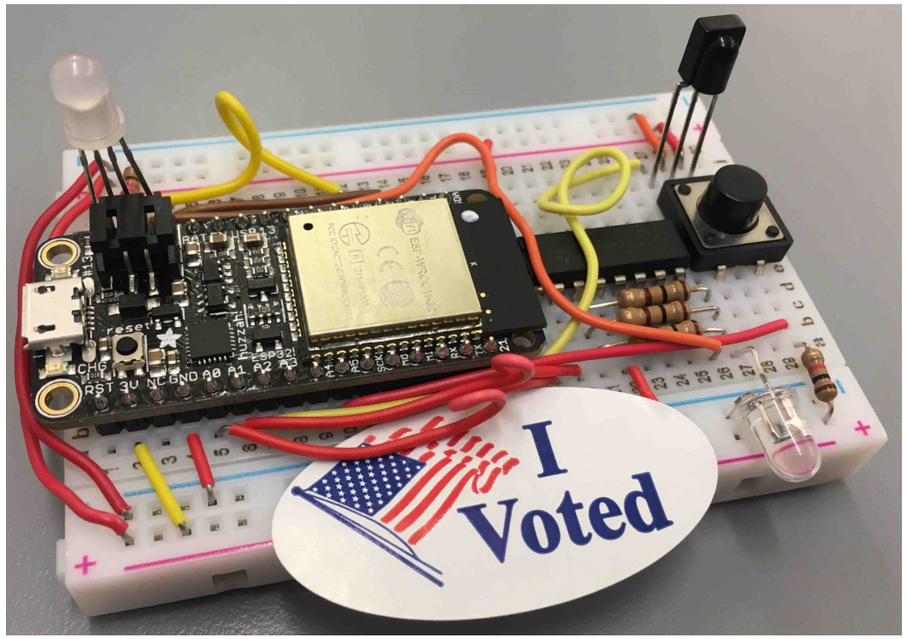
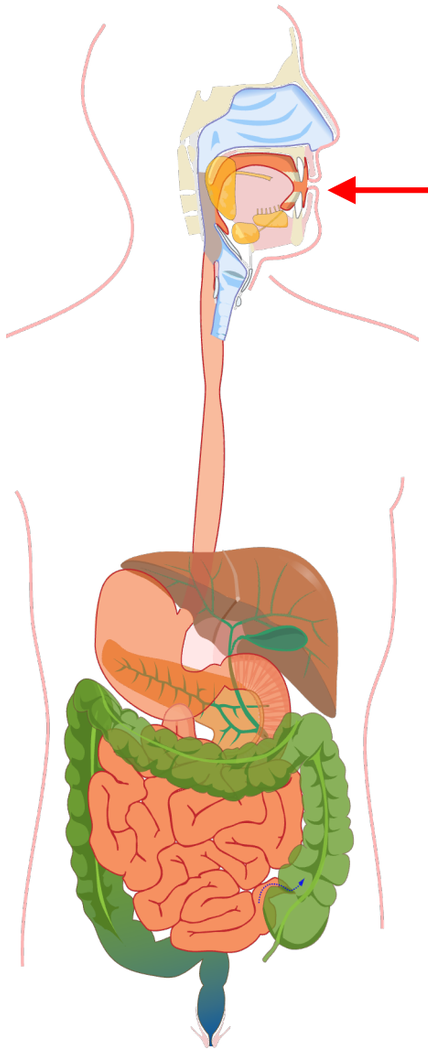



# BU-EC444

Fall 2024
Prof. Little

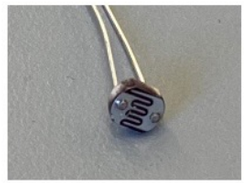
## Review of Quest 1: Smart Pill

# Q1: Ingestible “Smart Pill”



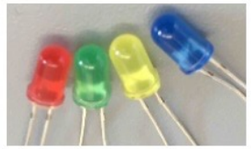
Smart Pill

- Temperature
- Light
- Battery
- Tilt



Photocell to detect light levels and changes

Thermistor to measure temperature changes



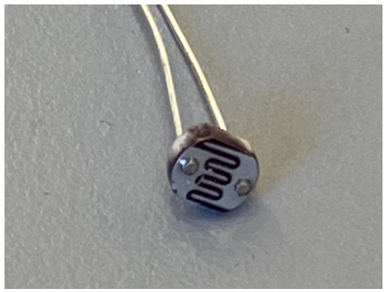
LEDs to indicate active

Tilt indicator for orientation

Console

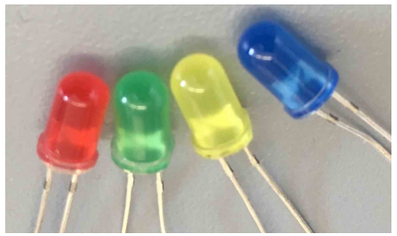
```
Time: 10 s, Temp: 38.1 C, Light: 0 Lux, Battery: 3.1 V, Tilt: vertical
Time: 12 s, Temp: 38.2 C, Light: 0 Lux, Battery: 3.1 V, Tilt: vertical
Time: 14 s, Temp: 38.1 C, Light: 0 Lux, Battery: 3.1 V, Tilt: horizontal
Time: 16 s, Temp: 38.2 C, Light: 0 Lux, Battery: 3.1 V, Tilt: horizontal
Time: 18 s, Temp: 38.3 C, Light: 0 Lux, Battery: 3.1 V, Tilt: vertical
Time: 20 s, Temp: 38.4 C, Light: 0 Lux, Battery: 3.1 V, Tilt: vertical
Time: 22 s, Temp: 38.7 C, Light: 0 Lux, Battery: 3.1 V, Tilt: vertical
Time: 24 s, Temp: 39.0 C, Light: 0 Lux, Battery: 3.1 V, Tilt: vertical
```

# Quest 1: Ingestible



Photocell to detect light levels and changes

Thermistor to measure temperature changes



LEDs to indicate active

Tilt indicator for orientation

Key skills

- Console IO
- Analog sensors
- GPIO input and output
- LEDs
- RTOS – tasks, queues
- Interrupts

