EE491 - BTP 1:

Voice-controlled interface on RPi Pico W using Zephyr RTOS and TensorFlow (TFLM)

K P Lakshmeesh (22B1246)
EE BTech, IIT Bombay

Guide: Prof. Siddharth Tallur

INDEX

1. Introduction

Project motivation:
Getting comfortable with creating applications on top of an RTOS and also learn how TinyML models can be implemented on resource constrained devices.
To explore this, we chose Zephyr RTOS, to make use of its networking stacks along with TFLM to build a speech keyword classification device on a Raspberry Pi Pico W, and perform offline inference directly on device and then communicate commands wirelessly to other devices based on the inferred result.

Why zephyr?
Zephyr provides a full OS ecosystem. It comes built-in with device trees and drivers for several boards, a build system. It also provides a standard set of APIs. It has several modules, the relevant ones being TFLM and FFT modules. Zephyr's networking stack also makes it convenient to develop IOT applications.

2. Useful beginner's references

1. Shawn Hymel's Youtube playlist [13] [14]
   A beginner introduction to various aspects of ZephyrRTOS. Helpful for building foundational understanding before working on Zephyr projects

2. Official zephyr documentation: [15]
   Primary reference for all aspects of Zephyr, including supported boards, Kconfig options, and detailed API documentation, which were used extensively throughout the project

3. Zephyr's Github repo and community: [16]
   - Discord: [17]
   - GitHub issues: [18]
   Useful for discussions and resolving doubts

4. TensorFlow documentation: [19]

3. Environment setup

3.1. For alternate setup through VScode extensions:

- For Pico specific purposes: “Raspberry Pi Pico” extension;
  It also contains a GUI interface to configure for different Pico board automatically based on the application.
- “Zephyr IDE” extension

3.2. General setup

Follow the official setup guide until “build the blinky sample” section : [1] [2]

West is a Zephyr tool for managing the workspace, fetching and updating its modules from Github. It is also a wrapper over CMake and Ninja for convenient building. Also helps in flashing and debugging.
The workspace topologies can vary [3].The default layout was used by me.

Before building, confirm that the board is supported by zephyr [4],
I am using a RPi Pico W which is partly supported (wifi support is limited). I have attached a board file with which you can work with Pico W. It is a small extension of the Pico board file provided by zephyr. I have explained the changes below.

An applications folder was maintained to store multiple independent projects. Each Zephyr application builds into a directory named build by default. The build output location can be changed using:
west build -b <BOARD> -d <BUILD_DIR>

By default, the Pico W uses UART for console logs. To view the output on terminal via USB, include this built-in snippet in the build “-S cdc-acm-console” and initialise the USB at boot.
“west build -p always -b rpi_pico_rp2040_w -S cdc-acm-console samples/basic/blinky -- -DCONFIG_USB_DEVICE_INITIALIZE_AT_BOOT=y”
Or simply,
“west build -p always -b rpi_pico_rp2040_w -S cdc-acm-console samples/basic/blinky”
if we include CONFIG_USB_DEVICE_INITIALIZE_AT_BOOT=y in prj.conf

“-p always” ensures a clean rebuild when required.

Unlike the standard Pico, the LED in Pico W is not wired to a regular GPIO (gpio25), but instead it’s driven by the CYW43 wifi chip. Thus, the sample won’t be built due to the incomplete device tree. but for sake of testing this sample, you can modify this line and choose another GPIO to blink an external LED (Note: This workaround is not recommended, it’s just for a quick testing).

3.3. Flashing the build:

Flashing varies by board (see supported boards link), but for the Pico W:
- Hold the BOOTSEL button and plug the board into the PC.
- It appears as a mass storage device.

Now either, run *west flash -r uf2*, or manually drag-and-drop: <build_dir>/zephyr/zephyr.uf2 into the Pico’s USB storage.

