National Institute of Technology, Warangal

Department of Electronics and Communication Engineering

November,2024.

Project Report on

"Keyword Spotting using Arduino BLE 33 Sense and Tensorflow Lite"

Concluding project Digital Signal Processing course by:

Rohan Karampuri (22ECB0B28)

AR Sai Thirilok (22ECB0F21)

Supervised by:

Sri. Ravi Kumar Jatoth

Professor, ECE NITW

Introduction

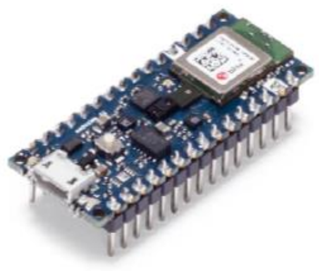
This project report is based implementation of Keyword Spotting on the Arduino Nano 33 BLE Sense using TensorFlow Lite (TensorFlow Lite Micro). Keyword spotting is a crucial feature in many voice-activated applications, allowing devices to recognize specific commands or words from audio input, enabling hands-free control. This is particularly useful for embedded systems and microcontrollers that lack the processing power of traditional computers.

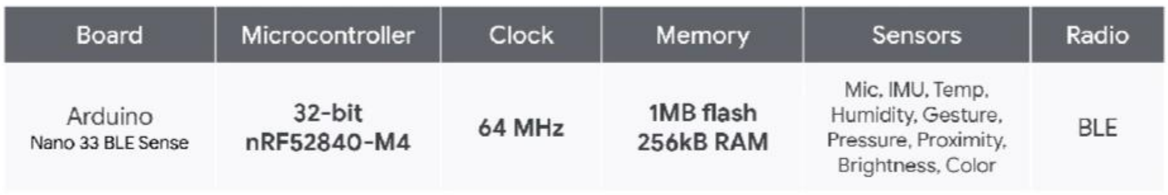
In this project, the Arduino Nano 33 BLE Sense, a compact microcontroller with Bluetooth capabilities, is used to perform speech recognition. TensorFlow Lite Micro, a lightweight version of Google's machine learning framework, is optimized for running models on resource-constrained devices like microcontrollers. The system leverages a pre-trained neural network model for speech recognition and performs real-time audio processing to detect specific keywords or commands.

This implementation demonstrates the integration of several key components:

- Audio data acquisition using onboard microphone of Arduino Nano 33 BLE.
- Feature extraction from the audio signals (spectrograms) to prepare the data for processing.
- Inference execution using a TensorFlow Lite model optimized for microcontrollers.
- Command recognition based on the model's output using on board LEDs.

By the end of this project, we demonstrate how TensorFlow Lite Micro can be used on the Arduino Nano 33 BLE Sense for keyword spotting. This implementation showcases real-time voice command recognition on a low-power, low-memory microcontroller, covering key steps like audio data collection, feature extraction, model inference, and command recognition for voice-activated applications.





Board	Microcontroller	Clock	Memory	Sensors	Radio
Arduino Nano 33 BLE Sense	32-bit nRF52840-M4	64 MHz	1MB flash 256kB RAM	Mic, IMU, Temp, Humidity, Gesture, Pressure, Proximity, Brightness, Color	BLE

About the Microcontroller

Processor : Nordic Semiconductor nRF52840 SoC with ARM Cortex-M4 core, 64 MHz clock speed.

Bluetooth Low Energy (BLE) : Bluetooth 5.0 for wireless communication with up to 100 meters range.

Memory : 256KB Flash and **32KB SRAM** for basic programs and small models.

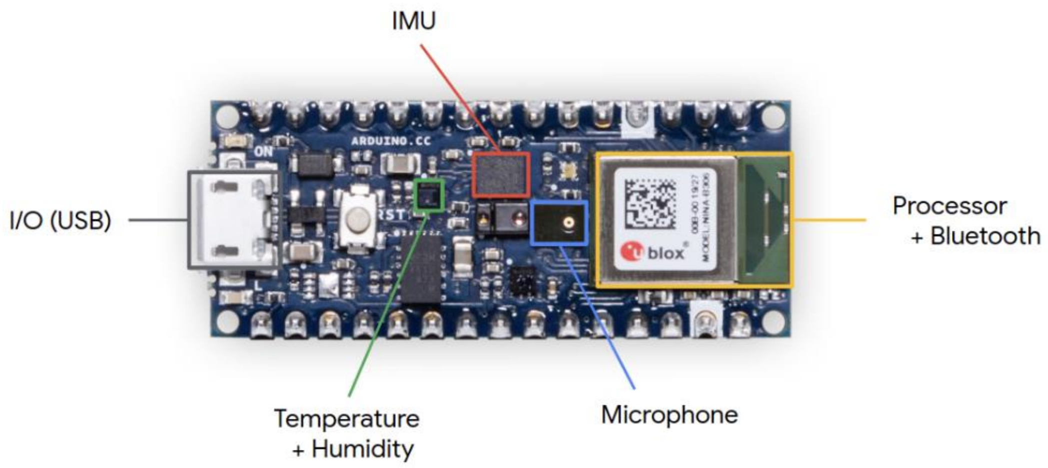
Onboard Sensors:

- **9-axis IMU** (accelerometer, gyroscope, magnetometer) for motion and orientation sensing.
- **BME680** (temperature, humidity, pressure, gas sensor) for environmental sensing.
- **MEMS Microphone** for sound capture and audio-based applications.
- **APDS-9960** for gesture and proximity detection.
- **TCS3200** for color detection and proximity sensing.

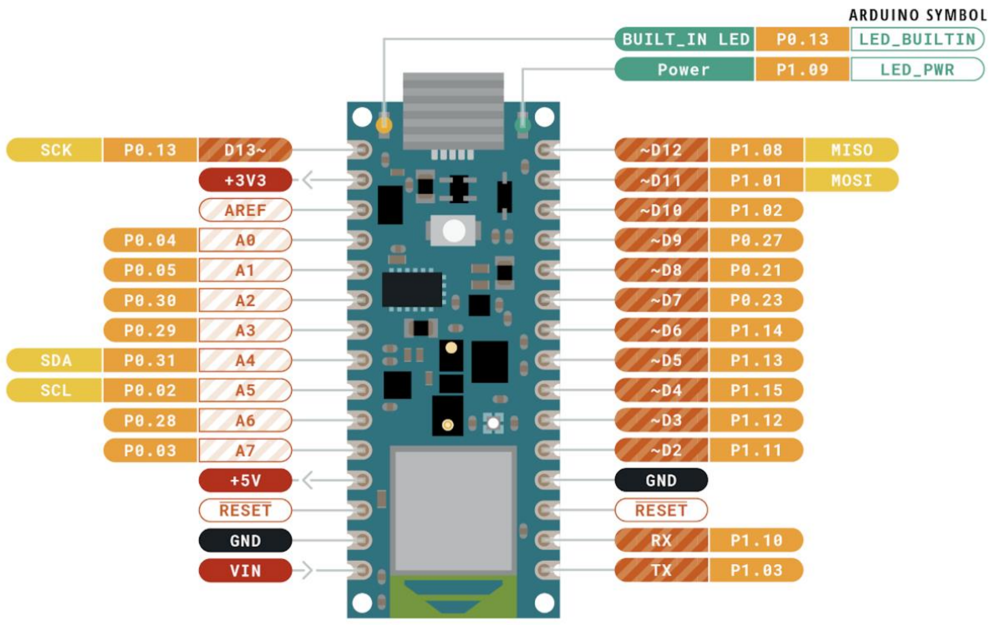
Camera Interface : Supports **OV7670 Camera Module** for visual data capture.

Power Supply : 3.3V operating voltage with **5V input** via USB or battery.

