MCU

Interrupts and PWM

{Learn, Create, Innovate};

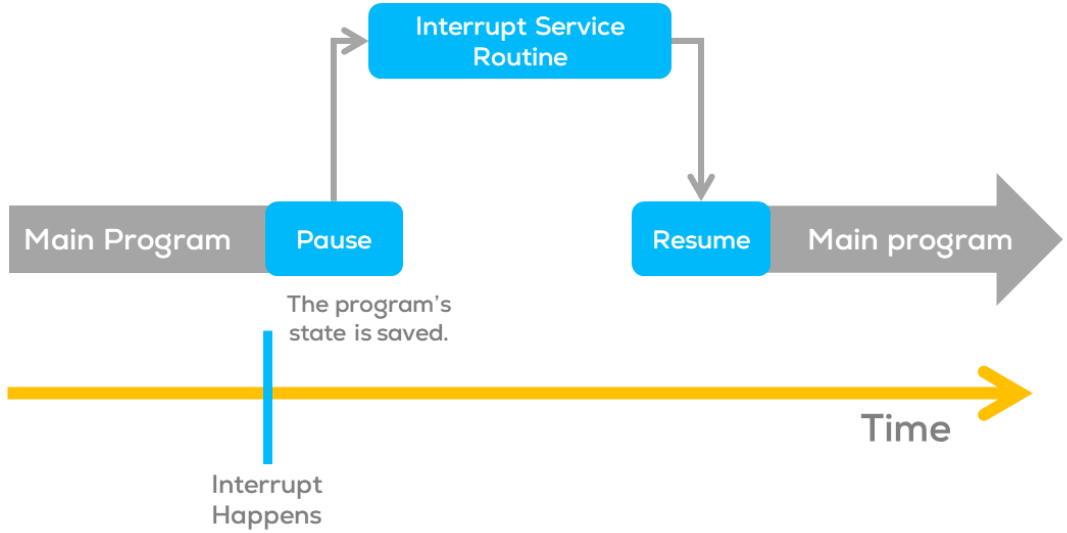
Interrupts

Arduino

Interrupts

Interrupts

- Interrupts are signals that temporarily pause the main program to handle specific events.
- Such signals can come from external hardware or internal processes to the MCU.
- These signals are typically called triggers.
- Interrupts allow the microcontroller to respond immediately when an event occurs.



