

Get started

Open in app



## Dmytro Korablyov

19 Followers

About

Follow



# First steps with ESP32 and TensorFlow Lite for Microcontrollers



Dmytro Korablyov Feb 11, 2020 · 11 min read

## Motivation

I have no doubt tiny edge devices will take a meaningful place in our life soon. Since Moore's Law applicable to such devices we are the spectators of maturing of mobile, embedded, wearable and implantable (augmenting) electronic devices with computational power enough to using AI.

Unveiling TensorFlow Lite for Microcontrollers on [TensorFlow Developer Summit](#) approaches us to deploy AI on embedded devices. Evidently, to successfully deliver AI applications for microcontrollers, developers (or developers team) should be familiar with both ML framework and microcontroller programming. So I decided to narrow this gap and do some hands-on exercises with TensorFlow Lite for Microcontrollers. I prefer approach with gradual complexity increasing.

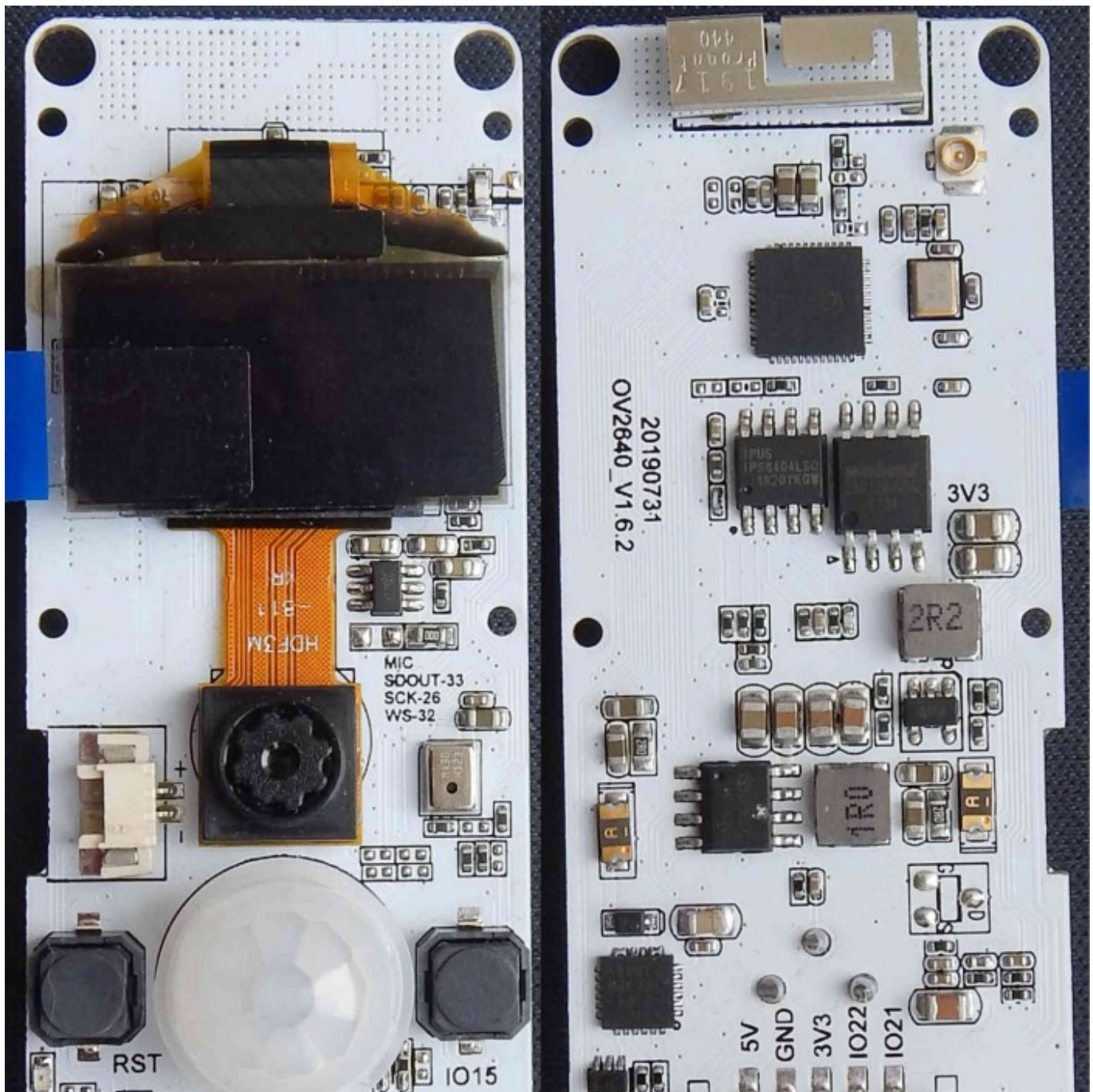
## Brief plan

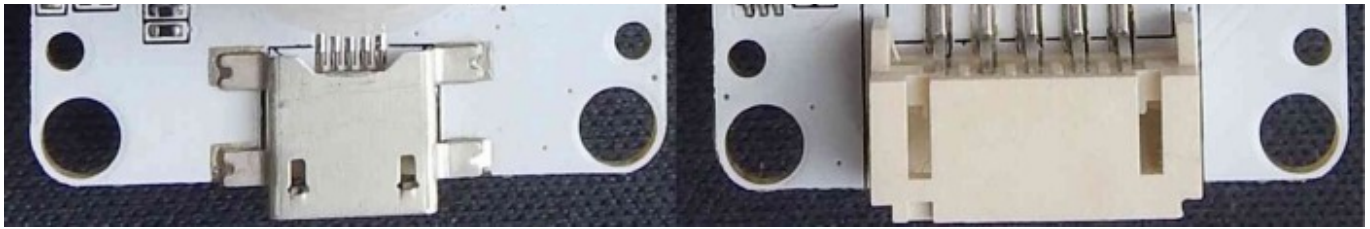
1. Choose the hardware. Choose the software development environment for microcontrollers. Setup the development environment.
2. Run "ESP32 Hello World" as a first step to be familiar with the new World.
3. Run "TensorFlow Lite Hello World" from TensorFlow samples repo.

4. Build, deploy and run “Fashion MNIST” application from scratch.
5. Conclusions. Next steps.

## 1. Hard and Soft

The best way to start choosing the platform for experiments is by looking at the list of Supported platforms by TensorFlow Lite for Microcontrollers. I've chosen the ESP32 platform. Since I am going to perform experiments with video and audio, my constraints for the board are: connected with micro-USB cable, no additional connections or soldering, Camera onboard, microphone onboard.





