UCSD CSE WES 237A

Winter 2026

Pulse Width Modulation Lab Report

(Due: 01 /21 /26)

Demonstration Date : 01 /21 /26 Group#: 7

Members: ***Benediction Bora*, *Gabriel Martinez***

INSTRUCTOR / TUTOR/

Nadir Weibel, Pushkal Mishra

Self-test Report Demo Reviewer

Name : ***Benediction Simeons***

Working / Not working Demo score Report score

Part1: _____ _____ _____: Executed Successfully

Part2: _____ _____ _____ : Executed Successfully

Part3: _____ _____ _____ : Executed Successfully

Part4: _____ _____ _____ : Executed Successfully

Part5: _____ _____ _____: Executed Successfully

Part6: _____ _____ _____: Executed Successfully

Video Demo Link: Video Demo Youtube

GitHub Repo: Asignment1 - WES237A GitHub Repo

TOTAL Score: _____

Pulse Width Modulation on RGB LED Using PYNQ-Z2

This report documents the design, implementation, and evaluation of a PWM RGB LED scheme implemented on the **PYNQ-Z2** board as part of WES 237A; Intro to Wireless Embedded Systems. The system integrates **MicroBlaze GPIO** control, Python **asyncio**-based task scheduling, push-button user interaction.

1. Objectives

The objectives of this assignment were to:

(1) generate PWM signals using C++ blocks and python in Jupyter Notebook
(2) determine an appropriate PWM frequency to avoid visible flicker,
(3) control perceived LED brightness via duty cycle adjustment, and
(4) design a responsive, asynchronous system that reacts to user button inputs in real time.

2. Design Methodology

A top-down design methodology was employed. The complete system was first divided into independent subsystems: low-level GPIO access, PWM signal generation, and user interaction. Each subsystem was implemented and validated independently within dedicated Jupyter notebook cells prior to full system integration. Asynchronous programming using asyncio was selected to enable concurrent LED blinking and button monitoring without blocking execution. A shared state dictionary was used to safely coordinate behavior between tasks.

3. Task and State Architecture

Figure below illustrates the interaction between asynchronous tasks and shared system state. The button task monitors user input and updates the system state, while the blinking task generates PWM-driven LED output based on the current state.