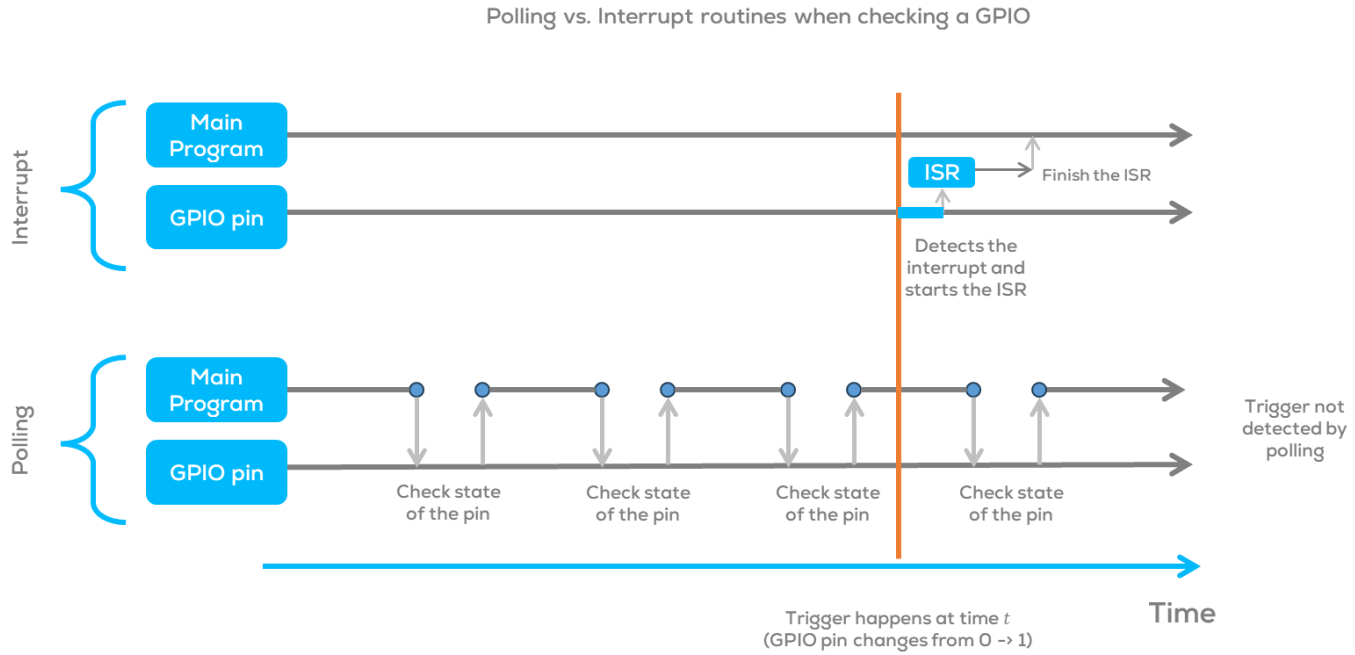
Interrupts

Advantages of Interrupts

- Multitasking capability: Handle multiple tasks simultaneously.
- Faster response time: Immediate action upon event detection.
- Lower power consumption: The microcontroller can sleep between interrupts.
- Precise timing: Ideal for time-critical tasks.
- Simplified code: No need for constant event checking.
- Compared with polling, where the user must continuously monitor the state of a variable or pin, interrupts respond to an event upon detection of the trigger signal.

Uses

- Interrupts are useful when solving timing problems.
- Some application examples may include reading a rotary encoder or monitoring user input (button, etc.).



Interrupts

Types of interrupts

- **External Interrupts**
  - Triggered by external events, falling or rising edges; such as button presses or changes in sensor signals.
  - Ideal for real-time event-driven applications such as button presses, sensor readings, and rotary encoders.
- **Timer Interrupts**
  - Generated by internal timers at specific intervals.
  - Useful for precise timing, generating PWM signals, real-time clock, controlling stepper motors, and time-dependent tasks.
- **Pin Change Interrupts**
  - Triggered when the logic level changes on certain pins.
  - Suitable for scenarios with multiple pins changing simultaneously, like keypad inputs.
- **Special Interrupts**
  - Reset and watchdog timer interrupts.
  - Essential for system recovery and managing system stability.

Interrupts

Interrupt Modes (Arduino and ESP32)

- **LOW** to trigger the interrupt whenever the pin is low,
- **CHANGE** to trigger the interrupt whenever the pin changes value
- **RISING** to trigger when the pin goes from low to high,
- **FALLING** for when the pin goes from high to low

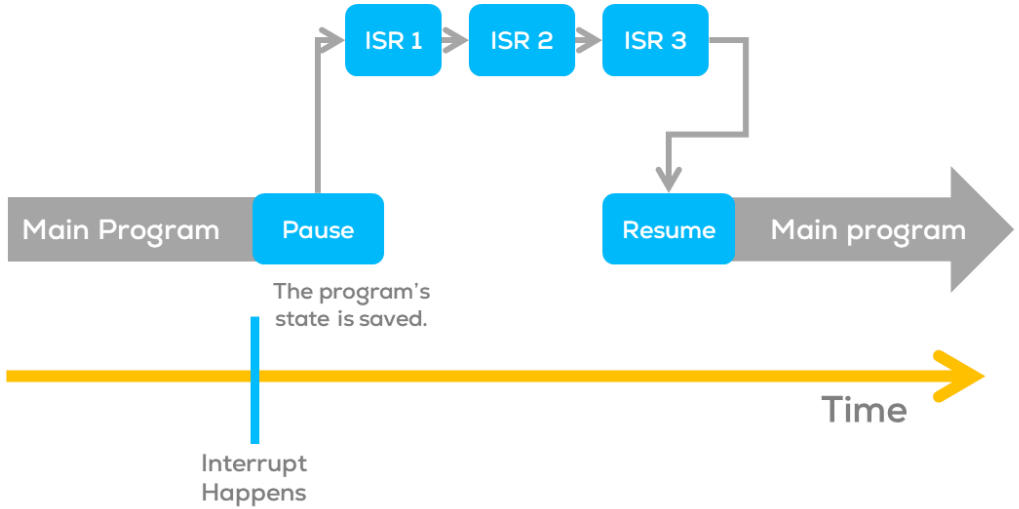
Interrupts

ISR (Interrupt Service Routines)

