Embedded Swift Workshop

Swift Island 2025

Frank Lefebvre

Preparation

Introduction

Embedded Swift is still a work in progress, and thus it is not possible (yet) to use the version of the Swift compiler that comes with Xcode. We must install a nightly build of the Swift toolchain.

Furthermore, in order to take advantage of the capabilities of the ESP32 microcontroller, we can benefit from components provided by the manufacturer:

- the open source ESP-IDF framework (C/C++) makes it easier to access all hardware capabilities of the microcontroller,
- a build system based on `CMake` and `Ninja` allows to manage dependencies and to link all compiled files (from C and Swift).

Although it is technically possible to use Xcode for Embedded Swift, the recommended environment is Visual Studio Code, with the ESP-IDF extension.

All components can be installed on a standard user account, without requiring admin privileges. You can either use your regular account, or create a dedicated non-admin account on your system, in order to isolate the embedded setup from your day-to-day environment.

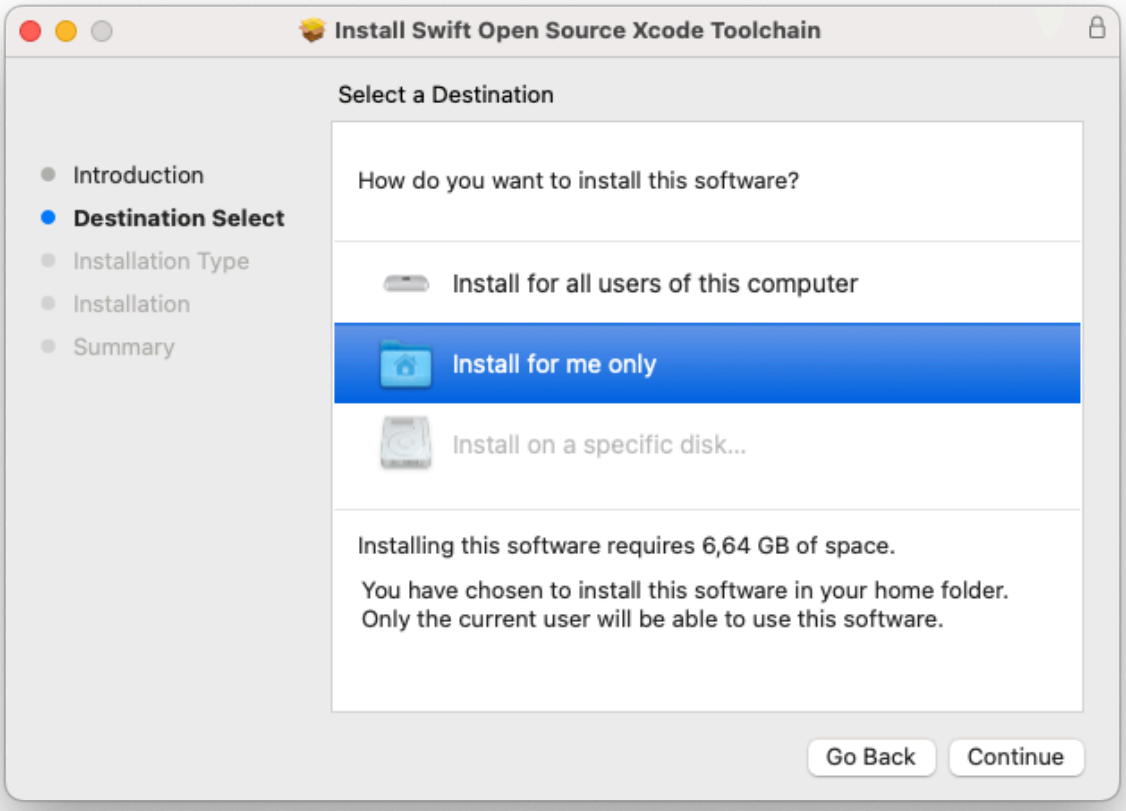
A regular (online) installation requires downloading more than 6 GB of material. For this workshop, you can install the tools from the provided USB drive without the need for an Internet connection.

Installation

Swift Toolchain

The `swift-DEVELOPMENT-SNAPSHOT-2025-08-23-a-osx.pkg` package contains a development build of the Swift toolchain. Installing this package is roughly equivalent to running `swiftly install main-snapshot`.
The toolchain can be installed either for all users (the default option) or for the current user only. Here we'll assume the toolchain is installed for the **current user** (in

`~/Library/Developer/Toolchains`), but the tools are designed to work in both cases.



Visual Studio Code

Extract `VSCode-darwin-universal.zip` and move the `Visual Studio Code` application to either `/Applications` or `~/Applications`.

ESP-IDF Framework & Tools

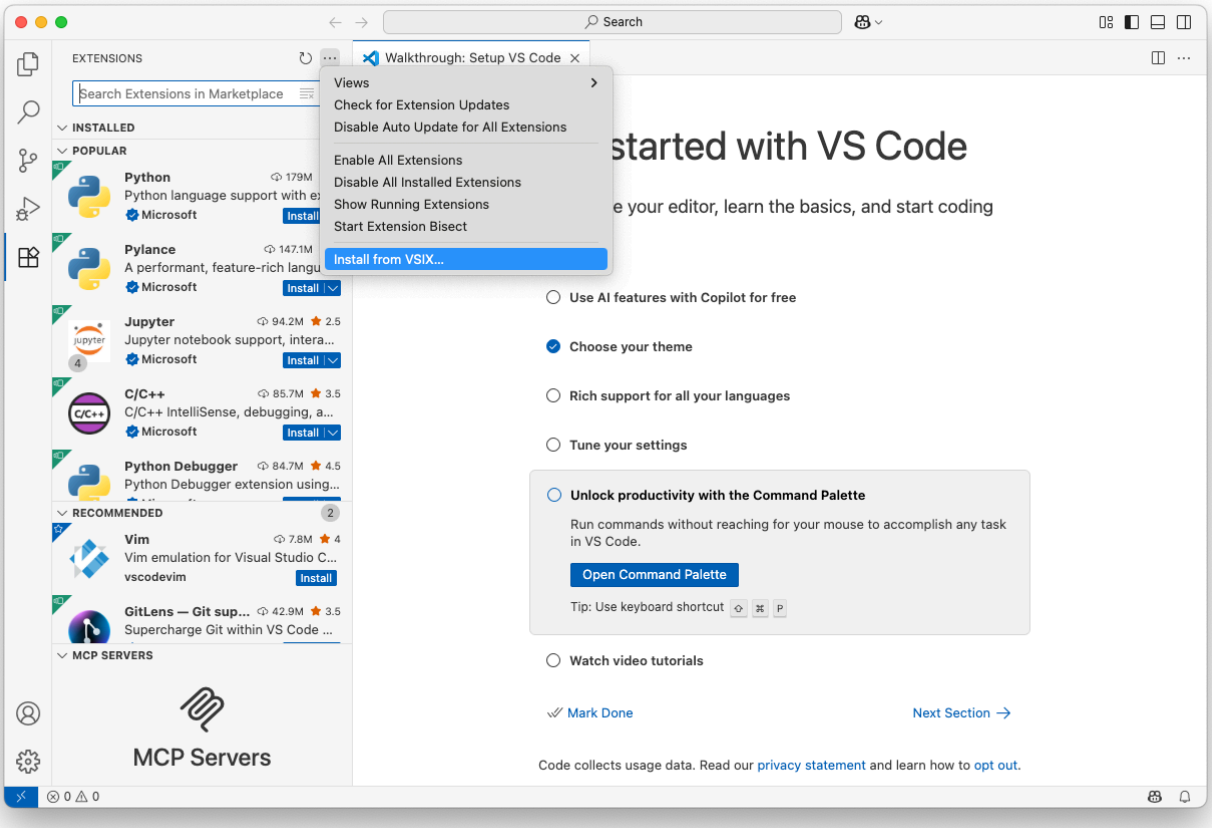
In your home directory, create a folder named `esp`, and extract `esp-idf-v5.5.zip` there. Extract `espressif.zip` in your home directory. This will create a `.espressif` directory.

```
mkdir ~/esp
chmod 700 ~/esp
unzip path/to/esp-idf-v5.5.zip -d ~/esp
xattr -d com.apple.quarantine path/to/espressif.zip
unzip path/to/espressif.zip -d ~
```

ESP-IDF Extension

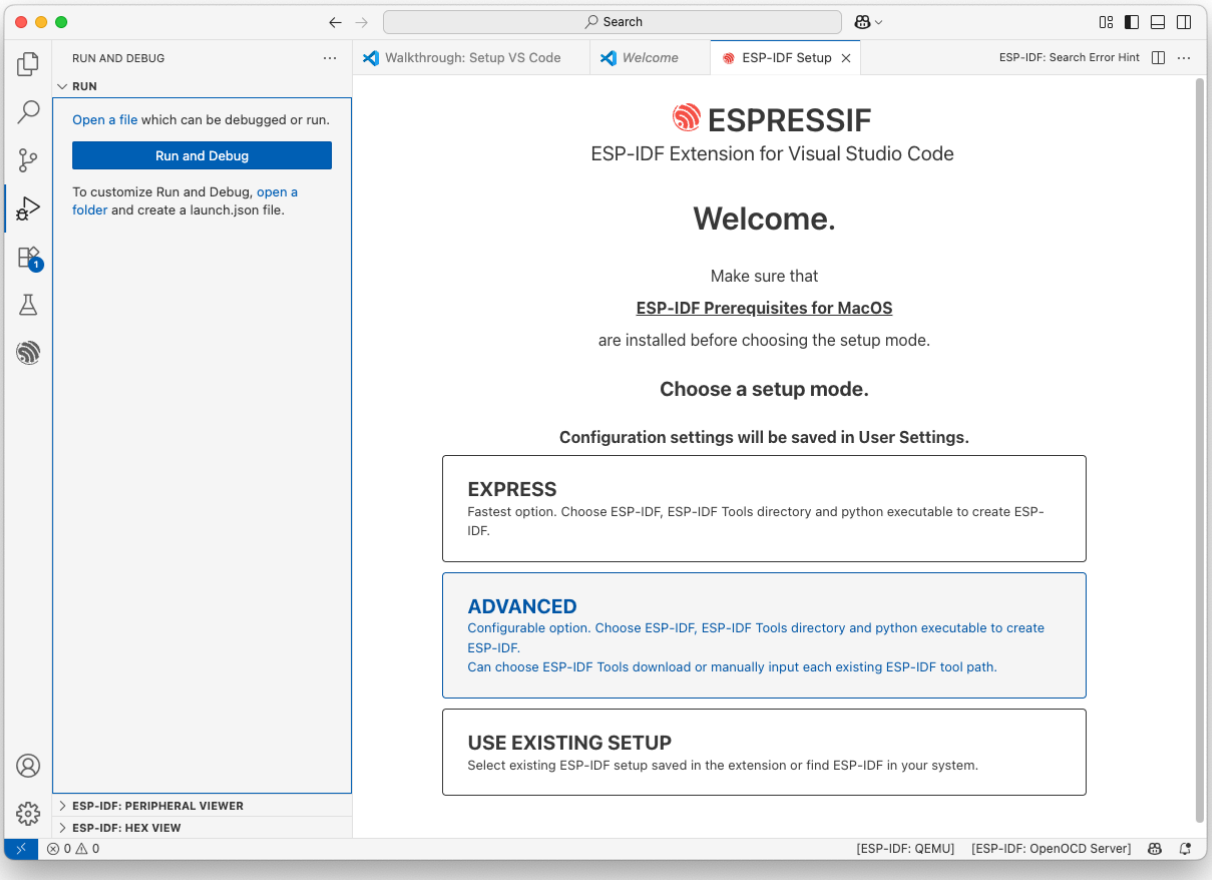
Open Visual Studio Code.
In the Extensions tab in the sidebar, click the `...` menu and choose `Install from VSIX...`

