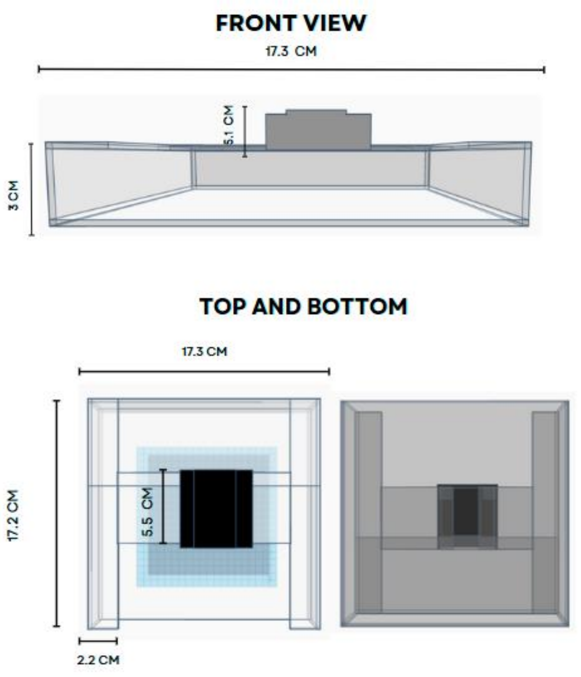
Title: VOICE - CONTROLLED SIGNAL LIGHTS VEST FOR CYCLISTS FOR ENHANCED SAFETY AND COMMUNICATION ON THE ROAD
Name of Researchers: Añoso, Arjay T
Beligañio, Arjay S.
Brigildo, John Rafael H.
ADVISER: Engr. Madonna D. Castro

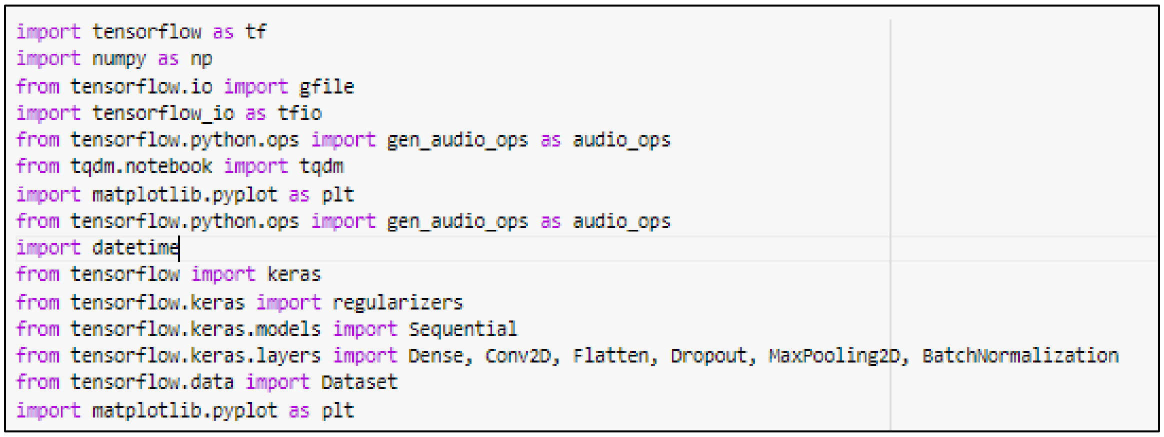
Recommendations	**Old**	**Revised**	**Page**
Put the reference cited for the signals used in the prototype/ also in chapter II study on symbols • Ma'am Ceribo	None	Traffic Signals Traffic control signals are devices placed along, beside, or above a roadway to guide, warn, and regulate the flow of traffic, which includes motor vehicles, motorcycles, bicycles, pedestrians, and other road users. The green arrow pointing right or left allows you to make a protected turn; oncoming vehicles, bicycles, and pedestrians are stopped by a red light as long as the green arrow is lit. A flashing red signal light means exactly the same as a stop sign: STOP! After stopping, proceed when safe and observe the right-of-way rules. A flashing yellow signal light warns you to be careful. Slow down and be especially alert.	Page 72
Add the recommendations • Ma'am Ceribo	1. Future researchers could use a wireless connection for the microphone of the device wherein it can be in Bluetooth or wifi connections. 2. Future researchers could use other algorithms for voice recognition	1. Future researchers could use a wireless connection for the microphone of the device wherein it can be in Bluetooth or wifi connections. 2. Future researchers could use other algorithms for voice recognition system model that implement more advanced pattern recognition techniques that distinguish the different between the specific command words to its similar sound words. 3. Future researchers could add features to the device that can communicate with another cyclist. 4. Future researchers could have an indicator that the cyclists see or know whenever it commands.	Page 71

	system model that implement more advanced pattern recognition techniques that distinguish the different between the specific command words to its similar sound words. 3. Future researcher could add features to the device that can communicate with another cyclist.		
Revision for the program (Technicalities, Training of voice, Test of training, testing) • Ma'am Ceribo		Add importing libraries, audio file validation, converting audio signals into spectrogram and process background noise (below the table)	Page 45
Revision for the program.(Figure. 11) How and Why? • Ma'am Ceribo		Add get_spectrogram function (below the table)	Page 46

Revision for the program.(Figure. 17) Present the figure clearly • Ma'am Ceribo		Change the figure with screenshot	Page 51
Efficiency of voice • Ma'am Ceribo		Table 11. Results of Response of the Signal Light Displayed based on 5 consecutive input commands	Page 57
Loud voice command • Ma'am Ceribo		Table 10. Results of Response of the Signal Light Displayed based on distance of the user	Page 60
Cover for the Prototype • Ma'am Ceribo		 FRONT VIEW 17.3 CM 5.1 CM 3 CM TOP AND BOTTOM 17.3 CM 17.2 CM 5.5 CM 2.2 CM	Page 42

Add in the Programming Part

The figure below shows how to import necessary libraries and modules for the voice command recognition model.