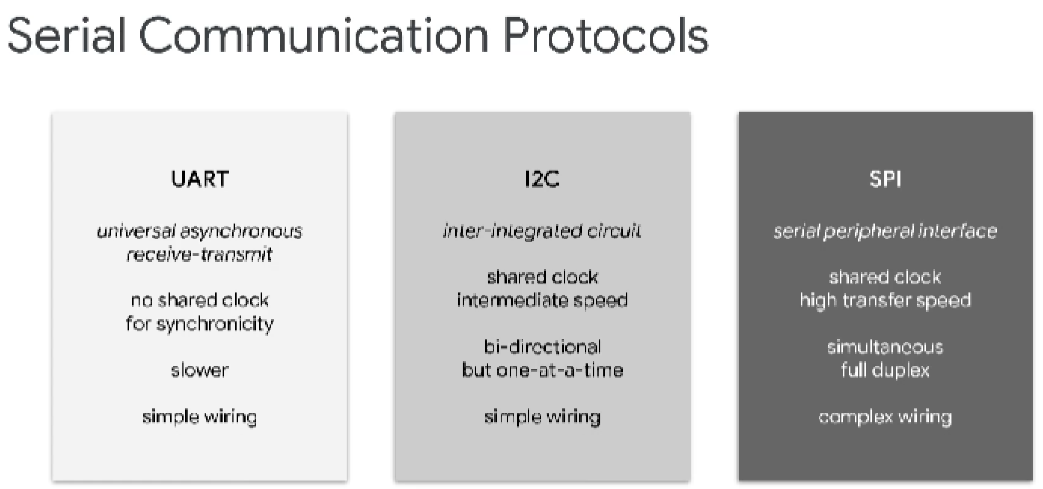
I/O Pins : 14 Digital I/O Pins (11 PWM), **8 Analog Pins**, with UART, I2C, and SPI interfaces



Arduino BLE Pin configuration:



Tiny Machine Learning Kit:



Serial Communication Protocols

UART	I2C	SPI
universal asynchronous receive-transmit	*inter-integrated circuit*	*serial peripheral interface*
no shared clock for synchronicity	shared clock intermediate speed	shared clock high transfer speed
slower	bi-directional but one-at-a-time	simultaneous full duplex
simple wiring	simple wiring	complex wiring

TensorFlow Lite and TensorFlow Lite Micro:

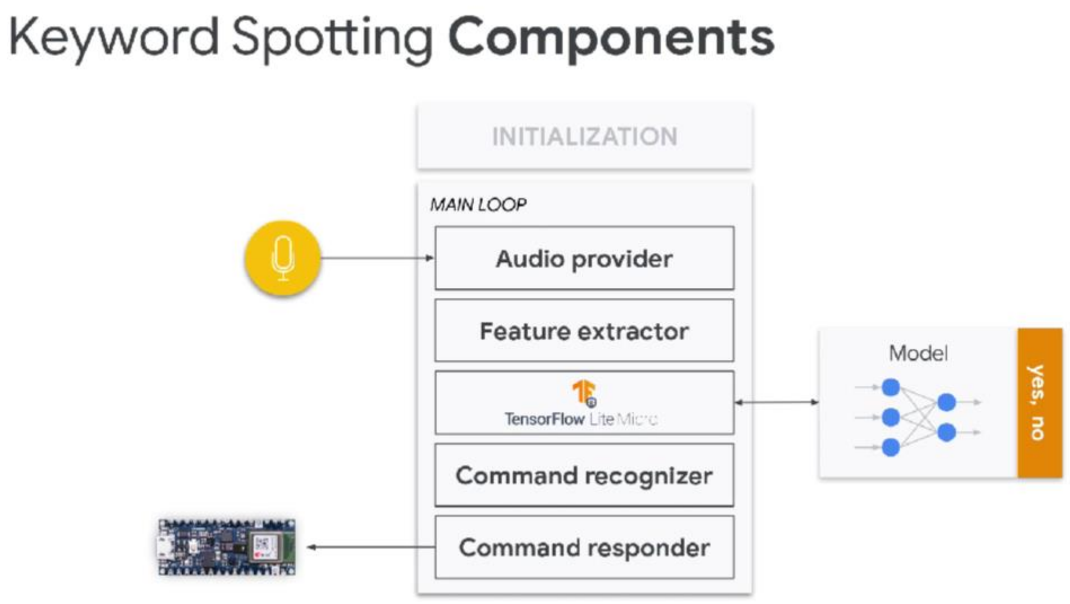
TensorFlow Lite (TFLite) incorporates quantization techniques to optimize machine learning models for resource-constrained environments, such as microcontrollers. **Quantization reduces the precision of model weights and activations from 32-bit floating-point to lower-precision formats like 8-bit integers**. This significantly decreases the model's size and computational requirements, while maintaining acceptable accuracy for many applications.

In TensorFlow Lite Micro, quantization is crucial for running models on devices with extremely limited memory and processing power, like the Arduino BLE Sense. For keyword spotting, quantized models are used to process audio inputs (e.g., spectrograms or MFCCs) efficiently. The neural network weights, activations, and operations are quantized to integer formats, allowing the model to perform inference with reduced power consumption and memory usage.

This enables real-time processing of audio data for recognizing specific keywords with minimal resource usage. Quantization not only allows TensorFlow Lite Micro to fit neural networks in constrained environments but also accelerates computation, making it ideal for low-power, always-on applications like keyword spotting on IoT and wearable devices.

High power
High bandwidth
High latency
Google Assistant
Low power
Low bandwidth
Low latency

Code



1. Arduino Main INO file

```
#include <TensorFlowLite.h>
#include "audio_provider.h"
#include "command_responder.h"
#include "feature_provider.h"
#include "main_functions.h"
#include "micro_features_micro_model_settings.h"
#include "micro_features_model.h"
#include "recognize_commands.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_log.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"

#undef PROFILE_MICRO_SPEECH

namespace {
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;
TfLiteTensor* model_input = nullptr;
FeatureProvider* feature_provider = nullptr;
RecognizeCommands* recognizer = nullptr;
int32_t previous_time = 0;

constexpr int kTensorArenaSize = 10 * 1024;

alignas(16) uint8_t tensor_arena[kTensorArenaSize];
int8_t feature_buffer[kFeatureElementCount];
int8_t* model_input_buffer = nullptr;
}
```

```
void setup() {
  tflite::InitializeTarget();

  model = tflite::GetModel(g_model);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    MicroPrintf(
        "Model provided is schema version %d not equal "
        "to supported version %d.",
        model->version(), TFLITE_SCHEMA_VERSION);
    return;
  }

  static tflite::MicroMutableOpResolver<4> micro_op_resolver;
  if (micro_op_resolver.AddDepthwiseConv2D() != kTfLiteOk) {
    return;
  }
  if (micro_op_resolver.AddFullyConnected() != kTfLiteOk) {
    return;
  }
  if (micro_op_resolver.AddSoftmax() != kTfLiteOk) {
    return;
  }
  if (micro_op_resolver.AddReshape() != kTfLiteOk) {
    return;
  }

  static tflite::MicroInterpreter static_interpreter(
      model, micro_op_resolver, tensor_arena, kTensorArenaSize);
  interpreter = &static_interpreter;

  TfLiteStatus allocate_status = interpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    MicroPrintf("AllocateTensors() failed");
    return;
  }

  model_input = interpreter->input(0);
  if ((model_input->dims->size != 2) || (model_input->dims->data[0] != 1) ||
      (model_input->dims->data[1] !=
       (kFeatureSliceCount * kFeatureSliceSize)) ||
      (model_input->type != kTfLiteInt8)) {
    MicroPrintf("Bad input tensor parameters in model");
    return;
  }
  model_input_buffer = model_input->data.int8;

  // Prepare to access the audio spectrograms from a microphone
  // that will provide the inputs to the neural network.
  static FeatureProvider static_feature_provider(kFeatureElementCount,
                                                 feature_buffer);
  feature_provider = &static_feature_provider;
  static RecognizeCommands static_recognizer;
  recognizer = &static_recognizer;
  previous_time = 0;

  TfLiteStatus init_status = InitAudioRecording();
  if (init_status != kTfLiteOk) {
    MicroPrintf("Unable to initialize audio");
    return;
  }
  MicroPrintf("Initialization complete");
}
void loop() {
#ifdef PROFILE_MICRO_SPEECH
  const uint32_t prof_start = millis();
  static uint32_t prof_count = 0;
  static uint32_t prof_sum = 0;
  static uint32_t prof_min = std::numeric_limits<uint32_t>::max();
  static uint32_t prof_max = 0;
#endif
  // Fetch the spectrogram for the current time.
  const int32_t current_time = LatestAudioTimestamp();
  int how_many_new_slices = 0;
  TfLiteStatus feature_status = feature_provider->PopulateFeatureData(
      previous_time, current_time, &how_many_new_slices);
  if (feature_status != kTfLiteOk) {
    MicroPrintf("Feature generation failed");
    return;
  }
  previous_time += how_many_new_slices * kFeatureSliceStrideMs;
  // If no new audio samples have been received since last time, don't bother
  if (how_many_new_slices == 0) {
    return;
  }

  // Copy feature buffer to input tensor
  for (int i = 0; i < kFeatureElementCount; i++) {
    model_input_buffer[i] = feature_buffer[i];
  }

  // Run the model on the spectrogram input and make sure it succeeds.
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    MicroPrintf("Invoke failed");
    return;
  }

  // Obtain a pointer to the output tensor
  TfLiteTensor* output = interpreter->output(0);
  // Determine whether a command was recognized based on the output of inference
  const char* found_command = nullptr;
  uint8_t score = 0;
  bool is_new_command = false;
  TfLiteStatus process_status = recognizer->ProcessLatestResults(
      output, current_time, &found_command, &score, &is_new_command);
  if (process_status != kTfLiteOk) {
    MicroPrintf("RecognizeCommands::ProcessLatestResults() failed");
    return;
  }

  RespondToCommand(current_time, found_command, score, is_new_command);
```

```
#ifdef PROFILE_MICRO_SPEECH
  const uint32_t prof_end = millis();
  if (++prof_count > 10) {
    uint32_t elapsed = prof_end - prof_start;
    prof_sum += elapsed;
    if (elapsed < prof_min) {
      prof_min = elapsed;
    }
    if (elapsed > prof_max) {
      prof_max = elapsed;
    }
    if (prof_count % 300 == 0) {
      MicroPrintf("## time: min %dms  max %dms  avg %dms", prof_min, prof_max,
                  prof_sum / prof_count);
    }
  }
#endif
}
```

