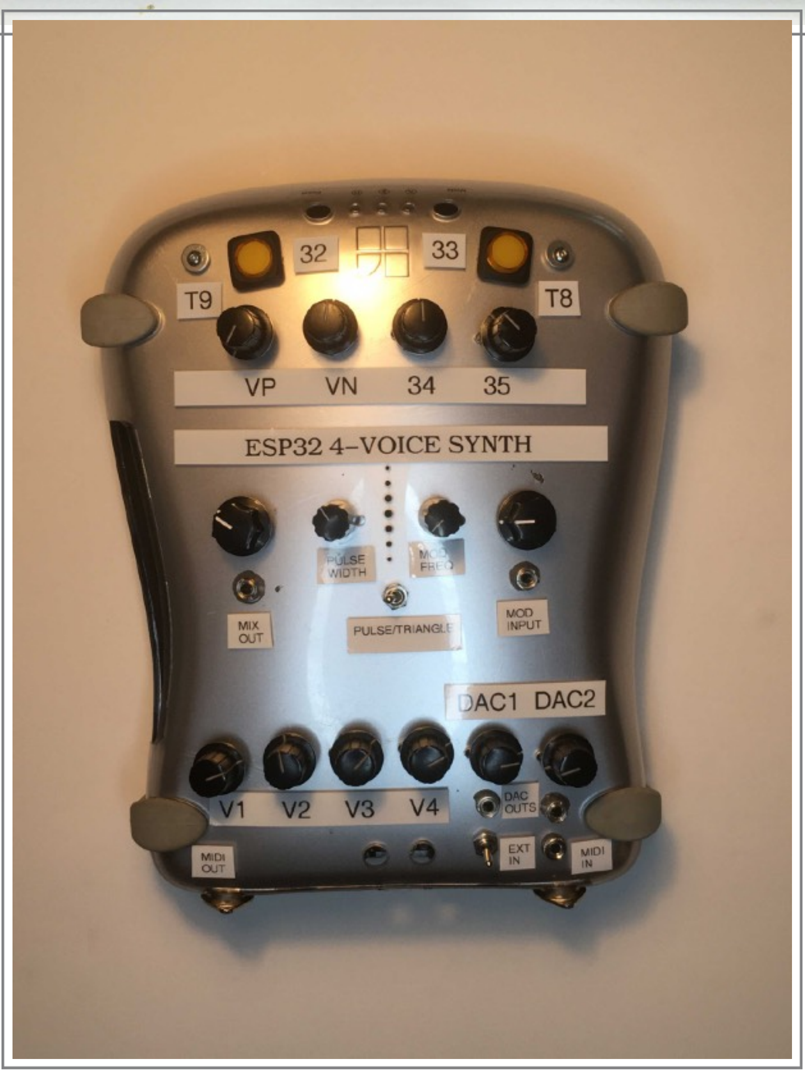



A7 Arcade Game
4 Voice Synthesizer


10K
LOG
100K
LOG
10K



V1
V2
V3
V4
DAC OUTS
EXT IN
MIDI OUT
MIDI IN
PULSE WIDTH
MOD FREQ
AY Arcade Game



32
33
T9
T8
VP
VN
34
35
ESP32 4-VOICE SYNTH
PULSE WIDTH
MOD FREQ
MIX OUT
PULSE/TRIANGLE
MOD INPUT
DAC1
DAC2
V1
V2
V3
V4
DAC OUTS
EXT IN
MIDI OUT
MIDI IN

ESP32 Programming

The ESP32

The ESP32 microprocessor is like a supercharged Arduino. It has a 32-bit processor, and can be programmed with the Arduino IDE app. The ESP32 is compatible with over 90% of the Arduino core programming language and many of its libraries. However, it has expanded capabilities such as wireless WiFi and Bluetooth. Its clock speed is 80MHz/240MHz compared with 48MHz for the Arduino MKR. The flash memory for user programs is 4MB/8MB compared to 256KB for the Arduino MKR. The SRAM memory for user variables is 520KB compared to 32KB for the MKR. The ESP32 processor is dual-core, enabling it to run two program threads simultaneously. Its peripherals can include up to 43 GPIOs, 1 full-speed USB OTG interface, SPI, I2S, UART, I2C, LED PWM, LCD interface, camera interface, ADC, DAC, touch sensors.

The Arduino programming package for the ESP32 has been expanded to include many useful features that were only accessible in the Arduino by hacking into its internal registers. This includes PWM setup with pulse width and clock speeds; timers with clock setup, mode and interrupt; touch (capacitive) sensor setups; ADC setups with resolution, width and speed.

Espressif is the company that develops the ESP32 (www.espressif.com). They provide a programming package "ESP32 for Arduino" on Github (https://github.com/espressif/arduino-esp32). Documentation and instructions for installing this package on your Arduino IDE app can be found at this Github site (https://docs.espressif.com/projects/arduino-esp32/en/latest/installing.html).

ESP32 Help

Given its expanded capabilities, programming the ESP32 can be difficult even given the Arduino's familiar IDE environment. Here are several sources for tutorials and books on the ESP32.

Espressif	https://www.espressif.com/en/products/modules	ESP32 Developer
Tech Explorations	https://techexplorations.com/pc/esp32/	Video Tutorials
Random Nerd	https://randomnerdtutorials.com/projects-esp32/	Tutorials, Books
Books	https://bookauthority.org/books/best-esp32-books	

The following Arduino IDE program sketches will illustrate how to program many of the special features available on the ESP32 using the AY Synthesizer Box shown above. They will culminate in a full blown emulation of the AY Arcade Game Sound Chip from the 70s.

Basic

This first sketch can act as a starting template for subsequent sketches as it sets up all the constants and variables needed for the pots, switches, LEDs and the multiplying DACs.

Note that the ESP32 pin constants SDA, SCL, DAC1, and DAC2 have already been defined in the Espressif ESP32 Arduino package (if you try to set them up yourself, you will get the error "redefinition of const"). Our first example of the many new and convenient features available from the Espressif ESP32 package.

The MCP_DAC Library by Rob Tillaart is used to implement the I2C protocol used for the MCP4921 Multiplyer DACs. This can be installed by downloading it from the Github site https://github.com/RobTillaart/MCP_DAC or using the Arduino IDE's Tools/ManageLibrary Menu.

To test the four multiplying DAC chips, tones are set up on the four main voices with a pot assigned to control each voice's volume. The Arduino Monitor screen will display a running readout of the 4 pot and 2 switch values.

```
/*
  ESP32 AY_Synth Basic Setup
*/

//already declared in ESP32 library
//static const uint8_t SCL = 22;        //Serial Lines to MCP4921 Multiplyer DACs
//static const uint8_t SDA = 21;

#include "MCP_DAC.h"                    //MCP_DAC Library by Rob Tillaart
MCP4921 myAYDAC1(SDA, SCL);
MCP4921 myAYDAC2(SDA, SCL);
MCP4921 myAYDAC3(SDA, SCL);
MCP4921 myAYDAC4(SDA, SCL);


// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~~
//                  GIOP Pin Assignments

static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;

short pot1;
short pot2;
short pot3;
short pot4;

static const uint8_t SWITCH1 = 33;    //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

bool switch1;
bool switch2;

//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;

static const uint8_t V1 = 25;           //Voice Pins to MCP4921 Multiplyer DACs
static const uint8_t V2 = 26;
static const uint8_t V3 = 12;
static const uint8_t V4 = 13;

static const uint8_t X1 = 27;
static const uint8_t X2 = 14;

static const uint8_t CS_V1 = 23;       //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O, also LED on MIDI Out
static const uint8_t MIDI_RX2 = 16;   //only on WROOM, won't work on WROVER ESP32s
```

```
static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;           //Blue LED on 32S boards



// ~~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  myAYDAC1.begin(CS_V1);
  myAYDAC2.begin(CS_V2);
  myAYDAC3.begin(CS_V3);
  myAYDAC4.begin(CS_V4);


  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

 //initialize 4 digital voice inputs to Multiplyer DACs
  pinMode(V1, OUTPUT);
  digitalWrite(V1, HIGH);
  pinMode(V2, OUTPUT);
  digitalWrite(V2, HIGH);
  pinMode(V3, OUTPUT);
  digitalWrite(V3, HIGH);
  pinMode(V4, OUTPUT);
  digitalWrite(V4, HIGH);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);

  Serial.begin(115200);

}
```

```
// ~~~~~~~~~~~~~~~~~~~~  LOOP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

   loadSensors();

//setup pots as voice volume controls

  myAYDAC1.analogWrite(pot1);
  myAYDAC2.analogWrite(pot2);
  myAYDAC3.analogWrite(pot3);
  myAYDAC4.analogWrite(pot4);

  //create a tone in all 4 voices

  digitalWrite(V1, LOW);
  digitalWrite(V2, LOW);
  digitalWrite(V3, LOW);
  digitalWrite(V4, LOW);

  delayMicroseconds(100);

  digitalWrite(V1, HIGH);
  digitalWrite(V2, HIGH);
  digitalWrite(V3, HIGH);
  digitalWrite(V4, HIGH);

  delayMicroseconds(10);

 //print pot and switch values

   Serial.print("pot1 = ");
  Serial.print(pot1);
    Serial.print("  pot2 = ");
  Serial.print(pot2);
    Serial.print("   pot3 = ");
  Serial.print(pot3);
    Serial.print("   pot4 = ");
  Serial.print(pot4);

 Serial.print("     ");

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

}
```

```
// ~~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
     pot1 =   analogRead(POT1) ;
     pot2 =   analogRead(POT2) ;
     pot3 =   analogRead(POT3) ;
     pot4 =   analogRead(POT4) ;

     switch1 = digitalRead(SWITCH1);
     switch2 = digitalRead(SWITCH2);
}
```

LED PWM

This program uses special ESP32 PWM functions to control the brightness of an LED using either a switch or a pot.

```
ledcSetup(ledChannelA, freq, resolution1);  // in Setup
ledcAttachPin(LED1, ledChannelA);           // in Setup
ledcWrite(ledChannelA, dutyCycleA);         // in Main Loop
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
/*
  ESP32 AY_Synth
  Using the LEDs connected to GPIO 2, 5, and 15

  PWM control of LED brightness from a switch or pot
*/

// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//                       GIOP Pin Assignments

static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;

short pot1;
short pot2;
short pot3;
short pot4;

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

bool switch1;
bool switch2;
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;

static const uint8_t V1 = 25;  //Voice Pins to MCP4921 Multiplyer DACs
static const uint8_t V2 = 26;
static const uint8_t V3 = 12;
static const uint8_t V4 = 13;

static const uint8_t X1 = 27;
static const uint8_t X2 = 14;

static const uint8_t CS_V1 = 23;   //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O, also LED on MIDI Out
static const uint8_t MIDI_RX2 = 16;   //only on WROOM, won't work on WROVER
ESP32s

static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards

// setting PWM properties
const int freq = 5000;
const int ledChannelA = 0;
const int ledChannelB = 1;
const int resolution1 = 8;
const int resolution2 = 12;

int dutyCycleA = 0;
int dutyCycleB = 0;
```

```
// ~~~~~~~~~~~~~~~~~~~~ SETUP ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

 //initialize 4 digital voice inputs to Multiplyer DACs
  pinMode(V1, OUTPUT);
  digitalWrite(V1, HIGH);
  pinMode(V2, OUTPUT);
  digitalWrite(V2, HIGH);
  pinMode(V3, OUTPUT);
  digitalWrite(V3, HIGH);
  pinMode(V4, OUTPUT);
  digitalWrite(V4, HIGH);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

 // configure LED PWM functionalitites
  ledcSetup(ledChannelA, freq, resolution1); // 0, 5000, 8  assigned above
  ledcSetup(ledChannelB, freq, resolution2); // 1, 5000, 12  assigned above

  // attach the channel to the GPIO to be controlled
  ledcAttachPin(LED1, ledChannelA);
  ledcAttachPin(LED2, ledChannelB);
```

```
  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);
}

// ~~~~~~~~~~~~~~~~~~~  LOOP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

   loadSensors();

if (!switch1) {
  // increase the LED brightness or simply turn on with digitWrite(LED1, HIGH)
  if(dutyCycleA < 255){
    // changing the LED brightness with PWM
    dutyCycleA++;
    ledcWrite(ledChannelA, dutyCycleA);
  }
}
else {
  // decrease the LED brightness or simply turn off with digitalWrite(LED1, LOW)
  if(dutyCycleA > 0){
    // changing the LED brightness with PWM
    dutyCycleA--;
    ledcWrite(ledChannelA, dutyCycleA);
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ledcWrite(ledChannelB, pot1);
    delay(20);
}
```

```
// ~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
    pot1 =   analogRead(POT1) ;
    pot2 =   analogRead(POT2) ;
    pot3 =   analogRead(POT3) ;
    pot4 =   analogRead(POT4) ;

    switch1 = digitalRead(SWITCH1);
    switch2 = digitalRead(SWITCH2);
}
```

MIDI Input and Output

MIDI Input and Output can be set up on the ESP32 RX2 and TX2 pins with the following program lines:

```
                    #include <MIDI.h>
MIDI_CREATE_INSTANCE(HardwareSerial, Serial2, MIDI);
```

Use the Arduino MIDI Library. Specify HardwareSerial and Serial2 which the ESP32 package will understand to be the RX2 and TX2 pins.

~~~~~~~~~~~~~~~~~~~~~~~~~~

```
/*
		MIDI INPUT and OUTPUT tested with a MIDI input CallBack Function
		On each NOTE On message received,
		2 extra arpeggiated notes will be played.

