Universidade Federal de Santa Catarina

LINSE - Laboratório de Circuitos e Processamento de Sinais

Desenvolvimento de um sistema de aquisição de áudio via ESP32

João Paulo Vieira

1 Introdução

Neste documento é descrito o desenvolvimento de um sistema de aquisição de áudio via ESP32. Inicialmente, o áudio adquirido é salvo em um cartão de memória (SD) conectado ao ESP32. Na sequência, o ESP32 envia os arquivos de áudio para um servidor da web. Posteriormente, tais arquivos podem ser baixados por um PC para avaliação ou classificação via Machine Learning. A Figura 1.1 apresenta o diagrama de blocos desse sistema.

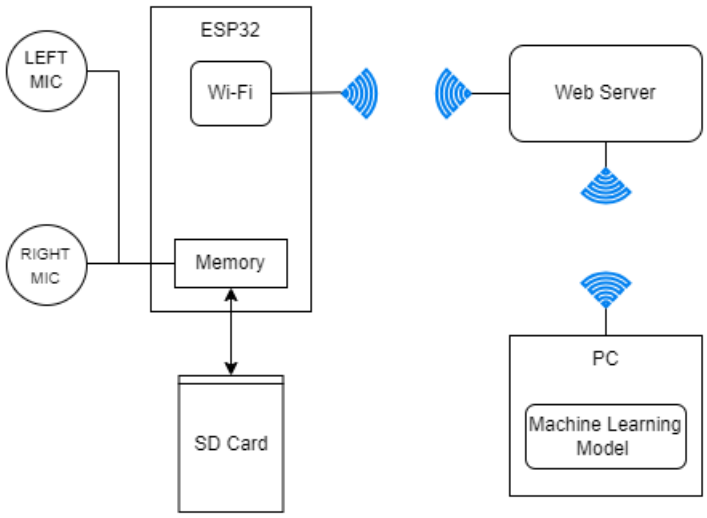

Figura 1.1: Diagrama de blocos do sistema.

2 Descrição dos blocos

A seguir, serão descritos os principais blocos que compõem o sistema: aquisição, Web Server e PC.

2.1 Aquisição

Os dispostivos que compõem o bloco de aquisição são: os microfones, o cartão SD, e a ESP32 (incluindo seus módulos WiFi e I2S). Deseja-se realizar a aquisição de forma periódica, com a identificação de data e hora em que as gravações ocorreram. Inicialmente, serão mostrados os dispositivos de hardware utilizados, e posteriormente, serão descritos os trechos de códigos para o ESP32 que implementam as funções do processo de aquisição, destacando as bibliotecas e configurações necessárias.

2.1.1 Hardware utilizado

2.1.1.1 Microfone INMP441

O INMP441 é um microfone do tipo MEMS, de baixa potência, alta perfomance, omnidirecional e saída digital. Ele consiste num sensor MEMS, um bloco para tratamento do sinal, um conversor analógico-digital, filtros anti-aliasing e uma interface I2S de 24 bits, permitindo fácil comunicação com microcontroladores. A Figura 2.1 exibe o diagrama de blocos do microfone.

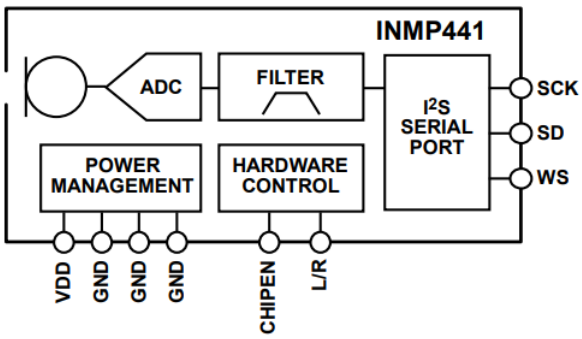

Figura 2.1: Diagrama de blocos do microfone INMP441.

De acordo com o datasheet do microfone, ele possui uma resposta em frequência plana numa faixa de 60 Hz - 15 kHz, além de uma SNR de 61dBa. Para esse projeto, optou-se por trabalhar numa frequência de amostragem de 16 kHz, que é possível ser escolhida através da configuração da interface I2S.

2.1.1.2 SD CARD

Para realizar a leitura de um cartão SD, utilizou-se o MH-SD Card Module, mostrado na Figura 2.2. O protocolo utilizado para a comunicação com o SD Card é o SPI.

Figura 2.2: Módulo MH-SD Card.

Esse módulo possui 8 pinos, sendo dois deles para o GND, um para alimentação de 3.3V e outro para 5V. Os pinos restantes cumprem a função de realizar a interface do SD card com o ESP32, sendo eles os pinos MISO, MOSI, SCK e CS.