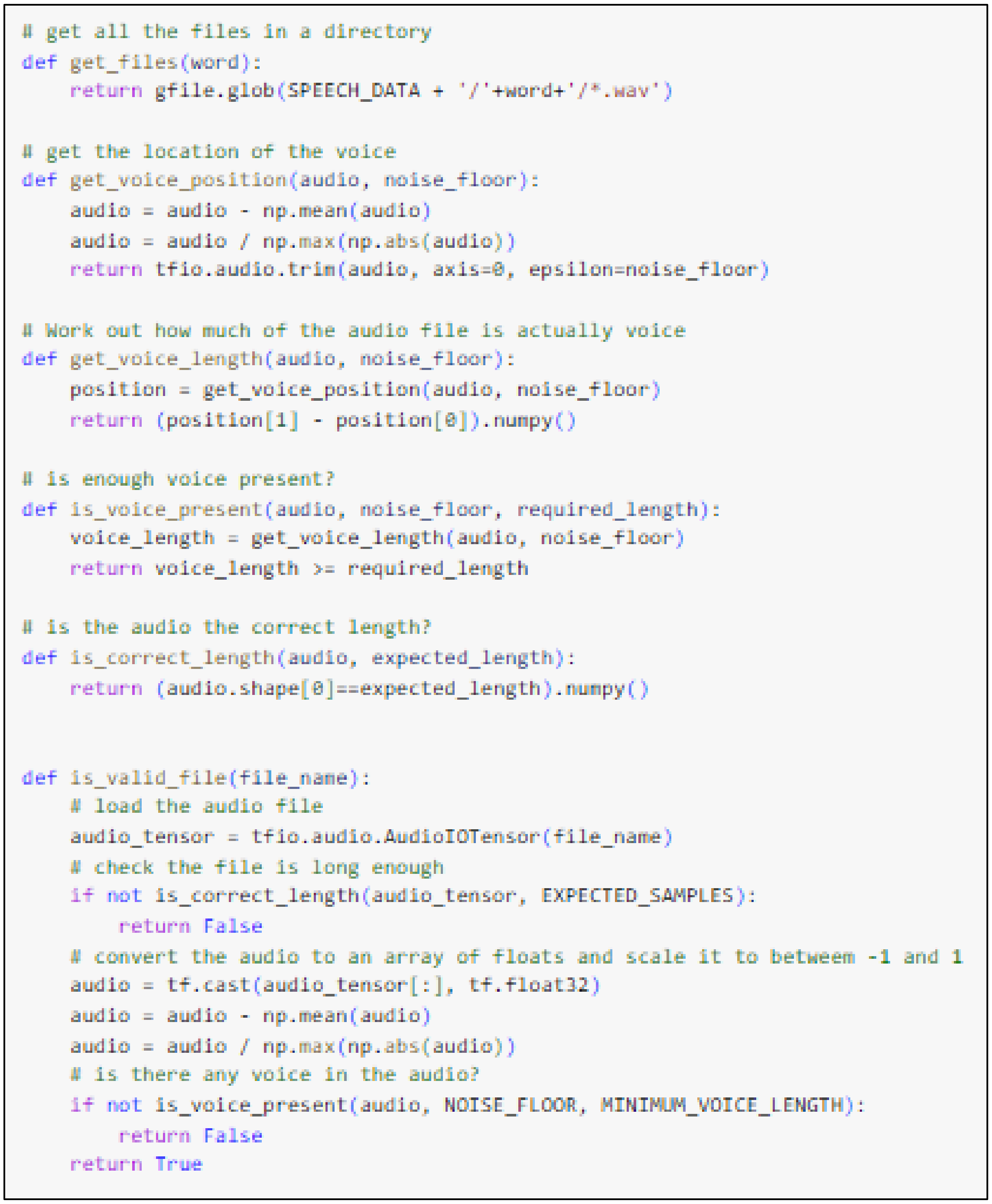


```python
import tensorflow as tf
import numpy as np
from tensorflow.io import gfile
import tensorflow_io as tfio
from tensorflow.python.ops import gen_audio_ops as audio_ops
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
from tensorflow.python.ops import gen_audio_ops as audio_ops
import datetime
from tensorflow import keras
from tensorflow.keras import regularizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D, BatchNormalization
from tensorflow.data import Dataset
import matplotlib.pyplot as plt
```

Fig. 12 Importing libraries

The code involved imports all the libraries and modules needed for working with audio data, visualize it, and develop and train neural network models.

The figure below shows how to validate audio files by checking if they meet certain criteria, such as containing enough voice content and having the correct length, to ensure they are suitable for further processing in a command recognition system.

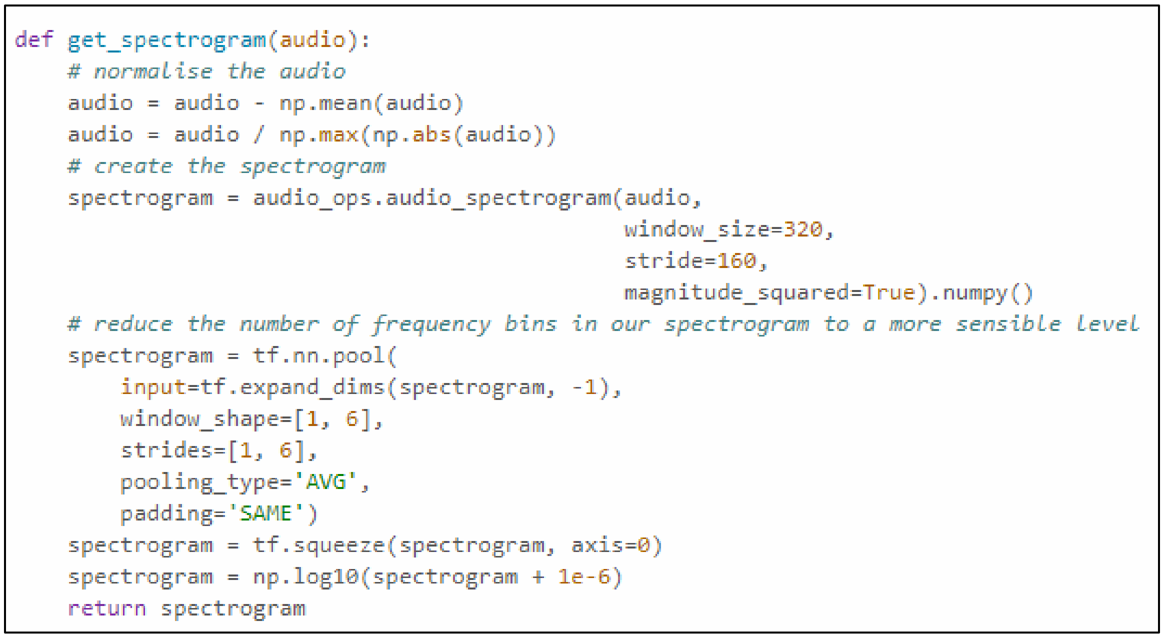


```
# get all the files in a directory
def get_files(word):
    return gfile.glob(SPEECH_DATA + '/'+word+'/*.wav')

# get the location of the voice
def get_voice_position(audio, noise_floor):
    audio = audio - np.mean(audio)
    audio = audio / np.max(np.abs(audio))
    return tfio.audio.trim(audio, axis=0, epsilon=noise_floor)

# Work out how much of the audio file is actually voice
def get_voice_length(audio, noise_floor):
    position = get_voice_position(audio, noise_floor)
    return (position[1] - position[0]).numpy()

# is enough voice present?
def is_voice_present(audio, noise_floor, required_length):
    voice_length = get_voice_length(audio, noise_floor)
    return voice_length >= required_length

# is the audio the correct length?
def is_correct_length(audio, expected_length):
    return (audio.shape[0]==expected_length).numpy()


def is_valid_file(file_name):
    # load the audio file
    audio_tensor = tfio.audio.AudioIOTensor(file_name)
    # check the file is long enough
    if not is_correct_length(audio_tensor, EXPECTED_SAMPLES):
        return False
    # convert the audio to an array of floats and scale it to betweem -1 and 1
    audio = tf.cast(audio_tensor[:], tf.float32)
    audio = audio - np.mean(audio)
    audio = audio / np.max(np.abs(audio))
    # is there any voice in the audio?
    if not is_voice_present(audio, NOISE_FLOOR, MINIMUM_VOICE_LENGTH):
        return False
    return True
```

Fig. 13 Audio file validation

This program makes sure audio files meet certain requirements in order to verify them for command word recognition. It scales the audio for consistency and verifies that the files are the proper length and have enough speech content above a noise threshold. The audio file must be loaded, normalized, and its voice duration and presence evaluated in order to validate it.

