Member-only story

Machine Learning on a Microcontroller Board

Running a TensorFlow Lite Micro Model on an ESP32

Leslie Foster Follow 15 min read · 4 days ago

20 2

An article from last month (https://medium.com/@lfoster.se.be/data-capture-for-machine-learning-25787e3237f3) told of how an STM32 “blue pill” microcontroller board could be used to *capture* data that was then used to train an ML model. To follow up, this article judges which of the two knock patterns a user is doing, based on the final model. While the capture operation was done using the blue pill, the predictions run on an ESP32. During the capture step, a pair of joined breadboards, a vibration sensor and a USB linkage were set up, and the data was fed into a Python TensorFlow

process running in a Jupyter notebook. It was initially hoped that the model could be used on the STM32 itself to carry out “predictions” or “inferences” against new data. Instead, the smaller yet more powerful ESP32-C6 Xiao board runs the application. The training data was captured on a blue pill, trained in a Jupyter notebook, and deployed to an ESP32.

> *If you are not a Medium member, here is a “friend link”, so you can still read about this topic.*
>
> *https://medium.com/@lfoster.se.be/whos-knocking-your-microcontroller-can-tell-29d6e584f10e?sk=6bdffe54e85931cbf279e89678dd8a22*

The ML Capture Project

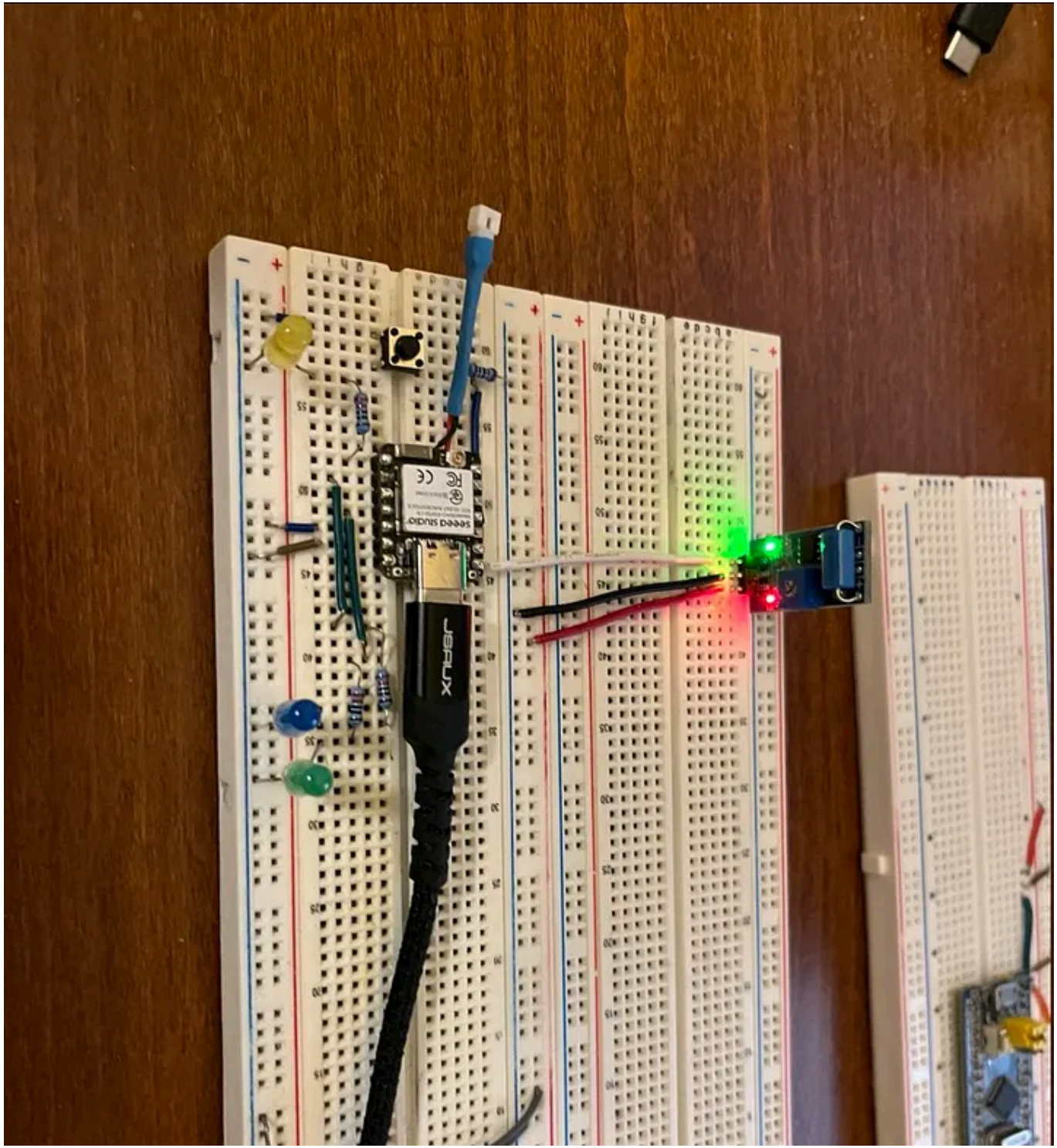
Let’s review the “capture” project briefly. Its main goal was to exploit a very small development board as a custom peripheral to grab data. It was meant to be cheap, easy and fast to construct, but still reliable enough to do the job. Although cumbersome, with its screen scraping and PC connection, the circuit allowed data to be collected.