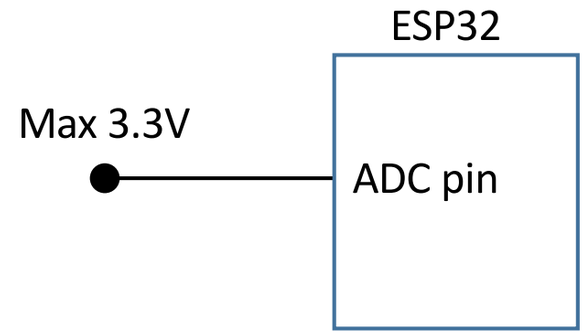
## Quest 1 Skill Cluster

- Battery Monitor
- Thermistor
- Solar Cell/Photocell
- Console IO
- RTOS Tasks
- Hardware Interrupts

# Analog Signal Measurement

# Measuring Voltage with ADC

- ESP32 ADC measures from 0V to maximum of 3.3V
  - But depends on how you scale it
  - 2^12 to work with (4096)
- If sensor output is greater than max input, then need divide down to range of 0—3.3V.
- Pick ADC channel and assign
- Decide how often to read the ADC
- Converting to engineering units means converting counts to desired units (e.g., counts to Volts)
- Need to know how many Volts/count

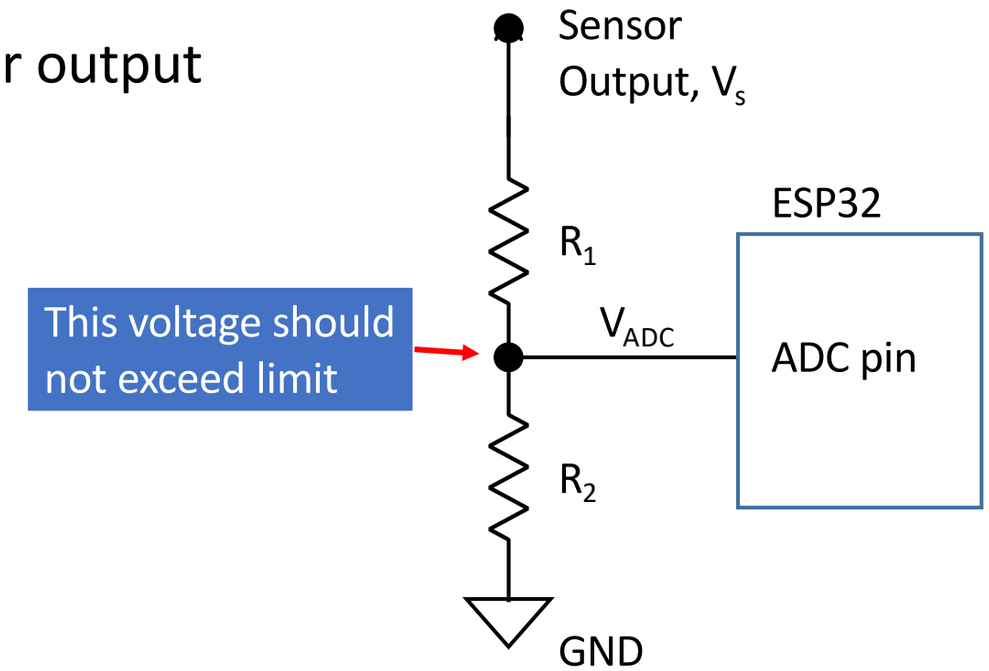


# Voltage Dividers

- Steps
  - Determine sensor output range, e.g., 5V
  - Figure out ADC range, e.g., 3.3V
  - Determine $R_1$ and $R_2$ to step down sensor output voltage to within Max range

- Remember Ohm's Law
  - $V = IR$
  - $V_s = I(R_1 + R_2) \Rightarrow I = V_s/(R_1 + R_2)$
  - $V_{ADC} = IR_2$
  - $V_{ADC} = (V_s/(R_1+R_2))R_2 = V_sR_2/(R_1+R_2)$

- Best practice: Choose high-value resistors for low current draw



## From the ADC section of the ESP32 document set

- The ESP32 integrates two 12-bit SAR (Successive Approximation Register) ADCs supporting a total of **18 measurement channels** (analog enabled pins).
- The ADC driver API supports
  - ADC1 (8 channels, attached to GPIOs 32 - 39)
  - ADC2 (10 channels, attached to GPIOs 0, 2, 4, 12 - 15 and 25 - 27).
    - **Note that ADC2 is used by the Wi-Fi driver** -- avoid
    - Therefore, the application can only use ADC2 when the Wi-Fi driver has not started.
    - In addition, some other pins cannot be freely used (see documents)
- Configuration: (width: bits; gain: attenuation)
  - For ADC1, configure desired precision and attenuation by calling functions `adc1_config_width()`and `adc1_config_channel_atten()`.
  - For ADC2, configure the attenuation by `adc2_config_channel_atten()`. The reading width of ADC2 is configured every time you take the reading.

## Other notes

- **You are not required to calibrate the ADC in the course**. This is optional. Calibration uses a reference voltage to improve accuracy of measurement
- Attenuation refers to software-based signal scaling