The figure below shows the function to transform an audio signal into a spectrogram, which is a visual representation of the frequency spectrum of the audio as it varies over time.

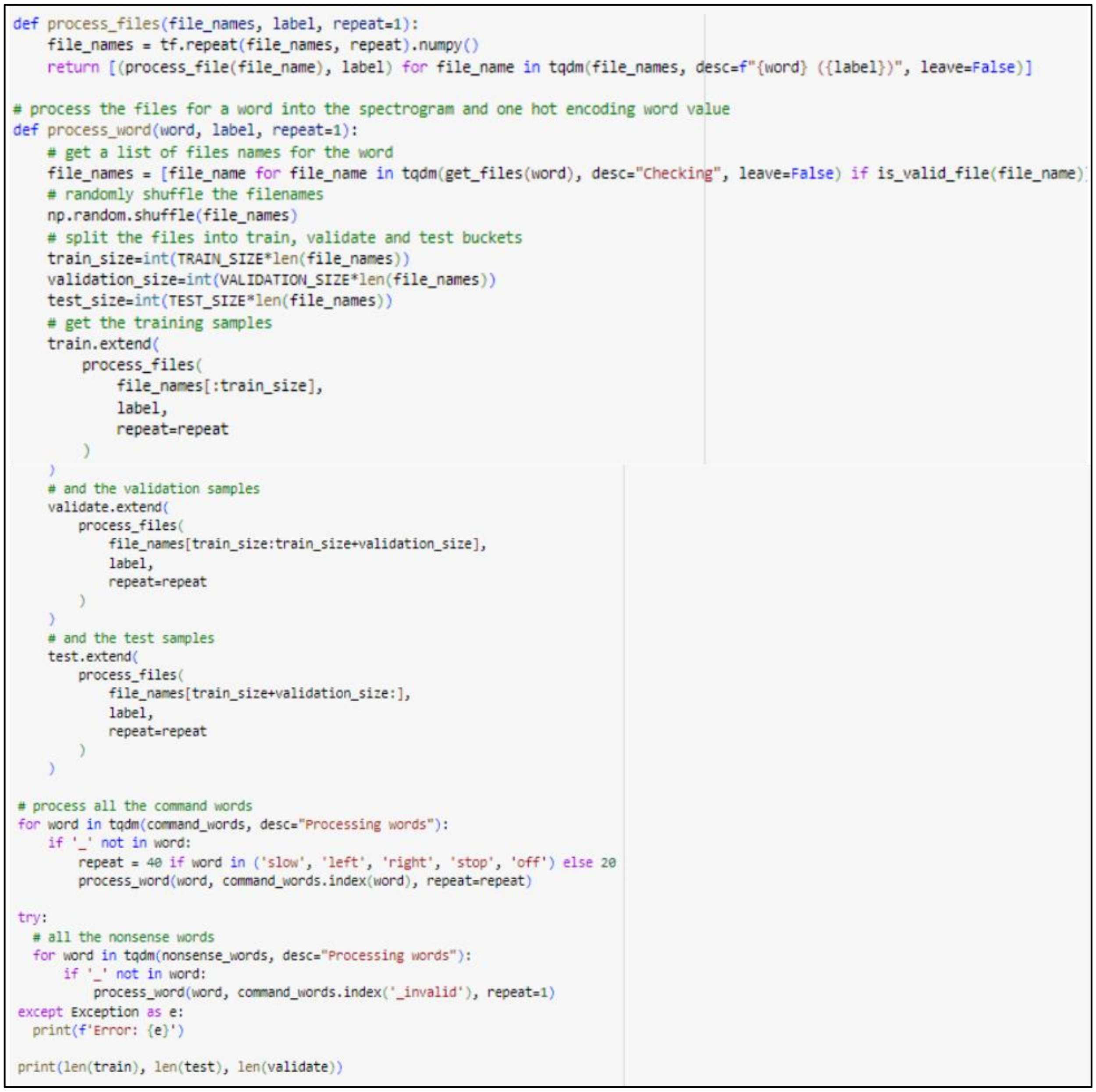


```python
def get_spectrogram(audio):
    # normalise the audio
    audio = audio - np.mean(audio)
    audio = audio / np.max(np.abs(audio))
    # create the spectrogram
    spectrogram = audio_ops.audio_spectrogram(audio,
                                              window_size=320,
                                              stride=160,
                                              magnitude_squared=True).numpy()
    # reduce the number of frequency bins in our spectrogram to a more sensible level
    spectrogram = tf.nn.pool(
        input=tf.expand_dims(spectrogram, -1),
        window_shape=[1, 6],
        strides=[1, 6],
        pooling_type='AVG',
        padding='SAME')
    spectrogram = tf.squeeze(spectrogram, axis=0)
    spectrogram = np.log10(spectrogram + 1e-6)
    return spectrogram
```

Fig. 14 Converting audio signal into spectrogram

The get_spectrogram function normalizes an audio input, computes its spectrogram using a Short-Time Fourier Transform (STFT), and, in order to make the spectrogram much simpler to use, averaging pooled the frequency bins. The processed spectrogram is then returned after the dynamic range of the spectrogram data is compressed by a logarithmic transformation for better visualization.

The figure below shows how the code prepares audio files related to specific words for training a machine learning model.