		Slider 1 sets the playback speed.
		Slider 2 sets the note spread
		Switch 1 plays random notes at speed set by Slider 4.
*/
//
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~
//			CONSTANTS and Variables
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//
//On an ESP32 WROOM.  MIDI set up on Serial2 --> RX2 and TX2 on pins 16 and 17.

		#include <MIDI.h>  // version 4.x.x
	MIDI_CREATE_INSTANCE(HardwareSerial, Serial2, MIDI);

// ~~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~~
```
```

```
// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//                                  GIOP Pin Assignments

static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;

short pot1;
short pot2;
short pot3;
short pot4;

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

bool switch1;
bool switch2;

static const uint8_t CS_V1 = 23;   //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O
static const uint8_t MIDI_RX2 = 16;   //only on WROOM, won't work on WROVER
ESP32s

static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                    Callback MIDI_In Test function
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// user created Callback Function for MIDI Input Test

 void myHandleNoteOn(byte channel, byte note, byte velocity){

  //Serial.println(" Saw a NoteOn ");

  int x = (pot1 >> 4) + 20;     //pot1 sets arpeggio speed

  int y = pot2;                //pot2 sets arpeggio pitch range
  y = map(y, 0, 4095, 0, 20);

  delay(x);
  MIDI.sendNoteOn(note + y, velocity, 1);
  delay(x);
  MIDI.sendNoteOn(note + y + y, velocity, 1);
  delay(x);

  MIDI.sendNoteOn(note, 0, 1);
  MIDI.sendNoteOn(note + y, 0, 1);
  MIDI.sendNoteOn(note + y + y, 0, 1);
 }

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


void setup() {

delay(1000);

  MIDI.setHandleNoteOn(myHandleNoteOn); //for Callback MIDI In Test
  MIDI.begin(MIDI_CHANNEL_OMNI);

  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);
```

```
 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);

  Serial.begin(115200);


} //End of Setup

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

void loop() {

loadSensors();

// On MIDI.read() MIDI class will call Callback functions.
// User created callback function myHandleNoteOn() in section before setup()
// and MIDI.setHandleNoteOn(myHandleNoteOn) in setup() section

MIDI.read();
```

```
if(!switch1){    // test MIDI Output with Switch 1 and Pot 4
  int note = random(30, 90);
  MIDI.sendNoteOn(note, 64, 1);
  delay((pot4 >> 4) + 10);
  MIDI.sendNoteOn(note, 0, 1);
}

} //End of Main Loop


// ~~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
     pot1 =   analogRead(POT1) ;
     pot2 =   analogRead(POT2) ;
     pot3 =   analogRead(POT3) ;
     pot4 =   analogRead(POT4) ;

     switch1 = digitalRead(SWITCH1);
     switch2 = digitalRead(SWITCH2);
}
```

DAC Waveforms

The ESP32 has two 8-bit DACs (Digital to Analog Converters) on pins DAC1 and DAC2 (25 and 26). This Program mathematically creates 3 different waveform arrays using them to create a DAC output voice. Selection between the Waveforms happens in the Monitor window by typing "s", "q", or "t" for sine wave, square wave, or triangle wave. Pot 1 controls the frequency with a delay function.

The special ESP32 function "**dacWrite(DAC1, value)**" is used to load the 8-bit DAC.

~~~~~~~~~~~~~~~~~~~~~~

```
/*
 ESP32 DACs

 * 1. ESP32 Datasheet: https://www.espressif.com/sites/default/files/
        documentation/esp32_datasheet_en.pdf
 * 2. ESP32 has two 8-bit DACs (digital to analog converter) channels, connected to
        GPIO25 (Channel 1) and GPIO26 (Channel 2)
 * 3. ESP32 Arduino Core header file: https://github.com/espressif/arduino-esp32/
        blob/master/cores/esp32/esp32-hal-dac.h
 *
 * Created on March 26 2019 by Peter Dalmaris
*/

// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//              GIOP Pin Assignments

static const uint8_t POT1 = 36;
short pot1;

//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;
```
~~~~~~~~~~~~~~~~~~~~~~

```
static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards

uint8_t Triangle[360];
uint8_t Sine[360];
uint8_t Square[360];

 char wave = 0;
 int w = 0;

// ~~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  digitalWrite(LED2, HIGH);

  Serial.begin(115200);

  for(int deg = 0; deg < 360; deg++){

    Sine[deg] = int(128 + 80 * (sin(deg*PI/180)));
    Square[deg] = int(128 + 80 * (sin(deg*PI/180)+sin(3*deg*PI/180)/3+sin(5*deg*PI/
          180)/5+sin(7*deg*PI/180)/7+sin(9*deg*PI/180)/9+sin(11*deg*PI/180)/11));
    Triangle[deg] = int(128 + 80 * (sin(deg*PI/180)+1/pow(3,2)*sin(3*deg*PI/180)+1/
          pow(5,2)*sin(5*deg*PI/180)+1/pow(7,2)*sin(7*deg*PI/180)+1/
          pow(9,2)*sin(9*deg*PI/180)));
  }

  Serial.println("");
  Serial.println(" Type 's' for Sinewave then Return");
  Serial.println(" Type 'q' for Squarewave then Return");
  Serial.println(" Type 't' for Triangle Waveform then Return");
  Serial.println(" Any other character turns off the tone ");

}
```

```
// ~~~~~~~~~~~~~~~~~~~~  LOOP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

  pot1 =   analogRead(POT1) ;

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

 if (Serial.available() > 0) {    // is a character available?
   wave = Serial.read();       // get the character

   if(wave == 's'){
     Serial.println("Sinewave output");
     w = 1;
   }
   else if(wave == 'q'){
     Serial.println("Squarewave output");
     w = 2;
   }
   else if(wave == 't'){
     Serial.println("Triangle output");
     w = 3;
   }
   else { w = 0; }
 }

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

 switch (w) {
  case 1:
   for (int deg = 0; deg < 360; deg++){
       dacWrite(DAC1, Sine[deg]);
       delayMicroseconds(pot1 >> 6 );
     }
   break;
  case 2:
   for (int deg = 0; deg < 360; deg++){
       dacWrite(DAC1, Square[deg]);
       delayMicroseconds(pot1 >> 6 );
```

```
 case 2:
   for (int deg = 0; deg < 360; deg++){
       dacWrite(DAC1, Square[deg]);
       delayMicroseconds(pot1 >> 6 );
     }
   break;
   case 3:
   for (int deg = 0; deg < 360; deg++){
       dacWrite(DAC1, Triangle[deg]);
       delayMicroseconds(pot1 >> 6 );
     }
   break;

 }

}
```

Touch Switch

Many of the ESP32 pins can act as Touch Switches. Pins IO32 and IO33 have been assigned to T9 and T8 in the ESP32 Arduino Package. The touch pins do not act as high or low digital outputs. They put out an analog type value that changes with the proximity of your finger to the pin, or any conductive material connected to the pin. Screws were mounted, in our enclosure, next to the two switches and connected by wire to each of the T9 and T8 pins. When touched the touch value hovered around 10 and when not touched the value was around 40.

The ESP32 function used to read the touch pin value is:
touchRead(T8)

In this sketch the Monitor window prints a running value of each touch pin so you can see for yourself how the proximity of your finger affects the touchRead() value.

Two methods are demonstrated for changing the touch value into a digital on or off. One method is to turn on or off an LED by continuously comparing the touch value to a threshold value at 20. The second method doesn't restrict you to constantly watching the touchRead(). It sets up a background interrupt routine (**gotTouch**) that will flash the other LED when a threshold is crossed. It uses this convenient ESP32 touch interrupt function: "gotTouch" will run when T8 goes below "threshold".

touchAttachInterrupt(T8, gotTouch, threshold);

~~~~~~~~~~~~~~~~~~~

```
/*
 ESP32 AY_Synth Touch Sensors
 The 2 Switch pins are connected to chassis screws next to each switch
 to accommodate the Touch Sensing function.

 One Touch Sensor value is compared to a threshold value
 to directly affect LED1

 The second Touch Sensor uses an interrupt to flash LED2 for
 a half second.
*/
```
~~~~~~~~~~~~~~~~~~~

```
// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//                                  GIOP Pin Assignments

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor T8
static const uint8_t SWITCH2 = 32;  //Also Touch Sensor T9

bool switch1;
bool switch2;

static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards

int threshold = 20;  // This threshhold is determined experimentally. If the touch
                     // sensor returns a value below this number, the interrupt is triggered.
bool touch8detected = false;
                    // used to communicate between the loop and the interrupt routine

void gotTouch(){     //Ideally, Interrupt Service Routines are very short, like this
 touch8detected = true;
}


// ~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  digitalWrite(LED3, HIGH); //Blue Board LED

  // ESP32 Library Function that attaches interrupt pin T8 to the service routine
  touchAttachInterrupt(T8, gotTouch, threshold);
```

```
  Serial.begin(115200);

}

// ~~~~~~~~~~~~~~~~~~~  LOOP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

Serial.print(touchRead(T8));  // get value using T8 (GPIO33)
Serial.print("    ");
Serial.println(touchRead(T9));  // get value using T9 (GPIO32)



  if (touchRead(T9) < 20)    // The value 20 is determined experimentally
    digitalWrite(LED2, HIGH);
  else
    digitalWrite(LED2, LOW);



  if (touch8detected)  //Set high by interrupt service routine
  {
    digitalWrite(LED1, HIGH);   // blink LED1
    Serial.println("Touch detected");
    touch8detected = false;
    delay(500);
    digitalWrite(LED1, LOW);
  }

}
// ~~~~~~~~~~~~~~~~~~~
// As an Alternative:  Set up Interrupt Function on Switch2, assign interrupt
// function, and set FALLING mode (High to Low Switch transition).
// attachInterrupt(digitalPinToInterrupt(SWITCH2), switch2Function, FALLING);
// ~~~~~~~~~~~~~~~~~~~
```

PWM Noise Voice

This sketch employs the same functions used to dim an LED except that here, a PCM Pulse Waveform is created.

ledcSetup(channel, frequency, resolution)
ledcAttachPin(VoicePin, channel)
ledcWrite(channel, pulse_width)

If the Pulse Waveform is continually sent a random pulse_width, the result will be pitched noise centered around "frequency".