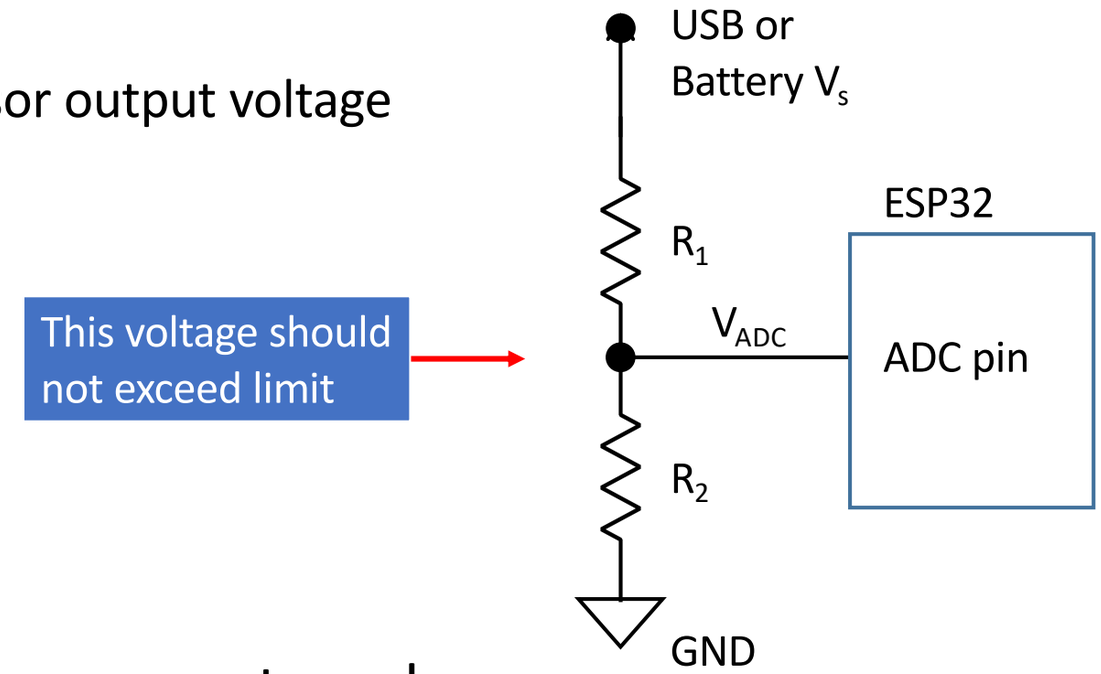
    1. First you need to ensure that the input signal does not exceed the Max value

    2. Then you can use the attenuation settings to best map the sensor range to the ADC range

# Battery Monitor

# Battery monitor skill

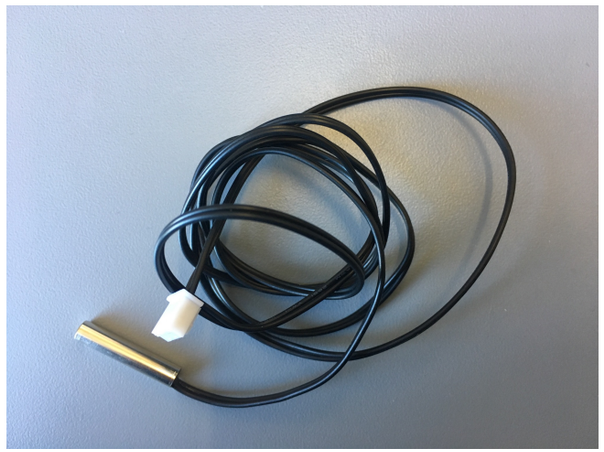
- Steps
  - The USB input to the ESP operates at 5V (this is larger than the Max voltage of the ADC)
  - Figure out ADC range, e.g., 3.3V
  - Determine $R_1$ and $R_2$ to step down sensor output voltage
  - Use a conservative max of 3V
- Use Ohm's Law
  - $V = IR$
  - $V_s = I(R_1 + R_2) \Rightarrow I = V_s/(R_1 + R_2)$
  - $V_{ADC} = IR_2$
  - $V_{ADC} = (V_s/(R_1+R_2))R_2 = V_sR_2/(R_1+R_2)$
- Pick $R_1 + R_2$
- Adapt the example code to generate raw counts and convert to engineering units



# Thermistor

# Thermistor skill

- Wire up the thermistor (doesn’t matter which one) using a voltage divider
- Investigate how to convert from measured ADC counts to °C
- Send measured values to the console



Two types of Thermistors

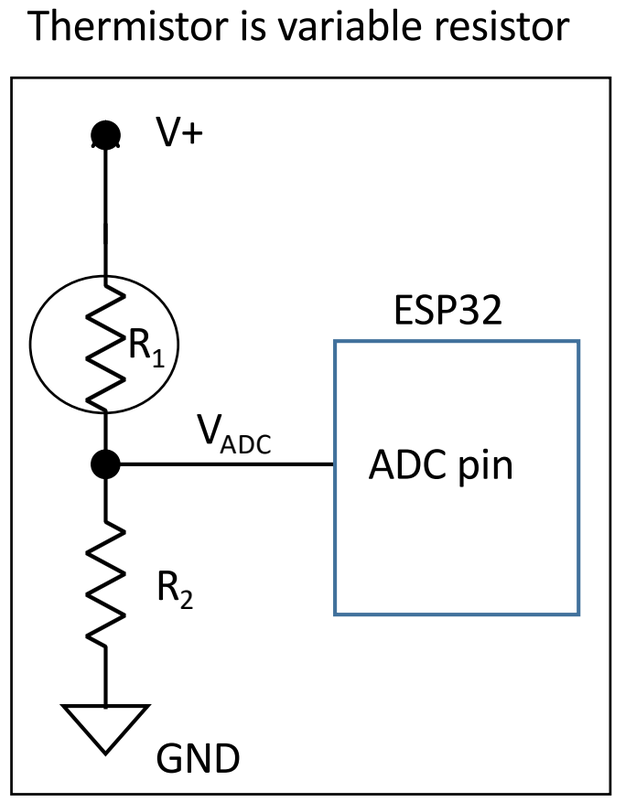
# Voltage divider for thermistor

- Steps
  - Need to ensure ADC input limit is maintained (< 3.3V)
  - Determine $R_1$ and $R_2$ to step down sensor output voltage

- Remember Ohm's Law
  - $V = IR$
  - $V_s = I(R_1 + R_2) \Rightarrow I = V_s/(R_1 + R_2)$
  - $V_{ADC} = IR_2$
  - $V_{ADC} = (V_+/(R_1+R_2))R_2 = V_+R_2/(R_1+R_2)$