Choose the `esp-idf-extension-1.10.1.vsix` file from the thumb drive.

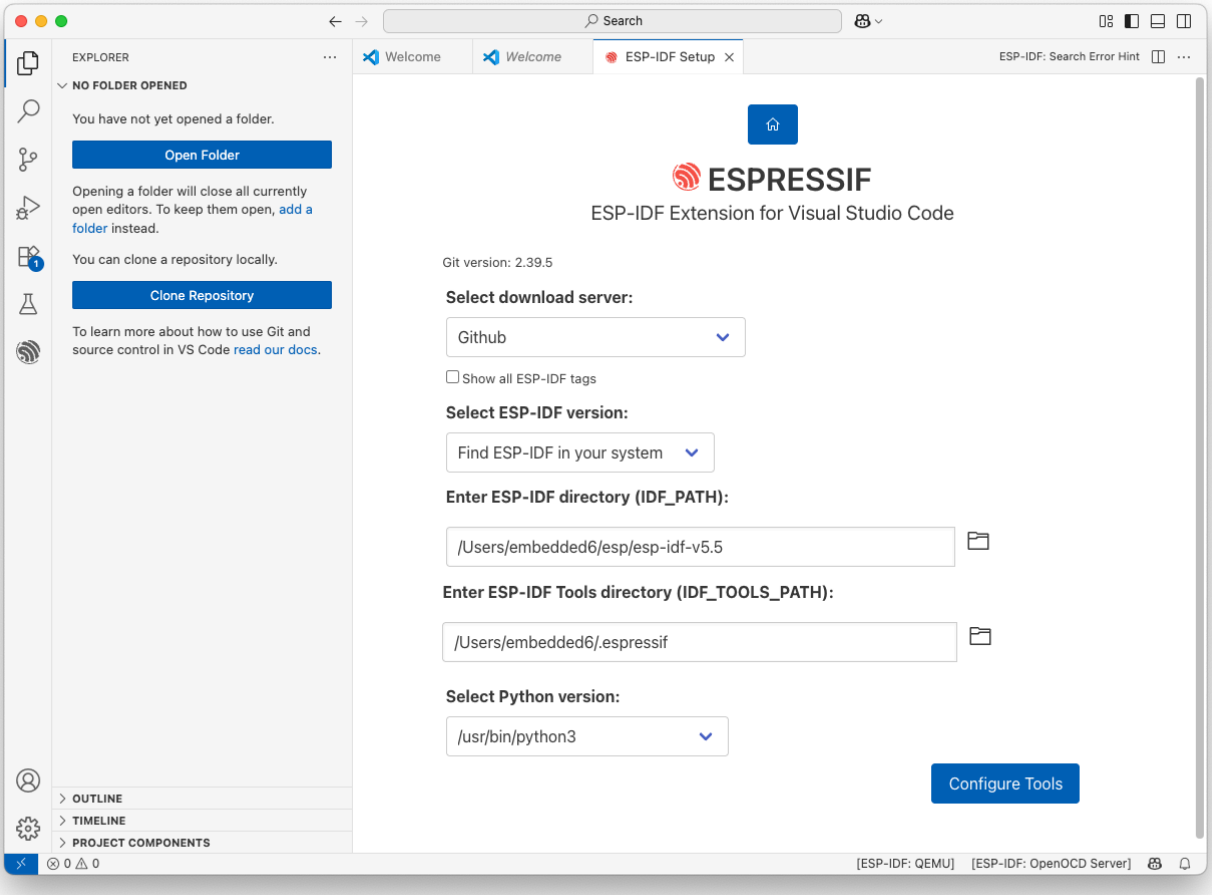


ESP-IDF Extension settings

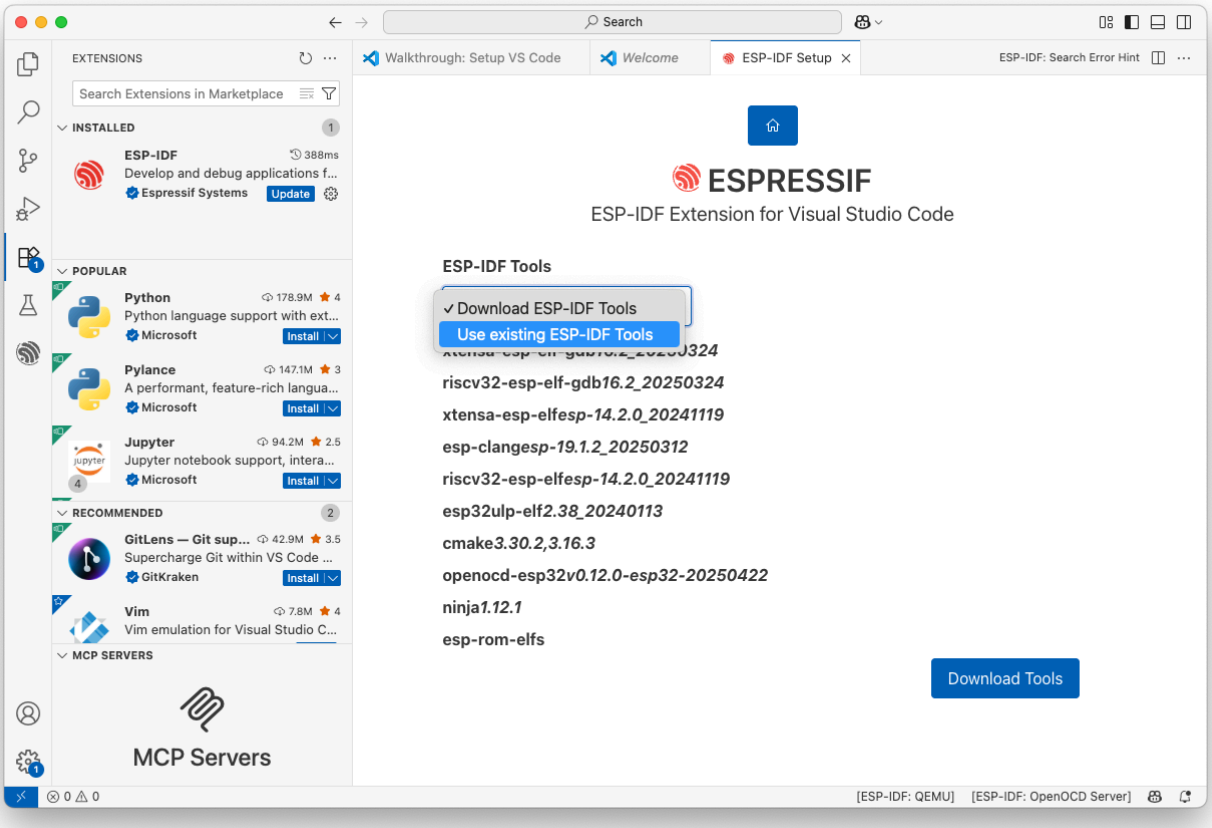
In the Command Palette (shift-cmd-P), choose ESP-IDF: Configure ESP-IDF Extension. Click "ADVANCED", and select the ~/esp/esp-idf-v5.5 directory.