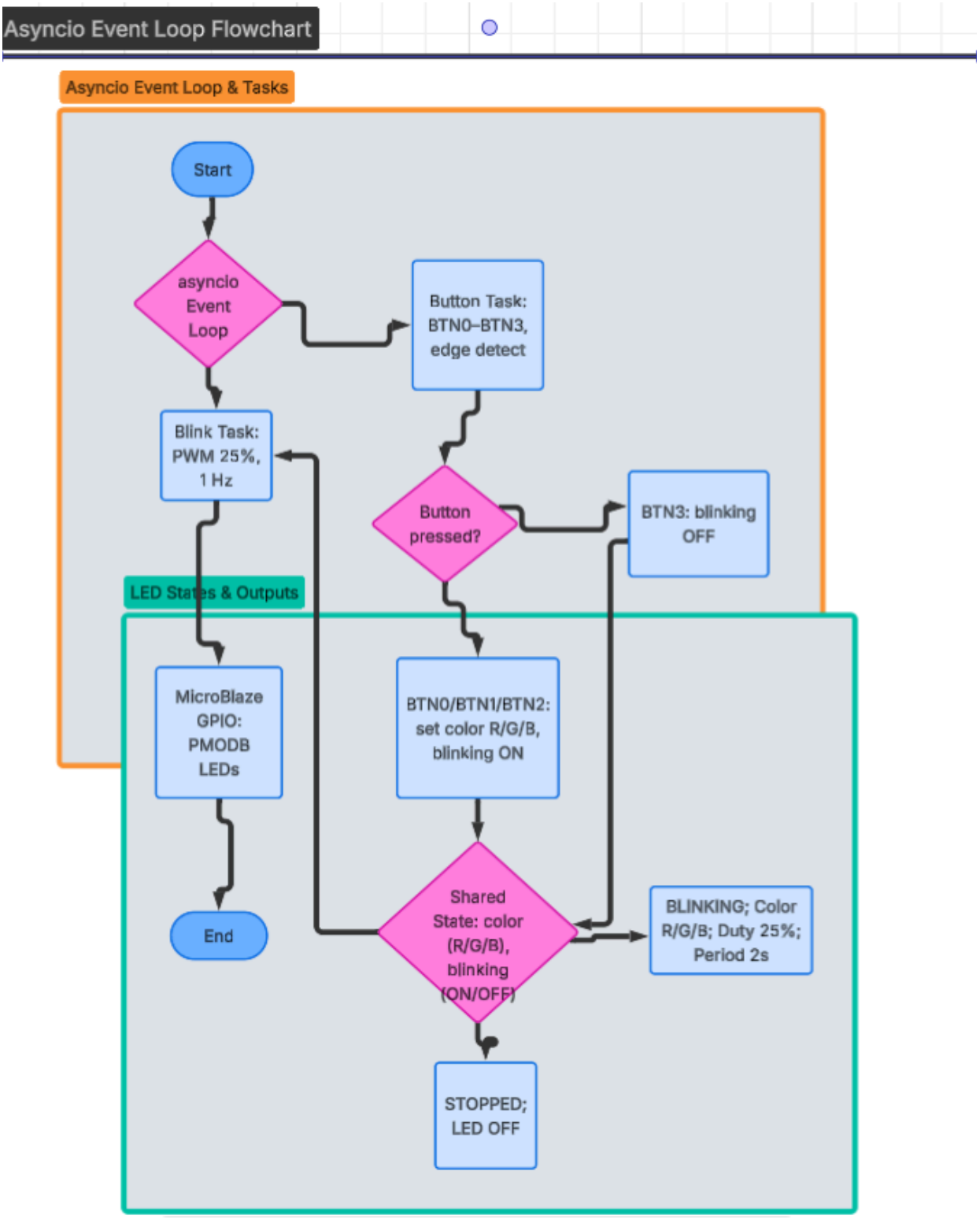


Figure (1) *System state and Asynchronous tasks*

4. Results and Observations

Experimental testing demonstrated that a PWM frequency of approximately 1 kHz eliminates visible flicker while remaining within software timing constraints. Duty cycles of 25%, 50%, 75%, and 100% produced progressively increasing brightness levels, with perceptual nonlinearity clearly observed.

```
Duty Cycle:   0%  →  Perceived Brightness:    0.0%
Duty Cycle:   5%  →  Perceived Brightness:   25.6%
Duty Cycle:  10%  →  Perceived Brightness:   35.1%
Duty Cycle:  25%  →  Perceived Brightness:   53.3%
Duty Cycle:  50%  →  Perceived Brightness:   73.0%
Duty Cycle:  75%  →  Perceived Brightness:   87.7%
Duty Cycle: 100%  →  Perceived Brightness:  100.0%
```

Figure(2) *Plot Data at a glance*

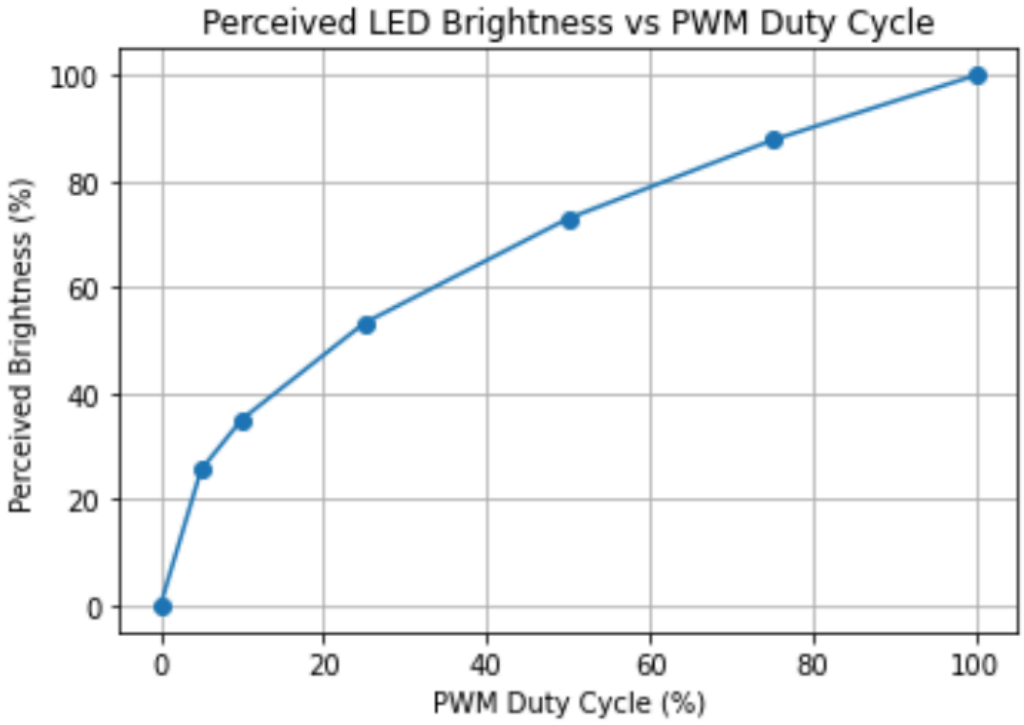


Figure (3) *Plot showing Perceived Brightness Vs PWM Duty Cycle*

The brightness curve above follows a gamma-corrected relationship **($\gamma \approx 2.2$)**, indicating that perceived brightness increases more slowly at low duty cycles and saturates at higher values.