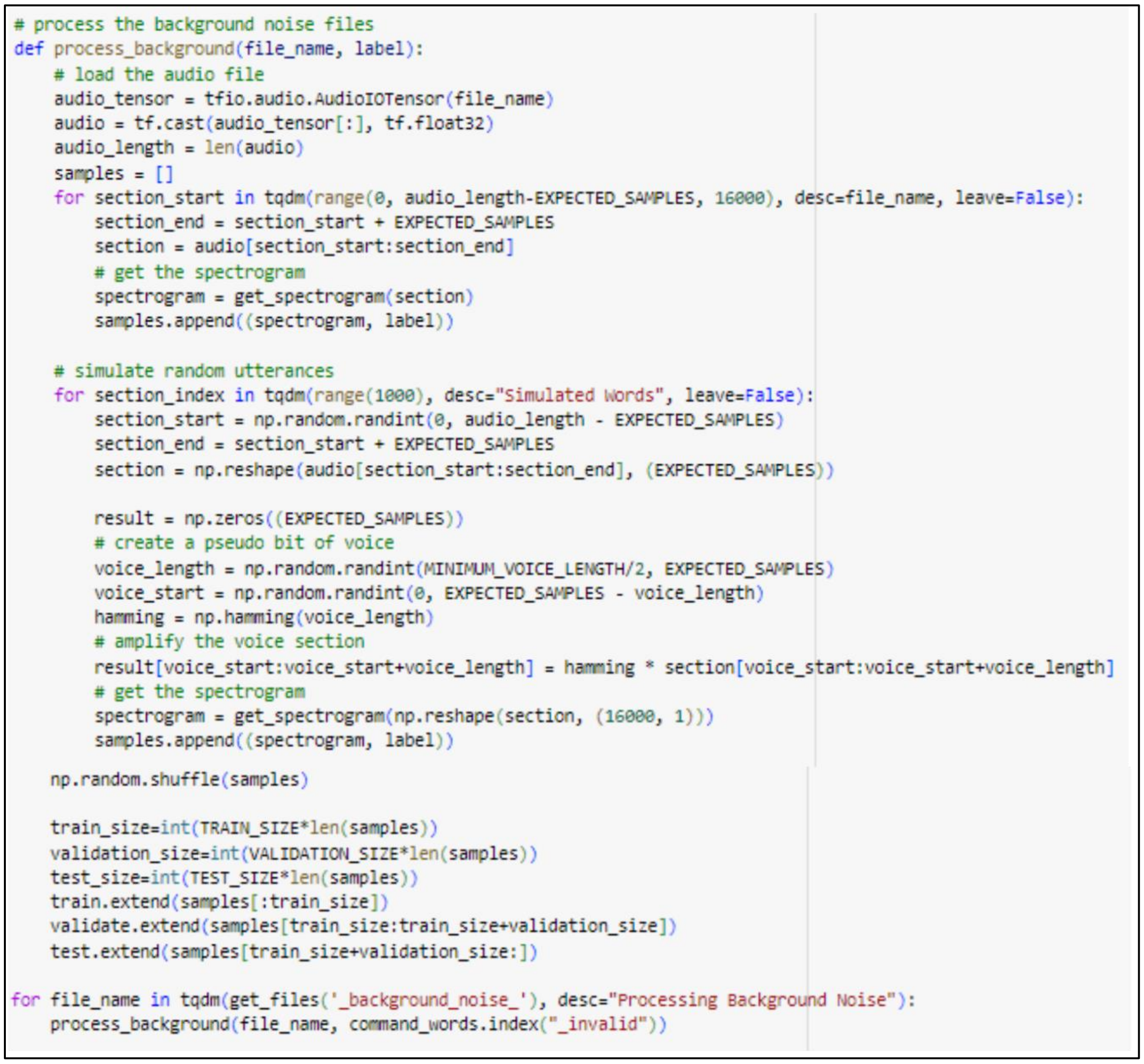


```python
def process_files(file_names, label, repeat=1):
    file_names = tf.repeat(file_names, repeat).numpy()
    return [(process_file(file_name), label) for file_name in tqdm(file_names, desc=f"{word} ({label})", leave=False)]

# process the files for a word into the spectrogram and one hot encoding word value
def process_word(word, label, repeat=1):
    # get a list of files names for the word
    file_names = [file_name for file_name in tqdm(get_files(word), desc="Checking", leave=False) if is_valid_file(file_name)
    # randomly shuffle the filenames
    np.random.shuffle(file_names)
    # split the files into train, validate and test buckets
    train_size=int(TRAIN_SIZE*len(file_names))
    validation_size=int(VALIDATION_SIZE*len(file_names))
    test_size=int(TEST_SIZE*len(file_names))
    # get the training samples
    train.extend(
        process_files(
            file_names[:train_size],
            label,
            repeat=repeat
        )
    )
    # and the validation samples
    validate.extend(
        process_files(
            file_names[train_size:train_size+validation_size],
            label,
            repeat=repeat
        )
    )
    # and the test samples
    test.extend(
        process_files(
            file_names[train_size+validation_size:],
            label,
            repeat=repeat
        )
    )

# process all the command words
for word in tqdm(command_words, desc="Processing words"):
    if '_' not in word:
        repeat = 40 if word in ('slow', 'left', 'right', 'stop', 'off') else 20
        process_word(word, command_words.index(word), repeat=repeat)

try:
  # all the nonsense words
  for word in tqdm(nonsense_words, desc="Processing words"):
      if '_' not in word:
          process_word(word, command_words.index('_invalid'), repeat=1)
except Exception as e:
  print(f'Error: {e}')

print(len(train), len(test), len(validate))
```

Fig. 15 Generating Training data

It organizes the files into training, validation, and test sets, and repeats the process for both command words and nonsense words. The goal is to have the data ready for training the model.

To train a command word recognition model, the figure below shows how to extract spectrogram samples, simulate random spoken utterances, and divide these samples into training, validation, and test datasets.

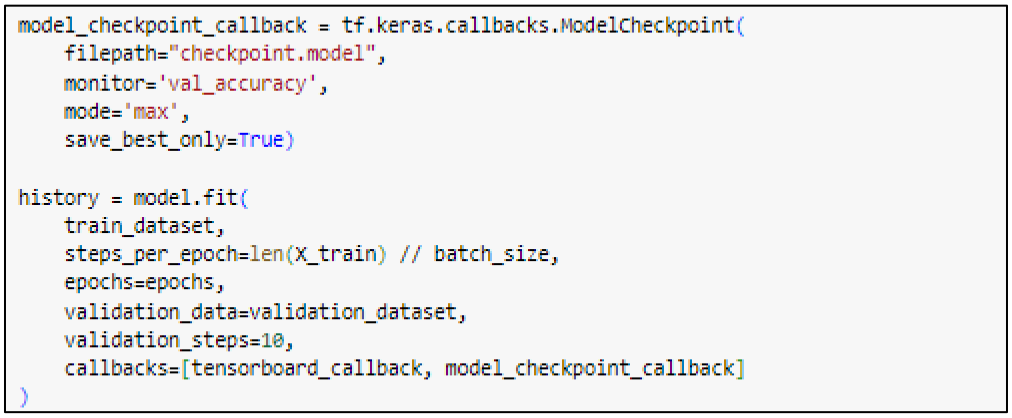


```
# process the background noise files
def process_background(file_name, label):
    # load the audio file
    audio_tensor = tfio.audio.AudioIOTensor(file_name)
    audio = tf.cast(audio_tensor[:], tf.float32)
    audio_length = len(audio)
    samples = []
    for section_start in tqdm(range(0, audio_length-EXPECTED_SAMPLES, 16000), desc=file_name, leave=False):
        section_end = section_start + EXPECTED_SAMPLES
        section = audio[section_start:section_end]
        # get the spectrogram
        spectrogram = get_spectrogram(section)
        samples.append((spectrogram, label))

    # simulate random utterances
    for section_index in tqdm(range(1000), desc="Simulated Words", leave=False):
        section_start = np.random.randint(0, audio_length - EXPECTED_SAMPLES)
        section_end = section_start + EXPECTED_SAMPLES
        section = np.reshape(audio[section_start:section_end], (EXPECTED_SAMPLES))

        result = np.zeros((EXPECTED_SAMPLES))
        # create a pseudo bit of voice
        voice_length = np.random.randint(MINIMUM_VOICE_LENGTH/2, EXPECTED_SAMPLES)
        voice_start = np.random.randint(0, EXPECTED_SAMPLES - voice_length)
        hamming = np.hamming(voice_length)
        # amplify the voice section
        result[voice_start:voice_start+voice_length] = hamming * section[voice_start:voice_start+voice_length]
        # get the spectrogram
        spectrogram = get_spectrogram(np.reshape(section, (16000, 1)))
        samples.append((spectrogram, label))

   np.random.shuffle(samples)

   train_size=int(TRAIN_SIZE*len(samples))
   validation_size=int(VALIDATION_SIZE*len(samples))
   test_size=int(TEST_SIZE*len(samples))
   train.extend(samples[:train_size])
   validate.extend(samples[train_size:train_size+validation_size])
   test.extend(samples[train_size+validation_size:])

for file_name in tqdm(get_files('_background_noise_'), desc="Processing Background Noise"):
   process_background(file_name, command_words.index("_invalid"))
```

Fig. 16 Processing background noise

Spectrogram samples for training a command word recognition model are produced by this function processing background noise audio files. Extracting portions of the audio, it creates spectrograms from simulated speech segments, randomly shuffles and separates the segments into training, validation, and test datasets. This ensures that a range of background noise conditions are trained into the model.