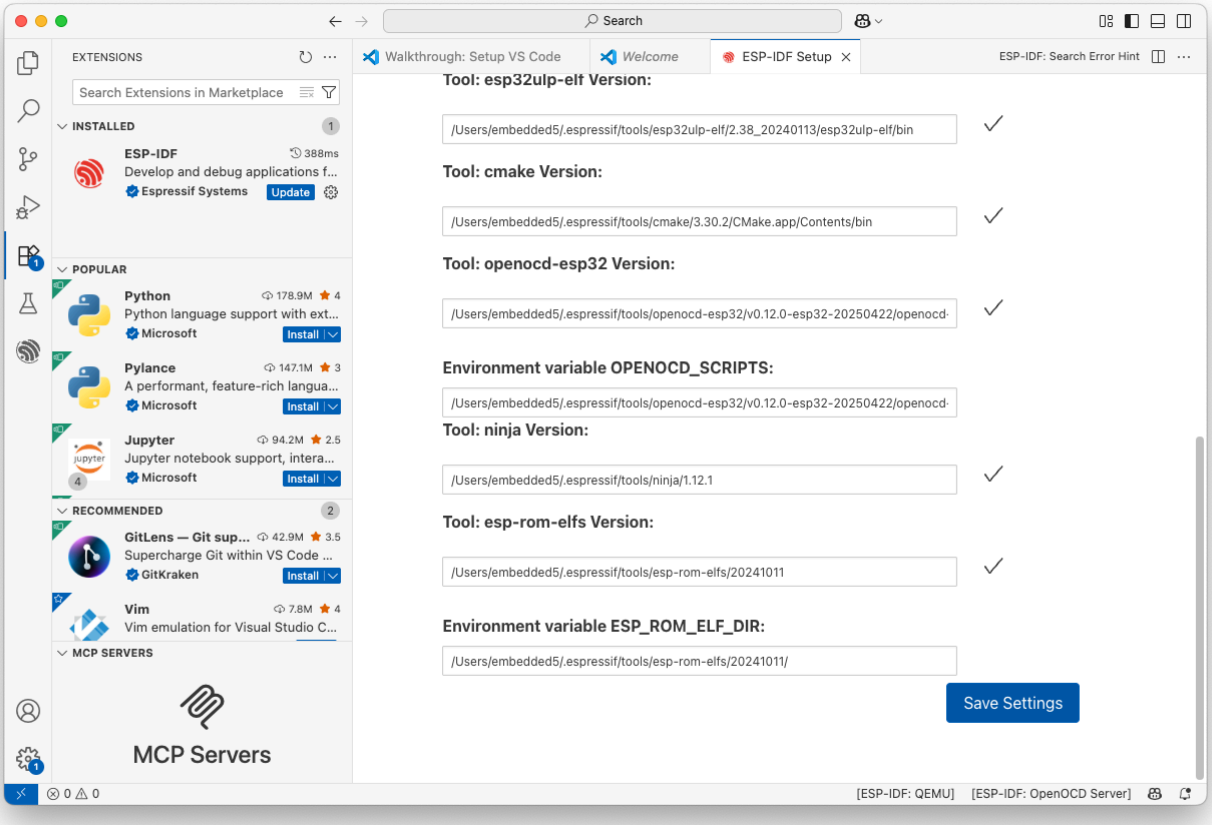
Then for the **Select ESP-IDF Version** option pick "Find ESP-IDF in your system", and if necessary select `~/esp/esp-idf-v5.5` for the ESP-IDF directory.



On the following page, pick "Use existing ESP-IDF tools".



After validating the tools, the plugin will allow you to save the configuration.



esp-setup script

A shell script is provided to make it easier to enable the latest Swift toolchain and to detect the USB port the microcontroller is connected to.

Copy `esp-setup` from the `Workshop Resources` folder to `~/esp`, and make it executable:

```
cp path/to/esp-setup ~/esp/
chmod +x ~/esp/esp-setup
```

In your shell profile (`~/.profile`, `~/.zshrc`, etc), add the following command:

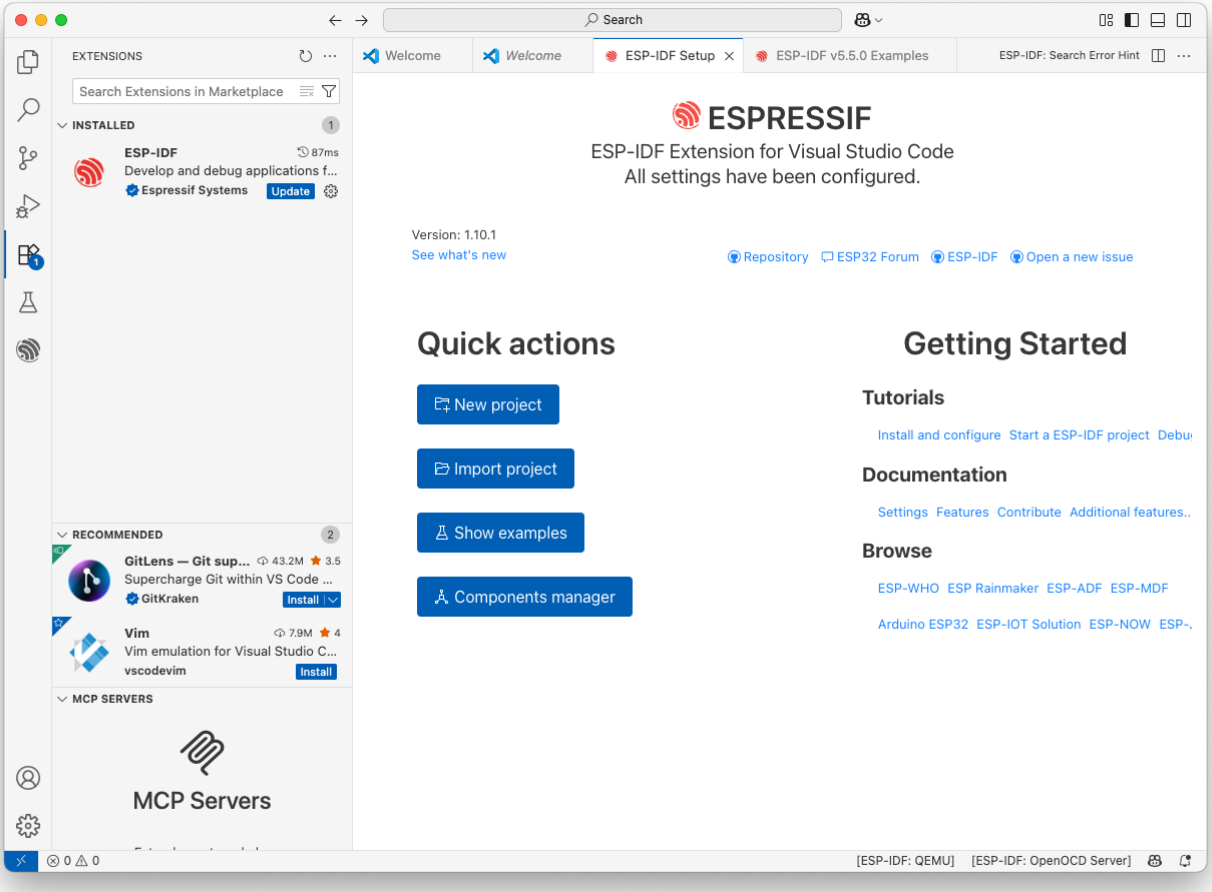
```
alias esp-setup=". $HOME/esp/esp-setup"
```

(the `.` and `space` before `$HOME` are important)

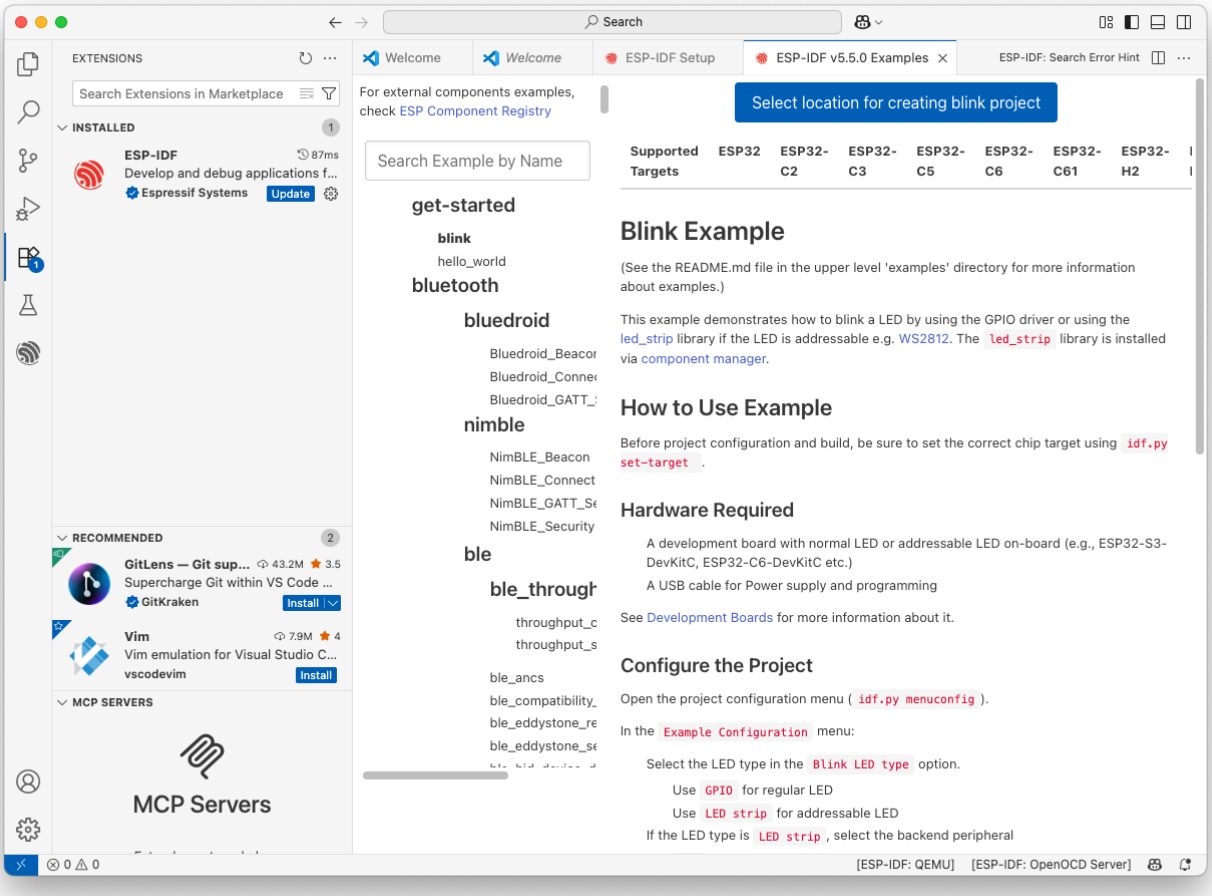
Test

Now it's time to test your configuration, to make sure everything is set up properly.

