Embedded Swift Workshop

ARCtic Conference 2025

Frank Lefebvre

Preparation

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 address 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 these components can be installed from scratch, the easiest way to install the framework and the build tools at once is to use Visual Studio Code with the ESP-IDF extension.

All components can be installed on a standard user account, without requiring admin privileges.

Installation

Swift Toolchain

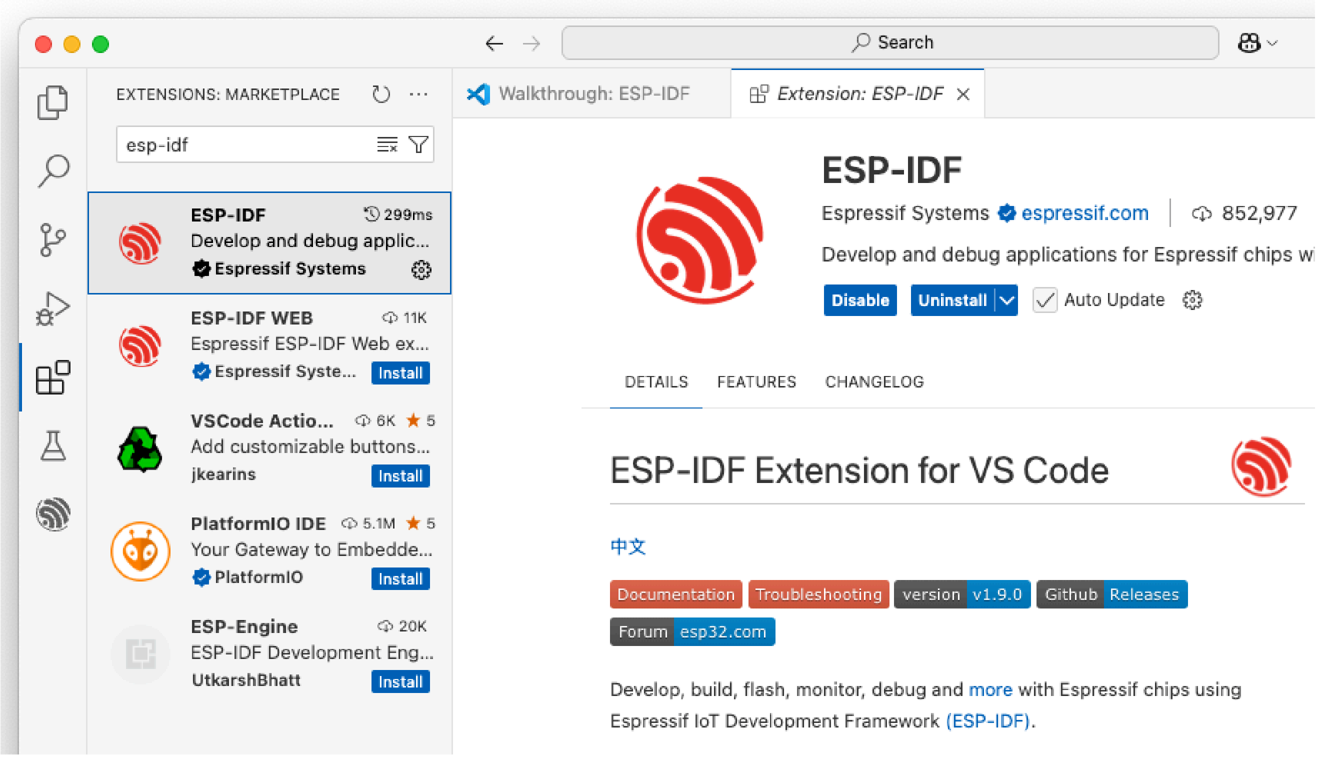
On the [Install Swift] webpage, in the "Development Snapshots" section, download the "main" toolchain. The `swift-DEVELOPMENT-SNAPSHOT-2025-XX-XX-x-osx.pkg` package can be installed either for all users (the default option) or for the current user only. Here we'll assume the toolchain is installed for the current user (in `~/Library/Developer/Toolchains`). If the "all users" option is selected, you'll need to change the `export TOOLCHAINS` line in the `esp-setup` script (below).

Visual Studio Code

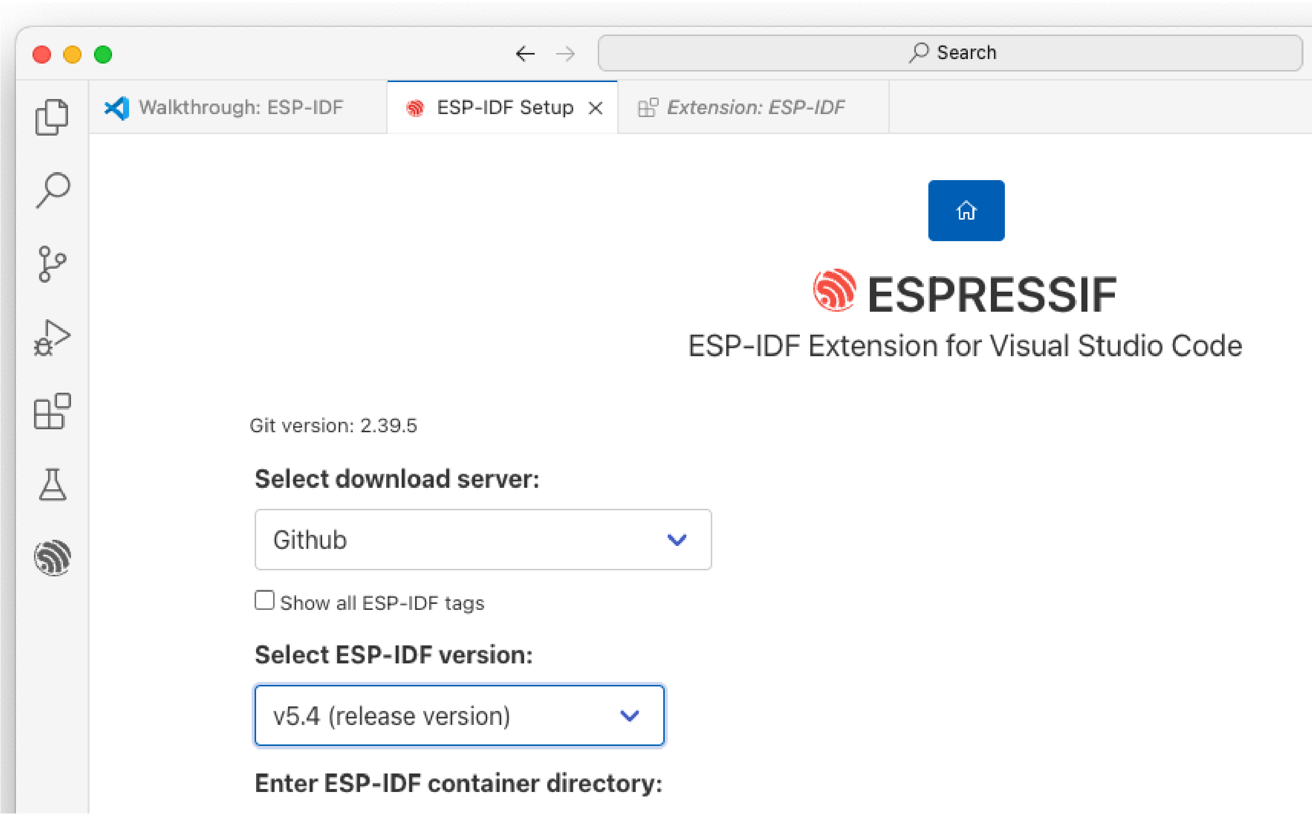
Download [Visual Studio Code] and move the application to `/Applications` or `~/Applications`.

ESP-IDF Extension

Open Visual Studio Code, and search for **ESP-IDF** in the Extensions pane.