The figure below shows how to train the machine learning model using TensorFlow's Keras framework.

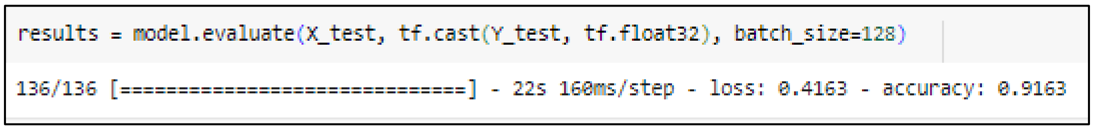


```
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath="checkpoint.model",
    monitor='val_accuracy',
    mode='max',
    save_best_only=True)

history = model.fit(
    train_dataset,
    steps_per_epoch=len(X_train) // batch_size,
    epochs=epochs,
    validation_data=validation_dataset,
    validation_steps=10,
    callbacks=[tensorboard_callback, model_checkpoint_callback]
)
```

Fig. 17 Training the model

It set up a ModelCheckpoint callback to save the best-performing model based on validation accuracy. Additionally, it uses TensorBoard to visualize the training process. This setup ensures efficient training and allows us to monitor and save the optimal model for future use.

The figure below shows how to test the created voice command recognition model.

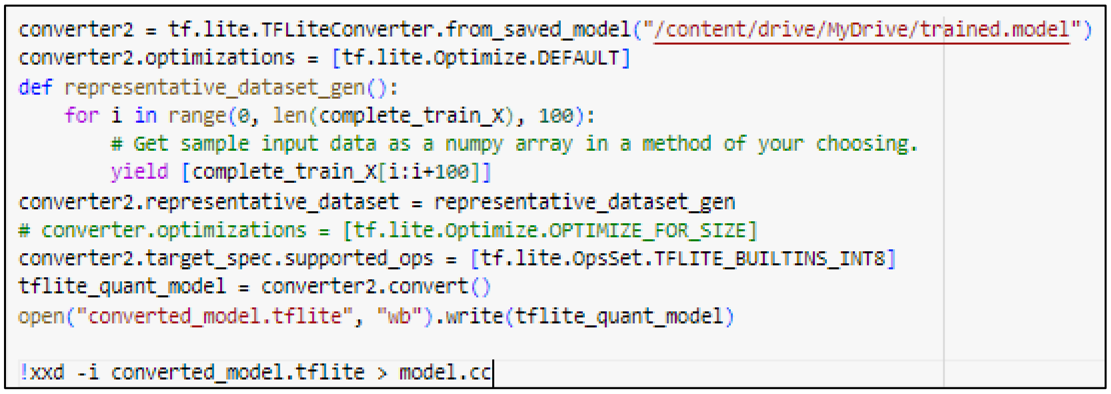


```
results = model.evaluate(X_test, tf.cast(Y_test, tf.float32), batch_size=128)

136/136 [==============================] - 22s 160ms/step - loss: 0.4163 - accuracy: 0.9163
```

Fig. 18 Testing the model

After training the model, the researchers evaluate the model with the accuracy of 91.63%.

The figure below shows the conversion of TensorFlow SavedModel to a TensorFlow Lite model for deployment.



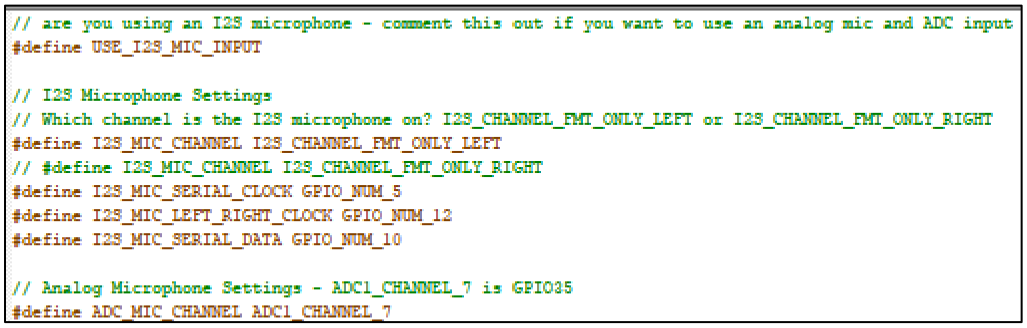
```
converter2 = tf.lite.TFLiteConverter.from_saved_model("/content/drive/MyDrive/trained.model")
converter2.optimizations = [tf.lite.Optimize.DEFAULT]
def representative_dataset_gen():
    for i in range(0, len(complete_train_X), 100):
        # Get sample input data as a numpy array in a method of your choosing.
        yield [complete_train_X[i:i+100]]
converter2.representative_dataset = representative_dataset_gen
# converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
converter2.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
tflite_quant_model = converter2.convert()
open("converted_model.tflite", "wb").write(tflite_quant_model)

!xxd -i converted_model.tflite > model.cc
```

Fig. 19 Converting the test model into C++ source code

It saves the converted TensorFlow Lite model to a file named "converted_model.tflite". Finally, it converts the file into C++ binary code that will be used to instruct the microcontroller.

The figure below shows the configuration of pins of the INMP441 MEMS microphone.