### Main features of the board:

Module: ESP32-D0WDQ6, dual-core MCU with low-power co-processor

Wireless: Wi-Fi and Bluetooth

Flash: 4 MB

SRAM: 520 KB

Pseudo static RAM (PSRAM): 8 MBytes

OLED Display, PIR sensor, Charging controller, Microphone, Camera

There are many IDEs for programming ESP32. Which to choose? Let's take a glance at the TensorFlow Lite for Microcontrollers documentation, which says:

---

*It doesn't require operating system support, any standard C or C++ libraries, or dynamic memory allocation. The core runtime fits in 16 KB on an Arm Cortex M3, and with enough operators to run a speech keyword detection model, takes up a total of 22 KB.*

---

Consequently, we can choose any even “vim+console”. Great! Before choosing a specific software environment (IDE) I looked at the TensorFlow Lite ESP32 demo and I found that the TensorFlow team uses ESP-IDF version 4.0. Now it is an unstable release but that fact will not be an obstacle to us. We will use nowadays actual version 4.1. It turns out the Official Visual Studio Code Extension for ESP-IDF Projects exists. So choosing the editor seems obvious. This time we will use VSCode as a smart editor but not as a fully functional IDE.

Let's prepare the software. My development computer running Ubuntu. I prefer to put repositories and software in certain places, so sometime I will choose the customized installation. I suppose you familiar with the terminal. Then we will do a few steps based on official installation instructions:

1. Prepare the directory structure.
2. Install Prerequisites for ESP-IDF. Then get ESP-IDF. Then set up the ESP IDF tools.

