LECTURE VII

Timers and Interrupts

T-Shirt Handout!

- **OPS T-Shirts have arrived!**
  - Pick up your t-shirt at the front from an instructor
- If you arrived late, you can pick yours up after lecture is over!
- T-shirt pickup will also be available during normal lab hours or after this lecture

Would You Rather?

Be on the Titanic on its last cruise

or

Eat an apple

Would You Rather?

or

SECTION I

Project VI Review

Project 6 Review

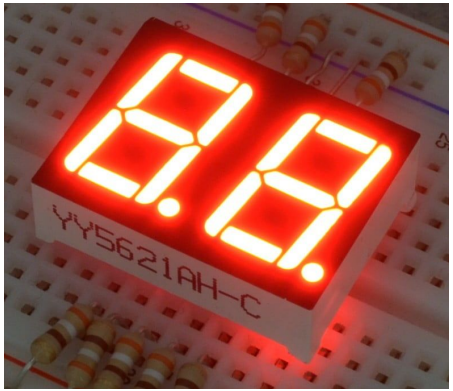
- Build a **weather station** that wirelessly transmits **temperature** and **humidity data** to an indoor display.
- Due date: 2/9/2026 at 11:59PM

Learning Concepts:

- Communication Protocols Pt. 2
- I^2C Protocol
- SPI Protocol
- Arduino IDE Libraries (Continued)

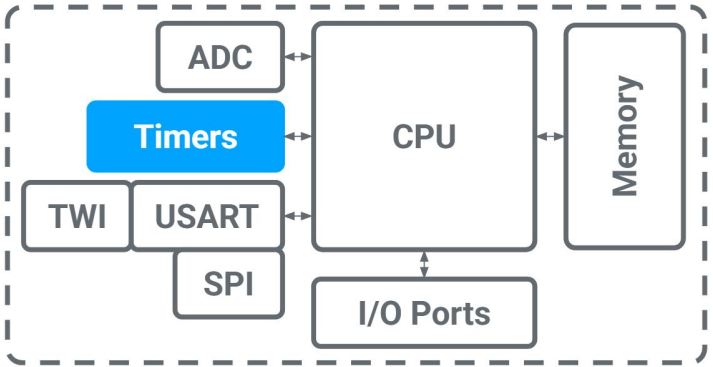
Project 7

- Create a **digital stopwatch** using interrupts, timers, and a 7-segment display
- Due date: 2/23/2026 at 11:59PM



Learning Concepts:

- Interrupts
- Timers
- Arduino IDE Libraries (Continued)



