SIGPwny

Embedded FA2025 • 2025-09-22

Embedded 102: Microcontroller Programming

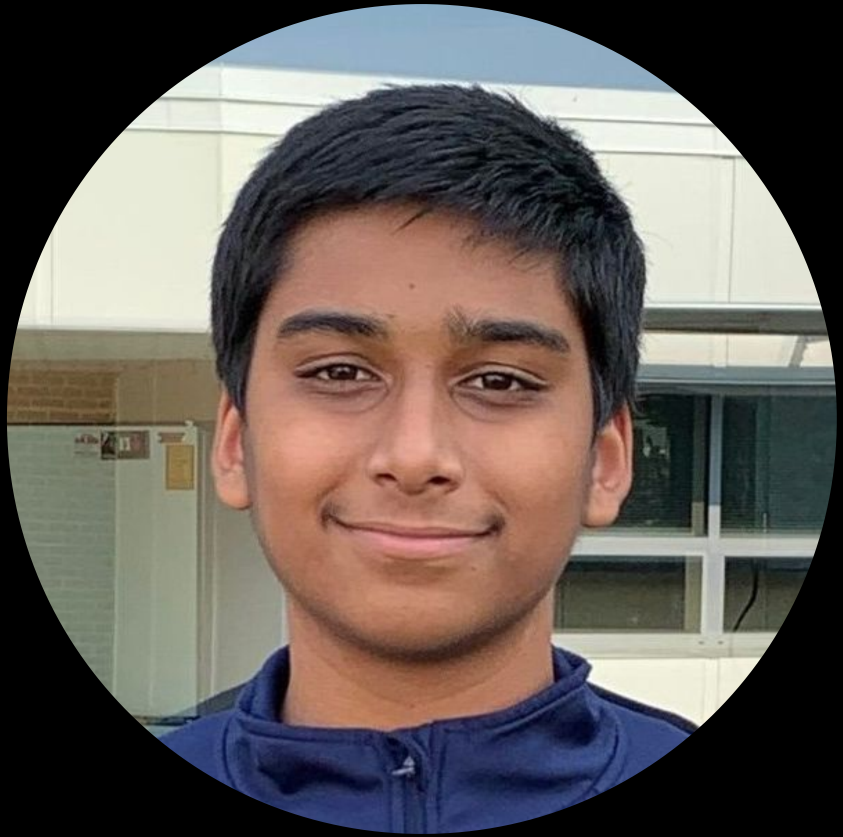
Nikhil and Krishnan

Nikhil Date

- Admin
- Computer Science
- Fact: I lived in Columbus this summer

Krishnan Shankar

- SIGPwny Helper
- Computer Engineering ‘28
- Fun fact: I’m from the Washington, D.C Area