3.4. Debugging tools: [5]

- We can use debuggers like GDB + OpenOCD, J-link hardware debuggers, CMSIS-DAP probes
- For larger projects, logging features are also useful:
  - Zephyr logging: [6]
  - TFLM testing library:
    see “micro_test.h” (zephyrproject/optional/modules/lib/tflite-micro/tensorflow/lite/micro/testing)

4. Understanding Build system and Hardware abstraction

4.1. Build configuration:

- CMAKE:
    - We can directly use the g++ compiler for a small number of files but, with very large no. of files and libraries it becomes hard to manage them. Thus, we use CMake.
    - CMake helps building large projects by reading the CMakeLists.txt, locating sources, libraries, setting compiler flags, and generating the required Makefiles.
    - When we perform “make” on the makefiles, it executes g++ commands to compile and link the project.
    - The CMake system integrates device tree configuration, Kconfig options, and driver files to produce the final firmware image.
    - Every application must contain:
        - A src folder (name as per CMakeLists.txt) with the application code (main.c)
        - A CMakeLists.txt file (name fixed) which lists the source files
        - A prj.conf file (name fixed) that selects Zephyr features.
- KConfig:
    - Used to control which kernel features are enabled (like enabling ADC, UART, wifi, TFLM etc.) and setting stack sizes.
    - It is needed to ensure that only the required components are included in the build, keeping RAM space minimal.
    - These are resolved while building, and corresponding codes are added.

4.2. Hardware description:

- Device trees (.dts):
    - Each board provides a device tree describing its hardware (eg, IO peripherals, memory, clocks, interrupts etc.). It has a tree-like structure.
    - We do application-level customisations by writing overlay files to enable/disable device hardware or overwrite some properties without editing the base board files.
    - It helps in the separation of the board hardware description and driver codes.
      Thus, allowing the same driver codes to be reused across different boards which use the same hardware components.
- Binding files (.yaml):
    - Describes the mandatory and optional properties, property values for a .dts file to meet the driver's requirements.
    - While building, Zephyr validates the device tree using these bindings and also helps developers build the .dts
    - The bindings file name usually matches the compatible string value.
- Device driver:
    - It contains the actual low-level code implementation that interacts with the hardware registers, and actually does the job.
    - They operate independently of board-specific details because the device tree supplies configuration parameters at build time

4.3. Example workflow (ADC):

- Developer writes an overlay file to enable ADC channels and define properties (gain, reference, acquisition time, resolution, etc.). In the main code, the device is instantiated and setup, and standard Zephyr APIs are used to access the ADC, so that the codes remain the same across all devices.
- While building the project, we specify the board being used. The build system verifies the dts file using the corresponding bindings file. It extracts the parameters from the device tree file and overlays, and feeds it to the driver code, enabling the ADC to sample as requested.

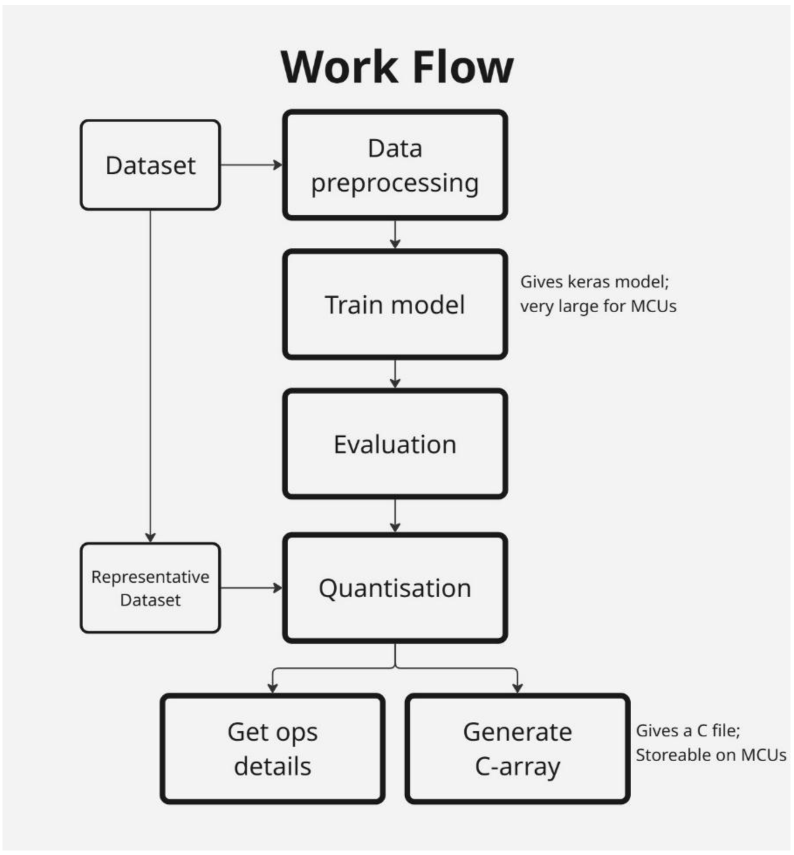
5. TensorFlow Lite for Microcontrollers

- TFLM is much more compact than typical PC ML frameworks. It avoids dynamic memory usage through a fixed sized tensor arena to carry out operations, and builds only the operators the model actually needs. Usually, INT8 quantised weights and activations are used instead of float. All of this reduces memory requirements, and computational costs heavily and makes inference on microcontrollers feasible.