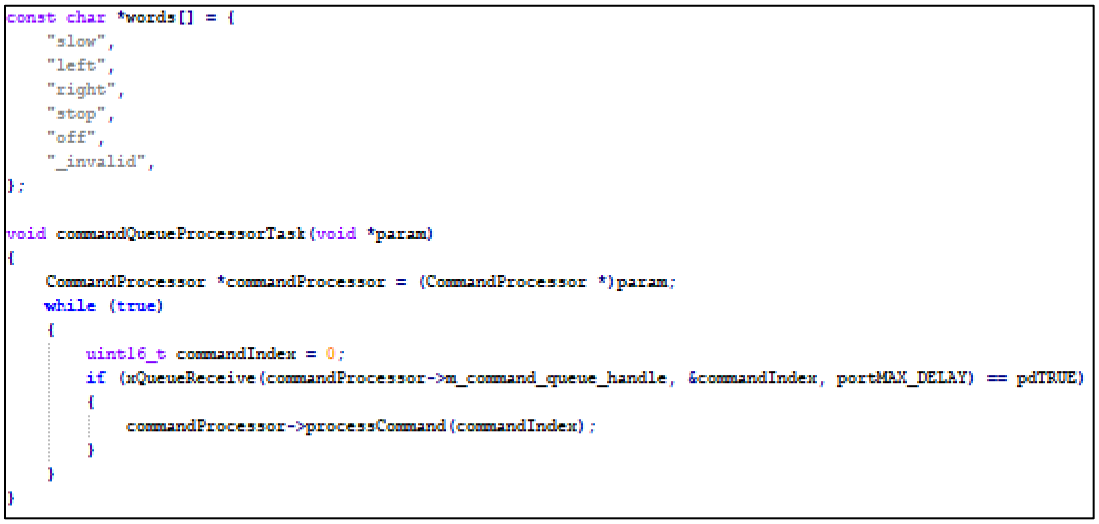
Fig. 20 config.h file



```
// are you using an I2S microphone - comment this out if you want to use an analog mic and ADC input
#define USE_I2S_MIC_INPUT

// I2S Microphone Settings
// Which channel is the I2S microphone on? I2S_CHANNEL_FMT_ONLY_LEFT or I2S_CHANNEL_FMT_ONLY_RIGHT
#define I2S_MIC_CHANNEL I2S_CHANNEL_FMT_ONLY_LEFT
// #define I2S_MIC_CHANNEL I2S_CHANNEL_FMT_ONLY_RIGHT
#define I2S_MIC_SERIAL_CLOCK GPIO_NUM_5
#define I2S_MIC_LEFT_RIGHT_CLOCK GPIO_NUM_12
#define I2S_MIC_SERIAL_DATA GPIO_NUM_10

// Analog Microphone Settings - ADC1_CHANNEL_7 is GPIO35
#define ADC_MIC_CHANNEL ADC1_CHANNEL_7
```

The pins are connected to GPIO_NUM_5 for SCK, GPIO_NUM_12 for WS, and GPIO_NUM_10 for DIN.

The figure below shows declaring specific commands such as ‘slow’, ‘left’, ‘right’, ‘slow’ and ‘off’.

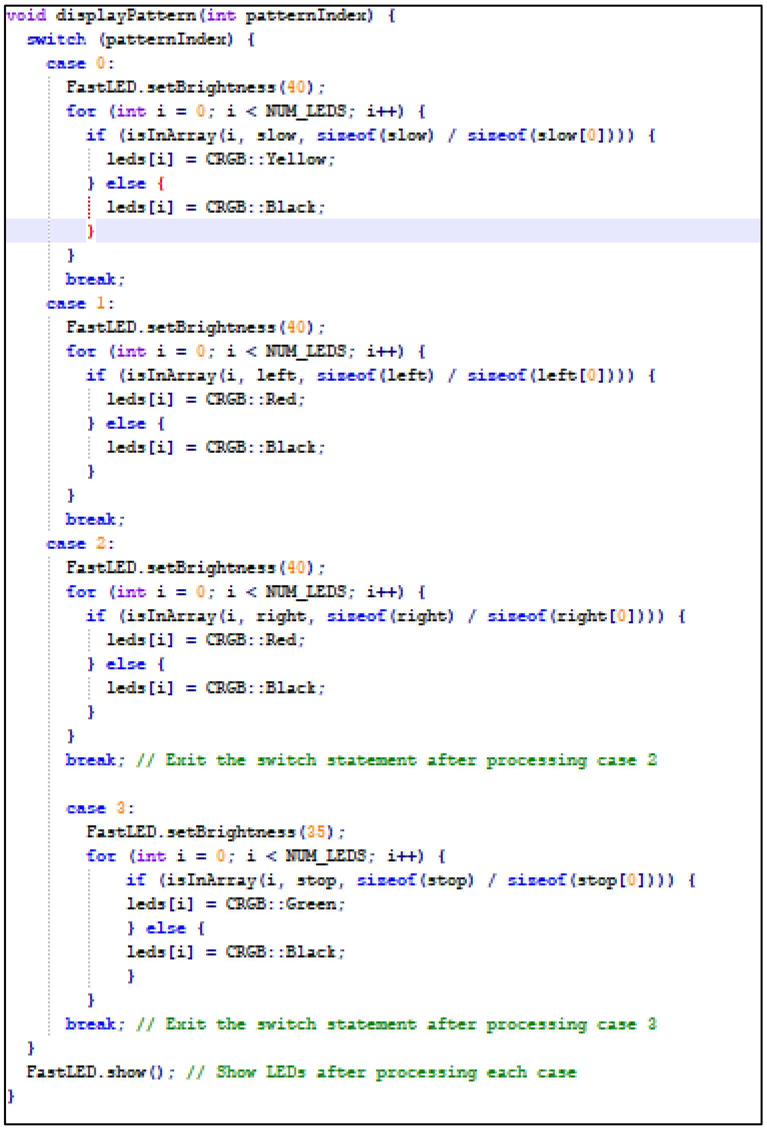


```
const char *words[] = {
    "slow",
    "left",
    "right",
    "stop",
    "off",
    "_invalid",
};

void commandQueueProcessorTask(void *param)
{
    CommandProcessor *commandProcessor = (CommandProcessor *)param;
    while (true)
    {
        uint16_t commandIndex = 0;
        if (xQueueReceive(commandProcessor->m_command_queue_handle, &commandIndex, portMAX_DELAY) == pdTRUE)
        {
            commandProcessor->processCommand(commandIndex);
        }
    }
}
```

Fig. 21 Command words

The code also defines a task function that is responsible for receiving commands from a queue and executes them by invoking the processCommand method in the CommandProcessor object. The program is cyclic and runs endlessly if there are commands to deal with. They are processed one at a time.

The figure below shows the function designed to switch between different predefined LED patterns.

