Firmware Thoughts

With our

System

In 6.9000

Two Big Pieces

- As you design your projects your firmware will get more and more complicated…

- Process/Instructions/Behavior:
  - How to structure all your code (RTOS?)

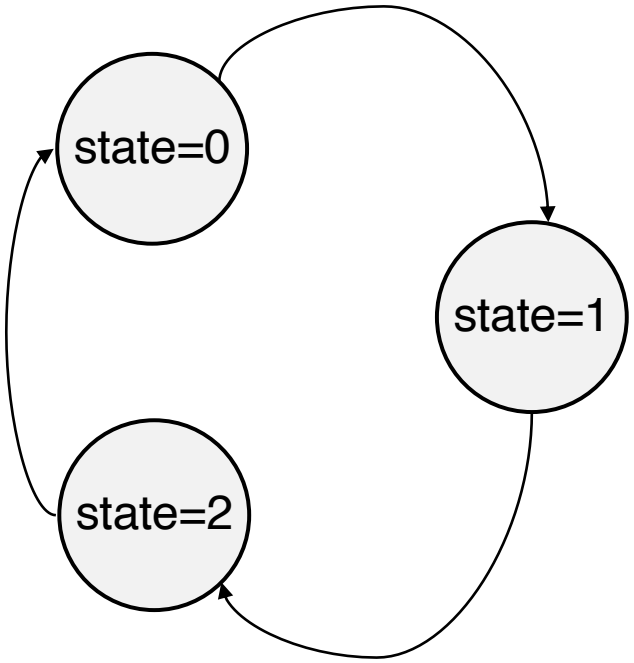
- Memory:
  - How to store all your data
  - Issues with this maybe

Process/Behavior: Thoughts for 6.9000

- For the most part, based on what I can see at this point, none of you are going to be needing to squeeze the last little bit of performance out of our microcontrollers.
- However a lot of you teams are actually doing pretty complicated meshes of interwoven behaviors that all need to run seamlessly and then we have to worry about using energy intelligently as well.

Finite State Machine

- Many of your tasks are going to boil down to running finite state machines type of behaviors.



- You do a thing
- You do another thing
- You do another thing
- You repeat

- And so on…

One way…

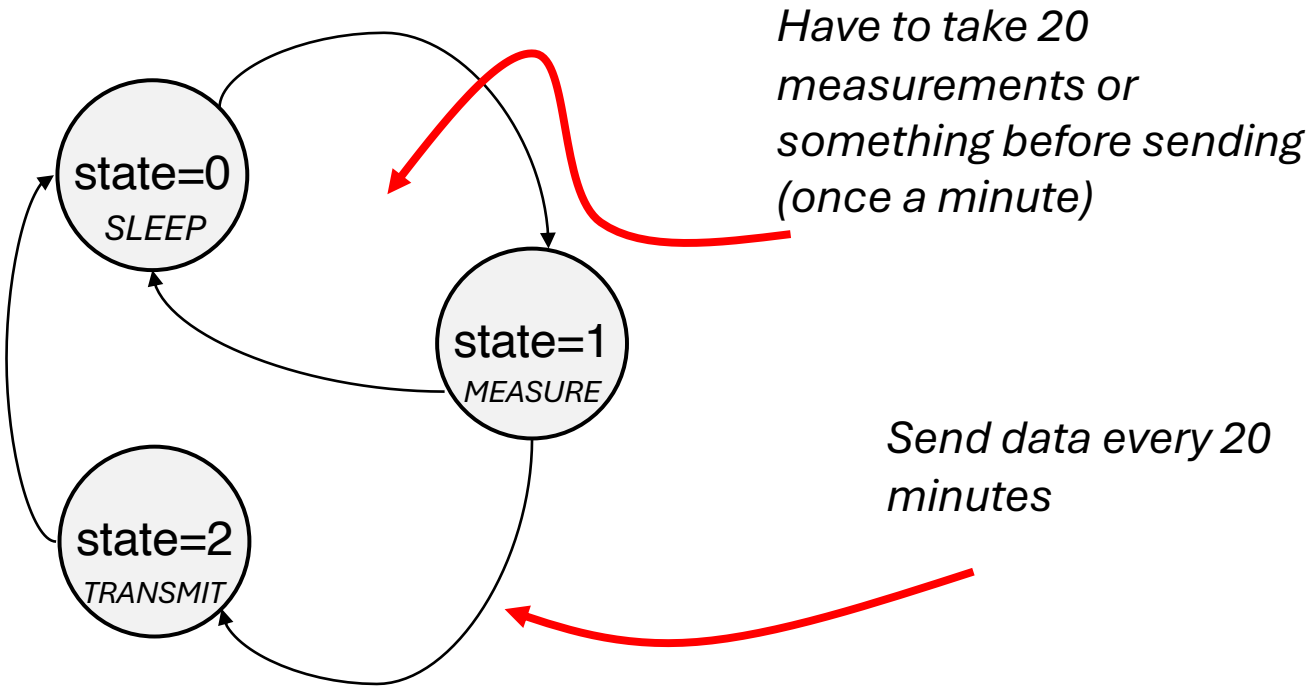
- One way to implement FSM code

```c
typedef enum {
    STATE_1,
    STATE_2,
    STATE_3
} state_t;

static state_t current_state = STATE_1;

void f1(void) {
    printf("f1\n");
    current_state = STATE_2;
}

void f2(void) {
    printf("f2\n");
    current_state = STATE_3;
}

void f3(void) {
    printf("f3\n");
    current_state = STATE_1;
}
```

```c
void app_main(void) {
    while (1) {
        switch (current_state) {
            case STATE_1: f1(); break;
            case STATE_2: f2(); break;
            case STATE_3: f3(); break;
        }
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

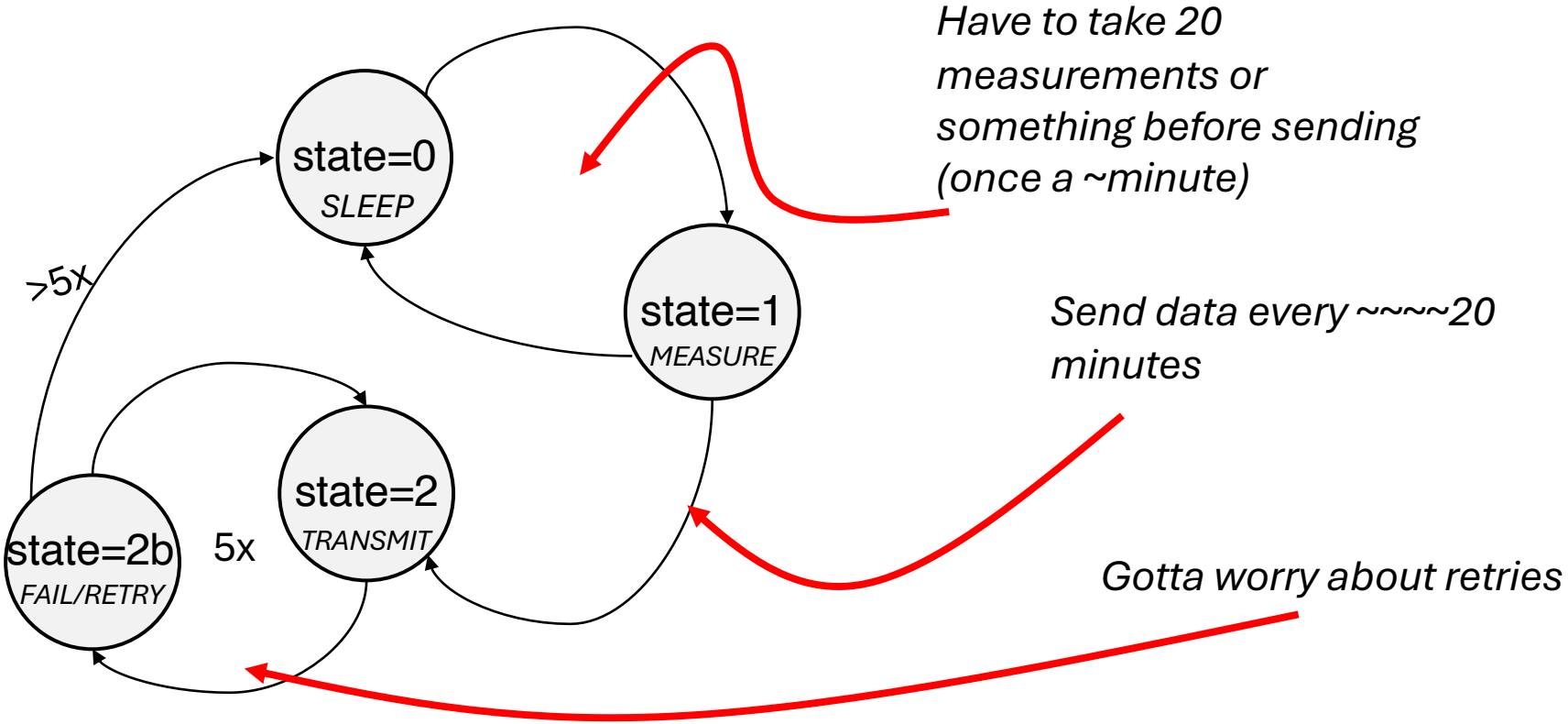
Finite State Machine

- You may start having mini FSM loops and larger FSM loops as you build up your overall design



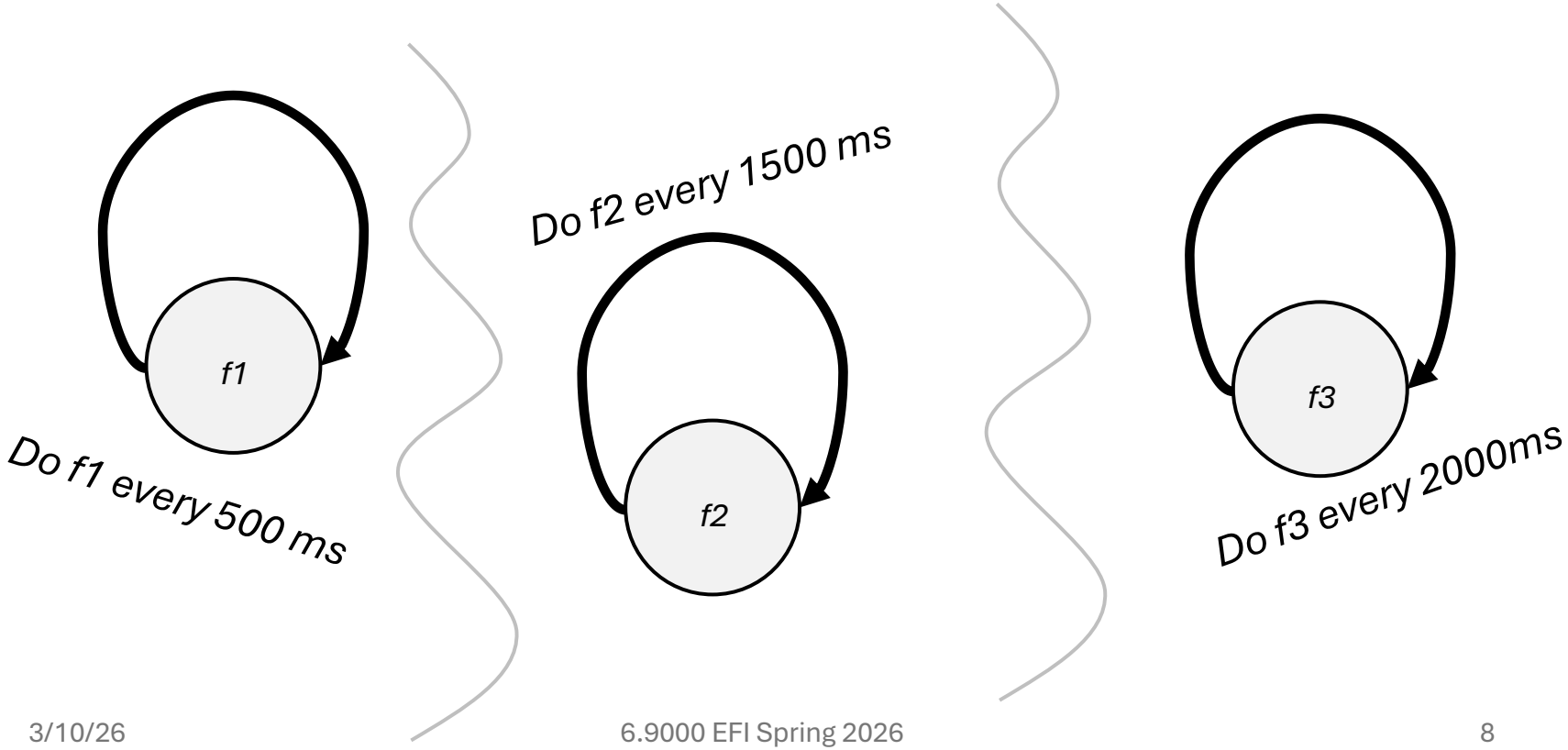
Finite State Machine

- As things get more complicated….



Another Functional Pattern Emerging

- Lots of little “things” you gotta do periodically which aren’t necessarily tied together.



“Super Loop”/Polling/Event Loop

- One loop that runs continuously and you use timers to check on when to run “tasks”

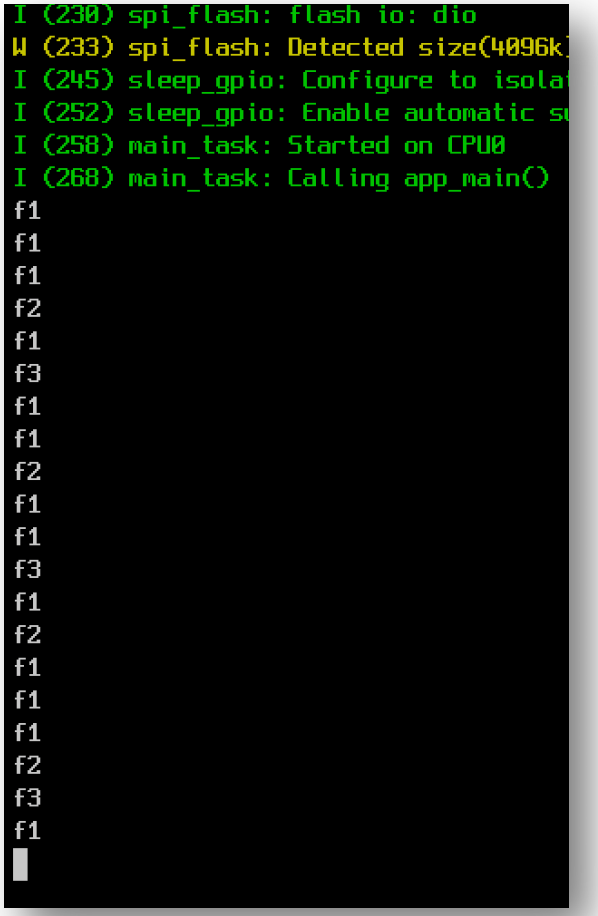
```c
void f1(void) {
    printf("f1\n");
}

void f2(void) {
    printf("f2\n");
}

void f3(void) {
    printf("f3\n");
}
```

```c
void app_main(void) {
    uint64_t last_f1 = 0;
    uint64_t last_f2 = 0;
    uint64_t last_f3 = 0;
    while (1) {
        uint64_t now = esp_timer_get_time();
        if (now - last_f1 >= 500000) {      //
            f1();
            last_f1 = now;
        } else if (now - last_f2 >= 1500000) {
            f2();
            last_f2 = now;
        } else if (now - last_f3 >= 2000000) {
            f3();
            last_f3 = now;
        }
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```

Event Loop In action…

- You can see the three functions are all kinda “running in parallel”

```
void app_main(void) {
    uint64_t last_f1 = 0;
    uint64_t last_f2 = 0;
    uint64_t last_f3 = 0;
    while (1) {
        uint64_t now = esp_timer_get_time();  /
        if (now - last_f1 >= 500000) {      //
            f1();
            last_f1 = now;
        } else if (now - last_f2 >= 1500000) {
            f2();
            last_f2 = now;
        } else if (now - last_f3 >= 2000000) {
            f3();
            last_f3 = now;
        }
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```



"Inputs" to system don't have to be just time

```
    uint64_t last_f1 = 0;
    uint64_t last_f2 = 0;
    uint64_t last_f3 = 0;

    while (1) {
        uint64_t now = esp_timer_get_time();

        if (now - last_f1 >= 500000) {
            f1();
            last_f1 = now;
        } else if (now - last_f2 >= 1500000) {
            f2();
            last_f2 = now;
        } else if (now - last_f3 >= 2000000) {
            if (gpio_get_level(GPIO_NUM_5) == 0) {
                f3();
            }
            last_f3 = now;
        }

        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```

- Can bring in lots of checks and stimuli to cause flow control variations in event loop

Only do a thing every 2 seconds if a button is pushed
Or something…

Can be perfectly fine way to run things

- But as your state machine(s) get more and more complicated…
- And the rate they need to step gets more complicated and varied…
- this can start to get harder and harder to manage
- May need to develop non-trivial structures and protocols to share information between all those separate

Solution 1: Interrupts

- Interrupts provide a manner for external signals (timers, electrical signals, bytes arriving) to trigger code
- You write a function, link it to

Push Interrupt

```c
#define BUTTON_PIN  GPIO_NUM_5

static void IRAM_ATTR button_isr(void *arg) {
    printf("pressed\n");
}

void app_main(void) {
    // trigger on push of button
    gpio_config_t io_conf = {
        .pin_bit_mask = (1ULL << BUTTON_PIN),
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_ENABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_NEGEDGE,
    };
    gpio_config(&io_conf);

    gpio_install_isr_service(0);
    gpio_isr_handler_add(BUTTON_PIN, button_isr, NULL);
    //do nothing in loop forever
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

A piece of code you want to run in reaction to an event

Set event to be on the negative edge…

Trigger this function

https://components.espressif.com/components/espp/interrupt/versions/1.0.22/examples/example?language=

Timer Interrupt

```c
static bool IRAM_ATTR timer_isr(gptimer_handle_t timer,
                                const gptimer_alarm_event_data_t *edata,
                                void *arg) {
    printf("Timer fired!\n");
    return false;
}
void app_main(void) {
    gptimer_handle_t timer = NULL;
    gptimer_config_t timer_config = {
        .clk_src = GPTIMER_CLK_SRC_DEFAULT,
        .direction = GPTIMER_COUNT_UP,
        .resolution_hz = 1000000,
    };
    gptimer_new_timer(&timer_config, &timer);

    gptimer_alarm_config_t alarm_config = {
        .alarm_count = 500000,
        .reload_count = 0,
        .flags.auto_reload_on_alarm = true,
    };
    gptimer_set_alarm_action(timer, &alarm_config);

    gptimer_event_callbacks_t cbs = {
        .on_alarm = timer_isr,
    };
    gptimer_register_event_callbacks(timer, &cbs, NULL);

    gptimer_enable(timer);
    gptimer_start(timer);
    //do nothing in loop forever
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

A piece of code you want to run in reaction to an event

A hardware timer

An alarm listening to the timer

Setting things up so that when alarm happens, it triggers the timer_isr

Interrupts…similar (kinda) to FastAPI

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def home():
    return {"message": "hello"}

@app.get("/items/{item_id}")
def get_item(item_id: int):
    return {"item_id": item_id, "name": "widget"}

