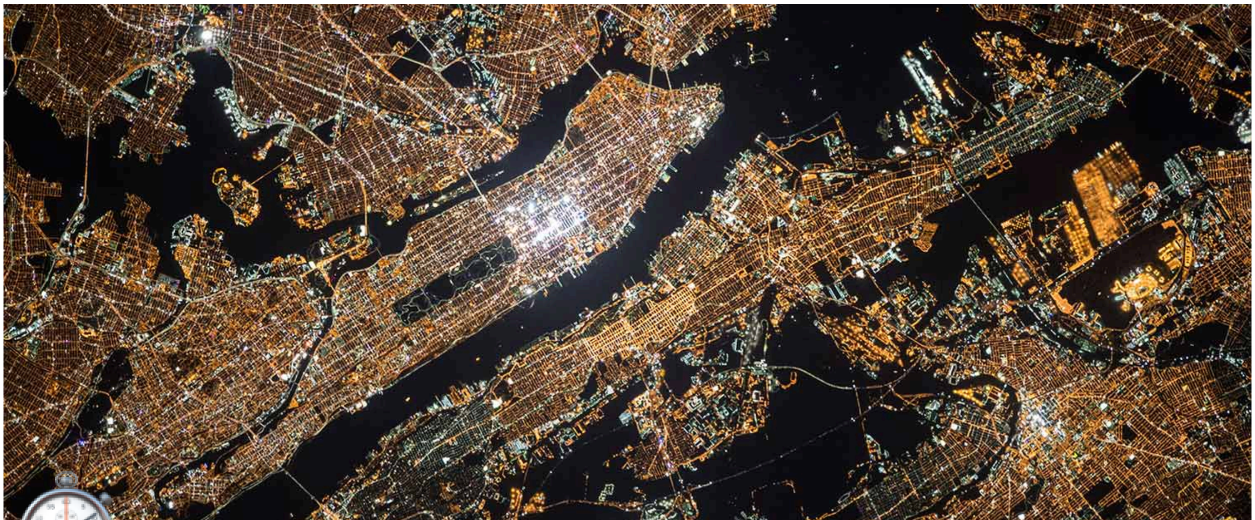


Week 6 - Timer Ports and PWM

Timers are based on counters. So for example, 8-bit timer has a max value of 255.

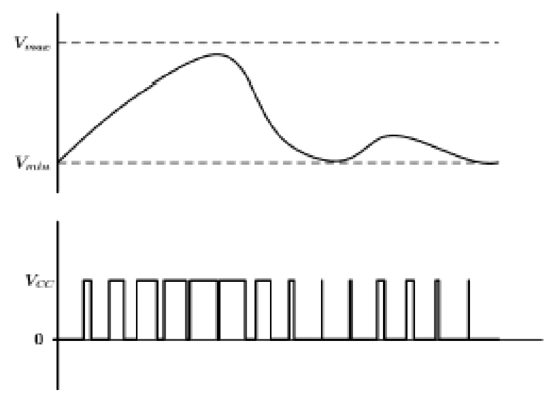
Pulse Width Modulation

- Allows transmission of analog signals digitally by rapidly switching the output pin between HIGH and LOW states.
- Duty cycle = percentage of ON time, which decides the amplitude i.e. the average voltage

$$\textit{Duty cycle}, D = \frac{\textit{High duration in a single period}}{\textit{Period}}$$

= Percentage of ON time

$$D = \frac{V_{sig} - V_{min}}{V_{max} - V_{min}}$$



In ATmega328p

- Has 3 counters with 2 output channels each. (A and B)
- Two PWM Modes:
    - **Fast PWM**
        - Timer increments value with every clock cycle.
        - When it reaches max value (say 255) it resets to 0.
        - The time it takes for the counter to count from 0 up to its maximum and reset constitutes one complete **period (T)** of the PWM signal. The frequency of the PWM is determined by the clock speed and the timer's maximum count value.
        - OCR (Output Compare Register) holds a value that is used to constantly compare the timer value.
        - While timer value < OCR, output pin = HIGH, else LOW.
        - Thus, the OCR value determines the duty cycle. If its max (255) then duty cycle is 100%.
        - Frequency of Pwm = Frequency of clock/ max value
    - Phase-correct PWM
        - The timer counts up to max value then counts down to 0. So, the N is doubled, thus frequency gets halved from Fast PWM.