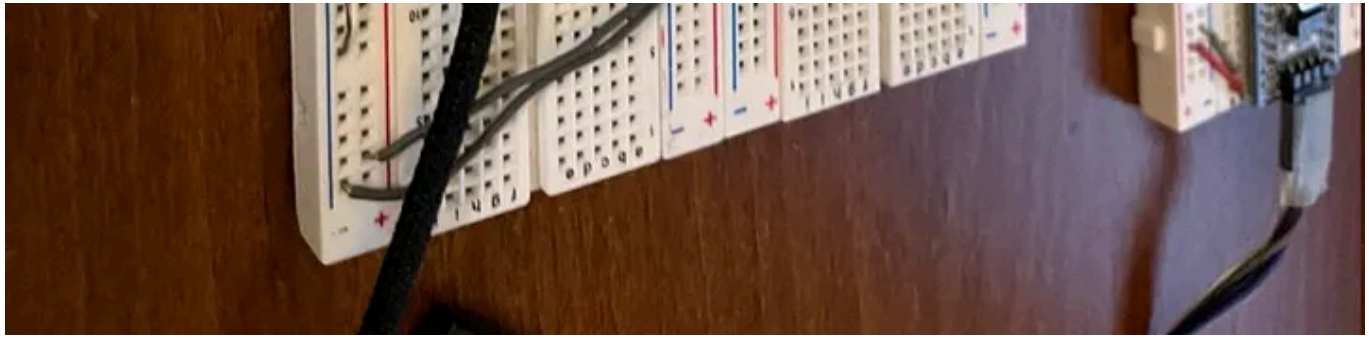
The data being collected were “knock patterns”. Two door-knock sequences from popular culture were repeatedly fed as samples. One was the obvious “shave and a haircut” knock. The other was from the TV program “Doctor

Who": two sets of four equally-spaced knocks. To differentiate the patterns, the time lapses between the raps were used. Fortunately for the project described in this article, the time periods were already saved as standard millisecond intervals, rather than ticks from a specific microcontroller timer.

The ML Inference Project

That standardization made it possible to recognize the knock patterns on a different microcontroller, an ESP32C6. Shown in the image below, this one is on a tiny development board called "Xiao". There are a whole line of these ESP32 Xiao systems, which, like the STM32 are 32 bits. However, unlike that STM32F103C8T6 system, this one has 512KB SRAM and 4MB Flash. Its clock speed is over twice that of the F103. As an aside, please do not get the impression that STM chips are underpowered. The F103 is only one of a large family of STM processors. STM produces offerings that are actually very well suited for Tiny Machine Learning applications. The ESP was chosen here because it had previously been purchased, was powerful enough, and also cheap at around $12 each. They are available from Amazon or other Seeed Studio outlets.


seeed studio
JBALIX



The ML Inference Setup with the ESP32C6, indicator LEDs and Vibration Sensor for detection. In the background, the STM32F103C8T6 “Blue Pill” that was used to collect data

Some Background

Microcontrollers, as discussed elsewhere in this series of articles, are relatively low powered and slow compared to a desktop, laptop or even a modern smart phone. However, they have been getting steadily more powerful while using less and less actual current, over the last few decades. They take very little space, are self-contained for their intended purposes (requiring no external memory, often; requiring no hard drive for program load; usually not being connected to a display). They frequently have bundled on-chip “peripherals” for specialized signalling and communications. This ESP32-C6 setup is specially packaged with a lot of extras, and can run with very little current. That low-current feature is not exploited here, but it was in the previous article https://medium.com/@lfoster.se.be/esp32-c6-wifi-strength-meter-8f9a28ade3aa, in which it was put to sleep when not in use. As a comparative

reference to how powerful MCUs have become, consider that circa 2000, a reasonably powerful *desktop* would have a CPU about as powerful as this one. We got a lot done around the year 2000.

As these micro-marvels have gotten so powerful in recent decades, it has become possible to offload some AI tasking onto them. MCUs have always been involved in IoT (Internet of Things/Edge Computing), being the local brains of the operation. With sensor data collection and effector operation, the MCU would frequently either offload its data or receive instructions from a connected network. However, as more was learned about IoT, it was realized that this poses problems. For one, there can be a time lag as data are pushed back and forth to some decisioning system. For another, they can become a security issue, transmitting a lot of data. Possibly worse still, using communications uses more energy, and some deployment environments do not offer much.

So, in recent years, more attention is being paid to allowing the MCUs at the edge to run their own models. Imagine this typical scenario: the edge device has a model for anomaly detection. When triggered, it makes the decision to turn off equipment to avoid damage — only communicating that such action has been taken.

This project, then, illustrates how getting a model onto an MCU is possible, and shows a rather simplistic application. The implementation uses a vibration sensor priced under $2 to detect knocks. The MCU records the intervals between successive knocks, feeds the full sequence into an “Input Tensor”, runs an inference (asks for a prediction) and the picks up the data in the “Output Tensor”. This output will be either a 1 for one knock pattern or a 0 for the other pattern. This could have been accomplished — given these patterns, which differ in length — by just counting the knocks. It could also have been done with interval measurements having a tolerance. Like some other examples out there (the sine wave; computing the sum of two numbers), this one may not require ML. However, the model is actually in use. Predictions are being made. And the accuracy is sufficient, even with only 100 samples.

Note that all of the training for this model was done in a Jupyter notebook. Once the model is running on the device, the model is used to categorize its incoming data by differentiating between one knock pattern and the other.