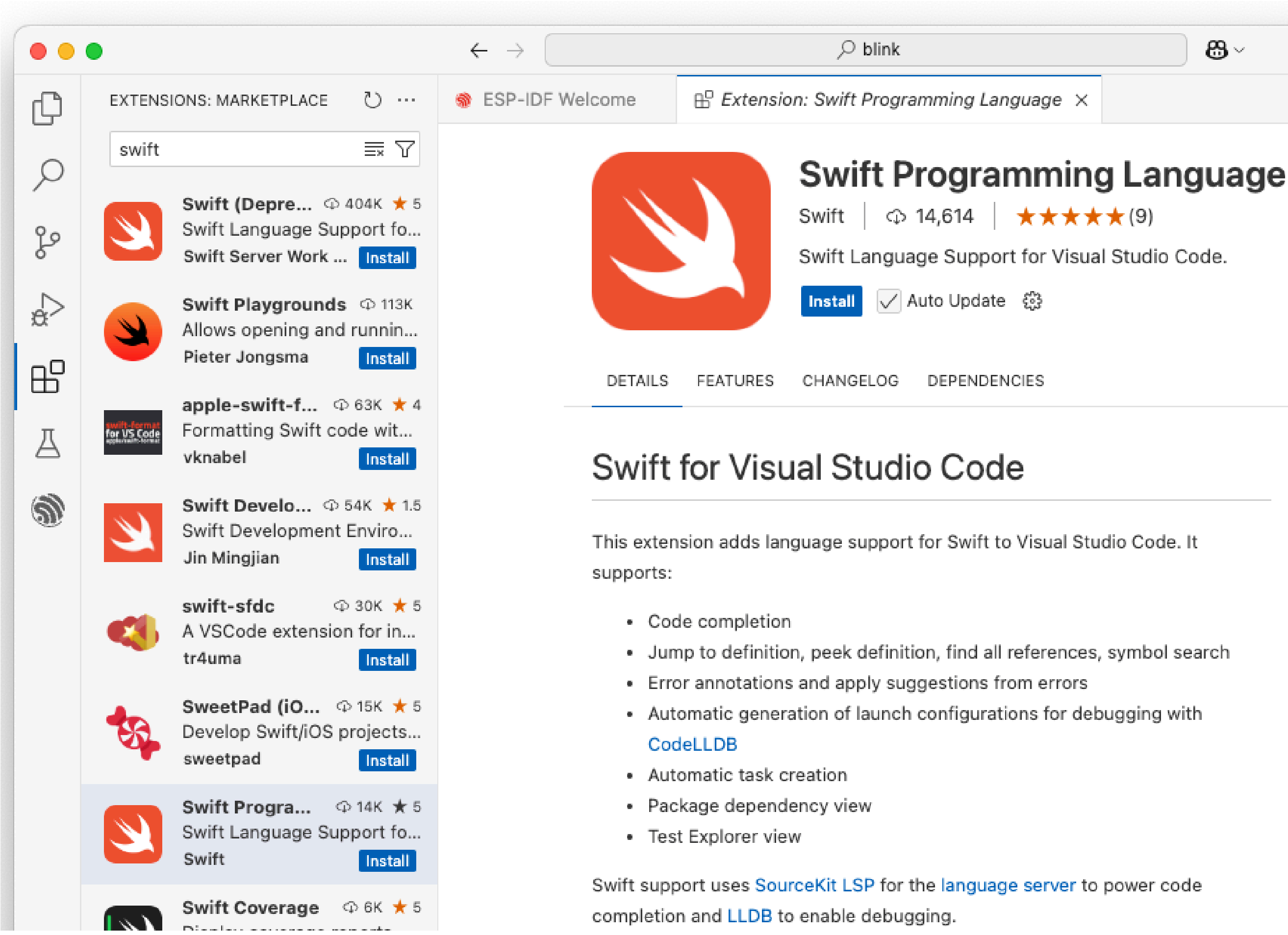


When the extension is installed, click "Configure ESP-IDF Extension". Pick "Express Setup", and select the latest release version of the framework. All other settings can be left unchanged.



Swift Extension

To enable autocomplete and syntax coloring features, you should install the **Swift Programming Language** extension for VS Code.



Testing

Without connecting the ESP32, list the available serial ports:

```
ls -la /dev/tty.*
```

Then connect the ESP32 to a USB port, and list the ports again. You should have a new port available, and its path should be in the form: `/dev/tty.usbmodemXXXX` (the actual name depends on the USB port the board is connected to, or the position in the USB device tree if a USB hub is used).

With this knowledge, update the `ESPPORT` environment variable in the `esp-setup` script provided in the workshop resources folder, and move (or copy) the script to the `~/esp` directory. Then add this line to your shell profile (usually `~/.zshrc`):

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

Now you are ready to test your setup.

In the ESP-IDF examples, choose "blink" in the "get-started" section, and create a new project from this example. Make sure the path to this project doesn't contain any spaces (some ESP-IDF scripts might fail otherwise).

Open a terminal window and `cd` into this project (or open the project with Visual Studio Code and use the terminal pane therein), and type:

```
esp-setup
```

This makes the IDF tools available in the PATH, and defines some additional environment variables. Significant ones are:

- `ESPPORT` is used by the `idf.py flash` command (below) to find the device;
- `TOOLCHAINS` corresponds to the bundle identifier of the Swift toolchain to be used for compiling. The `esp-setup` script uses the toolchain pointed to by `latest`.

```
swift -version
```

Verify that the returned version number has a `-dev` suffix. If not so, either the development snapshot of the Swift toolchain is not installed properly, or the `TOOLCHAINS` variable is not defined correctly.

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

This command creates a `build` directory inside the project, and generates the configuration files necessary for the build system to generate a RISC-V executable suited to the ESP32C6 chip.

```
idf.py build
```

The build should succeed with a `Project build complete.` message, followed with flash instructions.

```
idf.py flash
```

This sends the built application to the chip over the USB connection. After the firmware is flashed, the LED should blink.

The example code contains log statements. By default the logs are transmitted over the serial connection on USB. To view the logs, type:

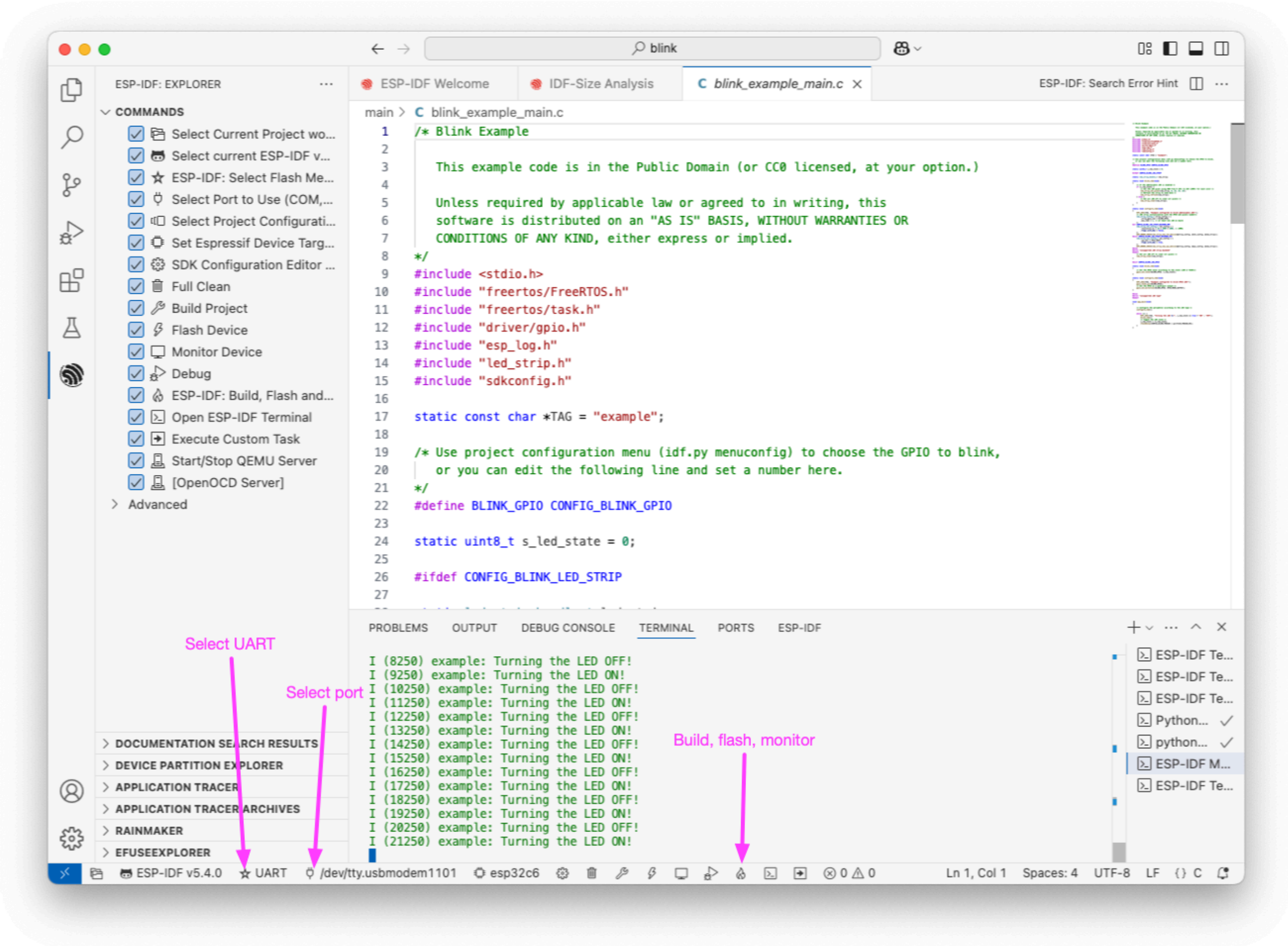
```
screen $ESPPORT 115200
```

To exit screen, type `ctrl-A`, then `k` and `y`.

The text sent in the logs may contain control sequences that can interfere with the terminal session (CR-LF, etc). If such is the case, you can restore the default behavior with:

```
stty sane
```

Note: it is also possible to perform the `build`, `flash` and `monitor` operations in a single step from the GUI. For this to work, the flash method (UART) and the port (/dev/tty.usbserial*) need to be configured in the IDE beforehand.