2. Initialization

```
#ifndef
TENSORFLOW_LITE_MICRO_EXAMPLES_MICRO_SPEECH_MICRO_FEATURES_MICRO
_MODEL_SETTINGS_H_
#define
TENSORFLOW_LITE_MICRO_EXAMPLES_MICRO_SPEECH_MICRO_FEATURES_MICRO
_MODEL_SETTINGS_H_

constexpr int kMaxAudioSampleSize = 512;
constexpr int kAudioSampleFrequency = 16000;

constexpr int kFeatureSliceSize = 40;
constexpr int kFeatureSliceCount = 49;
constexpr int kFeatureElementCount = (kFeatureSliceSize * kFeatureSliceCount);
constexpr int kFeatureSliceStrideMs = 20;
constexpr int kFeatureSliceDurationMs = 30;

constexpr int kSilenceIndex = 0;
constexpr int kUnknownIndex = 1;
constexpr int kCategoryCount = 4;
extern const char* kCategoryLabels[kCategoryCount];

#endif
```

3. Keywords

```
#include "micro_features_micro_model_settings.h"

const char* kCategoryLabels[kCategoryCount] = {
    "silence",
    "unknown",
    "yes",
    "no",
};
```

4. Audio Provider Module

```
#if defined(ARDUINO) && !defined(ARDUINO_ARDUINO_NANO33BLE)
#define ARDUINO_EXCLUDE_CODE
#endif
#ifndef ARDUINO_EXCLUDE_CODE
#include <algorithm>
#include <cmath>
#include "PDM.h"
#include "audio_provider.h"
#include "micro_features_micro_model_settings.h"
#include "tensorflow/lite/micro/micro_log.h"
#include "test_over_serial/test_over_serial.h"
using namespace test_over_serial;
namespace {
bool g_is_audio_initialized = false;
constexpr int kAudioCaptureBufferSize = DEFAULT_PDM_BUFFER_SIZE * 16;
int16_t g_audio_capture_buffer[kAudioCaptureBufferSize];
int16_t g_audio_output_buffer[kMaxAudioSampleSize];
volatile int32_t g_latest_audio_timestamp = 0;
uint32_t g_test_sample_index;
bool g_test_insert_silence = true;
}
void CaptureSamples() {
  const int number_of_samples = DEFAULT_PDM_BUFFER_SIZE / 2;
  const int32_t time_in_ms =
      g_latest_audio_timestamp +
      (number_of_samples / (kAudioSampleFrequency / 1000));
  const int32_t start_sample_offset =
      g_latest_audio_timestamp * (kAudioSampleFrequency / 1000);
  const int capture_index = start_sample_offset % kAudioCaptureBufferSize;
  int num_read =
      PDM.read(g_audio_capture_buffer + capture_index, DEFAULT_PDM_BUFFER_SIZE);
  if (num_read != DEFAULT_PDM_BUFFER_SIZE) {
    MicroPrintf("### short read (%d/%d) @%dms", num_read,
                DEFAULT_PDM_BUFFER_SIZE, time_in_ms);
    while (true) {
    }
  }
  g_latest_audio_timestamp = time_in_ms;
}

TfLiteStatus InitAudioRecording() {
  if (!g_is_audio_initialized) {
    PDM.onReceive(CaptureSamples);
    PDM.begin(1, kAudioSampleFrequency);
    PDM.setGain(13);
    while (!g_latest_audio_timestamp) {
    }
    g_is_audio_initialized = true;
  }
  return kTfLiteOk;
}

TfLiteStatus GetAudioSamples(int start_ms, int duration_ms,
                          int* audio_samples_size, int16_t** audio_samples) {

  const int start_offset = start_ms * (kAudioSampleFrequency / 1000);
  const int duration_sample_count =
      duration_ms * (kAudioSampleFrequency / 1000);
  for (int i = 0; i < duration_sample_count; ++i) {
    const int capture_index = (start_offset + i) % kAudioCaptureBufferSize;
    g_audio_output_buffer[i] = g_audio_capture_buffer[capture_index];
  }

  *audio_samples_size = duration_sample_count;
  *audio_samples = g_audio_output_buffer;

  return kTfLiteOk;
}
namespace {
void InsertSilence(const size_t len, int16_t value) {
  for (size_t i = 0; i < len; i++) {
    const size_t index = (g_test_sample_index + i) % kAudioCaptureBufferSize;
    g_audio_capture_buffer[index] = value;
  }
  g_test_sample_index += len;
}

int32_t ProcessTestInput(TestOverSerial& test) {
  constexpr size_t samples_16ms = ((kAudioSampleFrequency / 1000) * 16);

  InputHandler handler = [](const InputBuffer* const input) {
    if (0 == input->offset) {
      g_test_insert_silence = false;
    }

    for (size_t i = 0; i < input->length; i++) {
      const size_t index = (g_test_sample_index + i) % kAudioCaptureBufferSize;
      g_audio_capture_buffer[index] = input->data.int16[i];
    }
    g_test_sample_index += input->length;

    if (input->total == (input->offset + input->length)) {
      g_test_insert_silence = true;
    }
    return true;
  };

  test.ProcessInput(&handler);

  if (g_test_insert_silence) {
    InsertSilence(samples_16ms, 0);
  }
  g_latest_audio_timestamp = (g_test_sample_index / (samples_16ms * 4)) * 64;
  return g_latest_audio_timestamp;
}
}
int32_t LatestAudioTimestamp() {
  TestOverSerial& test = TestOverSerial::Instance(kAUDIO_PCM_16KHZ_MONO_S16);
  if (!test.IsTestMode()) {
    test.ProcessInput(nullptr);
  }
  if (test.IsTestMode()) {
    if (g_is_audio_initialized) {
      PDM.end();
      g_is_audio_initialized = false;
      g_test_sample_index =
          g_latest_audio_timestamp * (kAudioSampleFrequency / 1000);
    }
    return ProcessTestInput(test);
  } else {
    return g_latest_audio_timestamp;
  }
}
#endif
```