Getting the Model Onto the Device

There were a few things to be considered in adapting this model from a simple Python program to an MCU application. What made it possible was TensorFlow Lite Micro. Micro is for very low power environments.

Remember, many AI applications, like ChatGPT, will run on whole server farms. Although powerful for its tiny package, this is still one system and it does not have a lot of wiggle room. The Micro variant is written in C++, which is fine for modern microcontrollers with even the 20Kb we see on the STM32F103C8T6 (as an aside, an experienced C++ programmer might succeed in running the model on the F103). But it means that we have to move from the Python notebook to an IDE. This requires a more complicated development cycle, in which we build the output with a special program, and burn it onto the Flash of the ESP32-C6. We will refer to it as the "ESP", "ESP32" or by its full name from here.

The model's transition from Python requires dumping a file, converting it, and then using a utility to turn it into C code.

```
model.save("c:/Users/lesfo/OneDrive/Documents/STM32/knock_data/knock_model.keras
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
open("c:/Users/lesfo/OneDrive/Documents/STM32/knock_data/knock_model.tflite", "w
```

The dump code is shown above. First save the model as a keras file. Then make a converter, run the converter, and finally write the model back as a

tflite file.

The tflite file is still not suitable. There is a utility available in Linux (WSL was used here, but it probably exists in git bash's environment) called "xxd". Here is how that can be run.

```
xxd -i knock_model.tflite > knock_model.h
```

This produces a C header file. With some modifications, the C header served as the in-memory model source.

```
#ifndef KNOCK_MODEL_H_
#define KNOCK_MODEL_H_

#ifdef __cplusplus
extern "C" {
#endif

// This file was generated by the command:
// xxd -i knock_model.tflite > knock_model.h

unsigned char knock_model_tflite[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
  0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
```

```
  0x08, 0x00, 0x04, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
  0x98, 0x00, 0x00, 0x00, 0xf0, 0x00, 0x00, 0x00, 0xb8, 0x05, 0x00, 0x00,
  0xc8, 0x05, 0x00, 0x00, 0x28, 0x0a, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,
  0x01, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0a, 0x00,
  0x10, 0x00, 0x0c, 0x00, 0x08, 0x00, 0x04, 0x00, 0x0a, 0x00, 0x00, 0x00,
  0x0c, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x3c, 0x00, 0x00, 0x00,
  0x0f, 0x00, 0x00, 0x00, 0x73, 0x65, 0x72, 0x76, 0x69, 0x6e, 0x67, 0x5f,
...
  0xf4, 0xff, 0xff, 0xff, 0x19, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x19,
  0x0c, 0x00, 0x0c, 0x00, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00,
  0x0c, 0x00, 0x00, 0x00, 0x09, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09
};
unsigned int knock_model_tflite_len = 2700;
#ifdef __cplusplus
}
#endif
#endif
```

Like any C/C++ header, this just needs to be imported. The uniqueness define block was used to avoid clashes if this were to find itself in multiple different headers all being used in the same program. The other ifdef-wrapper is to allow use with either C or C++.

The setup code below was used to leverage the model.

```
// The name of this function is important for Arduino compatibility.
void setup() {
  if (setup_io_pins() != ESP_OK) {
```

```
  MicroPrintf("Failed to setup IO pins.\n");
  return;
}

// Map the model into a usable data structure. This doesn't involve any
// copying or parsing, it's a very lightweight operation.
model = tflite::GetModel(knock_model_tflite);
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;
}

// Pull in only the operation implementations we need.
static tflite::MicroMutableOpResolver<6> resolver;
  if (resolver.AddDequantize() != kTfLiteOk) {
    return;
  }
  if (resolver.AddFullyConnected() != kTfLiteOk) {
    return;
  }
  if (resolver.AddQuantize() != kTfLiteOk) {
    return;
  }
  if (resolver.AddRelu() != kTfLiteOk) {
    return;
  }
  if (resolver.AddReshape() != kTfLiteOk) {
    return;
  }
  if (resolver.AddSoftmax() != kTfLiteOk) {
    return;
  }

// Build an interpreter to run the model with.
static tflite::MicroInterpreter static_interpreter(
    model, resolver, tensor_arena, kTensorArenaSize);
```

```
  interpreter = &static_interpreter;

  // Allocate memory from the tensor_arena for the model's tensors.
  TfLiteStatus allocate_status = interpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    MicroPrintf("AllocateTensors() failed");
    return;
  }

}
```

Here we see a hint about how the ESPs are marketed. Arduino code uses a “setup()” and a “loop()” method. It is akin to the “template method” pattern to those familiar. Some operations are carried out for the convenience of the system itself, after which the user’s code is run from methods with convenient names (every Arduino programmer is aware of setup() and loop() methods). This is not an Arduino project, but perhaps with appropriate attention, it could be. There have certainly been plenty of projects running TinyML / TensorFlow Lite Micro on Arduino-compatible devices.

However, this project embraced mostly ESP32 conventions.