This Noise Voice will be used in the AY chip simulation. The pots here are set up to explore all the possible parameter of this Pulse Width Modulated voice.

pot1 -> sets the voice pitch when Switch2 is pressed or touched
pot2 -> sets either pcm pulse width or the range of random pulse widths,
as selected by Switch1
pot3 -> sets the delay between pcm pulse width loads
pot4 -> sets the volume of the PCM Voice4

~~~~~~~~~~~~~~~~~~~~~~~~~~

```
/*
  ESP32 AY_Synth Setup

  A pitched noise PCM Voice is created by loading its pulse width with random values.
  PCM library functions are available for ESP32.  They are labeled as LED functions
used to
  control LED brightness, but here they will be used to create an audio voice.

  pot1 -> sets the voice pitch when Switch2 is pressed or touched
  pot2 -> sets pcm pulse width or the range of random pulse widths, selected by
Switch1
  pot3 -> sets the delay between pcm pulse width loads
  pot4 -> sets the volume of the PCM Voice4

*/
```
~~~~~~~~~~~~~~~~~~~~~~~~~~

```
//already declared in ESP32 library
//static const uint8_t SCL = 22;   //Serial Lines to MCP4921 Multiplyer DACs
//static const uint8_t SDA = 21;

#include "MCP_DAC.h"  //MCP_DAC Library by Rob Tillaart
MCP4921 myAYDAC1(SDA, SCL);
MCP4921 myAYDAC2(SDA, SCL);
MCP4921 myAYDAC3(SDA, SCL);
MCP4921 myAYDAC4(SDA, SCL);


// ~~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~~
//                                        GIOP Pin Assignments

static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;

short pot1;
short pot2;
short pot3;
short pot4;

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

bool switch1;
bool switch2;

//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;

static const uint8_t V1 = 25;  //Voice Pins to MCP4921 Multiplyer DACs
static const uint8_t V2 = 26;
static const uint8_t V3 = 12;
static const uint8_t V4 = 13;

static const uint8_t X1 = 27;
static const uint8_t X2 = 14;
```

```
static const uint8_t CS_V1 = 23;   //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O, also LED on MIDI Out
static const uint8_t MIDI_RX2 = 16; //only on WROOM not WROVER ESP32s

static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards

 int pcm_freq = 440;
 const int pcm_Channel = 0;  //0-15
 int pcm_resolution = 12;  // bits, 8 to 16.
 int pcm_width = 1000;

 int threshold = 20;
 bool touch8detected = false;

// Touch9 Interrupt Function: sets up PCM frequency from pot1
 void gotTouch(){
   ledcSetup(pcm_Channel, (pot1 + 30), pcm_resolution);
 }

/*
// Switch Interrupt Function: sets up PCM frequency from pot1
 void IRAM_ATTR switch2Function() {
   ledcSetup(pcm_Channel, (pot1 + 30), pcm_resolution);
}
*/
```

```
// ~~~~~~~~~~~~~~~~~~~~~ SETUP ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  myAYDAC1.begin(CS_V1);
  myAYDAC2.begin(CS_V2);
  myAYDAC3.begin(CS_V3);
  myAYDAC4.begin(CS_V4);

  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

 //initialize 4 digital voice inputs to Multiplyer DACs
  pinMode(V1, OUTPUT);
  digitalWrite(V1, HIGH);
  pinMode(V2, OUTPUT);
  digitalWrite(V2, HIGH);
  pinMode(V3, OUTPUT);
  digitalWrite(V3, HIGH);
  pinMode(V4, OUTPUT);
  digitalWrite(V4, HIGH);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
```

```
  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);

  //Serial.begin(115200);

// configure PWM functionalities
  ledcSetup(pcm_Channel, pcm_freq, pcm_resolution);

// attach the PCM channel to V4
  ledcAttachPin(V4, pcm_Channel);

// ~~~~~~~~~~~~~~~~~~~~
// As an Alternative:  Set up Interrupt Function on Switch2, assign interrupt
// function, and set FALLING mode (High to Low Switch transition).
// attachInterrupt(digitalPinToInterrupt(SWITCH2), switch2Function, FALLING);
// ~~~~~~~~~~~~~~~~~~~~

// Setup Interrupt Function on Switch2 Touch control (pin T9).
// ESP32 Library Function that attaches interrupt pin T9 to the service routine
  touchAttachInterrupt(T9, gotTouch, threshold);
}

// ~~~~~~~~~~~~~~~~~~~~  LOOP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {
  // pot1 is pcm frequency loaded by way of interrupt when Switch 2 is touched

   loadSensors();
   myAYDAC4.analogWrite(pot4);  // pot4 is volume control of V4, pcm voice

   if(switch1){ pcm_width = pot2; }
 // pot2 is pcm pulse width, Switch1 chooses between direct and random
   else { pcm_width = random(0, pot2); }
```

```
  ledcWrite(pcm_Channel, pcm_width );
  delay(pot3 >> 5);                  // pot3 is delay between pcm pulse width writes

}


// ~~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
    pot1 =   analogRead(POT1) ;
    pot2 =   analogRead(POT2) ;
    pot3 =   analogRead(POT3) ;
    pot4 =   analogRead(POT4) ;

    switch1 = digitalRead(SWITCH1);
    switch2 = digitalRead(SWITCH2);
}
```

Timer Interrupt Voices

The next several sketches will build up to our goal of emulating the AY Arcade Game Sound Chip of the 70s. This chip is based on 3 square wave voices plus a pitched noise voice.

Each square wave voice is created by decrementing a register from some initial value with a high frequency clock (1MHz/16). When the register reaches zero the digital output pin of the voice is toggled to its opposite state, the register is reloaded with its initial value and starts decrementing again. The resulting frequency is then one half the clock frequency divided by the register value. So the register's initial value determines the frequency of the voice square wave.

Decrementing a value, resetting it, and toggling a pin can easily be done in software but a steady clock is needed to time those actions at a steady and set interval. The ESP32 provides timers that can initiate an interrupt routine at set intervals, perfect for this application. Here is how it is set up:

```
hw_timer_t * timer = NULL;  //pointer to a hardware timer on the ESP32
void IRAM_ATTR onTimer() {.....}  //Interrupt Routine

timer = timerBegin(0, 80, true); //(timer# , 80MHz/80 prescaler, count up)
timerAttachInterrupt(timer, &onTimer, true); //(timer, InterruptRoutine, edge)
timerAlarmWrite(timer, 16, true); //(timer, (80Mhz/80)/16, repeat yes)
timerAlarmEnable(timer);  //Start
```

This sketch sets up three square waves. The frequency determining registers are freq1save, freq2save, and freq3save. The user can use the pots to change the values in these registers at any time, thus changing the voice frequencies.

Here is the Interrupt Routine for one voice that happens every 16 microseconds:

```
--freq1;                                  // decrement counter
 if (freq1 <= 0){                         // check if counter has reached zero
 digitalWrite(V1, !digitalRead(V1));      // if so, toggle Voice1 pin, and
 freq1 = freq1save;  }                    // reload counter from freq. register
```

~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~

```
/*
  ESP32 AY_Synth Setup
*/

//already declared in ESP32 library
//static const uint8_t SCL = 22;   //Serial Lines to MCP4921 Multiplyer DACs
//static const uint8_t SDA = 21;

#include "MCP_DAC.h"  //MCP_DAC Library by Rob Tillaart
MCP4921 myAYDAC1(SDA, SCL);
MCP4921 myAYDAC2(SDA, SCL);
MCP4921 myAYDAC3(SDA, SCL);
MCP4921 myAYDAC4(SDA, SCL);


// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//                                   GIOP Pin Assignments

static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;

short pot1;
short pot2;
short pot3;
short pot4;

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

bool switch1;
bool switch2;

//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;

static const uint8_t V1 = 25;  //Voice Pins to MCP4921 Multiplyer DACs
static const uint8_t V2 = 26;
static const uint8_t V3 = 12;
static const uint8_t V4 = 13;

static const uint8_t X1 = 27;
static const uint8_t X2 = 14;

static const uint8_t CS_V1 = 23;   //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O, also LED on MIDI Out
static const uint8_t MIDI_RX2 = 16;   //only on WROOM, won't work on WROVER ESP32s

static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards
```

```
uint16_t AY_MidiNote_Hi[128] = {   // AY course tune -- upper 4-bits of 12-bit tune
  15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
  14, 14, 13, 12, 11, 11, 10, 9, 9, 8, 8, 7,
  7, 7, 6, 6, 5, 5, 5, 4, 4, 4, 4, 3,
  3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0 };

uint16_t AY_MidiNote_Lo[128] = {  // AY fine tune -- lower 8 bits of 12-bit tune
  211, 211, 211, 211, 211, 211, 211, 211, 211, 211, 211, 211,
  239, 25, 78, 141, 218, 47, 143, 247, 104, 225, 97, 233,
  119, 12, 167, 71, 237, 152, 71, 252, 180, 112, 49, 244,
  188, 134, 83, 36, 246, 204, 164, 126, 90, 56, 24, 250,
  222, 195, 170, 146, 123, 102, 82, 63, 45, 28, 12, 253,
  239, 225, 213, 201, 190, 179, 169, 159, 150, 142, 134, 127,
  119, 113, 106, 100, 95, 89, 84, 80, 75, 71, 67, 63,
  60, 56, 53, 50, 47, 45, 42, 40, 38, 36, 34, 32,
  30, 28, 27, 25, 24, 22, 21, 20, 19, 18, 17, 16,
  15, 14, 13, 13, 12, 11, 11, 10, 9, 9, 8, 8,
  7, 7, 7, 6, 6, 6, 5, 5 };

uint16_t AY_Volume[16] = {  //4-bit AY synth volume to 12-bit logarithmic volue
  0, 10, 25, 51, 62, 102, 124, 205, 307, 512, 621, 1024, 1241, 2048, 2896, 4095 };

  volatile uint16_t freq1 = 0;
  volatile uint16_t freq2 = 0;
  volatile uint16_t freq3 = 0;
  volatile uint16_t freq4 = 0;

  volatile uint16_t freq1save = 0;
  volatile uint16_t freq2save = 0;
  volatile uint16_t freq3save = 0;
  volatile uint16_t freq4save = 0;

  int count = 0;
```

```
// ~~~~~~~~~~~~~~~~~  Timer Interrupt  ~~~~~~~~~~~~~~~

// The hardware timer pointer
hw_timer_t * timer = NULL;


// Interrupt Routine
void IRAM_ATTR onTimer() {

  --freq1;                         //toggle V1 at end of freq1 countdown
    if (freq1 <= 0){
        digitalWrite(V1, !digitalRead(V1));
        freq1 = freq1save;
    }
     --freq2;                        //toggle V2 at end of freq2 countdown
    if (freq2 <= 0){
        digitalWrite(V2, !digitalRead(V2));
        freq2 = freq2save;
    }
     --freq3;                        //toggle V3 at end of freq3 countdown
      if (freq3 <= 0){
        digitalWrite(V3, !digitalRead(V3));
        freq3 = freq3save;
    }
}


// ~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  myAYDAC1.begin(CS_V1);
  myAYDAC2.begin(CS_V2);
  myAYDAC3.begin(CS_V3);
  myAYDAC4.begin(CS_V4);


  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

 //initialize 4 digital voice inputs to Multiplyer DACs
  pinMode(V1, OUTPUT);
  digitalWrite(V1, HIGH);
  pinMode(V2, OUTPUT);
  digitalWrite(V2, HIGH);
  pinMode(V3, OUTPUT);
  digitalWrite(V3, HIGH);
  pinMode(V4, OUTPUT);
  digitalWrite(V4, HIGH);
```

```
//set DACs to maximum for Voice inputs
  myAYDAC1.analogWrite(4095);
  myAYDAC2.analogWrite(4095);
  myAYDAC3.analogWrite(4095);
  myAYDAC4.analogWrite(4095);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);

  // Serial.begin(115200);

  // Initilise the timer.
  // Parameter 1 is the timer we want to use. Valid: 0, 1, 2, 3 (total 4 timers)
  // Parameter 2 is the prescaler. The ESP32 default clock is at 80MhZ.
  // (look under Arduino Menu Tools/Flash Frequency) The value "80" will
  // divide the clock by 80, giving us 1,000,000 ticks per second.
  // Parameter 3 is true means this counter will count up, instead of down (false).
  timer = timerBegin(0, 80, true);

  // Attach the timer to the interrupt service routine named "onTimer".
  // The 3rd parameter is set to "true" to indicate that we want to use the "edge" type (instead of "flat").
  timerAttachInterrupt(timer, &onTimer, true);

  // This is where we indicate the frequency of the interrupts.
  // The value "16" (because of the prescaler we set in timerBegin) will produce
  // one interrupt every 16 microseconds.
  // The 3rd parameter is true so that the counter reloads when it fires an interrupt, and so we
  // can get periodic interrupts (instead of a single interrupt).
  timerAlarmWrite(timer, 16, true);

  // Start the timer
  timerAlarmEnable(timer);

}

// ~~~~~~~~~~~~~~~~~~~  LOOP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

 count++;

if(count >= 10000){
   freq1save = midi_to_Freq((analogRead(POT1)) >> 5);
   freq2save = midi_to_Freq((analogRead(POT2)) >> 5);
   freq3save = midi_to_Freq((analogRead(POT3)) >> 5);
   count = 0;
}
```

```
}
// ~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
      pot1 =   analogRead(POT1) ;
      pot2 =   analogRead(POT2) ;
      pot3 =   analogRead(POT3) ;
      pot4 =   analogRead(POT4) ;

      switch1 = digitalRead(SWITCH1);
      switch2 = digitalRead(SWITCH2);
}

void serviceVoices(){
    --freq1;                        //toggle V1 at end of freq1 countdown
    if (freq1 <= 0){
        digitalWrite(V1, !digitalRead(V1));
        freq1 = freq1save;
    }
     --freq2;                       //toggle V2 at end of freq2 countdown
    if (freq2 <= 0){
        digitalWrite(V2, !digitalRead(V2));
        freq2 = freq2save;
    }
     --freq3;                       //toggle V3 at end of freq3 countdown
      if (freq3 <= 0){
        digitalWrite(V3, !digitalRead(V3));
        freq3 = freq3save;
    }
}

uint16_t midi_to_Freq(uint8_t note){
  uint16_t x;
  x = AY_MidiNote_Lo[note] | (AY_MidiNote_Hi[note] << 8);
  return x;
}
```

Timer Voice Core

The ESP32 has two cores that can run instructions simultaneously. This sketch sets up both Cores with Tasks that will run simultaneously.