2.1.2 Software

2.1.2.1 Montagem do SD Card:

Para montar o SD Card, são usadas as bibliotecas “FS”, “SPI” e “SD” disponíveis para a ESP32. Com as funções mountSDCARD e unmountSDCARD monta-se e desmonta-se o cartão SD, quando necessário. A Figura 2.3 exibe as funções citadas.

```
/// mountSDCARD: monta o SD card no ESP32. -> Após isso, está pronto para uso.
bool mountSDCARD(){

  if(!SD.begin()){
    Serial.println("Card Mount Failed, restarting ESP.");
    return false;
  }

  uint8_t cardType = SD.cardType();

  if(cardType == CARD_NONE){
    Serial.println("No SD card attached");
    return false;
  }

  Serial.print("SD Card Type: ");

  if(cardType == CARD_MMC){
        Serial.println("MMC");
  } else if(cardType == CARD_SD){
        Serial.println("SDSC");
  } else if(cardType == CARD_SDHC){
        Serial.println("SDHC");
  } else {
        Serial.println("UNKNOWN");
  }
```

Figura 2.3: Funções mountSDCARD e unmountSDCARD.

Observa-se na função mountSDCARD a execução da função "SD.begin()", que inicia a montagem do cartão. Essa função retorna uma variável booleana, indicando se a montagem do cartão ocorreu. 1

2.1.2.2 Interface I2S - Comunicação entre o microfone e a ESP32

Inicialmente, configura-se a interface I2S através da função I2SMEMSAMPLER, mostrada na Figura 2.4.

```
I2SSampler *input = new I2SMEMSSampler(I2S_NUM_0, i2s_mic_pins, i2s_mic_Config); // configura a interface I2S
```

Figura 2.4: Função I2SMEMSAMPLER, para realizar a configuração da I2S.

Nessa função, são passados parâmetros como os pinos em que estão conectados os microfones e outras configurações (presentes na variável i2s_mic_config). A seguir, na Figura 2.5, são dados os valores configurados nessa variável.

```
// i2s config for reading from I2S
i2s_config_t i2s_mic_Config = {
    .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX),
    .sample_rate = SAMPLE_RATE,
    .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT,
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
    .dma_buf_count = 32,
    .dma_buf_len = 1024,
    .use_apll = false,
    .tx_desc_auto_clear = false};
```

Figura 2.5: Struct i2s_mic_Config, contendo os parâmetros de configuração da I2S.

Para o parâmetro ".mode", configura-se a ESP32 em modo receiver (para recepção de dados). É também definida a taxa de amostragem (16 kHz), o número de bits por amostra (32) e o formato do canal (I2S_CHANNEL _FMT_LEFT_RIGHT), configurado para operar em stereo. Através de diversos testes, constatou-se que é necessário utilizar um número grande de buffers (.dma_buf_count = 32) e também um tamanho adequado para cada buffer (.dma_buf_len = 1024), pois caso não o seja feito, são geradas descontinuidades nas gravações (a ESP não consegue processar as amostras na velocidade desejada). Otimizaram-se tais valores de modo a garantir uma amostragem segura em 16 kHz. Observou-se que aumentar o número de buffers levava a um overflow na memória de dados, sendo necessário controlar esse valor para que não ocorresse nenhuma falha.

2.1.2.3 Gravação

Para gravar o áudio, aloca-se dinamicamente na memória um buffer de 1024 posições de 64 bits (totalizando 8kiB), permitindo que a cada leitura sejam lidas 2048 amostras (1024 para cada canal). A Figura 2.6 exibe o trecho de código em que é declarado o buffer e inicia-se a comunicação I2S.

Figura 2.6: Declaração do buffer utilizado para armazenar as amostras adquiridas.

```
int buffer_size =  1024;
int64_t* samples = (int64_t *)malloc(sizeof(int64_t) * buffer_size); // buffer utilizado para gravação
input->start(); // Inicia o I2S
```

Após a alocação do buffer, cria-se o arquivo e é escrita uma header no formato WAV, de modo que o arquivo possa ser lido posteriormente. Para escrever a header, é utilizada a função writeWavHeader (Figura 2.7), que escreve no arquivo as variáveis da struct _wav_header, dada na Figura 2.8.

```
File fp = SD.open(fname, "wb");
int count_r = 0;  int mult = ceil(TEMPO*SAMPLE_RATE/buffer_size); // tempo de gravação

int fsize = mult * 2 * buffer_size * 2;
writeWavHeader(fp, fsize);
```

Figura 2.7: Declaração do arquivo de áudio e escrita da header WAVE.

```
typedef struct _wav_header
{
  // RIFF Header
  uint8_t riff_header[4]; // Contains "RIFF"
  int wav_size = 0;    // Size of the wav portion of the file, which follows the first 8 bytes. File size - 8
  uint8_t wave_header[4]; // Contains "WAVE"

  // Format Header
  uint8_t fmt_header[4];      // Contains "fmt " (includes trailing space)
  int fmt_chunk_size = 16; // Should be 16 for PCM
  short audio_format = 1;  // Should be 1 for PCM. 3 for IEEE Float
  short num_channels = 2;
  int sample_rate = 16000;
  int byte_rate = sample_rate* num_channels * 2;      // Number of bytes per second. sample_rate * num_channels * Bytes Per Sample
  short sample_alignment = 4; // num_channels * Bytes Per Sample
  short bit_depth = 16;       // Number of bits per sample

  // Data
  uint8_t data_header[4]; // Contains "data"
  int data_bytes = 0;  // Number of bytes in data. Number of samples * num_channels * sample byte size
  // uint8_t bytes[]; // Remainder of wave file is bytes
  _wav_header()
  {
    riff_header[0] = 'R';
    riff_header[1] = 'I';
    riff_header[2] = 'F';
    riff_header[3] = 'F';
    wave_header[0] = 'W';
    wave_header[1] = 'A';
    wave_header[2] = 'V';
    wave_header[3] = 'E';
    fmt_header[0] = 'f';
    fmt_header[1] = 'm';
    fmt_header[2] = 't';
    fmt_header[3] = ' ';
    data_header[0] = 'd';
    data_header[1] = 'a';
    data_header[2] = 't';
    data_header[3] = 'a';
  }
} wav_header_t;
```