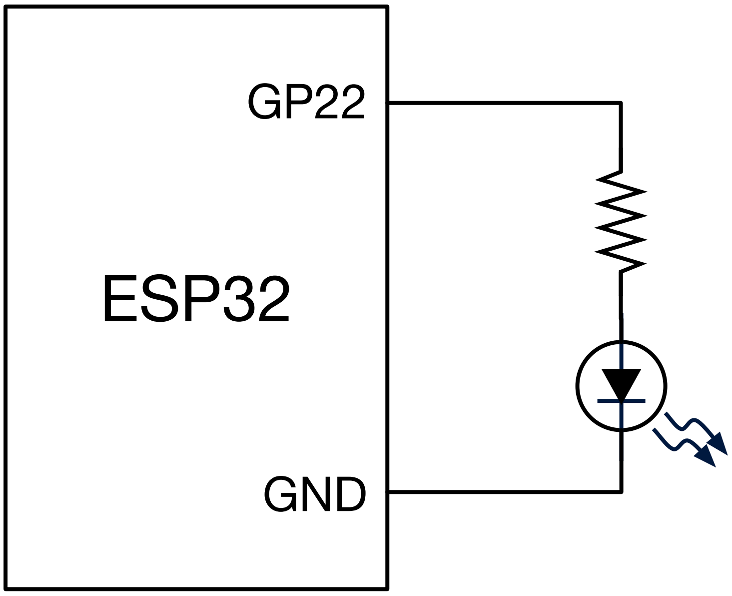


Blinking LED

Every embedded journey starts with a blinking LED.

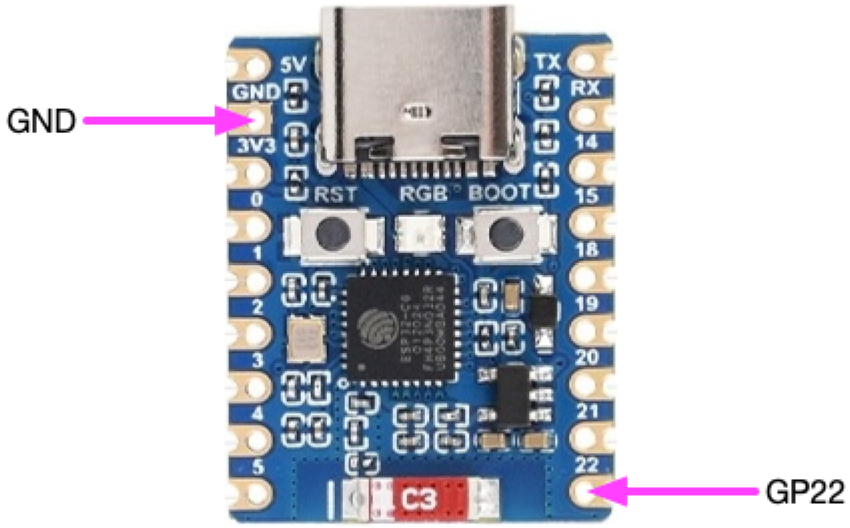
Wiring

Connect a LED to GPIO22, as follows:

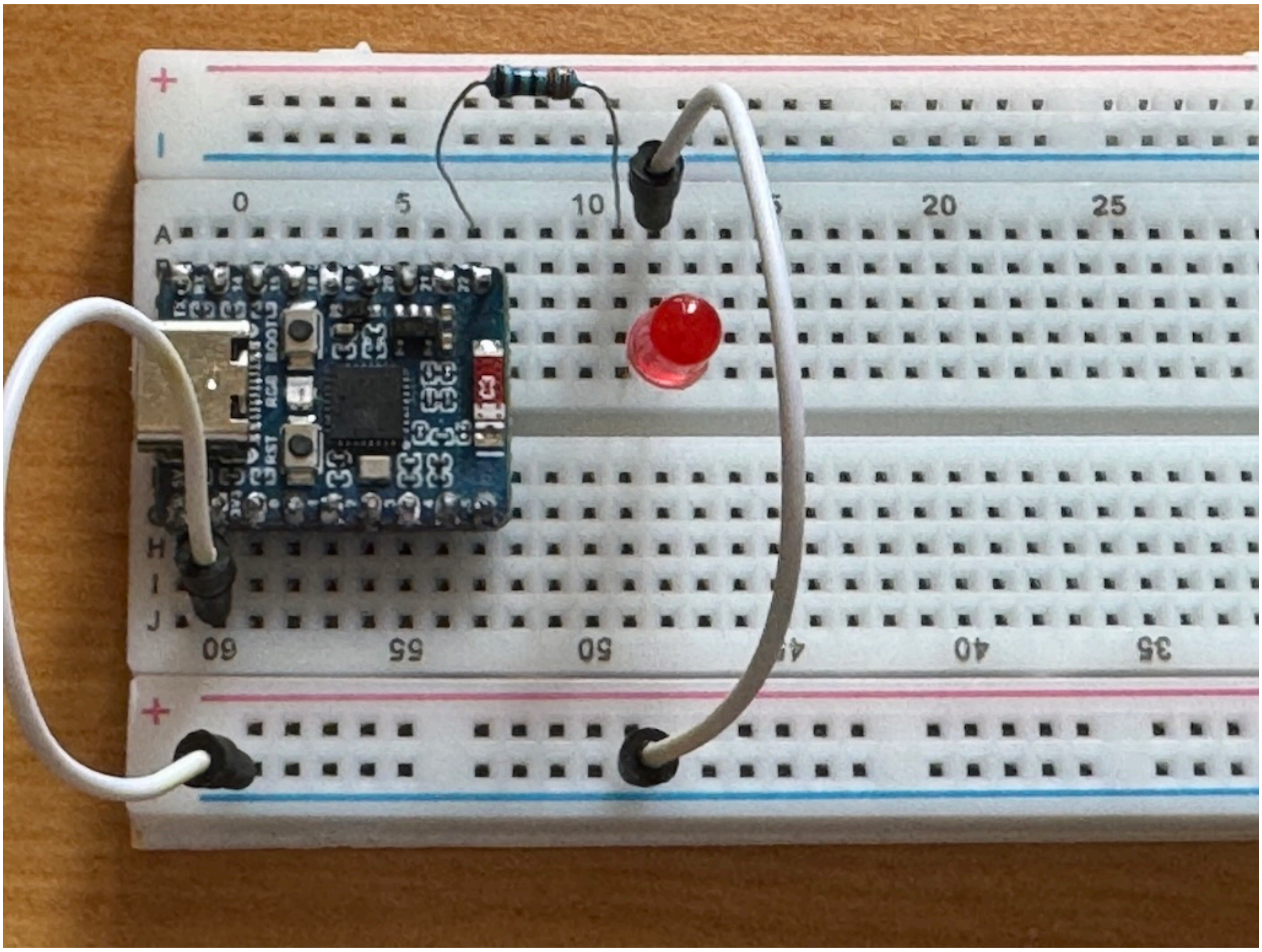


The value of the resistor depends on the characteristic voltage of the LED. The ESP32 is powered at 3.3V and an output can deliver up to 10mA. With a 1.7V LED (typical for bright red), the resistor should be in the 220Ω-1kΩ range (1.6 to 7.3 mA) to keep a reasonable safety margin.

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.)
This starting point is a copy of the `esp32-led-blink-sdk` example available here: https://github.com/apple/swift-embedded-examples.

Explore the project

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

- `BridgingHeader.h` makes definitions from ESP-IDF available to the Swift code.
- `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.
- `Led.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.swift` implements the entry point for our application.

Change the output pin

The example assumes the LED is connected to GPIO 8. However, with our hardware, GPIO 8 can't be reached easily (it is exposed under the board, and there is no pin connected to it). Modify `Main.swift` to use pin 22 instead of 8.

Build and run

In the project directory:

- `esp-setup`
- `idf.py set-target esp32c6`
- `idf.py build`
- Connect the board (if not done already)
- `idf.py flash`
  After the binary is uploaded to the chip, the LED should blink.

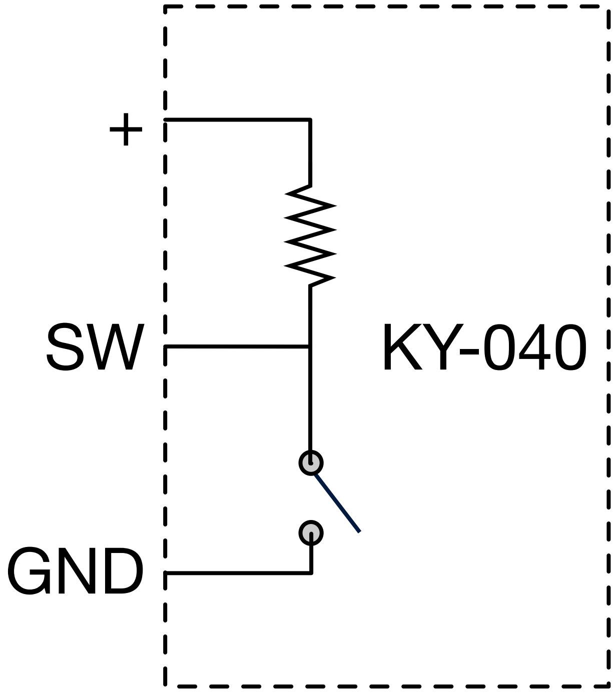
Add logging

- Add a `print` statement 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` or `idf.py monitor`.