- Could use V+ = 3.3V or V+ = 5V

This voltage should not exceed limit



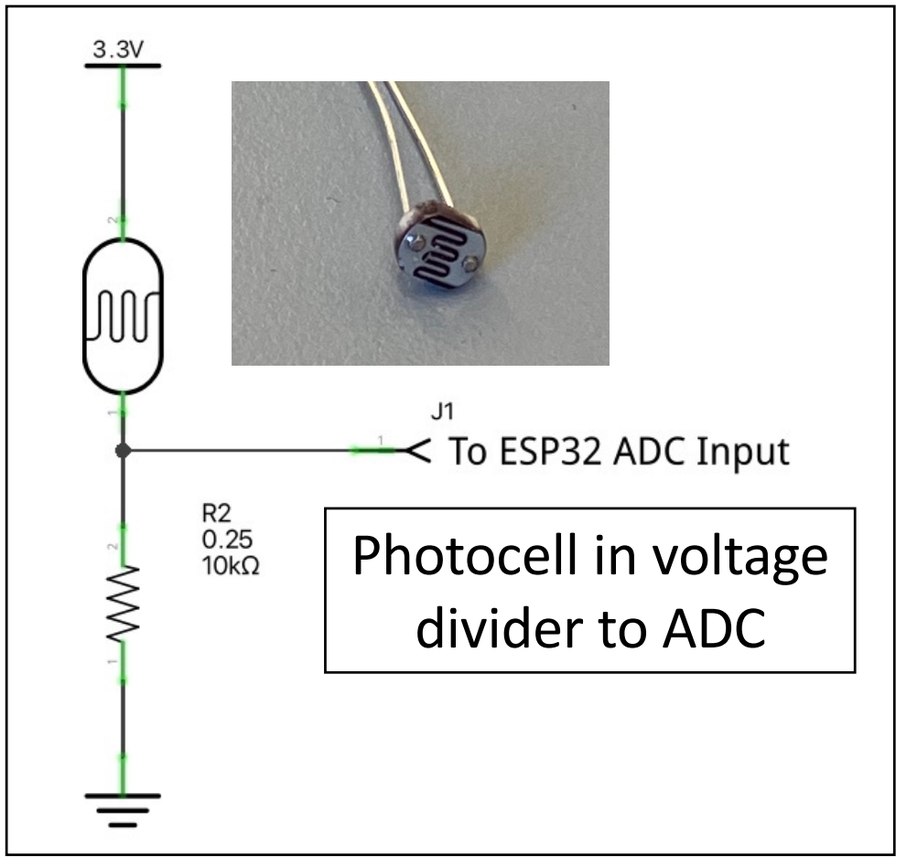

Next: find model for converting voltage to engineering units

# Solar Cell/Photocell

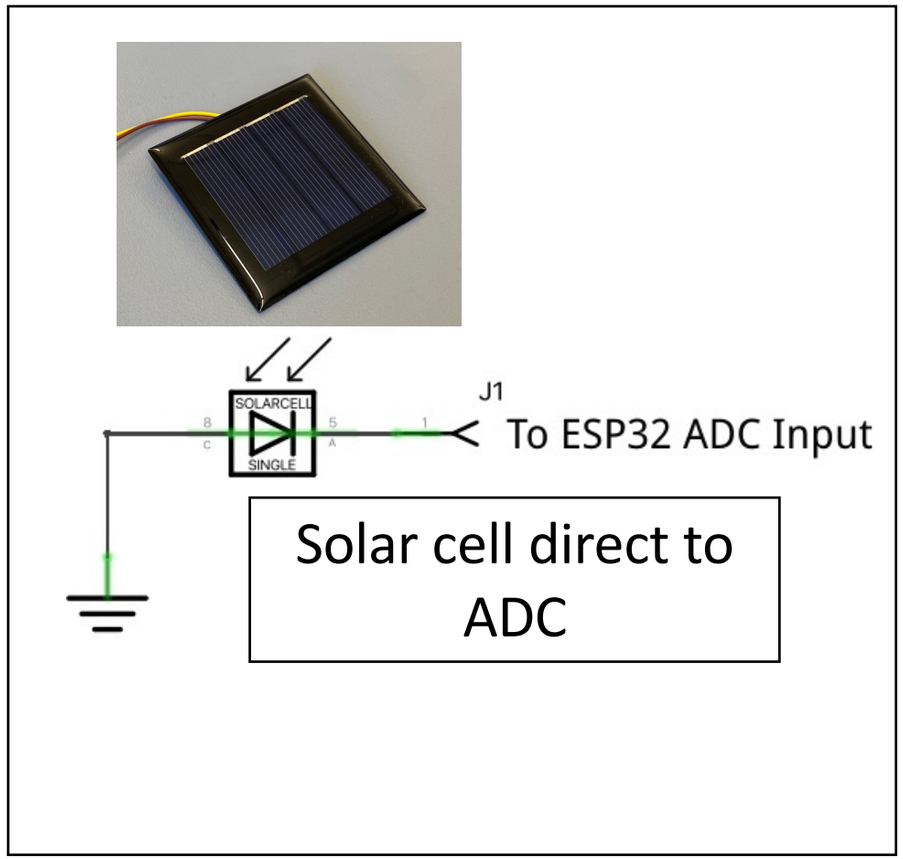
# Solar Cell/Photocell skill

Two options for measuring light intensity

Photocell is variable resistor

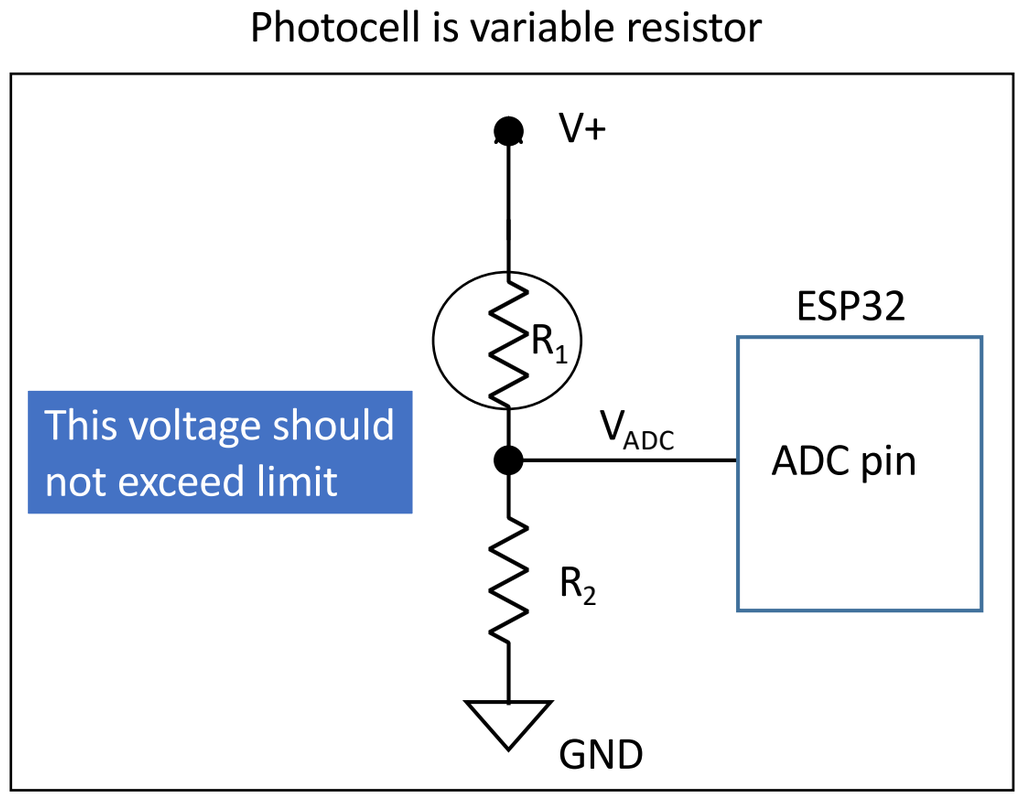



Solar cell is a power source



# Voltage divider for photocell

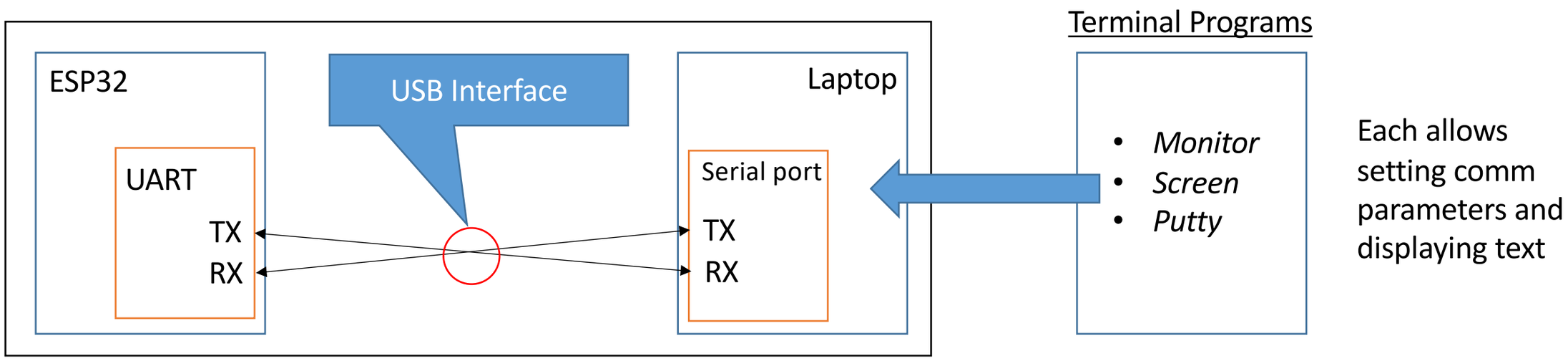
- Steps
  - Need to ensure ADC input limit is maintained (< 3.3V)
  - Determine $R_1$ and $R_2$ to step down sensor output voltage
- Remember Ohm's Law
  - $V = IR$
  - $V_s = I(R_1 + R_2) \Rightarrow I = V_s/(R_1 + R_2)$
  - $V_{ADC} = IR_2$
  - $V_{ADC} = (V_+/(R_1+R_2))R_2 = V_+R_2/(R_1+R_2)$
- Could use V+ = 3.3V or V+ = 5V



# Serial Communications and Console I/O

# Serial Communications

- Data are streamed one bit at a time
- Baud Rate, or communication speed, is specified beforehand
- We're going to be using the UART (**u**niversal **a**synchronous **r**eceiver/ **t**ransmitter)
- It is typically, 2 wires (TX / RX) but the USB interface takes care of that



## *Monitor* program

- `Monitor`: serial program that communicates with ESP32
- https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/tools/idf-monitor.html
- It is basically a terminal program with a few other features

- Can also use other terminal programs such as `putty` (windows), or `screen` (mac and linux)

- Make sure the port parameters are what you expect (ESP32 and terminal program need the same parameters)

## `stdio` versus reading bytes directly

- Can use `stdio` c library functions without initiating explicit UART drivers
  - `gets(), puts(),` etc.
  - Non-blocking read: not as reliable
- With UART driver
  - Have blocking behaviors for `gets(), puts(),` etc.
- **We recommend** using `uart_read_bytes()`
  - Reads in a byte at a time
  - Blocking
  - More control as you are reading a byte, can format data as you please
