audio mixing using MAX98357A

audio mixing from SPIFFS:

the following is a guide on how to stream a mixed audio using the MAX98357A amplifier and ESP32's SPIFFS. for this tutorial, we used the Arduino IDE.

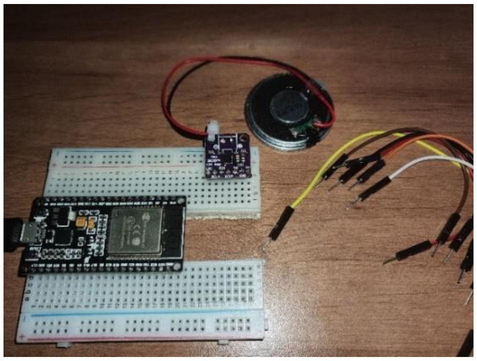
assumptions:

- you have downloaded the Arduino IDE.
- you have configured the IDE to work with the "DOIT ESP32 DIVKIT V1" board.
- you know how to upload files to ESP32's SPIFFS.

a guide to all the steps above can be found in the "bank of knowledge".

needed material:

- ESP32 microcontroller
- MAX98357A amplifier
- breadboard
- WiFi connection
- 6 wires
- a speaker that works with a 3[WATT]/4Ω



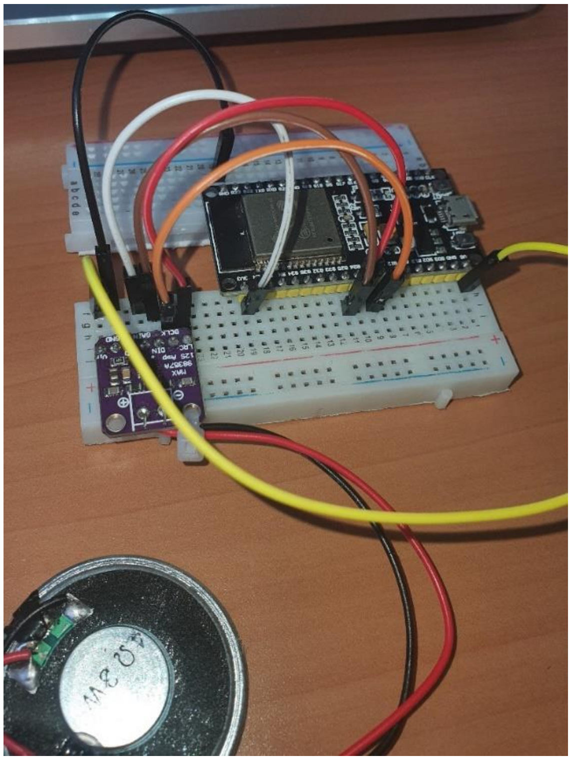
NOTE: the mixed sound's quality is not good because:

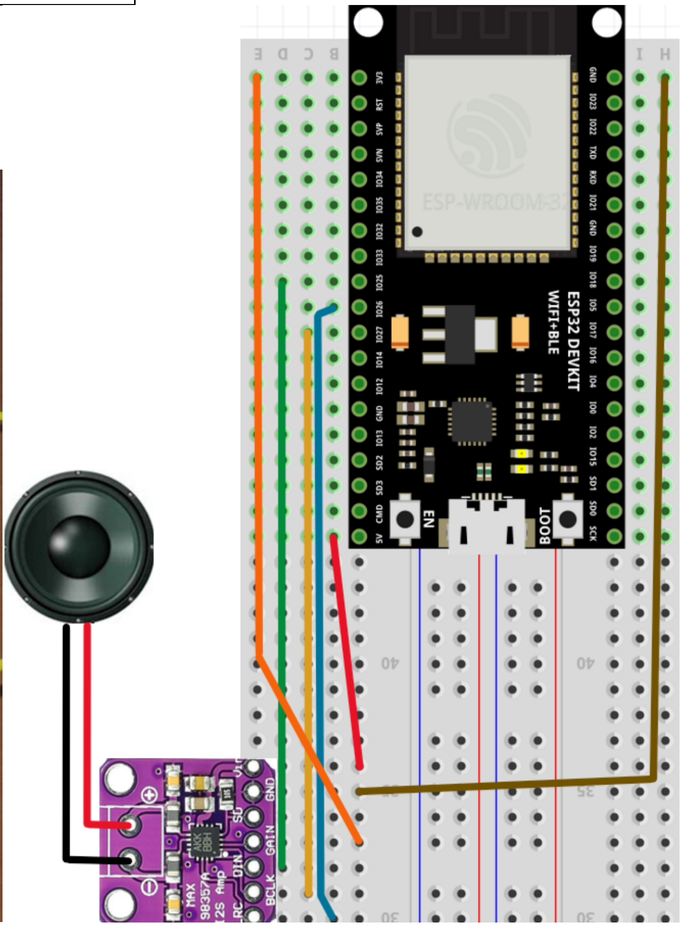
- SPIFFS memory size is small, so the audio files' sizes to be used will be short (2-3 seconds each).
- we're using only one module of the MAX98357A .
  for a much better result, use an SD card for larger audio files, and 2 modules of the MAX98357A for stereo mode. a guide on how to do so is provided after this one.
- the bit-depth(encoding) to be used in our case should be 16-bits

step 1: upload the audio files "wav1_16.wav" and "wav2_16.wav" to your esp32's SPIFFS.

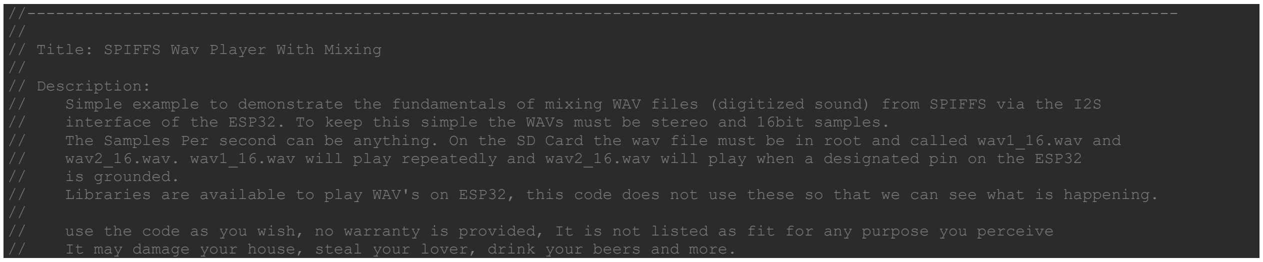
step 2: setup the wiring as follows:

MAX98357A	ESP32
Vin (2.5V-5.5V)	Vcc (preferably 3.3V but can be 5V)
GND	GND
BCK or BCLK	Pin 27 (G27)
DIN	Pin 25 (G25)
LRC	Pin 26 (G26)
GAIN	3.3V
SD	-

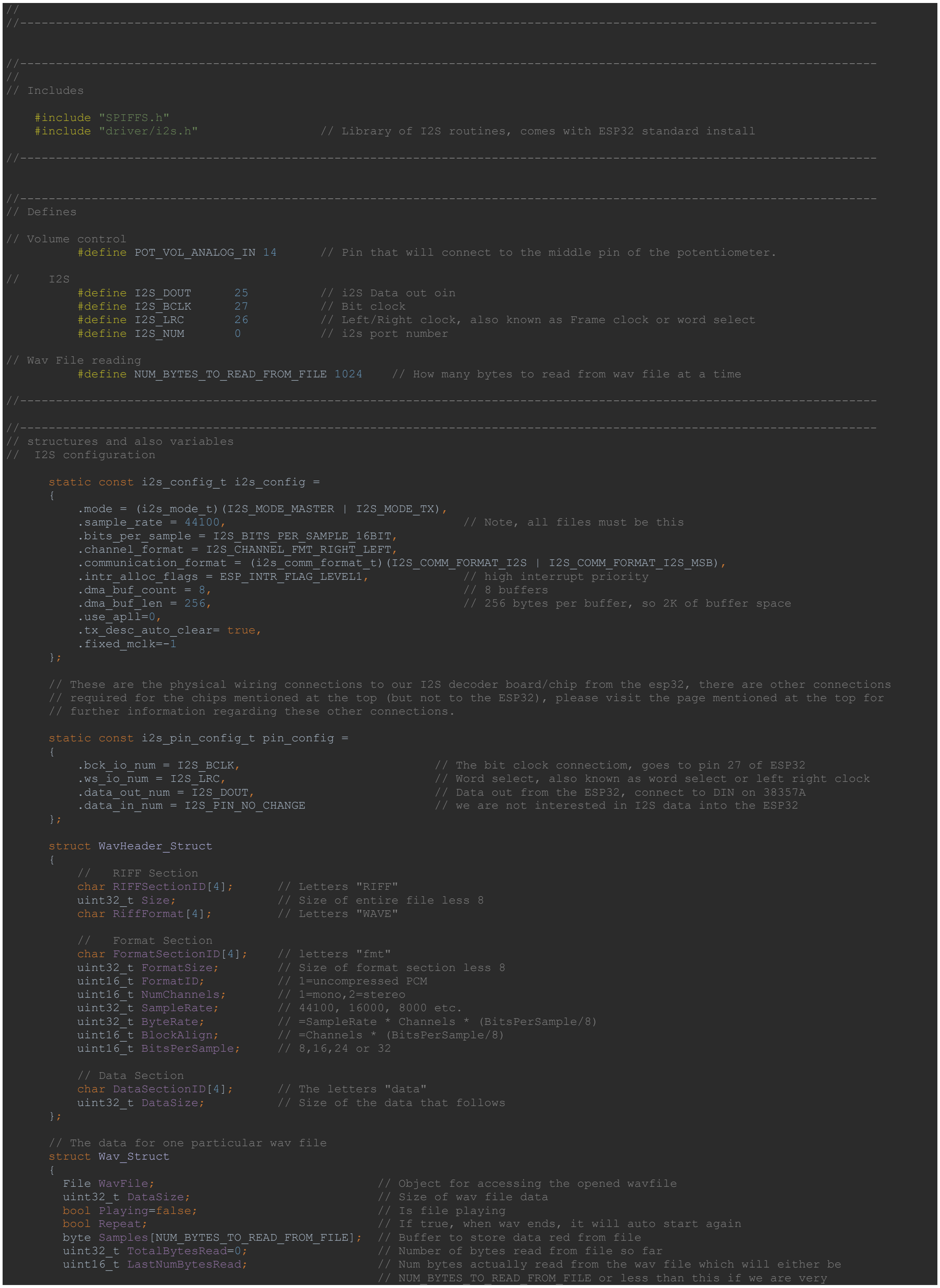




step 3: copy and paste the following code (the code is also provided in file of its own):



```
//----------------------------------------------------------------------------------------------------------------------
//
// Title: SPIFFS Wav Player With Mixing
//
// Description:
//    Simple example to demonstrate the fundamentals of mixing WAV files (digitized sound) from SPIFFS via the I2S
//    interface of the ESP32. To keep this simple the WAVs must be stereo and 16bit samples.
//    The Samples Per second can be anything. On the SD Card the wav file must be in root and called wav1_16.wav and
//    wav2_16.wav. wav1_16.wav will play repeatedly and wav2_16.wav will play when a designated pin on the ESP32
//    is grounded.
//    Libraries are available to play WAV's on ESP32, this code does not use these so that we can see what is happening.
//
//    use the code as you wish, no warranty is provided, It is not listed as fit for any purpose you perceive
//    It may damage your house, steal your lover, drink your beers and more.
```



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


//------------------------------------------------------------------------------------------------------------------------
//
// Includes

    #include "SPIFFS.h"
    #include "driver/i2s.h"                 // Library of I2S routines, comes with ESP32 standard install

//------------------------------------------------------------------------------------------------------------------------


//------------------------------------------------------------------------------------------------------------------------
// Defines

// Volume control
          #define POT_VOL_ANALOG_IN 14      // Pin that will connect to the middle pin of the potentiometer.