3. Setup VSCode by downloading and installing .deb from the official site.
4. Install Official Visual Studio Code Extension for ESP-IDF Projects into VSCode.

Prepare the directory structure. Run follow command:

```
mkdir ~/esp && mkdir ~/esp/idf && mkdir ~/esp/idf-tools && mkdir  
~/esp/backup && mkdir ~/esp/projects
```

Install Prerequisites for ESP-IDF.

```
sudo apt-get install git wget flex bison gperf python python-pip  
python-setuptools python-serial python-click python-cryptography  
python-future python-pyparsing python-pyelftools cmake ninja-build  
ccache libffi-dev libssl-dev
```

Then get ESP-IDF.

```
cd ~/esp/idf  
git clone --recursive https://github.com/espressif/esp-idf.git
```

Since the ESP-IDF build system works with Python, its documentation says:

---

*To manage the Python version more generally via the command line, check out the tools [pyenv](#) or [virtualenv](#). These let you change the default python version.*

---

For convenient work with the console, we will create an alias to run the script which sets needed environment variables for the current session. We need to add these lines to your

`.bashrc` file:

```
# For ESP-IDF v4.1  
alias get_esp32='. $HOME/esp/idf/esp-idf/export.sh'
```

Set the environment variable as preparation for setting up ESP-IDF tools. We need to add these lines to your `.profile` file:

```
# For ESP-IDF v4.1
export IDF_TOOLS_PATH=$HOME/esp/idf-tools
export PATH="$HOME/esp/idf/esp-idf/tools:$PATH"
```

Reload `.profile` and `.bashrc` files to apply our changes.

```
source ~/.profile && source ~/.bashrc
```

Set up ESP-IDF tools.

```
cd ~/esp/idf/esp-idf
./install.sh
```

Now we are ready to check communication with your ESP32 board. First of all, we need to know the communication port name to flash and monitoring the board. Be sure that your board is NOT connected to your development computer with USB-MicroUSB cable. Run following command:

```
dmesg | grep -e tty
```

Look at the result. Then connect your board to your development computer with USB-MicroUSB cable. Run the command again:

```
dmesg | grep -e tty
```

Look at the new result. You should see that the new row appeared in the new result, something like:

```
[ 2471.977971] usb 1-1: cp210x converter now attached to ttyUSB0
```

The **ttyUSB0** is the communication port. But in future commands, we will use full port name: **/dev/ttyUSB0**. In your computer, probably you could receive other ports similar to **ttyUSB\***.

**NOTE:** Use your port name in future commands.

Add user read-write permission to communicate with serial port.

```
sudo usermod -a -G dialout $USER
```

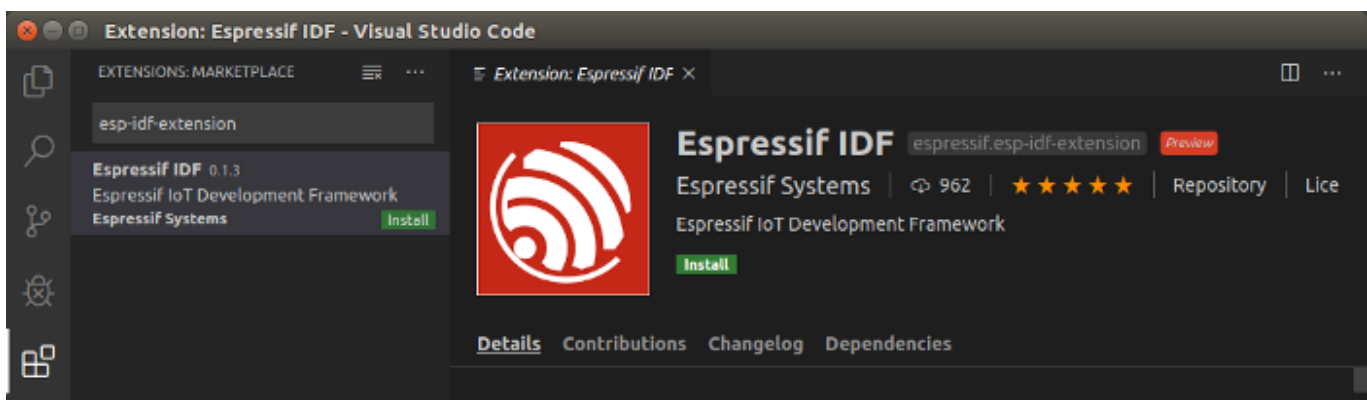
**Now re-login to enable read and write permissions for the serial port.**

We are ready to start experimenting with our board. But I recommend make backup original firmware before operating with the board. We will read all (4MB) flash memory from address 0 to 0x400000.