5.1. Setting up TFLM: [7]

- By default, some zephyr modules (including TFLM) are optional and not fetched to reduce space usage. (see "submanifests/optional.yaml").
- To include TFLM, do:
  west config manifest.project-filter -- +tflite-micro
  west update
  You can see now that within ".west/config.py" indicates that tflite-micro module has been fetched.
- These Kconfig options are also required in "prj.conf":
  CONFIG_CPP=y
  CONFIG_REQUIRES_FULL_LIBC=y
  CONFIG_TENSORFLOW_LITE_MICRO=y
- Note: TFLM depends on C++ libraries thus it has to be written in a .cpp file.

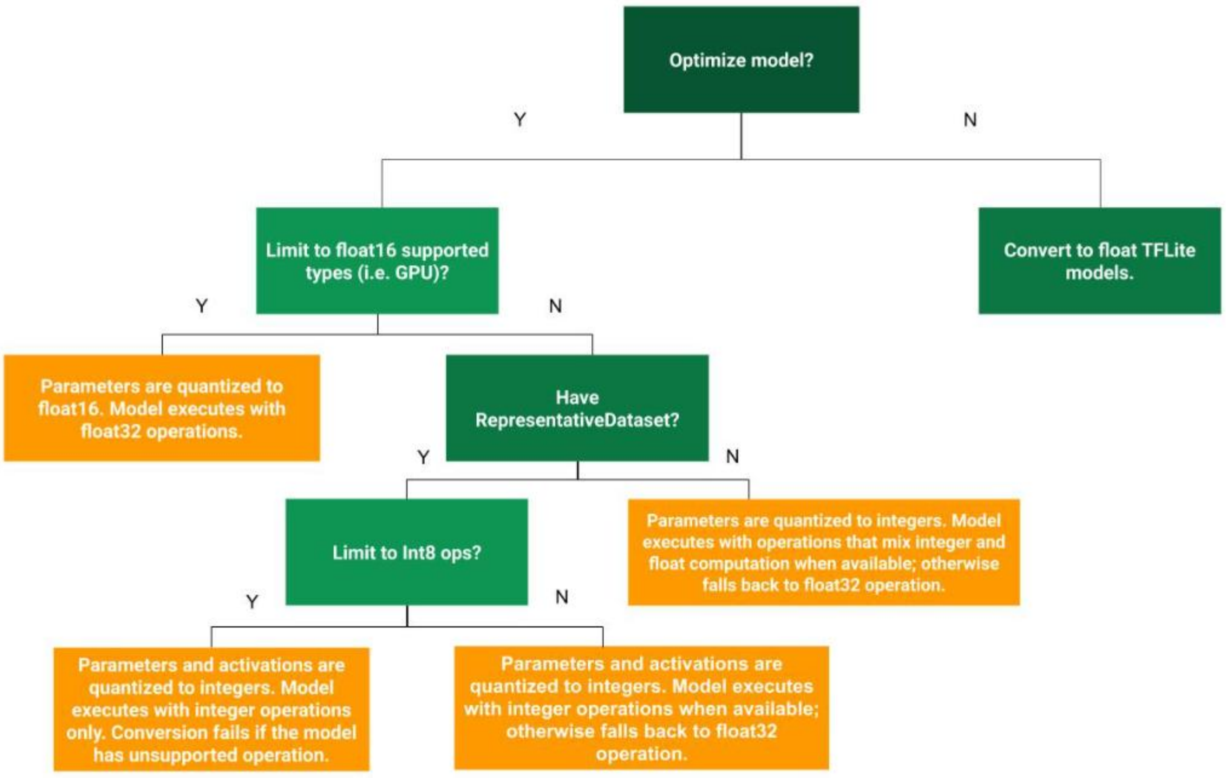
5.2. Jupyter notebook flow ("Simple_audio.ipynb"):