@app.post("/items")
def create_item(name: str, price: float):
    return {"name": name, "price": price, "status"
```

"when GET at / happens, run this code"

"when a GET at /items happens do this"

"when a GET at /items happens do this"

Similarities

- Both Interrupts and FastAPI are event-driving forms of programming
- Both rely on some grand deity-like entity in the background to actually be running and checking things and executing
- Interrupts happen at the hardware level
- FastAPI happens at the program (Python) level

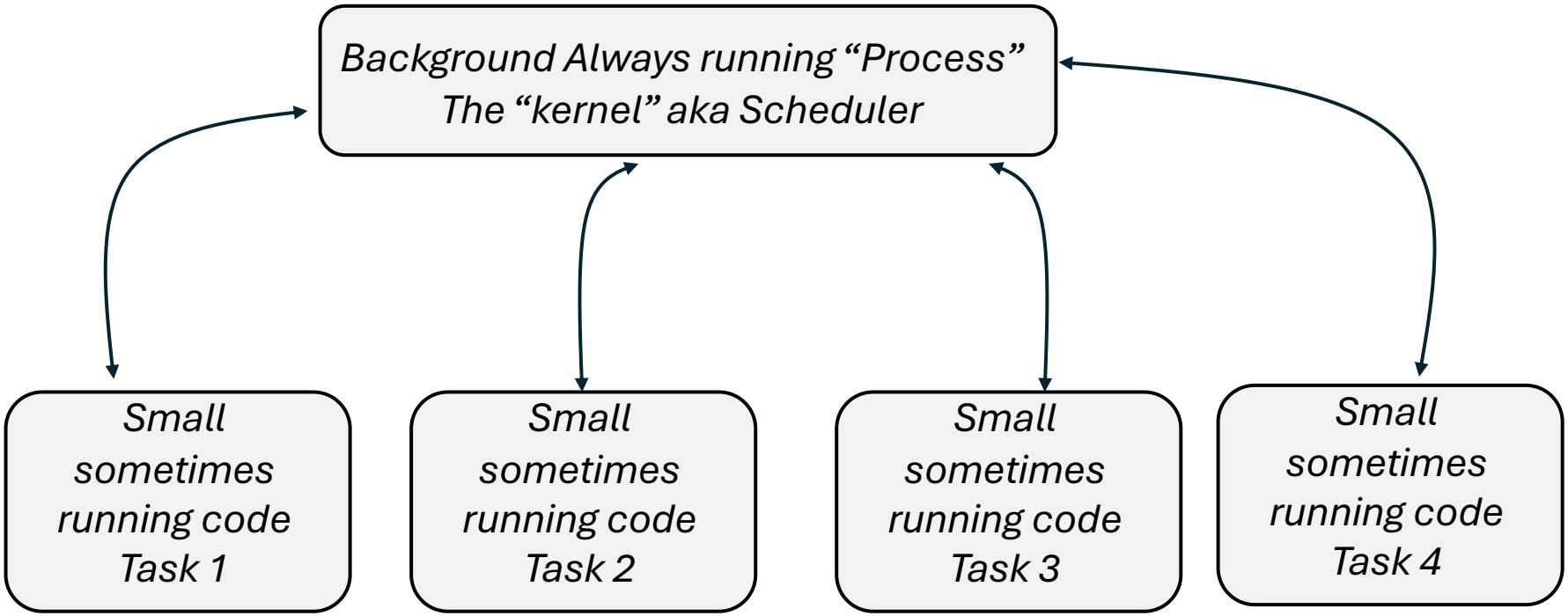
Solution 2: RTOS

- The Espressif comes with something that is actually much closer to the how FastAPI works (more so at the software level)

- It is known as an RTOS

Solution 2: RTOS

- **R**eal **T**ime **O**perating **S**ystem is an operating system that is particular sensitive to timing!



The Scheduler

- The user spins up various ***tasks*** that run “at the same time”
- The Scheduler, the only piece of software that is actually always running, manages all the ***tasks*** using a priority queue of sorts.
- Tasks are effectively the same as threads

Espressif (ESP32C3) uses FreeRTOS

- By far the dominant RTOS in the embedded world.
- Relatively lightweight (and small).
- Very well documented.
- Mature.
- Not super heavy duty (no file system, drivers, user accounts, etc… Things like that)

Alternatives to FreeRTOS

- FreeRTOS that Espressif uses is basically just a glorified task manager loop (this is one reason why it is
- If you want more features of an actual OS, there is the **Zephyr framework**…feel free to read up on that if you hate yourself and don't like being productive.
- I would strongly, strongly encourage you to not use Zephyr for 6.9000 project. In fact we forbid it.

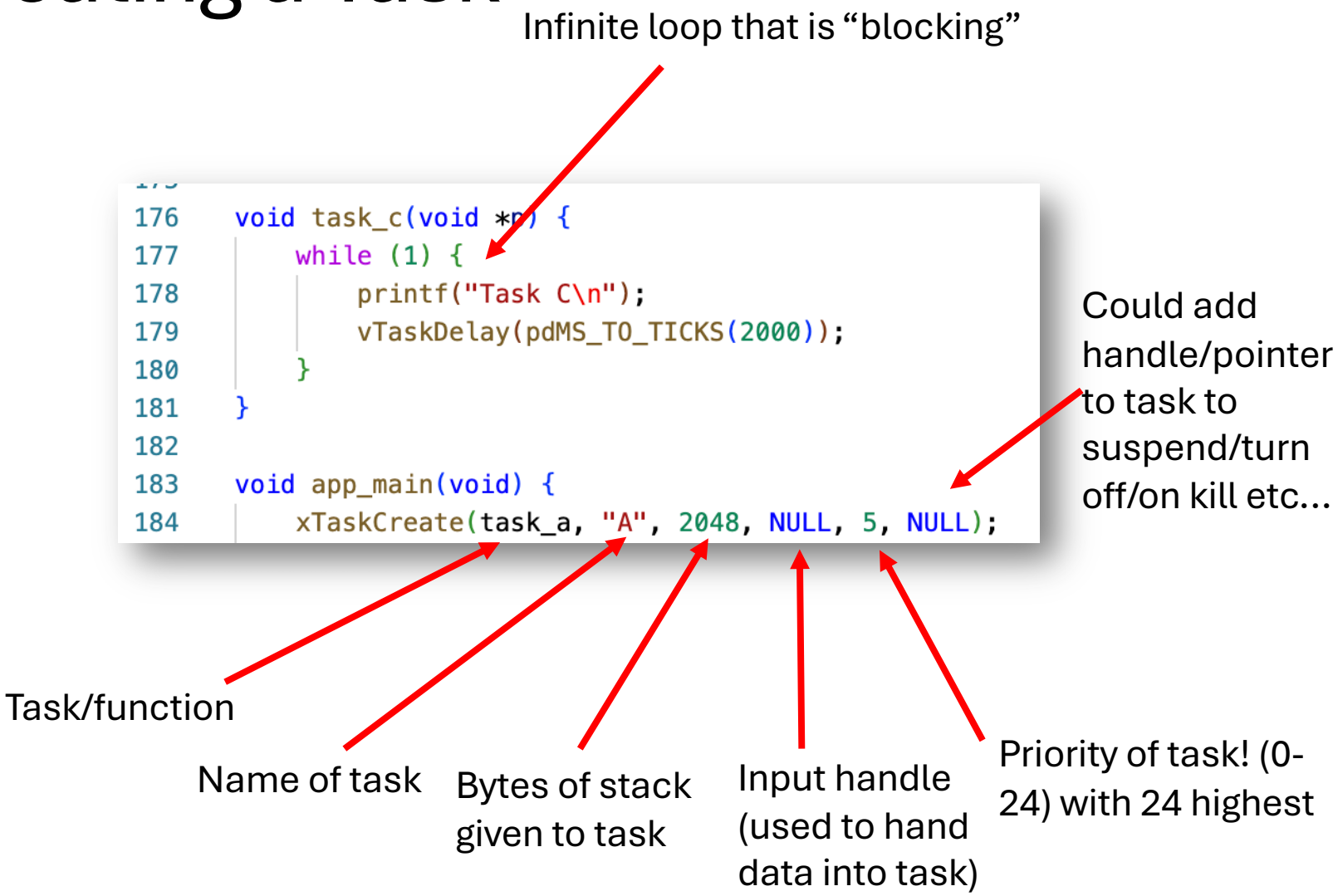
https://www.zephyrproject.org

Setting up Tasks

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void task_a(void *p) {
    while (1) {
        printf("Task A\n");
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void task_b(void *p) {
    while (1) {
        printf("Task B HIHIHIHIHIHIH\n");
        vTaskDelay(pdMS_TO_TICKS(1500));
    }
}

void task_c(void *p) {
    while (1) {
        printf("Task C HEY HEY\n");
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

void app_main(void) {
    xTaskCreate(task_a, "A", 2048, NULL, 5, NULL);
    xTaskCreate(task_b, "B", 2048, NULL, 5, NULL);
    xTaskCreate(task_c, "C", 2048, NULL, 5, NULL);
}
```

- Can have multiple pieces of code that running “at the same time (wink)”
- In reality, the scheduler is hopping between them, giving each task some CPU cycles to run before moving onto the next one.
- Does so fast enough to give the illusion of “at the same time”

Creating a Task



Running that code...

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void task_a(void *p) {
    while (1) {
        printf("Task A\n");
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void task_b(void *p) {
    while (1) {
        printf("Task B HIHIHIHIHIHIH\n");
        vTaskDelay(pdMS_TO_TICKS(1500));
    }
}

void task_c(void *p) {
    while (1) {
        printf("Task C HEY HEY\n");
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

void app_main(void) {
    xTaskCreate(task_a, "A", 2048, NULL, 5, NULL);
    xTaskCreate(task_b, "B", 2048, NULL, 5, NULL);
    xTaskCreate(task_c, "C", 2048, NULL, 5, NULL);
}
```

```
I (258) main_task: Started on CPU0
I (268) main_task: Calling app_main()
Task A
Task B HIHIHIHIHIHIH
Task C HEY HEY
I (268) main_task: Returned from app_main()
Task A
Task A
Task B HIHIHIHIHIHIH
Task A
Task C HEY HEY
Task A
Task A
Task B HIHIHIHIHIHIH
Task A
Task A
Task C HEY HEY
Task A
Task B HIHIHIHIHIHIH
Task A
Task A
Task A
Task C HEY HEY
Task B HIHIHIHIHIHIH
Task A
Task A
Task A
Task B HIHIHIHIHIHIH
Task A
Task C HEY HEY
Task A
Task A
Task B HIHIHIHIHIHIH
Task A
Task A
Task C HEY HEY
Task A
```

Running that code…

```c
void task_a(void *p) {
    while (1) {
        printf("Task A\n");
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void task_b(void *p) {
    while (1) {
        printf("Task B HIHIHIHIHIHIH\n");
        vTaskDelay(pdMS_TO_TICKS(1500));
    }
}

void task_c(void *p) {
    while (1) {
        printf("Task C HEY HEY\n");
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

- Each task that is running should have some calls to:
  - vTaskDelay
  - Or other assorted ”blocking” calls
- These “yield” control of a task back to the scheduler

What happens if...

```c
void task_a(void *p) {
    while (1) {
        printf("Task A\n");
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void task_b(void *p) {
    while (1) {
        printf("Task B HIHIHIHIHIHIH\n");

    }
}

void task_c(void *p) {
    while (1) {
        printf("Task C HEY HEY\n");
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

```
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
```

task_b never releases control

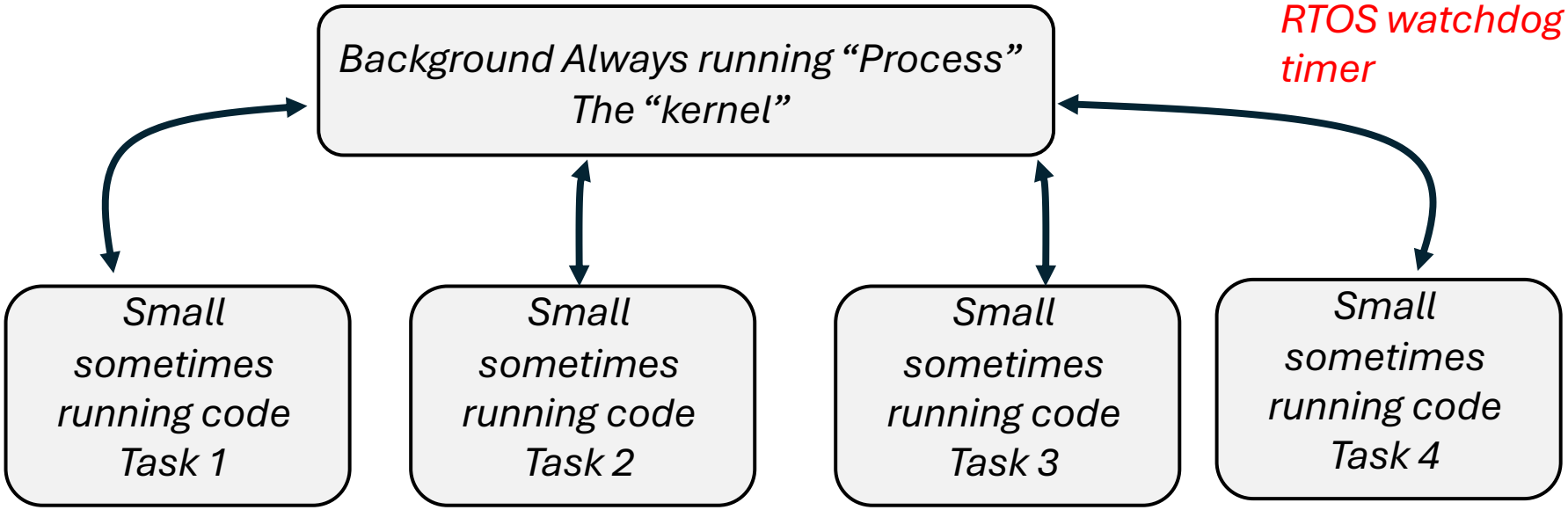
- After a while of that printing mess…