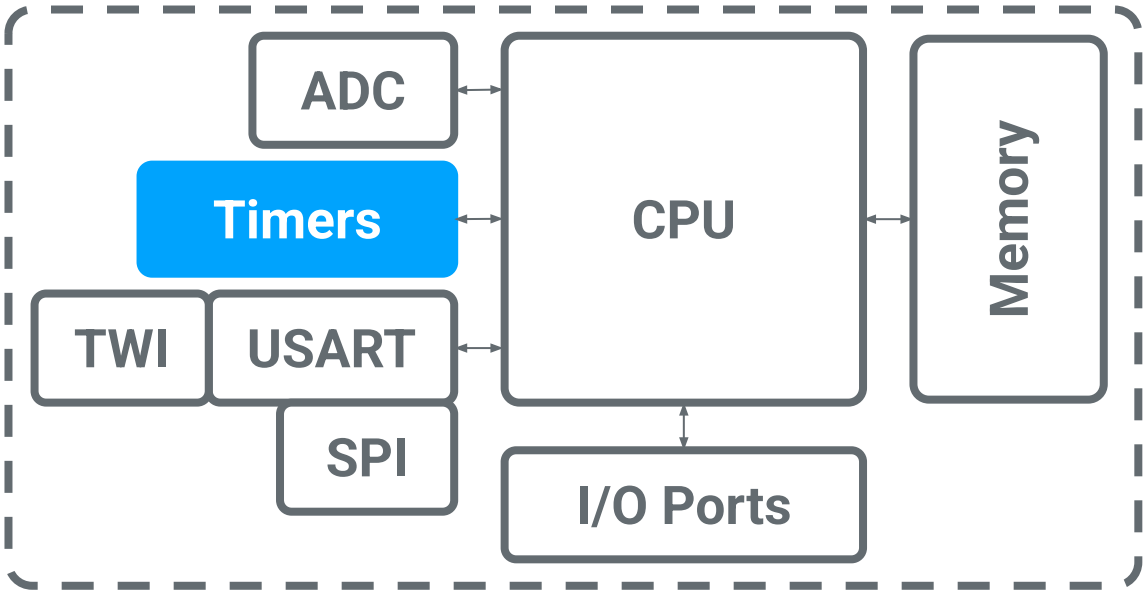
IEEE Board Applications

- IEEE board applications are now open!
- If you have enjoyed OPS or are interested in gaining leadership experience in a teaching role, look no further!
- We urge those of you with a passion for learning to apply for either **OPS Lab Supervisor** or **OPS Lab Instructor**
- Applications are due **February 19th at 11:59PM**

SECTION II

Timers

Timers



Timers are used by the microcontroller to **control the timing** of program execution or output signals and to **measure time**.

Types of Timers

- Timer types can be distinguished by when the **timeout** triggers
  - A **timeout** is an event that occurs after a preset period of time
- A **one-shot timer** has a single timeout
  - For the timer to be run again, it must be **reloaded manually**
  - Ex) A countdown timer runs just once, stopping when it reaches zero
- A **periodic** (or **auto-reload**) **timer** has *periodic* timeouts that occur at a fixed interval
  - The timer **reloads automatically**
  - Ex) An alarm clock rings once every morning

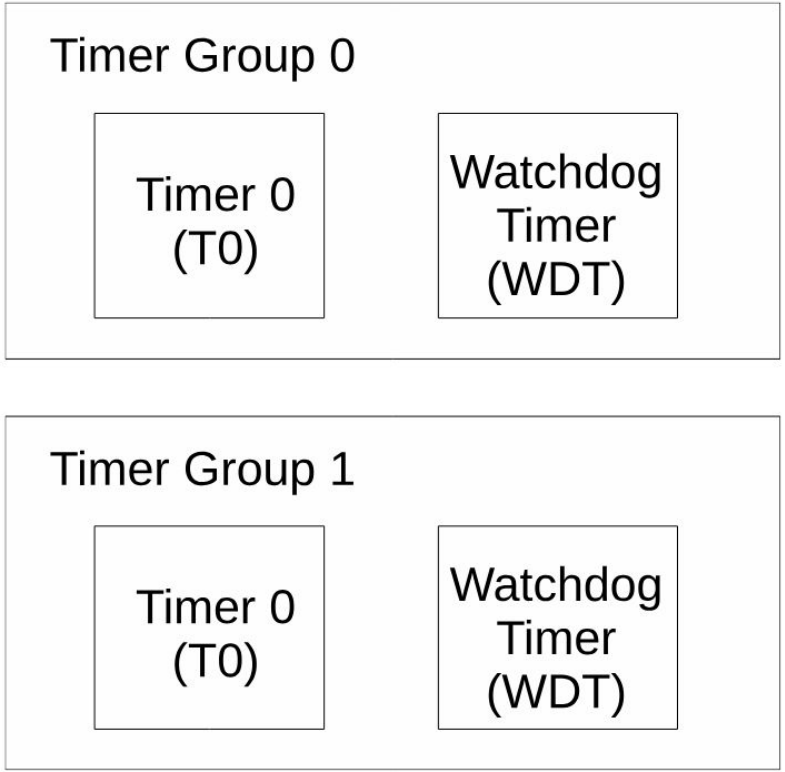
Timer Use Cases

- **Pausing/delaying the program**
  - `delay` uses a timer to suspend code execution for a specific period of time (Arduino)
  - This is considered a **blocking** timer function since it *blocks* program execution
- **Measuring time**
  - `millis` uses a timer to count the number of milliseconds since the program began (Arduino)
  - This is a **non-blocking** timer function since the program continues to execute as the timer runs