5. Feature Provider

```
#include "feature_provider.h"
#include "audio_provider.h"
#include "micro_features_micro_features_generator.h"
#include "micro_features_micro_model_settings.h"
#include "tensorflow/lite/micro/micro_log.h"

FeatureProvider::FeatureProvider(int feature_size, int8_t* feature_data)
    : feature_size_(feature_size),
      feature_data_(feature_data),
      is_first_run_(true) {
  for (int n = 0; n < feature_size_; ++n) {
    feature_data_[n] = 0;
  }
}
FeatureProvider::~FeatureProvider() {}
TfLiteStatus FeatureProvider::PopulateFeatureData(int32_t last_time_in_ms,
                                                  int32_t time_in_ms,
                                                  int* how_many_new_slices) {
  if (feature_size_ != kFeatureElementCount) {
    MicroPrintf("Requested feature_data_ size %d doesn't match %d",
                feature_size_, kFeatureElementCount);
    return kTfLiteError;
  }
  const int last_step = (last_time_in_ms / kFeatureSliceStrideMs);
  int slices_needed =
      ((((time_in_ms - last_time_in_ms) - kFeatureSliceDurationMs) *
        kFeatureSliceStrideMs) /
           kFeatureSliceStrideMs +
       kFeatureSliceStrideMs) /
      kFeatureSliceStrideMs;
  if (is_first_run_) {
    TfLiteStatus init_status = InitializeMicroFeatures();
    if (init_status != kTfLiteOk) {
      return init_status;
    }
    is_first_run_ = false;
    return kTfLiteOk;
  }
  if (slices_needed > kFeatureSliceCount) {
    slices_needed = kFeatureSliceCount;
  }
  if (slices_needed == 0) {
    return kTfLiteOk;
  }
  *how_many_new_slices = slices_needed;

  const int slices_to_keep = kFeatureSliceCount - slices_needed;
  const int slices_to_drop = kFeatureSliceCount - slices_to_keep;
  if (slices_to_keep > 0) {
    for (int dest_slice = 0; dest_slice < slices_to_keep; ++dest_slice) {
      int8_t* dest_slice_data =
          feature_data_ + (dest_slice * kFeatureSliceSize);
      const int src_slice = dest_slice + slices_to_drop;
      const int8_t* src_slice_data =
          feature_data_ + (src_slice * kFeatureSliceSize);
      for (int i = 0; i < kFeatureSliceSize; ++i) {
        dest_slice_data[i] = src_slice_data[i];
      }
    }
  }

  if (slices_needed > 0) {
    for (int new_slice = slices_to_keep; new_slice < kFeatureSliceCount;
         ++new_slice) {
      const int new_step = last_step + (new_slice - slices_to_keep);
      const int32_t slice_start_ms = (new_step * kFeatureSliceStrideMs);
      int16_t* audio_samples = nullptr;
      int audio_samples_size = 0;
      GetAudioSamples(slice_start_ms, kFeatureSliceDurationMs,
                      &audio_samples_size, &audio_samples);
      constexpr int wanted =
          kFeatureSliceDurationMs * (kAudioSampleFrequency / 1000);
      if (audio_samples_size != wanted) {
        MicroPrintf("Audio data size %d too small, want %d", audio_samples_size,
                    wanted);
        return kTfLiteError;
      }
      int8_t* new_slice_data = feature_data_ + (new_slice * kFeatureSliceSize);
      size_t num_samples_read;
      TfLiteStatus generate_status = GenerateMicroFeatures(
          audio_samples, audio_samples_size, kFeatureSliceSize, new_slice_data,
          &num_samples_read);
      if (generate_status != kTfLiteOk) {
        return generate_status;
      }
    }
  }
  return kTfLiteOk;
}
```

6. Feature Generator

```
#include "micro_features_micro_features_generator.h"

#include <cmath>
#include <cstring>

#include "micro_features_micro_model_settings.h"
#include "tensorflow/lite/experimental/microfrontend/lib/frontend.h"
#include "tensorflow/lite/experimental/microfrontend/lib/frontend_util.h"
#include "tensorflow/lite/micro/micro_log.h".
#define FIXED_POINT 16
namespace {
FrontendState g_micro_features_state;
bool g_is_first_time = true;
}
TfLiteStatus InitializeMicroFeatures() {
  FrontendConfig config;
  config.window.size_ms = kFeatureSliceDurationMs;
  config.window.step_size_ms = kFeatureSliceStrideMs;
  config.noise_reduction.smoothing_bits = 10;
  config.filterbank.num_channels = kFeatureSliceSize;
  config.filterbank.lower_band_limit = 125.0;
  config.filterbank.upper_band_limit = 7500.0;
  config.noise_reduction.smoothing_bits = 10;
  config.noise_reduction.even_smoothing = 0.025;
  config.noise_reduction.odd_smoothing = 0.06;
  config.noise_reduction.min_signal_remaining = 0.05;
  config.pcan_gain_control.enable_pcan = 1;
  config.pcan_gain_control.strength = 0.95;
  config.pcan_gain_control.offset = 80.0;
  config.pcan_gain_control.gain_bits = 21;
  config.log_scale.enable_log = 1;
  config.log_scale.scale_shift = 6;
  if (!FrontendPopulateState(&config, &g_micro_features_state,
                             kAudioSampleFrequency)) {
    MicroPrintf("FrontendPopulateState() failed");
    return kTfLiteError;
  }
  g_is_first_time = true;
  return kTfLiteOk;
}
void SetMicroFeaturesNoiseEstimates(const uint32_t* estimate_presets) {
  for (int i = 0; i < g_micro_features_state.filterbank.num_channels; ++i) {
    g_micro_features_state.noise_reduction.estimate[i] = estimate_presets[i];
  }
}
TfLiteStatus GenerateMicroFeatures(const int16_t* input, int input_size,
                                   int output_size, int8_t* output,
                                   size_t* num_samples_read) {
  const int16_t* frontend_input;
  if (g_is_first_time) {
    frontend_input = input;
    g_is_first_time = false;
  } else {
    frontend_input = input;
  }
  FrontendOutput frontend_output = FrontendProcessSamples(
    &g_micro_features_state, frontend_input, input_size, num_samples_read);

  for (size_t i = 0; i < frontend_output.size; ++i) {
    constexpr int32_t value_scale = 256;
    constexpr int32_t value_div = static_cast<int32_t>((25.6f * 26.0f) + 0.5f);
    int32_t value =
        ((frontend_output.values[i] * value_scale) + (value_div / 2)) /
        value_div;
    value -= 128;
    if (value < -128) {
      value = -128;
    }
    if (value > 127) {
      value = 127;
    }
    output[i] = value;
  }

  return kTfLiteOk;
}
```

7. Command Responder

```
#if defined(ARDUINO) && !defined(ARDUINO_ARDUINO_NANO33BLE)
#define ARDUINO_EXCLUDE_CODE
#endif
#ifndef ARDUINO_EXCLUDE_CODE
#include "Arduino.h"
#include "command_responder.h"
#include "tensorflow/lite/micro/micro_log.h"

void RespondToCommand(int32_t current_time, const char* found_command,
                      uint8_t score, bool is_new_command) {
  static bool is_initialized = false;
  if (!is_initialized) {
    pinMode(LED_BUILTIN, OUTPUT);
    pinMode(LEDR, OUTPUT);
    pinMode(LEDG, OUTPUT);
    pinMode(LEDB, OUTPUT);

    digitalWrite(LEDR, HIGH);
    digitalWrite(LEDG, HIGH);
    digitalWrite(LEDB, HIGH);
    is_initialized = true;
  }
  static int32_t last_command_time = 0;
  static int count = 0;

  if (is_new_command) {
    MicroPrintf("Heard %s (%d) @%dms", found_command, score, current_time);
    // If we hear a command, light up the appropriate LED
    digitalWrite(LEDR, HIGH);
    digitalWrite(LEDG, HIGH);
    digitalWrite(LEDB, HIGH);
    if (found_command[0] == 'y') {
      digitalWrite(LEDG, LOW);  // Green for yes
    } else if (found_command[0] == 'n') {
      digitalWrite(LEDR, LOW);  // Red for no
    } else if (found_command[0] == 'u') {
      digitalWrite(LEDB, LOW);  // Blue for unknown
    } else {
    }

    last_command_time = current_time;
  }
  if (last_command_time != 0) {
    if (last_command_time < (current_time - 3000)) {
      last_command_time = 0;
      digitalWrite(LEDR, HIGH);
      digitalWrite(LEDG, HIGH);
      digitalWrite(LEDB, HIGH);
    }
  }
  ++count;
  if (count & 1) {
    digitalWrite(LED_BUILTIN, HIGH);
  } else {
    digitalWrite(LED_BUILTIN, LOW);
  }
}

#endif
```