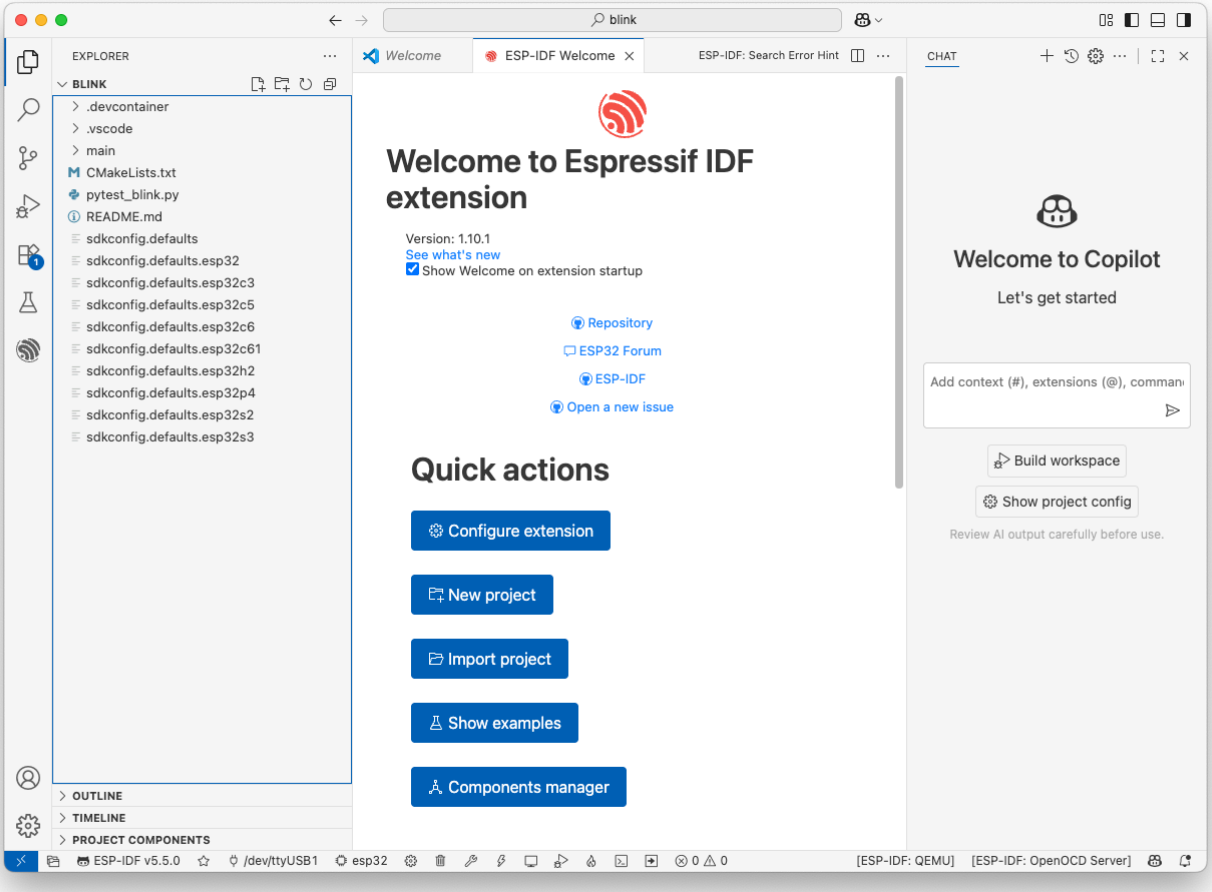
In the ESP-IDF plugin pane, click "Show examples" and choose "blink" in the "get-started" section.



Then click "Select location for creating blink project" and pick a directory.



The resulting project should look like this (in the left pane):



Open a terminal session in VS Code, and type:

```
esp-setup
```

The output should end with something like:

```
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build

Detecting connected device...
Found: /dev/tty.usbmodem3201
```

Type:

```
idf.py set-target esp32c6
idf.py build
idf.py flash
```

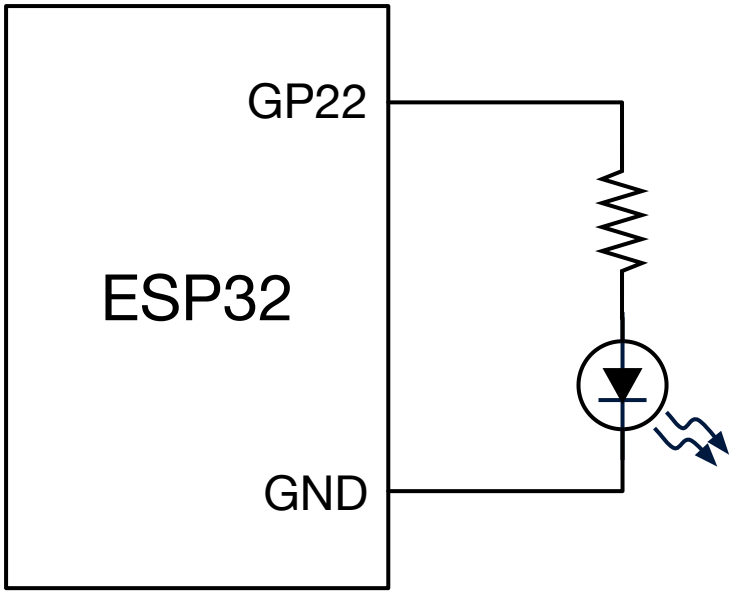
The LED on the ESP32 board should blink.

Pulsing LED

Every embedded journey starts with a blinking LED.

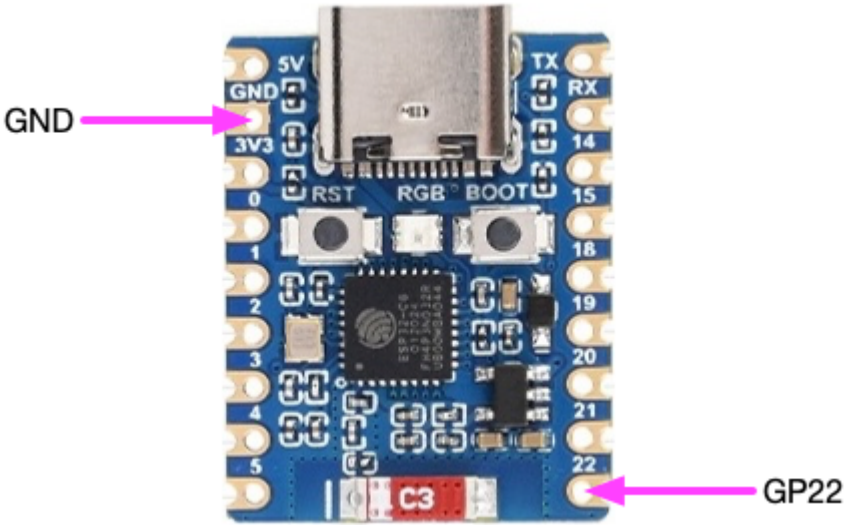
Wiring

Connect a LED to GPIO22, as follows:

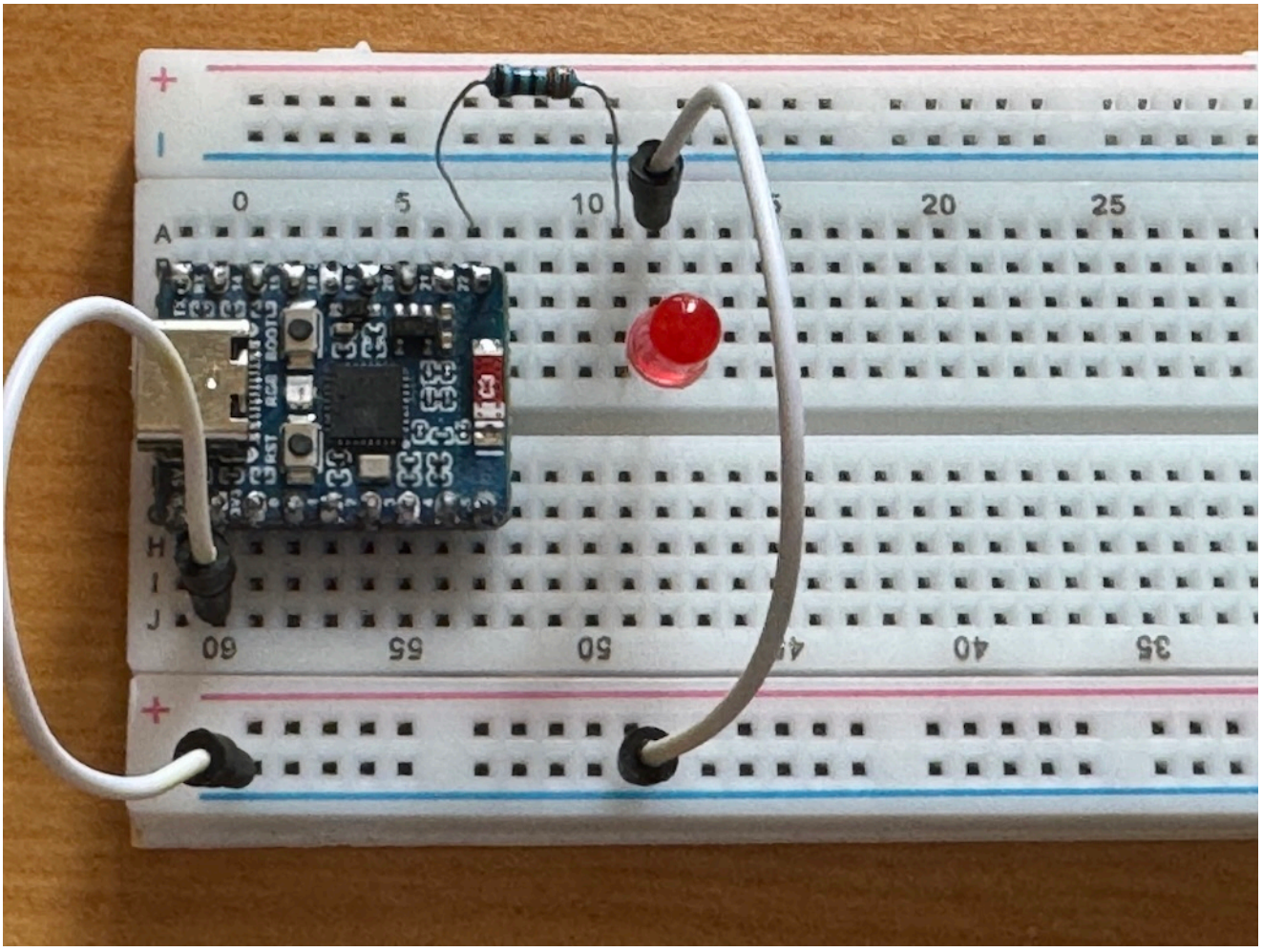


The value of the resistor depends on the characteristic voltage of the LED. The ESP32 is powered at 3.3V and an output can deliver up to 10mA. With a 1.7V LED (typical for bright red), the resistor should be in the 220Ω-1kΩ range (1.6 to 7.3 mA) to keep a reasonable safety margin.