Figura 2.8: Dados da header WAVE.

Nessa struct (Figura 8), são dadas informações sobre as faixas de áudio, como a taxa de amostragem, número de canais, taxa de bytes, alinhamento dos samples, e etc. Estão presentes, também, os caracteres necessários para a header (”RIFF”, ”WAVE”, ”fmt ”e ”data”).

Para realizar a leitura da porta I2S, utiliza-se a função I2SMEMSAMPLER::read, dada na Figura 2.9.

```
int I2SMEMSSampler::read(int64_t *samples, int count)
{
    size_t bytes_read = 0;
    i2s_read(m_i2sPort, samples, sizeof(int64_t) * count, &bytes_read, portMAX_DELAY);
    int samples_read = bytes_read / sizeof(int64_t);
    return samples_read;
}
```

Figura 2.9: Função para adquirir a data.

Nessa função. todos os dados lidos são salvos no buffer samples, definido na Figura 6. Para gravar as amostras lidas no cartão SD, executa-se a função "read"e são realizados deslocamentos nos bits das amostras originais, pois o microfone envia apenas 16 bits para cada amostra. Assim, os dados são escritos no cartão, como mostrado no trecho de código da Figura 2.10.

```
while (turn == 1)
{
  int samples_read = input->read(samples, buffer_size);
  uint8_t acquired_data[4];

  for(int i=0; i<samples_read;i++){
    acquired_data[0] = samples[i]>>48;
    acquired_data[1] = samples[i]>>56;
    acquired_data[2] = samples[i]>>16;
    acquired_data[3] = samples[i]>>24;
    fp.write(acquired_data,4);
  }

  count_r += samples_read;
  if (count_r >= mult*buffer_size){turn = 0; count_r = 0;}

}
```

Figura 2.10: Função de leitura das amostras.

Após a gravação, os buffers alocados são liberados da memória (de modo garantir que não haja overflow). Observou-se que durante a gravação, é necessário manter o WiFi desligado, pois há conflito entre as interrupções (I2S x WiFi).

2.1.2.4 Temporização

Para realizar uma temporização adequada, e marcar o horário de cada gravação, é feito uso da biblioteca NTPClient, do qual requer conexão Wi-Fi. A linha de código da Figura 2.11 mostra como é feita a aquisição da data.

```
String date = timeClient.getFormattedDate();
```

Figura 2.11: Função para adquirir a data.

A partir dessa temporização, cada arquivo de áudio terá o nome contendo a data e horário em que foi feita a gravação.

2.2 Web server

Para realizar o envio das gravações, utiliza-se a ESP32 e o Google AppScript, plataforma que permite utilizar a API do Google Drive. Através da AppScript, gera-se um endereço para o qual a ESP32 pode enviar HTTP Gets e Posts, sendo assim possível realizar a transmissão dos dados. A seguir, serão descritos os códigos para as duas plataformas (ESP32 e AppScript).

2.2.1 Código ESP32

Nessa seção, serão descritas as funções utilizadas para realizar o envio dos áudios pela ESP32.

2.2.1.1 sendBinaryDataToAppScript

Para realizar o envio do arquivo, utiliza-se a função 'sendBinaryDataToAppScript'. Nela, inicialmente é feito um HTTP Get, enviando ao servidor o nome do arquivo e seu tamanho, para que posteriormente seja checado se o arquivo recebido equivale (em tamanho) ao que foi enviado. Para garantir um resultado positivo no Get, são feitas até 5 tentativas, e caso não seja possível enviar o filename, é encerrada a operação. Esse mesmo procedimento é também realizado para outras funções que dependem do WiFi, como os HTTP Posts, por onde são enviadas as amostras. A Figura 2.12 exibe as configurações para realizar o Get, em que a variável "url"contém o endereço fornecido para acessar a API.

```
HTTPClient http;
http.begin(url + "?filename=" + result + "&fileSize=" + String(fileSize) + "&check_filesize=false"  + "&delete_file=false");
int GETCODE = http.GET();  i = 0;
Serial.println("GET result code: " + String(GETCODE));
while((GETCODE != 302)) {
  GETCODE = http.GET(); i++;
  Serial.println("GET result code: " + String(GETCODE));
  if (i>= n_try) {Serial.println("Failed to send filename!"); return; }
}
http.end();
```

Figura 2.12: HTTP Get realizado para enviar o nome e tamanho do arquivo.