You can see above the steps for getting the model, establishing the operations resolver, creating the interpreter and allocating tensors. Once these are done, predictions can be made with the methods below.

```
static void set_input(const uint16_t* input_data, int length)
{
    TfLiteTensor* input = interpreter->input(0);
    // Pass raw values without normalization (matching updated training approach
    for (int i = 1; i < 8 && i < length; i++)
    {
        input->data.f[i-1] = (float)input_data[i]; // Direct conversion to float
    }
    input->data.f[7] = (float)length-1; // Set length as the 8th input
     // Pad with zeros if we have fewer than 7 intervals
    for (int i = length; i < 7; i++)
    {
        input->data.f[i] = 0.0f; // Pad with zeros
    }
    printf("input type = %d\n", input->type);
    MicroPrintf("%f %f %f %f %f %f %f %f",
         input->data.f[0], input->data.f[1], input->data.f[2], input->data.f[3],
         input->data.f[4], input->data.f[5], input->data.f[6], input->data.f[7])
}

static void run_inference(void)
{
    TfLiteStatus invoke_status = interpreter->Invoke();
    if (invoke_status != kTfLiteOk) {
      MicroPrintf("Invoke failed with: %d\n",
                           invoke_status);
      return;
    }
}

// Returns the index of the highest scoring prediction
static int get_prediction(void)
{
    TfLiteTensor* output = interpreter->output(0);

    int best_index = 0;
```

```
    float best_score = output->data.f[0];

    for (int i = 1; i < output->dims->data[1]; i++)
    {
        float score = output->data.f[i];
        if (score > best_score)
        {
            best_score = score;
            best_index = i;
        }
    }

    return best_index;
}
```

At inference time, we feed input using the input tensor, then run an inference using the interpreter. Once the interpreter returns, we can fetch back the prediction using the output tensor. Scores are mapped to the ids of choices made. This model has one output tensor and only two classes. The code above starts by assuming the “0th” choice is the best, and subsequently checks any others to see if they scored better. The best score is the most probable correct answer according to the model.

How it is Used

Unlike many other microcontroller projects in this series, this one strongly leverages log output. Since it employs LEDs to signal the choice, this project

could proceed with only a power supply, but if connected to a PC the convenient “monitor” action is supplied by the ESP code.

```
C:\...\ESP32Projects\knock_classifier>idf.py -p COM6 monitor
Executing action: monitor
Running idf_monitor in directory C:\...\ESP32Projects\knock_classifier
Executing "C:\python_env\idf5.3_py3.11_env\Scripts\python.exe C:\tools\Espressif
--- Warning: GDB cannot open serial ports accessed as COMx
--- Using \\.\COM6 instead...
--- esp-idf-monitor 1.5.0 on \\.\COM6 115200
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
ESP-ROM:esp32c6-20220919
Build:Sep 19 2022
rst:0x15 (USB_UART_HPSYS),boot:0x1e (SPI_FAST_FLASH_BOOT)
Saved PC:0x408037de
--- 0x408037de: rv_utils_wait_for_intr at C://esp-idf-v5.3.1/components/riscv/in
 (inlined by) esp_cpu_wait_for_intr at C:/tools/Espressif/frameworks/esp-idf-v5.

SPIWP:0xee
mode:DIO, clock div:2
load:0x40875720,len:0x1974
load:0x4086c110,len:0xfe4
load:0x4086e610,len:0x2f4c
entry 0x4086c11a
I (23) boot: ESP-IDF v5.3.1-dirty 2nd stage bootloader
I (23) boot: compile time Jan 12 2026 21:50:25
I (24) boot: chip revision: v0.1
I (26) qio_mode: Enabling default flash chip QIO
I (32) boot.esp32c6: SPI Speed      : 80MHz
I (36) boot.esp32c6: SPI Mode       : QIO
I (41) boot.esp32c6: SPI Flash Size : 2MB
I (46) boot: Enabling RNG early entropy source...
I (51) boot: Partition Table:
```

```
...
I (287) coexist: coexist rom version 5b8dcfa
I (292) main_task: Started on CPU0
I (292) main_task: Calling app_main()
Entering capture mode
Enabling interrupt to sensor GPIO
Capture end signal received
Exited capture mode. Analyzing intervals...
Captured intervals: 1532, 338, 274, 154, 358, 623, 382, 0
input type = 1
338.000000 274.000000 154.000000 358.000000 623.000000 382.000000 0.000000 6.000
Predicted: 0

Entering capture mode
Enabling interrupt to sensor GPIO
Capture end signal received
Exited capture mode. Analyzing intervals...
Captured intervals: 876, 226, 228, 262, 790, 250, 248, 255
input type = 1
226.000000 228.000000 262.000000 790.000000 250.000000 248.000000 255.000000 7.0
Predicted: 1
```

Above is a session from the IDF “monitor” command. On windows, simply run:

> *idf.py -P comN monitor*

In your project directory, replacing “N” by your ESP-connected Windows COM port, and it will start monitoring. A similar command should work on

Linux. You can add your own output with code like this:

```
      printf("input type = %d\n", input->type);
      MicroPrintf("%f %f %f %f %f %f %f %f",
           input->data.f[0], input->data.f[1], input->data.f[2], input->data.f[3],
           input->data.f[4], input->data.f[5], input->data.f[6], input->data.f[7])
```

As mentioned above, there are also three LEDs and a button on the bread board. The amber LED shows when the knock input has been received (it is easy to knock too hard or too light, so the indicator is good feedback), and the other LEDs indicate which choice the model made. Blue is for “Shave and a Haircut” and Green is for the two-sets-of-four knock. The video link below shows the project in action.

https://youtube.com/shorts/XkiJY0CFPg8?feature=share

Software

As with most or all microcontroller projects, there are two completely different environments where development happens. One is the microcontroller itself, and requires coding plus deployment (flashing the code). The other is a workstation where you can do cross compiling to the

microcontroller's took set. This usually results in a binary in a format such as ".elf".