Simple Input

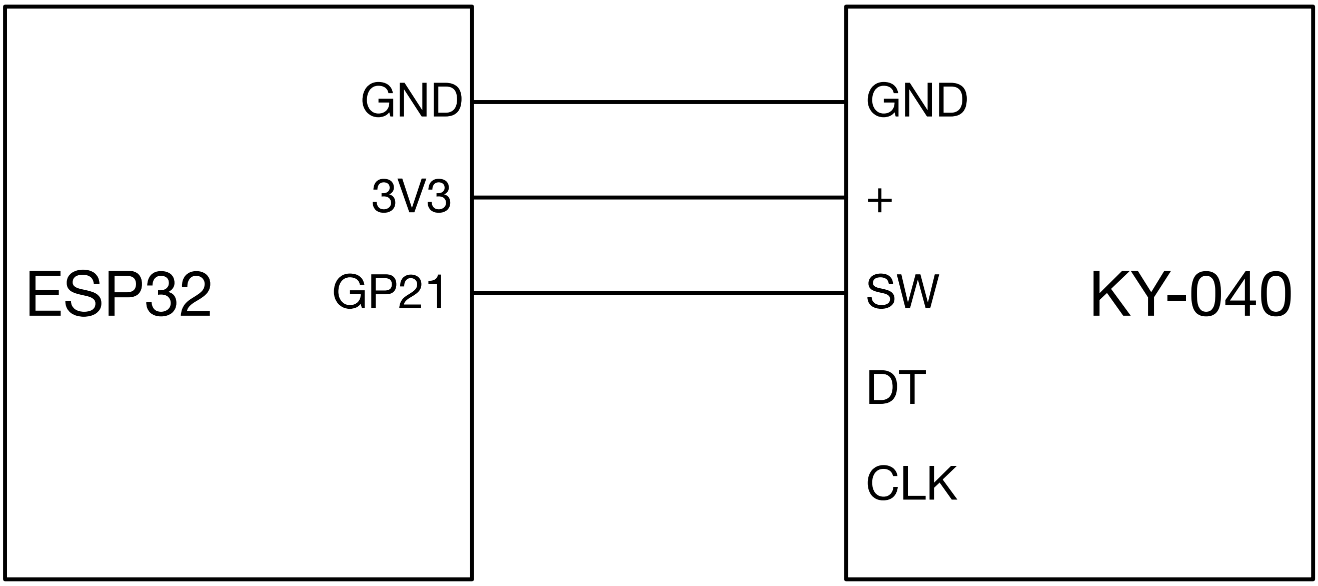
*If you didn't complete the previous step, start with the **01-Blinking-LED** project.*

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



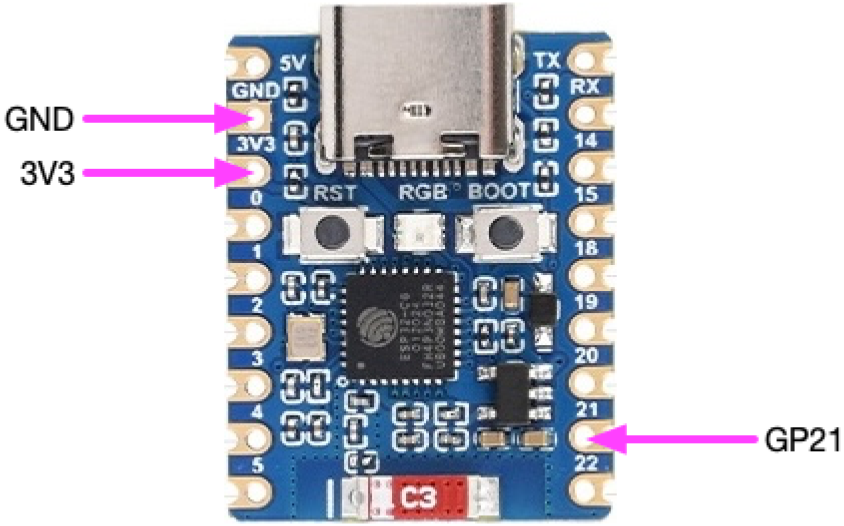
Wiring

Connect the KY-040 switch to GPIO21:

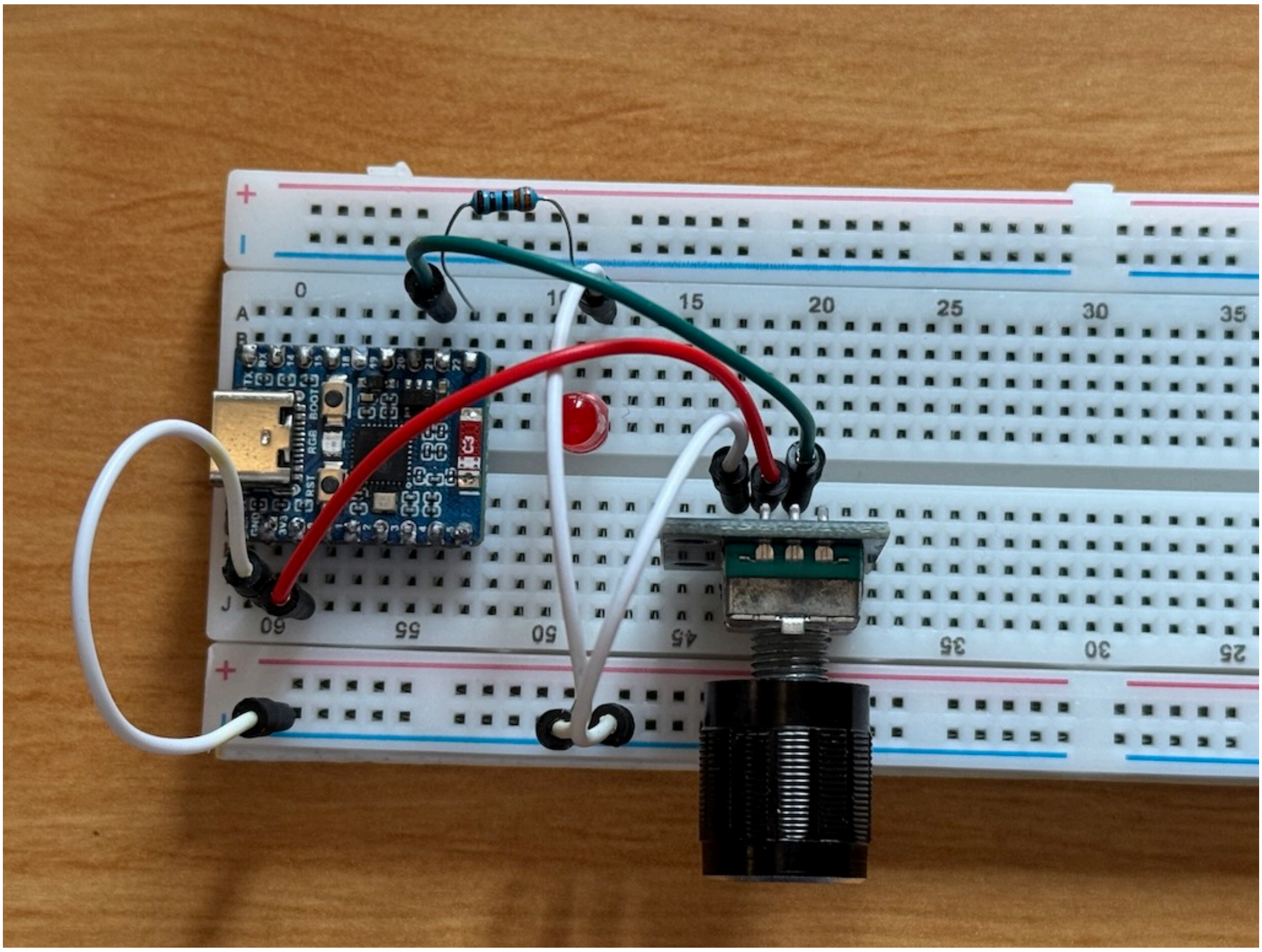


With this setup, the voltage on GPIO21 will be 0V (logical 0) when the switch is pressed, and +3.3V (logical 1) when it is released.

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 var pin of type `gpio_num_t`.
- The `init()` function of this class should take a pin number as a 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.
- In the 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`).
- Shorten the delay (`vTaskDelay`) to 100 milliseconds. Note: the default tick rate is 100 Hz, and we don't intend to change it.

CMakeLists.txt

The new `Input.swift` file must be compiled with `Led.swift` and `Main.swift` when the project is built.
The list of files to be compiled is defined in the `target_sources` command. You can either keep this list explicit and add `Input.swift`, or have `CMake` look for all `.swift` files in the current directory.
For the latter option, you can create a `SWIFT_SOURCES` variable with this command (to be inserted before the `target_sources` line):

```
file(GLOB_RECURSE SWIFT_SOURCES *.swift)
```

Then the explicit list of Swift files can be replaced with `${SWIFT_SOURCES}`.

Testing

Build and run. 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.

Interrupt-driven Callback

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

Polling is not the most efficient way to receive events. Let's update our `Input` class with a callback function that will be invoked each time the state of the input changes.
In order to achieve this, we can rely on an interrupt handler attached to the input.

An interrupt handler is a C function with this signature:

```
void gpio_isr(void *args);
```

`args` is an opaque pointer sent at installation time, with:

```
typedef void (*gpio_isr_t)(void *arg);
esp_err_t gpio_isr_handler_add(gpio_num_t gpio_num, gpio_isr_t isr_handler
, void *args);
```

It is used by the interrupt handler to obtain context. In our case, the context should provide at least the pin number and the callback.

InterruptHandlerContext

Create an `InterruptHandlerContext` struct to hold the pin number and the callback.