Na Figura 2.13, é definido um buffer (utilizado para armazenar os dados lidos do SDCARD), e define-se o tipo de dado a ser enviado (binário).

```
uint8_t * postbuffer = (uint8_t *)malloc(sizeof(uint8_t) * BUFFER_SIZE);
int bytesRead; int totalSent = 0;

http.setTimeout(20000); // Timeout elevado pois os buffers possuem 32kB.
http.begin(url);
http.addHeader("Content-Type", "application/octet-stream"); // Headerless binary files
```

Figura 2.13: Configurações para envio das amostras.

Após isso, são enviados os dados em pacotes de tamanho 32kiB, através de HTTP Posts, conforme o trecho de código da Figura 2.14.

```
// Leitura do arquivo:
while (file.available()) {

  bytesRead = file.read(postbuffer, BUFFER_SIZE);
  Serial.println(bytesRead);

  if (bytesRead > 0) {
    Serial.printf("Sending chunk %d - %d bytes...\n", totalSent, totalSent + bytesRead);
    bool success;
    for (int retry = 0; retry < NUM_RETRIES; retry++) {

      int httpResponseCode = http.POST((postbuffer), bytesRead);

      if ((httpResponseCode > 0)){
        Serial.printf("HTTP Response code: %d\n", httpResponseCode);
        String payload = http.getString();
        Serial.println(payload);
        //Serial.println("Sent chunk successfully.");
        success = true; break;
      } else {
        Serial.printf("Error sending data (Retry %d): %s\n", retry + 1, http.errorToString(httpResponseCode).c_str());
        delay(RETRY_DELAY);
      }

    }

    if (!success) {
      Serial.println("Failed to send buffer. Aborting transmission.");
      break;
    }

    totalSent += bytesRead;

  } else {

    Serial.println("Error reading file");
    break;
  }
}
```

Figura 2.14: Trecho que realiza consecutivamente os HTTP Posts.

Nesse código, lê-se 32kiB do arquivo, armazenando as informações no buffer, e então é realizado o envio. Esse processo é repetido até que o arquivo tenha sido enviado por completo. Caso o servidor retorne um erro na recepção de um buffer, são feitas até 5 tentativas.

Por fim, libera-se o buffer da memória, fecha-se o arquivo, e desliga-se a conexão HTTP.

2.2.1.2 Checando a equivalência entre os tamanhos dos arquivos

Com a função "checkFileSize", envia-se um Get para o servidor de modo a identificar se o tamanho do arquivo salvo no servidor é igual ao tamanho do arquivo no cartão SD, como ilusta a Figura 2.15.

```
String checkFileSizeWithAppsScript(const String& fname) {

  char result[100]; // You can adjust the size as per your requirement
  getSubstringFromSecond(fname, result);
  String urlGET = url + "?filename=" + result + "&fileSize=" + String(fileSize) + "&check_filesize=true"  + "&delete_file=false"; // Include the parameters in the URL
  //String urlGET = url + "?check_filesize=true"; // Include the parameters in the URL
  Serial.println(urlGET);
  HTTPClient http;
  http.begin(urlGET);
  http.setFollowRedirects(HTTPC_STRICT_FOLLOW_REDIRECTS);
  int httpCode = http.GET();
  Serial.println(httpCode);
  if (httpCode == 200 ) {
    String response = http.getString();
    http.end(); // Release resources and close the HTTP connection
    return response;
  } else {
    Serial.println("Error in HTTP GET request to Google Apps Script");
    http.end(); // Release resources and close the HTTP connection
    return ""; // Return an empty string to indicate an error
  }
}
```

Figura 2.15: Função para checar o tamanho do arquivo no Drive.

Nessa função, o parâmetro "check_filesize"presente na URL é definido como "true", o que fará com o que o servidor execute uma função para comparar os tamanhos dos arquivos. Caso o servidor retorne "false", é chamada a função deleteOldFile, que realiza outro Get, agora com um comando para deletar o arquivo

(parâmetro delete_file = true). Desse modo, o programa tenta novamente enviar o arquivo, até um limite máximo de tentativas (5). A Figura 2.16 exibe a função deleteOldFile.

```
String deleteOldFile(const String& fname)
{
  char result[100]; // You can adjust the size as per your requirement
  getSubstringFromSecond(fname, result);
  String urlGET = url + "?filename=" + result + "&fileSize=" + String(fileSize) + "&check_filesize=false" + "&delete_file=true"; // Include the parameters in the URL

  HTTPClient http;
  http.begin(urlGET);
  http.setFollowRedirects(HTTPC_STRICT_FOLLOW_REDIRECTS);
  int httpCode = http.GET();
  Serial.println(httpCode);
  if (httpCode == 200 ) {
    String response = http.getString();
    http.end(); // Release resources and close the HTTP connection
    return response;
  } else {
    Serial.println("Error in HTTP GET request to Google Apps Script");
    http.end(); // Release resources and close the HTTP connection
    return ""; // Return an empty string to indicate an error
  }
}
```

Figura 2.16: Função para deletar o arquivo no Drive.