Core 0 Task 1 is to clock the Voice Frequencies using a timer interrupt as shown in the previous sketch. Keeping the pot and switch variables updated is also thrown in as a job for the Core 0 Task.

Core 1 Task is the Main Loop. It will handle Voice Performance which involves handling voice frequencies and volumes over time.

Core 0 Task 1 is declared inside SETUP, and the actual task is built like a function placed between SETUP and the main LOOP.

```
                    TaskHandle_t Task1;

    //create a task executed in Task1code() function, priority , executed on core 0
         xTaskCreatePinnedToCore(
              Task1code, /* Task function. */
               "Task1", /* name of task (shown below). */
              10000,  /* Stack size of task */
               NULL,   /* parameter of the task */
               1,     /* priority of the task */
                &Task1, /* handle to keep track of task */
               0      /* pin task to core0*/
         );


              void Task1code( void * pvParameters ){
              Serial.print("Task1 running on core ");
              Serial.println(xPortGetCoreID());
                     ……………..
               }

              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
/*
  ESP32 AY_Synth Setup

  Core 0 Task is to clock the Voice Frequencies using a timer interrupt
  and to keep the pot and switch variables updated

  Core 1 Task is the Main Loop that handles Voice Performance
*/

//already declared in ESP32 library
//static const uint8_t SCL = 22;   //Serial Lines to MCP4921 Multiplyer DACs
//static const uint8_t SDA = 21;

TaskHandle_t Task1;

#include "MCP_DAC.h"  //MCP_DAC Library by Rob Tillaart
MCP4921 myAYDAC1(SDA, SCL);
MCP4921 myAYDAC2(SDA, SCL);
MCP4921 myAYDAC3(SDA, SCL);
MCP4921 myAYDAC4(SDA, SCL);

// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//                                  GIOP Pin Assignments

static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;

volatile short pot1;
volatile short pot2;
volatile short pot3;
volatile short pot4;

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

volatile bool switch1;
volatile bool switch2;

//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;

static const uint8_t V1 = 25;  //Voice Pins to MCP4921 Multiplyer DACs
static const uint8_t V2 = 26;
static const uint8_t V3 = 12;
static const uint8_t V4 = 13;

static const uint8_t X1 = 27;
static const uint8_t X2 = 14;

static const uint8_t CS_V1 = 23;   //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O, also LED on MIDI Out
static const uint8_t MIDI_RX2 = 16;   //only on WROOM, won't work on WROVER ESP32s
```

```
static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards

uint16_t AY_MidiNote_Hi[128] = {   // AY course tune -- upper 4-bits of 12-bit tune
  15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
  14, 14, 13, 12, 11, 11, 10, 9, 9, 8, 8, 7,
  7, 7, 6, 6, 5, 5, 5, 4, 4, 4, 4, 3,
  3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0 };

uint16_t AY_MidiNote_Lo[128] = {  // AY fine tune -- lower 8 bits of 12-bit tune
  211, 211, 211, 211, 211, 211, 211, 211, 211, 211, 211, 211,
  239, 25, 78, 141, 218, 47, 143, 247, 104, 225, 97, 233,
  119, 12, 167, 71, 237, 152, 71, 252, 180, 112, 49, 244,
  188, 134, 83, 36, 246, 204, 164, 126, 90, 56, 24, 250,
  222, 195, 170, 146, 123, 102, 82, 63, 45, 28, 12, 253,
  239, 225, 213, 201, 190, 179, 169, 159, 150, 142, 134, 127,
  119, 113, 106, 100, 95, 89, 84, 80, 75, 71, 67, 63,
  60, 56, 53, 50, 47, 45, 42, 40, 38, 36, 34, 32,
  30, 28, 27, 25, 24, 22, 21, 20, 19, 18, 17, 16,
  15, 14, 13, 13, 12, 11, 11, 10, 9, 9, 8, 8,
  7, 7, 7, 6, 6, 6, 5, 5 };

uint16_t AY_Volume[16] = {  //4-bit AY synth volume to 12-bit logarithmic volue
  0, 10, 25, 51, 62, 102, 124, 205, 307, 512, 621, 1024, 1241, 2048, 2896, 4095 };

  volatile uint16_t freq1 = 0;
  volatile uint16_t freq2 = 0;
  volatile uint16_t freq3 = 0;
  volatile uint16_t freq4 = 0;

  volatile uint16_t freq1save = 0;
  volatile uint16_t freq2save = 0;
  volatile uint16_t freq3save = 0;
  volatile uint16_t freq4save = 0;

  int count = 0;
  volatile int pcm_freq = 440;
  const int pcm_Channel0 = 0;
  const int pcm_resolution = 12;  // bits, 8 to 16.
  volatile int pcm_width = 2000;

// Touch9 Interrupt Function: sets up PCM frequency from pot4
 void gotTouch(){
    ledcSetup(pcm_Channel0, (pot4 + 30), pcm_resolution);
 }
```

```
// ~~~~~~~~~~~~~~~~~  Timer Interrupt  ~~~~~~~~~~~~~~~

// The hardware timer pointer
hw_timer_t * timer = NULL;


// Interrupt Routine.  Run every 16 microseconds.
void IRAM_ATTR onTimer() { serviceVoices(); }


// ~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  myAYDAC1.begin(CS_V1);
  myAYDAC2.begin(CS_V2);
  myAYDAC3.begin(CS_V3);
  myAYDAC4.begin(CS_V4);


  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

 //initialize 4 digital voice inputs to Multiplyer DACs
  pinMode(V1, OUTPUT);
  digitalWrite(V1, HIGH);
  pinMode(V2, OUTPUT);
  digitalWrite(V2, HIGH);
  pinMode(V3, OUTPUT);
  digitalWrite(V3, HIGH);
  pinMode(V4, OUTPUT);
  digitalWrite(V4, HIGH);

//set DACs to maximum for Voice inputs
  myAYDAC1.analogWrite(4095);
  myAYDAC2.analogWrite(4095);
  myAYDAC3.analogWrite(4095);
  myAYDAC4.analogWrite(4095);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);
```

```
   Serial.begin(115200);

 int threshold = 20;

// Setup Interrupt Function on Switch2 Touch control (pin T8).
// ESP32 Library Function that attaches interrupt pin T8 to the service routine
  touchAttachInterrupt(T8, gotTouch, threshold);


//create a task executed in Task1code() function, with priority 1 and executed on core 0
  xTaskCreatePinnedToCore(
                Task1code, /* Task function. */
                  "Task1", /* name of task (shown below). */
                  10000,  /* Stack size of task */
                  NULL,   /* parameter of the task */
                  1,      /* priority of the task */
                  &Task1, /* handle to keep track of task */
                  0       /* pin task to core0*/
  );

 delay(500);

}  //end of Setup

// ~~~~~~~~~~~~~~~~~~~  Task1 Code  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//Task1code:
  void Task1code( void * pvParameters ){
  Serial.print("Task1 running on core ");
  Serial.println(xPortGetCoreID());

// V4 Noise Voice created by random pulse width in a PWM waveform
// configure PWM functionalities
  ledcSetup(pcm_Channel0, pcm_freq, pcm_resolution);

// attach the PCM channel to V4
  ledcAttachPin(V4, pcm_Channel0);

// V1, V2, V3 voices are squarewaves. Freq set by a 16 microsecond timer interrupt routine.
// Initilise the timer interrupt.
  // Parameter 1 is the timer we want to use. Valid: 0, 1, 2, 3 (total 4 timers)
  // Parameter 2 is the prescaler. The ESP32 default clock is at 80MhZ.
  // (look under Arduino Menu Tools/Flash Frequency) The value "80" will
  // divide the clock by 80, giving us 1,000,000 ticks per second.
  // Parameter 3 is true means this counter will count up, instead of down (false).
  timer = timerBegin(0, 80, true);

  // Attach the timer to the interrupt service routine named "onTimer".
  // The 3rd parameter is set to "true" to indicate that we want to use the "edge" type (instead of "flat").
  timerAttachInterrupt(timer, &onTimer, true);

  // This is where we indicate the frequency of the interrupts.
  // The value "16" (because of the prescaler we set in timerBegin) will produce
  // one interrupt every 16 microseconds.
  // The 3rd parameter is true so that the counter reloads when it fires an interrupt, and so we
  // can get periodic interrupts (instead of a single interrupt).
  timerAlarmWrite(timer, 16, true);

  // Start the timer to perform the "onTimer" routine every 16 microseconds
  timerAlarmEnable(timer);
```

```
for(;;){  //loop to update pot and switch variables for use in Main Loop
          //timer interrupts for the voices happen in the background

    delay(30);
    loadSensors();
}
}
// ~~~~~~~~~~~~~~~~~~~  MAIN LOOP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {  //Here is where Voice Performance is programmed for the 4 voices

 count++;  //Simple test Voice Performance

if(count >= 10000){   // limit Voice changes
   freq1save = midi_to_Freq(pot1 >> 5);  // V1 Frequency set
   freq2save = midi_to_Freq(pot2 >> 5);  // V2 Frequency set
   freq3save = midi_to_Freq(pot3 >> 5);  // V3 Frequency set
   count = 0;

   // V4 Pitched Noise Freq set to pot4 value when touch8 is touched  ->
   // ledcSetup(pcm_Channel0, (pot4 + 30), pcm_resolution);  (pot4+30) = frequency in Hertz
}
} //end of Loop

// ~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
      pot1 =   analogRead(POT1) ;
      pot2 =   analogRead(POT2) ;
      pot3 =   analogRead(POT3) ;
      pot4 =   analogRead(POT4) ;

      switch1 = digitalRead(SWITCH1);
      switch2 = digitalRead(SWITCH2);
}

void serviceVoices(){
    --freq1;                        //toggle V1 at end of freq1 countdown
    if (freq1 <= 0){
        digitalWrite(V1, !digitalRead(V1));
        freq1 = freq1save;
    }
     --freq2;                       //toggle V2 at end of freq2 countdown
    if (freq2 <= 0){
        digitalWrite(V2, !digitalRead(V2));
        freq2 = freq2save;
    }
     --freq3;                       //toggle V3 at end of freq3 countdown
      if (freq3 <= 0){
        digitalWrite(V3, !digitalRead(V3));
        freq3 = freq3save;
    }
                                  //load random pulse Width into PCM V4

      ledcWrite(pcm_Channel0, random(2, 4093) );
}

uint16_t midi_to_Freq(uint8_t note){
  uint16_t x;
  x = AY_MidiNote_Lo[note] | (AY_MidiNote_Hi[note] << 8);
  return x;
}
```