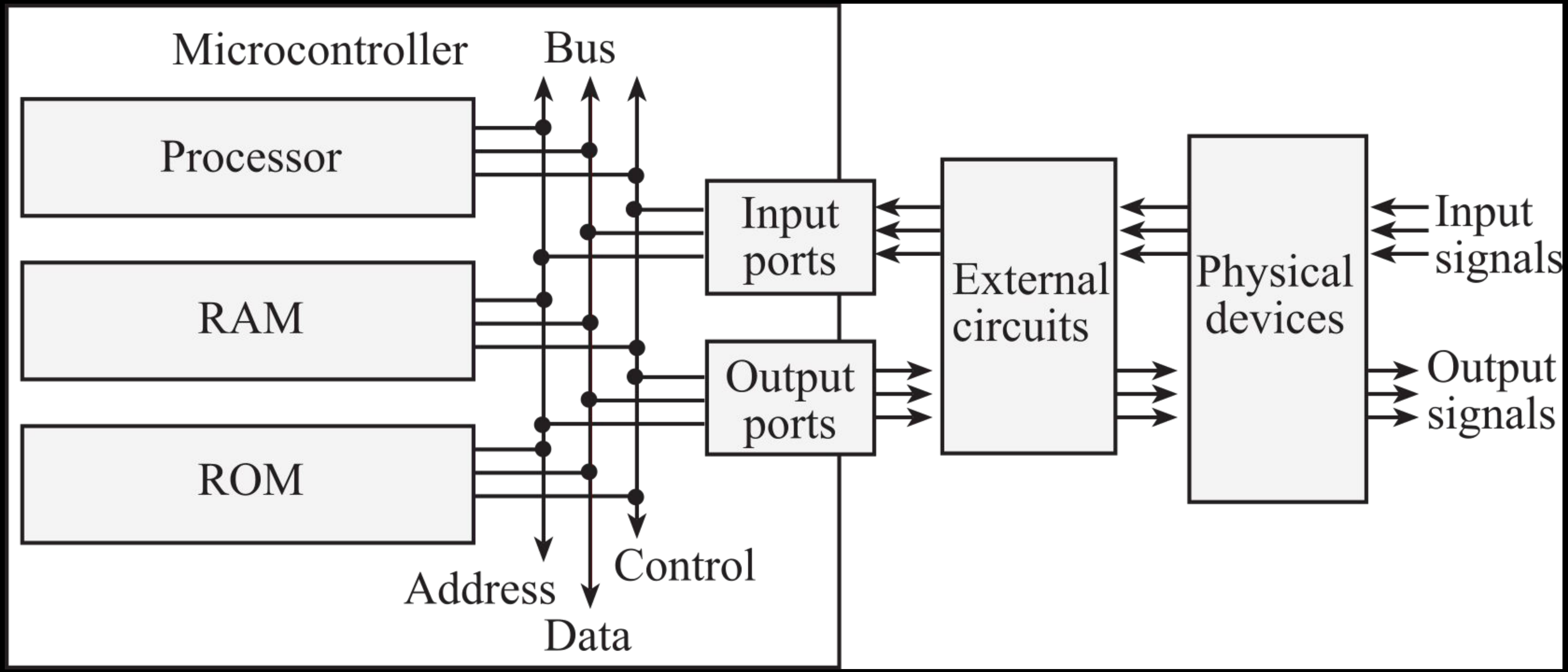


Meeting content can be found at sigpwny.com/meetings.

SIGPwny

The Basics

Recall: Embedded Systems



1. The C programming language

- Compiled “high-level” programming language
- Created in the 70s by Dennis Ritchie while developing Unix
- Widely used for low-level programming
- Offers easy access to memory and hardware peripherals

```
#include “stdio.h”
int main() {
    int x = 1;
    int y = 2;
    int z = x + y;
    printf(“z = %d”, z);
    return 0;
}
```

2. Compilation

- The C program gets turned into assembly code
- Which assembly language?
  - You can pick!
  - e.g., you can compile your code “for” x86_64 (i386), ARM (aarch64), etc.

2. Compilation

```c
#include "stdio.h"
int main() {
    int x = 1;
    int y = 2;
    int z = x + y;
    printf("z = %d", z);
    return 0;
}
```

```asm
main:
        push    {r11, lr}
        mov     r11, sp
        sub     sp, sp, #24
        mov     r0, #0
        str     r0, [sp, #4]
        str     r0, [r11, #-4]
        mov     r0, #1
        str     r0, [r11, #-8]
        mov     r0, #2
        str     r0, [sp, #12]
        ldr     r0, [r11, #-8]
        ldr     r1, [sp, #12]
        add     r0, r0, r1
        str     r0, [sp, #8]
        ldr     r1, [sp, #8]
        ldr     r0, .LCPI0_0
.LPC0_0:
        add     r0, pc, r0
        bl      printf
        ldr     r0, [sp, #4]
        mov     sp, r11
        pop     {r11, lr}
        bx      lr
.LCPI0_0:
```