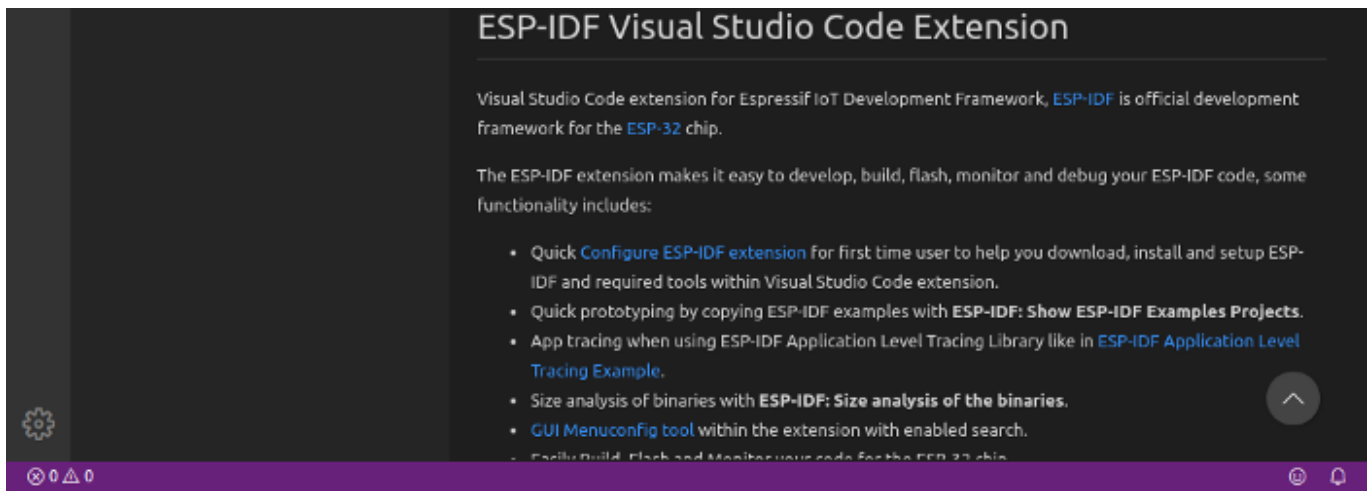
```
~/esp/esp-idf/components/esptool_py/esptool/esptool.py -p  
/dev/ttyUSB0 -b 460800 read_flash 0 0x400000  
~/esp/backup/ttgo_cam_original.bin
```

Now, download and install the Visual Studio Code from the [official site](#).

Run VSCode. Select *Menu -> View -> Extensions* and enter in the search field: `esp-idf-extension`







esp-idf-extension found in Marketplace

Then press “Install” and follow the [Configure ESP-IDF Extension manual](#).

If you successfully completed all previous actions you are ready to run the first project.

• • •

## 2. ESP32 Hello World

It is time to start our trivial first ESP32 project. Since we have already backed up firmware we can confidently start experiments.

Before using ESP IDF in the terminal we will need to activate ESP IDF environment variables for the current terminal session. Run following command in terminal:

```
get_esp32
```

Let’s copy “Hello World” from ESP IDF examples.

```
cd ~/esp/projects
cp -r $IDF_PATH/examples/get-started/hello_world .
cd ~/esp/projects/hello_world
```

At the next step, you may need to config project build configuration but you can skip this step because now we do not need any specific settings.

```
# You can skip this step  
idf.py menuconfig
```