2.2.2 Código AppScript

A AppScript é uma plataforma da Google com um ambiente de desenvolvimento que permite a criação de aplicações em javascript que se integram com o Google Workspace (Google Drive, GMail e etc.). Nesse trabalho, desenvolveu-se um código na AppScript que implementa duas funções: doGet e doPost, associadas as operações de Get e Post realizadas pela ESP32. Essas funções viabilizam a comunicação entre a ESP32 e o Google, e vice-versa, permitindo o envio dos arquivos.

2.2.2.1 doGet

A função doGet permite com que a ESP32 realize uma requisição para a Google, enviando parâmetros e recebendo uma resposta do servidor. Nessa função, os parâmetros passados são: nome do arquivo e seu tamanho (filename e fileSize), e variáveis condicionais (checkFileSize e deleteFile) para realizar a checagem do tamanho ou deletar o arquivo (na situação em que os tamanhos diferem). Na Figura 2.17, é mostrado um trecho da função doGet, em que são declaradas variáveis locais e globais (que são utilizadas no POST).

```
const DESTINATION_FOLDER_ID = "XXXXX";

function doGet(e) {
  var filename = e.parameter.filename;
  var fileSize = e.parameter.fileSize;
  var checkFileSize = e.parameter.check_filesize;
  var deleteFile = e.parameter.delete_file;

  // Store the filename in Script Properties
  PropertiesService.getScriptProperties().setProperty("filename", filename);
  PropertiesService.getScriptProperties().setProperty("fileSize", fileSize);
```

Figura 2.17: Declaração das variáveis da função doGet.

Após a declaração, são checadas as variáveis condicionais mencionadas anteriormente, ou seja, se o código deve ou não comparar os tamanhos ou deletar o arquivo. A Figura 2.18 exibe o trecho de código para essas condicionais.

```
// Check if the checkFileSize parameter is set to true
if (checkFileSize && checkFileSize.toLowerCase() === "true") {

  var fileId = getFileIdByName(filename); // Get the file ID of the file with the provided filename
  if (fileId) {

    PropertiesService.getScriptProperties().setProperty("fileId", fileId);
    var fileSizeOnDrive = getFileSize(fileId);  // Get the file size of the file in Google Drive
    var isFileSizeMatching = (fileSizeOnDrive == fileSize); // Compare the file size from the ESP32 with the file size on Google Drive

    return ContentService.createTextOutput(isFileSizeMatching.toString());   // Return the result as a response
  } else {
    return ContentService.createTextOutput("false"); // File not found in Google Drive, return false

  }
}

if (deleteFile && deleteFile.toLowerCase() == "true") {

  var fileId = getFileIdByName(filename);
  var result = deleteFileById(fileId);

  var i = 0;

  while (result == "false") {
    i = i+1;
    result = deleteFileById(fileId);
    if (i>= 3) {
      break;
    }
  }

  return ContentService.createTextOutput(result);
}
else {
  return ContentService.createTextOutput("Request received successfully");
}
```

Figura 2.18: Trecho da função doGet para comparar o tamanho do arquivo ou deletá-lo.

2.2.2.2 doPost

A função doPost é responsável por receber os dados enviados pela ESP32, em chunks de 32kiB. No envio do primeiro chunk, cria-se o arquivo com o filename enviado, e após isso os próximos chunks são adicionados ao arquivo. Utiliza-se o formato 'octet-stream', que representa dados binários sem header. A Figura 2.19 exibe a função doPost completa, contendo comentários para cada seção.

```
function doPost(e) {

  var filename = PropertiesService.getScriptProperties().getProperty("filename");
  var fileSize = PropertiesService.getScriptProperties().getProperty("fileSize");
  var contentBlob = e.postData;
  var bytes = contentBlob.getBytes();


  var folder = DriveApp.getFolderById(DESTINATION_FOLDER_ID); // Save the binary data to Google Drive
  var files = folder.getFilesByName(filename);

  // Check if the file already exists
  if (files.hasNext()) {
    var file = files.next();
    var fileId = file.getId();

    // Fetch the existing file content as a Blob
    var existingData = file.getBlob();

    // Append the new data to the existing file content
    var updatedData = concatUint8Arrays(existingData.getBytes(),bytes);
    Drive.Files.update({}, fileId, Utilities.newBlob(updatedData,"application/octet-stream"));  // Use the Advanced Drive Service to update the file content
  } else {
    // If the file doesn't exist, create a new one
    var blob = Utilities.newBlob(bytes, "application/octet-stream", filename);
    var file = folder.createFile(blob);
  }

  // Return a response to the ESP32
  var response = ContentService.createTextOutput("File saved successfully");
  response.setMimeType(ContentService.MimeType.TEXT);
  return response;
}
```

Figura 2.19: Função doPost.