//    I2S
          #define I2S_DOUT      25          // i2S Data out oin
          #define I2S_BCLK      27          // Bit clock
          #define I2S_LRC       26          // Left/Right clock, also known as Frame clock or word select
          #define I2S_NUM       0           // i2s port number

// Wav File reading
          #define NUM_BYTES_TO_READ_FROM_FILE 1024    // How many bytes to read from wav file at a time

//------------------------------------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------------------------------------
// structures and also variables
//  I2S configuration

      static const i2s_config_t i2s_config =
      {
          .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_TX),
          .sample_rate = 44100,                                 // Note, all files must be this
          .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,
          .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
          .communication_format = (i2s_comm_format_t)(I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB),
          .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,             // high interrupt priority
          .dma_buf_count = 8,                                   // 8 buffers
          .dma_buf_len = 256,                                   // 256 bytes per buffer, so 2K of buffer space
          .use_apll=0,
          .tx_desc_auto_clear= true,
          .fixed_mclk=-1
      };

      // These are the physical wiring connections to our I2S decoder board/chip from the esp32, there are other connections
      // required for the chips mentioned at the top (but not to the ESP32), please visit the page mentioned at the top for
      // further information regarding these other connections.

      static const i2s_pin_config_t pin_config =
      {
          .bck_io_num = I2S_BCLK,                           // The bit clock connectiom, goes to pin 27 of ESP32
          .ws_io_num = I2S_LRC,                             // Word select, also known as word select or left right clock
          .data_out_num = I2S_DOUT,                         // Data out from the ESP32, connect to DIN on 38357A
          .data_in_num = I2S_PIN_NO_CHANGE                  // we are not interested in I2S data into the ESP32
      };

      struct WavHeader_Struct
      {
          //   RIFF Section
          char RIFFSectionID[4];      // Letters "RIFF"
          uint32_t Size;              // Size of entire file less 8
          char RiffFormat[4];         // Letters "WAVE"

          //   Format Section
          char FormatSectionID[4];    // letters "fmt"
          uint32_t FormatSize;        // Size of format section less 8
          uint16_t FormatID;          // 1=uncompressed PCM
          uint16_t NumChannels;       // 1=mono,2=stereo
          uint32_t SampleRate;        // 44100, 16000, 8000 etc.
          uint32_t ByteRate;          // =SampleRate * Channels * (BitsPerSample/8)
          uint16_t BlockAlign;        // =Channels * (BitsPerSample/8)
          uint16_t BitsPerSample;     // 8,16,24 or 32

          // Data Section
          char DataSectionID[4];      // The letters "data"
          uint32_t DataSize;          // Size of the data that follows
      };

      // The data for one particular wav file
      struct Wav_Struct
      {
        File WavFile;                               // Object for accessing the opened wavfile
        uint32_t DataSize;                          // Size of wav file data
        bool Playing=false;                         // Is file playing
        bool Repeat;                                // If true, when wav ends, it will auto start again
        byte Samples[NUM_BYTES_TO_READ_FROM_FILE];  // Buffer to store data red from file
        uint32_t TotalBytesRead=0;                  // Number of bytes read from file so far
        uint16_t LastNumBytesRead;                  // Num bytes actually read from the wav file which will either be
                                                    // NUM_BYTES_TO_READ_FROM_FILE or less than this if we are very
                                                    // near the end of the file. i.e. we can't read beyond the file.

      };
//------------------------------------------------------------------------------------------------------------------------

//  Global Variables/objects

    static const i2s_port_t i2s_num = I2S_NUM_0;  // i2s port number
    Wav_Struct Wav1;                              // Main Wave to play
    Wav_Struct Wav2;                              // Secondary "short" wav
    float Volume;                              // Volume

//------------------------------------------------------------------------------------------------------------------------


void setup() {
    Serial.begin(115200);                               // Used for info/debug
     // Mount the SPIFFS file system
    if (!SPIFFS.begin(true)) {
      Serial.println("Failed to mount file system");
      return;
    }
    i2s_driver_install(i2s_num, &i2s_config, 0, NULL);
    i2s_set_pin(i2s_num, &pin_config);
    if(InitWavFiles()==false)
      while(true);                                      // If a problem terminate program
    Wav1.Repeat=true;                                   // Wav1 will auto repeat
    Wav1.Playing=true;                                  // We set wav1 to play comtinuously
    Wav2.Repeat=true;                                   // Wav2 will auto repeat
    Wav2.Playing=true;                                  // We set wav2 to play comtinuously
}


void loop()
{
    PlayWavs();                                         // Have to keep calling this to keep the wav file playing
    // Your normal code to do your task can go here
}

void PlayWavs()
{
  static bool ReadingFile=true;                       // True if reading files from SD. false if filling I2S buffer
  static byte Samples[NUM_BYTES_TO_READ_FROM_FILE];   // Memory allocated to store the data read in from the wav files
  static uint16_t BytesReadFromFile;                  // Max Num bytes actually read from the wav files which will either be
                                                      // NUM_BYTES_TO_READ_FROM_FILE or less than this if we are very
                                                      // near the end of all files.

  Volume=float(analogRead(POT_VOL_ANALOG_IN))/2047;   // You possibly don't need to sample volume this often, perhaps every 1/10
sec would be fine
  if(ReadingFile)                                     // Read next chunk of data in from files
  {
    ReadFiles();                                      // Read data into the wavs own buffers
    BytesReadFromFile=MixWavs(Samples);                       // Mix the samples together and store in the samples buffer
    ReadingFile=false;                                // Switch to sending the buffer to the I2S
  }
  else
   ReadingFile=FillI2SBuffer(Samples,BytesReadFromFile);   // We keep calling this routine until it returns true, at which point
                                                            // this will swap us back to Reading the next block of data from the
file.
                                                            // Reading true means it has managed to push all the data to the I2S
                                                            // Handler, false means there still more to do and you should call
this
                                                            // routine again and again until it returns true.
}

uint16_t MixWavs(byte* Samples)
{
  // Mix all playing wavs together, returns the max bytes that are in the buffer, usually this would be the full buffer but
  // in rare cases wavs may be close to the end of the file and thus not fill the entire buffer

  uint16_t Wav1Idx,Wav2Idx;                           // Index into the wavs sample data
  int16_t Sample;                                     // The mixed sample
  uint16_t i;                                         // index into main samples buffer
  uint16_t MaxBytesInBuffer;                          // Max bytes of data in buffer, most of time buffer will be full

  Wav1Idx=0;
  Wav2Idx=0;
  while((Wav1Idx<Wav1.LastNumBytesRead)|(Wav2Idx<Wav2.LastNumBytesRead))
  {
    Sample=0;
    if(Wav1.Playing)
      Sample=*((int16_t *)(Wav1.Samples+Wav1Idx));
    if(Wav2.Playing)
      Sample+=*((int16_t *)(Wav2.Samples+Wav2Idx));    // This does the actual mix, just add togther
    *((int16_t *)(Samples+i))=Sample;
    Wav1Idx+=2;
    Wav2Idx+=2;
    i+=2;
  }
  if(Wav1.LastNumBytesRead>Wav2.LastNumBytesRead)
    MaxBytesInBuffer=Wav1.LastNumBytesRead;
  else
    MaxBytesInBuffer=Wav2.LastNumBytesRead;