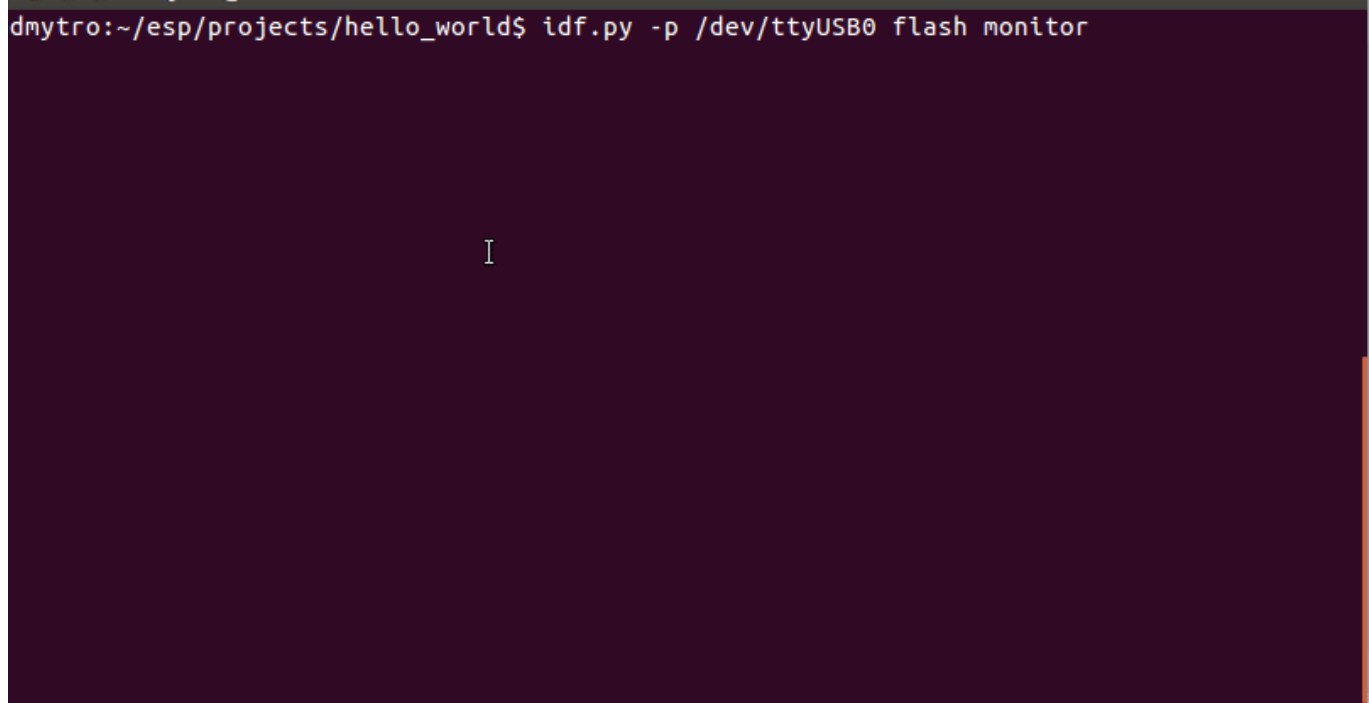
Build the project by running:

```
idf.py build
```

If the previous steps have been done correctly we have firmware binary .bin file which we should send to our board (or flash it). Let's flash the firmware and run the monitor to see the output in our terminal. Press `Ctrl+J` to exit from the monitor.

**NOTE:** Be sure your board is connected to your development computer with the USB cable and remember to specify the correct port `tttyUSB*`.

```
idf.py -p /dev/tttyUSB0 flash monitor
```



```
dmytro:~/esp/projects/hello_world$ idf.py -p /dev/tttyUSB0 flash monitor
```

I



Great!

Now, “Hello world” firmware stay in the board ROM memory even you disconnect the power. If then you connect the board to development computer again the firmware starts and you will be able to monitor its output again by just entering the command:

```
idf.py -p /dev/ttyUSB0 monitor
```

. . .

### 3. TensorFlow Lite Hello World

Now, you are ready to deploy the ML application to ESP32. I suppose you have experience with TensorFlow and already cloned the TensorFlow repository to your development computer.