Random Voices (2Core)

This sketch starts with the 2 Core, 3 Voice base of the previous sketch, then, in the Main Loop, a complete performance device is built based on random pitches and voice durations all controlled by the 4 pots and 2 switches.

~~~~~~~~~~~~~~~~~~~~~~~

```
/*
  ESP32 AY_Synth Setup

  Core 0 Task is to clock the Voice Frequencies using a timer interrupt
  and to keep the pot and switch variables updated

  Core 1 Task is the Main Loop to handle Voice Performance
*/

//already declared in ESP32 library
//static const uint8_t SCL = 22;   //Serial Lines to MCP4921 Multiplyer DACs
//static const uint8_t SDA = 21;

TaskHandle_t Task1;


#include "MCP_DAC.h"  //MCP_DAC Library by Rob Tillaart
MCP4921 myAYDAC1(SDA, SCL);
MCP4921 myAYDAC2(SDA, SCL);
MCP4921 myAYDAC3(SDA, SCL);
MCP4921 myAYDAC4(SDA, SCL);


// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//                              GIOP Pin Assignments

static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;

volatile short pot1;
volatile short pot2;
volatile short pot3;
volatile short pot4;

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

volatile bool switch1;
volatile bool switch2;
```
~~~~~~~~~~~~~~~~~~~~~~~

```
//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;

static const uint8_t V1 = 25;  //Voice Pins to MCP4921 Multiplyer DACs
static const uint8_t V2 = 26;
static const uint8_t V3 = 12;
static const uint8_t V4 = 13;

static const uint8_t X1 = 27;
static const uint8_t X2 = 14;

static const uint8_t CS_V1 = 23;   //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O, also LED on MIDI Out
static const uint8_t MIDI_RX2 = 16;   //only on WROOM, won't work on WROVER ESP32s

static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards

uint16_t AY_MidiNote_Hi[128] = {   // AY course tune -- upper 4-bits of 12-bit tune
  15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
  14, 14, 13, 12, 11, 11, 10, 9, 9, 8, 8, 7,
  7, 7, 6, 6, 5, 5, 5, 4, 4, 4, 4, 3,
  3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0 };

uint16_t AY_MidiNote_Lo[128] = {  // AY fine tune -- lower 8 bits of 12-bit tune
  211, 211, 211, 211, 211, 211, 211, 211, 211, 211, 211, 211,
  239, 25, 78, 141, 218, 47, 143, 247, 104, 225, 97, 233,
  119, 12, 167, 71, 237, 152, 71, 252, 180, 112, 49, 244,
  188, 134, 83, 36, 246, 204, 164, 126, 90, 56, 24, 250,
  222, 195, 170, 146, 123, 102, 82, 63, 45, 28, 12, 253,
  239, 225, 213, 201, 190, 179, 169, 159, 150, 142, 134, 127,
  119, 113, 106, 100, 95, 89, 84, 80, 75, 71, 67, 63,
  60, 56, 53, 50, 47, 45, 42, 40, 38, 36, 34, 32,
  30, 28, 27, 25, 24, 22, 21, 20, 19, 18, 17, 16,
  15, 14, 13, 13, 12, 11, 11, 10, 9, 9, 8, 8,
  7, 7, 7, 6, 6, 6, 5, 5 };

uint16_t AY_Volume[16] = {  //4-bit AY synth volume to 12-bit logarithmic volue
  0, 10, 25, 51, 62, 102, 124, 205, 307, 512, 621, 1024, 1241, 2048, 2896, 4095 };

  volatile uint16_t freq1 = 0;
  volatile uint16_t freq2 = 0;
  volatile uint16_t freq3 = 0;
  volatile uint16_t freq4 = 0;

  volatile uint16_t freq1save = 0;
  volatile uint16_t freq2save = 0;
  volatile uint16_t freq3save = 0;
  volatile uint16_t freq4save = 0;
  int count = 0;

  int freq=0;
  int durA=0;
  int durA_count=0;
  int durB=0;
  int durB_count=0;
  int durC=0;
  int durC_count=0;
  int envA=0;
  int envB=0;
  int envC=0;
  int dur=0;


// ~~~~~~~~~~~~~~~~~  Timer Interrupt  ~~~~~~~~~~~~~~~~

// The hardware timer pointer
hw_timer_t * timer = NULL;


// Interrupt Routine.  Run every 16 microseconds.
void IRAM_ATTR onTimer() { serviceVoices(); }


// ~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  myAYDAC1.begin(CS_V1);
  myAYDAC2.begin(CS_V2);
  myAYDAC3.begin(CS_V3);
  myAYDAC4.begin(CS_V4);


  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

 //initialize 4 digital voice inputs to Multiplyer DACs
  pinMode(V1, OUTPUT);
  digitalWrite(V1, HIGH);
  pinMode(V2, OUTPUT);
  digitalWrite(V2, HIGH);
  pinMode(V3, OUTPUT);
  digitalWrite(V3, HIGH);
  pinMode(V4, OUTPUT);
  digitalWrite(V4, HIGH);
```

```
//set DACs to minimum for Voice inputs
  myAYDAC1.analogWrite(0);
  myAYDAC2.analogWrite(0);
  myAYDAC3.analogWrite(0);
  myAYDAC4.analogWrite(0);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);

   Serial.begin(115200);

//create a task executed in Task1code() function, with priority 1 and executed on core 0
  xTaskCreatePinnedToCore(
                Task1code, /* Task function. */
                  "Task1", /* name of task (shown below). */
                  10000,  /* Stack size of task */
                  NULL,   /* parameter of the task */
                  1,      /* priority of the task */
                  &Task1, /* handle to keep track of task */
                  0       /* pin task to core0*/
  );

 delay(500);

}  //end of Setup

// ~~~~~~~~~~~~~~~~~~~  Task1 Code  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//Task1code:
  void Task1code( void * pvParameters ){
  Serial.print("Task1 running on core ");
  Serial.println(xPortGetCoreID());

// Initilise the timer interrupt.
  // Parameter 1 is the timer we want to use. Valid: 0, 1, 2, 3 (total 4 timers)
  // Parameter 2 is the prescaler. The ESP32 default clock is at 80MhZ.
  // (look under Arduino Menu Tools/Flash Frequency) The value "80" will
  // divide the clock by 80, giving us 1,000,000 ticks per second.
  // Parameter 3 is true means this counter will count up, instead of down (false).
  timer = timerBegin(0, 80, true);

  // Attach the timer to the interrupt service routine named "onTimer".
  // The 3rd parameter is set to "true" to indicate that we want to use the "edge" type (instead of "flat").
  timerAttachInterrupt(timer, &onTimer, true);

  // This is where we indicate the frequency of the interrupts.
  // The value "16" (because of the prescaler we set in timerBegin) will produce
  // one interrupt every 16 microseconds.
  // The 3rd parameter is true so that the counter reloads when it fires an interrupt, and so we
  // can get periodic interrupts (instead of a single interrupt).
  timerAlarmWrite(timer, 16, true);
```

```
  // Start the timer to perform the "onTimer" routine every 16 microseconds
  timerAlarmEnable(timer);

for(;;){  //loop to update pot and switch variables for use in Main Loop

    delay(30);
    loadSensors();
}
}

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

void loop() {

 dur = pot1 >> 3; //set envelope durations

// -----------Toggle Switch turns on noise in one voice ------------------
/*
               Toggle = digitalRead(A2);
               if (Toggle){
               ldEnable(B110000);  //Enable noise and tones in one voice (low enable)
               }
               else {
               ldEnable(B111000);  //Enable only tones (low enable)
               }

*/
// ------------Voice A from AY Chip---------------------

if (envA != 0){  //ramping down voice A envelope, 4095 to 0

           if (durA_count != 0){  //wait for a count of durA
                durA_count -= 1;
           }
             else {  // when the count reaches zero decrement voice A envelope, reset count
                durA_count = durA;
                envA -= 1;
                myAYDAC1.analogWrite(envA);
             }
  }
  else{  // when envelope reaches zero, reset voice A with new frequency and envelope

           //get new random pitch for voice A
           if(switch2) { freq1save = getFreq(); }
           else { freq1save = midi_to_Freq(getNote()); }

           durA = random(1, dur) ;  // get random 8 bit duration for envelope A

           durA_count = durA;
           envA = 4095;
           myAYDAC1.analogWrite(envA); //set voice A full on
   }
```

```
 // ------------Voice B from AY Chip---------------------

if (envB != 0){  //ramping down voice B envelope, 4095 to 0

        if (durB_count != 0){  //wait for a count of durB
            durB_count -= 1;
        }
          else {  // when the count reaches zero decrement voice B envelope, reset count
            durB_count = durB;
            envB -= 1;
            myAYDAC2.analogWrite(envB);
          }
 }
 else{  // when envelope reaches zero, reset voice B with new frequency and envelope

        //get new random pitch for voice A
        if(switch2) { freq2save = getFreq(); }
        else { freq2save = midi_to_Freq(getNote()); }

        durB = random(1, dur) ;  // get random 8 bit duration for envelope B

        durB_count = durB;
        envB = 4095;
        myAYDAC2.analogWrite(envB); //set voice B full on
  }


// ------------Voice C from AY Chip---------------------

if (envC != 0){  //ramping down voice C envelope, 4095 to 0

        if (durC_count != 0){  //wait for a count of durC
            durC_count -= 1;
        }
          else {  // when the count reaches zero decrement voice C envelope, reset count
            durC_count = durC;
            envC -= 1;
            myAYDAC3.analogWrite(envC);
          }
 }
 else{  // when envelope reaches zero, reset voice C with new frequency and envelope

        //get new random pitch for voice A
        if(switch2) { freq3save = getFreq(); }
        else { freq3save = midi_to_Freq(getNote()); }

        durC = random(1, dur) ;  // get random 8 bit duration for envelope C

        durC_count = durC;
        envC = 4095;
        myAYDAC3.analogWrite(envC);//set voice C full on
  }

 // ------------Switch 1 Slows everything to almost a standstill---------------------
  if (switch1 == 0){
   delay(pot4 >> 4);
  }
  //delayMicroseconds(100);
} // End of Loop
```

```
// ~~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

int getFreq() {   // getting random frequency for Voices
   int basefreq = (pot3 >> 4);
   int result = basefreq + random(pot2 >> 3);
   return result;
}

int getNote() {  // getting random MIDI Note for Voices
   int basenote = ((pot3 >> 5) + 10);  // 0 to 127
   int note = basenote + random(pot2 >> 5);
   int result = min(note, 127);
   return result;
}

void loadSensors(){      // load all current sensor values
     pot1 =   analogRead(POT1) ;
     pot2 =   analogRead(POT2) ;
     pot3 =   analogRead(POT3) ;
     pot4 =   analogRead(POT4) ;

     switch1 = digitalRead(SWITCH1);
     switch2 = digitalRead(SWITCH2);
}

void serviceVoices(){
   --freq1;                        //toggle V1 at end of freq1 countdown
   if (freq1 <= 0){
      digitalWrite(V1, !digitalRead(V1));
      freq1 = freq1save;
   }
    --freq2;                       //toggle V2 at end of freq2 countdown
   if (freq2 <= 0){
      digitalWrite(V2, !digitalRead(V2));
      freq2 = freq2save;
   }
    --freq3;                       //toggle V3 at end of freq3 countdown
     if (freq3 <= 0){
      digitalWrite(V3, !digitalRead(V3));
      freq3 = freq3save;
   }
}

uint16_t midi_to_Freq(uint8_t note){
  uint16_t x;
  x = AY_MidiNote_Lo[note] | (AY_MidiNote_Hi[note] << 8);
  return x;
}
```

SPIFFS File System

SPI Flash File Storage is non-volatile flash memory that acts like a small SD Card onboard the ESP32 chip. Looking at the Tools/PartitionScheme in the Arduino IDE Menus, you will typically see 1.5 MB of the 4MB flash memory of the ESP32 allocated to SPIFFS.

SPIFFS is used for storing files of any kind. In our next sketch, it will make a convenient space to hold AY Arcade Game sound files to be played back.