- UART programming allows other devices to be the target instead of the console (like sensors, or actuators, or modems, LIDAR devices, or WiFi)

# UART on the ESP32

Include the UART library

Definitions like pins and UART number

```
#include "driver/uart.h                  // Include UART header
#define XYZ                              // Define your UART pins

void app_main() {                        // Your main program
        .baud_rate =                     // Assign values to UART data structure
        .data_bits =
        .parity =
        .stop_bits =
        .flow_control =
        uart_                            // Write above values to config UART
        uart_set_pin                     // Configure pins
        uart_driver_install              // Install UART driver

        uint8_t *data =                  // Instantiate UART buffer

        while (1) {                      // Loop indefinitely reading or writing
                uart_read_bytes()        // Read data from UART
        uart_write_bytes()               // Or write to UART
}}
```

**See ESP-IDF UART example**: Echo and in course code-examples

# Console.io skill

- The ESP IDF includes the monitor program which serves as a configuration and listening tool on the serial (USB) port. Here we step away from this app.
- We will explore different ways to communicate between the PC and the ESP32 over the wired USB interface using the UART directly
- Example code: Echo, as part of the ESP32 repo
  - Note: **need to modify example: UART_0 is used on the Huzzah board (not UART_1)**
  - Make sure baud rate on ESP32 and your terminal program are the same; no flow control
- Assignment:
  - Review the design pattern (linked from skill)
  - Program 3 modes:
    1. Toggle the onboard LED based on key input ‘t’
    2. Echo input (2 or more characters) from the keyboard to the console
    3. Echo decimal number input as a hexadecimal
  - Figure out how to switch between the three modes based on key input ‘s’