The TensorFlow team published an awesome tutorial to deploy their “Hello World” application. Now, you have some experience with the ESP IDF framework and I believe it would be easy for you successfully follow TensorFlow [Hello World example for ESP32](#). You should receive a similar output:

```
dmytro: ~/esp/projects/hello_world_tf$
```

I

## Flashing and monitoring ESP32 Hello World sample from TensorFlow Lite

Great! But it still looks like magic. Ok. Let's try to cook handmade magic.

. . .

## 4. Let's type more with your fingers

**Note:** TensorFlow Lite for Microcontrollers is still an experimental port and codebase changes every single day. Probably you would bump into some other issues than me.

To build an application from scratch with ESP IDF we have to understand how the build system works. The [ESP IDF build system](#) is a CMake-based build system, which is the default since ESP-IDF V4.0. It uses the concept of “components” to structurize the project. An [example of the project directory tree](#) we can find in Espressif documentation:

```
myProject
├── CMakeLists.txt
├── sdkconfig
├── components
│   ├── component1
│   │   ├── CMakeLists.txt
│   │   ├── Kconfig
│   │   └── src1.c
│   └── component2
│       ├── CMakeLists.txt
│       ├── Kconfig
│       ├── src1.c
│       └── include
│           └── component2.h
├── main
│   ├── CMakeLists.txt
│   ├── src1.c
│   └── src2.c
└── build
```

“Component” is a code that implements holistic functionality or a library. It could be OLED screen driver or implementation of web-server or TensorFlow Lite for Microcontrollers!

Another important point is the `CMakeLists.txt` file. Each project has a single top-level `CMakeLists.txt` file that contains build settings for the entire project. Also, each

component has `CmakeLists.txt`. Starting from top-level `CmakeLists.txt` the ESP-IDF build system recursively evaluates project dependencies. I strictly recommend you carefully read the [original documentation](#) to the exact understanding of the project build process.

If we look now at the TensorFlow Lite Hello World project we will understand its internals. In the list of files below, I omitted files that we not interested in at the moment, just for show structure.

```
hello_world
├── esp-idf
│   ├── CMakeLists.txt
│   ├── components
│   │   └── tfmicro
│   │       ├── CMakeLists.txt
│   │       ├── tensorflow
│   │       └── third_party
│   └── main
│       ├── CMakeLists.txt
│       ├── esp
│       │   └── main.cc
│       ├── main_functions.cc
│       ├── main_functions.h
│       ├── sine_model_data.cc
│       └── sine_model_data.h
```

Next .cc source files contain:

`main.cc` — program (application) entry point,

`main_functions.cc` —TensorFlow pipeline,

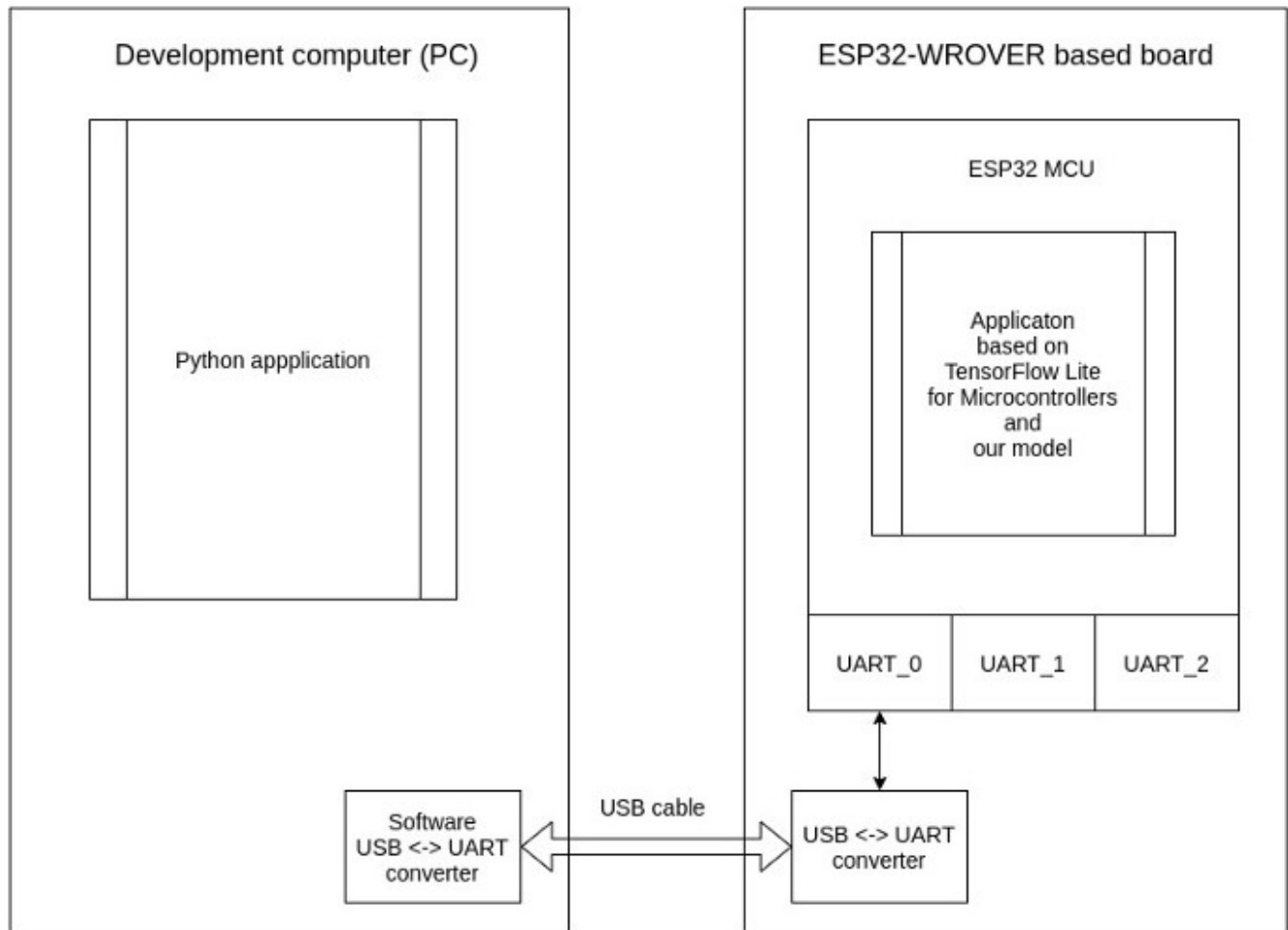
`sine_model_data.cc` — the Model represented by an array.

The official TensorFlow documentation [recommends](#) using the Hello world project as a template for your own projects.

**Now we are ready to implement our own application.**

**The idea:** Sometimes it makes sense using more than one MCU in the IoT project. In this project, I decided to consider ESP32 MCU as a standalone computational device. In my case, I did not want to dig in ESP32 camera API and wanted to be focused on ML functionality. I wanted just only send data from my PC to the microcontroller to feed NN

and then receive results back. Almost all microcontrollers have at least one UART controller which makes such an approach quite universal (at least for testing your NN on certain hardware).



PC will send data to the microcontroller to run inference on the NN

In general, building an ML project for a microcontroller consists of two stages: ML-related (building model, optimizing, converting) and microcontroller-related (deploy the model to microcontroller and integrate with the business logic) part. Thus, we will need three things: the Model, the TensorFlow Lite and the Data source.

### **The first stage. Building model. Converting it to TensorFlow Lite.**

We need to build the TensorFlow Lite model. For beginners in TensorFlow Lite, I recommend to complete the [Introduction to TensorFlow Lite from Udacity](#) and follow the TensorFlow Team tutorial on the [preparation model for the Hello World project](#).

I used the Fashion MNIST data to build model with input images 14x14 pixels size. This is very similar to what is shown in the mentioned Udacity course. The structure of my model is shown below:

```
model = tf.keras.Sequential([
    Conv2D(4, 3, activation='relu', input_shape=(14, 14, 1)),
    AveragePooling2D(),
    Conv2D(6, 3, activation='relu'),
    Flatten(),
    Dense(10, activation='softmax')])
```

Further, I believe you have already built and converted the TensorFlow Lite model to `model_data.cc` file and prepared test .jpg samples 14x14 pixels size.

## Second stage. Building and deploying the application.

Start from cloning [esp-iot-solution](#) repository. Then copy the [empty\\_project](#) and use it as the base of our ESP32\_Fashion\_MNIST project.

Copy TensorFlow Lite for Microcontrollers **component** `components/fmicro` from TensorFlow Hello World project to `components` of our project. Remove `component1` and `component2` from the project. Put the `model_data.cc` file to the `main` directory. Now we need to connect all it together. Here I explain the only pipeline. All the details you can see in the [source code](#).

For convenience, we will put the pipeline in the `main.cc` file and other useful code will put in `main_functions.cc` file.

For our pipeline will need the logger `MicroErrorReporter` to receive messages if something went wrong.

```
static tflite::MicroErrorReporter micro_error_reporter;
error_reporter = &micro_error_reporter;
```

The pipeline is very similar as we use in python projects but implemented in C++. We prepare the `model` based on `model_data.cc` file that we already prepared during the ML

stage.

```
model = tflite::GetModel(fashion_mnist_model_tflite);
```

Instantiate operations resolver that contains operations needed to operate with model.

```
tflite::OpResolver &resolver = getFullMicroOpResolver();
```

Here I use not optimal resolver that contains all supported by TensorFlow Lite for Microcontrollers operations. In production (and upcoming) applications recommended using the subset of operations that only really mentioned in your computational graph. I have implemented the stub in my project.

```
// Use optimized resolver in production applications  
// to use memory only by really needed operations  
tflite::OpResolver &resolver = getOptimizedMicroOpResolver();
```

Instantiate interpreter to operate with the model. Here you will see the `tensor_arena` — space where TensorFlow Lite will allocate input and output tensors and other memory objects.

```
tflite::MicroInterpreter static_interpreter(model, resolver,  
tensor_arena, kTensorArenaSize, error_reporter);
```

Now we are ready to allocate tensors.

```
interpreter->AllocateTensors();
```

and obtain pointers to input and output data buffers:

```
input = interpreter->input(0);  
output = interpreter->output(0);
```

We are ready to receive data and feed it to the model with the interpreter. As I mentioned earlier, I am going to receive data (14x4 pixel monochrome) from my development PC with the UART controller.

```
readUartBytes(input->data.f, totalExpectedDataAmount);
```

Then run inference.

```
interpreter->Invoke();
```

Read predictions from the NN output and send results to the development PC with UART.

```
sendBackPredictions(output);
```

To communicate with ESP32 MCU I have implemented the iPython notebook where I send 50 .jpg files to MCU and compare results of inference with results obtained with TensorFlow Lite on PC computer.

Now, flash your board with a sketch then run [the iPython notebook](#) and see results.

The full source code is in [GitHub project repository](#).

## Conclusions

- It works! But you can run into bugs.
- It is quite easy to start with a new class of computational units like Microcontrollers.
- It would be great to generate OpsResolver automatically as a model\_data.cc.



- The inference is a computationally “heavy” process that could suppress other processes. FreeRtos has a mechanism to solve this problem. But how about using without OS on single MCU?

## What Next?

- Deploy 28x28 input pixels NN with a better accuracy model.
- Measure Inference latency and compare it with other platforms.
- Measure memory consumption.
- Determine maximum model size we can fit in ESP32.
- Develop advanced communicating protocol with MCU.
- How about using SPI RAM (PSRAM) and SPI Flash?

## Useful links

1. [TensorFlow Lite for Microcontrollers](#)
2. [Introduction to TensorFlow Lite from Udacity](#)
3. [Edge TPU live demo: Coral Dev Board & Microcontrollers \(TF Dev Summit '19\)](#)
4. [Espressif modules table](#)
5. [ESP-IDF Programming Guide](#)
6. [ESP32 IoT Solutions repository](#)
7. [Hello World with ESP32 Explained](#)
8. [TensorFlow, Meet The ESP32](#)
9. [One large microcontroller, or lots of small microcontrollers?](#)

Tensorflow Lite

Microcontroller

Esp32

IoT

Machine Learning



[About](#) [Write](#) [Help](#) [Legal](#)

---

Get the Medium app