```
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HE (10268) task_wdt: Task watchdog got triggered. The following tasks/users did not reset the watchdog in time:
E (10268) task_wdt:  - IDLE (CPU 0)
E (10268) task_wdt: Tasks currently running:
E (10268) task_wdt: CPU 0: B
E (10268) task_wdt: Print CPU 0 (current core) registers
Core  0 register dump:
MEPC    : 0x42005064  RA      : 0x420061f4  SP      : 0x3fc91330  GP      : 0x3fc8c400
--- 0x42005064: uart_ll_get_txfifo_len at /Users/jodalyst/esp/esp-idf/components/hal/esp32c3/include/hal/uart_ll.h:417
--- (inlined by) uart_tx_char at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:190
--- 0x420061f4: uart_write at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:237
TP      : 0x3fc91490  T0      : 0x00000000  T1      : 0x00000000  T2      : 0x00000000
S0/FP   : 0x00000008  S1      : 0x00000049  A0      : 0x00000000  A1      : 0x00000049
A2      : 0x00000004  A3      : 0x60000000  A4      : 0x00000001  A5      : 0xe07f8000
A6      : 0x42006186  A7      : 0x00000000  S2      : 0x00000000  S3      : 0x00000015
--- 0x42006186: uart_write at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:227
S4      : 0x3fc8e72c  S5      : 0x4200504c  S6      : 0x00000000  S7      : 0x00000000
--- 0x4200504c: uart_tx_char at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:186
S8      : 0x00000001  S9      : 0x00000000  S10     : 0x00000000  S11     : 0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      : 0x00000000
MSTATUS : 0x00001889  MTVEC   : 0x40380001  MCAUSE  : 0xdeadc0de  MTVAL   : 0xdeadc0de
--- 0x40380001: _vector_table at /Users/jodalyst/esp/esp-idf/components/riscv/vectors_intc.S:54
MHARTID : 0x00000000
Please enable CONFIG_ESP_SYSTEM_USE_FRAME_POINTER option to have a full backtrace.
IHIHIHIHIHIH
Task B HIHIHIHIHIHIH
```

RTOS Task Watchdog

- When a task releases control back to the scheduler/kernel, occasionally the scheduler gets a chance to feed the watchdog
- Stay in a task too long…you'll starve the watchdog and it'll freak out

I believe 5 seconds is the RTOS watchdog timer



So make sure…

- You aren’t waiting on external signals only in your tasks!

- Add some calls to `vTaskDelay` since that is one way to “pass” control back to the scheduler

You were already in an RTOS…

- The `app_main` function in every piece of code you've written for the class so far is actually a task running on your core already.

- And if you were running Wifi in a lab/assignment, that had its *own* task that got spun up behind the scenes doing *a lot* of stuff.

You can see what's going on

- In some code from week 5 I had….

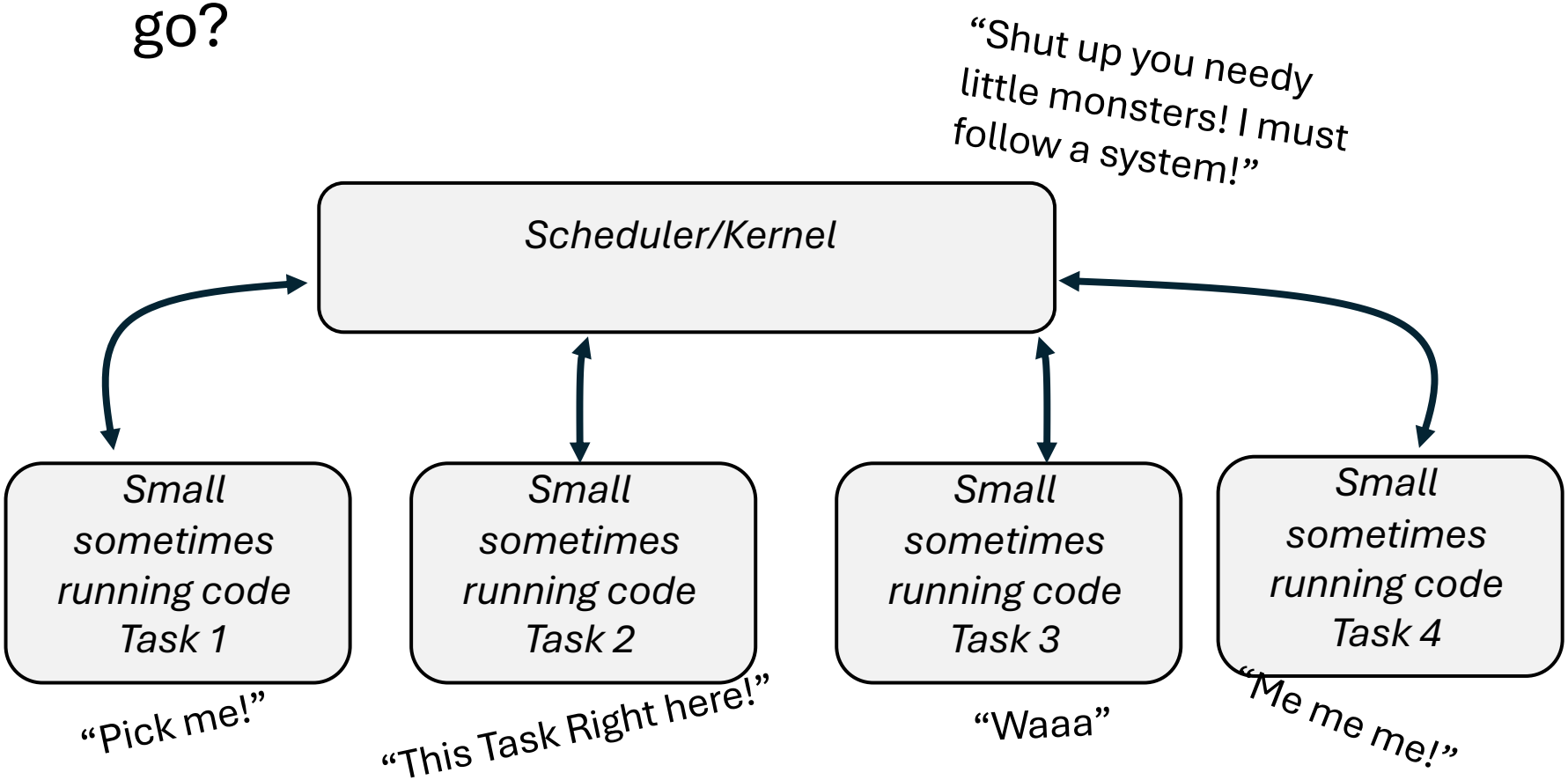
```c
void print_tasks(void) {
    char buf[1024];
    vTaskList(buf);
    printf("Name\t\tState\tPrio\tStack\tNum\n");
    printf("%s\n", buf);
}
```

+ some idf.py menuconfig action…

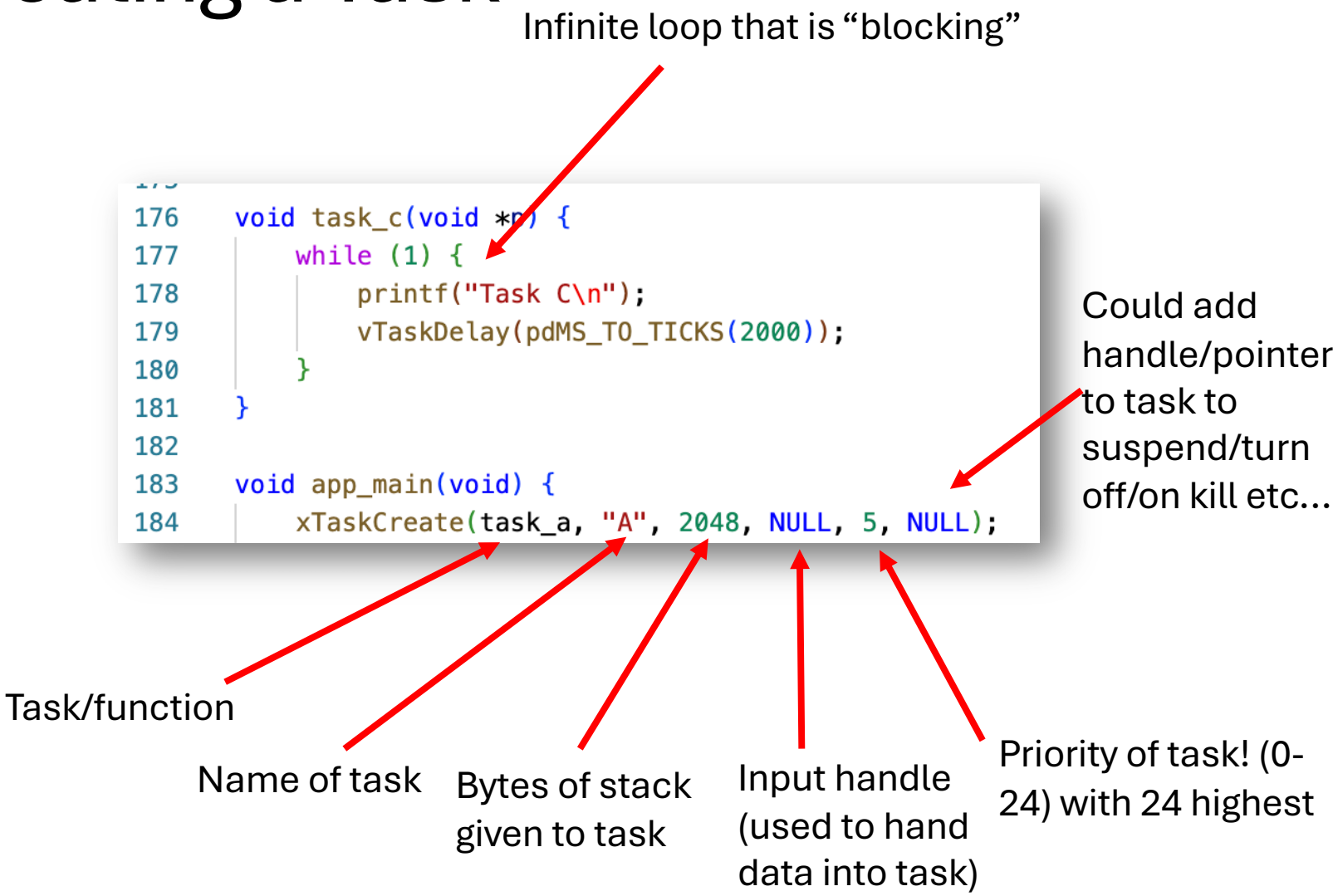
```
Name            State   Prio    Stack   Num
main            X       1       916     2
IDLE            R       0       1308    3
tiT             B       18      2692    5
sys_evt         B       20      944     6
Tmr Svc         B       1       1780    4
wifi            B       23      4116    7
esp_timer       S       22      3652    1
```

RTOS Priority

- How does the scheduler decide who gets to go?

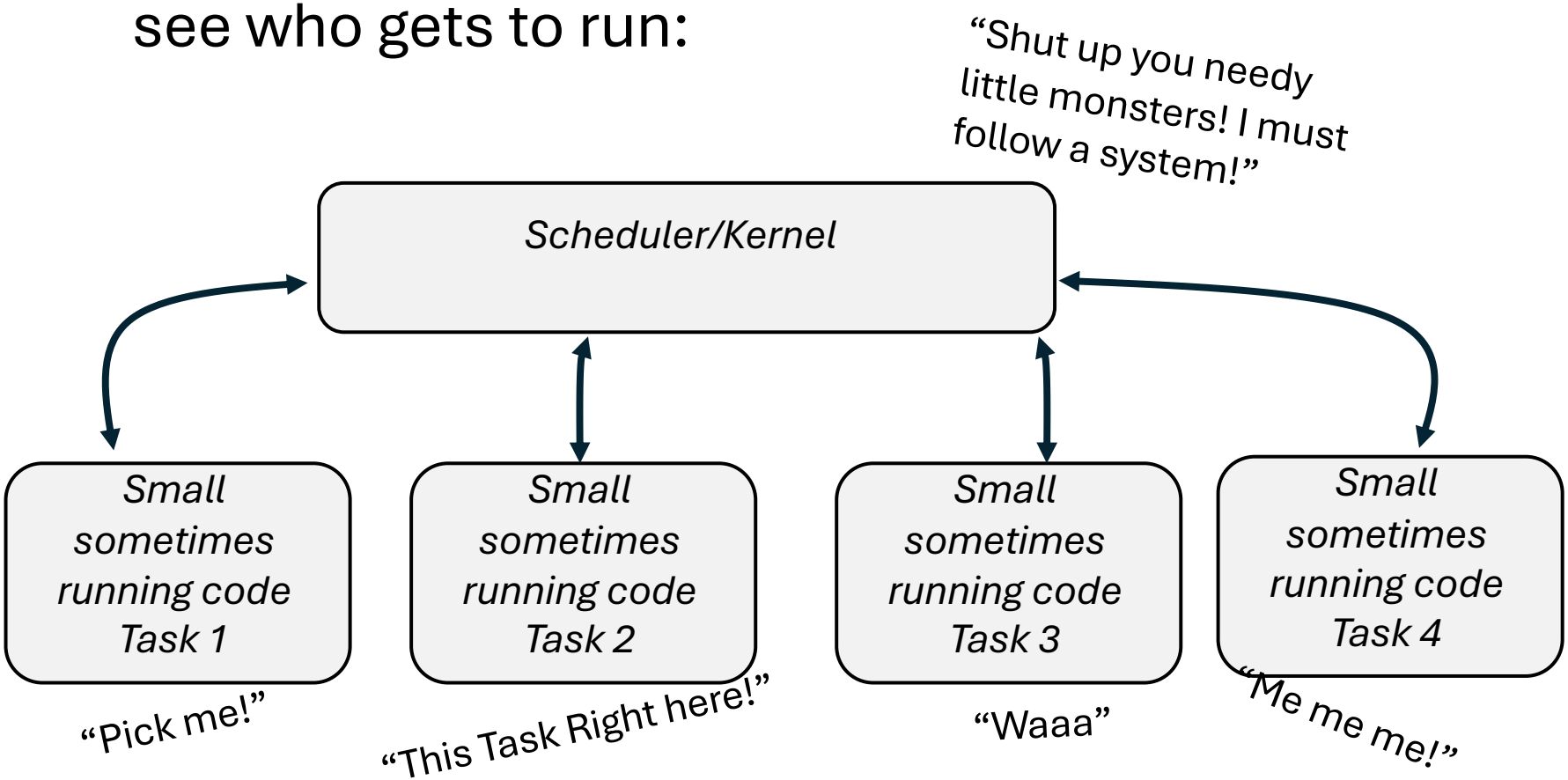


Creating a Task



RTOS Priority

- Every 10 ms (a “tick”) the Scheduler looks to see who gets to run:



Priority Queue

- The scheduler keeps track of which task wants to be run as well as the priority of all the tasks and picks a winner to give some CPU to.
- Higher priority always gets chosen
- Equal priority get round-robinned

Name	State	Prio	Stack	Num
main	X	1	916	2
IDLE	R	0	1308	3
tiT	B	18	2692	5
sys_evt	B	20	944	6
Tmr Svc	B	1	1780	4
wifi	B	23	4116	7
esp_timer	S	22	3652	1

Some Tasks (range from 0 to 24)

- Idle task has priority of 0 (lowest priority)
- app_main has priority 1
- esp_timer has priority 22
- Bluetooth and Wifi tasks have priority 23

Name	State	Prio	Stack	Num
main	X	1	916	2
IDLE	R	0	1308	3
tiT	B	18	2692	5
sys_evt	B	20	944	6
Tmr Svc	B	1	1780	4
wifi	B	23	4116	7
esp_timer	S	22	3652	1

Recall this...

```c
void task_a(void *p) {
    while (1) {
        printf("Task A\n");
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void task_b(void *p) {
    while (1) {
        printf("Task B HIHIHIHIHIHIH\n");

    }
}

void task_c(void *p) {
    while (1) {
        printf("Task C HEY HEY\n");
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

All at priority 5

```
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
```

task_b never releases control voluntarily

- After a while of that printing mess…

```
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HIHIHIHIHIHIH
Task B HE (10268) task_wdt: Task watchdog got triggered. The following tasks/users did not reset the watchdog in time:
E (10268) task_wdt:  - IDLE (CPU 0)
E (10268) task_wdt: Tasks currently running:
E (10268) task_wdt: CPU 0: B
E (10268) task_wdt: Print CPU 0 (current core) registers
Core  0 register dump:
MEPC    : 0x42005064  RA      : 0x420061f4  SP      : 0x3fc91330  GP      : 0x3fc8c400
--- 0x42005064: uart_ll_get_txfifo_len at /Users/jodalyst/esp/esp-idf/components/hal/esp32c3/include/hal/uart_ll.h:417
--- (inlined by) uart_tx_char at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:190
--- 0x420061f4: uart_write at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:237
TP      : 0x3fc91490  T0      : 0x00000000  T1      : 0x00000000  T2      : 0x00000000
S0/FP   : 0x00000008  S1      : 0x00000049  A0      : 0x00000000  A1      : 0x00000049
A2      : 0x00000004  A3      : 0x60000000  A4      : 0x00000001  A5      : 0xe07f8000
A6      : 0x42006186  A7      : 0x00000000  S2      : 0x00000000  S3      : 0x00000015
--- 0x42006186: uart_write at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:227
S4      : 0x3fc8e72c  S5      : 0x4200504c  S6      : 0x00000000  S7      : 0x00000000
--- 0x4200504c: uart_tx_char at /Users/jodalyst/esp/esp-idf/components/esp_driver_uart/src/uart_vfs.c:186
S8      : 0x00000001  S9      : 0x00000000  S10     : 0x00000000  S11     : 0x00000000
T3      : 0x00000000  T4      : 0x00000000  T5      : 0x00000000  T6      : 0x00000000
MSTATUS : 0x00001889  MTVEC   : 0x40380001  MCAUSE  : 0xdeadc0de  MTVAL   : 0xdeadc0de
--- 0x40380001: _vector_table at /Users/jodalyst/esp/esp-idf/components/riscv/vectors_intc.S:54
MHARTID : 0x00000000
Please enable CONFIG_ESP_SYSTEM_USE_FRAME_POINTER option to have a full backtrace.
IHIHIHIHIHIH
Task B HIHIHIHIHIHIH
```

Priority and Failing to Yield

Name	State	Prio	Stack	Num
main	X	1	916	2
IDLE	R	0	1308	3
tiT	B	18	2692	5
sys_evt	B	20	944	6
Tmr Svc	B	1	1780	4
wifi	B	23	4116	7
esp_timer	S	22	3652	1

- Failing to block/yield control to scheduler from a task can impact tasks at equal or lower priority
- Higher priority ones are not impacted
- Equal priority tasks will get potentially malnourished
- Lower priority tasks will get starved
- IDLE is the “housekeeping” task and does things like feed the watchdog…if that is never given a chance to run, issues will arise

Some Tasks (range from 0 to 24)

- Idle task has priority of 0 (lowest priority)
- app_main has priority 1
- esp_timer has priority 22
- Bluetooth and Wifi tasks have priority 23

- For things you write, try to aim in the 5-15 range maybe

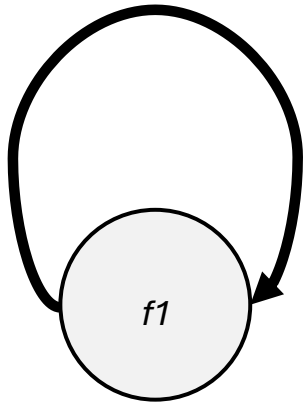
Name	State	Prio	Stack	Num
main	X	1	916	2
IDLE	R	0	1308	3
tiT	B	18	2692	5
sys_evt	B	20	944	6
Tmr Svc	B	1	1780	4
wifi	B	23	4116	7
esp_timer	S	22	3652	1

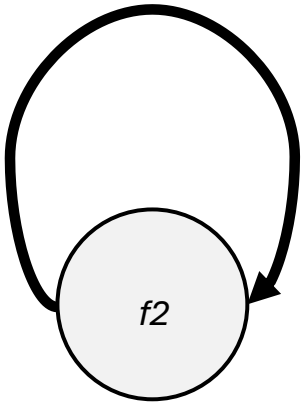
DO NOT

- You might think your tasks are very demanding and therefore deserve highest priority, but your demands are likely nothing compared to those of WiFi and BLE
- So don’t use 24
- I think general rec is to keep things below 19 if you’ve got things like WiFi or BLE going.

Another Issue: Talking Between Tasks in an RTOS

- These two things are pretty isolated it turns out…how to have them communicate?





Collect data every minute. Put in nice spot. When you've got 20 datas collected, f1 tells f2 that it needs to post the data

Listening for some message from f1 that says, "I've gotten all the data, grab it and send it up!"

Signaling Between Tasks

- The individual tasks have completely separate stacks (for safety!)
- They need to use other common resources to communicate between each other

```
void app_main(void) {
    xTaskCreate(task_a, "A", 2048, NULL, 5, NULL);
    xTaskCreate(task_b, "B", 2048, NULL, 5, NULL);
    xTaskCreate(task_c, "C", 2048, NULL, 5, NULL);
}
```

Shared Variables

```
127  int shared_value = 0;
128  bool new_data = false;
129
130  void producer(void *p) {
131      int count = 0;
132      while (1) {
133          shared_value = count++;
134          new_data = true;
135          vTaskDelay(pdMS_TO_TICKS(1000));
136      }
137  }
138
139  void consumer(void *p) {
140      while (1) {
141          if (new_data) {
142              printf("Got: %d\n", shared_value);
143              new_data = false;
144          }
145          vTaskDelay(pdMS_TO_TICKS(10));
146      }
147  }
148
149  void app_main(void) {
150      xTaskCreate(producer, "prod", 2048, NULL, 5,
151      xTaskCreate(consumer, "cons", 2048, NULL, 5,
152  }
```

- The `consumer` is checking the `new_data` variable to see if `producer` changed it.
- If so it reads `shared_value` and then sets `new_data` back to false

WILL NOT WORK

Shared Variables

```
127  volatile int shared_value = 0;
128  volatile bool new_data = false;
129
130  void producer(void *p) {
131      int count = 0;
132      while (1) {
133          shared_value = count++;
134          new_data = true;
135          vTaskDelay(pdMS_TO_TICKS(1000));
136      }
137  }
138
139  void consumer(void *p) {
140      while (1) {
141          if (new_data) {
142              printf("Got: %d\n", shared_value);
143              new_data = false;
144          }
145          vTaskDelay(pdMS_TO_TICKS(10));
146      }
147  }
148
149  void app_main(void) {
150      xTaskCreate(producer, "prod", 2048, NULL, 5,
151      xTaskCreate(consumer, "cons", 2048, NULL, 5,
152  }
```

- Variables in this situation must be declared as ***volatile*** to prevent compiler from simplifying them away

Volatile Variables

- Marking variables as volatile fixes the compiling-away issue
- For things much beyond one single variable, issues can arise because of the RTOS and its task switching.
- Where the RTOS ”time-slices” between tasks is not up to the individual tasks and is not predictable

Signaling Between Tasks

- Volatiles should be used very cautiously.
- Instead you have other options in RTOS:
  - Queues
  - Semaphores
  - Mutexes
  - Notifications

```
123  #include <stdio.h>
124  #include "freertos/FreeRTOS.h"
125  #include "freertos/task.h"
126
127  int shared_value = 0;
128  bool new_data = false;
129
130  void producer(void *p) {
131      int count = 0;
132      while (1) {
133          shared_value = count++;
134          new_data = true;
135          vTaskDelay(pdMS_TO_TICKS(1000));
136      }
137  }
138
139  void consumer(void *p) {
140      while (1) {
141          if (new_data) {
142              printf("Got: %d\n", shared_value);
143              new_data = false;
144          }
145          vTaskDelay(pdMS_TO_TICKS(10));
146      }
147  }
148
149  void app_main(void) {
150      xTaskCreate(producer, "prod", 2048, NULL, 5, NULL);
151      xTaskCreate(consumer, "cons", 2048, NULL, 5, NULL);
152  }
```

Semaphores and Mutexes

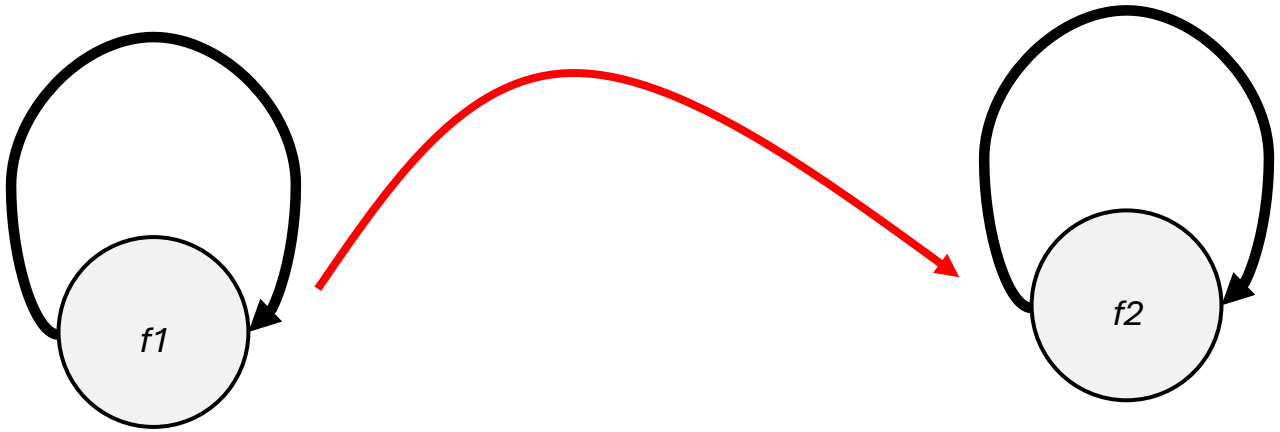
- Terminology is a little overlapped in the espressif framework but in general:
- Sempahores are unidirectional communication between tasks
- Mutexes are bidirectional communication between tasks
- Both are **thread-safe**…the RTOS manages all the volatile stuff behind the scenes for you

Semaphores

- Uni-directional: meant for signaling

Announces:
"Here is signal for you!"