This ESP32 related article https://medium.com/@lfoster.se.be/esp32-c6-wifi-strength-meter-8f9a28ade3aa goes into more detail about the workstation setup required to work with the ESP32's, but basically if you want to do projects like this one you will have to install the IDF environment on your workstation. Using the ESP-IDF plugin, VS Code integrates quite well with IDF. In addition to the editing environment, there is an ESP IDF command line console that can be launched and run outside of VS Code, which you can use to build the project and flash the code onto the ESP device. And of course, some means of running the "xxd" command would be necessary to turn the "tflite" model into a C / C++ header.

Something that was very helpful about choosing the combination of Tensorflow Lite and ESP, is that the ESP folks already bundled the basic "hello" project for TFLite Micro. Named "hello_world", it simply uses ML to generate a sine wave. Given that, the basic setup is all there and you simply need to modify it. Below is how this project was "bootstrapped".

- idf.py create-project-from-example "esp-tflite-micro:hello_world"
- ren hello_world knock_classifier

- cd knock_classifier
- idf.py set-target esp32c6
- idf.py -p COM6 flash

Just that will put a simple ML application onto your ESP. Of course, ESP32C6 is just one possible target. IDF works for a lot of other Espressif offerings.

The software also leverages ESP's style of interrupt handling, and tasks, tick timer, and queues from FreeRTOS. FreeRTOS is included in the IDF framework.

Components

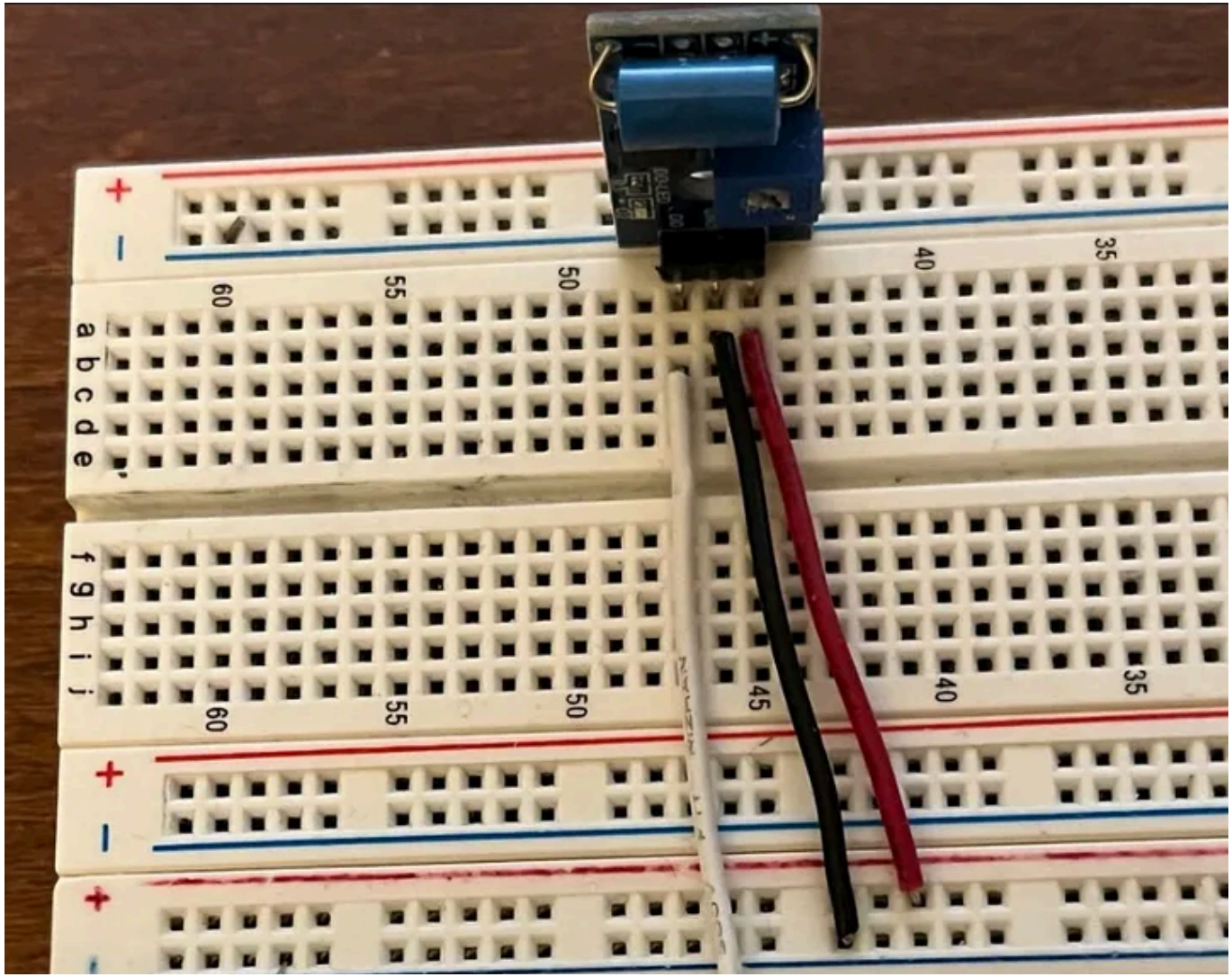
To build this you will need the following. Except where listed otherwise, these components can be found in most electronic hobby kits.