- Special functions that handle interrupts, allowing the microcontroller to respond immediately to events.
- ISR cannot have any parameters and shouldn't return anything.
- When an interrupt occurs, the microcontroller pauses the main program and jumps to the corresponding ISR.
- ISR executes, performs the necessary actions, and then returns to the main loop.

Warning:

- In Arduino, the functions *millis(), delay()* and *micros()* rely on interrupts to count, so they will never increment inside an ISR.
- When multiple ISRs are used, only one can run at a time. After the current one finishes, the rest of the interrupts will be executed in an order that depends on their priority. More information about priority can be found here.



Interrupts

Flags

- Interrupts use flags for different purposes; in Arduino and ESP the most common flags used are
  - **interrupts():** Re-enables interrupts (after they've been disabled.
    - Interrupts allow specific tasks to happen in the background and are enabled by default.
  - **nointerrupts():** Disables interrupts (you can re-enable them with interrupts()).
    - Some functions will not work while interrupts are disabled, and incoming communication may be ignored.
    - Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical code sections.

Variables

- Typically, global variables are used to pass data between an ISR and the main program.
- The variables used in an ISR must be declared as *volatile.*
  - Declaring variables as volatile, tells the compiler that such variables might change at any time, and thus the compiler must reload the variable whenever you reference it, instead of copying the value.

    volatile int var;

    volatile boolean flag;

Interrupts: Arduino Example

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);