Listens for that signal and reacts



Semaphore Code Example

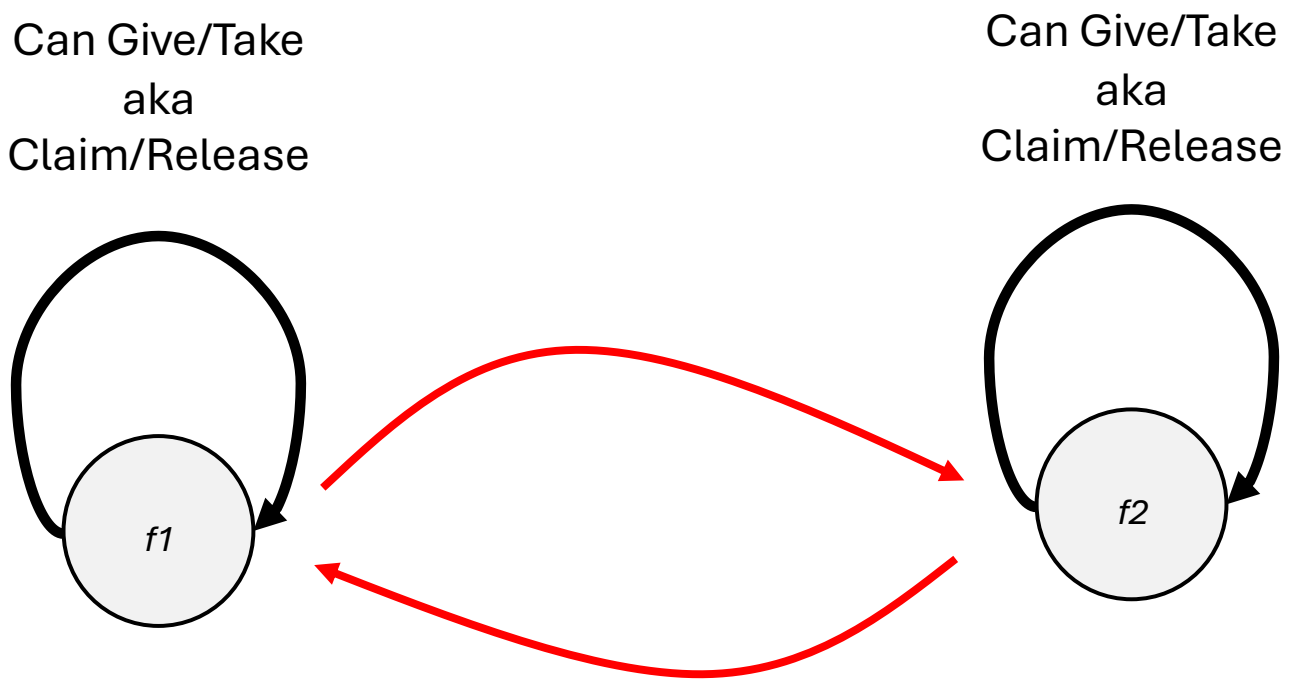
```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"

SemaphoreHandle_t sem;

void sensor_task(void *p) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(2000));
        printf("Sensor: data ready\n");
        xSemaphoreGive(sem);  // "hey other task"
    }
}

void display_task(void *p) {
    while (1) {
        xSemaphoreTake(sem, portMAX_DELAY);  // sleep until signaled
        printf("Display: showing data\n"); //Thanks! Got it
    }
}

void app_main(void) {
    sem = xSemaphoreCreateBinary();
    xTaskCreate(sensor_task, "sensor", 2048, NULL, 5, NULL);
    xTaskCreate(display_task, "display", 2048, NULL, 5, NULL);
}
```

Announces

Yields to scheduler until announcement made

Mutexes

- Bi-directional: meant for sharing (responsibly) resources



Mutex Code Example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"

SemaphoreHandle_t mutex; //same high level struct/object
int counter = 0;

void task_a(void *p) {
    while (1) {
        xSemaphoreTake(mutex, portMAX_DELAY);
        counter++;
        printf("A: counter = %d\n", counter);
        xSemaphoreGive(mutex);
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void task_b(void *p) {
    while (1) {
        xSemaphoreTake(mutex, portMAX_DELAY);
        counter += 10;
        printf("B: counter = %d\n", counter);
        xSemaphoreGive(mutex);
        vTaskDelay(pdMS_TO_TICKS(700));
    }
}

void app_main(void) {
    mutex = xSemaphoreCreateMutex(); //creates bidirectional mutex
    xTaskCreate(task_a, "A", 2048, NULL, 5, NULL);
    xTaskCreate(task_b, "B", 2048, NULL, 5, NULL);
}
```

Shared resource (doesn't have to be declared volatile)

claim resource
manipulate resource
release resource

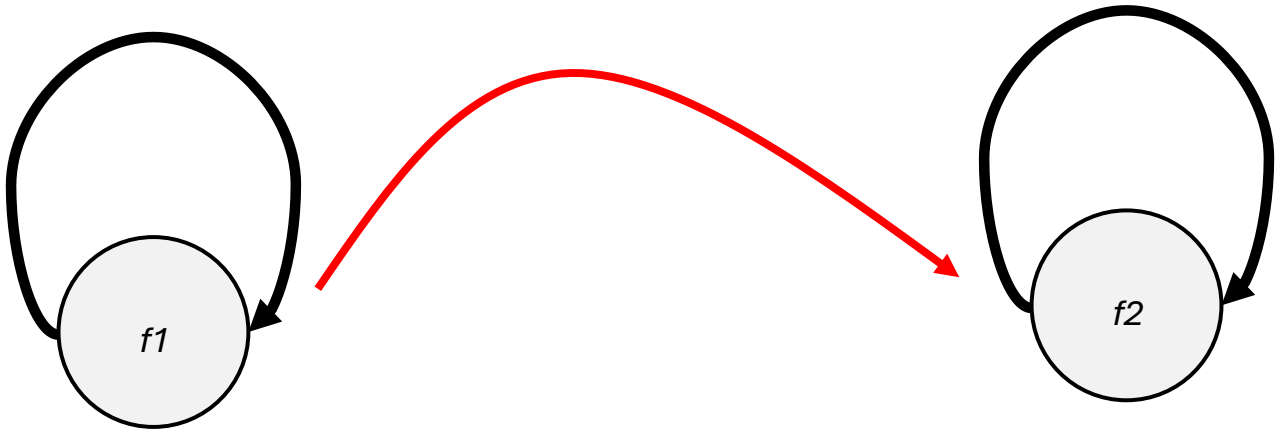
claim resource
manipulate resource
release resource

Notifications

- Uni-directional: meant for signaling…effectively implementing semaphores behind the scenes

Announces:
"Here is signal for you!"

Listens for that signal and reacts



Notifications

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

TaskHandle_t receiver_handle;

void sender(void *p) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(1000));
        xTaskNotifyGive(receiver_handle);//send notiifcation to receiver
    }
}

void receiver(void *p) {
    while (1) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        printf("I have been notified!\n");
    }
}

void app_main(void) {
    xTaskCreate(receiver, "recv", 2048, NULL, 5, &receiver_handle);
    xTaskCreate(sender, "send", 2048, NULL, 5, NULL);
}
```

Announces

Yields to scheduler until announcement made

Using these functionalities

- So good uses of semaphores?
- Good uses of Mutexes?

What about all this RTOS with Power?

- We’ve been doing all this weird stuff…how does power come into play.

- Remember esp_light_sleep, esp_deep_sleep,etc…?

RTOS Power Functionality

- Deep Sleep still pretty much turns everything off/reboots the device…not much change there.

- The RTOS has built-in functionality tied to light sleeping and some other cool functionality

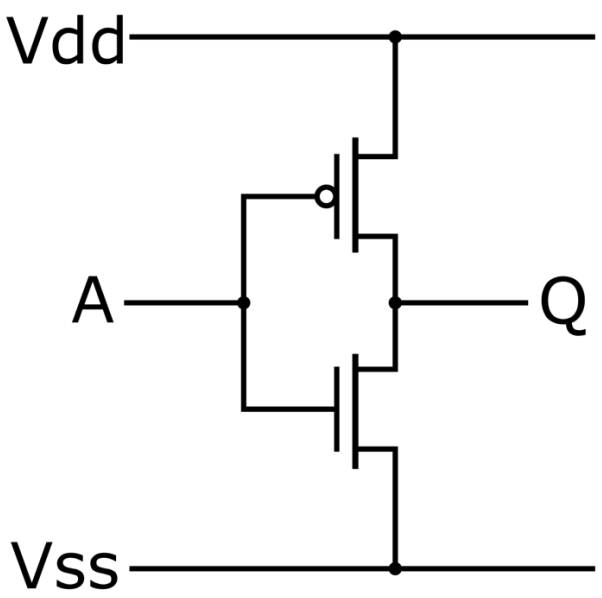
Summary: Digital Power Consumption

- P : total power consumed
- $\alpha_{0\to1}$: fraction of gates switching
- C : Capacitance of gates, busses, interconnects
- V : Operating voltage (V_{dd})
- f : frequency of operation
- I_{leak}: Leakage Current:
  - Sub-threshold leakage
  - Gate-Leakage

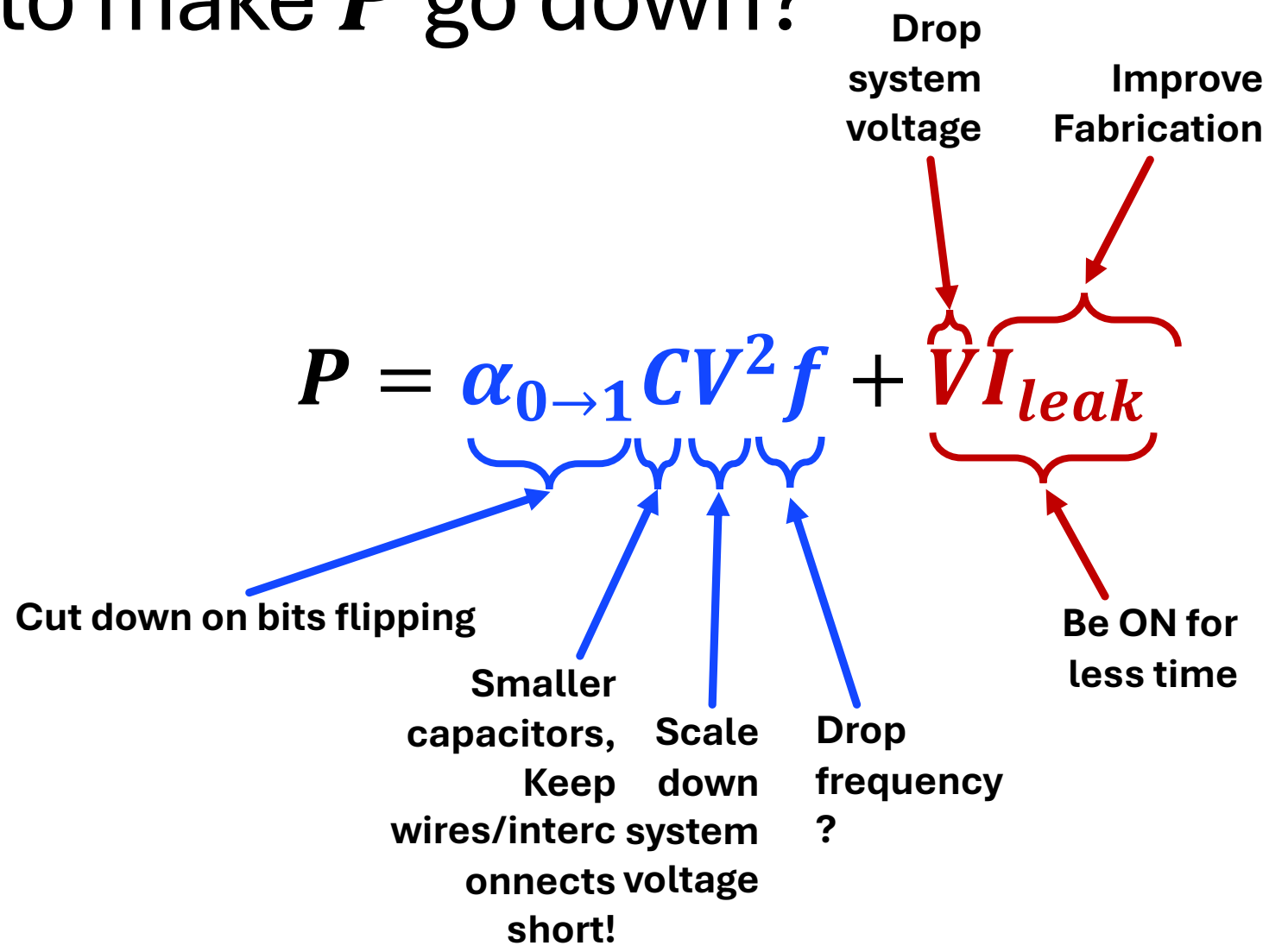
Dynamic power consumption

$$\boldsymbol{P = \alpha_{0\to1} C V^2 f + V I_{leak}}$$

Static power consumption



How to make P go down?



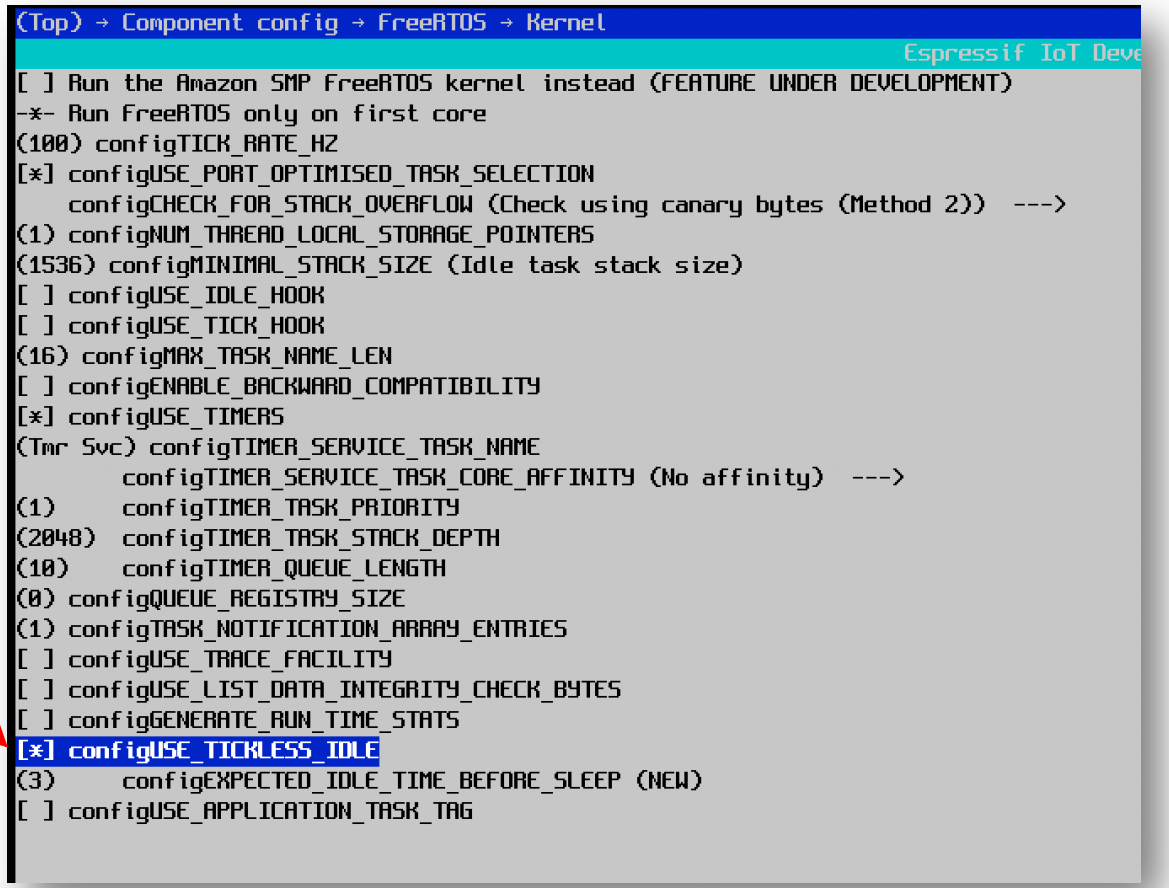
Underclocking

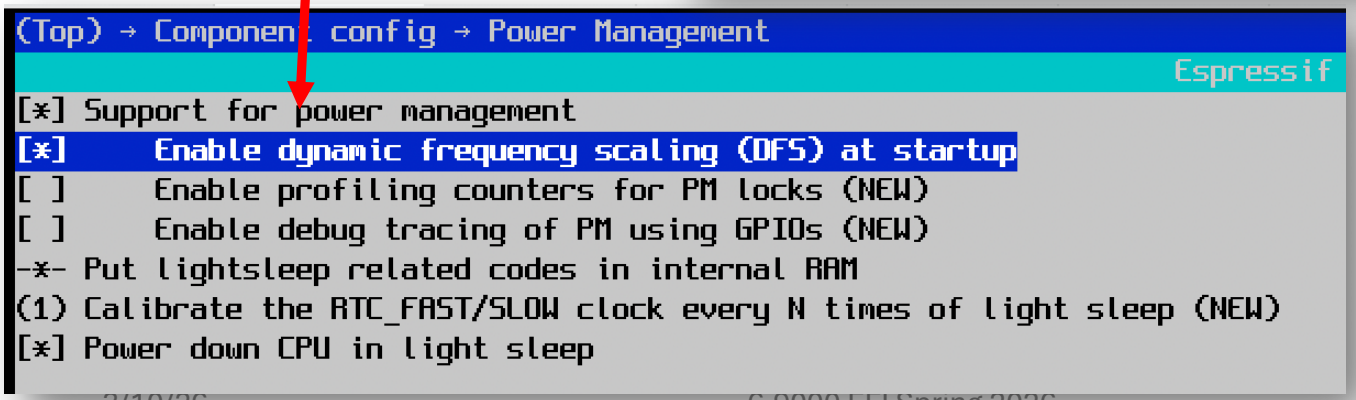
- If you have a lot of computation to get through, underclocking is rarely a good idea.
- If, however, you actually have nothing to do whatsoever, or…
- what you need to do does not need to be done very quickly (interacting with stupid slow human flesh computers)
- , simply running in place checking/counting a number is a waste of time