5.3. Quantization: ([8] [9])

- Quantisation converts float32 models into lower precision models, giving huge saving on space and performance.
  Full integer quantisation (done in the project) can reduce the size of .tflite model to upto 75% of the float model size, while maintaining comparable accuracy.
- Example from project:
  - In my project, these are the file sizes: keras file = 407KB; tflite file = 39KB; C-array file = 242KB
  - Note that .Keras file doesn't just contain the model, it also contains lots of other data, which is stripped away while converting to .tflite, indicating the 10x size reduction.
    But the model size is reduced only by ~4x.
  - You may also notice that the C-array file size is large. This is since C files are stored in text format, so each byte stored in tflite (eg. "0xab, ") occupies 6 bytes of space as characters (dependent on format of the generated C file).
    But in RAM it will only occupy ~39KB.

- **5.3.1 Post-training quantization:**
  We can perform several kinds of quantisation after training our model.
  - Dynamic range quantisation:
    The weights are converted to int8 precision, while the activations/output are still float32.
    We don't require a representative dataset to perform this quantisation.
  - Full integer quantisation:
    Both weights and activations are converted to int8.
    Calibrating the activation functions ranges requires a representative dataset to get an estimate.
  - You can also quantise such that weights are quantised to float16, or activation functions are int16 and weights are int8.

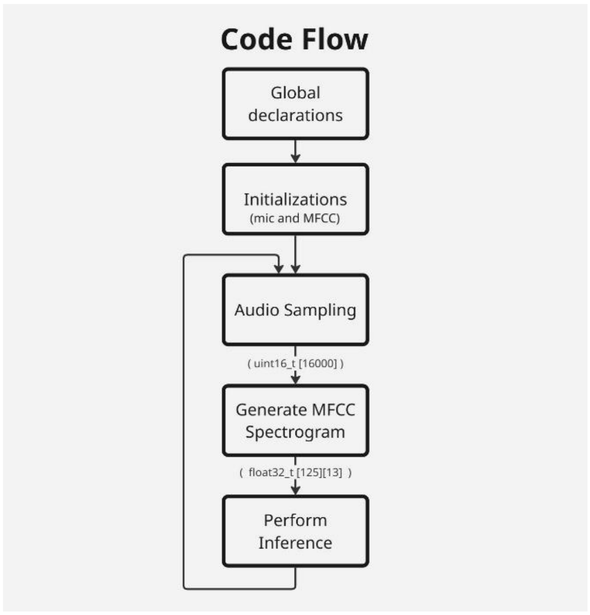

Optimize model?
Y
N
Limit to float16 supported types (i.e. GPU)?
Convert to float TFLite models.
Y
N
Parameters are quantized to float16. Model executes with float32 operations.
Have RepresentativeDataset?
Y
N
Limit to Int8 ops?
Parameters are quantized to integers. Model executes with operations that mix integer and float computation when available; otherwise falls back to float32 operation.
Y
N
Parameters and activations are quantized to integers. Model executes with integer operations only. Conversion fails if the model has unsupported operation.
Parameters and activations are quantized to integers. Model executes with integer operations when available; otherwise falls back to float32 operation.

- **5.3.2. Quantisation aware training:**
    - Quantisation is very lossy, as a lot of the precision which comes with float32 is lost while scaling them down to int8. During inference the errors accumulate.
    - In QAT, this error is minimised during the training thus, providing accuracies closer to the original float models.

5.4. TFLM application flow:

- Include TFLM libraries and declare global structures
- Create operator resolver. Only include kernels required by the model (identified from the notebook)
- Load the Model using GetModel() with the C-array model
- Create an interpreter to run the model with
- Allocate memory to Tensor arena. We find the optimal tensor arena size by trial-error. Print the bytes used by arena to optimise the allocated space.
- Get pointers to inference input/output
- Perform input quantisation using scale and zero_point parameters.
- Run inference on the input
- dequantize output data and interpret the class probabilities. Return the one which clears a good threshold.