  // We now alter the data according to the volume control
  for(i=0;i<MaxBytesInBuffer;i+=2)  // We step 2 bytes at a time as we're using 16bits per channel
      *((int16_t *)(Samples+i))=(*((int16_t *)(Samples+i)))*(0.1);
    //*((int16_t *)(Samples+i))=(*((int16_t *)(Samples+i)))*Volume;

  return MaxBytesInBuffer;
}


bool InitWavFiles()
{
  // initialise wav files
  if(LoadWavFileHeader("/wav1_16.wav",&Wav1))
    return LoadWavFileHeader("/wav2_16.wav",&Wav2);              // only bother trying to load this if first loads ok
  else
    return false;
}


void ReadFiles()
{
  // Read in all files samples into their buffers
  if(Wav1.Playing)
    ReadFile(&Wav1);
  if(Wav2.Playing)
    ReadFile(&Wav2);
}

void ReadFile(Wav_Struct *Wav)
{
    uint16_t i;                                         // loop counter
    int16_t SignedSample;                               // Single Signed Sample
    float Volume;

    if(Wav->TotalBytesRead+NUM_BYTES_TO_READ_FROM_FILE>Wav->DataSize)    // If next read will go past the end then adjust the
      Wav->LastNumBytesRead=Wav->DataSize-Wav->TotalBytesRead;                     // amount to read to whatever is remaining to
read
    else
      Wav->LastNumBytesRead=NUM_BYTES_TO_READ_FROM_FILE;                           // Default to max to read

    Wav->WavFile.read(Wav->Samples,Wav->LastNumBytesRead);                   // Read in the bytes from the file
    Wav->TotalBytesRead+=Wav->LastNumBytesRead;                         // Update the total bytes red in so far

    if(Wav->TotalBytesRead>=Wav->DataSize)              // Have we read in all the data?
    {
      if(Wav->Repeat)
      {
        Wav->WavFile.seek(44);                                  // Reset to start of wav data
        Wav->TotalBytesRead=0;                         // Clear to no bytes read in so far
      }
      else
        Wav->Playing=false;                                     // Flag that wav has completed
    }
}

bool LoadWavFileHeader(String FileName, Wav_Struct* Wav)
{
  // Load wav file, if all goes ok returns true else false
  WavHeader_Struct WavHeader;

  Wav->WavFile = SPIFFS.open(FileName);                  // Open the wav file
  if(Wav->WavFile==false)
  {
    Serial.print("Could not open :");
    Serial.println(FileName);
    return false;
  }
  else
  {
    Wav->WavFile.read((byte *) &WavHeader,44);        // Read in the WAV header, which is first 44 bytes of the file.
                                                      // We have to typecast to bytes for the "read" function
    if(ValidWavData(&WavHeader))
    {
      DumpWAVHeader(&WavHeader);                      // Dump the header data to serial, optional!
      Serial.println();
      Wav->DataSize=WavHeader.DataSize;                    // Copy the data size into our wav structure
      return true;
    }
    else
      return false;
  }
}



bool FillI2SBuffer(byte* Samples,uint16_t BytesInBuffer)
{
    // Writes bytes to buffer, returns true if all bytes sent else false, keeps track itself of how many left
    // to write, so just keep calling this routine until returns true to know they've all been written, then
    // you can re-fill the buffer

    size_t BytesWritten;                        // Returned by the I2S write routine,
    static uint16_t BufferIdx=0;                // Current pos of buffer to output next
    uint8_t* DataPtr;                           // Point to next data to send to I2S
    uint16_t BytesToSend;                       // Number of bytes to send to I2S
    // To make the code eaier to understand I'm using to variables to some calculations, normally I'd write this calcs
    // directly into the line of code where they belong, but this make it easier to understand what's happening

    DataPtr=Samples+BufferIdx;                                // Set address to next byte in buffer to send out
    BytesToSend=BytesInBuffer-BufferIdx;                      // This is amount to send (total less what we've already sent)
    i2s_write(i2s_num,DataPtr,BytesToSend,&BytesWritten,1);  // Send the bytes, wait 1 RTOS tick to complete
    BufferIdx+=BytesWritten;                                  // increasue by number of bytes actually written

    if(BufferIdx>=BytesInBuffer)
    {
      // sent out all bytes in buffer, reset and return true to indicate this
      BufferIdx=0;
      return true;
    }
    else
      return false;       // Still more data to send to I2S so return false to indicate this
}

bool ValidWavData(WavHeader_Struct* Wav)
{

  if(memcmp(Wav->RIFFSectionID,"RIFF",4)!=0)
  {
    Serial.print("Invalid data - Not RIFF format");
    return false;
  }
  if(memcmp(Wav->RiffFormat,"WAVE",4)!=0)
  {
    Serial.print("Invalid data - Not Wave file");
    return false;
  }
  if(memcmp(Wav->FormatSectionID,"fmt",3)!=0)
  {
    Serial.print("Invalid data - No format section found");
    return false;
  }
  if(memcmp(Wav->DataSectionID,"data",4)!=0)
  {
    Serial.print("Invalid data - data section not found");
    return false;
  }
  if(Wav->FormatID!=1)
  {
    Serial.print("Invalid data - format Id must be 1");
    return false;
  }
  if(Wav->FormatSize!=16)
  {
    Serial.print("Invalid data - format section size must be 16.");
    return false;
  }
  if((Wav->NumChannels!=1)&(Wav->NumChannels!=2))
  {
    Serial.print("Invalid data - only mono or stereo permitted.");
    return false;
  }
  if(Wav->SampleRate>48000)
  {
    Serial.print("Invalid data - Sample rate cannot be greater than 48000");
    return false;
  }
  if((Wav->BitsPerSample!=8)& (Wav->BitsPerSample!=16))
  {
    Serial.print("Invalid data - Only 8 or 16 bits per sample permitted.");
    return false;
  }
  return true;
}