attachInterrupt(digitalPinToInterrupt(interruptPin),
blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() {
  state = !state;
}
```

- This example expects an external interrupt given by a switch connected to pin 2 (Interrupt 0) of the Arduino Mega and changes the state of the built-in LED in Pin 13.
  - Normally, you should use digitalPinToInterrupt(pin), rather than place an interrupt number directly into your sketch. The specific pins with interrupts and their mapping to interrupt number varies for each type of board.

BOARD	INT.0	INT.1	INT.2	INT.3	INT.4	INT.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
32u4 based (e.g Leonardo, Micro)	3	2	0	1	7	

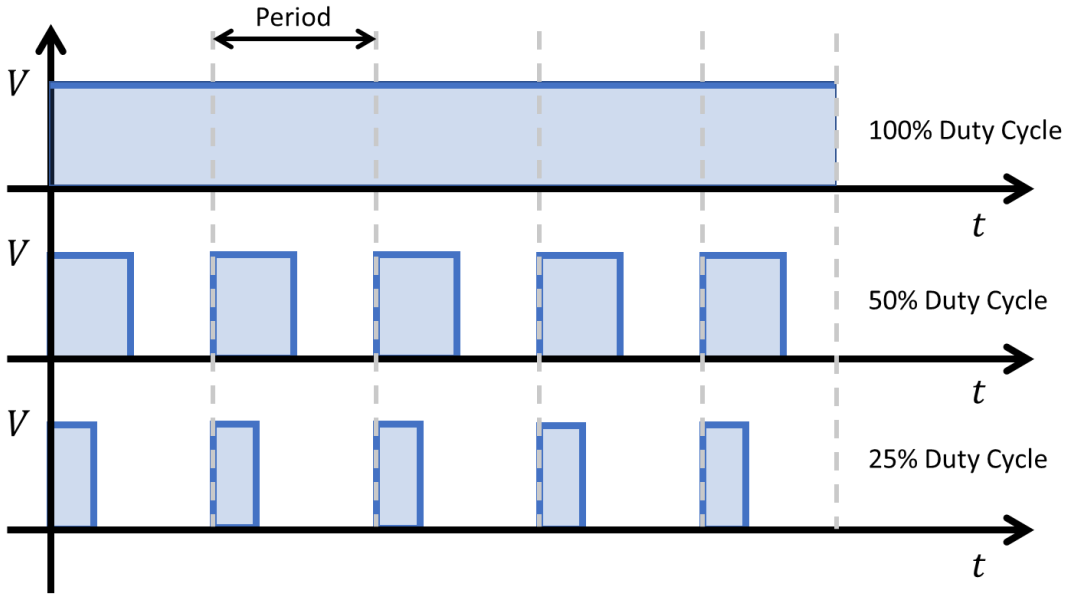
PWM Signals

Arduino

PWM

What is PWM?

- PWM stands for Pulse Width Modulation.
- It's a technique used in electronics to control the average voltage or current delivered to a device by rapidly switching between full power and no power (On/Off) over a fixed period of time.
- This creates an average voltage or current somewhere in between, effectively controlling the output (power delivered to the system).
- The duration of "on time" is called the pulse width.
- To get varying analog voltage values, the pulse width can be changed “modulated”.



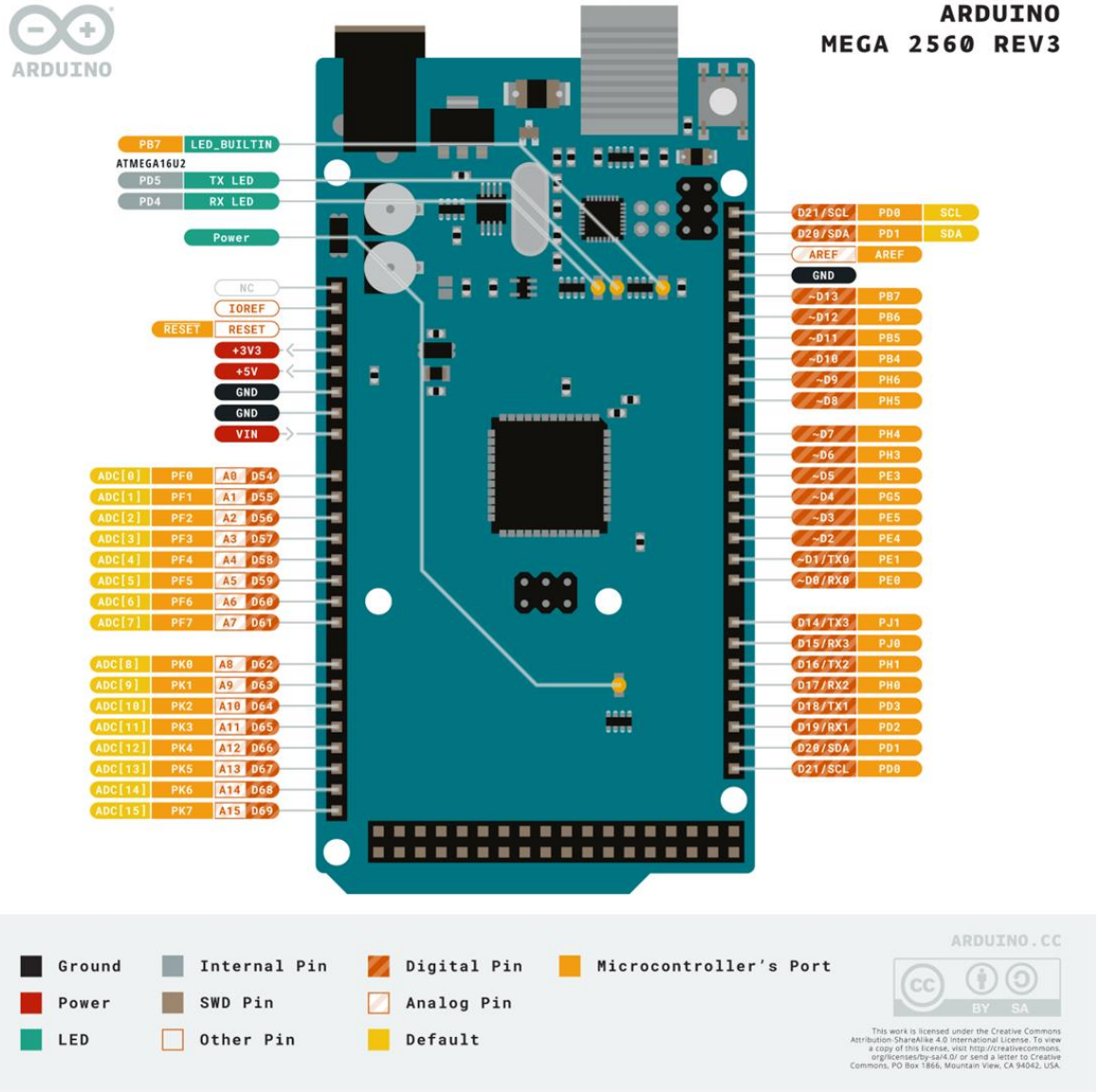
PWM

Why Use PWM?

- PWM is commonly used to control things like motor speed, LED brightness, and even audio signals.
- It's an efficient way to simulate varying levels of output using digital control, like from a microcontroller such as the Arduino.

Arduino PWM

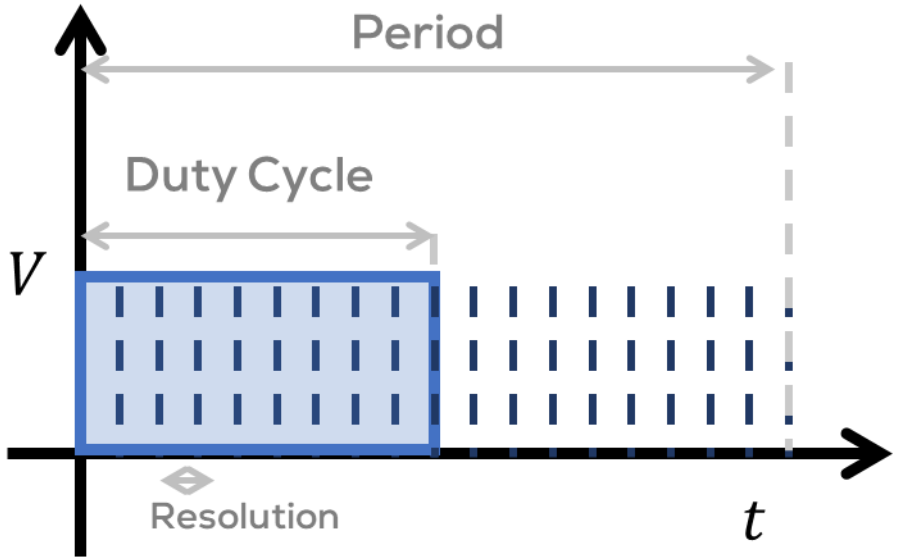