5. Discussion and Troubleshooting

The primary implementation challenge involved improper coroutine invocation, which initially resulted in runtime warnings, stalled execution and the board consequently freezing due to being run for a long period of time. These issues were resolved by ensuring all coroutines were created and awaited or scheduled via asyncio in a main() routine within a single event loop.

6. Conclusion

This assignment successfully demonstrated software-based PWM generation, human-perceptual effects in LED brightness control, and asynchronous system design on an ZYNQ-Z2 board and the Jupyter Python3 embedded platform. The final implementation satisfies all functional requirements.

In [ ]:
```
from pynq.overlays.base import BaseOverlay
import asyncio
import time

# Load overlay
base = BaseOverlay("base.bit")
btns = base.btns_gpio
```

In [2]:
```
%%microblaze base.PMODB

#include "gpio.h"
#include "pyprintf.h"

void write_gpio(unsigned int pin, unsigned int val){
    gpio g = gpio_open(pin);
    gpio_set_direction(g, GPIO_OUT);
    gpio_write(g, val);
}

#// Reset all PMODB pins
void clear_pin(unsigned int pin){
    write_gpio(pin, 0);
}
```

In [69]:
```
def setPWMpin(pin, dutyCyle, frequency):
    pwm_frequency = frequency        # Vary until no flicker Hz
    duty_cycle = dutyCyle       # percent (0-100)
    run_time = 20.0              # seconds to run PWM

    period = 1.0 / pwm_frequency        # total period (seconds)
    on_time = period * (duty_cycle / 100.0)
    off_time = period - on_time

    # if duty cylce is less than or equal to zero
    # hold pin off
    if duty_cycle <= 0:
        write_gpio(pin, 0)
        time.sleep(run_time)

    # if duty cylce is less than or equal to 100
    # hold pin on
    elif duty_cycle >= 100:
        write_gpio(pin, 1)
        time.sleep(run_time)

    else:
        start_time = time.time()

        while (time.time() - start_time) < run_time:
            write_gpio(pin, 1)          # ON
            time.sleep(on_time)
            #time.sleep(1)
            write_gpio(pin, 0)         # OFF
            time.sleep(off_time)
            #time.sleep(1)
```

In [70]:
```
# Set the PWM on pin 3: RED
# Clear all pins
PIN_RED   = 3
PIN_GREEN = 2
PIN_BLUE  = 1

clear_pin(PIN_BLUE)
clear_pin(PIN_BLUE)
clear_pin(PIN_RED)
frequency = 10  # 10 times a second
setPWMpin(3, 0.25, frequency)
```

In [71]:
```
# Set the PWM on pin 2: GREEN
# Clear all pins
clear_pin(PIN_BLUE)
clear_pin(PIN_BLUE)
clear_pin(PIN_RED)
frequency = 100  # 100 times a second
setPWMpin(2, 0.75, frequency)
```

In [72]:
```
# Set the PWM on pin 1: BLUE
# Clear all pins
clear_pin(PIN_BLUE)
clear_pin(PIN_BLUE)
clear_pin(PIN_RED)
frequency = 1000  # 1000 times a second
setPWMpin(1, 1, frequency)
```

In [3]:
```
# CONFIGURATION
DUTY = 0.25          # 25%
BLINK_PERIOD = 2.0   # 1 sec ON + 1 sec  OFF

# PMODB RGB pin mapping
PIN_RED   = 3
PIN_GREEN = 2
PIN_BLUE  = 1


# SHARED STATE
state = {
    "color": "red",
    "blinking": True
}


# LED CONTROL

def all_off():
    clear_pin(PIN_RED)
    clear_pin(PIN_GREEN)
    clear_pin(PIN_BLUE)

def set_color(color, value):
    all_off()
    if color == "red":
        write_gpio(PIN_RED, value)
    elif color == "green":
        write_gpio(PIN_GREEN, value)
```

```
    elif color == "blue":
        write_gpio(PIN_BLUE, value)


# BLINK TASK (1 Hz, 25% duty)

async def blink_task():
    on_time  = BLINK_PERIOD * DUTY
    off_time = BLINK_PERIOD * (1 - DUTY)

    while True:
        if state["blinking"]:
            set_color(state["color"], 1)
            await asyncio.sleep(on_time)

            all_off()
            await asyncio.sleep(off_time)
        else:
            all_off()
            await asyncio.sleep(0.1)


# BUTTON TASK

async def button_task():
    # Previous buttons
    prev = [0, 0, 0, 0]

    while True:
        # Read buttons pushed  and save in btn
        btn = [btns[i].read() for i in range(4)]

        if btn[0] and not prev[0]:
            state["color"] = "red"
            state["blinking"] = True
            print("BTN0 → RED")

        if btn[1] and not prev[1]:
            state["color"] = "green"
            state["blinking"] = True
            print("BTN1 → GREEN")

        if btn[2] and not prev[2]:
            state["color"] = "blue"
            state["blinking"] = True
            print("BTN2 → BLUE")

        if btn[3] and not prev[3]:
            state["blinking"] = False
            print("BTN3 → STOP")

        # Save previously pushed button
        prev = btn
        await asyncio.sleep(0.05)
```