void DumpWAVHeader(WavHeader_Struct* Wav)
{
  if(memcmp(Wav->RIFFSectionID,"RIFF",4)!=0)
  {
    Serial.print("Not a RIFF format file - ");
    PrintData(Wav->RIFFSectionID,4);
    return;
  }
  if(memcmp(Wav->RiffFormat,"WAVE",4)!=0)
  {
    Serial.print("Not a WAVE file - ");
    PrintData(Wav->RiffFormat,4);
    return;
  }
  if(memcmp(Wav->FormatSectionID,"fmt",3)!=0)
  {
    Serial.print("fmt ID not present - ");
    PrintData(Wav->FormatSectionID,3);
    return;
  }
  if(memcmp(Wav->DataSectionID,"data",4)!=0)
  {
    Serial.print("data ID not present - ");
    PrintData(Wav->DataSectionID,4);
    return;
  }
  // All looks good, dump the data
  Serial.print("Total size :");Serial.println(Wav->Size);
  Serial.print("Format section size :");Serial.println(Wav->FormatSize);
  Serial.print("Wave format :");Serial.println(Wav->FormatID);
  Serial.print("Channels :");Serial.println(Wav->NumChannels);
  Serial.print("Sample Rate :");Serial.println(Wav->SampleRate);
  Serial.print("Byte Rate :");Serial.println(Wav->ByteRate);
  Serial.print("Block Align :");Serial.println(Wav->BlockAlign);
  Serial.print("Bits Per Sample :");Serial.println(Wav->BitsPerSample);
  Serial.print("Data Size :");Serial.println(Wav->DataSize);
}

void PrintData(const char* Data,uint8_t NumBytes)
{
    for(uint8_t i=0;i<NumBytes;i++)
      Serial.print(Data[i]);
      Serial.println();
}
```

Important Notes:

- make sure that the defined Pins match the wiring!
- in the function “InitWavFiles()”, make sure the name of the audio files matches the ones you want to you use from your SPIFFS.
- the “MixWavs” function is the star function here, that’s where the actual mixing occurs!
- if you look at the following part in that function :

```
      if(Wav1.Playing)
        Sample=*((int16_t *)(Wav1.Samples+Wav1Idx));
       if(Wav2.Playing)
        Sample+=*((int16_t *)(Wav2.Samples+Wav2Idx));
```

  you can comment the second\fourth lines to hear only one of the audio files playing.
- in the same function “MixWav”, right before exiting, there is a value that is multiplied by the variable “VOLUME”, that’s the variable that is responsible of the mixed audio’s volume, change it accordingly, recommended values are “0 < VOLUME < 1”

step 4: connect the ESP32 to your computer, compile and run the code. you might need to press on the “reset” button on your ESP32.

step 5: enjoy the mixed sound 😊

audio mixing from SD card & 2 modules of the amplifier:

the following is a guide on how to stream a mixed audio using 2 MAX98357A amplifiers, SD Card and ESP32. for this tutorial, we used the Arduino IDE.

assumptions:

- you have downloaded the Arduino IDE.
- you have configured the IDE to work with the "DOIT ESP32 DIVKIT V1" board.

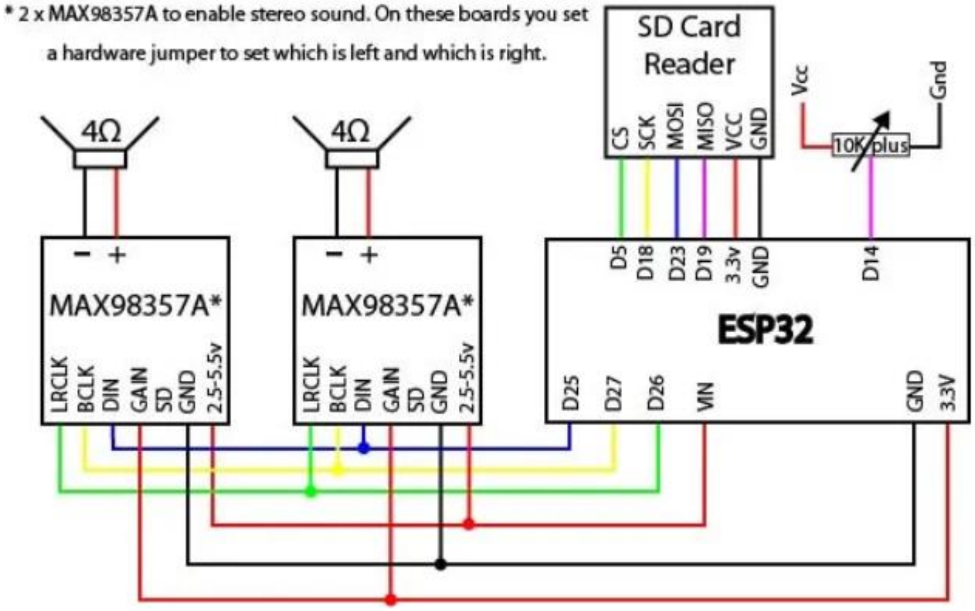
a guide to all the steps above can be found in the "bank of knowledge".

needed material:

- ESP32 microcontroller
- MAX98357A amplifier x 2
- breadboard
- WiFi connection
- wires
- a speaker that works with a 3[WATT]/4Ω x 2
- SD Card
- SD Card adapter

step 1: upload the audio files "wav1_16.wav" and "wav2_16.wav" to your SD Card.

step 2: setup the wiring as follows:



* 2 x MAX98357A to enable stereo sound. On these boards you set a hardware jumper to set which is left and which is right.

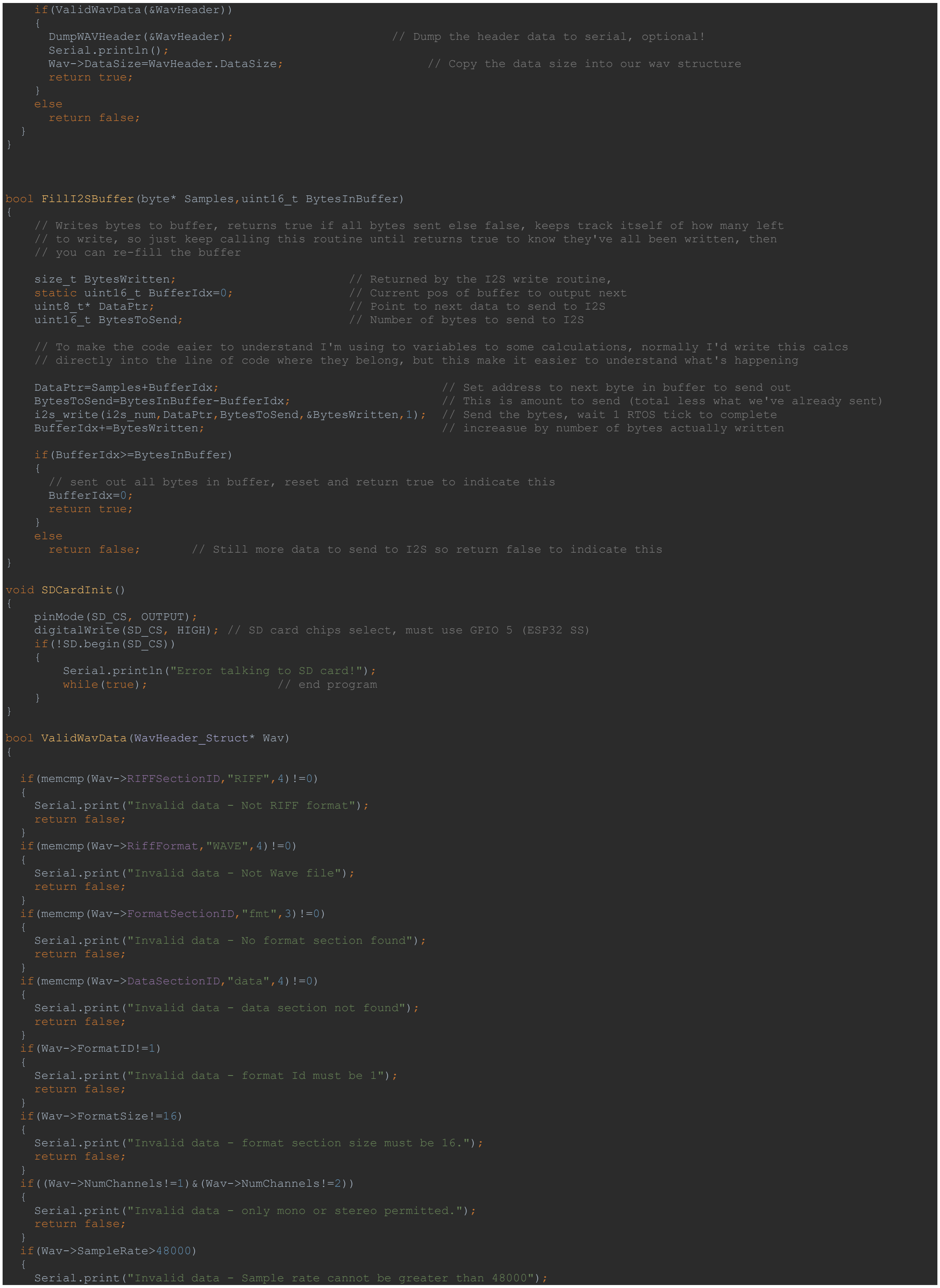
NOTE: in order to redirect the sound to the right\left channel, check out the "playing audio in mono or stereo using MAX98357A or PCM5102A" guide.

step 3: copy and paste the following code (the code is also provided in file of its own).

NOTE:

- the code is very similar to the one above, but modified to read files from the SD Card instead of SPIFFS.
- the “important notes” stated above also apply here.