Code Explanation

Microspeech.INO file

1. **Audio Provider**

Audio Sampling

- **Operation**: Continuously captures raw audio from the microphone or audio input device.
- **Buffering**: Buffers the audio samples in memory for further processing.

Buffer Management

- **Rolling Buffer**: Maintains a rolling buffer covering a short duration
- **Sliding Window**: Appends new audio data and discards older data for efficient memory usage.

Timestamp Management

- **Tracking**: Tracks timestamps of the latest audio samples.
- **Synchronization**: Uses timestamps to synchronize feature generation and model.inference.

2. **Interpreter**

Role

- Connects the TFLite model with the hardware it runs on.
- Parses the model structure and allocates necessary resources for inference.

Functions

- **Model Parsing**: Reads the FlatBuffer file to understand layers, operators, and tensors.
- **Memory Allocation**: Allocates memory for:

        - Input tensors.
        - Output tensors.
    - Intermediate computations.

Tensor Arena

- **Definition**: A contiguous block of memory allocated for efficient execution.
- **Function**: Optimizes memory usage during inference to avoid fragmentation.

3. **MicroLog**
    - Designed for logging in resource-constrained environments.
    - **Memory Optimization**: Minimizes memory usage.
    - **Efficiency**: Reduces processing overhead.
4. **Micro Op Resolver**

Memory Optimization

- **Resource Conservation**: Excludes unnecessary operators to conserve memory.

Schema

- A structured blueprint specifying the format, organization, and structure of TFLite models.
- **Serialization**: Defines how tensors, operators, and parameters are encoded into a FlatBuffer file.

Operations Registered with MicroMutableOpResolver

Depthwise Convolution

- Reduces computational complexity in image or speech processing models while maintaining accuracy.

Fully Connected Layer

- Commonly used in classification tasks, connecting every input neuron to every output neuron.

Softmax

- Converts raw logits into probabilities that sum to 1, used for classification outputs.

Reshape

- Changes tensor dimensions (e.g., flattening a 2D tensor to 1D).

5. **Input Tensor Validation**

- Ensures the input tensor's dimensions and data type match the model's requirements.

Input(0)

- Refers to the primary input tensor for the model during inference.

Invoke Function

Model Graph Execution

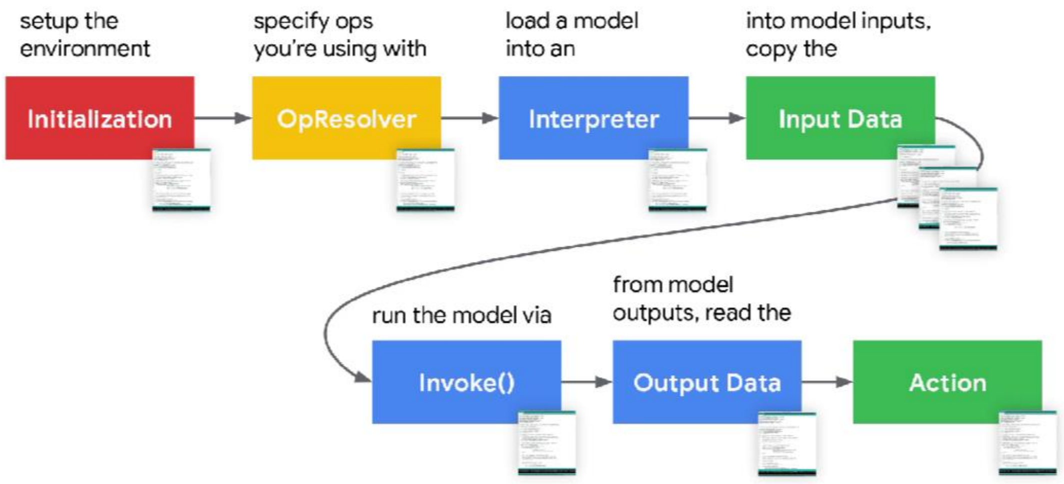
- Processes the input tensor sequentially through the model's layers.
- Each layer generates output tensors for subsequent layers.

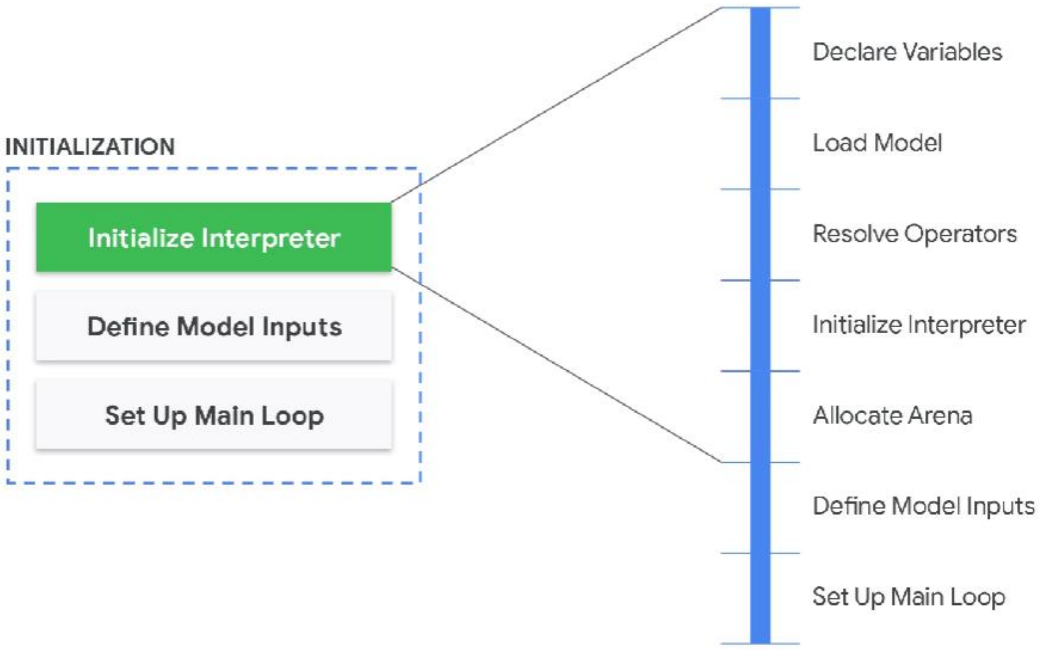
Operator Functions

- Executes layer computations using operators registered in MicroMutableOpResolver.

Memory Management

- **Dynamic Allocation**: Allocates and deallocates tensors in the tensor arena.





Feature provider

1. **Initialization**
    - Constructor initializes feature memory (feature_data_) to zeros.
    - Ensures is_first_run_ flag is set for proper feature generation setup on the first run.
2. **Feature Recycling**
    - Recycles overlapping feature slices by shifting existing data up in memory.
    - Retains slices from the previous time window to avoid redundant computations.
3. **New Feature Generation**
    - Computes the required number of new slices based on the time difference (time_in_ms - last_time_in_ms).

- Fetches new audio data using GetAudioSamples() for each slice and validates its size.
- Generates spectrogram features for each new slice using GenerateMicroFeatures().