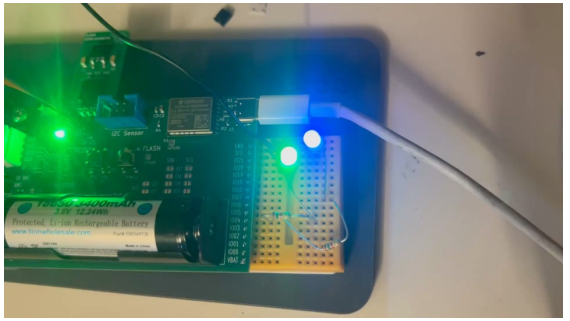
Some Settings:

Allow to go to light sleep (tickles)

Allow processor to scale its clock





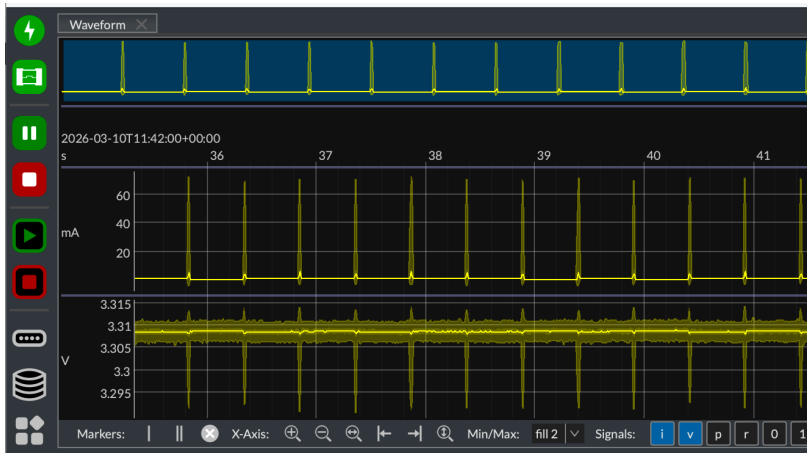


```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "esp_pm.h"
#include "esp_log.h"

#define LED_A  GPIO_NUM_4
#define LED_B  GPIO_NUM_5

void led_a_task(void *p) {
    gpio_set_direction(LED_A, GPIO_MODE_OUTPUT);
    bool state = false;
    while (1) {
        state = !state;
        gpio_hold_dis(LED_A);  //unlocks teh pin
        gpio_set_level(LED_A, state); //change level
        gpio_hold_en(LED_A); //holds the pin during the light sleep
        vTaskDelay(pdMS_TO_TICKS(500));  // blink every 500ms
    }
}

void led_b_task(void *p) {
    gpio_set_direction(LED_B, GPIO_MODE_OUTPUT);
    bool state = false;
    while (1) {
        state = !state;
        gpio_hold_dis(LED_B); //same as before
        gpio_set_level(LED_B, state); //same as above
        gpio_hold_en(LED_B); //same as it ever was
        vTaskDelay(pdMS_TO_TICKS(2000));  // blink every 2 seconds
    }
}
```

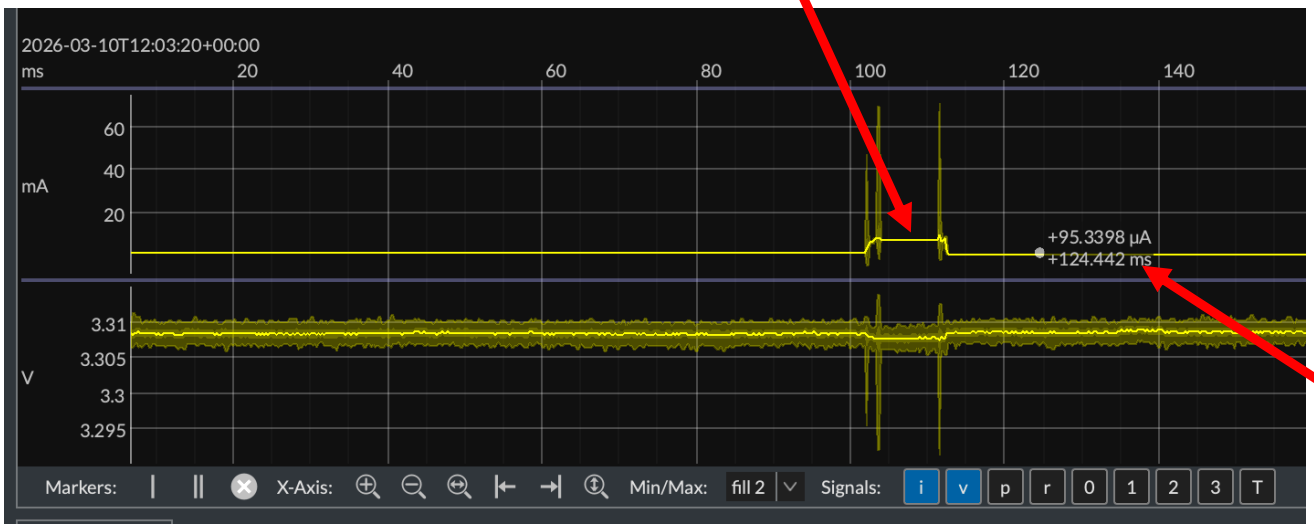
```c
esp_pm_config_t pm_config = {
    .max_freq_mhz = 160, //normaly max
    .min_freq_mhz = 10, //how ok you ar
    .light_sleep_enable = true
};
```

```c
void app_main(void) {
    // Enable DFS and automatic light sleep
    esp_pm_config_t pm_config = {
        .max_freq_mhz = 160, //normaly max freq
        .min_freq_mhz = 10, //how ok you are to scale the
        .light_sleep_enable = true
    };
    ESP_ERROR_CHECK(esp_pm_configure(&pm_config)); //

    //sping up the two tasks:
    xTaskCreate(led_a_task, "led_a", 2048, NULL, 5, NULL);
    xTaskCreate(led_b_task, "led_b", 2048, NULL, 5, NULL);
}
```

Check the Power!



*When full wake ups happen only draw **6mA** (not 18mA) because system running at lower clock*



95uA in light sleep! (not 195uA) because few things that are still on are underclocked)

Should you use the RTOS?

- The espressif toolchain is built around it, so can't avoid it
- The question is how much to embrace vs. avoid it.
- If you use it wisely and follow the docs, I think you can get very scalable and supportable code that ends up looking less spaghetti-like

Two Big Pieces

- As you design your projects your firmware will get more and more complicated…

- Process/Instructions/Behavior:
  - How to structure all your code (RTOS?)

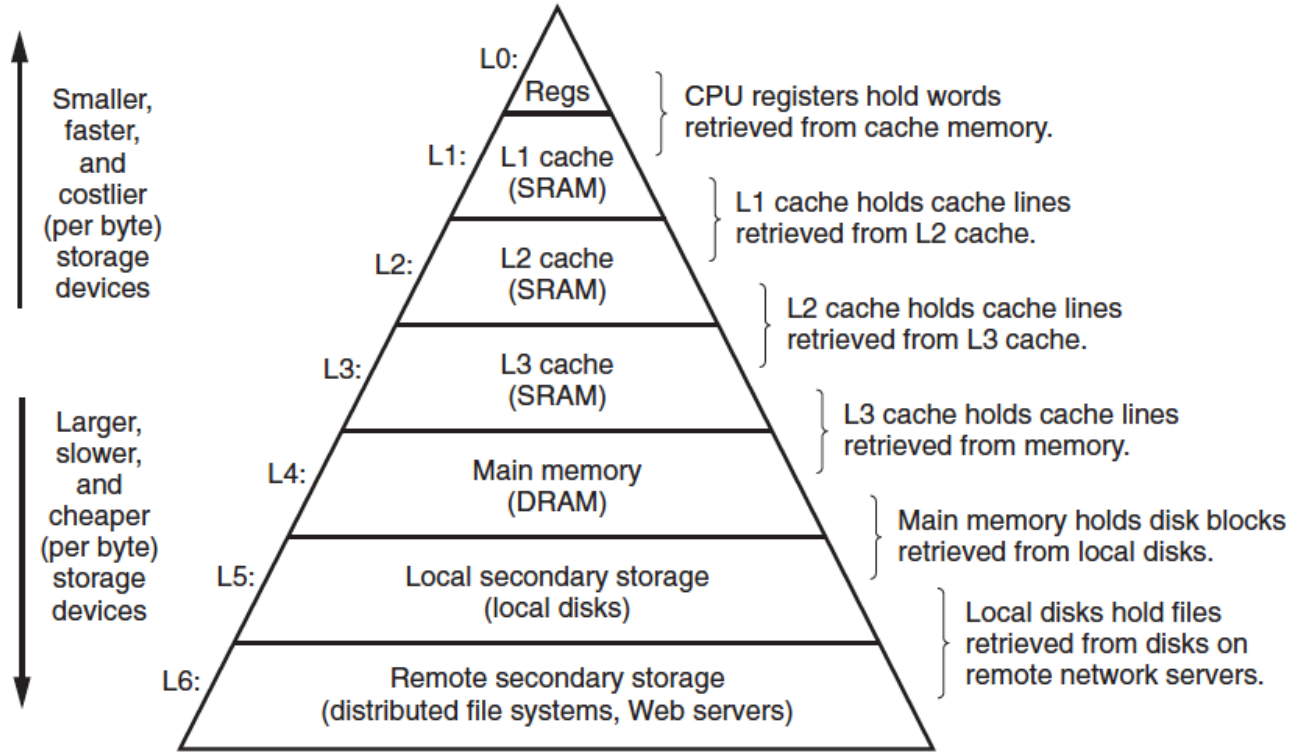
- Memory:
  - How to store all your data
  - Issues with this maybe

The Memory

- The ESP32C3 has a diverse set of memory resources.
- Using them smartly is key to making sure everything stays happy

In a Modern Computer

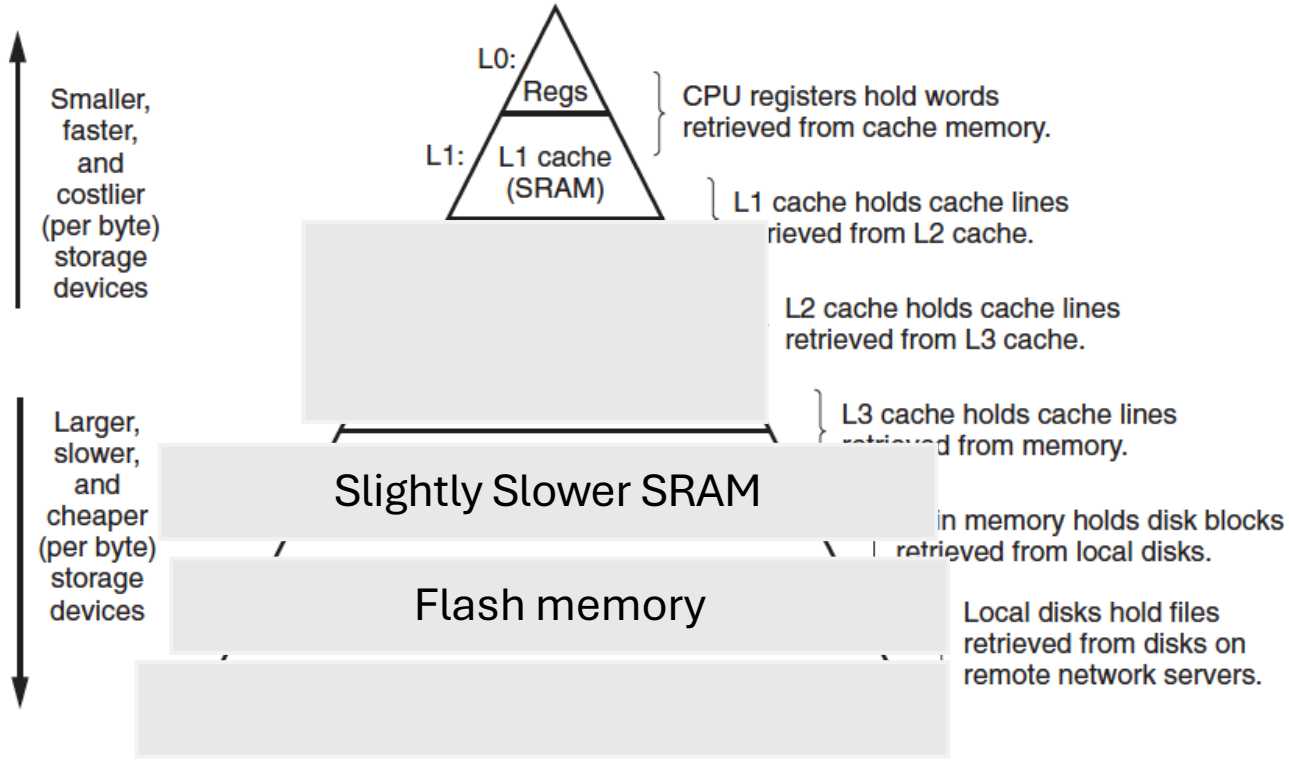
- Data storage is organized into a hierarchy



https://3000rain.medium.com/memory-hierarchy-afb83b61558c

On ESP32C3….more flattened

- Data storage is organized into a hierarchy



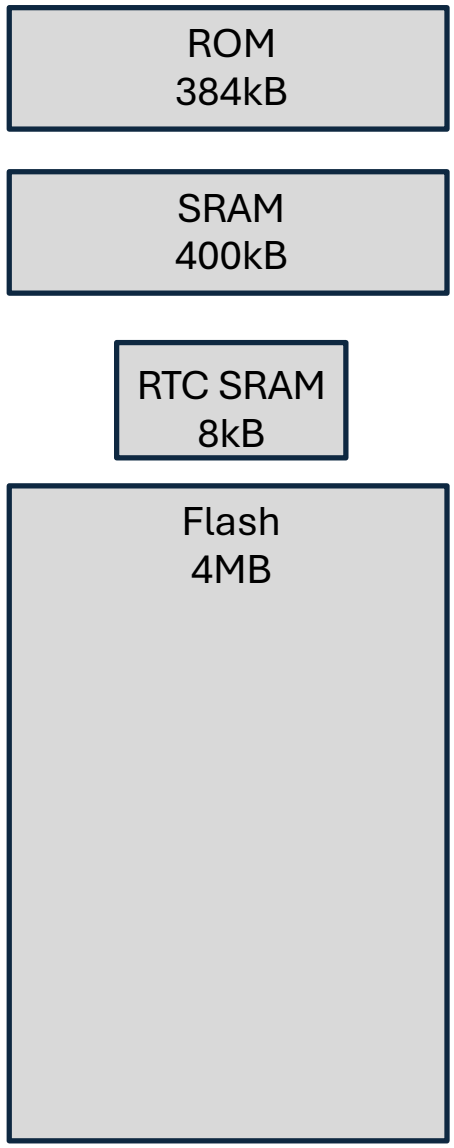
https://3000rain.medium.com/memory-hierarchy-afb83b61558c

So really on ESP32C3

- L0 is registers (RISCV32 got 32 of them)
- L1 cache is faster SRAM is (16 kB)
- L2 is rest of SRAM (~400 kB)
- L3 is the Flash memory (4MB)

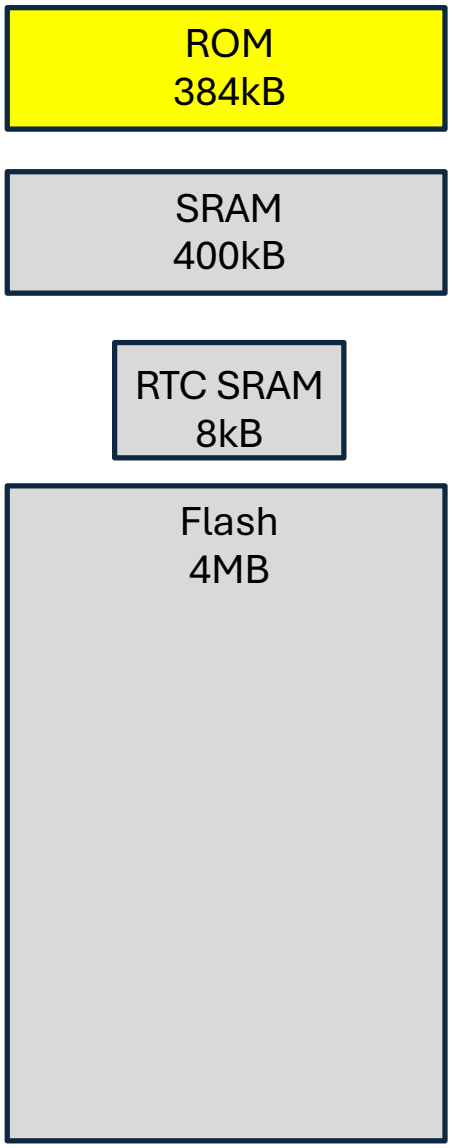
Memory on the ESP32C3

- This is the actual physical breakdown of memory is shown here



ROM

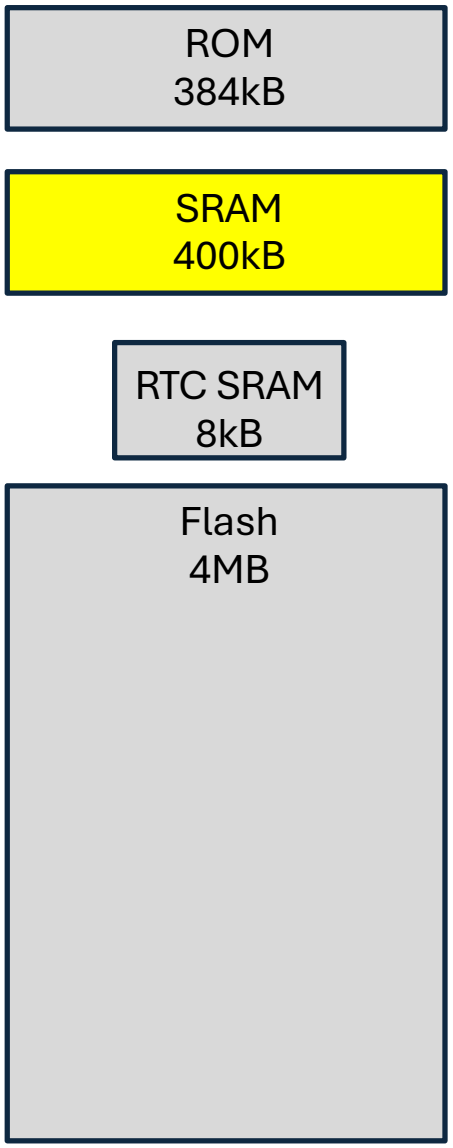
- Don’t touch this…has bootloader and stuff.
- Non-volatile, meaning it persists without power
- Not for you