Make sure to use these pins on the microcontroller board:



The final circuit can look like this:



Software

Open the `00-Start-Here` project with Visual Studio Code (if you want to rename the project, make sure the new name doesn't contain any spaces.)

Explore the project

Take some time to explore the project. You can find the most relevant files in the "main" directory. Of particular interest:

- `main/BridgingHeader.h` makes definitions from ESP-IDF available to the Swift code.
- `main/CMakeLists.txt` contains build directives in order to compile the Swift files to generate RISC-V binaries, and defines the list of Swift files to be compiled (for convenience, all `*.swift` files contained in the `main` directory and its subdirectories will be compiled).
- `main/Output.swift` is a Swift wrapper over the ESP32 outputs. It calls some functions in the ESP-IDF framework, namely `gpio_reset_pin`, `gpio_set_direction`, and `gpio_set_level`. These functions (along with the `gpio_num_t` type and other constants) are documented here: https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-reference/peripherals/gpio.html#api-reference-normal-gpio.
- `main/Main.swift` implements the entry point for our application.

Implement the main loop

The `main.swift` fine contains only the initialization.
Add an infinite loop to make the LED flash repeatedly. With the constants `onTicks` and `offTicks` the LED will flash for 100 ms every second, you may change them as you wish.

Build and run

- Connect the board (if not done already)
  Open a terminal session in the project directory (either with Terminal.app or from Visual Studio Code):
- `esp-setup`
- `idf.py set-target esp32c6`
- `idf.py build`
- `idf.py flash`
  After the binary is uploaded to the chip, the LED should pulse.

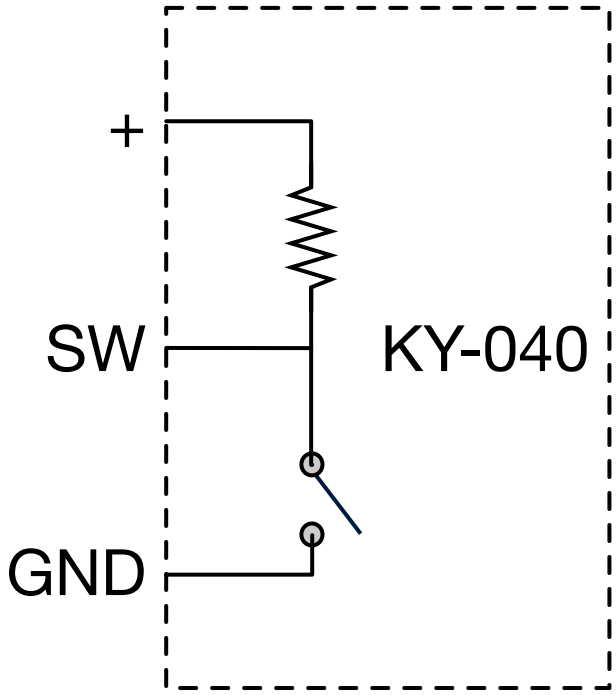
Add logging

- Add `print` statements inside the `while(true)` loop, to log the LED state.
- Build and flash (no need to use `esp-setup` or `idf.py set-target` again)
- Check the logs with `screen $ESPPORT 115200` or `idf.py monitor`.
- To exit from `screen`: type `ctrl-A`, then `k` and `y`
- To exit from `idf.py monitor`: type `ctrl-]`

Simple Input

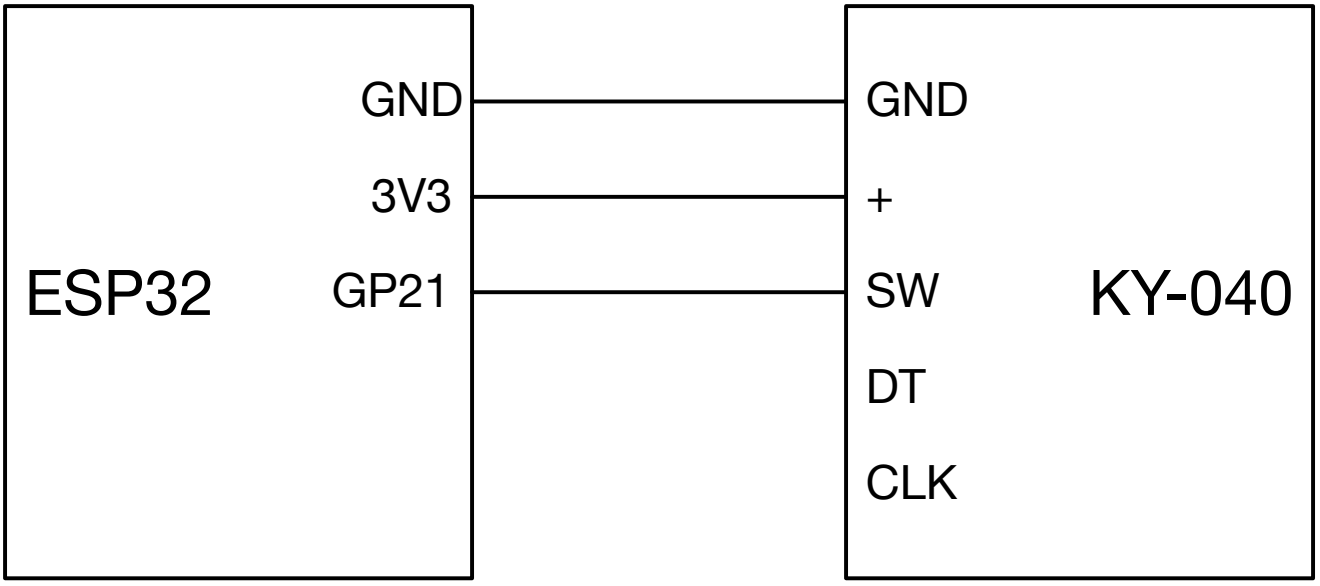
*If you didn't complete the previous step, start with the **01-Pulsing-LED** project.*

The KY-040 controller contains a switch with its pull-up resistor. Internally it is connected like this:



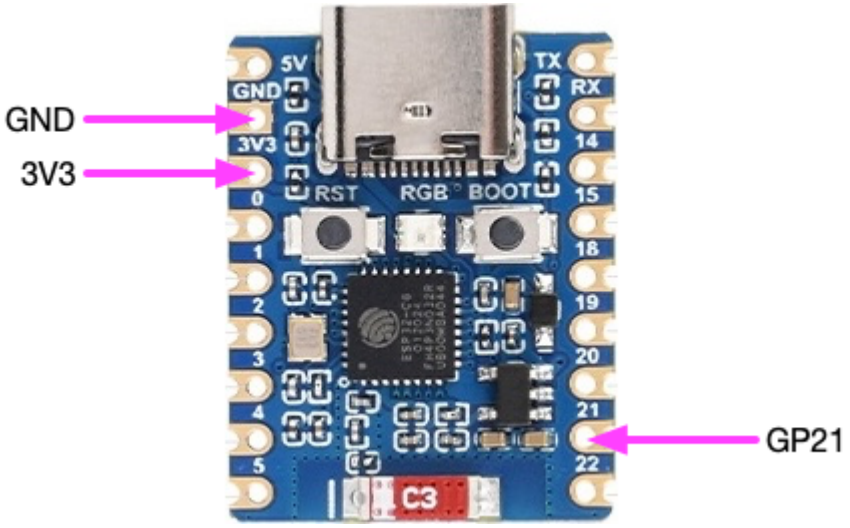
Wiring

Connect the KY-040 switch to GPIO21:

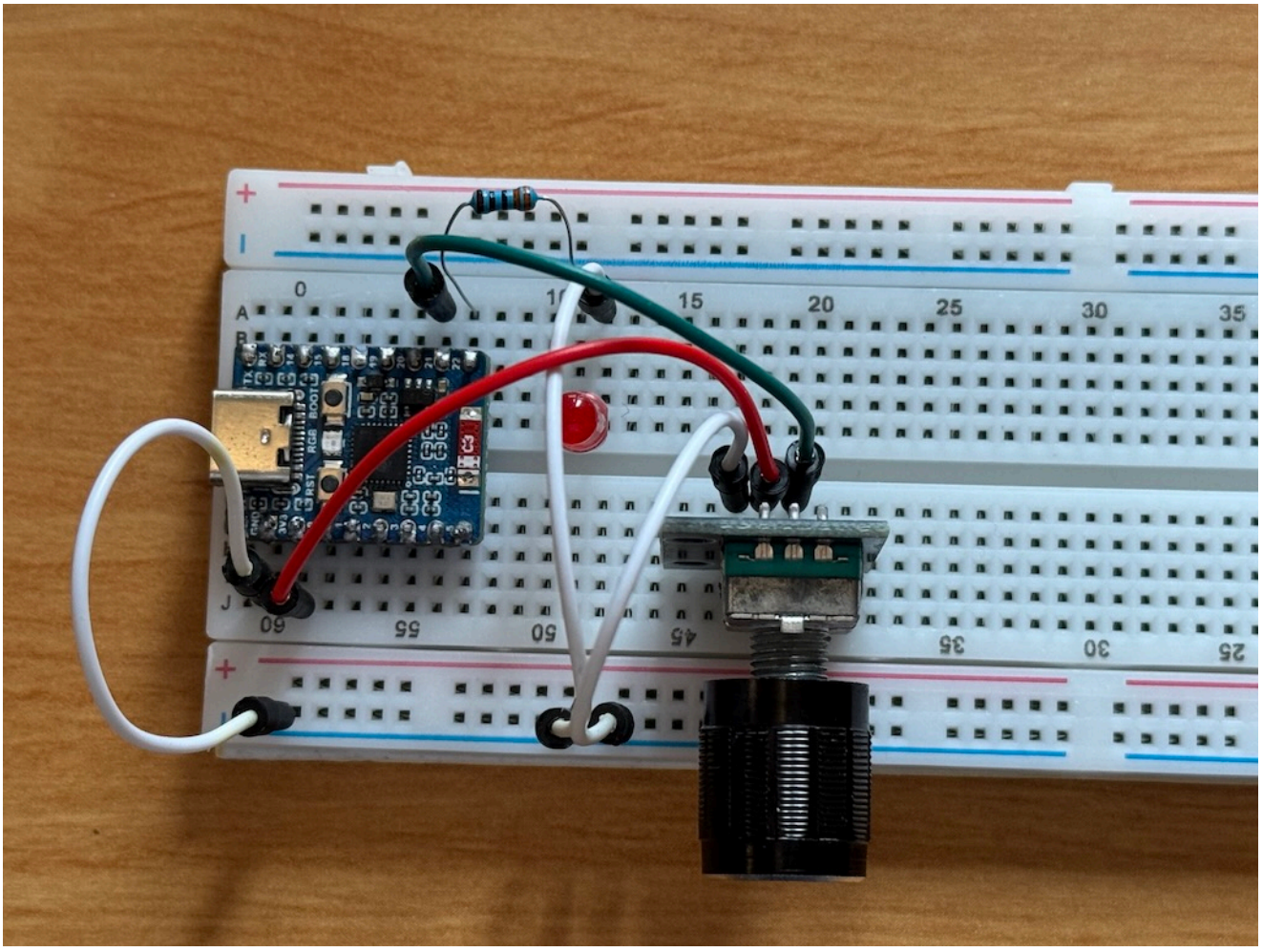


With this setup, the voltage on GPIO21 will be 0V (logical 0) when the switch is pressed, and +3.3V (logical 1) when it is released.