InterruptHandler

The interrupt handler must be declared as a C function accepting a single argument of type `void *`.
In Swift, `void *` is an optional `UnsafeMutableRawPointer`. The declaration can be:

```
fileprivate func interruptHandler(_ arg: UnsafeMutableRawPointer?)
// fileprivate if this function is declared in the Input.swift file
```

In this function, we must rebind `arg` to the `InterruptHandlerContext` type. Use `assumingMemoryBound(to:)` to get a typed pointer, then get its `pointee`.
Then use the context to perform the call.

Input class

Add two private properties to the Input class:

- the callback,
- a typed `UnsafeMutablePointer` for the context.

init

Add the `callback` parameter to `init`, and set the callback property accordingly.
After the pin direction is set (`gpio_set_direction`), if the callback is not nil, add the following operations:

- allocate a context pointer, set its properties, and set the context property.
- call `gpio_install_isr_service(ESP_INTR_FLAG_LEVEL1)` (this function should be called once, regardless of the number of instances of `Input`)
- `gpio_set_intr_type(pin, GPIO_INTR_ANYEDGE)`
- `gpio_isr_handler_add` (passing the interrupt handler and the context pointer)
- `gpio_intr_enable(pin)`

Main

In the `main()` function, add a closure parameter to `Input(gpioPin: 21)` , and increment the counter in this closure (you may need to move the property declaration to the top of the `main()` function).
Remove the counter increment from the loop.

Build and run

Notice that the counter is incremented way too often. This is expected, and will be fixed in next step.

Debouncing

*If you didn't complete the previous step, start with the **03-Interrupt-Input** project.*

Hardware is not perfect, let's mitigate with software.

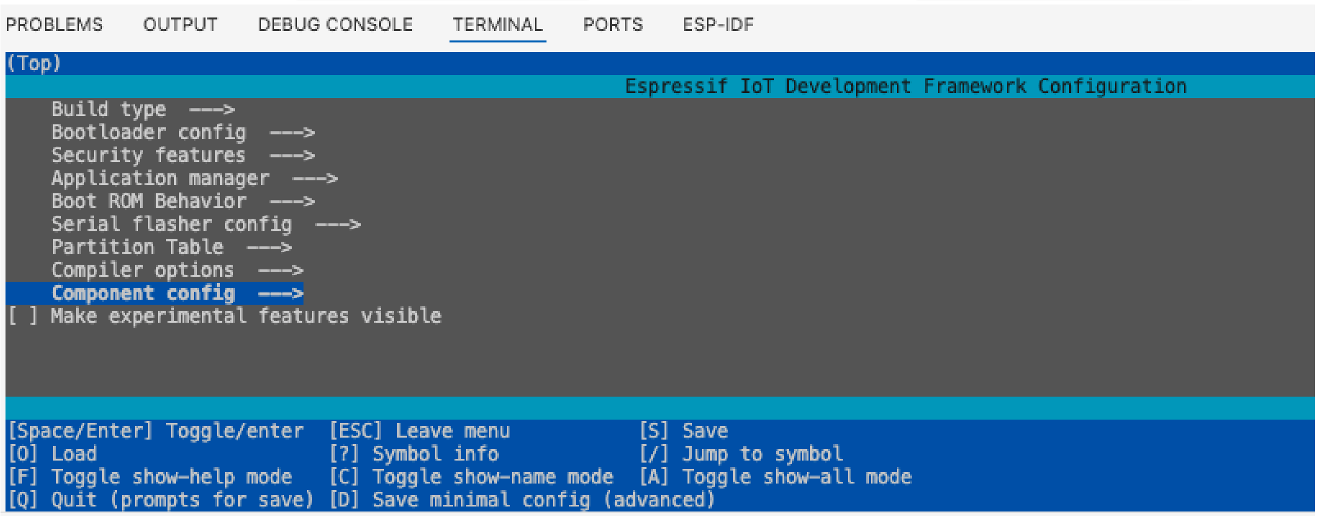
The goal is to ignore input state changes for a short duration after a first change occurred.

Timer

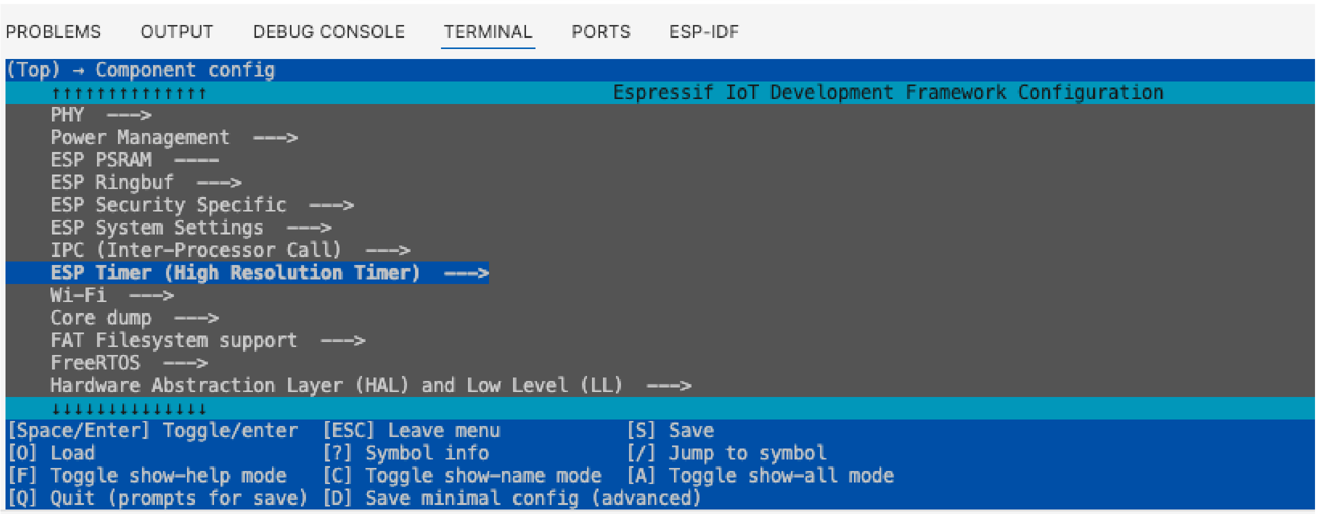
Add the `Timer.swift` and `Errors.swift` files (in the `resources` folder) next to the other `.swift` files in the project.

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.

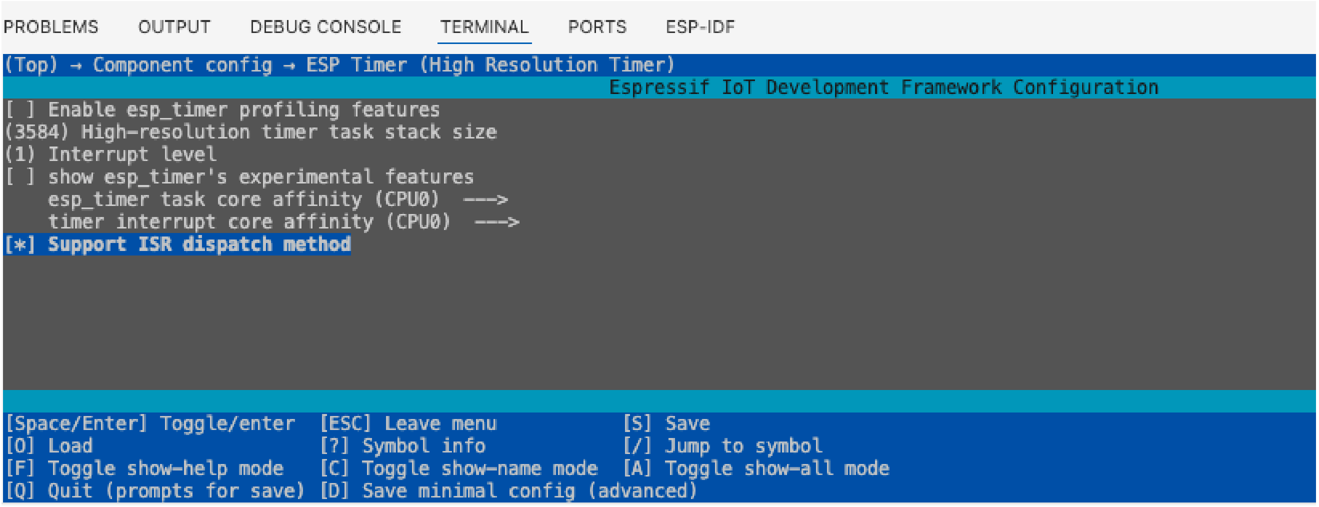
In the terminal, type `idf.py menuconfig`. In the menu, choose `Component config`.



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



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



Add `#include "esp_timer.h"` to `BridgingHeader.h`.

The project should compile without errors.

DebouncedInput

In a new file, create a class named `DebouncedInput` with the following private properties:

- a callback (similar to the `Input` callback)
- an instance of `Input`
- an instance of `Timer`
  and a boolean `state` that should be declared `private(set)`.
  The `init()` function should take two parameters: a pin number, and a callback.
  Initialize the input and timer properties so that:
- when the input state changes: if the timer is not running, start it for 10 milliseconds, then invoke the callback if the new state is different from the property (and update the property).
- when the timer expires: check the input state, and invoke the callback if it is different from the property (and update the property).

Main

Change the `button` declaration to make it a `DebouncedInput`.