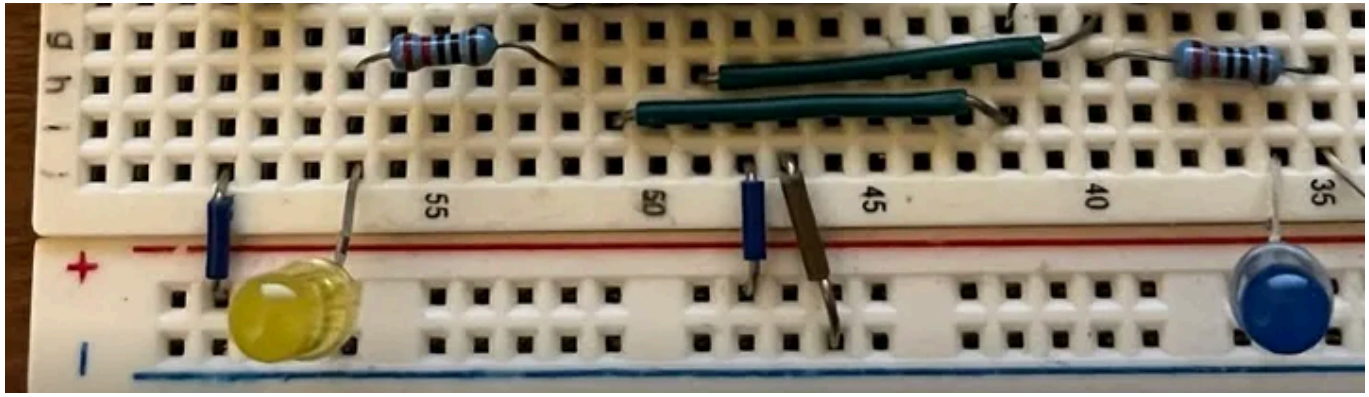
- Breadboard (two are linked here to isolate the vibration sensor, but may not be necessary)
- Wires (dupont cables are fine)
- Three 100Ω, 220Ω or 330Ω resistors for the LEDs
- 10kΩ resistor for the button

- Three LEDs
- Momentary switch
- Vibration sensor. The one below is an SW-420. These can be bought in sets of 5 for ~$6.50 USD. The one below was included in a sensor kit
- Xiao ESP32-C6. There are other ESP32 variants that may work just as well. This code may require considerable memory, so keep that in mind. This one can be had for ~$12.00 USD, and of course can be used in many other projects.

Wiring

As can be seen in the image below, this circuit requires several wires leading into and out of the ESP Xiao board.





Xiao Wiring

The wiring description below assumes the board is aligned as above, with the USB connector on the right.

The low resistance resistors (100, 220 or 330) are connected as follows:

- One has one leg to the lower left pin and the other to the long (anode) leg of an LED.
- One has one leg to the 2nd from lower left pin and the other to the long (anode) leg of an LED.
- One has one leg to the 4th from lower left pin and the other to the long (anode) leg of an LED.

The LEDs are connected to those resistors:

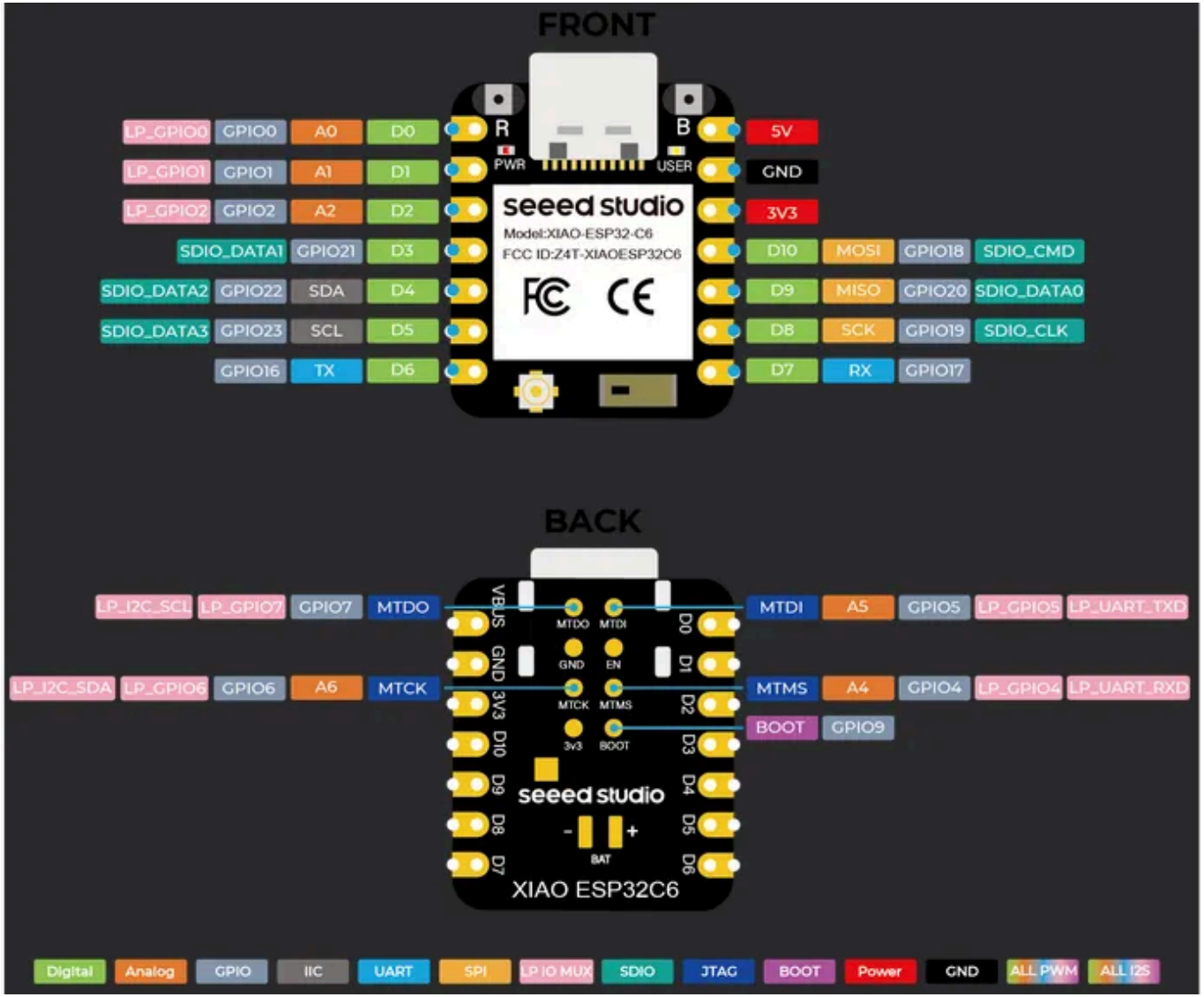
- The shorter (cathode) leg of each LED must be connect to ground. The resistors are necessary to avoid instant burn out of these components.
- The 3rd from lower right pin of the Xiao is the ground pin and should be connected to the breadboard ground.
- The 2nd from lower right pin of the Xiao is the output power and should be connected to the breadboard power bar.