6. Application: Speech keyword classification



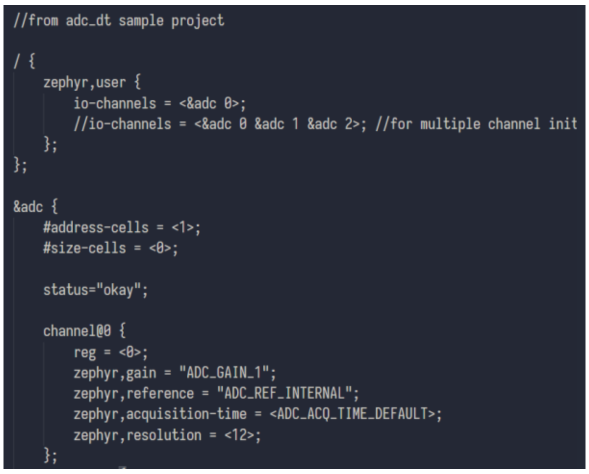
a) **Microphone interfacing:**

- Initial testing:

The microphone module (MAX9814) was tested using micro-python with "serial-to-wav.py" codes.

- Zephyr implementation:

- Setting up the ADC:
  (samples/drivers/adc/adc_dt was used for reference)
  i. Set "*CONFIG_ADC=y*" in "prj.conf"
  ii. We create a board overlay specifying the ADC channel and its configuration (refer to the bindings file "adc-controller.yaml")



```
//from adc_dt sample project

/ {
    zephyr,user {
        io-channels = <&adc 0>;
        //io-channels = <&adc 0 &adc 1 &adc 2>; //for multiple channel init
    };
};

&adc {
    #address-cells = <1>;
    #size-cells = <0>;

    status="okay";

    channel@0 {
        reg = <0>;
        zephyr,gain = "ADC_GAIN_1";
        zephyr,reference = "ADC_REF_INTERNAL";
        zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
        zephyr,resolution = <12>;
    };
```

  - We choose the ADC channels and set their configurations. The register value 0 corresponds to ADC A_0 for Pico. This can be referred from boards/raspberrypi/common/rpi_pico-pinctrl-common.dtsi.
  - We enable the ADC (and other devices) by setting *status="okay".*

  iii. Then referring to "drivers/adc/adc_dt", within src we setup the adc channel and read its buffer through APIs provided by zephyr.

- The sampling delay is ideally supposed to be (1/samp_rate) seconds. However, we also need to further account the other delays within the sampling loop, which can make the effective sampling rate slower, and make the audio which is actually heard by Pico slower.
  Thus the loop delay = (1/samp_rate) – (the additional delay). The additional delay is mainly due to the API call which reads the ADC.
  APIs related to timing which I used can be found on zephyr's documentation. [10]

- However, the recorded delay for “read_mic()” function is around 105us, making it impossible for a 16KHz sampling (62.5us).
  Few workarounds are:
  i. Lowering the effective sampling rate to 8KHz which is still adequate. Note that the MFCC constants have to be correspondingly updated.
  ii. Use Direct Memory Access (DMA) for faster ADC sampling.

b) **Generating Spectrogram:** [11]
The above notebook by TensorFlow was used as a reference.
- **STFT spectrogram** (Short-Time Fourier Transform):
  - The above reference uses a STFT of the audio.
  - If we just take the FFT of an audio, we only get the frequency information, but lose out on the information of when they occurred. Thus, the audio is split into overlapping time frames (so as to not miss any audio transitions),
  - a windowing function (like hamming) is applied to prevent sharp discontinuities and FFT is taken.
  - These are stacked together to create a 2D spectrogram.
    The resulting dimension is a design choice. The one in the above link has “square” dimensions so they have used a 2D CNN based model.

- **MFCC spectrogram** (Mel-Frequency Cepstral Coefficients):
  (for code implementations: [12])
  - MFCC is an extension of STFT, it also relies on splitting the audio into short overlapping time-frames, windowing and calculating FFT of each frame.
  - Then the Mel filter bank is applied on these FFTs, converting the linear frequency scale to a non-linear one, making it more aligned with human perception. This Mel-scale is linear for lower frequencies (below 1KHz) and then logarithmic at higher frequencies, to model how human ears are more sensitive to changes in lower pitch than higher. The output contains the energy captured within each filter bank.
  - Logarithm of the output of the filter bank is taken modelling humans logarithmic perception of loudness
  - Then, a DCT is applied on the log-mel spectrum to obtain the significant features from each frame. Thus, compressing the features and decorrelating information.
  - This spectrogram is then used for training the model. The dimensions of the spectrogram generated by me was designed to be (125,13) thus, a 1D CNN model was be used.