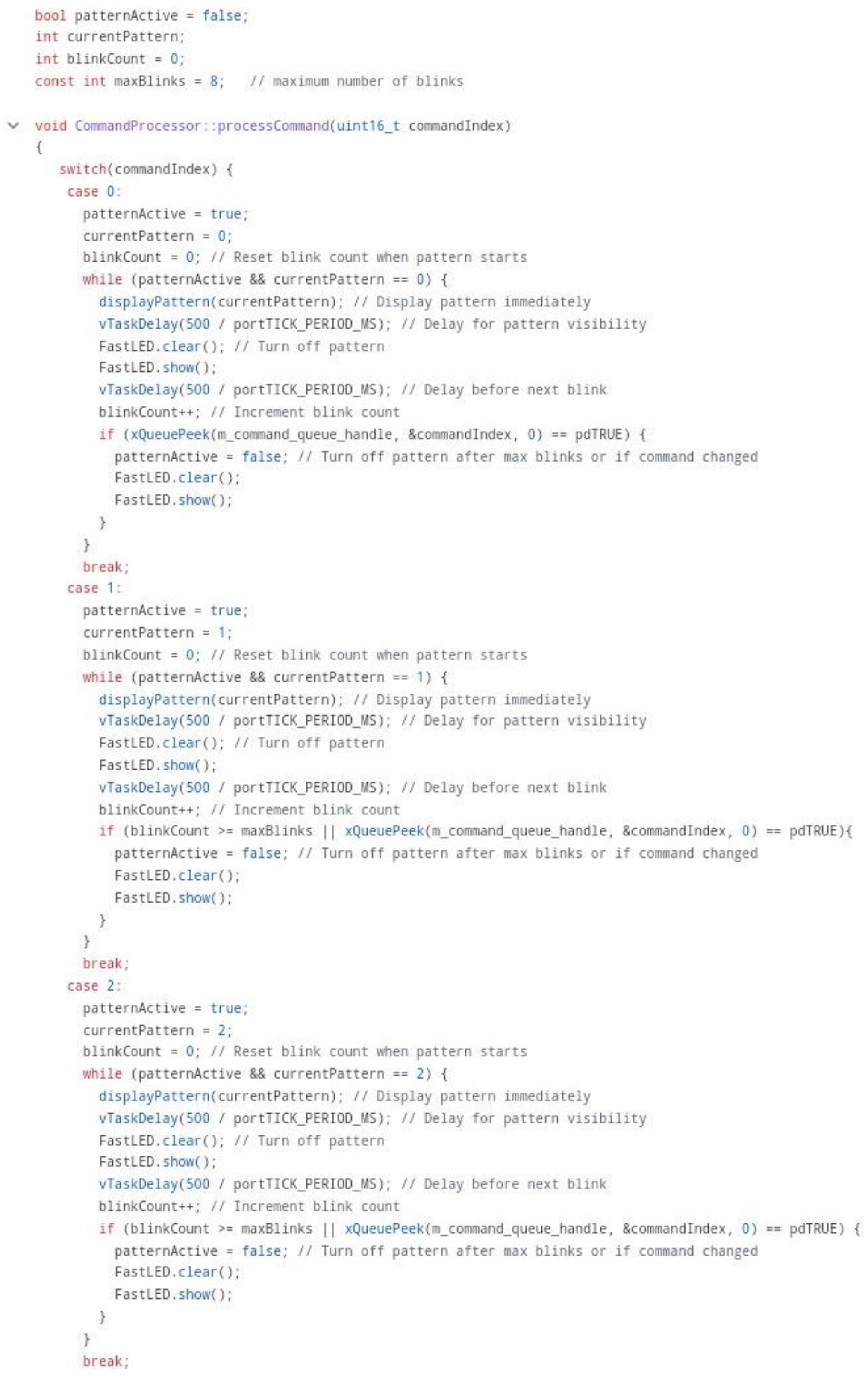


```
void displayPattern(int patternIndex) {
  switch (patternIndex) {
    case 0:
      FastLED.setBrightness(40);
      for (int i = 0; i < NUM_LEDS; i++) {
        if (isInArray(i, slow, sizeof(slow) / sizeof(slow[0]))) {
          leds[i] = CRGB::Yellow;
        } else {
          leds[i] = CRGB::Black;
        }
      }
      break;
    case 1:
      FastLED.setBrightness(40);
      for (int i = 0; i < NUM_LEDS; i++) {
        if (isInArray(i, left, sizeof(left) / sizeof(left[0]))) {
          leds[i] = CRGB::Red;
        } else {
          leds[i] = CRGB::Black;
        }
      }
      break;
    case 2:
      FastLED.setBrightness(40);
      for (int i = 0; i < NUM_LEDS; i++) {
        if (isInArray(i, right, sizeof(right) / sizeof(right[0]))) {
          leds[i] = CRGB::Red;
        } else {
          leds[i] = CRGB::Black;
        }
      }
      break; // Exit the switch statement after processing case 2

      case 3:
        FastLED.setBrightness(35);
        for (int i = 0; i < NUM_LEDS; i++) {
            if (isInArray(i, stop, sizeof(stop) / sizeof(stop[0]))) {
            leds[i] = CRGB::Green;
            } else {
            leds[i] = CRGB::Black;
            }
        }
      break; // Exit the switch statement after processing case 3
  }
  FastLED.show(); // Show LEDs after processing each case
}
```

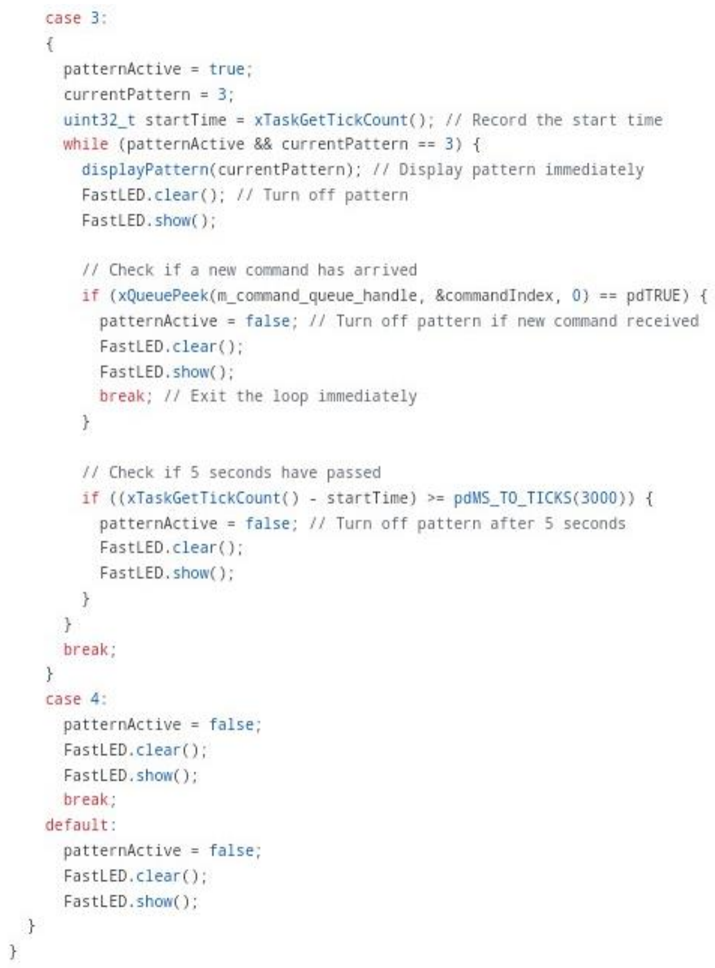
Fig. 22 Display pattern function

It operates based on a variable called patternIndex, which determines the current LED pattern to display. The function switches between different predefined LED patterns and updates the LEDs accordingly.