# Setting UART to USB port parameters for the Huzzah

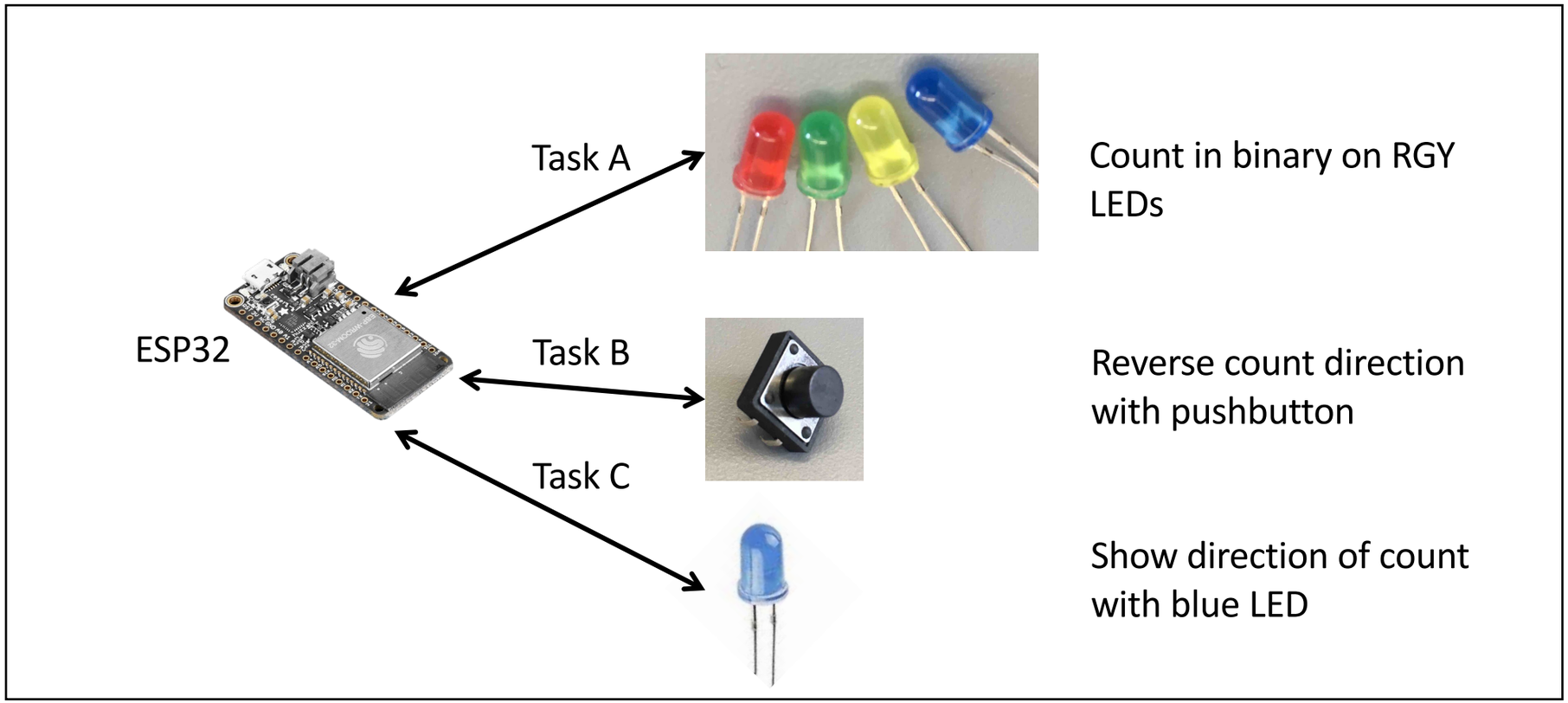
ESP32 demo code needs to be modified for the Huzzah USB port (UART_0)

- Port: UART0 (UART_NUM_0)
- Baud rate: 115200
- Flow control: off
- Parity bit: off
- Pin assignment: Default for UART_NUM_0
  - This means
  - `uart_set_pin(EX_UART_NUM, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);`

For terminal programs using UART, can use Putty (Wintel), Screen (Mac, Linux), Minicom (Linux), Monitor (IDF)