São utilizadas também, funções extras: concatUint8Arrays, getFileIdByName e deleteFileById. A função concatUint8Arrays (Figura 2.20) permite concatenar os dados de forma adequada, e as funções getFileIdByName, getFileSize e deleteFileById (Figura 2.21), obtêm o ID do arquivo, o tamanho, e deleta o arquivo.

```
function concatUint8Arrays(array1, array2) {
  var newArray = new Uint8Array(array1.length + array2.length);
  newArray.set(array1, 0);
  newArray.set(array2, array1.length);
  return newArray;
}
```

Figura 2.20: Função concatUint8Arrays.

```
function getFileIdByName(filename) {
  try {
    var files = DriveApp.getFilesByName(filename);
    if (files.hasNext()) {
      return files.next().getId();
    }
    return null; // Return null if the file is not found
  } catch (error) {
    return null; // Return null if an error occurs
  }
}

function getFileSize(fileId) {
  try {
    var file = DriveApp.getFileById(fileId);
    return file.getSize();
  } catch (error) {
    return -1; // Return -1 to indicate an error
  }
}

function deleteFileById(fileId) {
  try {
    Drive.Files.remove(fileId);
    Logger.log("File with ID '" + fileId + "' was successfully deleted.");
    return "true";
  } catch (error) {
    Logger.log("Error deleting file with ID '" + fileId + "': " + error);
    return "false";
  }
}
```

Figura 2.21: Funções getFileIdByName, getFileSize e deleteFileById.

2.3 PC

Após o envio dos arquivos de áudio ao servidor, é possível realizar algum tipo de processamento ou classificação dessas faixas. Neste trabalho, foram aplicadas dois modelos distintos de classificação: o SED (para classificação de cenas acústicas) e a SVM (para classificar anormalidades em motores), sendo o primeiro de propósito geral e o segundo restrito a uma aplicação.

2.3.1 Modelos de classificação

A seguir, serão descritos os modelos utilizados, destacando suas características e finalidade.

2.3.1.1 SED (LINSE)

O SED é um modelo treinado no Laboratório de Circuitos e Processamento de Sinais (UFSC), utilizado para classificação de cenas acústicas. Possui cerca de 500 classes, que variam entre diferentes cenas. O modelo utiliza a arquitetura PaSST (Patchout faSt Spectrogram Transformer), e retorna ao usuário as 5 predições com maior probabilidade. Para acessar o modelo, deve se estar conectado à rede WiFi do LINSE, e realizar um HTTP Post com o arquivo de áudio que deseja-se classificar, sendo retornado pelo servidor as predições e as probabilidades. O modelo possui uma alta acurácia, além de conseguir identificar mais de uma cena acústica por faixa, considerando que retorna 5 predições. A Figura 2.22 mostra um exemplo de predição utilizando o SED.

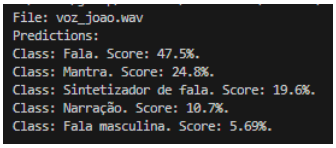

Figura 2.22: Predições do SED.

Observa-se na figura a predição de um arquivo de fala masculina, e que o modelo conseguiu classificá-lo corretamente.

2.3.1.2 SVM

Uma segunda aplicação desse trabalho foi utilizar um modelo de classificação (SVM) para classificar o funcionamento adequado de motores de portão no final da linha de montagem. Deseja-se realizar uma classificação entre 'normal' ou 'anormal', de modo a facilitar com que a empresa possa retornar o produto para manutenção. A classificação será feita através de informações extraídas do som do motor em funcionamento, e, para tanto, treinou-se o modelo com diversas características sonoras extraídas das faixas de áudio, do tipo espectral, temporal, harmônico e rítmico. Para se aproximar da situação real, utilizou-se o dataset MIMII [1], que dispõe de diversos arquivos de áudio que contêm sons de máquinas em bom ou mau funcionamento. Dentre as máquinas disponíveis no dataset, escolheu-se treinar o modelo com sons de ventoinha.

2.3.2 Descrição do código

O programa responsável por realizar a classificação foi desenvolvido em Python. Seu funcionamento consiste em realizar uma autenticação com a Drive API, baixar os arquivos de áudio que foram enviados pela ESP32 e passar o áudio pelo modelo. Sendo as variáveis de entrada desse modelo as características espectrais, temporais, e etc. citadas, é necessário extrair tais características dos áudios baixados para utilizá-las como entrada do modelo. A Figura 21 ilustra o trecho inicial do código, no qual carrega-se o modelo pré-treinado e são chamadas as funções que executam o passo a passo descrito acima.

Após esse primeiro trecho, o programa passa a entrar em um loop de 120 segundos (valor configurável) e repetir o processo (checar se há algum arquivo novo a ser baixado, e caso haja, realizar a predição).

A seguir, serão descritas as funções utilizadas nesse programa.

2.3.3 Autenticação com a Google Drive API

Para realizar a autenticação com a Google, utiliza-se a biblioteca google.auth disponível para o Python. É necessário, antes da execução do código, baixar um arquivo de autenticação gerado pela Google Drive API (no portal Google Cloud). Esse arquivo será acessado por funções da biblioteca, e permitirá ao usuário realizar o download (ou upload) de arquivos do Drive. Observa-se que na primeira execução, abre-se uma janela no navegador pedindo-se permissão para o acesso. Após isso, será possível realizar a autenticação de forma automática sempre que se executa tal função. A Figura 2.23 exibe a função que realiza o processo de autenticação.