Build and run

Now the counter should be incremented correctly.

Final step

In order to avoid dealing with `idf.py menuconfig` each time the project is cleaned, add the `sdkconfig.defaults` file to the root directory of the project. This enables the timer ISR dispatch (and other settings we'll need later) when `idf.py set-target esp32c6` is run.

Event Loop

*If you didn't complete the previous step, start with the **04-Debouncing** project.*

In order to call higher-level functions in the application when interrupts are triggered, we need to implement an event loop based on a queue.

Queue

Add `Queue.swift` to the project. `Queue.swift` implements a Swift wrapper for the FreeRTOS queue functions provided by the framework. For this reason, `#include "freertos/queue.h"` must be added to `BridgingHeader.h`.

`Queue` provides two methods to add elements:

- `send` can't be called from an interrupt handler (it blocks for the requested duration if the queue is full)
- `sendFromISR` is reserved for interrupt handlers; it throws if the queue is full.
  The `receive` method consumes and returns the first element in the queue. It can block for the requested duration if the queue is empty, and must not be called from an interrupt handler.

EventLoop

Create an `EventLoop.swift` file.

An event needs to provide a timestamp representing the moment it was created and inserted in the queue.
To that purpose, create a `TimedEvent<Event>` struct with two properties:

- the timestamp (Int64)
- the event itself (Event)
  `init(_ event: Event)` can initialize the timestamp property with `esp_timer_get_time()`.
  This `TimedEvent<Event>` struct will be used as the type of elements of the event queue.

Create an `EventLoop<Event>` class.
The role of this class is to allow event providers (such as interrupt handlers) to post events to the queue, and to dispatch events to clients in an infinite loop. Each client will provide a handler in the form:

```
typealias EventHandler = (Int64, Event) -> Void // first argument is timestamp
```

The `init()` function should take two parameters: the queue length and the event type. Implement a method to register an event handler, two methods to post an event (from an interrupt context and from the main context), and a run() method to start an infinite loop and dispatch the events to registered clients.

Main

In `Main.swift`, declare an `enum` to represent your events, with at least the `button` case. Rewrite the `main()` function to use an event loop:

- create an `eventLoop` property at the beginning of the function
- change the button closure to post a `.button` event with the state
- register a handler on the event loop to log the button state and toggle the LED state when the button is pushed
- remove the `while(true)` loop and run the event loop instead

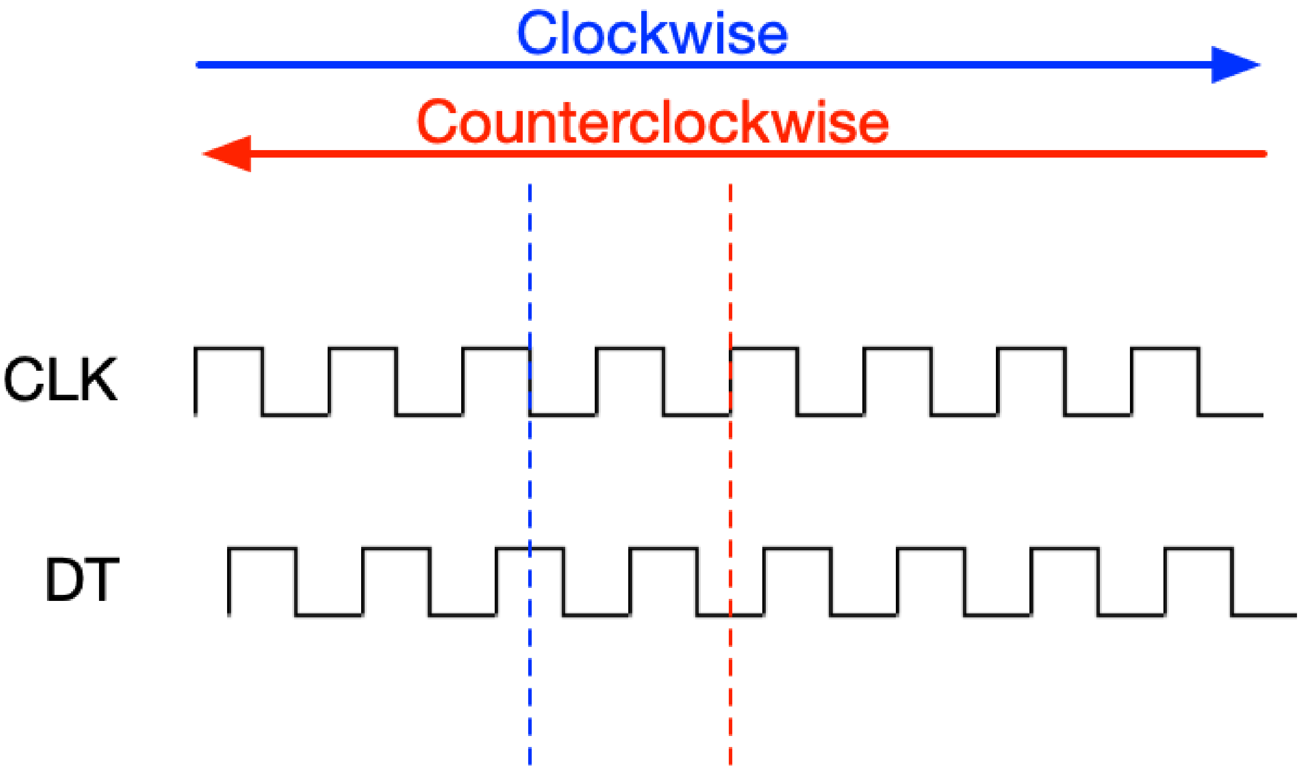
Timer

As a bonus, create a `Timer`, add it as a source to the event loop, use the button events to start/stop it (in repeating mode with a 100 ms duration), and use the timer events to control the LED.

Rotary Encoder

*If you didn't complete the previous step, start with the **05-Event-Loop** project.*

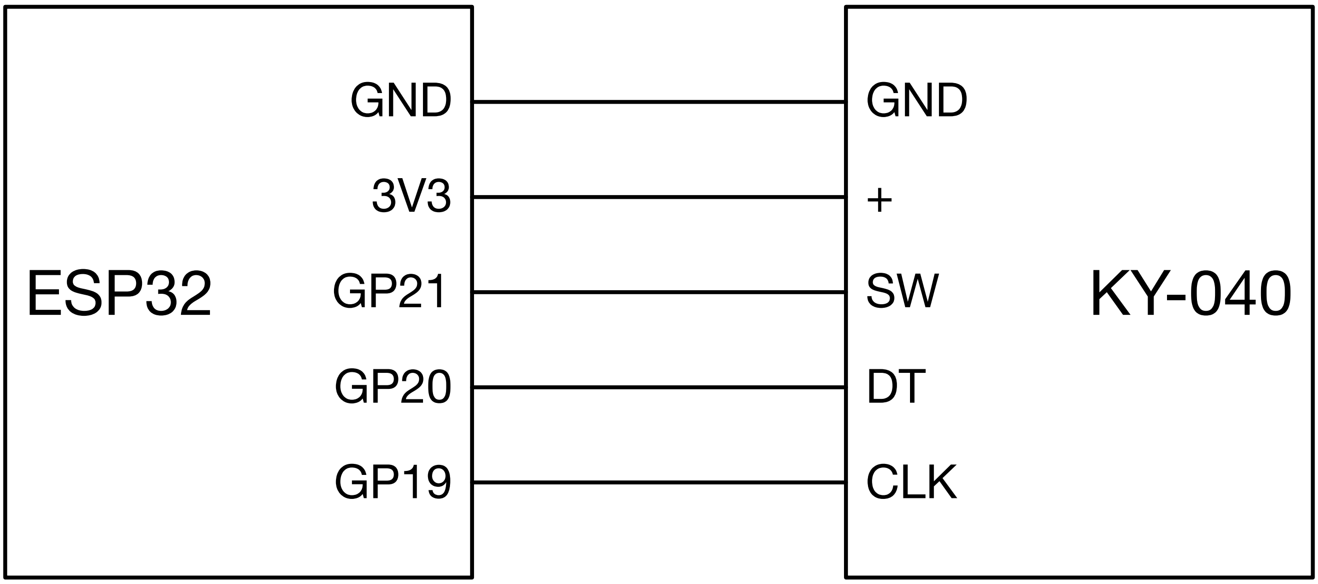
In addition to the switch we've been using so far, the KY-040 is a rotary encoder. It provides two signals, CLK and DT, to monitor the rotations of the knob.



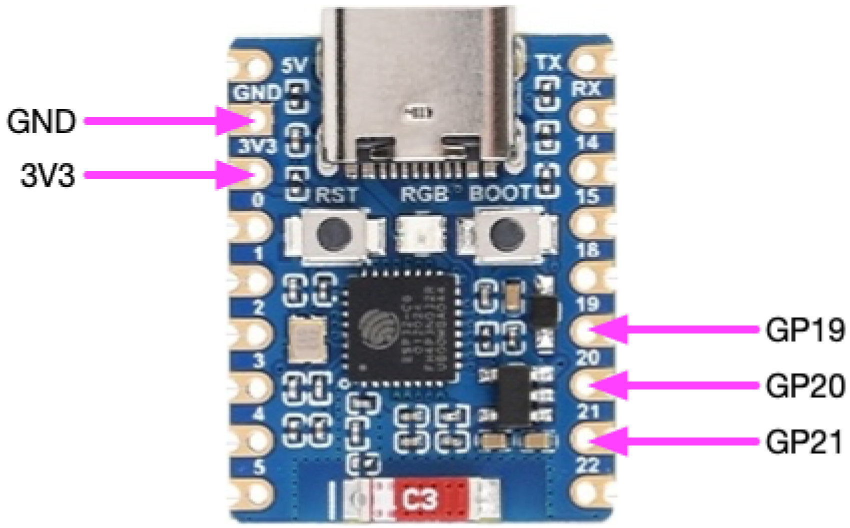
An easy way to decode the rotation events is to get the value of DT on each falling edge of CLK. If DT is high, the rotation is clockwise. If it is low, the rotation is counterclockwise.

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

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

- Add a case to the `Event` enum to represent a knob rotation
- Instantiate a RotaryController and post events in its callback
- In the event handler, increment the counter for a clockwise rotation and decrement it for a counterclockwise rotation, then log its value.

Display

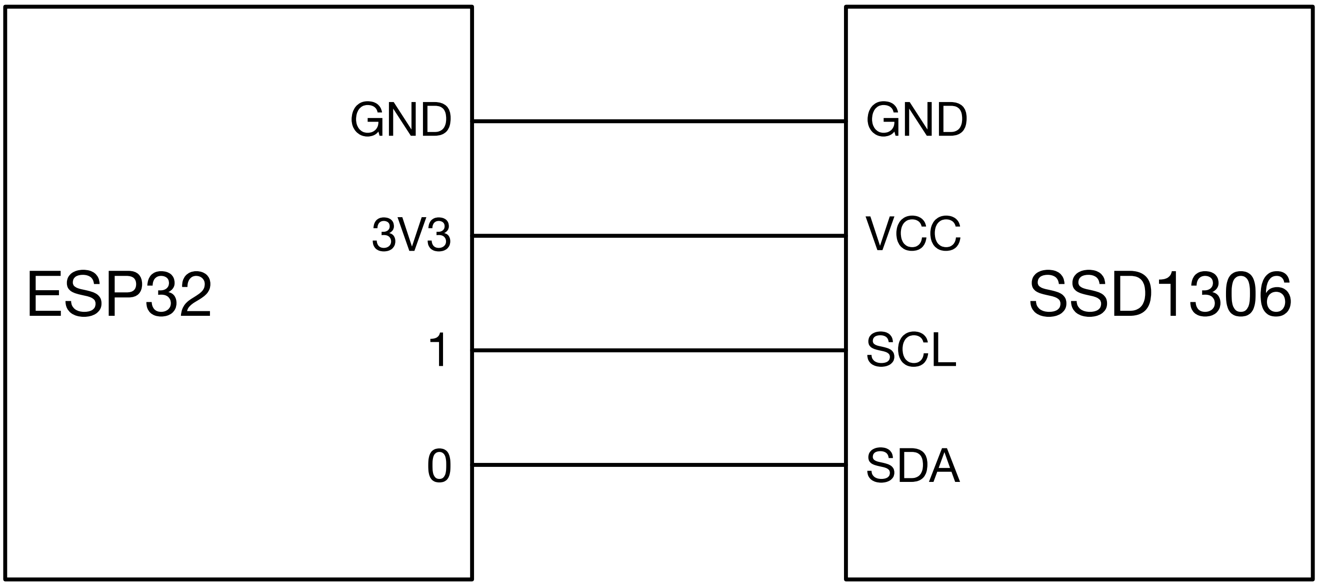
*If you didn't complete the previous step, start with the **06-Rotary-Encoder** project.*

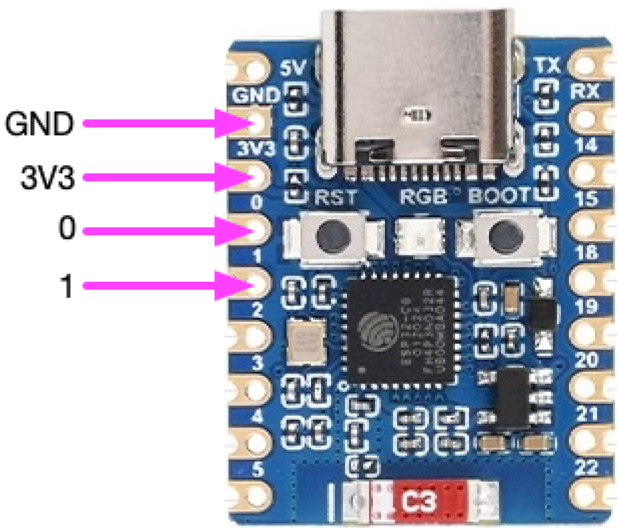
Let's use a third-party library to connect a I^2C display.

Wiring

The SSD1306 display implements the I^2C 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^2C 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.

Clone both repositories mentioned above, or download and extract the zip files. You should end up with `u8g2` and `u8g2-hal-esp-idf` directories.
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.

`u8g2-hal-esp-idf` does not support the C6 variant of the ESP32 microcontroller. As a temporary workaround, we can add this line right after the `#include` statements of `u8g2_esp32_hal.h` (line 17):

```
#define SOC_I2C_NUM 1
```

U8g2 defines its fonts as `const uint8_t[]` and the u8g2_SetFont() function expects `const uint8_t *`. 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 resources) 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, 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", at: Point(x: 10, y: 50), refresh: true)
```

In the event handler, display the value of the counter when it is updated.

The `Display` class is missing `frameRect` and `fillRect` methods, implement them.
Then, in the event loop, draw a bargraph to represent the value of the counter.


GND VCC SCL SDA
42

Bluetooth Peripheral

*If you didn't complete the previous step, start with the **07-Display** project and add the **u8g2** dependency to the **components** folder.*

Let's add a Bluetooth service to our device. This service will provide one characteristic representing the counter value.

Dependencies

The ESP-IDF framework provides a C API to enable and configure a BLE service. Additionally, the resources folder contains glue code to make these features available to a Swift application.

- add `ble-peripheral` to the `components` directory.
- add the `BLE` folder to the project
- make the C API visible to the Swift code by adding these lines to `BridgingHeader.h`:

```
#include "host/ble_uuid.h"
#include "host/ble_gatt.h"
#include "host/ble_gap.h"
#include "os/os_mbuf.h"

#include "ble_peripheral.h"
```

UUID

BLE service and characteristics are identified by a UUID. By default Bluetooth uses regular 128-bit UUIDs, but some "well-known" services and characteristics can use shorter 16- or 32-bit identifiers. https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Assigned_Numbers/out/en/Assigned_Numbers.pdf
The `BLEUUID` class provides initializers for the short UUIDs, but the 128-bit UUIDs are not fully implemented : `init?(uuid128: String)` calls the `decode()` function which always returns `nil`.
Implement the `decode()` function:

- the expected input is in the format `"%08X-%04X-%04X-%04X-%012X"` (return `nil` if the input string doesn't match this format)
- return a tuple with the 16 decoded bytes in reverse order
  Note that, while Embedded Swift doesn't prevent the use of advanced String functions, it

requires linking with `libUnicodeDataTables.a`, and that would exceed the flash capacity of our device. For this reason it is more efficient to deal with the UTF8 view.

Characteristic

Since our counter doesn't represent a "well-known" value, we should use a random 128-bit UUID. In the terminal, type `uuidgen` and use the returned value as the UUID.
Instantiate a `BLECharacteristic` with this UUID, and with messages of type `UInt8` (enough to represent values in the 0...100 range).
The characteristic supports a read handler and a write handler. The read handler will be used to return the value of the counter, and the write handler to update it and refresh the display.
Both handlers receive an arbitrary pointer (`UnsafeMutableRawPointer`). The read handler must return the number of bytes to be sent, and the write handler receives the number of bytes of the received value.
Since the handlers are called on another thread, it is recommended to use an event to update the display.

Service

Generate another UUID for the service.
Instantiate a `BLEService` with this UUID and the characteristic, and a `BLEServiceList` with this service.
Then start advertising with the device name of your choice.

Notifications

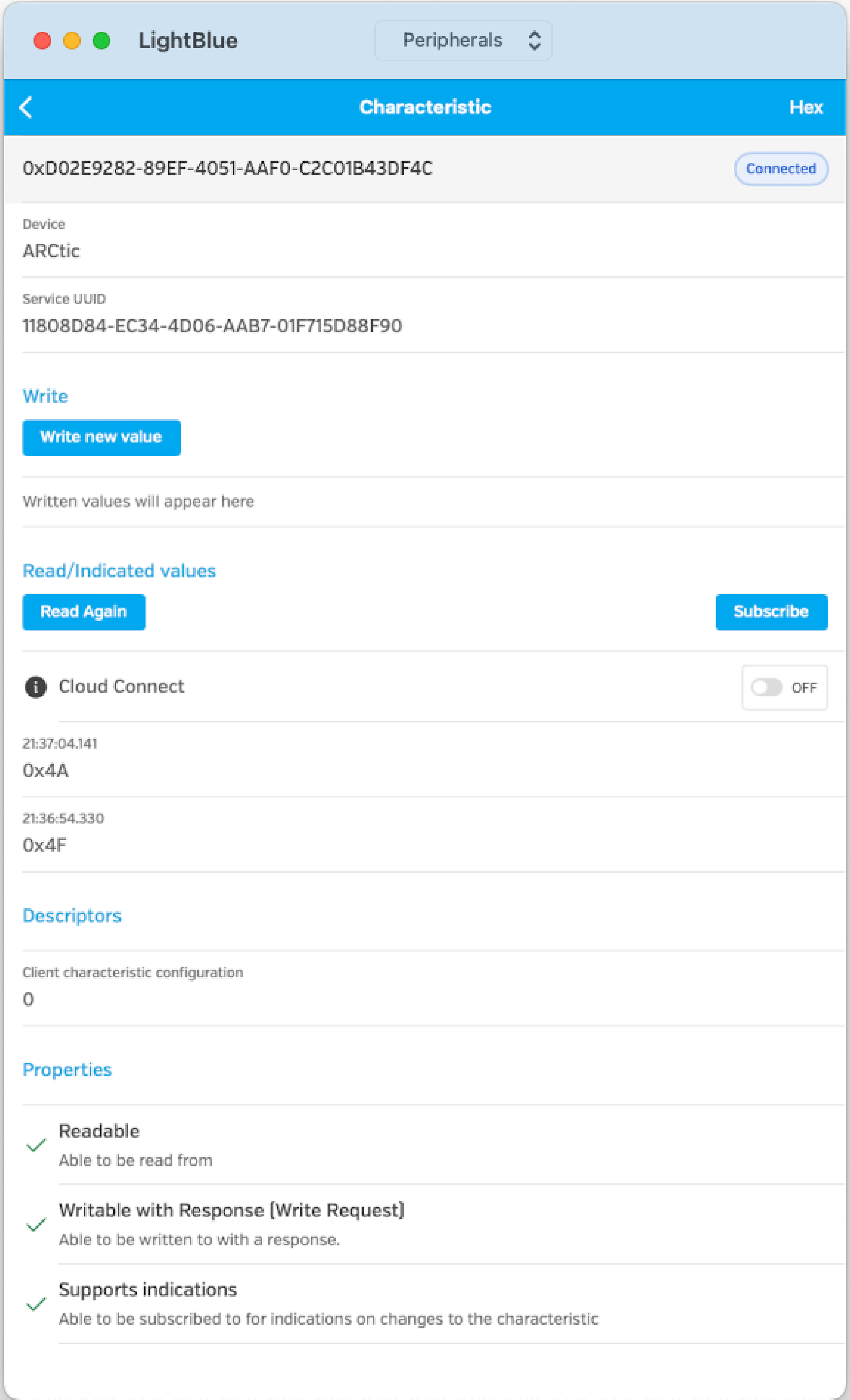
We can allow the BLE client to subscribe to values changes of the characteristic.
The `BLECharacteristic` must be initialized with `allowsSubscriptions: true`, and its `updateSubscribers()` must be called each time the value is updated.

Enable subscriptions when the characteristic is instantiated, and add the `updateSubscribers()` call to the event handler when the `.rotary` event is received.

Testing

Many applications are available to test a BLE device. **LightBlue** is available on macOS and iOS, and allows to scan for peripherals, list the services and characteristics, read/write values, and more. https://apps.apple.com/app/lightblue/id557428110.

Use LightBlue to verify that your device is advertised properly, make a connection, read and write values, and subscribe to notifications.


LightBlue
Peripherals
Characteristic
Hex
0xD02E9282-89EF-4051-AAF0-C2C01B43DF4C
Connected
Device
ARCtic
Service UUID
11808D84-EC34-4D06-AAB7-01F715D88F90
Write
Write new value
Written values will appear here
Read/Indicated values
Read Again
Subscribe
Cloud Connect
OFF
21:37:04.141
0x4A
21:36:54.330
0x4F
Descriptors
Client characteristic configuration
0
Properties
Readable
Able to be read from
Writable with Response (Write Request)
Able to be written to with a response.
Supports indications
Able to be subscribed to for indications on changes to the characteristic

iOS Application

*If you didn't complete the previous step, start with the **08-Bluetooth-Peripheral** project, add the **u8g2** dependency to the **components** folder, and generate UUIDs to replace the values for **BLEService** and **BLECharacteristic** in **main.swift**.*

It's time to create an iOS application to monitor and control our device.

Project

Create a new Xcode project for a SwiftUI-based iOS application.

In order to use Core Bluetooth, iOS requires an `Info.plist` key.
Add a `NSBluetoothAlwaysUsageDescription` property to `Info.plist` with a message saying your application needs a Bluetooth connection.

Add the `BluetoothConnectionManager.swift` file from the workshop resources.
The `BluetoothConnectionManager` class is a minimal implementation of a BLE client, allowing to connect to one service providing one characteristic.
`BluetoothConnectionManager` implements delegate methods for the required Core Bluetooth protocols, and it exposes (with `@Observable`) the current state of the discovery and connection, as well as read-write access to the characteristic.

In the `ContentView` struct, add a `@State` instance of `BluetoothConnectionManager`, and initialize it with the service and characteristic UUIDs defined in your embedded application. The value type of the characteristic should be `UInt8`.

In the body, display contents according to the state of your `BluetoothConnectionManager` instance:

- `unavailable`: display the error message given by BluetoothService
- `ready`: display a button to start scanning for devices
- `scanning`: display a progress indicator and a button to cancel the scan
- `discovered`: display the list of device names as buttons; a tap on a device should trigger a call to the connect(identifier:) method
- `connecting`: display a progress indicator
- `connected`: display the value when it is available

Build and run your application with your embedded device powered on. You should be able to initiate a scan, view the list of devices, and establish a connection with your device.

`BluetoothConnectionManager` is not complete: the methods to read and write values from/to the device are not implemented.

- update the `peripheral(_:didUpdateValueFor:error:)` delegate method to set the `readValue` property.
- update the `write(value:)` method to send the value to the characteristic (you can use `CBPeripheral.writeValue(_:for:type:)` to send data over Bluetooth)

Then the value should be displayed when you connect your app to the device, and it should update automatically when you move the knob.

As a last step, you can implement bidirectional communication to allow the iOS app to change the value on the device.

- in the `connected` state of your SwiftUI view, add a 0-100 slider when `readValue` is available; this slider should be bound to a `sliderValue` property declared `@State`.
- use `onChange(of: readValue)` to update `sliderValue`, and `onChange(of: sliderValue)` to send the new value to the device.

AccessorySetupKit

*If you didn't complete the previous step, start with the **09-iOS-Application** project, and use the **08-Bluetooth-Peripheral** project if necessary.*

Let's make the discovery process nicer with AccessorySetupKit!

Info.plist

- `AccessorySetupKit` can work with Bluetooth and Wi-Fi. The supported discovery modes must be specified in an array keyed by `NSAccessorySetupKitSupports`. For our app, the array will contain a single element: "Bluetooth".
- `AccessorySetupKit` takes care of the whole Bluetooth scan procedure. In order to detect and return the relevant devices, it needs to know the list of services the app is interested in. This list must be provided as an array of UUIDs whose key is `NSAccessorySetupBluetoothServices`.

DiscoveryService

Create a `DiscoveryService.swift` file importing `AccessorySetupKit` and `CoreBluetooth`, and a `DiscoveryService` class.

This class should expose a callback of type `(Event) -> Void`, with Event defined as an `enum` with at least two cases:

- `activated([RemoteDevice])`
- `selected(RemoteDevice)`

The initializer for this class should instantiate an `ASAccessorySession` (to be stored in a private property), and `activate` this session on the main queue with a closure to handle events.
In this closure, the following event types should be handled:

- `activated`: this event is sent at startup. If devices have been picked by the user in a previous session, `session.accessories` is not empty. In that case, invoke the callback with an `activated` event containing an array of `RemoteDevice` (it can be helpful to add an extension to `RemoteDevice` providing an initializer with an `ASAccessory`)
- `accessoryAdded`, `accessoryRemoved`: update a `selectedAccessory` property of this class accordingly (no event should be sent at this time).
- `pickerDidDismiss`: invoke the callback with a `selected` event if `selectedAccessory` is not `nil`.

Implement a `showPicker(serviceUUID: CBUUID)` method.
This method should create an `ASDiscoveryDescriptor`, set its `bluetoothServiceUUID` to the service UUID, and use this descriptor to create an `ASPickerDisplayItem`. The `productImage` can rely on the `ESP32-C6.png` that you can add to the `Assets` catalog.
Eventually this method should invoke the `ASAccessorySession`'s `showPicker` method with the `ASPickerDisplayItem`.

BluetoothConnectionManager

Now the `BluetoothConnectionManager` class can be updated to take advantage of this new `DiscoveryService`.

Add a `discoveryService` property to `BluetoothConnectionManager`, and instantiate it at `init` time.
Its callback function should perform these actions according to the received event:

- `activated`: set the state to `discovered` with the peripherals returned by the discovery service
- `selected`: initiate a connection, using the device identifier

When AccessorySetupKit is used, `CBCentralManager` doesn't show the `poweredOn` state until after the user has picked an accessory. For this reason, the device should be considered `ready` (instead of `unavailable`) in the `poweredOff` state. Update `centralManagerDidUpdateState` with this change.

Testing

When the app is used for the first time, it should display a Scan button. This button should display the accessory picker, and your device should be visible if it is powered on. Selecting it should initiate the connection and display the view with the slider.

Then the connected device should be remembered by the framework. Running the app again should display the name of the device and allow to initiate the connection.