- Many Arduino boards have pins that are capable of generating PWM signals. These pins are usually marked with a tilde (~) symbol on the board (e.g., ~3, ~5, ~6, etc.).



Arduino PWM

Arduino PWM

- For most Arduino boards, the default frequency is around 490 Hz.
- PWM resolution is the number of distinct levels or steps that a PWM signal can have within its duty cycle range. In other words, is the granularity with which the duty cycle can be modulated.
- The Arduino PWM, has a resolution of 8 bits that is $2^8 = 256$ values.
    - A call to analogWrite() is on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle



Arduino PWM

Generating PWM with Arduino:

- AnalogWrite Function: In Arduino, the analogWrite(pin, value) function is used to create a PWM signal. The pin parameter is the pin number capable of PWM, and the value parameter is a number between 0 (off) and 255 (full on), representing the duty cycle of the PWM signal. Duty cycle refers to the percentage of time the signal is on compared to the total period.

```
// Define the PWM output Pin.
const int PWMPin = 13;

void setup() {
    pinMode(PWMPin, OUTPUT);          // Setup pins as outputs.
}

void loop() {
    // Increase power to LED from Off state to brightest
    for (int pwmVal = 0; pwmVal < 255; pwmVal++) {
      analogWrite(PWMPin, pwmVal);  // Write the PWM value to the output
      delay(20);
    }
    // Decrease power to the LED from full power to off state
    for (int pwmVal = 255; pwmVal >= 0; pwmVal--) {
      analogWrite(PWMPin, pwmVal);  // Write the PWM value to the output
      delay(20);
    }
    delay(100);
}
```