[1] Web-site MIMII: https://zenodo.org/record/3384388.

```python
def authenticate_with_drive_api():
    # Create a flow instance for OAuth 2.0
    flow = InstalledAppFlow.from_client_secrets_file(
        'C:/Users/joaop/OneDrive/Documentos/Arduino/wifi_tests/APPSCRIPT/python - SVM (noise detection)/credentials.json'
        , scopes=SCOPES
    )

    if os.path.exists(TOKEN_PICKLE_FILENAME):
        with open(TOKEN_PICKLE_FILENAME, 'rb') as token:
            credentials = pickle.load(token)
    else:
        # Run the OAuth 2.0 authorization flow
        credentials = flow.run_local_server()

        # Save the token for future use
        with open(TOKEN_PICKLE_FILENAME, 'wb') as token:
            pickle.dump(credentials, token)

    drive_service = googleapiclient.discovery.build('drive', 'v3', credentials=credentials)

    return drive_service
```

Figura 2.23: Função authenticate_with_drive_api.

2.3.4 Obtenção dos nomes dos arquivos

Para obter o nome dos arquivos presentes na pasta do Drive ao qual se deseja acessar, realiza-se um HTTP GET para a Google API, contendo na url a ID da pasta e a chave da API disponível na Google Cloud. Desse modo, é possível verificar todos os arquivos .wav que deseja-se baixar, o que será necessário para identificar se há algum arquivo novo (em relação aos já presentes no disco). A Figura 2.24 exibe essa função.

```python
def retrive_wav_filenames_from_folder(folder_id):

    url = f"https://www.googleapis.com/drive/v3/files?q='{folder_id}'+in+parents+and+name contains '.wav'&key={API_KEY}"
    response = requests.get(url)
    data = response.json()

    if "files" not in data:
        print("No .wav files found in the folder.")
        return

    file_ids = []; filenames = [];
    for file_info in data["files"]:
        file_ids.append(file_info["id"])
        filenames.append(file_info["name"])

    return file_ids, filenames
```

Figura 2.24: Função retrieve_wav_filenames_from_folder.

2.3.5 Verificação dos arquivos em disco e comparação com a lista de arquivos obtida

Para verificar se é necessário fazer o download dos arquivos, compara-se a lista de arquivos presentes no Drive com o que está salvo no disco, através das funções "get_filelist_fromDisk"e "compare_and_delete", dadas na Figura 2.25.

```python
def get_filelist_fromDisk():
    file_list = os.listdir(output_directory)

    for i in file_list:
        if i.endswith(".wav") == False:
            file_list.remove(i)

    return file_list

def compare_and_delete(files_to_download, files_in_disk):
    set1 = set(files_to_download); set2 = set(files_in_disk)
    common_items = set1.intersection(set2)

    # Remove common items from both lists
    for item in common_items:
        while item in files_to_download:
            files_to_download.remove(item)
    return files_to_download
```

Figura 2.25: Funções get_filelist_fromDisk e compare_and_delete.

Na primeira função, utiliza-se a biblioteca "os"para acessar os arquivos em disco. Na segunda função, compara-se os valores das duas listas de arquivos, removendo os que estão em comum.

2.3.6 Download dos arquivos

Na função "Download Wav files from folder", entrega-se a entrada um dicionário contendo os nomes dos arquivos e suas respectivas IDs (do Drive). A função possui outras duas entradas, sendo elas o diretório no qual deseja-se salvar os arquivos, e uma variável que é retornada pelo processo de autenticação (necessária para o download). Desse modo, os arquivos são baixados através da função "googleapiclient.http.MediaIoBaseDownload". A Figura 2.26 exibe o código dessa função.

```
def download_wav_files_from_folder(file_dict, output_directory, drive_service):

    if not file_dict:
        print("There are no new files.")
        return False

    for file_id in file_dict:
        file_name = file_dict[file_id]

        request = drive_service.files().get_media(fileId=file_id)

        output_file_path = output_directory + '/' + file_name
        # Save the downloaded content to a file on disk
        with open(output_file_path, "wb") as file:
            downloader = googleapiclient.http.MediaIoBaseDownload(file, request)
            done = False
            while not done:
                status, done = downloader.next_chunk()
                print(f"Downloading file {file_name}. Progress:  {int(status.progress() * 100)}%")
                print("----------------------------------------")

    return True
```

Figura 2.26: Função Download WAV files from folder.

2.3.7 Predição

Por fim, na função "prediction", entrega-se como entrada a lista de arquivos ao qual deseja-se realizar a predição e o modelo (SVM). Nessa função, cada arquivo é passado por uma função de extração de features (extract_features), que serão as variáveis de entrada do modelo. É utilizada também a função "get_train_values", que acessa o dataset original utilizado para treinar o modelo e realiza a divisão de conjuntos. Isso é necessário para que se possa escalonar as features extraídas de forma adequada, através da função StandardScaler(). A Figura 2.27 exibe o código da função "prediction", enquanto a Figura 2.28 a função "extract_features"e a Figura 2.29 a função "get_train_values".

```python
def prediction(file_list, model):

    preds = ''
    X_train, columns = get_train_values()
    scaler = StandardScaler(); scaler.fit(X_train)

    for i in file_list:
        if i.endswith('.wav'):
            path = 'C:/Users/joaop/OneDrive/Documentos/Arduino/wifi_tests/APPSCRIPT/python - SVM (noise detection)/files/' + i
            data, sr = librosa.load(path, sr = None, mono=True)
            print('Reading file: ' + i)
            features = (extract_features(data)).reshape(1,-1)
            features = pd.DataFrame(features,columns=columns)
            X = scaler.transform(features)
            pred =  model.predict(X)
            print('Prediction: ' + pred)
            preds = preds + 'File: ' + i  + '. Prediction: ' + str(pred) + '\n'

            print("----------------------------------------")
            preds = preds + ("----------------------------------------\n")


    return True, preds
```

Figura 2.27: Função "prediction".