SRAM

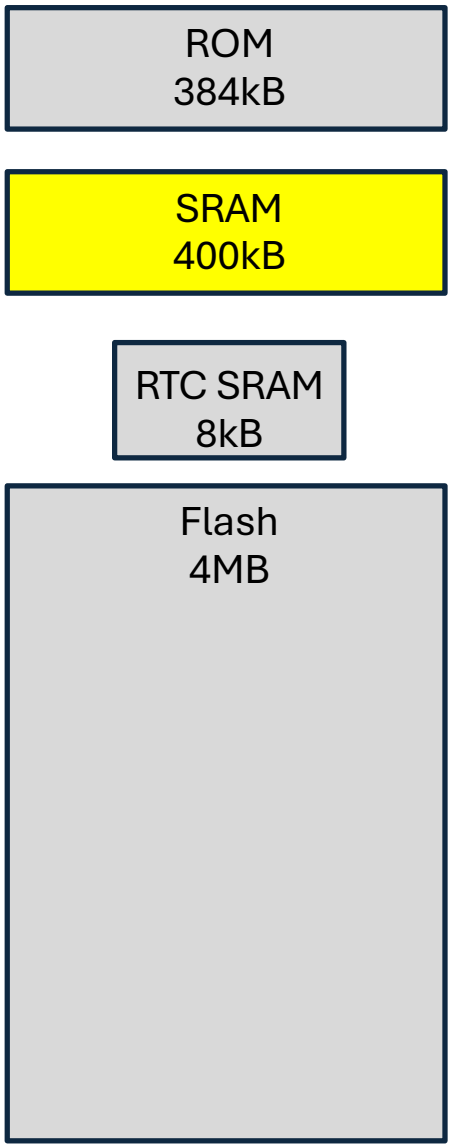
- Where *all active* relevant variables will be living at run time as well as a subset of instructions being actively run in the L1 cache

- All instructions do not live here, only a portion



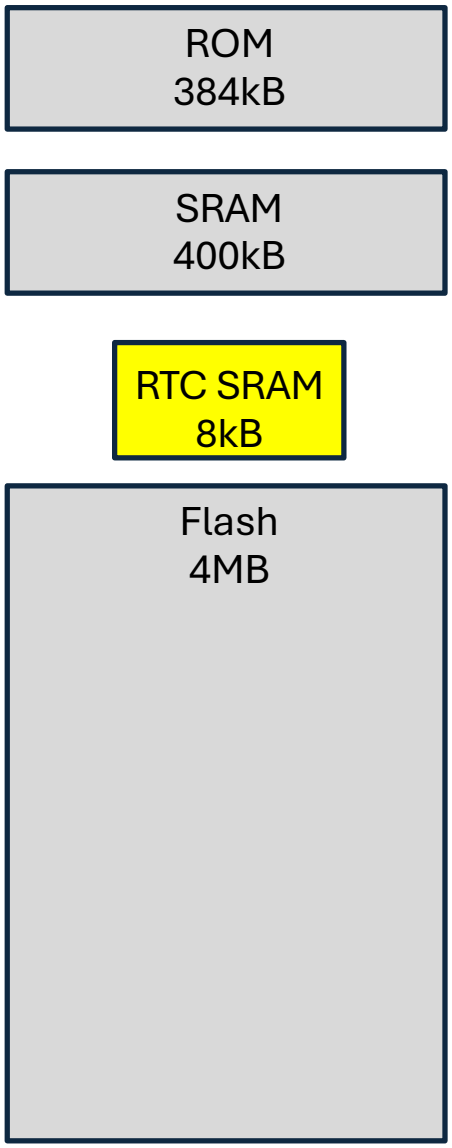
SRAM

- This SRAM is volatile meaning that if you deep sleep it gets wiped away.
- When on and in *light sleep* it stays preserved



RTC SRAM

- RTC SRAM is also SRAM so it is volatile and will fade away with no power
- But the RTC_SRAM gets to keep its power even in deep_sleep so its memory will persist in this mode



RTC SRAM

- How to interact and use it?

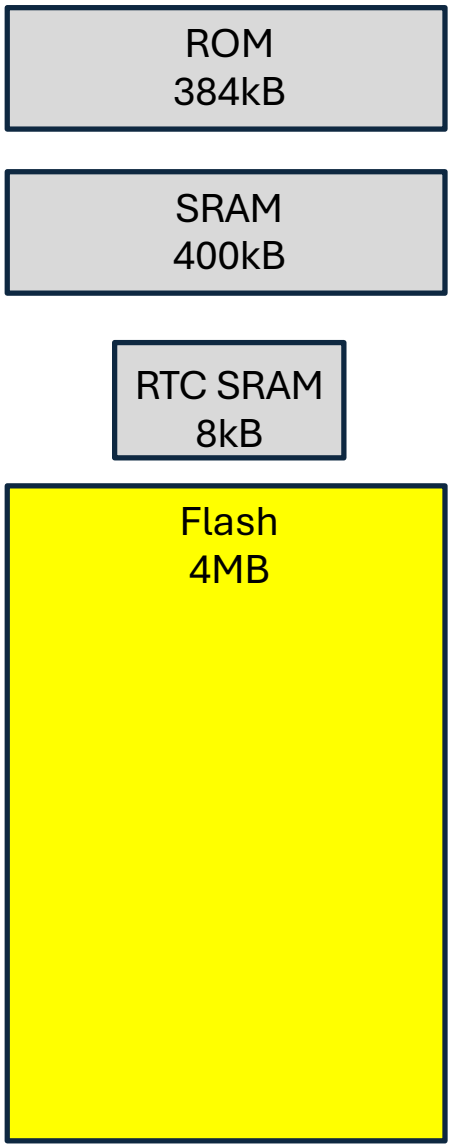
```
#include "esp_attr.h"

RTC_DATA_ATTR int joes_array[10] = {0};
//that's it...lives in RTC memory...will persist after deep sleep
```

- But remember there's only 8kB of it*

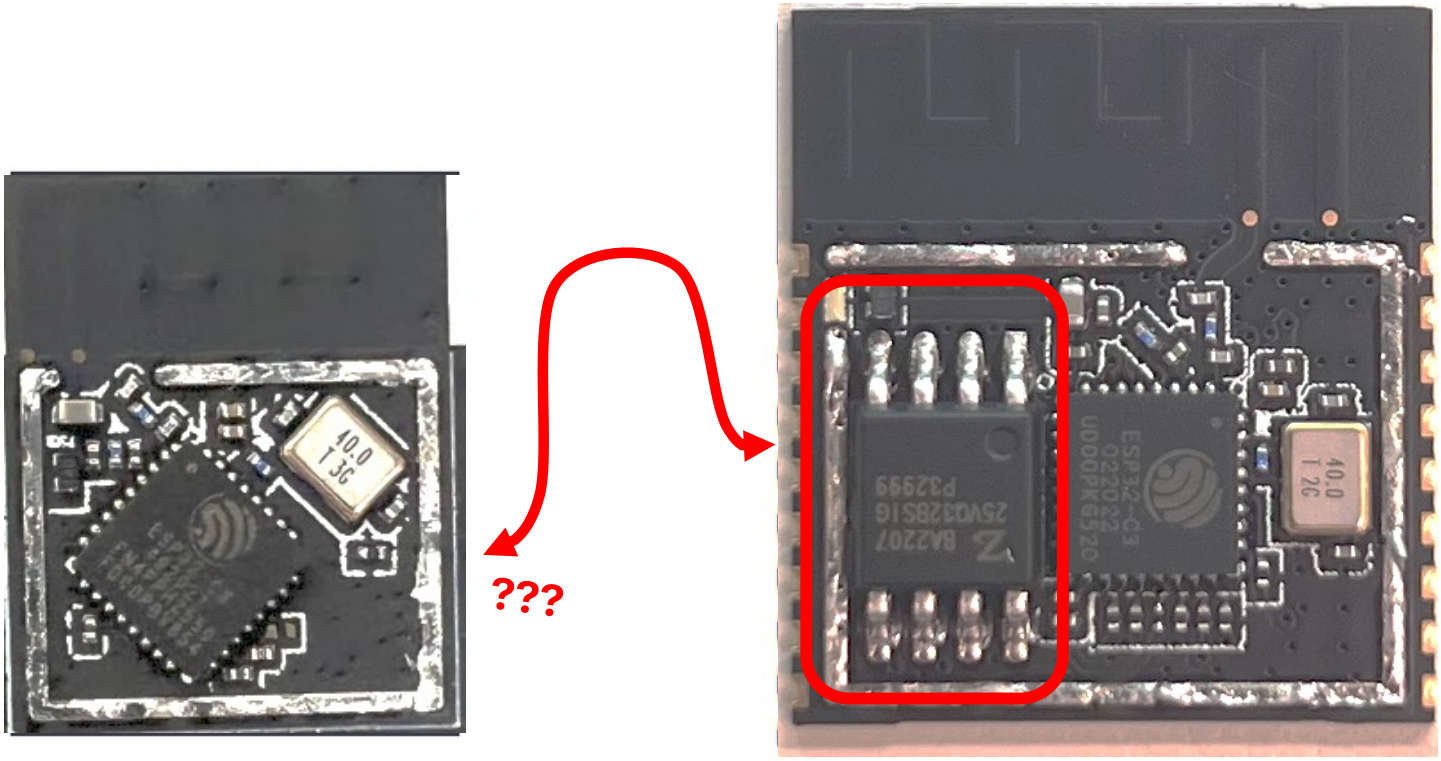
*I think the C6 can have up to 16 kB at slightly more power consumption

Flash Memory

- This is the largest chunk.
- Flash memory is non-volatile!
- When code compiles it goes here
- This can also be used as a non-volatile storage location for data (data can persist after power cycles!!!!)



ESP32 C3 Mini vs WROOM-02



“They’re the same. One is just smaller.”
-Joel Voldman
March 5, 2026 approximately 12pm

Find the Datasheets

- Both modules are ESP32C3's with 4 MB of Flash memory, with identical pin count.
- One module is smaller

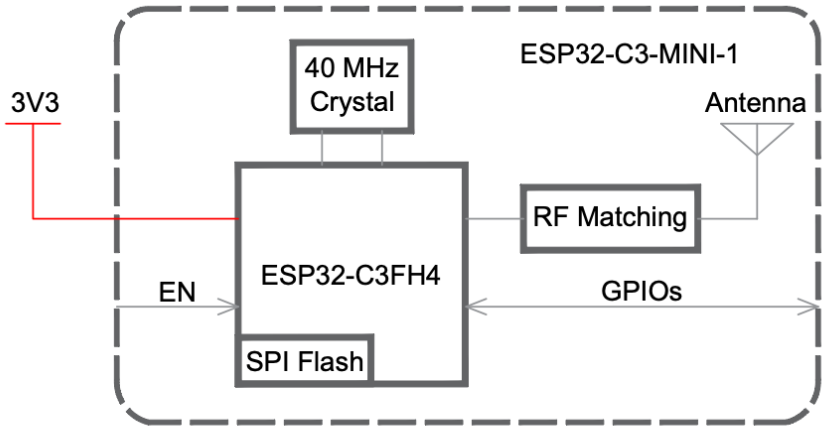


Figure 2-1. ESP32-C3-MINI-1 Block Diagram

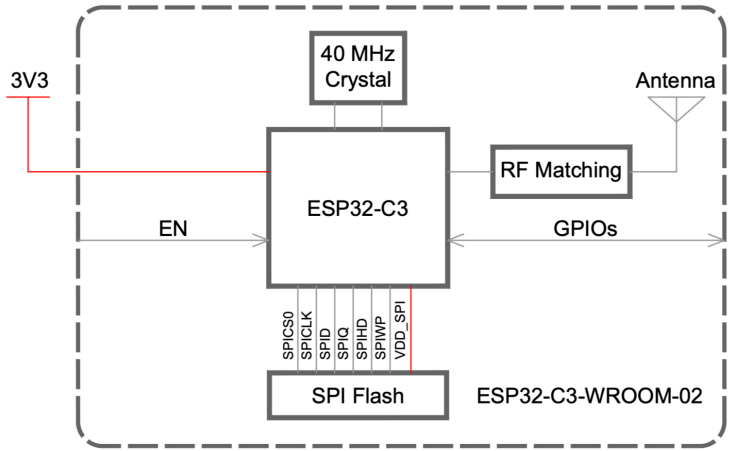


Figure 2-1. ESP32-C3-WROOM-02 Block Diagram

differences

ESP32C3-WROOM

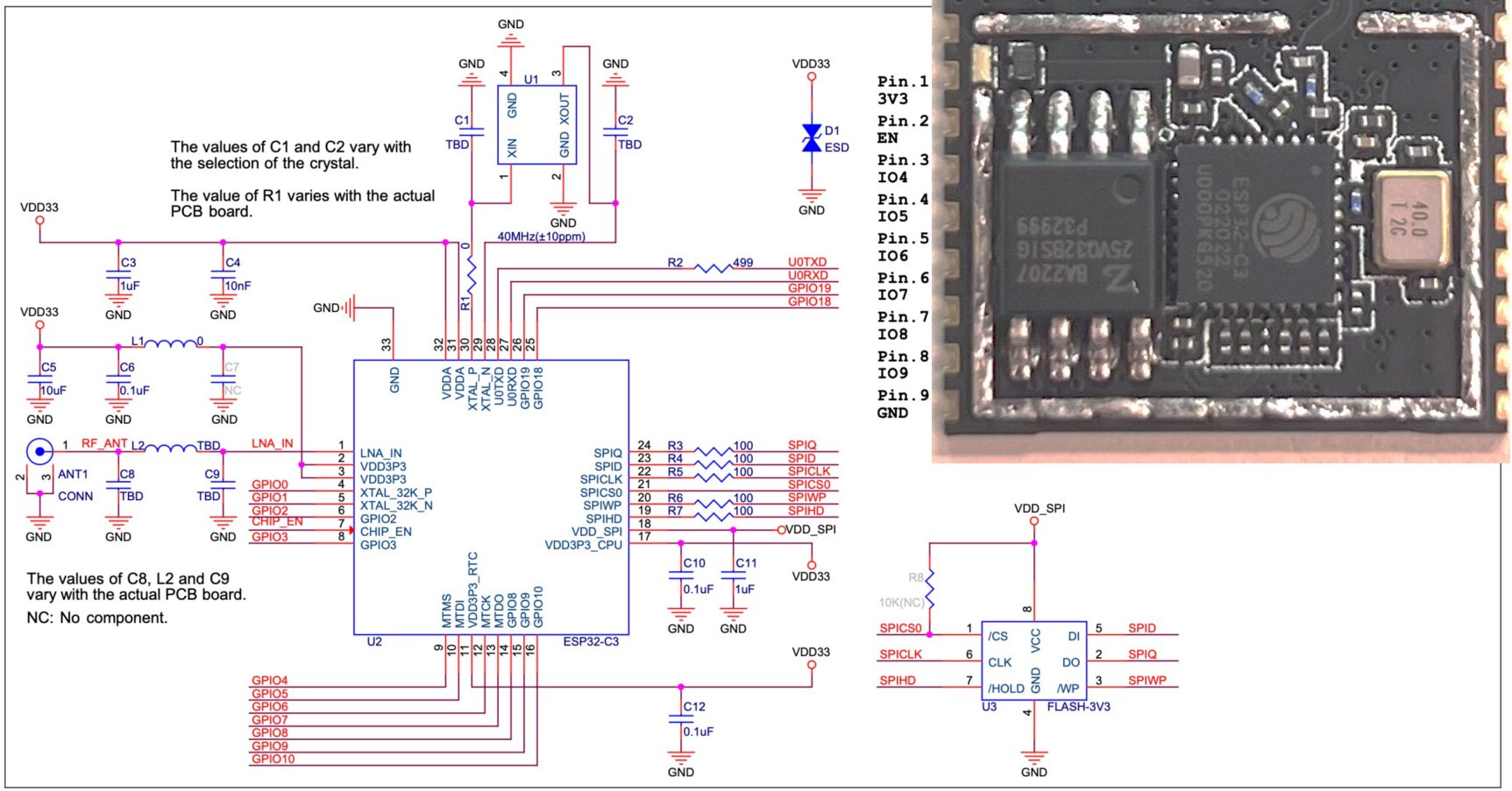
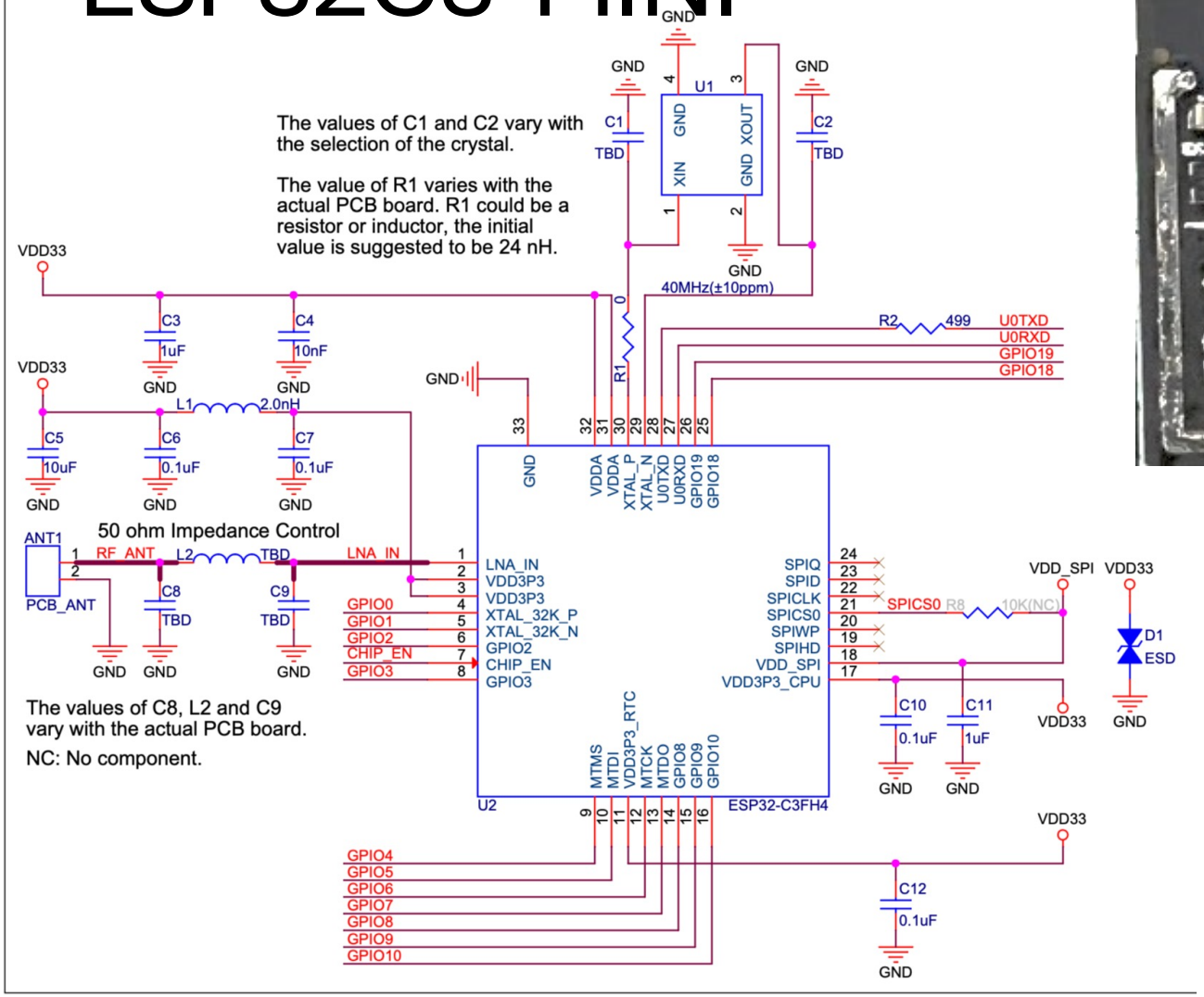