Make sure to use these pins on the microcontroller board:



The final circuit can look like this:



Software

Input.swift

- Create a new "Input.swift" file in the "main" directory of the project.
- In this file, declare a class named `Input` with a private constant `pin` of type `gpio_num_t`.
- The `init()` function of this class should take a pin number as its parameter.

- In this function, reset the pin and set its direction to `GPIO_MODE_INPUT`.
- Add a computed property of type `Bool` to return the state of the input.

Main.swift

- Create an instance of your input in the `main()` function. This input should be connected to pin 21.
- Change the main loop:
  - increment a counter each time the state of the input changes (and print this counter)
  - toggle the state of the LED each time the button is pressed (the input state changes from `true` to `false`).
  - Wait for 100 milliseconds (10 ticks).

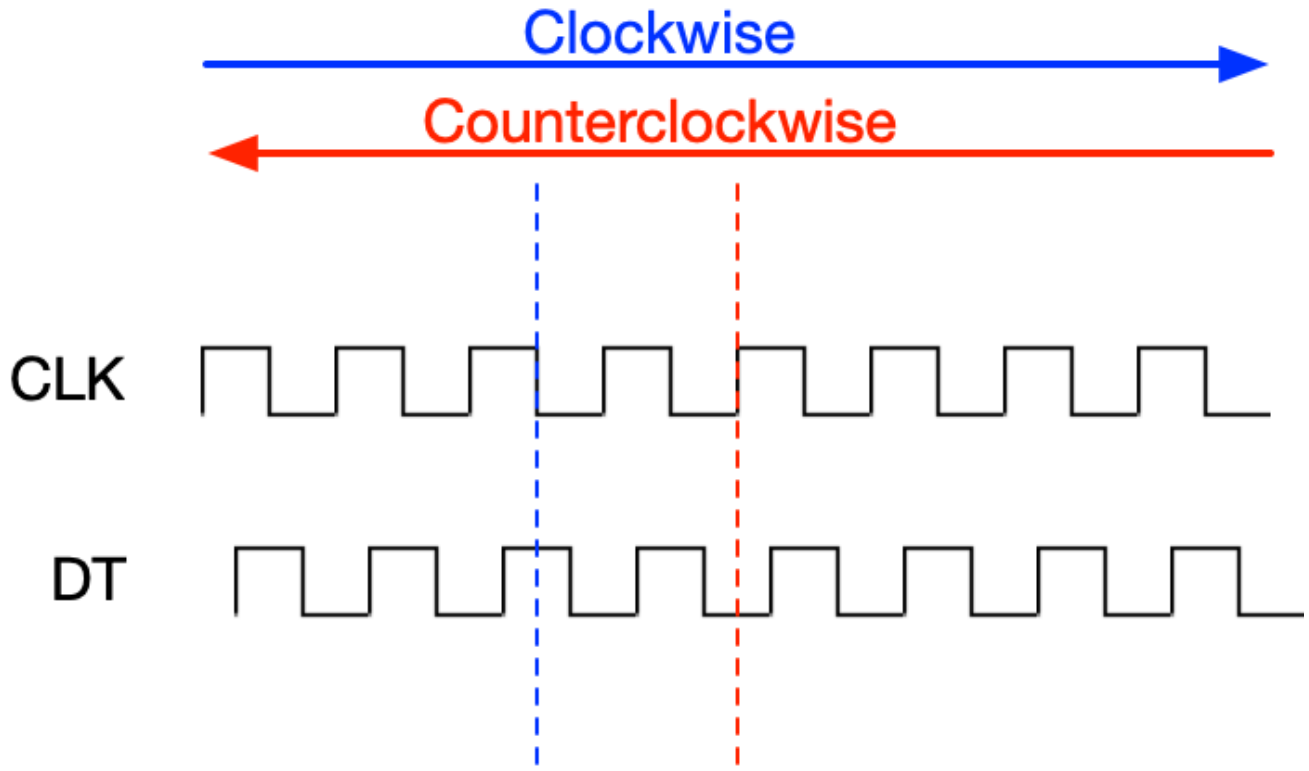
Testing

You may have to `touch CMakeLists.txt` after adding `Input.swift` to the project.
Build, flash, and watch the logs (it can be done with a single command: `idf.py build flash monitor`).
Verify that you get a line in the log each time you press or release the button, and that each button press toggles the LED.

Rotary Encoder

*If you didn't complete the previous step, start with the **02-Simple-Input** project.*

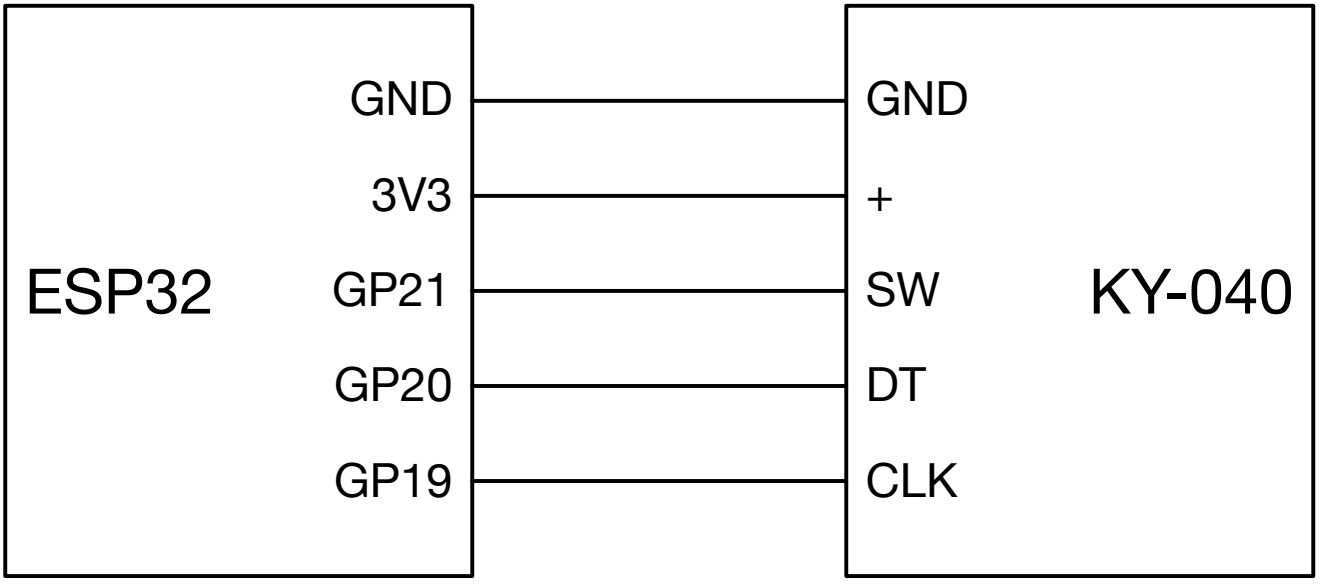
In addition to the switch we've been using so far, the KY-040 is a rotary encoder. It provides two signals, CLK and DT, to monitor the rotations of the knob.



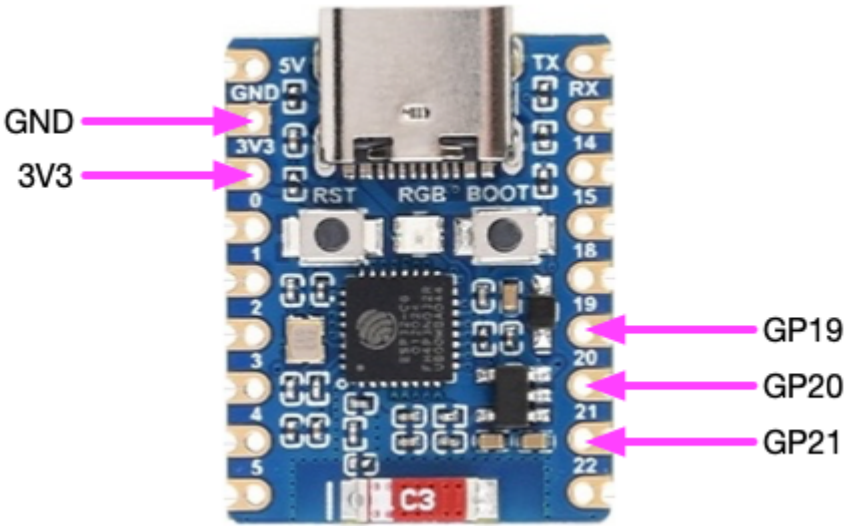
An easy way to decode the rotation events is to get the value of DT on each falling edge of CLK. If DT is high, the rotation is clockwise. If it is low, the rotation is counterclockwise.

Wiring

Connect the CLK and DT outputs of the encoder to GPIO19 and GPIO20 on the ESP32.



The KY-040 encoder provides pull-up resistors on DT and CLK signals.