Timer Use Cases (Cont'd)

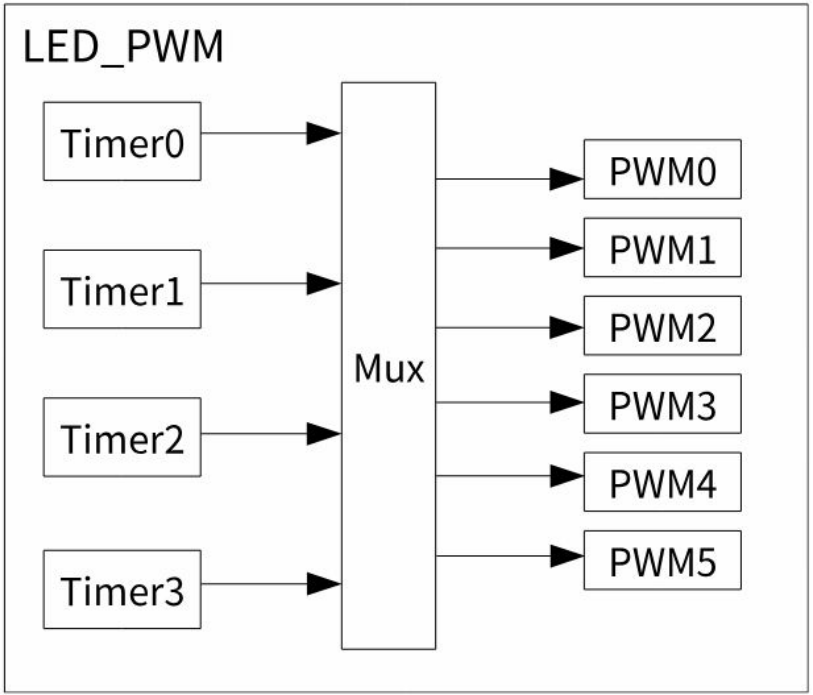
- **Calling a function upon timeout**
  - When the timer suspends, we may want specific instructions to occur. In which case, we call a function to *handle* the event
  - We'll discuss this use case in the **Interrupts** section…
- **Generating a PWM signal**
  - `analogWrite` uses a timer to time the rising and falling edges of the PWM waveform (Arduino)
  - See more in the Microcontroller Architecture and Arduino lecture…

ESP32C3 Timers



- The ESP32c3 has two general purpose timers (**GPTs**)
  - These timers are modules in the MCU used for various functions, PWM, and delays
  - Digital pins are capable of carrying the output of the timer to which they are connected
    - A pin used for timer output cannot also be used for a different purpose at the same time
  - There are more timers than this for other functions!

ESP32C3 Timers (Cont'd)

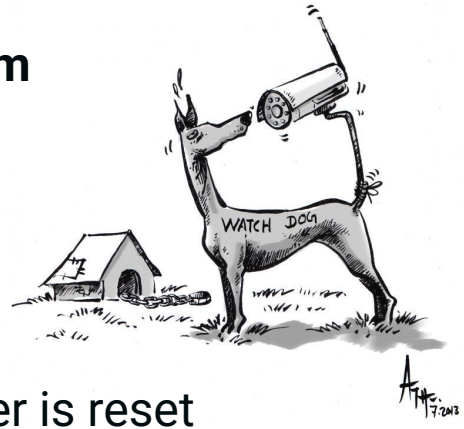


Some other timers

- System timer
  - 52 bit timer
  - Used for interrupts
  - Used by `delay` and `millis`
- LED_PWM
  - For duty cycle and frequency
  - 16-bit resolution

ESP32C3 Timers (Cont'd)

- A **watchdog timer** (**WDT**) is used to **detect and recover from** software or hardware **errors**
  - How it works:
    i. The WDT counts down to zero
    ii. Each time the program's main loop is run, the timer is reset
    iii. If the WDT reaches zero before its reset, then the WDT resets the entire processor
  - This is **not a perfect solution**! If a program loop takes a long time to execute, the WDT incorrectly assumes a fault, and the program is reset