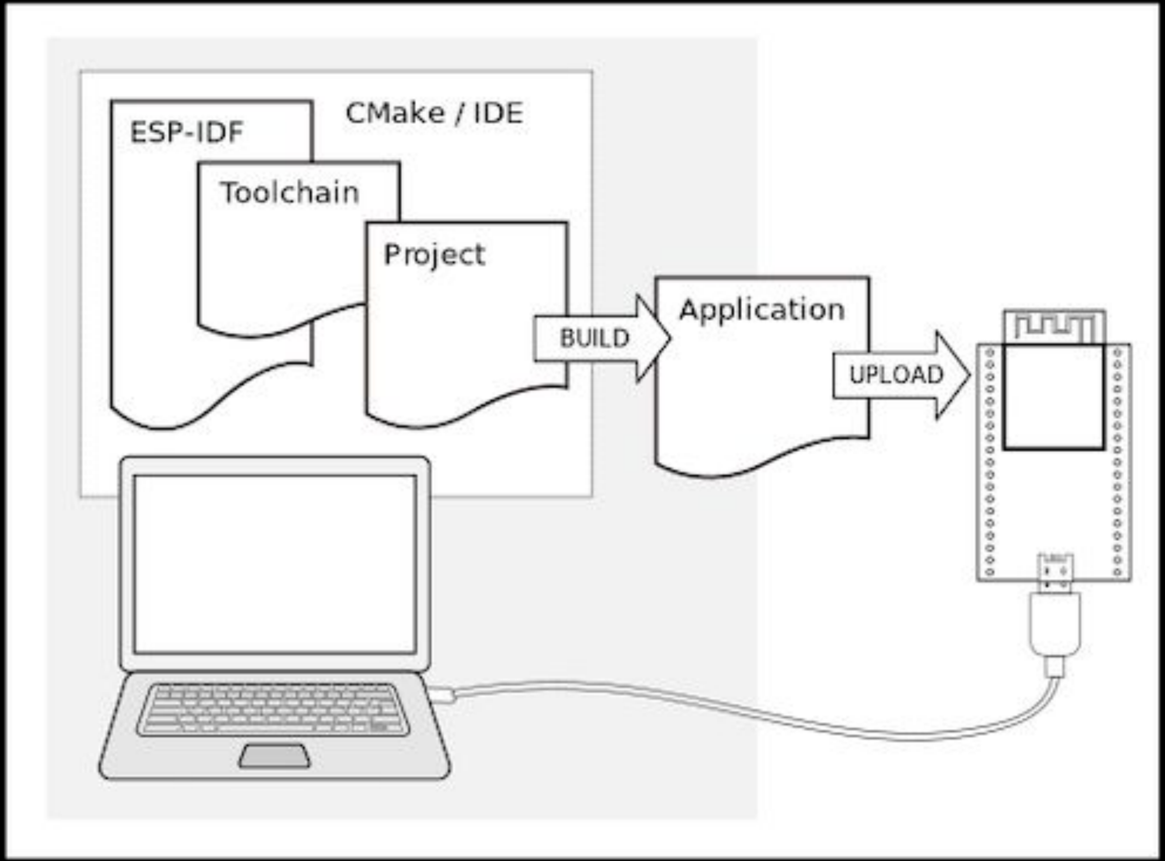
3. Bitstream

- Based on the ISA (instruction set architecture), every assembly instruction “becomes” a specific binary string
- This, along with some other things, forms an object file
- This object file, along with some other things, forms a bitstream
- This bitstream gets sent to the MCU over USB (using UART)

Flashing your FallCTF Badges!