4. **Memory Management**
    - Keeps a rolling buffer of feature slices, dropping old slices as needed.
    - Ensures efficient use of memory by allocating only kFeatureElementCount elements.

Micro Features Micro Model Module

This module defines key constants and **configuration settings essential for processing audio input and interpreting the output** of a speech-recognition model. It ensures compatibility between the preprocessing pipeline, trained model, and post-processing steps.

1. Audio Sampling Parameters

It defines the size and frequency of audio input data used for feature extraction.

1. **kMaxAudioSampleSize**
    - Maximum number of audio samples used per FFT calculation.
    - Here, we have taken **512**.
2. **kAudioSampleFrequency**
    - Audio sampling rate in Hz.
    - We are sampling at **16000 Hz** (16 kHz).

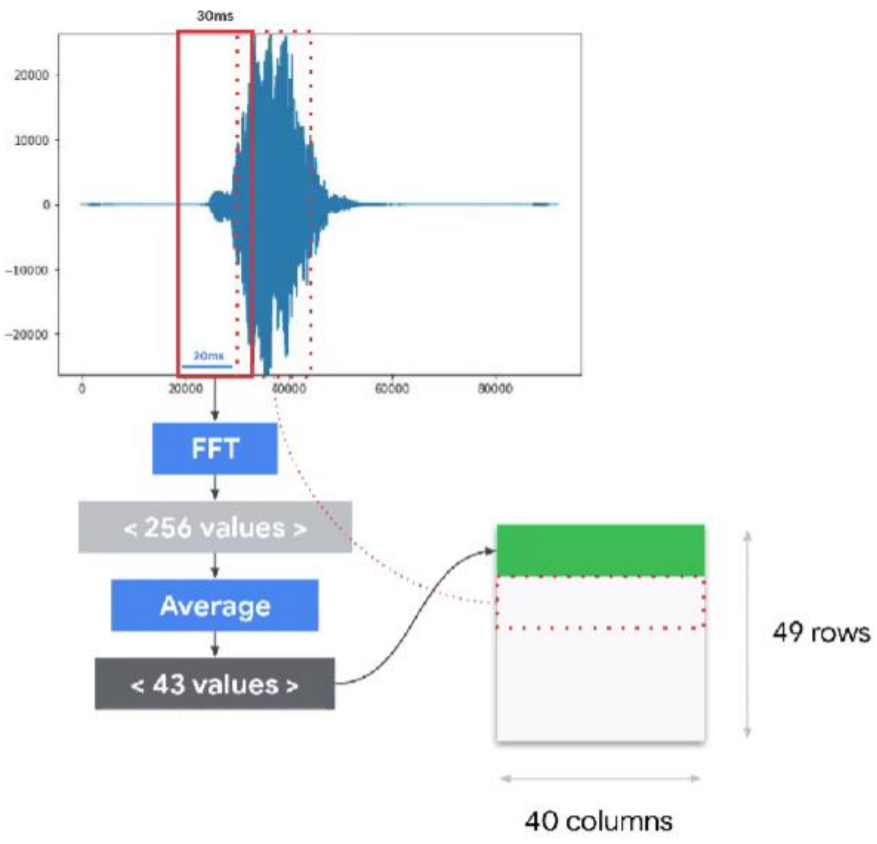
2. Feature Extraction Parameters

It defines the dimensions and characteristics of the spectrogram-like feature representation.

1. **kFeatureSliceSize**
    - **40** frequency bins in each feature slice.
2. **kFeatureSliceCount**
    - **49** time slices in the full feature set.
3. **kFeatureElementCount**
    - Total number of elements in the feature array (flattened spectrogram).
    - Computed as: **kFeatureSliceSize * kFeatureSliceCount**

= 40 * 49 = 1960.

4. **kFeatureSliceStrideMs**
    - Time step between consecutive feature slices of **20 ms.**
5. **kFeatureSliceDurationMs**
    - Duration of each feature slice of **30 ms.**



3. Model Output Categories

Here, we map the model's output indices to human-readable category labels.

1. **kCategoryCount**
   - We took a total of **4** output categories.
2. **Category Indices**
   - Numerical indices corresponding to specific categories.
     - **kSilenceIndex = 0** (Represents silent audio).
     - **kUnknownIndex = 1** (Represents unrecognized audio).
3. **kCategoryLabels**
   - Array of labels for the model's output categories.
     - **"silence"**
     - **"unknown"**
     - **"yes"**
     - **"no"**

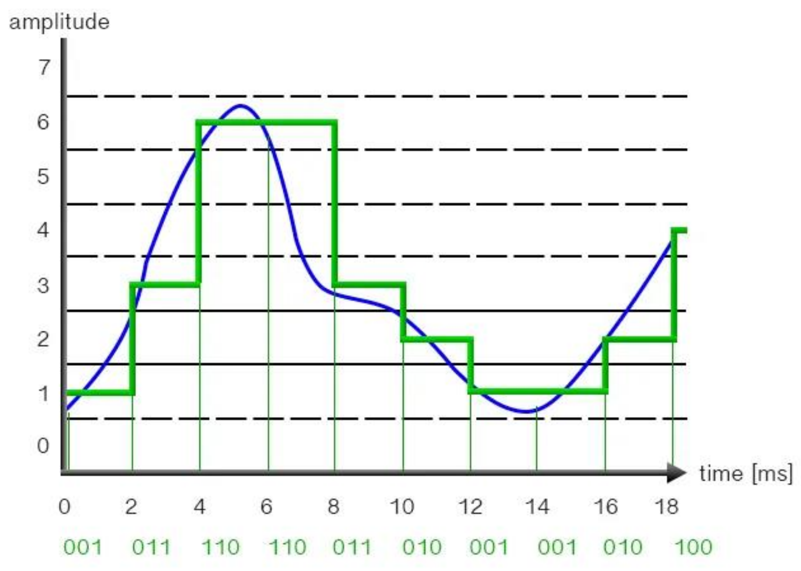
Audio Provider Module

The **Audio Provider Module** is responsible for **managing audio input**, a crucial step in any keyword detection system. By supplying audio samples and timestamps to other components, it ensures the smooth processing of raw audio into features that can be analyzed by the machine learning model. It has three main functions:

1. **GetAudioSamples**

One of the primary functions in this module is **GetAudioSamples**. This function **retrieves audio data in 16-bit Pulse Code Modulation (PCM)** format, which is a standard format for raw audio.

The function allows users to specify the start time (in milliseconds) and duration for which audio samples are needed. The retrieved samples are stored in an array, and the total number of samples is returned via a pointer. This design ensures that the data can be accessed efficiently without duplicating it, making the process memory-efficient.



2. **LatestAudioTimestamp**

The module also has a function named **LatestAudioTimestamp**, which **returns the time when audio data was last captured**. This timestamp is useful for synchronizing audio data with other operations, such as feature extraction or model inference. The value returned is in milliseconds, and while the exact reference point for "time zero" is not defined, the function ensures that subsequent calls return progressively increasing values. This behavior helps maintain consistency in audio processing workflows.

3. **InitAudioRecording**

Another critical function in this module is **InitAudioRecording**, which **initializes the audio recording system**. This setup step is necessary before any audio data can be captured or processed. In real-world applications, this function would typically interact with hardware-specific APIs to activate the microphone and prepare it for use. However, in reference implementations, it might perform a simplified initialization.