# RTOS and Tasks

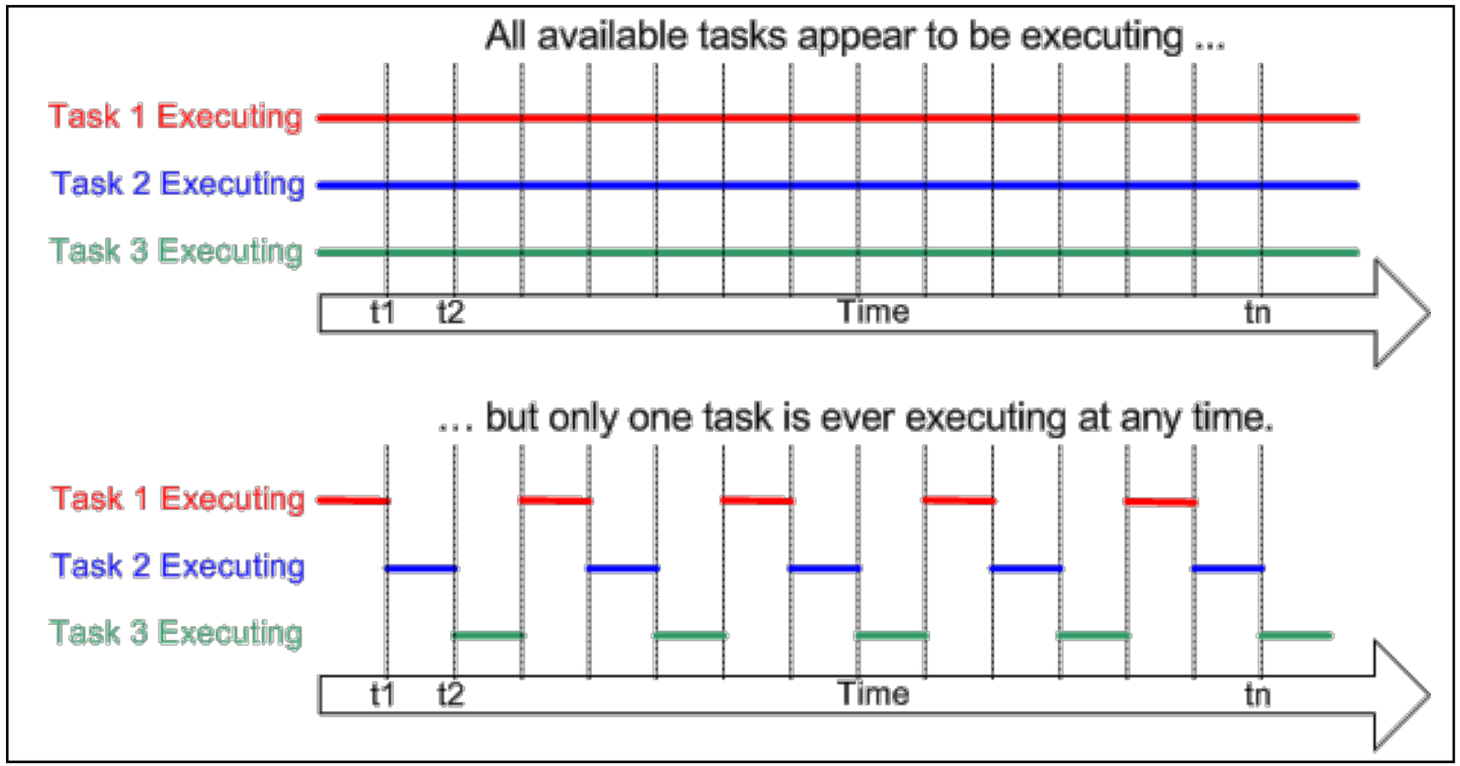
# RTOS skill




A: Count-up in binary
B: Change counting direction
C: Post direction on LED

# Role of tasks (see design pattern on tasks)

- Allow independent threads of execution
- Enable different timing
- Enable different priorities




From FreeRTOS

https://www.freertos.org/implementation/a00004.html

# Example single thread (e.g., example code from class Echo)

Includes

```
#include ...                          // Includes and defines
#define ...
```

Initialization function

```
void init() {                        // Convenient way to organize initialization
    ...                              // Do it in this sub
}
```

Body of program

```
void app_main()
{
        init();                      // Initialize stuff
        while(1){                    // Loop forever
                Do some stuff;       // Your code here
        }
}
```

Example: code in Echo https://github.com/BU-EC444/code-examples

(10ms delay inside task while loop suggested)

# Example using tasks (ESP32 echo example)

Includes

```
#include ”freertos/FreeRTOS.h”          // Includes and defines
# include “freertos/task.h”
#define ...
```

Initialization function

```
void init() {                           // Convenient way to organize initialization
    ...                                 // Do it in this sub
}
```

Task code

```
static void task_A()                    // Define a task here
{
        while(1){
                Do some stuff;          // Your task A code here
        }
}
```

Initialize, create and start tasks

```
void app_main()
{
        init();                         // Initialize stuff
        xTaskCreate(task_A, "task_A",1024*2, NULL, configMAX_PRIORITIES, NULL);
}                                       // Instantiate task with priority and stack size
```

Example: code in Echo https://github.com/espressif/esp-idf/tree/master/examples/peripherals/uart/uart_echo

## Notes on tasks

- **Stack:** need memory in task context to make function calls (which require stack space during calls)
- **Stack size:** documentation is less clear on how to determine size. Tools provided to assess how much is being used during execution
- Complexity of your program is **limited by the available memory** and performance
  - Ability to switch between tasks (memory limitation)
  - Ability to service time consuming tasks or high frequency tasks (execution speed)

- Learn about the underlying task execution system: https://www.freertos.org/implementation/main.html

# Example with multiple tasks

Includes

```
#include ”freertos/FreeRTOS.h”        // Includes and defines
# include “freertos/task.h”
#define ...
```

Initialization function

```
void init() {                          // Convenient way to organize initialization
    ...                                // Do it in this sub
}
```

Task A code

```
static void task_A()                   // Define a task here
{
        while(1){
                Do some stuff;         // Your task A code here
        }
}
```

Task B code

```
static void task_B()                   // Define a task here
{
        while(1){
                Do some stuff;         // Your task B code here
        }
}
```

**continued**

# Example with multiple tasks cont.

Initialize, create and start tasks

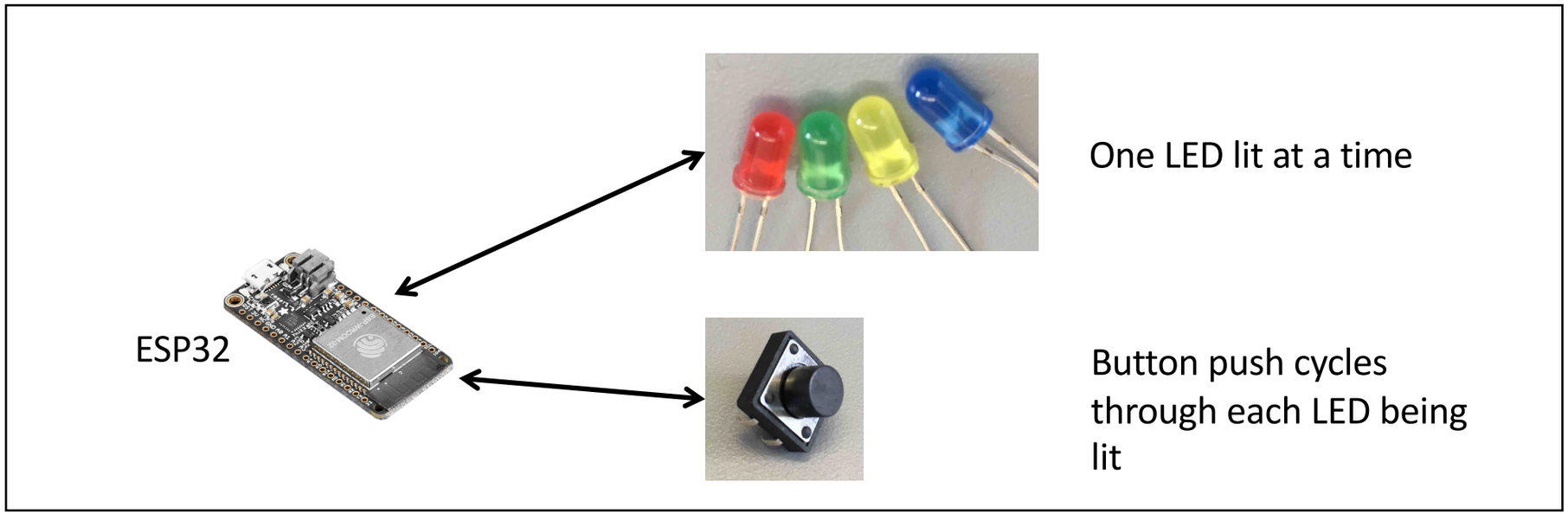
```
void app_main()
{
        init();                                 // Initialize stuff

        xTaskCreate(task_A, "task_A",1024*2, NULL, configMAX_PRIORITIES, NULL);
                                                // Instantiate task with priority and stack size

        xTaskCreate(task_B, "task_B",1024*2, NULL, configMAX_PRIORITIES-1, NULL);
}                                               // Instantiate task with priority and stack size
```