Timers on Arduino

Timer	Default Arduino setting		Effective PWM frequency
	Mode:	**Fast PWM**	
timer0	Top:	255	976.56 Hz
	Pre-scaler:	64	
	Mode:	**Phase-correct (only 8-bit used)**	
timer1 (16-bit)	Top:	255	490.20 Hz
	Pre-scaler:	64	
	Mode:	**Phase-correct**	
timer2	Top:	255	490.20 Hz
	Pre-scaler:	64	

Note:

- PWM can also be generated by software using the CPU on any GPIO pin (as opposed to only using hardware only on specific pins)

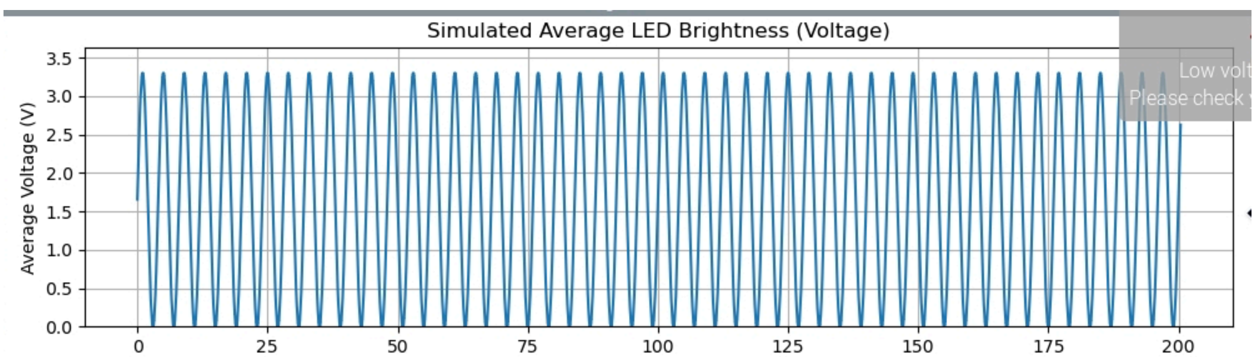
Demodulation

- Process of converting PWM to a true analog signal.
    - can be done with a low pass (RC) filter.
    - the cutoff frequency of the filter has to be more than the frequency of the pwm.

Raspberry Pi 3

- Has two dedicated hardware PWM module with two channels. PWM0 and PWM1
    - the modules have their own internal counters and compare mechanisms (same principle but implemented differently)
    - the channels can be routed to several GPIO pins