The Overview

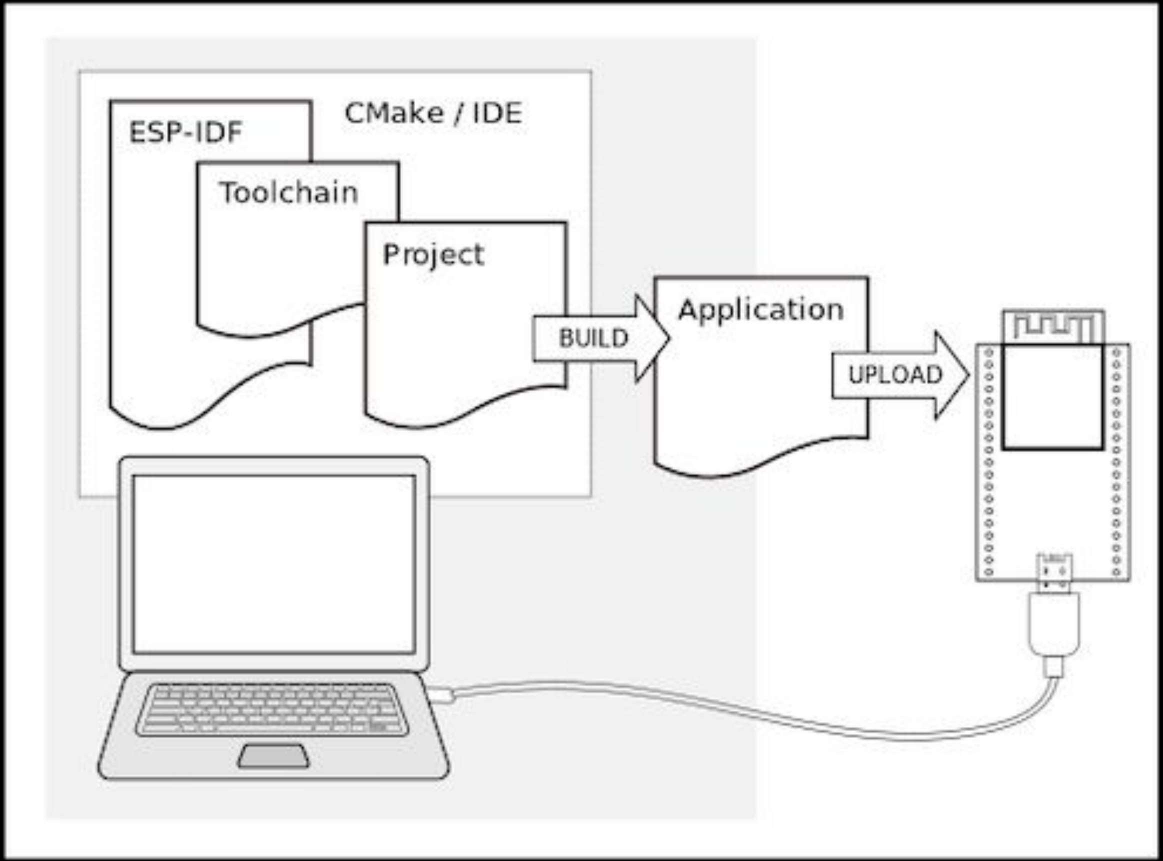
- We need a toolchain to compile our C code (`xtensa-esp-elf-*`)
- We need a build tool to create an ESP32-compatible application (CMake, Ninja)
- We need a flashing tool to copy that application to Flash Memory (esptool.py)



Source: ESP-IDF Programming Guide

The Overview

- We need a toolchain to compile our C code (`xtensa-esp-elf-*`)
- We need a build tool to create an ESP32-compatible application (CMake, Ninja)
- We need a flashing tool to copy that application to Flash Memory (esptool.py)
- **Espressif's IDF handles all of these!**



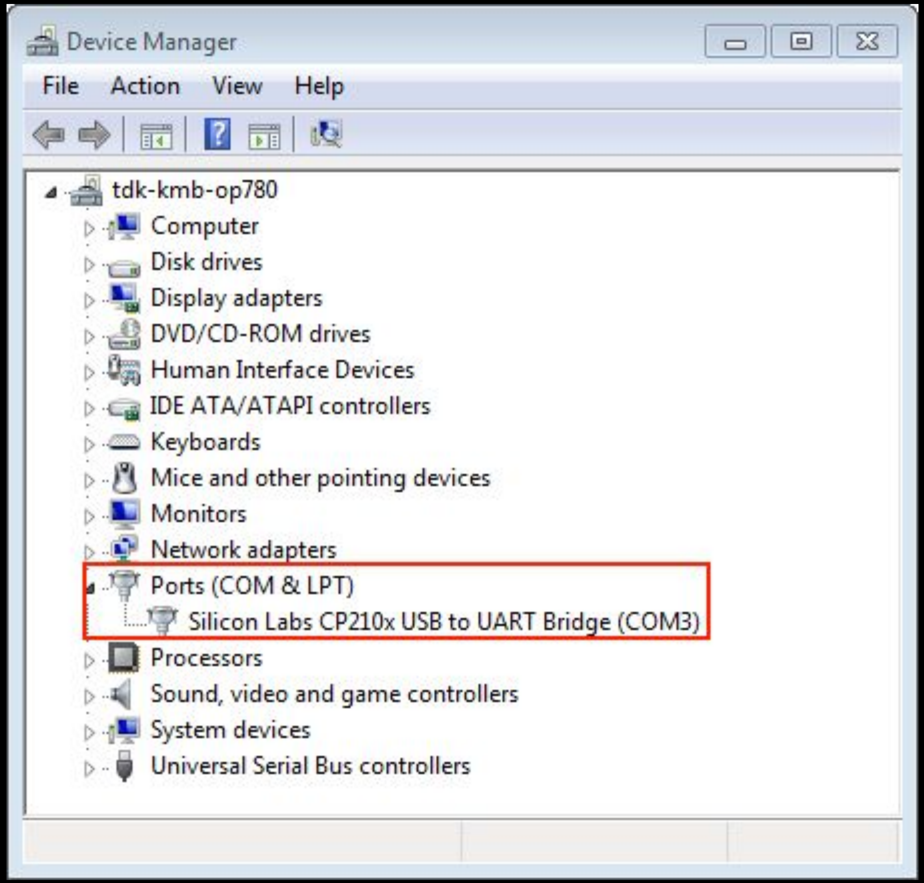
Source: ESP-IDF Programming Guide

Installing ESP IDF

- **Esp**ressif **I**oT **D**evelopment **F**ramework
- https://docs.espressif.com/projects/esp-idf/en/stable/esp32s2/get-started/index.html
- Simplest method:
  - Install the “ESP-IDF” VSCode Extension
  - Find the ESP-IDF tab on the left
  - Click “Configure ESP-IDF Extension”
  - Use “GitHub” download server
  - Click “Install”

Find your ESP32's Serial Port

- Plug in the ESP32
- On Windows:
  - Open "Device Manager"
  - Find the port that's newly added
  - Should be COM#
- On Mac/Linux:
  - `/dev/ttyUSB0` or `/dev/ttyACM0`
  - Check which one gets created when you plug in the ESP32 (`ls /dev`)
- Replace "PORT" in any future commands with this value



Source: ESP-IDF Programming Guide

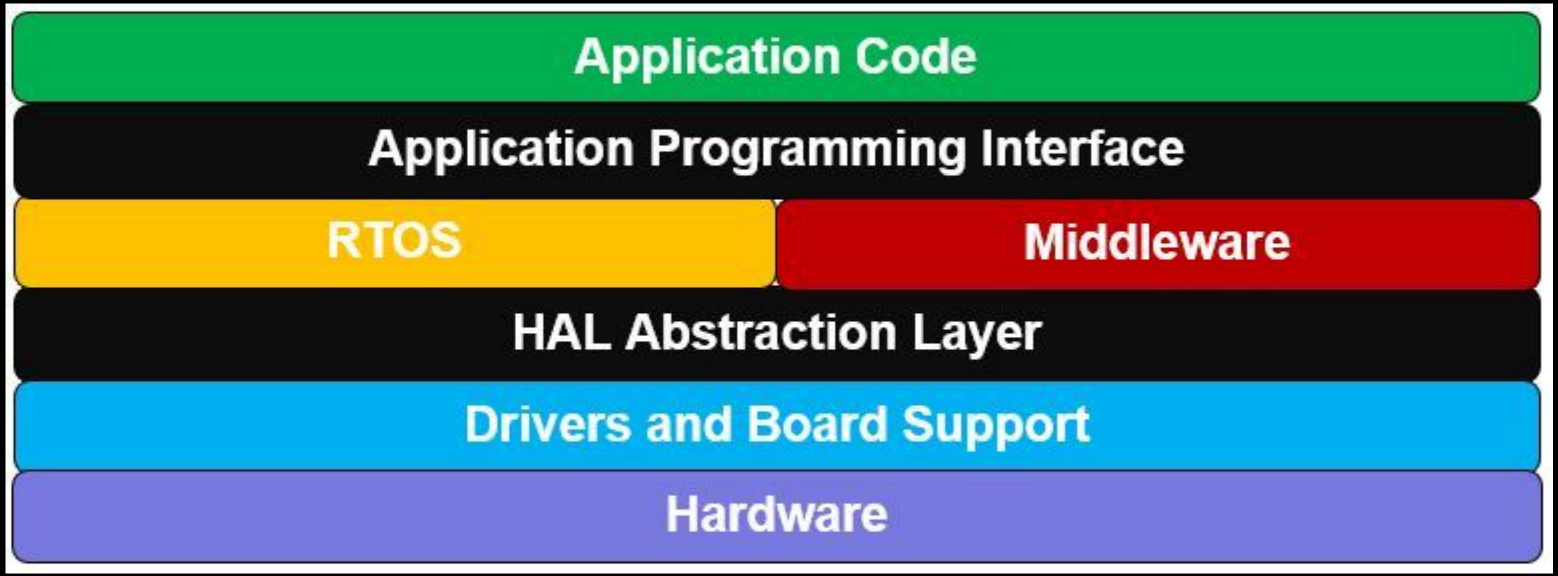
Download an Example Project

- They're actually already downloaded on your PC! (If you installed ESP IDF correctly)
- Use VSCode extension to get the hello_world project

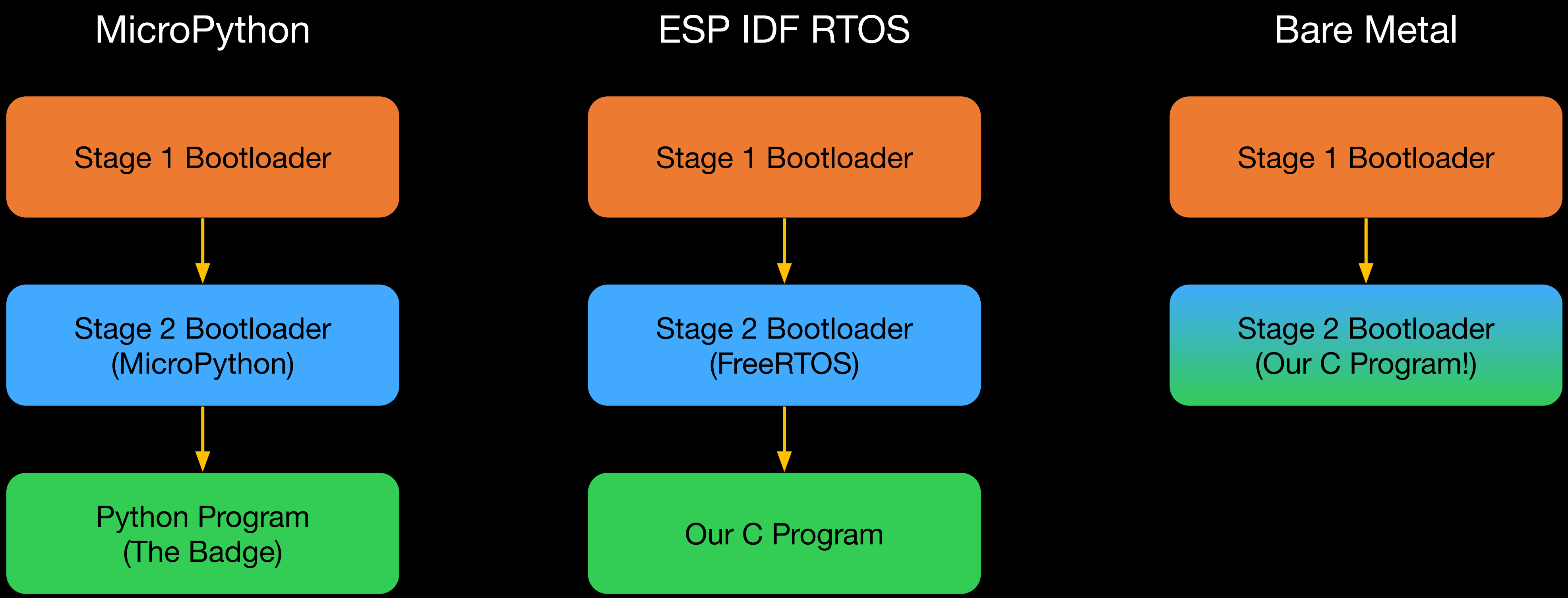
Flash your FallCTF Badges

- `cd project-directory`
- `idf.py set-target esp32s2`
- `idf.py menuconfig`
  - Component config > ESP System Settings > Channel for console output
  - Change from “Default: UART0” to “USB CDC”
- Switch the ESP to bootloader mode (aka download mode)
  - Press and hold “B” on the front (boot button)
  - Press and release “SW4” on the back (reset button)
  - Release “B”
- `idf.py -p PORT -b 115200 flash`
- `idf.py -p PORT -b 115200 monitor`

Embedded Software Stack



An Aside: RTOS vs Bare Metal Boot Process



The Boot Process, in words

- Stage 1 Bootloader
  - Hardcoded, cannot be modified
  - Once finished, it jumps to a fixed memory address (`0x1000`)
- Stage 2 Bootloader
  - Is flashed to address `0x1000`
  - Can be anything we want, but is generally FreeRTOS

Flashing

Then deploy the firmware to the board, starting at address 0x1000:

```
esptool.py --baud 460800 write_flash 0x1000 ESP32_BOARD_NAME-DATE-VERSION.bin
```

Replace `ESP32_BOARD_NAME-DATE-VERSION.bin` with the `.bin` file downloaded from this page.