Na Figura 2.28)a), são realizados os processos iniciais para se extrair as informações necessárias para os cálculos das features, como a retirada do espectrograma, a sua normalização, a energia de cada frame do espectrograma, e etc. Após isso, são calculadas algumas das features (entropy, spread, skewness, slope, decrease). Na Figura 2.28)b), são calculadas diversas das outras features do modelo, através de funções da biblioteca librosa. Por fim, retorna-se um vetor contendo todas essas features.

```python
def extract_features(y):
    sr = 16000; nperseg=2048; n_mfcc=25

    _, _, spectrogram_data = spectrogram(x=y, fs=sr, nperseg=2048)
    spectrogram_data = np.abs(spectrogram_data)  # Take the absolute value of the spectrogram
    power_spectrum = np.mean(spectrogram_data, axis=1)
    normalized_spectrum = power_spectrum / np.sum(power_spectrum)
    frequency_axis = np.fft.rfftfreq(nperseg, 1 / sr)
    log_power_spectrum = np.log10(power_spectrum)
    energy = np.sum(spectrogram_data, axis=1)
    n_frames = energy.shape[0]

    spectral_entropy = entropy(normalized_spectrum, base=2)
    spread = np.sqrt(np.sum((frequency_axis[:, np.newaxis] - np.mean(frequency_axis)) ** 2 * power_spectrum[:, np.newaxis]) / np.sum(power_spectrum))
    skewness = skew(power_spectrum)
    slope, _, _, _, _ = linregress(frequency_axis, log_power_spectrum)
    decrease = np.zeros(n_frames)
    for i in range(1, n_frames):
        decrease[i] = np.sum(energy[i:] - energy[i - 1]) / (n_frames - i)

    peak_amplitude = np.max(np.abs(y))
    diff = np.diff(y)  # Compute the differences between consecutive samples
    mean_diff = np.mean(np.abs(diff))  # Calculate the mean absolute difference
    smoothness = mean_diff / peak_amplitude
```

((a))

```python
features = []
features = np.append(features,np.mean(librosa.feature.chroma_stft(y=y,sr=sr)))
features = np.append(features,np.var(librosa.feature.chroma_stft(y=y,sr=sr)))
features = np.append(features, np.mean(librosa.feature.rms(y=y).flatten()))
features = np.append(features, np.var(librosa.feature.rms(y=y).flatten()))
features = np.append(features,librosa.feature.spectral_centroid(y=y,sr=sr).mean())
features = np.append(features,librosa.feature.spectral_centroid(y=y,sr=sr).var())
features = np.append(features,librosa.feature.spectral_bandwidth(y=y,sr=sr).mean())
features = np.append(features,librosa.feature.spectral_bandwidth(y=y,sr=sr).var())
features = np.append(features,librosa.feature.spectral_rolloff(y=y,sr=sr).mean())
features = np.append(features,librosa.feature.zero_crossing_rate(y=y).mean())
harmony, perceptual = librosa.effects.hpss(y=y)
features = np.append(features,harmony.var())
features = np.append(features,perceptual.mean()); features = np.append(features,perceptual.var())
mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n_mfcc)
features = np.append(features,np.mean(mfccs,axis=1).reshape(-1,1))
features = np.append(features,np.var(mfccs,axis=1).reshape(-1,1))

contrast = librosa.feature.spectral_contrast(y=y, sr=sr)
features = np.append(features,np.mean(contrast))
features = np.append(features,np.var(contrast)) #spectral contrast mean + var
stft = librosa.stft(y=y);  magnitude = np.abs(stft); flux = librosa.onset.onset_strength(S=magnitude,sr=sr);
features = np.append(features, np.mean(flux)); features = np.append(features,np.var(flux)) #spectral flux mean + var
crest = np.max(magnitude, axis=0) / (np.sum(magnitude, axis=0) + 1e-10)
features = np.append(features, np.mean(crest));
flatness = librosa.feature.spectral_flatness(y=y)
features = np.append(features, np.var(flatness)) #spectral flatness mean + var
features = np.append(features, np.mean(decrease));
features = np.append(features, slope); #spectral slope
features = np.append(features, skewness); #spectral skewness
features = np.append(features, spread); #spectral spread
features = np.append(features, spectral_entropy); #spectral entropy
features = np.append(features, smoothness); # peak smoothness

return features
```

((b))

Figura 2.28: Função extract features.