SECTION III

Arduino Timer Library

Arduino Timer Library

- We will use one of the Arduino's internal timers to **measure time**
  - To do so, we will enlist the help of the Timer library, which will need to be downloaded and installed to the Arduino IDE
  - Make sure to use `#include <Timer.h>`
- If you would like to explore how timers can be used to call other functions, check out the TimerInterrupt library
  - We will not discuss it in this lecture

Arduino Timer Library (Cont'd)

- First, instantiate a `Timer` object as `Timer timer;`
  - Each of the following are member functions that can be called from this object
  - Time is **measured in milliseconds** by default
    - Alternatively, you can pass `MICROS` as a single argument to the constructor to measure in microseconds
- `timer.start()` starts the timer
  - The timer is in the `RUNNING` state

Arduino Timer Library (Cont'd)

- `timer.pause()` pauses the timer
  - The timer is in the `PAUSED` state
  - The time elapsed since `timer.start()` freezes
- `timer.resume()` resumes the timer from the `PAUSED` state
  - The timer is in the `RUNNING` state
  - The time elapsed since `timer.start()` resumes counting

Arduino Timer Library (Cont'd)

- `timer.stop()` stops the timer
  - The timer is in the `STOPPED` state
  - The time elapsed since `timer.start()` will be the reset the next time the `timer.start()` is called
- `timer.read()` returns the time elapsed since `timer.start()`
  - Measured in milliseconds
- `timer.state()` returns the state as one of three predefined constants
  - The state can be `RUNNING`, `PAUSED`, or `STOPPED`

SECTION IV

Interrupts

Interrupts

- An **interrupt** is a request for the CPU to **halt the currently executing code** when an event occurs
  - The CPU suspends the current program to *handle* the event by executing a function called an **interrupt handler** or **interrupt service routine** (**ISR**)
  - When an interrupt halts the current code, a separate program / function can be called and run.
  - When the interrupt handler finishes execution, the CPU returns to the old program

Polling

- **Polling** is a method of periodically checking the status of a device
  - Ex) A person checking their phone for new messages every 10 minutes
    - We have been using polling in the `void loop()`
  - Polling loops cost additional CPU time
- Should we use interrupts instead of polling?
  - Interrupts signal *when* an event occurs whereas polling *checks* to see if an event occured
    - Ex) A person reading their phone upon receiving a push notification
  - **Use interrupts when an event is urgent and/or infrequent**

Types of Interrupts

- A **software interrupt** is triggered by an instruction
  - Also called a **trap** or **exception**
  - Often caused by an illegal operation
    - Ex) Dividing by zero, Accessing memory without permission
- A **hardware interrupt** is triggered by devices external to the CPU
  - Also called an **external interrupt**
  - Caused when a devices sends an **interrupt request** (**IRQ**) signal to the CPU
    - Ex) Timer timeout, button press, power failure

Interrupt Use Cases

- **Servicing timers**
  - When a timeout occurs, the programmer may want specific code to be run
    - This is when the timer sends an IRQ and an interrupt handler is called
- **Handling peripheral device events**
  - Ex) When a key is pressed, the keyboard sends an IRQ to the computer
    - An interrupt handler then responds to the event