# Hardware Interrupts

# Hardware interrupt skill




1. Write program as a polling loop: continuously check state of pushbutton, change LED when pushed
2. Write program with button (hardware) interrupt: button press triggers change to next LED

## Interrupts

- We want to write programs that respond to **events** → things that happen that are out of the ordinary
- We want to have **timed activities** that wake from idle waiting
- Events: external inputs, alarms, exceptions (hardware and software)
- "**Interrupts**" – the hardware terminology

Interrupt on external signal

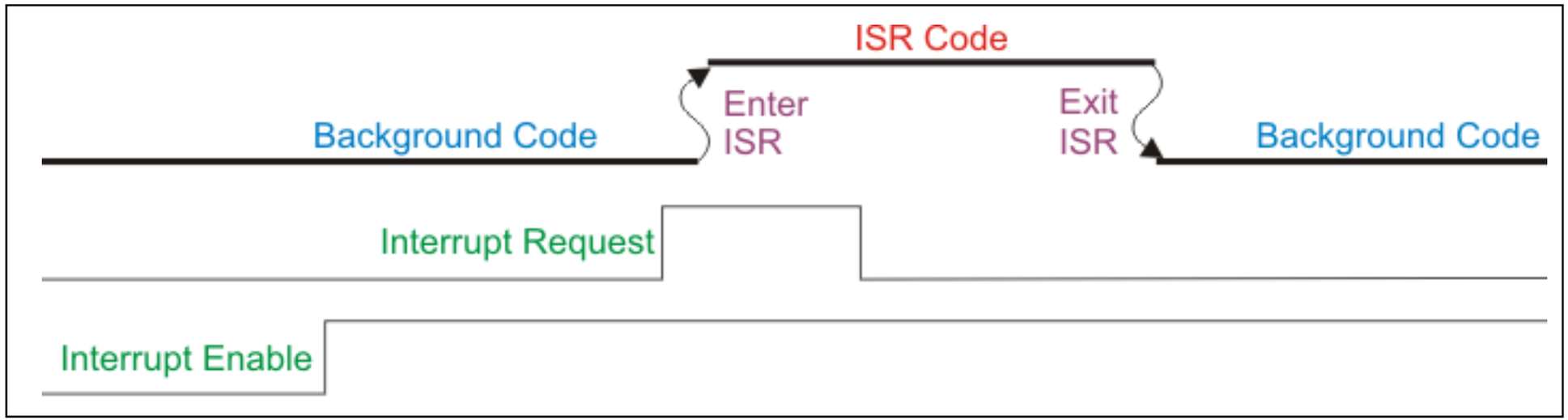
- Set up a GPIO pin to cause an interrupt

Notes:

- Can set to trigger on level, edge, rising, falling etc.
- Can resolve single or multiple triggers

# Interrupts

- A signal to the CPU to immediately stop what it is doing and start something else – **interrupt request** (IRQ)
- Vector to **interrupt service routine** (ISR)
- Current **program state pushed to stack**
- ISR typically very short – set a flag and return (pull saved state off stack)
- Examples – timer alarm, GPIO input (button push), packet arrival etc.




From https://www.embeddedrelated.com/

# Example (GPIO event)

Flag to signal from ISR

```
#include XYZ                                  // Your includes
#define XYZ                                  // Your defines
int flag = 0;                                // Flag for signaling from ISR
```

ISR in instruction RAM

```
static void IRAM_ATTR gpio_isr_handler(void* arg)  // Interrupt handler for your GPIO in IRAM
{
        flag ^= 1;                           // Toggle state of flag
}
```

Main sets up ISR, GPIO, etc.

```
void app_main() {                            // Your main program
    gpio_...                                 // Setup GPIO parameters
                                             // Setup GPIO to interrupt on pos edge
    gpio_intr_enable();                      // Enable interrupt on IO pin
    gpio_install_isr_service(ESP_INTR_FLAG_LEVEL3);          // Install ISR
    gpio_isr_handler_add();                  // Hook ISR to GPIO pin
```

Loops testing flag; delay allows access to other tasks

```
    while(1){                                // Forever
        if flag = 1 {
            gpio_set_level(PIN,flag};        // Set an LED to state of flag, when the button is pushed
            vTaskDelay(100);                 // Delay a bit, as there is nothing else to do
           }
}
```

This example is a design pattern

# Interrupts – other considerations

- Typically you need to disable interrupts while servicing an interrupt – to prevent retriggering or potential conflict with other interrupts
- Interrupts can conflict with writing to Flash memory (which requires specific timing)
- Potentially need to deal with **mutual exclusion** issues – prevent ‘race conditions’ where order of execution matters
- Many ways to set this up – keep it simple – use ESP code base
- Example ESP GPIO interrupts: https://github.com/espressif/esp-idf/tree/master/examples/peripherals/gpio/generic_gpio
- Example code here: https://github.com/BU-EC444/04-Code-Examples/tree/main/button-timer
- Used global variable in prior example. Other examples use a queue to pass data from the ISR and the main program or task