Figure 8-2. ESP32-C3-WROOM-02U Schematics

ESP32C3-MINI



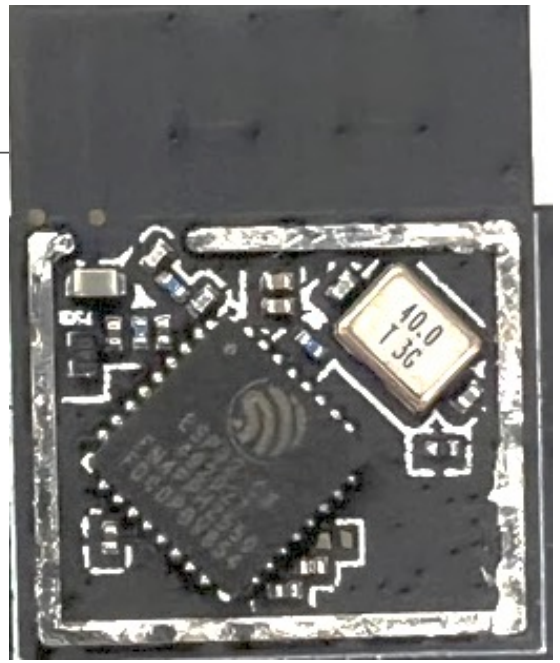
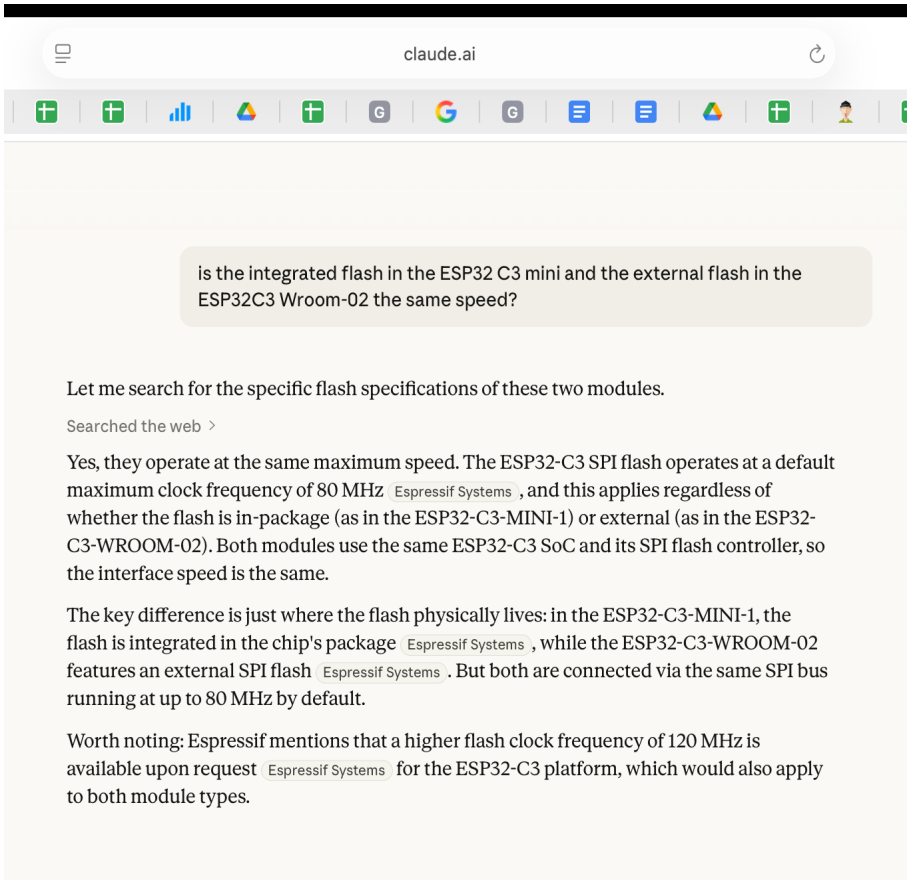


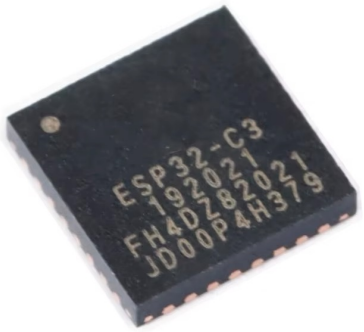
Figure 8-1. ESP32-C3-MINI-1 Schematics

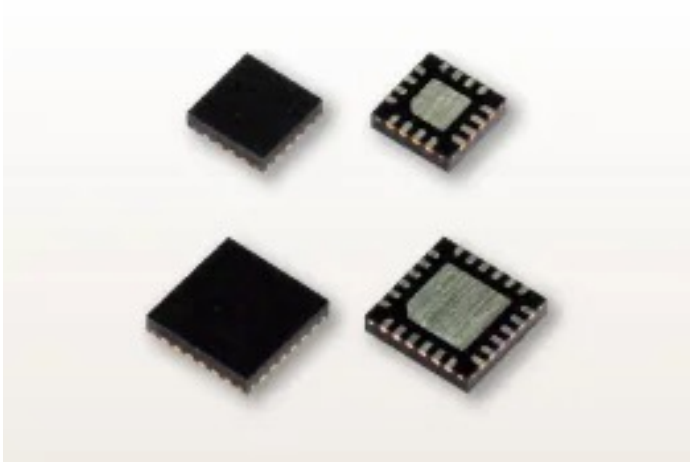
Interesting

- Claude claims that they’re the same.
- Must be true…OK



How do they integrate the Flash?

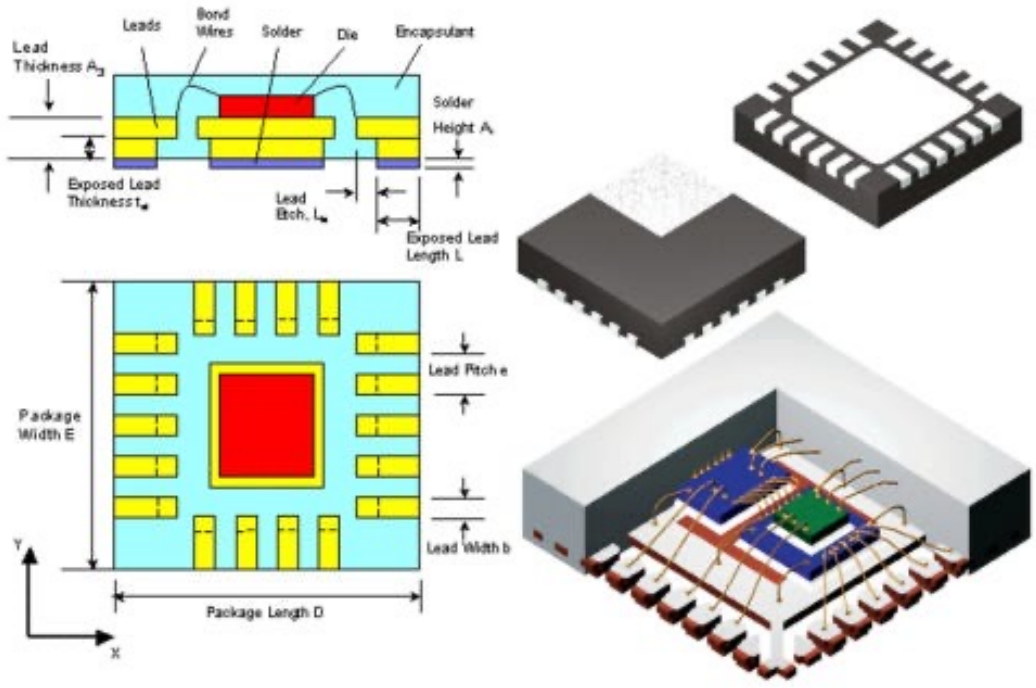


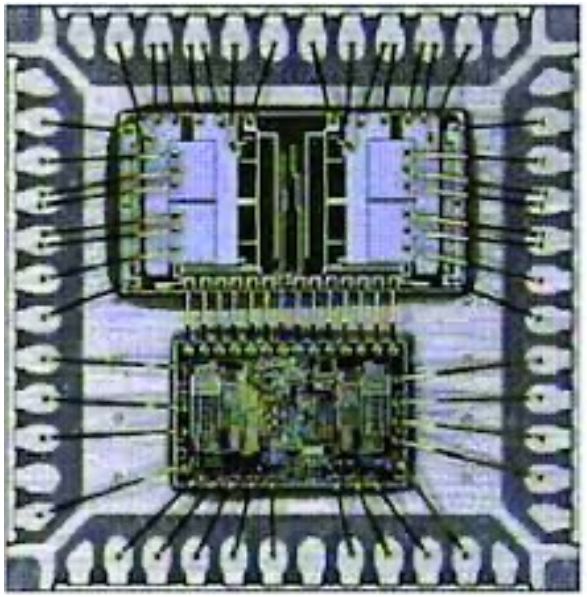


QFN32

Quad **F**lat **N**o-Leads 32

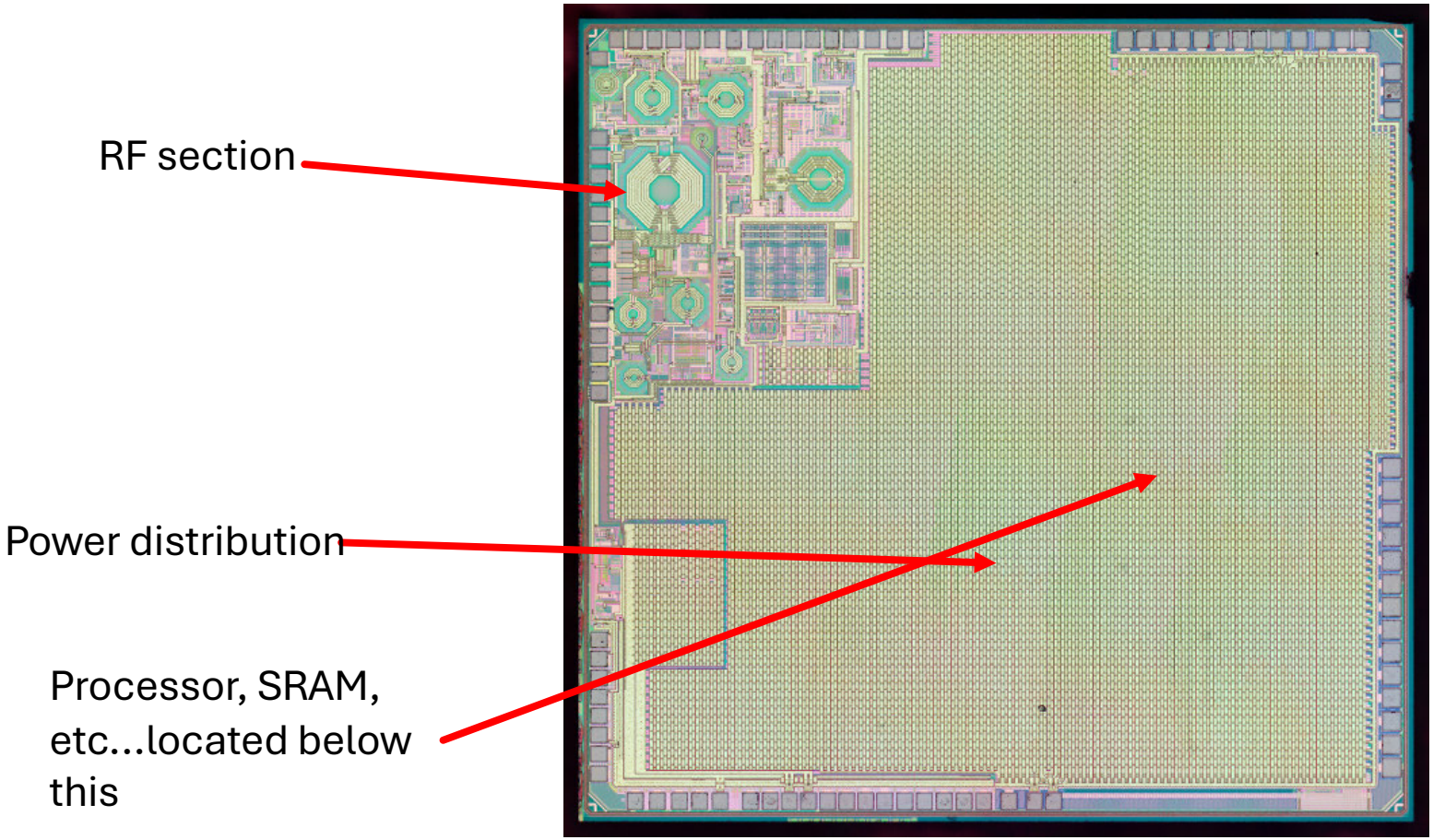
QFN with multiple dies





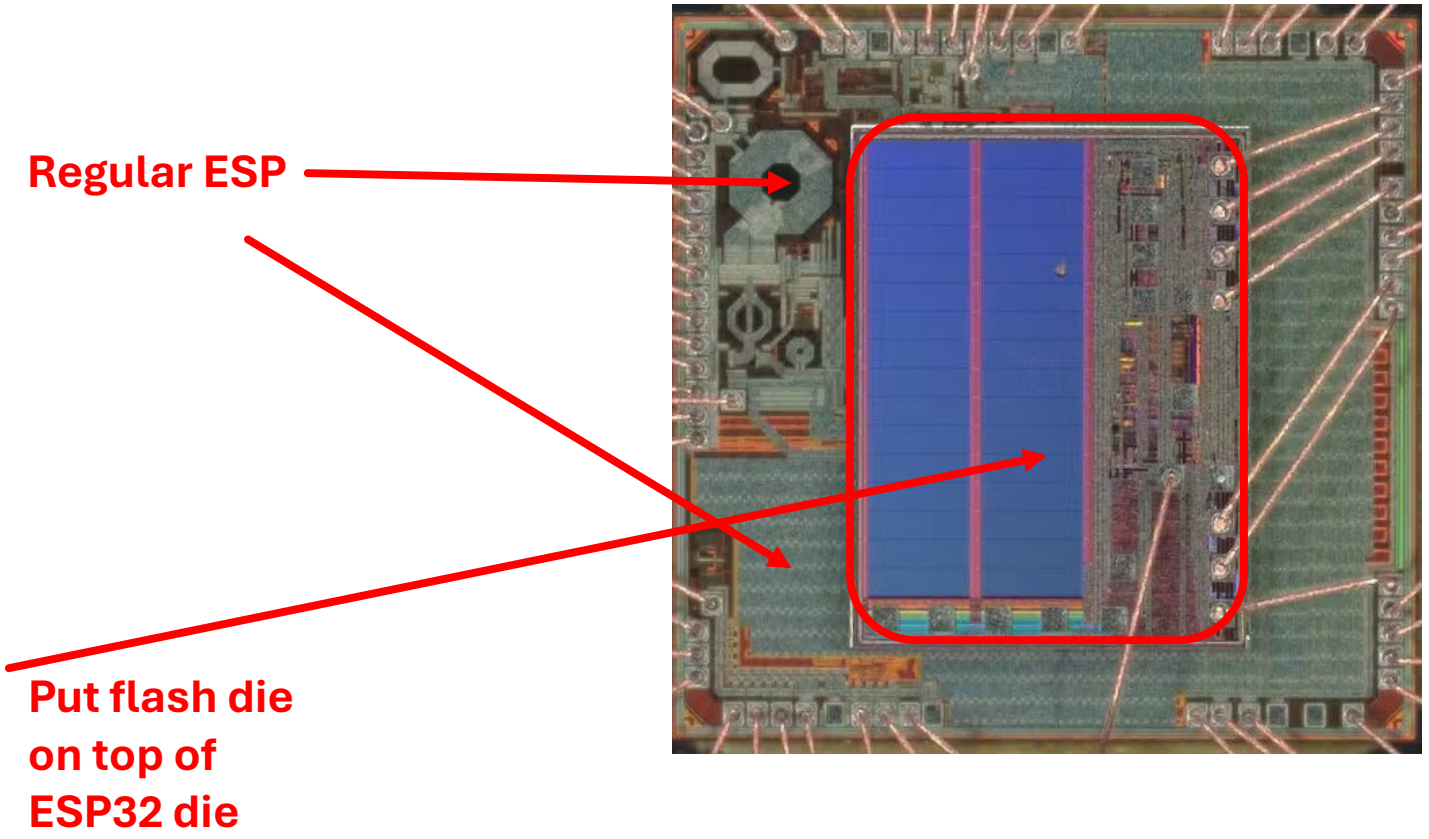
https://medium.com/@anil_21p/qfn-package-simulation-a592adaab4d0

ESP32 die shot



https://www.cnx-software.com/2017/02/06/oh-look-espressif-esp32-die-shot-thats-pretty/

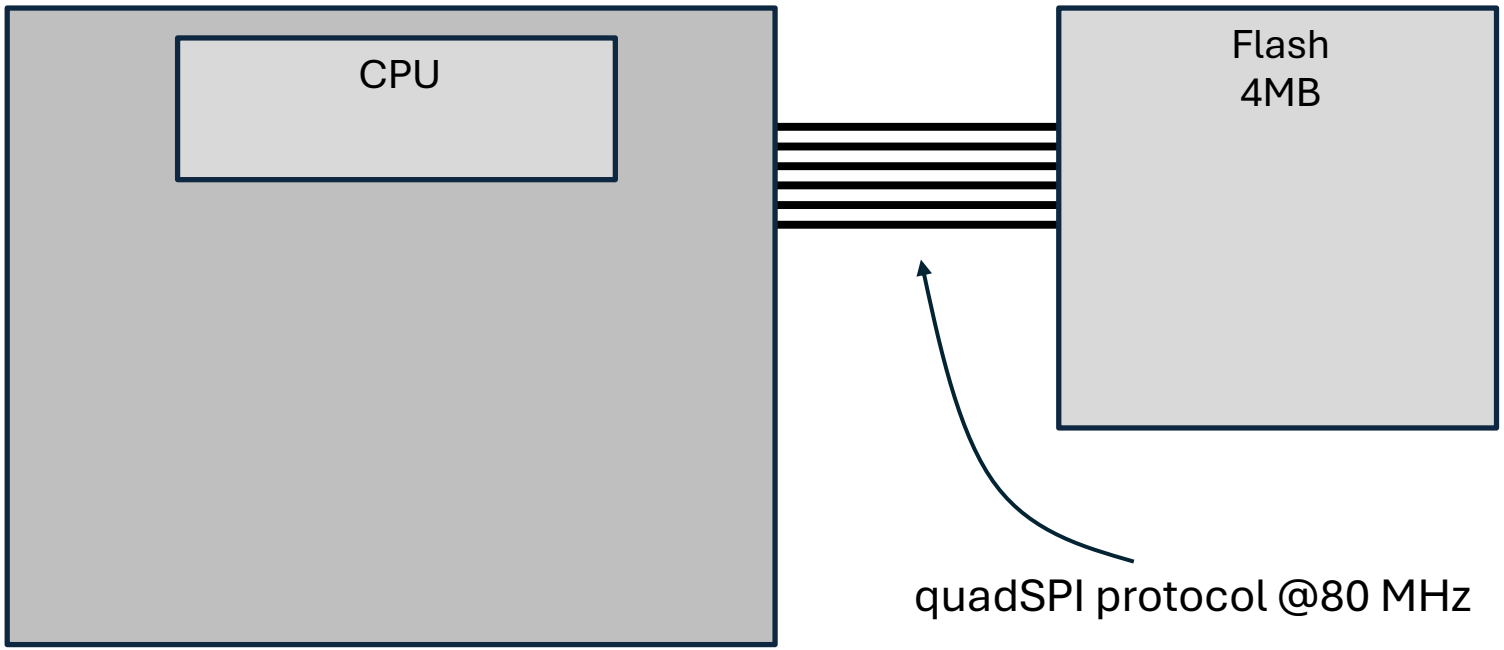
ESP32 C2 with integrated Flash memory



https://www.hackster.io/news/espressif-unveils-the-esp32-c2-an-optimized-iot-part-to-help-relieve-the-great-component-shortage-39342cf217c4

So Conclusions

- Regardless of if flash is on or off chip it uses an identical protocol



What is Flash Memory?