Feature Provider Module

The **Feature Provider Module** is responsible for **transforming raw audio input into a structured set of features** that can be used by a machine learning model. It **converts audio data into slices of spectrogram features** and ensures these features are managed efficiently over time, enabling real-time keyword detection or audio analysis. Below are the main functions and components of this module:

1. FeatureProvider Constructor

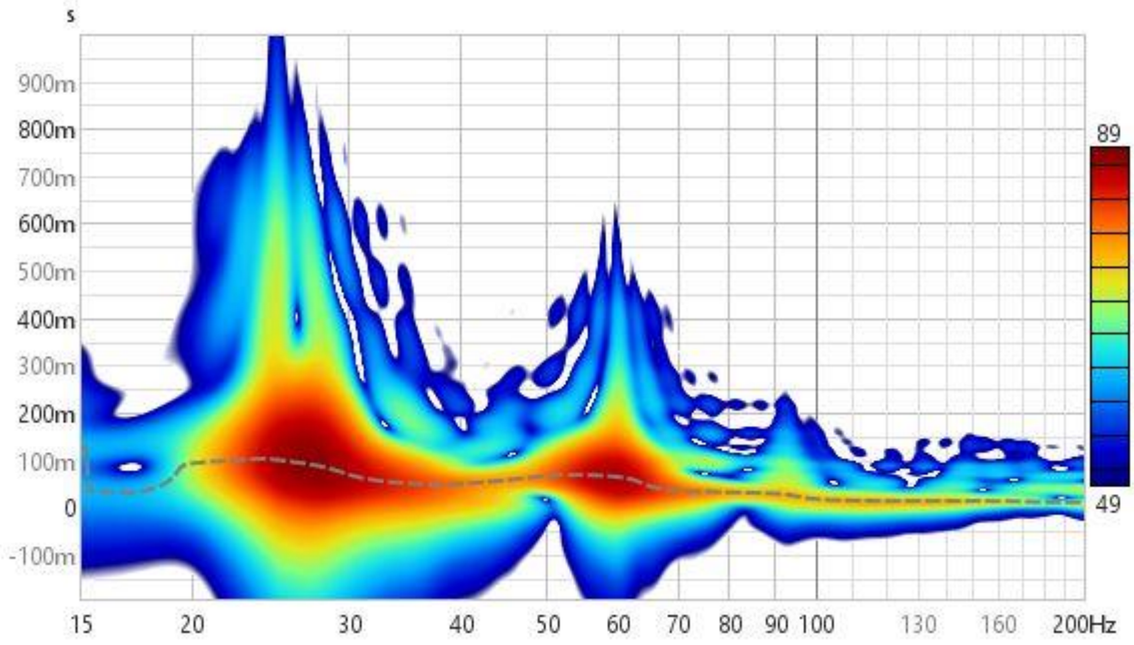
The constructor initializes the feature provider by **binding it to a pre-allocated memory region** where feature data will be stored. It also ensures this memory is cleared by setting all feature values to 0. This prepares the system for efficient reuse of the memory during subsequent operations.

2. PopulateFeatureData

The core function of this module is **PopulateFeatureData**, which updates the feature memory with new spectrogram slices based on the audio data from the **AudioProvider**.

Key operations:

- **Memory Size Validation**: Ensures the allocated memory size matches the expected size **(kFeatureElementCount)**, returning an error if it does not.
- **Time Quantization**: Converts time intervals into steps based on the slice stride **(kFeatureSliceStrideMs)** to determine the required new audio data.
- **Feature Recycling**: Moves existing feature data up in memory to recycle slices and avoid recomputation where possible.
- **Audio Data Fetching**: Calls **GetAudioSamples** to retrieve audio samples for each new slice and ensures the retrieved audio sample size matches the expected size, returning an error if not.
- **Feature Generation**: Converts audio samples into feature slices using **GenerateMicroFeatures** and updates the feature memory block with the newly generated slices.



3. InitializeMicroFeatures

Called during the first run of **PopulateFeatureData**, this function sets up the feature generation process. It ensures proper initialization of the micro feature extraction system before any feature processing.

4. Feature Recycling and Slicing

The module employs an efficient memory management technique:

- **Recycling slices**:
  - Retains slices of existing data that overlap with the new time window by moving them upward in memory.
- **Generating new slices**:
  - Computes new features only for the slices corresponding to the most recent time interval, reducing redundant calculations.

Micro Features Generator Module

The **Micro Features Generator Module** is a critical component in the audio processing pipeline, converting raw audio **signals into compact, quantized features suitable for neural network input.** It includes methods for initializing the feature generation system and processing audio data into a spectrogram representation.

1. InitializeMicroFeatures

This function sets up the necessary state for the feature generation process. It configures the frontend processor, which is responsible for extracting spectrogram-like features from audio data.

- **Key Operations**:
  - Calls **FrontendPopulateState** to initialize the **g_micro_features_state** object, setting up the filters and windowing parameters.
  - Resets the **g_is_first_time** flag to ensure that initial configurations are applied during the first feature extraction.

2. GenerateMicroFeatures

The primary function in this module, **GenerateMicroFeatures**, processes raw audio input into quantized features that can be fed into a neural network.

- **Key Steps**:
  - **Frontend Processing**:
    - Calls **FrontendProcessSamples** to **compute intermediate feature values** (energy levels for frequency bands) from the audio samples.
    - Tracks how many audio samples were processed.
  - **Feature Quantization**:
    - **Converts frontend output values into a compact integer representation** using scaling and quantization:
      - Scales frontend values to the range [-128, 127], ensuring compatibility with quantized neural networks.

3. SetMicroFeaturesNoiseEstimates

An auxiliary function to adjust the noise estimates in the frontend state. This is primarily used for testing or fine-tuning the noise reduction process.

It directly modifies the **g_micro_features_state** to update noise reduction parameters.

Command Responder Module

The Command Responder Module is responsible for processing the results of voice command recognition and providing corresponding visual feedback using LEDs. It has one main function and that is:

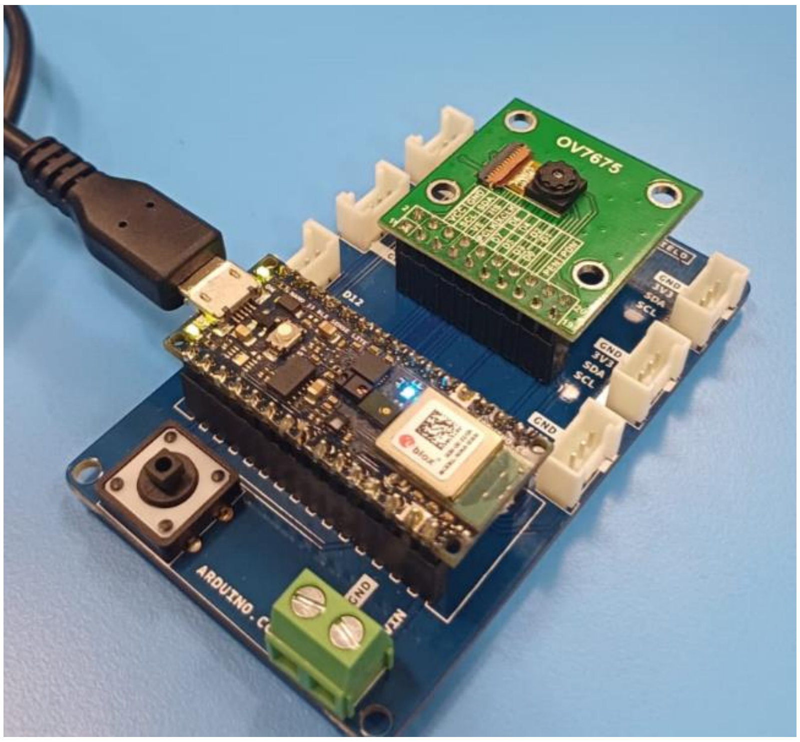
RespondToCommand:

The **RespondToCommand** function **interprets the detected command and controls the LEDs** accordingly. Its key functionalities include:

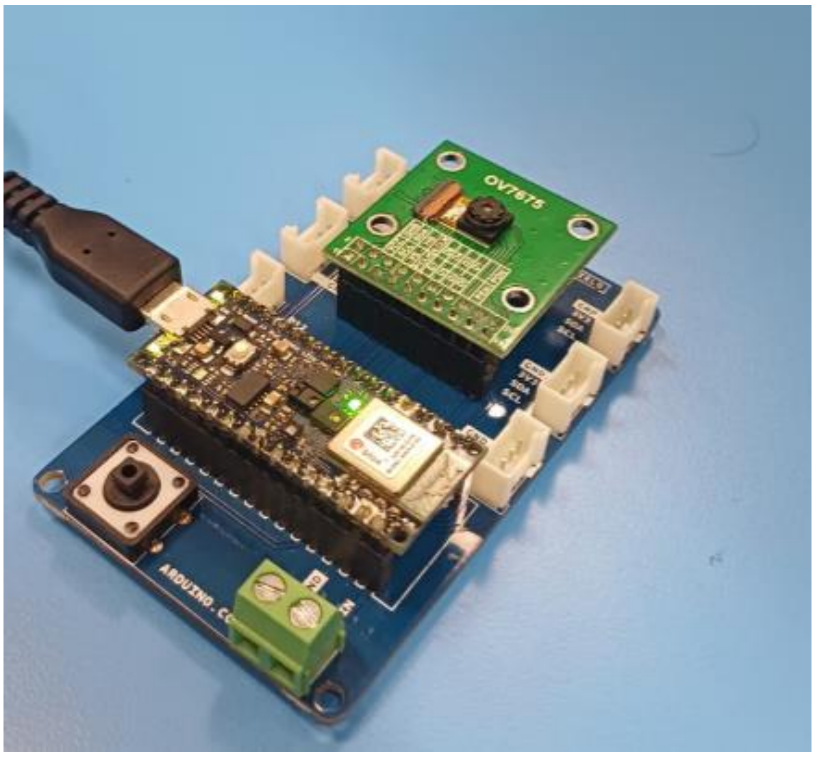
- **Initialization**:
  The function initializes the LED pins during the first call. It configures the built-in LED and RGB LED pins as outputs and ensures that the RGB LEDs are off by default.
- **Command-Based Feedback**:
  When a new command is detected, the function lights up the RGB LED corresponding to the command:
  - Green (LOW) for commands starting with "y" (yes).
  - Red (LOW) for commands starting with "n" (no).
  - Blue (LOW) for commands starting with "u" (unknown).
    If no command is recognized, the LEDs remain off.
- **Timeout Mechanism**:
  The RGB LEDs remain lit for up to 3 seconds after the last recognized command. If no new command is detected within this time, the LEDs are turned off to conserve energy and reset the system for the next interaction.
- **Inference Status Indication**:
  The built-in LED toggles its state (ON/OFF) after every inference cycle, providing a visual indication that the system is active and processing audio input.

Hardware Output

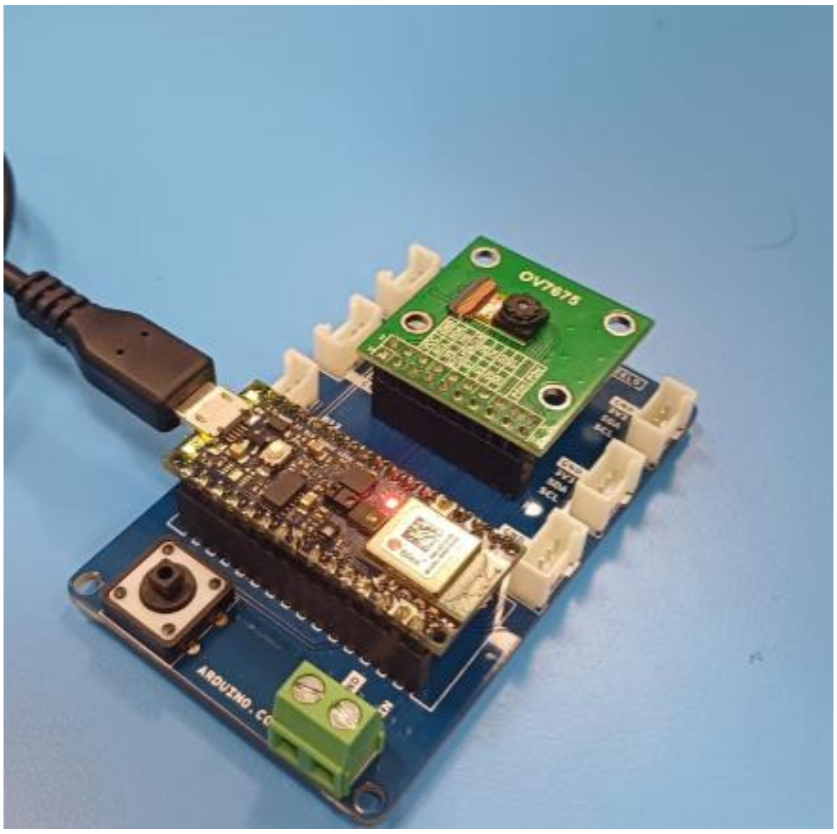
- Colour indicating Unknown voice(Blue)



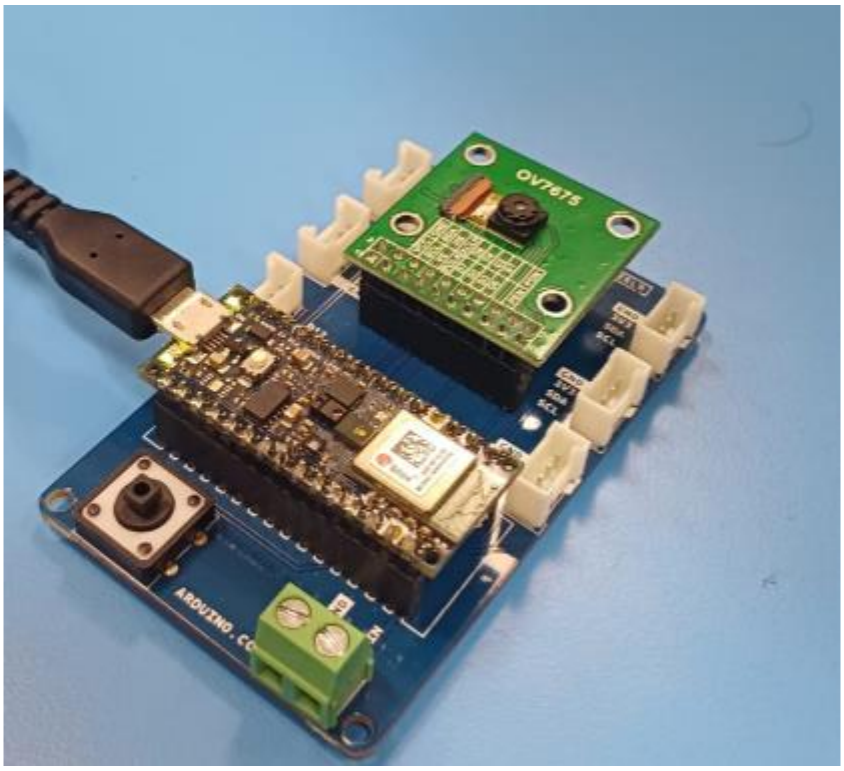
- Green colour indicating "Yes"



- Red colour indicating “No”



- No LED lights for silence



Conclusion:

The keyword spotting project using TensorFlow Lite and Arduino Nano 33 BLE Sense demonstrates the viability of deploying machine learning in resource-constrained environments. By leveraging the device's low-power **nRF52840 microcontroller** and 256 KB SRAM, this system processes real-time audio input to detect keywords like "yes" and "no" with minimal computational and memory overhead. Using **TensorFlow Lite for Microcontrollers**, the neural network model is quantized to optimize performance without sacrificing accuracy, enabling seamless operation on embedded systems.

The project highlights an efficient pipeline for compact feature extraction, where raw audio is converted into spectrogram-like features that are compatible with lightweight inference. This approach supports real-time keyword detection while ensuring low energy consumption, making it suitable for battery-powered applications such as wearables, IoT devices, and voice-controlled systems.

Furthermore, the modular design allows for scalability and customization, enabling developers to expand the system to recognize additional commands or adapt it for specific use cases. Practical applications include voice-activated appliances, smart assistants, and accessibility tools. Overall, the project underscores the potential of deploying AI-powered keyword spotting on low-power embedded platforms, bridging the gap between cutting-edge machine learning techniques and real-world implementation in constrained environments.