PWM Example - Fading LED with Sinusoidal Signal




```
import pigpio
import time
import math
import matplotlib.pyplot as plt
import numpy as np # Import numpy for array operations

# --- pigpio setup for PWM output ---
pi = pigpio.pi()
```

```
if not pi.connected:
    print("Failed to connect to pigpiod daemon. Is it running? Try 'sudo systemctl s
    exit()

# Define the GPIO pin connected to the LED (e.g., GPIO12)
# This pin MUST support hardware PWM on your Raspberry Pi.
# Common pins are GPIO12 (PWM0) or GPIO13 (PWM1).
pwm_gpio = 12

# Define the range of duty cycle values (like the ATmega's TOP value + 1)
# A range of 255 gives 256 steps (0 to 255), similar to 8-bit resolution.
# A higher range gives finer control (higher resolution).
pwm_range = 255 # Max duty cycle value will be 255 for 100%

# Define the base frequency of the PWM signal in Hz
# A higher frequency generally results in smoother dimming and is less visible fli
pwm_frequency = 1000 # Hz (1kHz)

# Configure the GPIO pin for output mode and set the PWM frequency and range
pi.set_mode(pwm_gpio, pigpio.OUTPUT)
actual_freq = pi.set_PWM_frequency(pwm_gpio, pwm_frequency)
actual_range = pi.set_PWM_range(pwm_gpio, pwm_range)

print(f"Configured PWM on GPIO{pwm_gpio}:")
print(f"  Requested Frequency: {pwm_frequency}Hz (Actual: {actual_freq}Hz)")
print(f"  Requested Range: {pwm_range} (Actual: {actual_range})")
print(f"  Possible Duty Cycle Steps: {actual_range + 1} (0 to {actual_range})")
print(f"  Each step is approx {(100 / actual_range) if actual_range > 0 else 0:.2f}%


# --- Fading parameters ---
fade_period_sec = 4 # Time for one full sine wave cycle (fade up and down)
max_gpio_voltage = 3.3 # Typical voltage for Raspberry Pi GPIO HIGH state

# Data lists for plotting (collect data while fading)
time_data = []
```

```
duty_value_data = []
analog_voltage_data = []
duty_percent_data = []

# Verbosity control
print_interval = 0.2 # seconds - print status update every this many seconds

print(f"\nAuto-fading LED on GPIO{pwm_gpio} over {fade_period_sec} seconds."
print("Press Ctrl+C to stop the script and show plots.")

start_time = time.time()
last_print_time = start_time

try:
    while True:
        current_time = time.time() - start_time

        # Calculate the position in the sine wave cycle (0 to 2*pi)
        # The pattern repeats every fade_period_sec
        angle = (current_time / fade_period_sec) * (2 * math.pi)

        # Calculate the sine value (ranges from -1 to 1)
        sin_value = math.sin(angle)

        # Map sine value (-1 to 1) to PWM duty cycle range (0 to pwm_range)
        # We want sin(-pi/2) to be 0% duty (value 0) and sin(pi/2) to be 100% duty
        # The sin wave goes from -1 to 1. Add 1 to shift it to 0 to 2. Divide by 2 to sca
        # Then multiply by pwm_range.
        duty_cycle_float = ((sin_value + 1) / 2) * pwm_range
        duty_cycle_value = int(round(duty_cycle_float)) # Round to nearest integer v

        # Ensure value is within the valid range (0 to pwm_range)
        duty_cycle_value = max(0, min(pwm_range, duty_cycle_value))

        # Calculate average voltage and duty percentage for plotting and printing
        # Average voltage is proportional to the duty cycle percentage
```

```
        duty_percent = (duty_cycle_value / pwm_range) * 100 if pwm_range > 0 else
        avg_voltage = (duty_cycle_value / pwm_range) * max_gpio_voltage if pwm_ra

        # Store data for plotting
        time_data.append(current_time)
        duty_value_data.append(duty_cycle_value)
        analog_voltage_data.append(avg_voltage)
        duty_percent_data.append(duty_percent)


        # Verbose printing (only print if enough time has passed)
        if current_time - last_print_time >= print_interval:
            print(f"Time: {current_time:.2f}s | Duty Value: {duty_cycle_value}/{pwm_ra
            last_print_time = current_time

        # Set the PWM duty cycle using pigpio
        pi.set_PWM_dutycycle(pwm_gpio, duty_cycle_value)

        # Small delay to control the rate of updates and data collection
        # Adjust this for smoother fading vs. faster data collection
        time.sleep(0.01) # Example: update 100 times per second

except KeyboardInterrupt:
    print("\nStopping...")

finally:
    # --- Cleanup ---
    pi.set_PWM_dutycycle(pwm_gpio, 0) # Set duty cycle to 0 (turn off LED)
    pi.stop() # Disconnect from pigpio daemon
    print("Cleanup complete. Disconnected from pigpiod.")

    # --- Plotting ---
    print("Generating plots...")

    if not time_data:
        print("No data collected to plot.")
```

```
else:
    # Create a figure and two subplots (one for average voltage, one for simulat
    fig, axs = plt.subplots(2, 1, figsize=(10, 8), sharex=False) # Don't share x-axi

    # Plot 1: Simulated Average LED Brightness (Voltage) vs Time
    axs[0].plot(time_data, analog_voltage_data)
    axs[0].set_xlabel("Time (s)")
    axs[0].set_ylabel("Average Voltage (V)")
    axs[0].set_title("Simulated Average LED Brightness (Voltage)")
    axs[0].grid(True)
    axs[0].set_ylim(0, max_gpio_voltage * 1.1) # Add some padding above max v

    # Plot 2: Simulated PWM Waveform (a few periods)
    # We'll simulate the PWM for a fixed duty cycle from the data,
    # for a duration covering a few PWM periods.

    # Choose a data point to represent the duty cycle for the simulation
    # Let's pick the duty cycle value that occurred around 1/4 of the way throug
    target_time_for_sim = fade_period_sec / 4
    # Find the index in the time_data list closest to our target time
    closest_index = min(range(len(time_data)), key=lambda i: abs(time_data[i] -
    sample_duty_value = duty_value_data[closest_index]
    sample_duty_percent = duty_percent_data[closest_index]

    # Calculate the duration of one PWM period
    pwm_period_sec = 1.0 / actual_freq if actual_freq > 0 else 0.001 # Avoid divi

    # Simulate a few periods (e.g., 5 periods)
    num_periods_to_show = 5
    # Generate time points for the simulation waveform over the desired duratio
    sim_duration_sec = pwm_period_sec * num_periods_to_show
    sim_time = np.linspace(0, sim_duration_sec, int(sim_duration_sec / (pwm_pe

    # Generate the simulated square wave based on the sample duty cycle
    sim_waveform = []
    if actual_range > 0:
```

```
        duty_duration_in_period_sec = (sample_duty_value / actual_range) * pwm_
        for t in sim_time:
            # Time within the current PWM period (using modulo)
            time_in_this_period = t % pwm_period_sec
            # If the time in the period is less than the HIGH duration, the output is H
            if time_in_this_period < duty_duration_in_period_sec:
                sim_waveform.append(max_gpio_voltage) # HIGH
            else:
                sim_waveform.append(0) # LOW
    else: # Handle case where range is 0
         sim_waveform = [0] * len(sim_time) # All zeros

    axs[1].plot(sim_time, sim_waveform, drawstyle='steps-post') # Use steps-po
    axs[1].set_xlabel("Time (s)")
    axs[1].set_ylabel("Voltage (V)")
    axs[1].set_title(f"Simulated PWM Waveform Sample ({actual_freq}Hz, ~{sam
    axs[1].set_ylim(-0.1, max_gpio_voltage * 1.2) # Add padding
    axs[1].grid(True)
    axs[1].axhline(y=0, color='k', linestyle='-', linewidth=0.5) # Add 0V line for c


    # Adjust layout and show the plots
    plt.tight_layout() # Automatically adjusts subplot params so that the subplot(
    plt.show()
```