```
//-----------------------------------------------------------------------------------------------------------------------
//
// Title: SD Card Wav Player With Mixing
//
// Description:
//    Simple example to demonstrate the fundamentals of mixing WAV files (digitized sound) from SPIFFS via the I2S
//    interface of the ESP32. To keep this simple the WAVs must be stereo and 16bit samples.
//    The Samples Per second can be anything. On the SD Card the wav file must be in root and called wav1_16.wav and
//    wav2_16.wav. wav1_16.wav will play repeatedly and wav2_16.wav will play when a designated pin on the ESP32
//    is grounded.
//    Libraries are available to play WAV's on ESP32, this code does not use these so that we can see what is happening.
//
//    use the code as you wish, no warranty is provided, It is not listed as fit for any purpose you perceive
//    It may damage your house, steal your lover, drink your beers and more.
//
//-----------------------------------------------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------------------------------------------
//
// Includes

    #include "SD.h"                             // SD Card library, usually part of the standard install
    #include "driver/i2s.h"                     // Library of I2S routines, comes with ESP32 standard install

//-----------------------------------------------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------------------------------------------
// Defines

// Volume control
          #define POT_VOL_ANALOG_IN 14      // Pin that will connect to the middle pin of the potentiometer.

//    SD Card
          #define SD_CS           5          // SD Card chip select

//    I2S
          #define I2S_DOUT      25           // i2S Data out oin
          #define I2S_BCLK      27           // Bit clock
          #define I2S_LRC       26           // Left/Right clock, also known as Frame clock or word select
          #define I2S_NUM       0            // i2s port number

// Wav File reading
          #define NUM_BYTES_TO_READ_FROM_FILE 1024    // How many bytes to read from wav file at a time

//-----------------------------------------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------------------------------------
// structures and also variables
//  I2S configuration

      static const i2s_config_t i2s_config =
      {
          .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_TX),
          .sample_rate = 44100,                                 // Note, all files must be this
          .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,
          .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
          .communication_format = (i2s_comm_format_t)(I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB),
          .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,             // high interrupt priority
          .dma_buf_count = 8,                                   // 8 buffers
          .dma_buf_len = 256,                                   // 256 bytes per buffer, so 2K of buffer space
          .use_apll=0,
          .tx_desc_auto_clear= true,
          .fixed_mclk=-1
      };

      // These are the physical wiring connections to our I2S decoder board/chip from the esp32, there are other connections
      // required for the chips mentioned at the top (but not to the ESP32), please visit the page mentioned at the top for
      // further information regarding these other connections.

      static const i2s_pin_config_t pin_config =
      {
          .bck_io_num = I2S_BCLK,                           // The bit clock connectiom, goes to pin 27 of ESP32
          .ws_io_num = I2S_LRC,                             // Word select, also known as word select or left right clock
          .data_out_num = I2S_DOUT,                         // Data out from the ESP32, connect to DIN on 38357A
          .data_in_num = I2S_PIN_NO_CHANGE                  // we are not interested in I2S data into the ESP32
      };

      struct WavHeader_Struct
      {
          //   RIFF Section
          char RIFFSectionID[4];      // Letters "RIFF"
          uint32_t Size;              // Size of entire file less 8
          char RiffFormat[4];         // Letters "WAVE"

          //   Format Section
          char FormatSectionID[4];    // letters "fmt"
          uint32_t FormatSize;        // Size of format section less 8
          uint16_t FormatID;          // 1=uncompressed PCM
          uint16_t NumChannels;       // 1=mono,2=stereo
          uint32_t SampleRate;        // 44100, 16000, 8000 etc.
          uint32_t ByteRate;          // =SampleRate * Channels * (BitsPerSample/8)
          uint16_t BlockAlign;        // =Channels * (BitsPerSample/8)
          uint16_t BitsPerSample;     // 8,16,24 or 32