The rest of the circuit is described below:

- The button is wired as shown, with its input going to the 3rd from upper right pin of the Xiao. It's output pin is also connected through the 10kΩ resistor to ground as a "pulldown". Since it is active high, the opposite pin is connected to power. When pressed, a high voltage appears on the output wire going to the Xiao pin.
- The output from the vibration sensor (leftmost as oriented in the image) goes to the upper right pin of the Xiao. Its center is ground and its rightmost pin is power.
- Not shown: power must be connected with wires or dupont cables from the Xiao to the sensor.

The extra shrink wrapped wires sticking out to the left are a secondary power input. After some careful soldering, a rechargeable battery pack can be connected to one of these devices. It is optional, here.

A pinout image is included here for reference.



Reference Image for Pinouts, As can be seen in the data sheet

Discussion

This process started out as an attempt at getting a model onto a much smaller device, but that presented some challenges. For one, the STM32F103C8T6 was not “native” to TensorFlow Lite. It would have been a generic setup. Other STM32s have more documentation around this topic. The memory was a bit too constrained as well, causing some unusual steps to be required.

One challenge with the ultimate setup was arriving at whether to “quantize” or if so, how. In this case, quantization was not needed. It does require some effort to get it right, and a lot of consistency must be maintained between the training code and the deployed code. Had this been the original smaller device, quantization might have saved a great deal of memory.

Input from both VS Code’s builtin Claude Code and ChatGPT were helpful resources in pulling this project together. As always, it is important to be aware of the details. A couple of times in accepting prompt code, some fine points got lost. But, ChatGPT definitely deserves credit for warning that there would be difficulty putting a model onto limited hardware.

Conclusions

To get started with IoT ML, there are some introductory projects using a camera and doing gesture recognition. This project is a bit simpler, probably

uses less memory, and can be done with cheaper components. It is a trivial application but it uses a real model.

Acknowledgements and Sources

Special thanks to Drew Foster for editorial input.

https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/ this is a starting point for ESP32s.

https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/gpio.html ESP technical info on their APIs.

https://documentation.espressif.com/esp32-c6_datasheet_en.html data sheet for ESP32-C6.

https://github.com/tensorflow/tflite-micro using the ESP setup and a hello-world based project, you may not need to pull any other code. But for other projects, this repo may be helpful.

https://docs.nordicsemi.com/bundle/ncs-2.5.2/page/zephyr/samples/modules/tflite-micro/hello_world/README.html additional information on the hello world sine wave project.

Machine Learning | Edge Computing | Edge Ai | Sensors | IoT

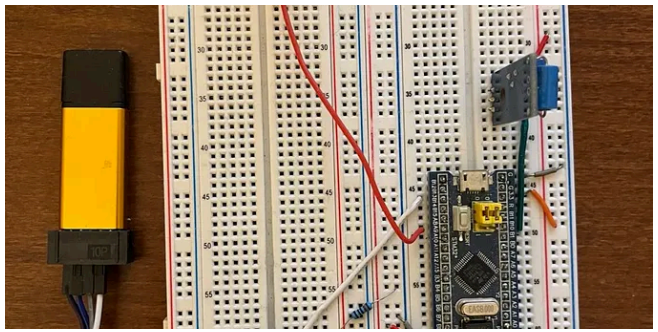
Written by Leslie Foster

61 followers · 5 following

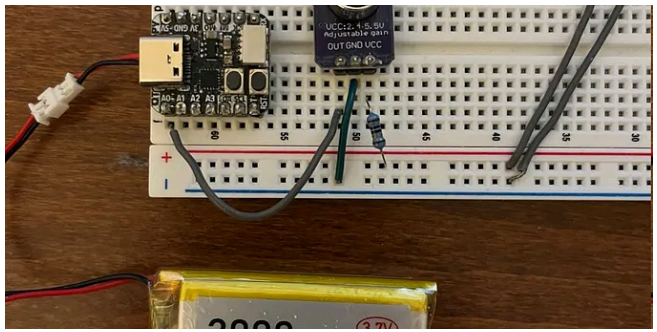
Programmer / Developer / Software Engineer since late 1980s