Software

Additional source files

Add these files from the `Workshop Resources` folder to your project (in the main directory, next to the other Swift files in the project):

- `Input.swift` (this will replace your `Input.swift` file)
- `DebouncedInput.swift`
- `Timer.swift`
- `Errors.swift`
- `Queue.swift`

Input.swift

The `Input` struct adds an interrupt-driven callback, called each time the state of the input changes.

DebouncedInput.swift, Timer.swift

`Timer` is a Swift wrapper around the low-level interrupt-driven timer provided by ESP-IDF. `DebouncedInput` uses this `Timer` to add a short inactivity delay (10 ms) when the input state changes, in order to ignore any detected bounces.

Queue.swift

`Queue<T>` is a generic wrapper around the queues provided by ESP-IDF (based on FreeRTOS queues). It can be used to create a communication channel between the interrupt callbacks and the main application.
It provides three main methods:

- send(): send an element from the main execution context
- sendFromISR(): send an element from the context of an interrupt handler
- receive(): return the first element in the queue, waiting if necessary

Errors.swift

`Errors.swift` provides types that can be thrown in response to return codes from the ESP framework (`ESPError`) or FreeRTOS (`OSError`).

RotaryController

Create a `RotaryController` class. It should emit rotation events through a callback `(Direction) -> Void` , where `Direction` is an `enum` with two cases, `clockwise` and `counterclockwise` .
The `init()` method should take three parameters: the pin numbers for the inputs connected to CLK and DT, and the callback. It should create a `DebouncedInput` for CLK and an `Input` for DT, and invoke the callback when the `DebouncedInput` triggers its own callback.
No other method is necessary.

Main

- Instantiate a `Queue` with a capacity of 10 elements of type `RotaryController.Direction` .
- Instantiate a `RotaryController` with the input pins defined earlier. In its callback, send the direction to the queue (remember, the callback is called from an interrupt handler)
- In the main loop, wait for elements in the queue. Increment the counter for a clockwise rotation and decrement it for a counterclockwise rotation, then log its value. The delay is not required anymore.

BridgingHeader

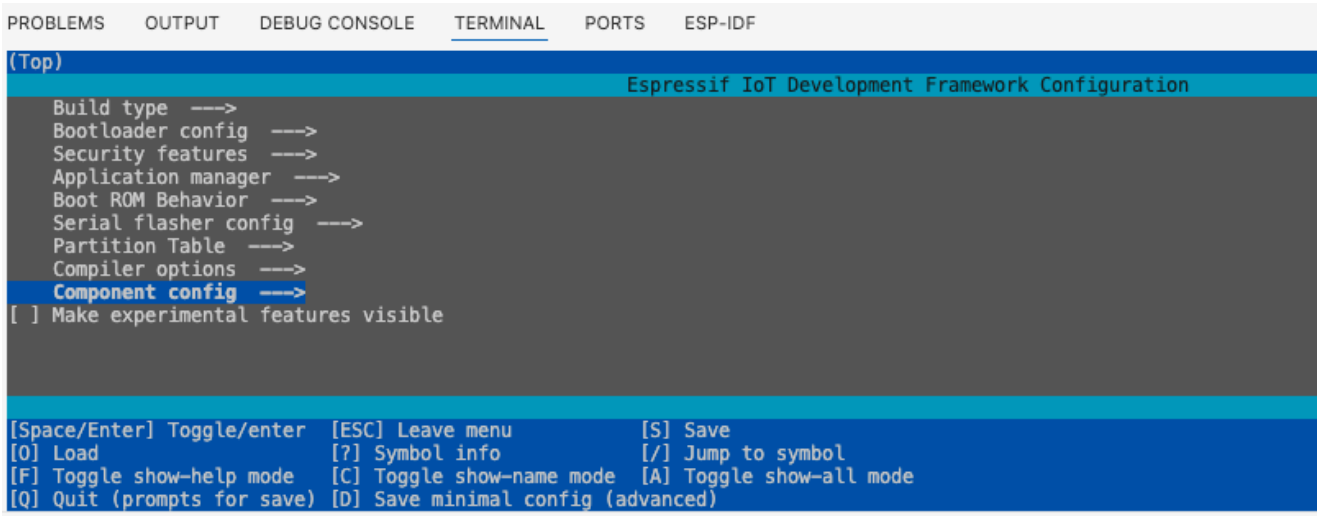
Add the following includes:

- `freertos/queue.h`
- `esp_timer.h`

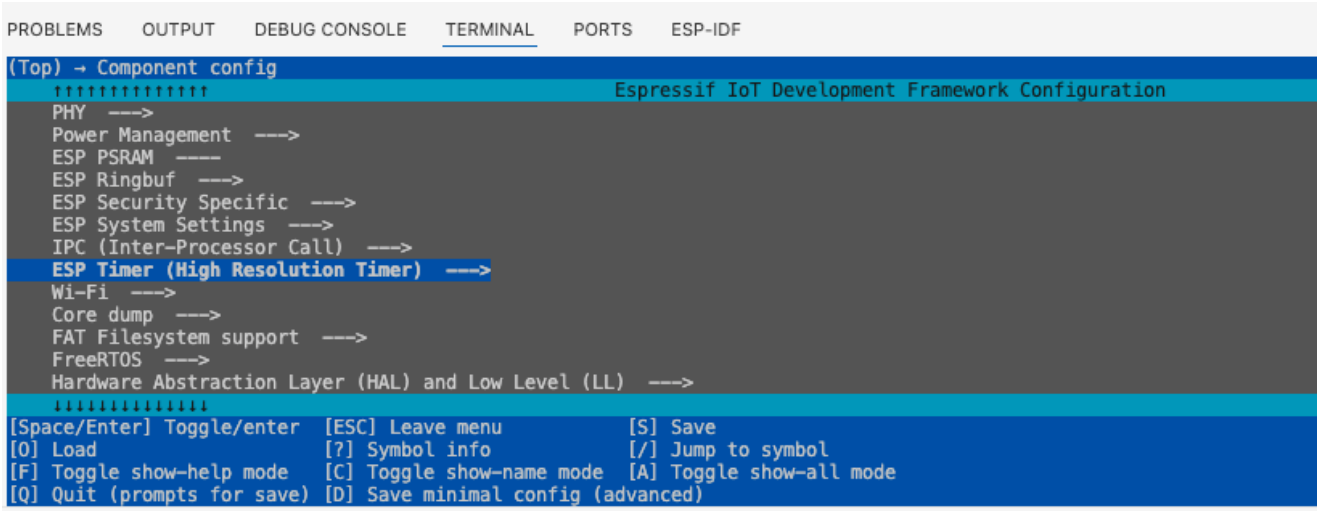
Build and test

The `Timer` class relies upon interrupts generated by the hardware timer, and this feature is not enabled by default. Let's enable it with the IDF configuration menu.

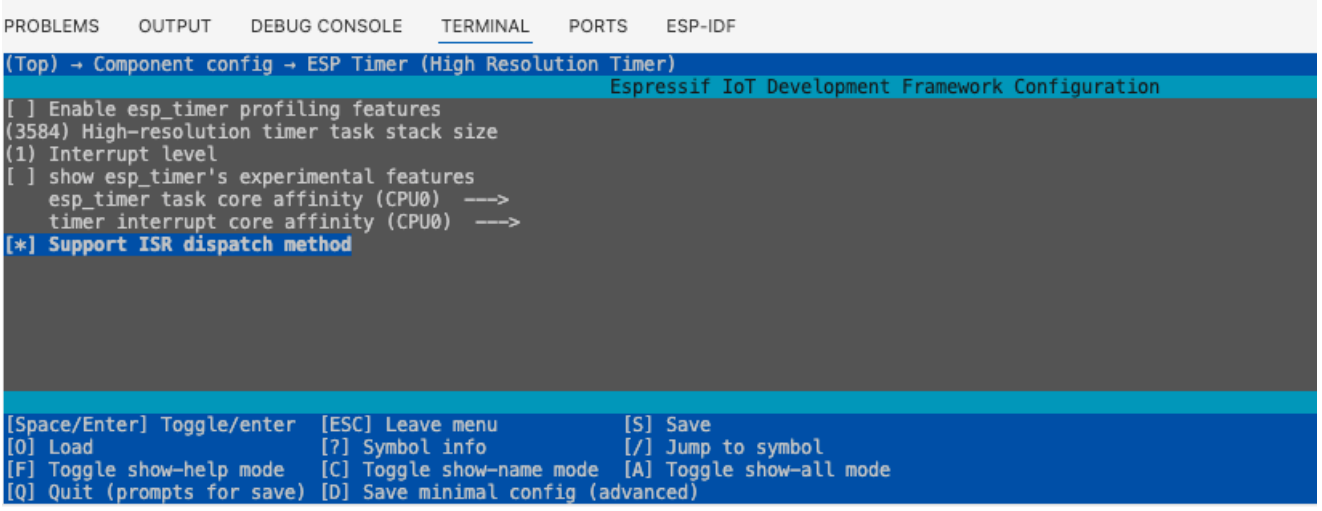
In the terminal, type `idf.py menuconfig`. In the menu, choose `Component config`.



In the submenu, choose `ESP Timer (High Resolution Timer)`.



Select `Support ISR dispatch method`, hit `space`, then type `q` and save the configuration.



Then run `idf.py build flash monitor`.