Source: [MicroPython (ESP32 S2)]()

The Boot Process, in words

- Stage 1 Bootloader
  - Hardcoded, cannot be modified
  - Once finished, it jumps to a fixed memory address (`0x1000`)
- Stage 2 Bootloader
  - Is flashed to address `0x1000`
  - Can be anything we want, but is generally FreeRTOS
- FreeRTOS
  - Reads address `0x8000` to view a list of all flashed images
  - Jumps to address `0x10000` to run the first user program
- The User Program
  - Is flashed to address `0x10000`

So How Do We Run Bare Metal?

- Write a C Program
  - To run bare metal, it should have a call_start_cpu0 function
- Write a custom linker script
  - This tells the compiler where to put certain parts of main.c
  - For example, code in IRAM and variables in DRAM
- Write a Makefile
  - This will compile your code to main.elf using the linker script
- Convert the ELF file to a binary image using esptool
- Flash the binary image at 0x1000 using esptool

The Bare Metal C Program

```c
#include <string.h>

extern unsigned int _sbss, _ebss, _sidata, _sdata, _edata;

void __attribute__((noreturn)) call_start_cpu0() {
   memset(&_sbss, 0, (&_ebss - &_sbss) * sizeof(_sbss));
   memmove(&_sdata, &_sidata, (&_edata - &_sdata) * sizeof(_sdata));

   main();
}

static volatile int a = 0;

int main(void) {
   while (1) {
       ++a;
   }
   return 0;
}
```

Source: Vivonomicon

Compile and Flash

- `cd project-directory`
- `make`
- `esptool.py -c esp32s2 elf2image --flash_mode="dio" --flash_freq "40m" --flash_size "4MB" -o main.bin main.elf`
- Switch the ESP to bootloader mode
- `esptool.py -c esp32s2 -p PORT -b 115200 --before default_reset -a hard_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_size detect 0x1000 main.bin`

Then What?

- The program is flashed, and esptool already rebooted it for us
- However, we won't see anything when we use idf.py monitor
    - Or even tools like screen, picocom, minicom, etc.
- Why?

Drivers

- To do anything useful, we need drivers
- To access the serial port, the device needs to communicate using UART (Universal Asynchronous Receiver-Transmitter)
- This is difficult (and complex)!

```
#define DLAB (1 << 7)
#define DR (1 << 0)
#define THRE (1 << 5)
#define DRIE (1 << 0)
#define THREIE (1 << 1)

void uart_init(void) {
    UART0.lcr = 0x0; // DLAB=0
    UART0.ier = DRIE; // enable DR intr.
}

struct uart_regs {
    union {
        char rbr; // DLAB=0, read
        char thr; // DLAB=0, write
        uint8_t dll; // DLAB=1
    };

    union {
        uint8_t ier; // DLAB=0
        uint8_t dlm; // DLAB=1
    };

    union {
        uint8_t iir; // read
        uint8_t fcr; // write
    };

    uint8_t lcr;
    uint8_t mcr;
    uint8_t lsr;
    uint8_t msr;
    uint8_t scr;
};
```

```
#define RBUFSZ 64 // must be a power of 2

struct ringbuf {
    volatile int hpos; // head position (from where chars are removed)
    volatile int tpos; // tail position (where chars are inserted)
    char data[RBUFSZ];
};

// Initialize a ring buffer (fixed-size FIFO)
void rbuf_init(struct ringbuf * rbuf) {
    rbuf->hpos = 0;
    rbuf->tpos = 0;
}

// Insert a character into the ring buffer at the tail of the queue
void rbuf_put(struct ringbuf * rbuf, char c) {
    rbuf->data[rbuf->tpos++ % RBUFSZ] = c;
}

// Remove a character from the ring buffer from the head of the queue
char rbuf_get(struct ringbuf * rbuf) {
    return rbuf->data[rbuf->hpos++ % RBUFSZ];
}
```

Source: K. Levchenko (ECE 391)

Next Meetings

2025-09-29 • Next Monday

- Embedded 103: Breadboarding and Hardware

Meeting content can be found at sigpwny.com/meetings.

SIGPwny