Follow

More from Leslie Foster



Leslie Foster



Leslie Foster

Data Capture for Machine Learning

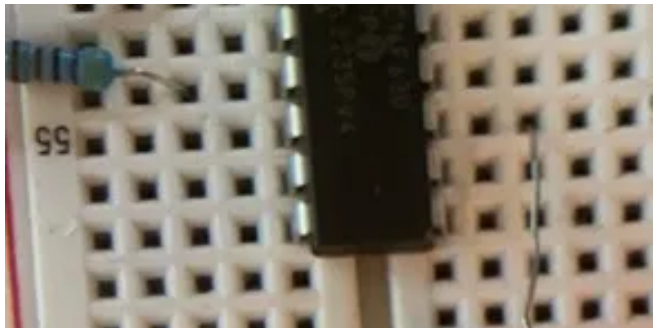
An STM32 Blue Pill becomes a Knock Recognition Collector

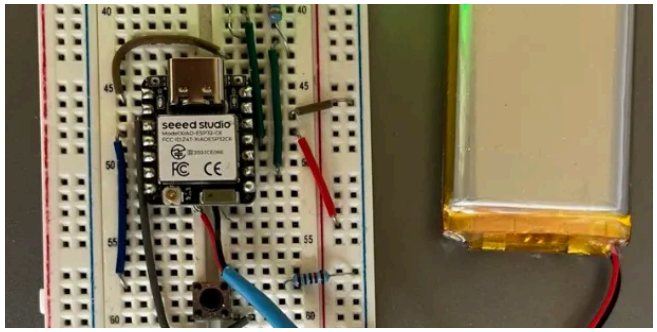
Dec 25, 2025 187 1

A Voice Controlled NeoPixel

A tiny 8051 microcontroller can be festive

Nov 21, 2025 54





Leslie Foster

PIC16F630 Ecosystem

How to start coding for the PIC16F630

Jun 16, 2024 2

Leslie Foster

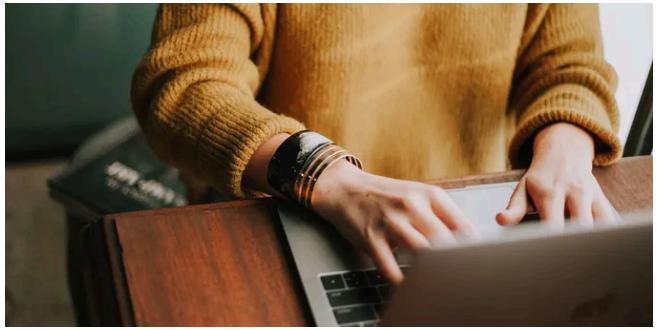
ESP32-C6 WiFi Strength Meter

A Cheap but Powerful Microcontroller Board at Work (without Arduino)

May 4, 2025 2 1

See all from Leslie Foster

Recommended from Medium

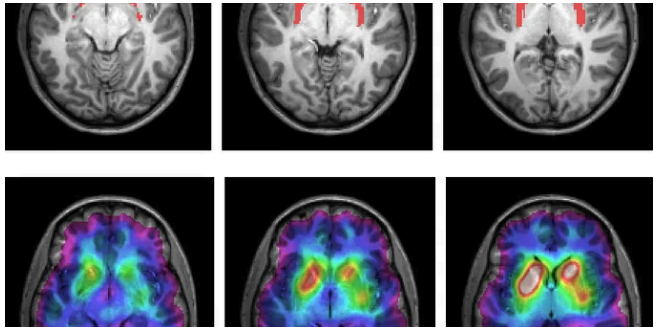


In The Pythonworld by Aashish Kumar

Stop Writing Utility Functions in Python (Here's the Better Pattern)

Utility functions feel productive—until they quietly turn your clean codebase into a...

6d ago 352 7



In Write A Catalyst by Dr. Patricia Schmidt

As a Neuroscientist, I Quit These 5 Morning Habits That Destroy You...

Most people do #1 within 10 minutes of waking (and it sabotages your entire day)

Jan 14 20K 345

Agent Native

Local LLMs That Can Replace Claude Code

Small team of engineers can easily burn >$2K/mo on Anthropic's Claude Code...

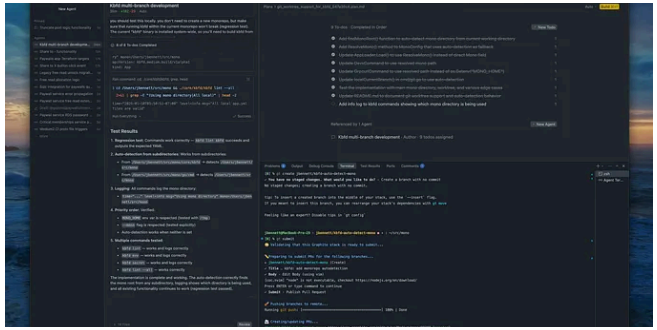
Jan 20 210 7

Nitin Sharma

Forget ChatGPT & Gemini—Here Are New AI Tools That Will Blow...

Here, I'm going to talk about the new AI tools that are actually worth your time.

Nov 17, 2025 2.7K 79



Jacob Bennett

The 5 paid subscriptions I actually use in 2026 as a Staff Software...

Barack Obama

A Wake-Up Call for Every American

The killing of Alex Pretti is a heartbreaking tragedy. It should also be a wake-up call to...

Tools I use that are (usually) cheaper than Netflix

3d ago 54K 743

Jan 18 1.5K 35

See more recommendations