SECTION V

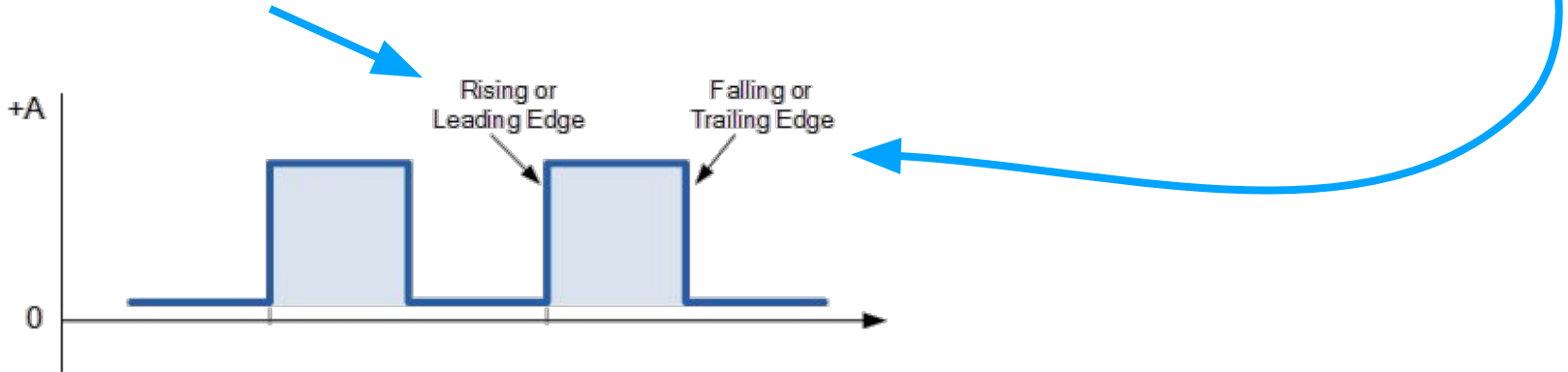
Interrupts with Arduino

attachInterrupt

- Any GPIO pin on the ESP32 can be an interrupt pin
- `void attachInterrupt(interrupt, ISR, mode)`
  - This function initializes an interrupt
  - *`interrupt`* - the interrupt number; only one interrupt can be initialized per interrupt number
    - The interrupt # should be the same for ESP32, but this is not always true.
  - *`ISR`* - the name of the function to call as the interrupt handler
    - You need to define a function and pass its name as *`ISR`*
  - *`mode`* - configures the timing of the IRQ
    - You will set the mode to one of the following constants…

attachInterrupt (Cont'd)

- The options for ***mode*** are…
  - **LOW** - when the pin is a LOW voltage
  - **CHANGE** - when the pin changes value
  - **FALLING** - when the pin voltage changes from HIGH to LOW
  - **RISING** - when the pin voltage changes from LOW to HIGH



`attachInterrupt` (Cont'd)

- `void digitalPinToInterrupt(pin)`
  - This function converts a digital pin number to its corresponding interrupt number
- It is **recommended that you call `attachInterrupt` as follows:**
  - `attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)`
  - This approach avoids confusion involving digital pin and interrupt numbers

SECTION VI

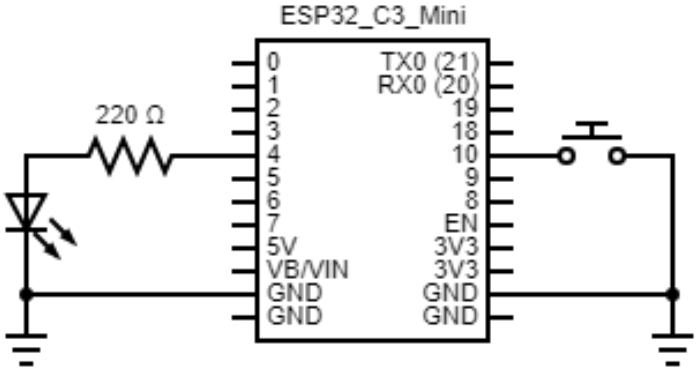
ESP32 Interrupt Exercise

ESP32 LED Toggle Button

I/A

Write a program that satisfies the following requirements:

- You must write a program in which the **ESP32 board toggles the state of its LED (on pin 4) when a button is pressed**
- The state of the LED must toggle only once for each button press
    - Meaning… The LED should not flicker ON and OFF
      if the button is held down
- This program must be implemented with `attachInterrupt`
- No loops may be used (that includes the main `loop`)
- No starter code!



FAIR USE DISCLAIMER

Copyright Disclaimer under section 107 of the Copyright Act 1976, allowance is made for “fair use” for purposes such as criticism, comment, news reporting, teaching, scholarship, education and research.

Fair use is a use permitted by copyright statute that might otherwise be infringing.

Non-profit, educational or personal use tips the balance in favor of fair use.

CC BY-NC-SA 4.0