- **Libraries considered:**
  - Zephyr includes kissfft, a very lightweight FFT library, which I used to try building the STFT spectrogram. However, I encountered a lot of build errors while integrating it, So I chose to temporarily switch to CMSIS-DSP library.
  - This is actually a better fit for the project since MFCC spectrogram features are more compact than STFT.
    MFCC performs better for speech tasks since they are designed to mimic how human percept audio, using the Mel scale which models the non-linear perception of pitch.
    It is also more robust to noise.
    Besides, the CMSIS-DSP is made to be optimised for ARM Cortex-M/A devices. (It also means that it cannot be ported to non-Cortex-M/A devices).
    Note that Pico’s Cortex-M0+ lacks DSP extensions thus the implementation won’t be optimised.

- **Using CMSIS-DSP MFCC library:**

- For MFCC, we require constants like Mel filters, window coefficients, DCT coefficients.
  This can be generated using GenMFCCDataForCPP.py script (modules/lib/cmsis-dsp/Scripts).
- Generating MFCC Constants:
  - The script relies on SciPy, and in the newer versions required a small modification in mfccdata.py: (or see applications/cmsis_gen)
    from scipy.signal.windows import hamming, hann
    After this, replace the hamming accordingly to make it work
  - A custom .yaml file has to be made to specify the sampling rate, FFT length, no. of mel filters, DCT outputs, fmin, and fmax (<= fs/2 due to Nyquist limit)
    (Refer to example config files provided by CMSIS)
  - Then run the script using:
    python GenMFCCDataForCPP.py -n "consts" -d out_c -i out_h mfccconfig.yaml
    Where "consts" is output file name, out_c/out_h is the destination of .c/.h files, and "mfccconfig.yaml" is the custom config file.
    The generated files can directly be included in the project.
- MFCC setup in code:
  - CMSIS provides MFCC instance structures (eg, arm_mfcc_instance_f32) in "cmsis-dsp/include/dsp/transform_functions.h".
  - We initialise the MFCC instance (eg, arm_mfcc_init_f32) with the required arrays.
  - MFCC transform can then performed (eg. "arm_mfcc_f32()").

- **Issues faced with CMSIS-DSP MFCC:**

- CMSIS-dsp MFCC was giving a lot of unexpected issues. In one case, the MFCC worked depending on the presence of some debug printk() statements, which on removing causes the function to hang.
- The order of declaration of the MFCC structs also matters. Declaring "arm_status status" and "arm_mfcc_instance_f32 mfcc_inst" before any other array was required to avoid issues.
- These problems were difficult to diagnose and not intuitive, consumed significant debugging time. Thus, I plan to explore alternative libraries or create one myself for generating spectrograms in future iterations.

7. References

1. https://docs.zephyrproject.org/latest/develop/getting_started/index.html
2. https://www.hackster.io/cdwilson/zephyr-rtos-on-raspberry-pi-pico-2-part-1-cf39f0
3. https://docs.zephyrproject.org/latest/develop/west/workspaces.html
4. https://docs.zephyrproject.org/latest/boards/index.html
5. https://docs.zephyrproject.org/latest/develop/west/build-flash-debug.html#debugging-west-debug-west-debugserver
6. https://docs.zephyrproject.org/latest/services/logging/index.html
7. https://docs.zephyrproject.org/latest/samples/modules/tflite-micro/hello_world/README.html#tflite-hello-world
8. https://www.tensorflow.org/model_optimization
9. https://ai.google.dev/edge/litert/models/model_optimization
10. https://docs.zephyrproject.org/apidoc/latest/group__clock__apis.html#ga59d9bd47b0caa662f0e289cf3df83a82
11. https://www.tensorflow.org/tutorials/audio/simple_audio
12. https://www.geeksforgeeks.org/nlp/mel-frequency-cepstral-coefficients-mfcc-for-speech-recognition/
13. https://www.youtube.com/playlist?list=PLEBQazB0HUyTmK2zdwhaf8bLwuEaDH-52
14. https://github.com/ShawnHymel/introduction-to-zephyr
15. https://docs.zephyrproject.org/latest/index.html
16. https://github.com/zephyrproject-rtos/zephyr
17. https://discord.com/invite/Ck7jw53nU2
18. https://github.com/zephyrproject-rtos/zephyr/issues
19. https://www.tensorflow.org/api_docs/python/tf