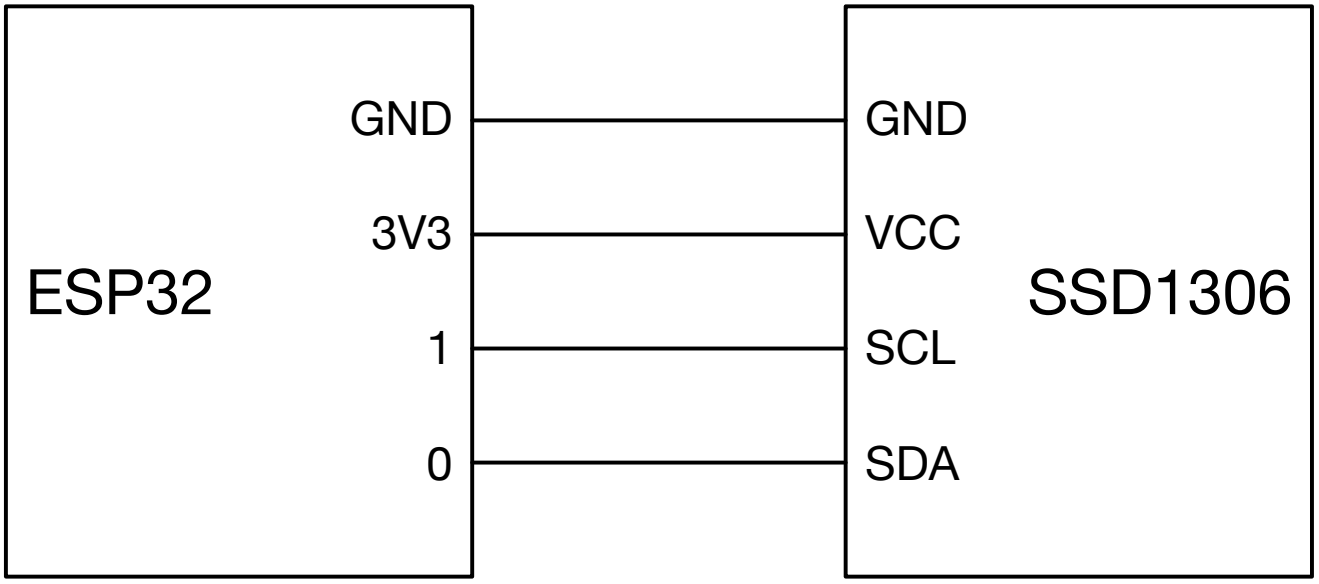
Display

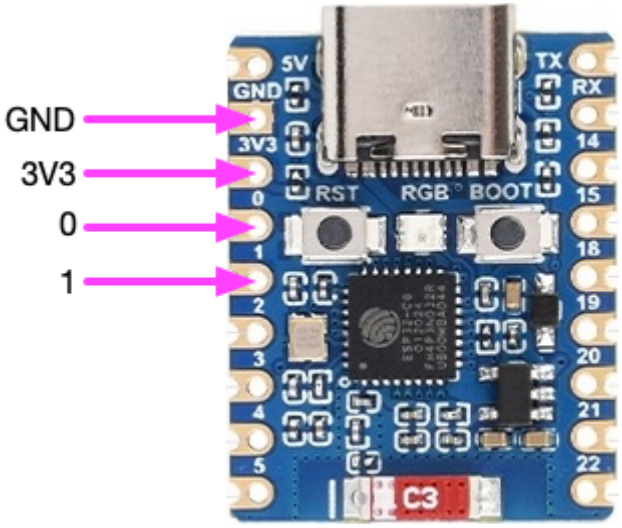
*If you didn't complete the previous step, start with the **03-Rotary-Encoder** project, and run `idf.py menuconfig` to enable ESP Timer ISR dispatch if necessary.*

Let's use a third-party library to connect a I^{2}C display.

Wiring

The SSD1306 display implements the I^{2}C protocol. In addition to the GND and 3.3V power, its connector has two pins labeled SCL and SDA.
The ESP32C6 can assign any pair of pins to the I^{2}C signals. Let's pick pin 0 for SDA and pin 1 for SCL.





Software

U8g2 (https://github.com/olikraus/u8g2) is an open-source graphics library for monochrome displays. In order to use it with the ESP32 microcontroller, we'll need an additional driver which can be found here: https://github.com/mkfrey/u8g2-hal-esp-idf.

Additionally, `Display.swift` (in the resources folder) provides a Swift wrapper for this library.

Extract `u8g2-master.zip` and `u8g2-hal-esp-idf-master.zip` from the thumb drive. You should end up with `u8g2-master` and `u8g2-hal-esp-idf-master` directories. Rename them `u8g2` and `u8g2-hal-esp-idf`.
Create a `components` folder at the root of the project directory (next to the `main` folder). Move `u8g2` and `u8g2-hal-esp-idf` into this folder. The build system will automatically compile CMake subprojects found in the `components` folder.

`u8g2-hal-esp-idf` does not support the C6 variant of the ESP32 microcontroller. As a temporary workaround, we can change the test line 20 of `u8g2_esp32_hal.h` to:

```
#if SOC_I2C_NUM > 1 && defined( I2C_NUM_1 )
```

U8g2 defines its fonts as `const uint8_t[]` and the `u8g2_SetFont()` function expects a `const uint8_t *` argument. While these types are equivalent from the point of view of the C compiler, the first construct is not interoperable with Swift. In order to make fonts available to Swift, add the `u8g2-fonts` directory (from the `Workshop Resources` folder) to the `components` directory in the project.

Update `BridgingHeader.h` to add these lines:

```
#include "u8g2.h"
#include "u8g2_esp32_hal.h"
#include "u8g2_font_ptr.h"
```

Now you can add the `Geometry.swift` and `Display.swift` wrappers to the project.

In the `main()` function, before the main loop, create a `Display` instance with the `sdaPin` and `sclPin` parameters matching your circuit. The `i2cAddress` and `orientation` parameters should be left to their default values.

Then, to test everything is set up properly, add this line:

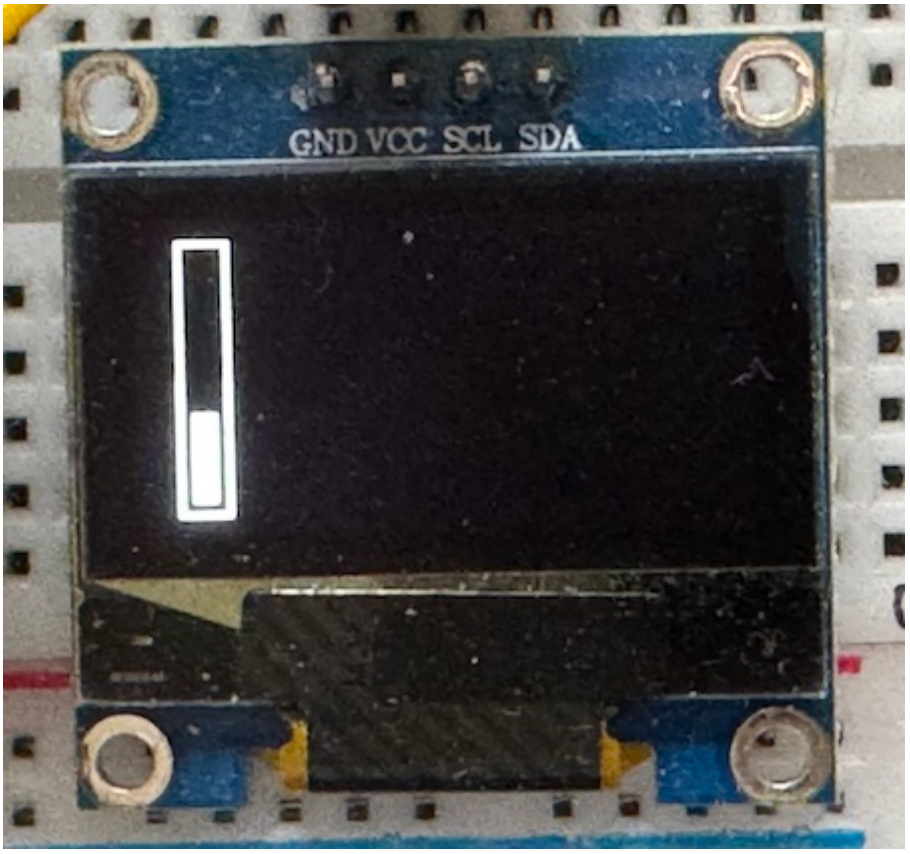
```
display.drawStr("Hello Swift Island!", at: Point(x: 0, y: 20), refresh:
true)
```

Build, flash, and run.

Gauge

- The `Display` class is missing `frameRect` and `fillRect` methods, implement them.
- In `Main.swift`, constrain the value of `counter` between 0 and 20.
- Remove the call to `drawStr()`.

- Before the main loop, display a rectangle frame of size (11, 26) at position (9, 4).
- In the main loop, display the value of the counter as a filled rectangle of width 5, using the value of the `counter` for the height.



Images

*If you didn't complete the previous step, start with the **04-Display** project, add `u8g2` to `components`, and run `idf.py menuconfig` to enable ESP Timer ISR dispatch if necessary.*

With our setup it is possible to display arbitrary images. However, the storage space on the ESP32 microcontroller is very constrained (typically 4 or 8 MB), so we need to compress the images heavily.
Since we are dealing with black&while (1 bpp) images, a good candidate is the CCITT Group 4 compression algorithm.

Software

Clone the repository at https://github.com/bitbank2/TIFF_G4, or extract the `TIFF_G4-master.zip` archive from the thumb drive. The resulting directory should be moved into the `components` directory of your project.

Additionally, copy the `resources` directory from the `Workshop Resources` folder into the `components` directory. `resources/files` contains the images that will be included in our final binary.

Add includes for `resources.h` and `TIFF_G4.h` to `BridgingHeader.h`.

Each TIFF file in `resources/files` will end up in the built binary as a sequence of bytes, and the build system generates two symbols (for start end end addresses) that can be used from C with the `asm()` directive.

For each image, `resources/include/resources.h` exposes a pointer and a size, and `resources/src/resources.c` computes them. Currently `swiftLogoPtr()` and `swiftLogoSize()` are implemented, for `Swift_logo_white.tiff`. Another image is provided: `Swift_Island_logo.tiff`. Update `resources.h` and `resources.c` to add the corresponding functions.

Add `TiffImage.swift` to your project. The `draw` method is missing the part that decodes the image, implement it.

In `Main.swift`, instantiate a `TiffImage` with the Swift Island logo, and display it at (50, 0).

Inverting the image

It can be useful to display an inverted image (1=black, 0=white). Add an `inverted` Bool argument to `draw()`, and propagate this flag to `drawCallback()` to invert all the bits if `inverted` is `true`.

Optimizing refresh

- Remove the `display.refreshAll()` line from `TiffImage.draw()`
- Add a `drawImage()` method to `Display` with two additional arguments: the `image` (a `TiffImage`) and a `refresh` flag.
- In this method, accumulate the rectangle to the dirty rect, call `image.draw()`, and trigger `refresh()` if requested.