The figure below shows a function that offers a detailed control system for LED patterns using different command indices.



```
bool patternActive = false;
int currentPattern;
int blinkCount = 0;
const int maxBlinks = 8;   // maximum number of blinks

void CommandProcessor::processCommand(uint16_t commandIndex)
{
   switch(commandIndex) {
    case 0:
      patternActive = true;
      currentPattern = 0;
      blinkCount = 0; // Reset blink count when pattern starts
      while (patternActive && currentPattern == 0) {
        displayPattern(currentPattern); // Display pattern immediately
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay for pattern visibility
        FastLED.clear(); // Turn off pattern
        FastLED.show();
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay before next blink
        blinkCount++; // Increment blink count
        if (xQueuePeek(m_command_queue_handle, &commandIndex, 0) == pdTRUE) {
          patternActive = false; // Turn off pattern after max blinks or if command changed
          FastLED.clear();
          FastLED.show();
        }
      }
      break;
    case 1:
      patternActive = true;
      currentPattern = 1;
      blinkCount = 0; // Reset blink count when pattern starts
      while (patternActive && currentPattern == 1) {
        displayPattern(currentPattern); // Display pattern immediately
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay for pattern visibility
        FastLED.clear(); // Turn off pattern
        FastLED.show();
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay before next blink
        blinkCount++; // Increment blink count
        if (blinkCount >= maxBlinks || xQueuePeek(m_command_queue_handle, &commandIndex, 0) == pdTRUE){
          patternActive = false; // Turn off pattern after max blinks or if command changed
          FastLED.clear();
          FastLED.show();
        }
      }
      break;
    case 2:
      patternActive = true;
      currentPattern = 2;
      blinkCount = 0; // Reset blink count when pattern starts
      while (patternActive && currentPattern == 2) {
        displayPattern(currentPattern); // Display pattern immediately
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay for pattern visibility
        FastLED.clear(); // Turn off pattern
        FastLED.show();
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay before next blink
        blinkCount++; // Increment blink count
        if (blinkCount >= maxBlinks || xQueuePeek(m_command_queue_handle, &commandIndex, 0) == pdTRUE) {
          patternActive = false; // Turn off pattern after max blinks or if command changed
          FastLED.clear();
          FastLED.show();
        }
      }
      break;
```

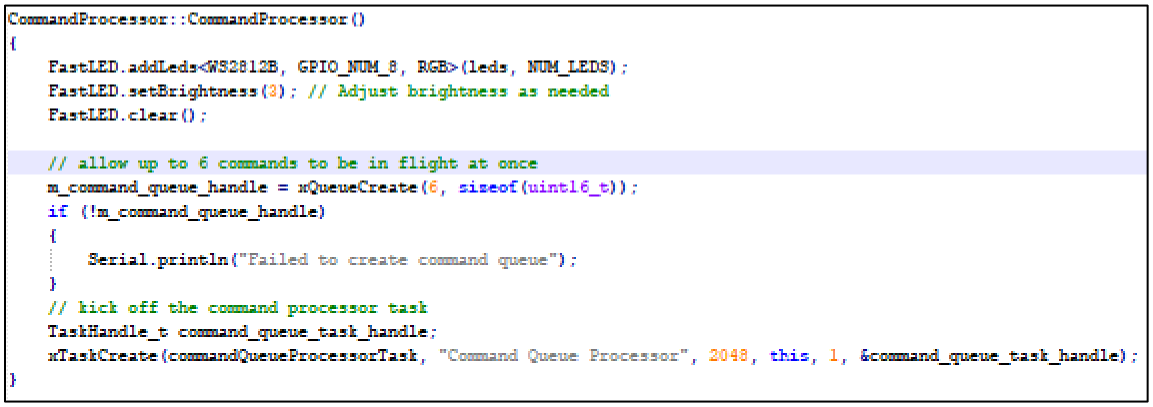


```
    case 3:
    {
      patternActive = true;
      currentPattern = 3;
      uint32_t startTime = xTaskGetTickCount(); // Record the start time
      while (patternActive && currentPattern == 3) {
        displayPattern(currentPattern); // Display pattern immediately
        FastLED.clear(); // Turn off pattern
        FastLED.show();

        // Check if a new command has arrived
        if (xQueuePeek(m_command_queue_handle, &commandIndex, 0) == pdTRUE) {
          patternActive = false; // Turn off pattern if new command received
          FastLED.clear();
          FastLED.show();
          break; // Exit the loop immediately
        }

        // Check if 5 seconds have passed
        if ((xTaskGetTickCount() - startTime) >= pdMS_TO_TICKS(3000)) {
          patternActive = false; // Turn off pattern after 5 seconds
          FastLED.clear();
          FastLED.show();
        }
      }
      break;
    }
    case 4:
      patternActive = false;
      FastLED.clear();
      FastLED.show();
      break;
    default:
      patternActive = false;
      FastLED.clear();
      FastLED.show();
  }
}
```

Fig. 23 Function to display pattern based on command index

This code specifies how to handle different LED pattern display. It controls cases such limited blink counts and timed patterns, activates patterns according to the command index, and delays their display. When needed, it disables patterns and responds to a command to turn off all LED light.

The figure below shows the setup of a CommandProcessor object, initializing an LED strip and creating a command queue for handling commands.



```
CommandProcessor::CommandProcessor()
{
    FastLED.addLeds<WS2812B, GPIO_NUM_8, RGB>(leds, NUM_LEDS);
    FastLED.setBrightness(3); // Adjust brightness as needed
    FastLED.clear();

    // allow up to 6 commands to be in flight at once
    m_command_queue_handle = xQueueCreate(6, sizeof(uint16_t));
    if (!m_command_queue_handle)
    {
        Serial.println("Failed to create command queue");
    }
    // kick off the command processor task
    TaskHandle_t command_queue_task_handle;
    xTaskCreate(commandQueueProcessorTask, "Command Queue Processor", 2048, this, 1, &command_queue_task_handle);
}
```

Fig. 24 Pin setup for LED Matrix

This code sets up a CommandProcessor object by initializing the LED strip, creating a command queue, and starting a task to process commands asynchronously. If the command queue creation fails, it prints an error message.

Add on the Project Evaluation part

The table 10 shows the results of a test on the efficiency of response based on the distance of the user. The user 1 was close to the microphone while the user 2 was far from the microphone with a loud voice input command.

Table 10. Results of Response of the Signal Light Displayed based on distance of the user

Trial	User 1 Input Command (Close from Microphone)	User 2 Input Command (Far from Microphone)	Response (Signal Light Displayed)
1	Left	Off	Left
2	Right	Stop	Right
3	Slow	Right	Slow

4	Stop	Left	Stop
5	Off	Slow	Off
6	Left	Off	Left
7	Right	Stop	Right
8	Slow	Right	Slow
9	Stop	Left	Stop
10	Off	Slow	Off

Based on the result above, the test has 10 trials with two users involved. After testing all the trials the device is responding efficiently on the user 1 which is close from the microphone with 10 out of 10 trials.

The table 11 shows the result of a test on the efficiency of response based on 5 consecutive input commands of the user. This test was to determine what will be the response of the device.

Table 11. Results of Response of the Signal Light Displayed based on 5 consecutive input commands

Trial	**5 Consecutive Input Command**	**Response (Signal Light Displayed)**
1	Left, Right, Slow, Stop,Off	Left, Right, Stop
2	Right, Slow, Stop, Off, Left	Right, Stop, Left
3	Slow, Stop, Off, Left, Right	Slow, Left
4	Stop, Off, Left, Right, Slow	Stop, Right
5	Off, Left, Right, Slow, Stop	Off, Right, Stop
6	Left, Right, Slow, Stop,Off	Left, Right
7	Right, Slow, Stop, Off, Left	Right, Stop

8	Slow, Stop, Off, Left, Right	Slow, Right
9	Stop, Off, Left, Right, Slow	Stop, Right
10	Off, Left, Right, Slow, Stop	Off, Left, Right

Based on the result above, it indicates that the device is not efficient for consecutive commands. In 4 out of 10 trials, only three commands were recognized; in 6 out of 10 trials, only two commands were recognized; this is because the device has a delay in gathering the input and processing by the model.