          // Data Section
          char DataSectionID[4];      // The letters "data"
          uint32_t DataSize;          // Size of the data that follows
      };

      // The data for one particular wav file
      struct Wav_Struct
      {
        File WavFile;                               // Object for accessing the opened wavfile
        uint32_t DataSize;                          // Size of wav file data
        bool Playing=false;                         // Is file playing
        bool Repeat;                                // If true, when wav ends, it will auto start again
        byte Samples[NUM_BYTES_TO_READ_FROM_FILE];  // Buffer to store data red from file
        uint32_t TotalBytesRead=0;                  // Number of bytes read from file so far
        uint16_t LastNumBytesRead;                  // Num bytes actually read from the wav file which will either be
                                                    // NUM_BYTES_TO_READ_FROM_FILE or less than this if we are very
                                                    // near the end of the file. i.e. we can't read beyond the file.

      };
//------------------------------------------------------------------------------------------------------------------------

//  Global Variables/objects

    static const i2s_port_t i2s_num = I2S_NUM_0;  // i2s port number
    Wav_Struct Wav1;                              // Main Wave to play
    Wav_Struct Wav2;                              // Secondary "short" wav
    float Volume;                              // Volume

//------------------------------------------------------------------------------------------------------------------------


void setup() {
    Serial.begin(115200);                               // Used for info/debug
    SDCardInit();
    i2s_driver_install(i2s_num, &i2s_config, 0, NULL);
    i2s_set_pin(i2s_num, &pin_config);
    if(InitWavFiles()==false)
      while(true);                                      // If a problem terminate program
    Wav1.Repeat=true;                                   // Wav1 will auto repeat
    Wav1.Playing=true;                                  // We set wav1 to play comtinuously
    Wav2.Repeat=true;                                   // Wav2 will auto repeat
    Wav2.Playing=true;                                  // We set wav2 to play comtinuously
}


void loop()
{
    PlayWavs();                                         // Have to keep calling this to keep the wav file playing
    // Your normal code to do your task can go here
}

void PlayWavs()
{
  static bool ReadingFile=true;                       // True if reading files from SD. false if filling I2S buffer
  static byte Samples[NUM_BYTES_TO_READ_FROM_FILE];   // Memory allocated to store the data read in from the wav files
  static uint16_t BytesReadFromFile;                  // Max Num bytes actually read from the wav files which will either be
                                                      // NUM_BYTES_TO_READ_FROM_FILE or less than this if we are very
                                                      // near the end of all files.

  Volume=float(analogRead(POT_VOL_ANALOG_IN))/2047;   // You possibly don't need to sample volume this often, perhaps every 1/10
sec would be fine
  if(ReadingFile)                                     // Read next chunk of data in from files
  {
    ReadFiles();                                      // Read data into the wavs own buffers
    BytesReadFromFile=MixWavs(Samples);                       // Mix the samples together and store in the samples buffer
    ReadingFile=false;                                // Switch to sending the buffer to the I2S
  }
  else
   ReadingFile=FillI2SBuffer(Samples,BytesReadFromFile);   // We keep calling this routine until it returns true, at which point
                                                            // this will swap us back to Reading the next block of data from the
file.
                                                            // Reading true means it has managed to push all the data to the I2S
                                                            // Handler, false means there still more to do and you should call
this
                                                            // routine again and again until it returns true.
}

uint16_t MixWavs(byte* Samples)
{
  // Mix all playing wavs together, returns the max bytes that are in the buffer, usually this would be the full buffer but
  // in rare cases wavs may be close to the end of the file and thus not fill the entire buffer
  uint16_t Wav1Idx,Wav2Idx;                           // Index into the wavs sample data
  int16_t Sample;                                     // The mixed sample
  uint16_t i;                                         // index into main samples buffer
  uint16_t MaxBytesInBuffer;                          // Max bytes of data in buffer, most of time buffer will be full

  Wav1Idx=0;
  Wav2Idx=0;
  while((Wav1Idx<Wav1.LastNumBytesRead)|(Wav2Idx<Wav2.LastNumBytesRead))
  {
    Sample=0;
    if(Wav1.Playing)
      Sample=*((int16_t *)(Wav1.Samples+Wav1Idx));
    if(Wav2.Playing)
      Sample+=*((int16_t *)(Wav2.Samples+Wav2Idx));    // This does the actual mix, just add togther
    *((int16_t *)(Samples+i))=Sample;
    Wav1Idx+=2;
    Wav2Idx+=2;
    i+=2;
  }
  if(Wav1.LastNumBytesRead>Wav2.LastNumBytesRead)
    MaxBytesInBuffer=Wav1.LastNumBytesRead;
  else
    MaxBytesInBuffer=Wav2.LastNumBytesRead;

  // We now alter the data according to the volume control
  for(i=0;i<MaxBytesInBuffer;i+=2)  // We step 2 bytes at a time as we're using 16bits per channel
    *((int16_t *)(Samples+i))=(*((int16_t *)(Samples+i)))*Volume;

  return MaxBytesInBuffer;
}


bool InitWavFiles()
{
  // initialise wav files
  if(LoadWavFileHeader("/wav1_16.wav",&Wav1))
    return LoadWavFileHeader("/wav2_16.wav",&Wav2);             // only bother trying to load this if first loads ok
  else
    return false;
}


void ReadFiles()
{
  // Read in all files samples into their buffers
  if(Wav1.Playing)
    ReadFile(&Wav1);
  if(Wav2.Playing)
    ReadFile(&Wav2);
}

void ReadFile(Wav_Struct *Wav)
{
    uint16_t i;                                         // loop counter
    int16_t SignedSample;                               // Single Signed Sample
    float Volume;

    if(Wav->TotalBytesRead+NUM_BYTES_TO_READ_FROM_FILE>Wav->DataSize)   // If next read will go past the end then adjust the
      Wav->LastNumBytesRead=Wav->DataSize-Wav->TotalBytesRead;                    // amount to read to whatever is remaining to
read
    else
      Wav->LastNumBytesRead=NUM_BYTES_TO_READ_FROM_FILE;                          // Default to max to read