The sketch here lists the sources for downloading the necessary libraries used to access the SPIFFS memory. The sketch includes two libraries:

include "FS.h" // File System
include "SPIFFS.h" // SPI Flash File Storage

FS library: https://github.com/espressif/arduino-esp32/tree/master/libraries/FS
SPIFFS library: https://github.com/espressif/arduino-esp32/tree/master/libraries/SPIFFS

These libraries also provide a SPIFFS_Test.ino sketch in the Arduino Examples Menu. The functions demonstrated in that library are a bit different from some of the SPIFFS access functions in this sketch, but both do the same thing, allowing you to openFile, readFile, writeFile, appendFile, renameFile, deleteFile, listDirectories, and so on.

Loading files into SPIFFS memory can be done from Arduino code, but it is not easy. Fortunately, there is a plugin for the Arduino IDE that makes loading files easy.

ESP32 sketch data upload tool: https://github.com/me-no-dev/arduino-esp32fs-plugin/releases/

https://randomnerdtutorials.com/install-esp32-filesystem-uploader-arduino-ide/

Follow the directions from the random-nerd-turtorials given above to install the Arduino ESP32 Filesystem Uploader. Once installed, put your files into a folder named "data" and put that folder inside the folder containing this .ino sketch. Open the sketch and execute the command on the Arduino Menu Tools/ESP32_Sketch_Data_Upload. While uploading be sure the Arduino Monitor window is closed, and press the Boot button on the ESP32 board. The Data_Upload will also fail if the data folder size is greater than about 1MB.

~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~

```
/*********
 *
 *  - Reading and writing AY synth register files from the SPIFFS
 *
 * This sketch demonstrates how to read, write and append to files stored in
 * the SPIFFS of an ESP32.
 *
 * AY .reg (register) files uploaded to SPIFF using Tools/"ESP32 Sketch Data Upload"
 * after loading files into "data" folder in the Arduino folder of this sketch.

 * create a new folder named "data". Store the AY register files in the data directory. Use the
 * SPIFFS upload tool from the Arduino IDE to upload the files to the SPIFFS. Then run
 * the sketch.
 *
 *  1. ESP32 Datasheet: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
 *  2. FS library: https://github.com/espressif/arduino-esp32/tree/master/libraries/FS
 *  3. SPIFFS library: https://github.com/espressif/arduino-esp32/tree/master/libraries/SPIFFS
 *  4. ESP32 sketch data upload tool: https://github.com/me-no-dev/arduino-esp32fs-plugin/releases/
 *  5. ESP32 FS tool (useful for many things, including erasing flash memory): https://github.com/espressif/esptool
 *
 *  File manipulation functions taken from SPIFFS Library Examples/SPIFFS/SPIFFS_Test
 *  And SPIFFS_Manipulating_File.ino in "Learn ESP32 with Arduino IDE" by Rui and Sara Santos.
 *
 *  Arcade AYregister files from Daniel Tufvesson at http://www.waveguide.se/?article=ym-playback-on-the-ymz284
 *
*********/

#include "FS.h"
#include "SPIFFS.h"

// Create a File object to manipulate your file
File myFile;

// File paths
const char* myFilePath1 = "/cybernoid.reg";
const char* myFilePath2 = "/delta.reg";
const char* myFilePath3 = "/stormlord.reg";
const char* myFilePath4 = "/outrun1.reg";
const char* myFilePath5 = "/sidewinder.reg";

  uint8_t AYdata[14];
  //~~~~~~~~~~~~~~~~~~~~~~~~~  SETUP ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup(){
  // Serial Monitor
  Serial.begin(115200);



  Serial.println("");
  Serial.println("");

  // Initialize SPIFFS
  if(!SPIFFS.begin(true)){
    Serial.println("Error while mounting SPIFFS");
    return;
  }

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

   if(SPIFFS.remove(myFilePath1)){
    Serial.println("File successfully deleted");
  }
  else{
    Serial.print("Deleting file failed!");
  }

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

   if(SPIFFS.remove(myFilePath2)){
    Serial.println("File successfully deleted");
  }
  else{
    Serial.print("Deleting file failed!");
  }

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

   if(SPIFFS.remove(myFilePath3)){
    Serial.println("File successfully deleted");
  }
  else{
    Serial.print("Deleting file failed!");
  }

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

   if(SPIFFS.remove(myFilePath4)){
    Serial.println("File successfully deleted");
  }
  else{
    Serial.print("Deleting file failed!");
  }

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

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

   if(SPIFFS.remove(myFilePath5)){
    Serial.println("File successfully deleted");
  }
  else{
    Serial.print("Deleting file failed!");
  }

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

  listDir(SPIFFS, "/", 0);  //see function below
    Serial.println("");

    myFile = SPIFFS.open(myFilePath2, FILE_READ);  //specify file to be printed

    Serial.print(myFile.name());
    Serial.print(" Opened for Printing ------- File size: ");
    Serial.println(myFile.size());
    Serial.println("");

//  14 AY File register functions
    Serial.print("A_FreqL A_FreqH B_FreqL B_FreqH C_FreqL C_FreqH ");
    Serial.print("N_Freq Enables  A_Amp   B_AMP   C_AMP");
    Serial.println("");
    Serial.println("");

   while(myFile.available()) {  //Print 11 AY registers per line,
    //print in a format that could be copied and pasted to a playback 2-dimension array

      Serial.print("{ ");          // print bank start
      for(int x=0; x<10; x++){

      PrintHex8(myFile.read(), 1); //print 10 hex values with cammas
      Serial.print(",    ");
      }

      PrintHex8(myFile.read(), 1);  // print 11th value without camma
      Serial.print(" }, ");         // print bank end and comma

      myFile.read(); myFile.read();  myFile.read();  // read 3 unused ENV bytes

      Serial.println(" ");          // new line for new bank

  } // End of while available

    myFile.close();
    Serial.println("File Closed ");

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


/*
```

```
//~~~~~~~~~~~~~~~~~~~Other SPIFFS Functions~~~~~~~~~~~~~~~~


  // Open file and write data to it
  myFile = SPIFFS.open(myFilePath, FILE_WRITE);
  if (myFile.print("Example message in write mode")){
    Serial.println("Message successfully written");
  }
  else{
    Serial.print("Writting message failled!!");
  }
  myFile.close();

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


  // Append data to file
  myFile = SPIFFS.open(myFilePath, FILE_APPEND);
  if(myFile.print(" - Example message appended to file")){
    Serial.println("Message successfully appended");
  }
  else{
    Serial.print("Appending failled!");
  }
  myFile.close();

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

  // Read file content
  myFile = SPIFFS.open(myFilePath, FILE_READ);
  Serial.print("File content: \"");
  while(myFile.available()) {
    Serial.write(myFile.read());
  }
  Serial.println("\"");

  // Check file size
  Serial.print(myFile.name());
  Serial.print(" File size: ");
  Serial.println(myFile.size());

  myFile.close();

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

  void renameFile(fs::FS &fs, const char * path1, const char * path2){
    Serial.printf("Renaming file %s to %s\r\n", path1, path2);
    if (fs.rename(path1, path2)) {
        Serial.println("- file renamed");
    } else {
        Serial.println("- rename failed");
    }
}

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

```
  // Delete file
  if(SPIFFS.remove(myFilePath)){
    Serial.println("File successfully deleted");
  }
  else{
    Serial.print("Deleting file failed!");
  }

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

*/
}

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

void loop(){   // Everything runs in Setup one time only

}

//~~~~~~~~~~~~~~~~~~~  FUNCTIONS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void PrintHex8(uint8_t data, uint8_t length) // prints 8-bit data in hex with leading zeroes
{
        for (int i=0; i<length; i++) {
          Serial.print("0x");
          if (data<0x10) {Serial.print("0");}
          Serial.print(data,HEX);
        }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  void listDir(fs::FS &fs, const char * dirname, uint8_t levels){
    Serial.printf("Listing directory: %s\r\n", dirname);

    File root = fs.open(dirname);
    if(!root){
        Serial.println("- failed to open directory");
        return;
    }
    if(!root.isDirectory()){
        Serial.println(" - not a directory");
        return;
    }

    File file = root.openNextFile();
    while(file){
        if(file.isDirectory()){
            Serial.print("  DIR : ");
            Serial.println(file.name());
            if(levels){
                listDir(fs, file.name(), levels -1);
            }
        } else {
            Serial.print("  FILE: ");
            Serial.print(file.name());
            Serial.print("\tSIZE: ");
            Serial.println(file.size());
        }
        file = root.openNextFile();
    }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

AY Arcade Playback

The **AY-3-8910** is a 3-voice programmable sound generator chip designed by General Instrument. The AY-3-8910 and its variants became popular chips in many arcade games during the 70s and 80s. It was essentially a state machine, with the state being set up in a series of sixteen 8-bit registers. See the next page diagram of the chip architecture with its registers.

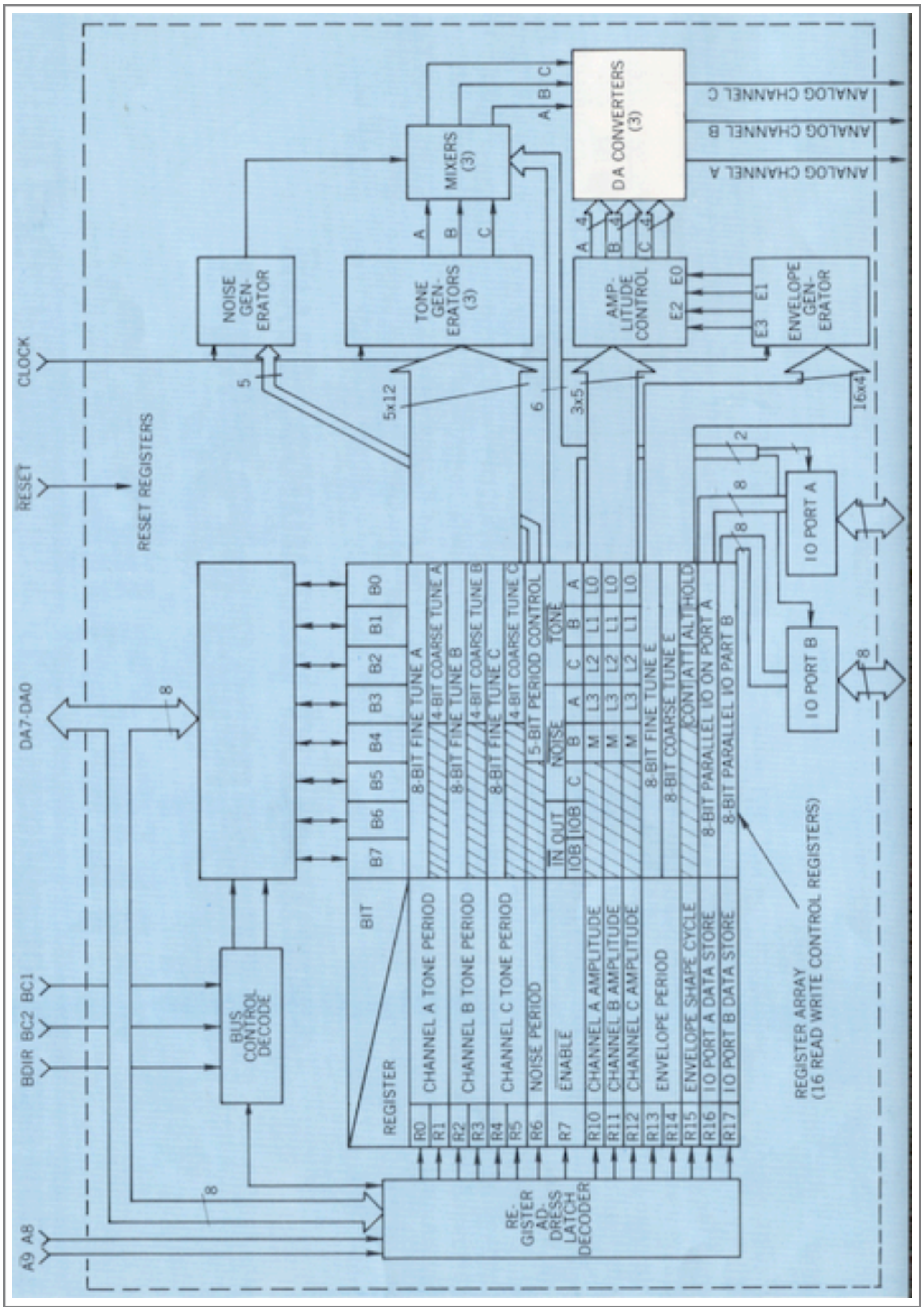
Six registers controlled the pitches produced in the three primary channels. Three squarewaves were generated by dividing down a master clock of 1MHz/16 by the contents of one eight-bit and one 4-bit register dedicated to each channel. 12 bits then gave a total of 4095 possible pitches. Another register controlled the period of a pseudo-random noise generator. A 6-bit Enable register controlled the mixing — one bit to enable or disable tones in each of 3 output channels, and one bit to enable or disable noise in each channel. Three additional 4-bit registers controlled the logarithmic volume of the 3 channels. Finally the last three registers controlled the times of an envelope controller.

When it came to actual practice, however, the three envelope registers were hardly ever used. The practice was to load a bank of all 14 8-bit registers every 50 milliseconds from a file of consecutive register banks, whether or not the values actually changed. Three 4-bit volume registers loaded every 50 ms allowed for real-time volume envelopes negating the need for any other envelope control.

AY playback files can still be found on the Web (**www.modland.net**), however they come heavily compressed. Daniel Tufvesson (**http://www.waveguide.se/?article=ym-playback-on-the-ymz284)** has outlined a method for decompressing the AY files into Register Dump files and has provided several example ".reg" files used for playback in this sketch. Several of these register bank files were uploaded to the ESP32 SPIFFS flash memory following the directions presented in the previous sketch.

The following AY Arcade Playback sketch follows closely the inner workings of the AY chip and the conventional Arcade Game method for feeding data to the chip every 50 ms.

Core0 Task1 clocks the 3 square wave voices from a 16 microsecond Timer Interrupt. The voice pins are toggled after counting down from a 12-bit frequency counter, just as done in hardware on the AY chip. A fourth noise voice is generated in the Main Loop by feeding random pulse width values to a PWM signal. The pitched part of the noise signal is generated by directly loading a 5-bit frequency Hertz value into the ledcSetup() function.



A9 A8
BDIR BC2 BC1
DA7-DA0
RESET
CLOCK
RESET REGISTERS
BUS CONTROL DECODE
REGISTER ADDRESS LATCH DECODER
8
NOISE GENERATOR
TONE GENERATORS (3)
AMPLITUDE CONTROL
ENVELOPE GENERATOR
MIXERS (3)
DA CONVERTERS (3)
ANALOG CHANNEL A
ANALOG CHANNEL B
ANALOG CHANNEL C
E0 E1 E2 E3
A B C
4
5
5x12
6
3x5
16x4
2
IO PORT A
IO PORT B
REGISTER
BIT
B7 B6 B5 B4 B3 B2 B1 B0
R0 CHANNEL A TONE PERIOD 8-BIT FINE TUNE A
R1 4-BIT COARSE TUNE A
R2 CHANNEL B TONE PERIOD 8-BIT FINE TUNE B
R3 4-BIT COARSE TUNE B
R4 CHANNEL C TONE PERIOD 8-BIT FINE TUNE C
R5 4-BIT COARSE TUNE C
R6 NOISE PERIOD 5-BIT PERIOD CONTROL
R7 ENABLE IN OUT IOB IOB NOISE C B A TONE C B A
R10 CHANNEL A AMPLITUDE M L3 L2 L1 L0
R11 CHANNEL B AMPLITUDE M L3 L2 L1 L0
R12 CHANNEL C AMPLITUDE M L3 L2 L1 L0
R13 ENVELOPE PERIOD 8-BIT FINE TUNE E
R14 8-BIT COARSE TUNE E
R15 ENVELOPE SHAPE CYCLE CONT ATT ALT HOLD
R16 IO PORT A DATA STORE 8-BIT PARALLEL IO ON PORT A
R17 IO PORT B DATA STORE 8-BIT PARALLEL IO PART B
REGISTER ARRAY (16 READ WRITE CONTROL REGISTERS)

The Main Loop (Core 1) has the job of reading consecutive AY register banks from an AY register file every 50 milliseconds, translating and feeding frequency data to the Core 0 tone generator and volume data to the four Multiplying DAC chips.

The function AY_SynthLoad( ) combines the the 4-bit Course Tune from the AY registers and the 8-bit Fine tune for each voice to load into the 12-bit voice frequency registers freq1save, freq2save, and freq3save. The AY Enable register is used to turn on or off each of the 3 square wave and the one noise voice source connected to the inputs of the 4 Multiplying DACs . If a voice is "ON" then its 4-bit volume value from the AY registers is used to address the 16 element array AY_Volume[] which translates the 4-bit volume value to a 12-bit logarithmic volume fed to the 4 Multiplying DAC chips.

The Main Loop also controls playback. Pot 1 is used to speed up or slow down the playback around the normal 50 millisecond delay value used to time the register bank loads. Switch 1 is used to jump back or Rewind the playback to any time point in the AY register bank file, as set by pot 4. Switch 2 can stop playback at the current notes.

The ESP32 has some not so obvious limitations. When pushed too far the program will crash as exhibited by constantly resetting. Here are some of the compromises found necessary to deal with some ESP32 limitations.

1. Originally, I planned to read one bank of AY registers at a time as needed, directly from the SPIFFS memory. This caused the program to crash when attempted inside either core. As it turns out, Timer and SPIFFS operations cannot run at the same time. To accommodate this limitation, one user-chosen AY file is read from SPIFFS memory and loaded into a large 2 dimensional SRAM memory array at the start of Setup. This happens before Core 0 Tasks are started. The Main Loop then accesses each bank of AY registers from this SRAM array instead of directly from the SPIFFS flash memory.

2. Even though the ESP32 has much more Ram memory than any of the Arduinos, it doesn't turn out to have enough to hold some of the AY files which can be as big as 300KB. 100KB seemed to be the maximum size that would not crash the program. It was possible to compress the file size somewhat by throwing out the 3 unused envelope bytes thus cutting the AY bank size to 11 bytes instead of 14. If the size was still over 100KB the end of the file was cut.

3. The forth voice is a PCM noise source that must be fed periodic random pulse widths. When this operation was added to the Core 0 tasks the ESP would crash, perhaps due to conflicting timers? This problem was solved by putting the operation in the Main Loop instead. None of the AY example files actually used this voice. The percussive sounds are instead frequency modulated low pitches.

4. On playing the AY file examples, the pitches were found to be two octaves too low. One fix was to lower the Timer Interrupt time to 4 microseconds instead of 16. That introduced some rough artifacts into the voices making them a little less clear. It was pushing the timing of the software too close to the ESP32's speed limitations.

Instead, the 12-bit pitch countdown values for the 3 voices were bit shifted to the right by 2 bits, throwing out 2 lower bits. That made the voices two octaves higher at the cost of losing some pitch accuracy since the pitch values are now 10 bits instead of 12. That loss of accuracy will mainly affect the higher pitches.

~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~

```
/*
  ESP32 AY_Synth Setup

  1. First in Setup, download Arcade game sound file from SPIFFS flash memory and load into AYArray[][]
  2. In Core 0 create a timer interrupt to generate 3 squarewave voices (in Setup).  ServiceVoices()
  3. Create a PWM waveform for 4th voice. Feed it random pulse widths in Main Loop.
  4. In Main Loop, control playback of sound file with sensor pots and switches.


/////////////////////////////////////////////////////////////////////
 AY registers files already placed into SPIFFS flash Memory

   Arcade game AY register files from Daniel Tufvesson at
   http://www.waveguide.se/?article=ym-playback-on-the-ymz284

  file name.reg ---- file size ------ filesize/14 --> number of register banks in file
 cybernoid.reg 279720, delta.reg 64540, outrun1.reg 134400, sidewinder.reg 102270, stormlord.reg 260736
        19980            4610             9600                 7305               18624
*/

//~~~~~~~~~~~ TYPE IN AY FILE NAME TO PLAY BACK HERE ~~~~~~~~~~~~~~~~~//

          const char* AYFilePath = "/cybernoid2.reg";

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

/////////////////////////////////////////////////////////////////////

  #include "FS.h"
  #include "SPIFFS.h"

  File AYFile;

  #define INDEXMAX 10000  // Sorry, largest # of Register Banks that will compile without error
  int  FileSize = 0;      // Actual AY file size, read from SPIFFS AYFile.size()
  int  AYIndexMax = 0;    // calculated from AY File size, limited to INDEXMAX
  int  AYIndex = 0;
  uint8_t  AYArray[INDEXMAX][11];  //Allot memory space for maximum size even if not needed

  TaskHandle_t Task1;

//already declared in ESP32 library
//static const uint8_t SCL = 22;   //Serial Lines to MCP4921 Multiplyer DACs
//static const uint8_t SDA = 21;

#include "MCP_DAC.h"  //MCP_DAC Library by Rob Tillaart
MCP4921 myAYDAC1(SDA, SCL);
MCP4921 myAYDAC2(SDA, SCL);
MCP4921 myAYDAC3(SDA, SCL);
MCP4921 myAYDAC4(SDA, SCL);

// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~
//                                    GIOP Pin Assignments


static const uint8_t POT1 = 36;
static const uint8_t POT2 = 39;
static const uint8_t POT3 = 34;
static const uint8_t POT4 = 35;
```

```
volatile short pot1;
volatile short pot2;
volatile short pot3;
volatile short pot4;

static const uint8_t SWITCH1 = 33;  //Also Touch Sensor
static const uint8_t SWITCH2 = 32;

volatile bool switch1;
volatile bool switch2;

//already declared in ESP32 library
//static const uint8_t DAC1 = 25;
//static const uint8_t DAC2 = 26;

static const uint8_t V1 = 25;  //Voice Pins to MCP4921 Multiplyer DACs
static const uint8_t V2 = 26;
static const uint8_t V3 = 12;
static const uint8_t V4 = 13;

static const uint8_t X1 = 27;
static const uint8_t X2 = 14;

static const uint8_t CS_V1 = 23;   //Chip Select to MCP4921 Multiplyer DACs
static const uint8_t CS_V2 = 19;
static const uint8_t CS_V3 = 18;
static const uint8_t CS_V4 = 4;

static const uint8_t MIDI_TX2 = 17;   //MIDI I/O, also LED on MIDI Out
static const uint8_t MIDI_RX2 = 16;   //only on WROOM, won't work on WROVER ESP32s

static const uint8_t LED1 = 5;
static const uint8_t LED2 = 15;
static const uint8_t LED3 = 2;  //Blue LED on 32S boards

uint16_t AY_Volume[16] = {  //4-bit AY synth volume to 12-bit logarithmic volue
  0, 10, 25, 51, 62, 102, 124, 205, 307, 512, 621, 1024, 1241, 2048, 2896, 4095 };

  volatile uint16_t freq1 = 0;
  volatile uint16_t freq2 = 0;
  volatile uint16_t freq3 = 0;
  volatile uint16_t freq4 = 0;

  volatile uint16_t freq1save = 0;
  volatile uint16_t freq2save = 0;
  volatile uint16_t freq3save = 0;
  volatile uint16_t freq4save = 0;

  int count = 0;
  volatile int pcm_freq = 440;
  const int pcm_Channel0 = 0;
  const int pcm_resolution = 12;  // bits, 8 to 16.
  volatile int pcm_width = 2000;
  uint8_t AYNoiseFreq = 0;
  uint8_t AYNoiseAmpA = 0;
  uint8_t AYNoiseAmpB = 0;
  uint8_t AYNoiseAmpC = 0;


// ~~~~~~~~~~~~~~~~~  Timer Interrupt  ~~~~~~~~~~~~~~~

// The hardware timer pointer
hw_timer_t * timer = NULL;


// Interrupt Routine.  Run every 16 microseconds.
void onTimer(){ serviceVoices(); }



// ~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

  Serial.begin(115200);

  AYFileLoad(); // AY game sound file load, be sure to enter file name above

  myAYDAC1.begin(CS_V1);
  myAYDAC2.begin(CS_V2);
  myAYDAC3.begin(CS_V3);
  myAYDAC4.begin(CS_V4);


  // initialize Switches with pullup resistor
  pinMode(SWITCH1, INPUT_PULLUP);
  pinMode(SWITCH2, INPUT_PULLUP);

 //initialize DAC chip selects, LOW select, unselect all
  pinMode(CS_V1, OUTPUT);
  digitalWrite(CS_V1, HIGH);
  pinMode(CS_V2, OUTPUT);
  digitalWrite(CS_V2, HIGH);
  pinMode(CS_V3, OUTPUT);
  digitalWrite(CS_V3, HIGH);
  pinMode(CS_V4, OUTPUT);
  digitalWrite(CS_V4, HIGH);

 //initialize 4 digital voice inputs to Multiplyer DACs
  pinMode(V1, OUTPUT);
  digitalWrite(V1, HIGH);
  pinMode(V2, OUTPUT);
  digitalWrite(V2, HIGH);
  pinMode(V3, OUTPUT);
  digitalWrite(V3, HIGH);
  pinMode(V4, OUTPUT);
  digitalWrite(V4, HIGH);

//set DACs to maximum for Voice inputs
  myAYDAC1.analogWrite(4095);
  myAYDAC2.analogWrite(4095);
  myAYDAC3.analogWrite(4095);
  myAYDAC4.analogWrite(4095);
```

```
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  digitalWrite(LED3, HIGH);  //flash board's blue LED
  delay(500);
  digitalWrite(LED3, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
  digitalWrite(LED3, LOW);

  //create a task executed in Task1code() function, with priority 1 and executed on core 0
  xTaskCreatePinnedToCore(
              Task1code, /* Task function. */
                "Task1", /* name of task (shown below). */
                10000,  /* Stack size of task */
                NULL,   /* parameter of the task */
                1,      /* priority of the task */
                &Task1, /* handle to keep track of task */
                0       /* pin task to core0*/
  );

 delay(500);

}  //End of Setup


// ~~~~~~~~~~~~~~~~~~~  Task1 Code -- Core 0 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Do a one time load of an AY performance file from previously loaded SPIFFS memory before this Task1
// Perform 16 microsecond timer interrupt function to toggle voice squarewaves at prescribed frequencies,
// Load Voice 4 Noise with random pulse widths only in Main Loop
// Also read sensor pots and switches for use in main loop

//Task1code:
  void Task1code( void * pvParameters ){
  Serial.print("");
  Serial.print("Task1 running on core ");
  Serial.println(xPortGetCoreID());

 // V4 Noise Voice created by random pulse width in a PWM waveform
// configure PWM functionalities
  ledcSetup(pcm_Channel0, pcm_freq, pcm_resolution);

// attach the PCM channel to V4
  ledcAttachPin(V4, pcm_Channel0);

// V1, V2, V3 voices are squarewaves. Freq set by a 4 microsecond timer interrupt routine.
// Initilise the timer interrupt.
  // Parameter 1 is the timer we want to use. Valid: 0, 1, 2, 3 (total 4 timers)
  // Parameter 2 is the prescaler. The ESP32 default clock is at 80MhZ.
  // (look under Arduino Menu Tools/Flash Frequency) The value "80" will
  // divide the clock by 80, giving us 1,000,000 ticks per second.
  // Parameter 3 is true means this counter will count up, instead of down (false).
  timer = timerBegin(0, 80, true);
```

```
  // Attach the timer to the interrupt service routine named "onTimer".
  // The 3rd parameter is set to "true" to indicate that we want to use the "edge" type (instead of "flat").
  timerAttachInterrupt(timer, &onTimer, true);

  // This is where we indicate the frequency of the interrupts.
  // The value "16" (because of the prescaler we set in timerBegin) will produce
  // one interrupt every 16 microseconds.
  // The 3rd parameter is true so that the counter reloads when it fires an interrupt, and so we
  // can get periodic interrupts (instead of a single interrupt).
  timerAlarmWrite(timer, 16, true);

  // Start the timer to perform the "onTimer" routine every 16 microseconds
  timerAlarmEnable(timer);

for(;;){  //loop to update pot and switch variables for use in Main Loop

    delay(100);
    loadSensors();

}  //End of for loop
}  //End of Task1 code


// ~~~~~~~~~~~~~~~~~~~~  MAIN LOOP -- Core 1  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {  //Here is where Voice Performance is programmed for the 4 voices

    AY_SynthLoad(AYIndex);  // Load the software synth from one set of 8 AY_Synth registers at AYIndex

    AYIndex++;  // Increment index in preparatioon to load next set of 8 registers after some milliseconds

                                                   // PLAYBACK CONTROLS
  if (AYIndex >= AYIndexMax) {AYIndex = 0;}                    // playback repeat (at end of file)
  if (!switch1) { AYIndex = map(pot4, 0, 4095, 0, AYIndexMax); }  // playback rewind (switch1) to any point (on pot2)
  delay((pot1 >> 3) + 15);                                // playback speed set by pot 1
  while (!switch2){ switch2 = digitalRead(SWITCH2); }            // Stop playback at current notes with switch 2

  ledcWrite(pcm_Channel0, random(2, 4093) );                    // generating noise voice
  // doing this in Core 0 Task will crash the program.  PWM timer vs Interrupt timer ??

} //end of Loop



// ~~~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
      pot1 =   analogRead(POT1) ;
      pot2 =   analogRead(POT2) ;
      pot3 =   analogRead(POT3) ;
      pot4 =   analogRead(POT4) ;

      switch1 = digitalRead(SWITCH1);
      switch2 = digitalRead(SWITCH2);
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
void serviceVoices(){
   --freq1;                    //toggle V1 at end of freq1 countdown
   if (freq1 <= 0){
      digitalWrite(V1, !digitalRead(V1));
      freq1 = freq1save;
   }
    --freq2;                   //toggle V2 at end of freq2 countdown
   if (freq2 <= 0){
      digitalWrite(V2, !digitalRead(V2));
      freq2 = freq2save;
   }
    --freq3;                   //toggle V3 at end of freq3 countdown
     if (freq3 <= 0){
      digitalWrite(V3, !digitalRead(V3));
      freq3 = freq3save;
   }
                              //load random pulse Width into PCM V4

  //   ledcWrite(pcm_Channel0, random(2, 4093) );  //caused crash with timer interrupt?
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void AY_SynthLoad(int index){   // Load the software synth from the 11 AY_Synth registers

  freq1save = (AYArray[index][0] | ((AYArray[index][1] & 0xF) << 8 )) >> 2;  // V1 Frequency set
  freq2save = (AYArray[index][2] | ((AYArray[index][3] & 0xF) << 8 )) >> 2;  // V2 Frequency set
  freq3save = (AYArray[index][4] | ((AYArray[index][5] & 0xF) << 8 )) >> 2;  // V3 Frequency set

  if ( AYArray[index][6] != AYNoiseFreq) {  //V4 Noise Frequency set
    AYNoiseFreq = AYArray[index][6] ;
    ledcSetup(pcm_Channel0, AYNoiseFreq + 10, pcm_resolution);
  }

  if (bitRead(AYArray[index][7], 0)) { myAYDAC1.analogWrite(0); }  // V1 Amp set
  else { myAYDAC1.analogWrite(AY_Volume[ AYArray[index][8] & B1111 ]); }

  if (bitRead(AYArray[index][7], 1)) { myAYDAC2.analogWrite(0); }  // V2 Amp set
  else { myAYDAC2.analogWrite(AY_Volume[ AYArray[index][9] & B1111 ]); }

  if (bitRead(AYArray[index][7], 2)) { myAYDAC3.analogWrite(0); }  // V3 Amp set
  else { myAYDAC3.analogWrite(AY_Volume[ AYArray[index][10] & B1111 ]); }


  if ((!AYArray[index][7] & B00111000) == 0) { myAYDAC4.analogWrite(0); } // V4 Noise Amp set
  else {
     if( bitRead(AYArray[index][7], 3 )){AYNoiseAmpA = 0; } else{AYNoiseAmpA = AYArray[index][8] & B1111; }
     if( bitRead(AYArray[index][7], 4 )){AYNoiseAmpB = 0; } else{AYNoiseAmpB = AYArray[index][9] & B1111; }
     if( bitRead(AYArray[index][7], 5 )){AYNoiseAmpC = 0; } else{AYNoiseAmpC = AYArray[index][10] & B1111; }

     myAYDAC4.analogWrite( AY_Volume[ (AYNoiseAmpA | AYNoiseAmpB) | AYNoiseAmpC ] );
  }

} //end of AY_SynthLoad
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
void PrintHex8(uint8_t data, uint8_t length) // prints 8-bit data in hex with leading zeroes
{
        for (int i=0; i<length; i++) {
          Serial.print("0x");
          if (data<0x10) {Serial.print("0");}
          Serial.print(data,HEX);
        }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/*
 *                   AYFileLoad
 *
* Function to load AYArray from SPIFFS file
*
* AY Register Files are to be previously loaded into SPIFFS flash memory
* See sketch "ESP32AY_SPIFFS_ArcadeFile" for instruction how to do this
*
* The AY Register files used are from Daniel Tufvesson at
* http://www.waveguide.se/?article=ym-playback-on-the-ymz284
*
* These files are simple consecutive banks of 14 AY register bytes
* Normally one bank is loaded into the AY synth every 50ms.
* The last three registers in a bank are Envelope Bytes which are not normally used.
* The register file can then be compressed to one with 11 register bytes instead of 14
*
* The SPIFFS cannot be used simultaneously with the Timer Interrupts needed to create the voices
* Thus SPIFFS is used only to fill an SRAM 2-dimension array (AYArray[][]) with the file data
* SPIFFS file access is turned off before the timer interrupts are started
*
* Though the ESP32 is deemed to have lots of RAM space, experimentally, only array sizes less than about
* 110KB (10K banks of 11 registers) will compile without error.  Some of the AY files are too big
* and thus will be cut off to fit this smaller size when AYArray is loaded.
*
*/

void AYFileLoad(){
          // Initialize SPIFFS
         // SPIFFS access and TIMERS can't be run together without crashing
         // AY file is loaded into an array and then closed before anything else is started

  Serial.println("Mounting SPIFFS");
  if(!SPIFFS.begin(true)){
    Serial.println("Error while mounting SPIFFS");
    return;
  }

  AYFile = SPIFFS.open(AYFilePath, FILE_READ);
  Serial.print("");
  Serial.print("Loading File  ");
  Serial.print(AYFile.name());
  Serial.print("    File size: ");
  FileSize = AYFile.size();
  Serial.print(FileSize);
  Serial.print("   File Index size: ");
```

```
 AYIndexMax = int(FileSize/14);  // number of register banks,
 AYIndexMax =  min(AYIndexMax, INDEXMAX);  // number of AY banks, experimental compile maximum is 10k
 Serial.println(AYIndexMax);


if(AYFile.available()){

   for(int x=0; x<AYIndexMax; x++){  //Register Bank index

   for (int y=0; y<11; y++){         // load one Bank of AY registers
         AYArray[x][y] = AYFile.read();
   }  //End y

      AYFile.read(); AYFile.read();  AYFile.read();  //Toss the 3 unused Envelope bytes

   }  //End x
   }  //End if available


 AYFile.close();   //close AY file
 delay(2000);
 Serial.println("Finished Loading -- File Closed");

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~  Print Out File Registers  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/*
 //  11 AY File register functions
   Serial.print("A_FreqL A_FreqH B_FreqL B_FreqH C_FreqL C_FreqH ");
   Serial.print("N_Freq Enables  A_Amp   B_AMP   C_AMP  ");
   Serial.println("");
   Serial.println("");

 //~~~~~~~~~~~~~~~~Print 11 AY registers per line~~~~~~~~~~~~~~~~~~~~~~~

 for(int x=0; x<AYIndexMax; x++){     //Register Bank index
     Serial.print("{ ");             // print bank start

   for (int y=0; y<11; y++){          // load one Bank of AY registers

         PrintHex8(AYArray[x][y], 1); //print 13 hex values with cammas
         Serial.print(",   ");

   }  //End y
       Serial.print(" }, ");        // print bank end and comma
       Serial.println(" ");          // new line for new bank
   }  //End x
*/

}  //End function AYFileLoad
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```