- It is *non-volatile* memory
- You use quantum tunneling to set bits and then read them out later
- Same family of technologies in SSD's and USB sticks (different variants to be clear)

Options to Work with Flash

- You will use it no matter what. That's literally what gets "flashed" when you "flash" the board.
- Your code lives in flash and at boot a portion of what's there is loaded into SRAM
- Then during run-time the L1 cache will try to stay fresh by retrieving instructions from the Flash

- BUT….

idf.py size

- Can be flash left over

Memory Type Usage Summary

Memory Type/Section	Used [bytes]	Used [%]	Remain [bytes]	Total [bytes]
Flash Code	589040			
.text	589040			
DRAM	112006	34.86	209290	321296
.text	83126	25.87		
.bss	17168	5.34		
.data	11712	3.65		
Flash Data	101600			
.rodata	101344			
.appdesc	256			
RTC SLOW	56	0.68	8136	8192
.force_slow	32	0.39		
.rtc_reserved	24	0.29		

You can use portions for file storage

- Non-volatile storage in fact!!!
- As much as a ***couple of MB*** which will persist even after full power cycle because it is non-volatile
- Will need to configure stuff keep in mind via idf.py menuconfig

NVS library

- Probably simplest:
- Some hierarchical organization, but largely key-value storage!

```c
#include "nvs_flash.h"
#include "nvs.h"
nvs_flash_init();

// do this wherever
nvs_handle_t handle;
nvs_open("storage", NVS_READWRITE, &handle);
nvs_set_i32(handle, "boot_count", 42);
nvs_set_str(handle, "device_name", "sensor_01");
nvs_commit(handle);
nvs_close(handle);

// later, read with a get
nvs_open("storage", NVS_READONLY, &handle);
int32_t count;
nvs_get_i32(handle, "boot_count", &counter);
nvs_close(handle);
```

https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/nvs_flash.html

SPIFFS

- More like file system,
- Higher level

```c
#include "esp_spiffs.h"
https://docs.espressif.com/projects/esp-idf/e
esp_vfs_spiffs_conf_t conf = {
    .base_path = "/storage",
    .partition_label = NULL,
    .max_files = 5,
};
esp_vfs_spiffs_register(&conf);

// then just use standard C file I/O
FILE *f = fopen("/storage/data.txt", "w");
fprintf(f, "sensor reading: %f\n", 23.5);
fclose(f);
```

https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/spiffs.html

Flash Caveat #1: General Usage

- Flash relies on quantum tunneling and extremely high electric fields to operate:
  - Often >10V over 5nm of gate oxide
- Quite large power consumption (should measure) but flash die will pull 10 to 20mA on reads, more on writes
- Most flash cells can only get flashed 10K to 100K times
- Careful not to write too many times!

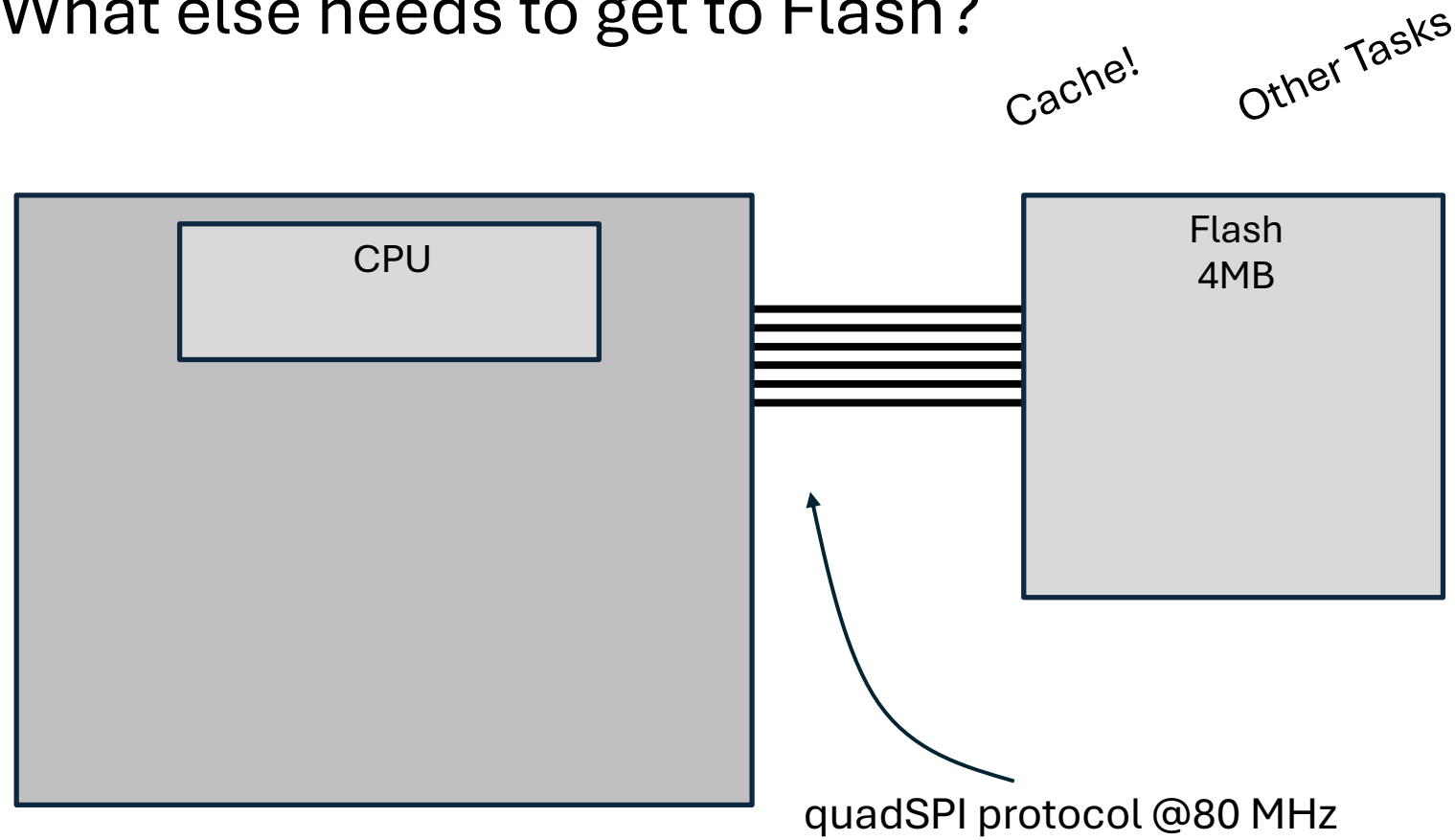
Flash Caveat #2: Timing

- Sector erases of flash can take actual human amounts of time (100 ms or so!!!)

- Implications!

There's Only One Way in and Out

- What else needs to get to Flash?



Sharing the Flash

- The Cache will need periodic access (ideally high priority) to the Flash…if not program stalls
- The Wifi libraries on espressif put configuration and other large things into flash and reference them during run. Those associated tasks need to be able to read parts of that stuff periodically so jamming the flash up with needless writes and reads can be a problem!!!
- The flash is NOT thread-safe. ***Will likely need to use mutexes***

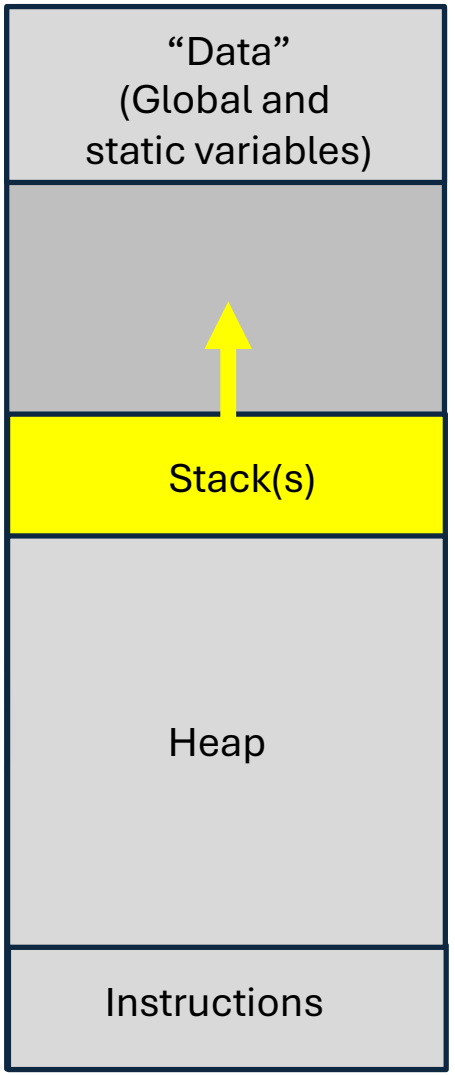
Flash Caveat #3: Over-the-Air (OTA)

- If you do use OTA, as Joel said, a big portion of the flash is lost since you need to essentially have two copies of program at one time
  - One being written/downloaded ota
  - One running
  - Then they swap
- In addition to losing flash size, this can potentially be very tricky with WiFi and cache hit/miss situations so needs testing!!!

The SRAM Organization: Stack

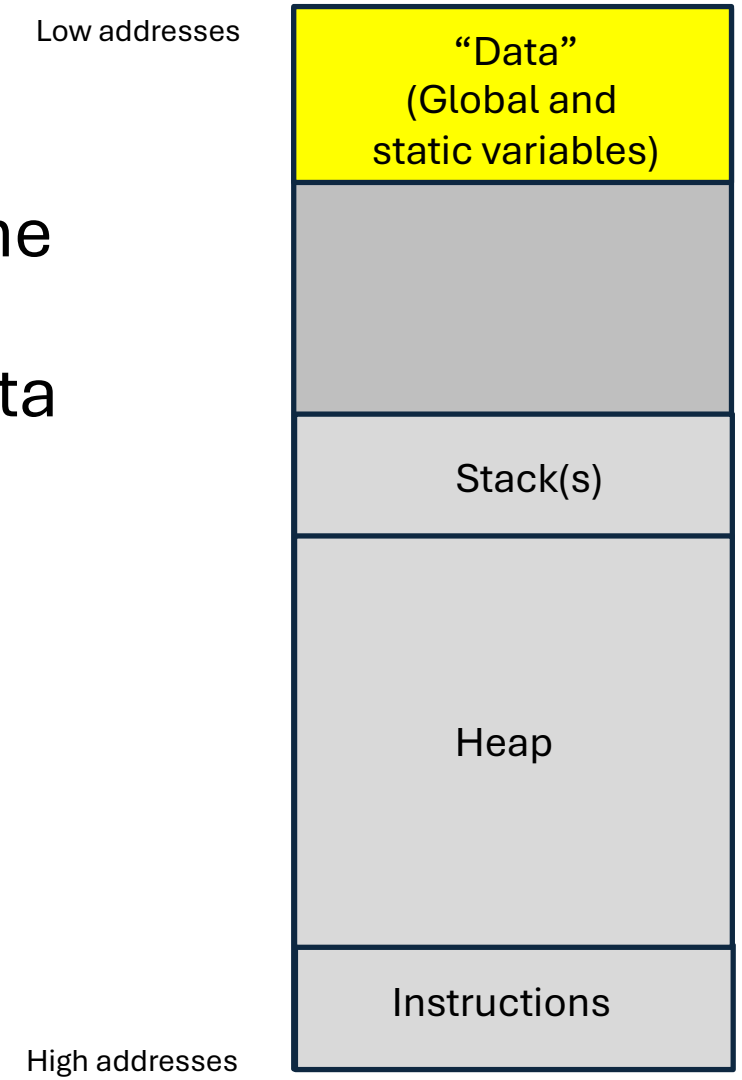
- Where local variables and local data live
- Keep in mind sizes!

```
void app_main(void) {
    xTaskCreate(task_a, "A", 2048, NULL, 5, NULL);
    xTaskCreate(task_b, "B", 2048, NULL, 5, NULL);
    xTaskCreate(task_c, "C", 2048, NULL, 5, NULL);
}
```



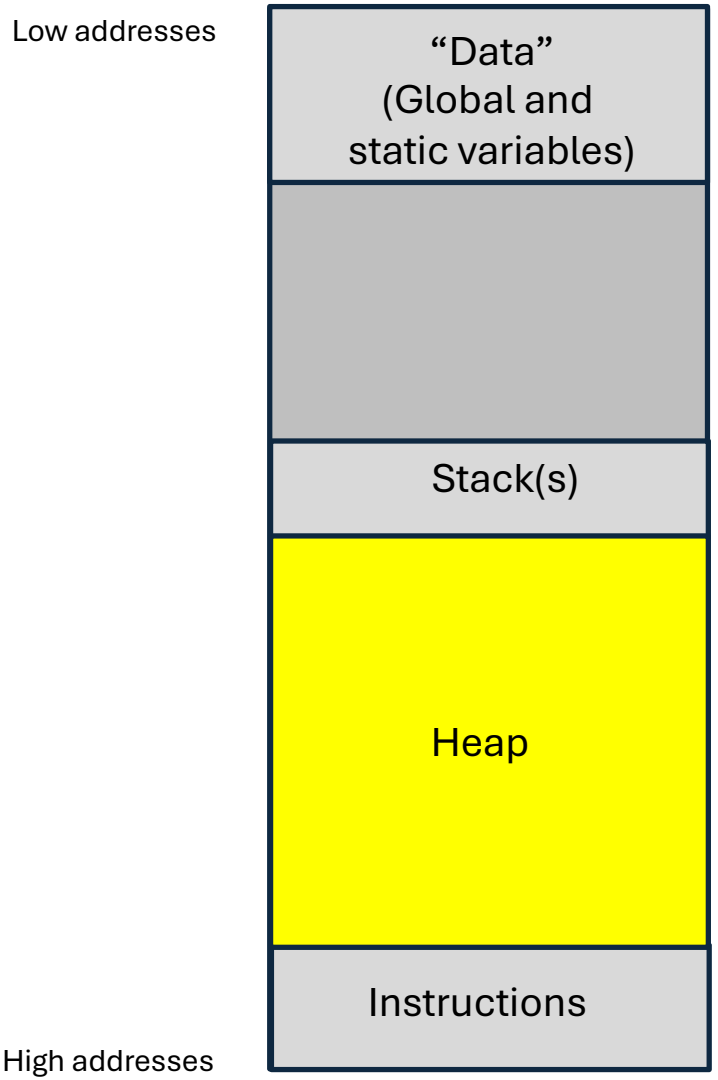
The SRAM Organization: Globals

- Put big buffers and thing in the global scope…they live in different area of memory (data section)



The SRAM Organization: Heap

- Bigger things,
- Things with sizes known only at run-time,
- These things all go into the memory heap



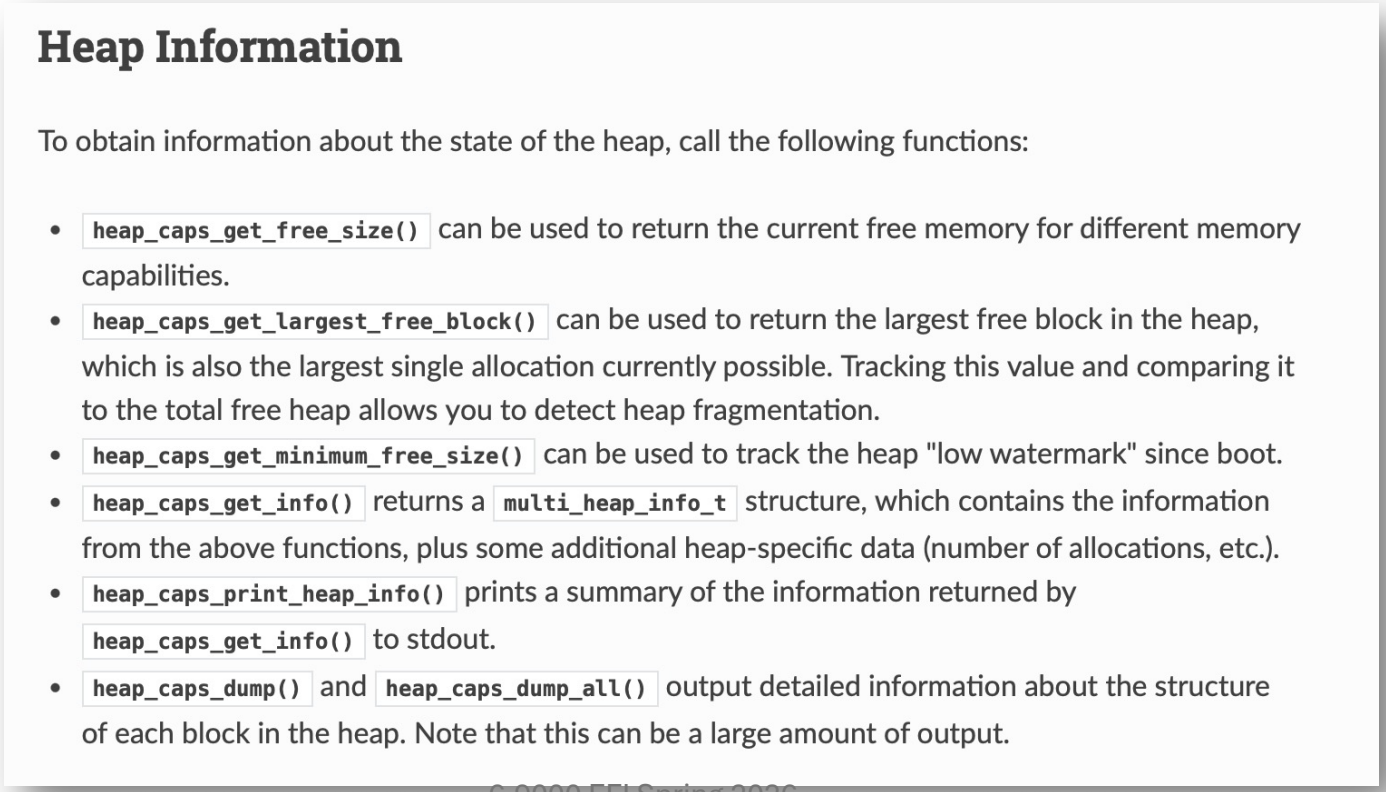
Long-Term Memory Stability

- Try to stick to using the stack and global variables wherever possible!

- Using malloc/free can potentially lead to memory leaks!

Heap vs. Stack vs. Memory

- Docs have tons of great analysis mechanisms to study the heap and get readings off of it



Heap Information

To obtain information about the state of the heap, call the following functions:

- `heap_caps_get_free_size()` can be used to return the current free memory for different memory capabilities.
- `heap_caps_get_largest_free_block()` can be used to return the largest free block in the heap, which is also the largest single allocation currently possible. Tracking this value and comparing it to the total free heap allows you to detect heap fragmentation.
- `heap_caps_get_minimum_free_size()` can be used to track the heap "low watermark" since boot.
- `heap_caps_get_info()` returns a `multi_heap_info_t` structure, which contains the information from the above functions, plus some additional heap-specific data (number of allocations, etc.).
- `heap_caps_print_heap_info()` prints a summary of the information returned by `heap_caps_get_info()` to stdout.
- `heap_caps_dump()` and `heap_caps_dump_all()` output detailed information about the structure of each block in the heap. Note that this can be a large amount of output.