    Wav->WavFile.read(Wav->Samples,Wav->LastNumBytesRead);                  // Read in the bytes from the file
    Wav->TotalBytesRead+=Wav->LastNumBytesRead;                        // Update the total bytes red in so far

    if(Wav->TotalBytesRead>=Wav->DataSize)              // Have we read in all the data?
    {
      if(Wav->Repeat)
      {
        Wav->WavFile.seek(44);                                 // Reset to start of wav data
        Wav->TotalBytesRead=0;                         // Clear to no bytes read in so far
      }
      else
        Wav->Playing=false;                                    // Flag that wav has completed
    }
}

bool LoadWavFileHeader(String FileName, Wav_Struct* Wav)
{
  // Load wav file, if all goes ok returns true else false
  WavHeader_Struct WavHeader;

  Wav->WavFile = SD.open(FileName);                  // Open the wav file
  if(Wav->WavFile==false)
  {
    Serial.print("Could not open :");
    Serial.println(FileName);
    return false;
  }
  else
  {
    Wav->WavFile.read((byte *) &WavHeader,44);        // Read in the WAV header, which is first 44 bytes of the file.
                                                      // We have to typecast to bytes for the "read" function
```



```
    if(ValidWavData(&WavHeader))
    {
      DumpWAVHeader(&WavHeader);                      // Dump the header data to serial, optional!
      Serial.println();
      Wav->DataSize=WavHeader.DataSize;                    // Copy the data size into our wav structure
      return true;
    }
    else
      return false;
  }
}



bool FillI2SBuffer(byte* Samples,uint16_t BytesInBuffer)
{
    // Writes bytes to buffer, returns true if all bytes sent else false, keeps track itself of how many left
    // to write, so just keep calling this routine until returns true to know they've all been written, then
    // you can re-fill the buffer

    size_t BytesWritten;                        // Returned by the I2S write routine,
    static uint16_t BufferIdx=0;                // Current pos of buffer to output next
    uint8_t* DataPtr;                           // Point to next data to send to I2S
    uint16_t BytesToSend;                       // Number of bytes to send to I2S

    // To make the code eaier to understand I'm using to variables to some calculations, normally I'd write this calcs
    // directly into the line of code where they belong, but this make it easier to understand what's happening

    DataPtr=Samples+BufferIdx;                               // Set address to next byte in buffer to send out
    BytesToSend=BytesInBuffer-BufferIdx;                     // This is amount to send (total less what we've already sent)
    i2s_write(i2s_num,DataPtr,BytesToSend,&BytesWritten,1);  // Send the bytes, wait 1 RTOS tick to complete
    BufferIdx+=BytesWritten;                                 // increasue by number of bytes actually written

    if(BufferIdx>=BytesInBuffer)
    {
      // sent out all bytes in buffer, reset and return true to indicate this
      BufferIdx=0;
      return true;
    }
    else
      return false;       // Still more data to send to I2S so return false to indicate this
}

void SDCardInit()
{
    pinMode(SD_CS, OUTPUT);
    digitalWrite(SD_CS, HIGH); // SD card chips select, must use GPIO 5 (ESP32 SS)
    if(!SD.begin(SD_CS))
    {
        Serial.println("Error talking to SD card!");
        while(true);                  // end program
    }
}

bool ValidWavData(WavHeader_Struct* Wav)
{

  if(memcmp(Wav->RIFFSectionID,"RIFF",4)!=0)
  {
    Serial.print("Invalid data - Not RIFF format");
    return false;
  }
  if(memcmp(Wav->RiffFormat,"WAVE",4)!=0)
  {
    Serial.print("Invalid data - Not Wave file");
    return false;
  }
  if(memcmp(Wav->FormatSectionID,"fmt",3)!=0)
  {
    Serial.print("Invalid data - No format section found");
    return false;
  }
  if(memcmp(Wav->DataSectionID,"data",4)!=0)
  {
    Serial.print("Invalid data - data section not found");
    return false;
  }
  if(Wav->FormatID!=1)
  {
    Serial.print("Invalid data - format Id must be 1");
    return false;
  }
  if(Wav->FormatSize!=16)
  {
    Serial.print("Invalid data - format section size must be 16.");
    return false;
  }
  if((Wav->NumChannels!=1)&(Wav->NumChannels!=2))
  {
    Serial.print("Invalid data - only mono or stereo permitted.");
    return false;
  }
  if(Wav->SampleRate>48000)
  {
    Serial.print("Invalid data - Sample rate cannot be greater than 48000");
    return false;
  }
  if((Wav->BitsPerSample!=8)& (Wav->BitsPerSample!=16))
  {
    Serial.print("Invalid data - Only 8 or 16 bits per sample permitted.");
    return false;
  }
  return true;
}


void DumpWAVHeader(WavHeader_Struct* Wav)
{
  if(memcmp(Wav->RIFFSectionID,"RIFF",4)!=0)
  {
    Serial.print("Not a RIFF format file - ");
    PrintData(Wav->RIFFSectionID,4);
    return;
  }
  if(memcmp(Wav->RiffFormat,"WAVE",4)!=0)
  {
    Serial.print("Not a WAVE file - ");
    PrintData(Wav->RiffFormat,4);
    return;
  }
  if(memcmp(Wav->FormatSectionID,"fmt",3)!=0)
  {
    Serial.print("fmt ID not present - ");
    PrintData(Wav->FormatSectionID,3);
    return;
  }
  if(memcmp(Wav->DataSectionID,"data",4)!=0)
  {
    Serial.print("data ID not present - ");
    PrintData(Wav->DataSectionID,4);
    return;
  }
  // All looks good, dump the data
  Serial.print("Total size :");Serial.println(Wav->Size);
  Serial.print("Format section size :");Serial.println(Wav->FormatSize);
  Serial.print("Wave format :");Serial.println(Wav->FormatID);
  Serial.print("Channels :");Serial.println(Wav->NumChannels);
  Serial.print("Sample Rate :");Serial.println(Wav->SampleRate);
  Serial.print("Byte Rate :");Serial.println(Wav->ByteRate);
  Serial.print("Block Align :");Serial.println(Wav->BlockAlign);
  Serial.print("Bits Per Sample :");Serial.println(Wav->BitsPerSample);
  Serial.print("Data Size :");Serial.println(Wav->DataSize);
}

void PrintData(const char* Data,uint8_t NumBytes)
{
    for(uint8_t i=0;i<NumBytes;i++)
      Serial.print(Data[i]);
      Serial.println();
}
```

step 4: connect the ESP32 to your computer, compile and run the code. you might need to press on the "reset" button on your ESP32.

step 5: enjoy the mixed sound 😊

for a reference of this experiment, look up the following link here.