In [ ]:

```
# Schedule all tasks in main()
async def main():
    asyncio.create_task(blink_task())
    asyncio.create_task(button_task())
```

```
    await asyncio.sleep(9999)

# Launch tasks
asyncio.run(main())
```

```
BTN1 → GREEN
BTN2 → BLUE
BTN3 → STOP
BTN0 → RED
BTN1 → GREEN
BTN2 → BLUE
BTN3 → STOP
BTN0 → RED
BTN0 → RED
BTN0 → RED
BTN3 → STOP
BTN0 → RED
BTN0 → RED
BTN1 → GREEN
BTN2 → BLUE
BTN3 → STOP
BTN0 → RED
BTN1 → GREEN
```

In [56]:

```
# Visual Perception cell

import numpy as np

# PWM duty cycle values (percent)
duty_cycles = np.array([0, 5, 10, 25, 50, 75, 100])

# Gamma value for human vision
gamma = 2.2

# Normalize duty cycle
duty_norm = duty_cycles / 100.0

# Compute perceived brightness
brightness = (duty_norm ** (1 / gamma)) * 100

# Show results
for d, b in zip(duty_cycles, brightness):
    print(f"Duty Cycle: {d:>3}%  →  Perceived Brightness: {b:6.1f}%")
```

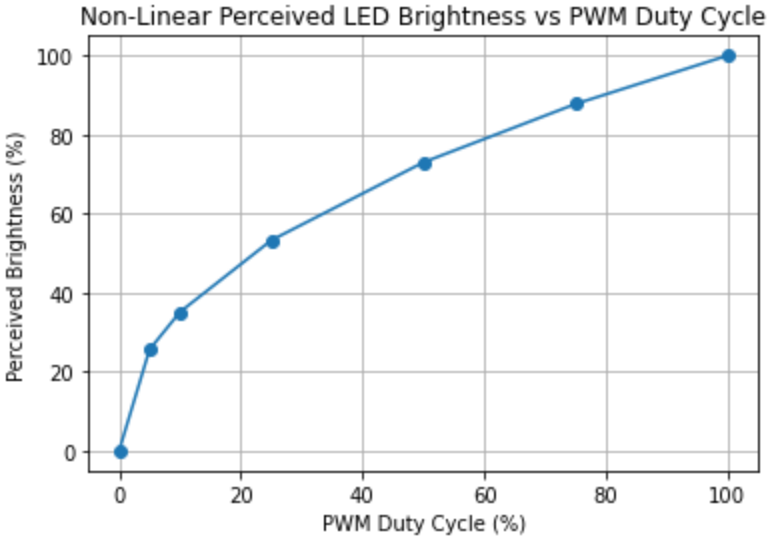
```
Duty Cycle:   0%  →  Perceived Brightness:    0.0%
Duty Cycle:   5%  →  Perceived Brightness:   25.6%
Duty Cycle:  10%  →  Perceived Brightness:   35.1%
Duty Cycle:  25%  →  Perceived Brightness:   53.3%
Duty Cycle:  50%  →  Perceived Brightness:   73.0%
Duty Cycle:  75%  →  Perceived Brightness:   87.7%
Duty Cycle: 100%  →  Perceived Brightness:  100.0%
```

In [57]:

```
import matplotlib.pyplot as plt

# Plot duty_cycle to show non-linearity
plt.figure()
plt.plot(duty_cycles, brightness, marker='o')
plt.xlabel("PWM Duty Cycle (%)")
plt.ylabel("Perceived Brightness (%)")
plt.title("Non-Linear Perceived LED Brightness vs PWM Duty Cycle")
```

```
plt.grid(True)
plt.show()
```



In [ ]: