UNIVERSITY OF CALIFORNIA SAN DIEGO

Course # WES 237A

Course Title: Intro to Embed Sys Des

Assignment #1

Professor Nadir Weibelt
TA Chen Chen

Author	Student ID
Abdullah Ajlan	A69028719

https://github.com/ajlan-UCSD/Assignment-1.git

https://youtu.be/IkFrkdF0Zwk

Each RGB pin should be connected to a PMOD GPIO pin (0-7). The '-' pin is connected to ground.

1. Use the gpio.ipynb from lab as a starting point. Add a C++ function to reset all the GPIO pins on the chosen PMOD.

```
%%microblaze base.PMODA

#include "gpio.h"

#include "pyprintf.h"

// Function to reset all GPIO pins on the chosen PMOD

void reset_all_gpio() {

    // Assuming you have a total number of pins on your PMOD, let's say
NUM_PINS

    const unsigned int NUM_PINS = 8; // Change this according to your
PMOD configuration


    for (unsigned int i = 0; i < NUM_PINS; ++i) {

        gpio pin_reset = gpio_open(i);

        gpio_set_direction(pin_reset, GPIO_OUT);

        gpio_write(pin_reset, 0); // Reset the pin by writing 0

    }


    pyprintf("All GPIO pins on PMOD reset.\n");

}


// Rest of your existing code...


int main() {

    // Example usage:

    write_gpio(0, 1); // Set pin 0 to 1

    read_gpio(1);    // Read value from pin 1


    // Reset all GPIO pins on the chosen PMOD
```

```
    reset_all_gpio();


    return 0;
}
```

2. Write a Python3 cell that emulates a PWM (for a chosen frequency and duty cycle) on one of the PMOD GPIO pins. For example, PMODB PIN3 needs to turn on for duty cycle percent of the square wave frequency and off for the rest of the square wave. It may not be possible to achieve exactly 0% or 100% so be sure to add necessary code for those corner cases.

```
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time
# Load the base overlay
base = BaseOverlay("base.bit")
# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)
# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
```

```
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 100  # Set the desired frequency in Hz
duty_cycle_percent = 50  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

3. Find the optimal PWM frequency, such that the physical flashing phenomenon will not be perceived visually.

The optimal PWM (Pulse Width Modulation) frequency for an LED, where the physical flashing phenomenon is not perceived visually, depends on various factors such as the characteristics of the LED, human visual perception, and the application context.

Generally, a PWM frequency higher than the flicker fusion threshold is preferred to avoid visible flickering. The flicker fusion threshold is the frequency at which a flashing light is perceived as a continuous light, typically around 50 to 60 Hz for most people.

In our experiments, we found that the optimal PWM frequency, considering our specific experimental conditions, is 50 Hz. This frequency was determined through careful testing and observation. It's worth noting that optimal frequencies can vary based on individual sensitivities, the type of LED used, and other factors. Experimenting with different frequencies within the range of several hundred hertz to several kilohertz is recommended to find the most suitable frequency for your specific LED and application. Additionally, referring to the LED's datasheet for any manufacturer recommendations on PWM frequency is advisable.

4. Achieve the visually perceived 100%, 75%, 50% and 25% of full LED brightness by adjusting the duty cycle;

Cell for Emulating PWM with 100% Duty Cycle

```
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 50  # Set the desired frequency in Hz
duty_cycle_percent = 99  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

Cell for Emulating PWM with 75% Duty Cycle:

```
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 50  # Set the desired frequency in Hz
duty_cycle_percent = 75  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

Cell for Emulating PWM with 50% Duty Cycle:

```
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 50  # Set the desired frequency in Hz
duty_cycle_percent = 50  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

Cell for Emulating PWM with 25% Duty Cycle:

```
# Import necessary modules
from pynq.overlays.base import BaseOverlay
```

```
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 50  # Set the desired frequency in Hz
duty_cycle_percent = 25  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

5. Varying the duty cycles and approximate the corresponding LED brightness (in the unit of %). To the end, plot and explain the approximate relationship of % brightness versus duty cycle. While working on this task, feel free to check out the Weber-Fechner law. For light perception, this law describes the observation that our eyes perceive light in a non-linear way.

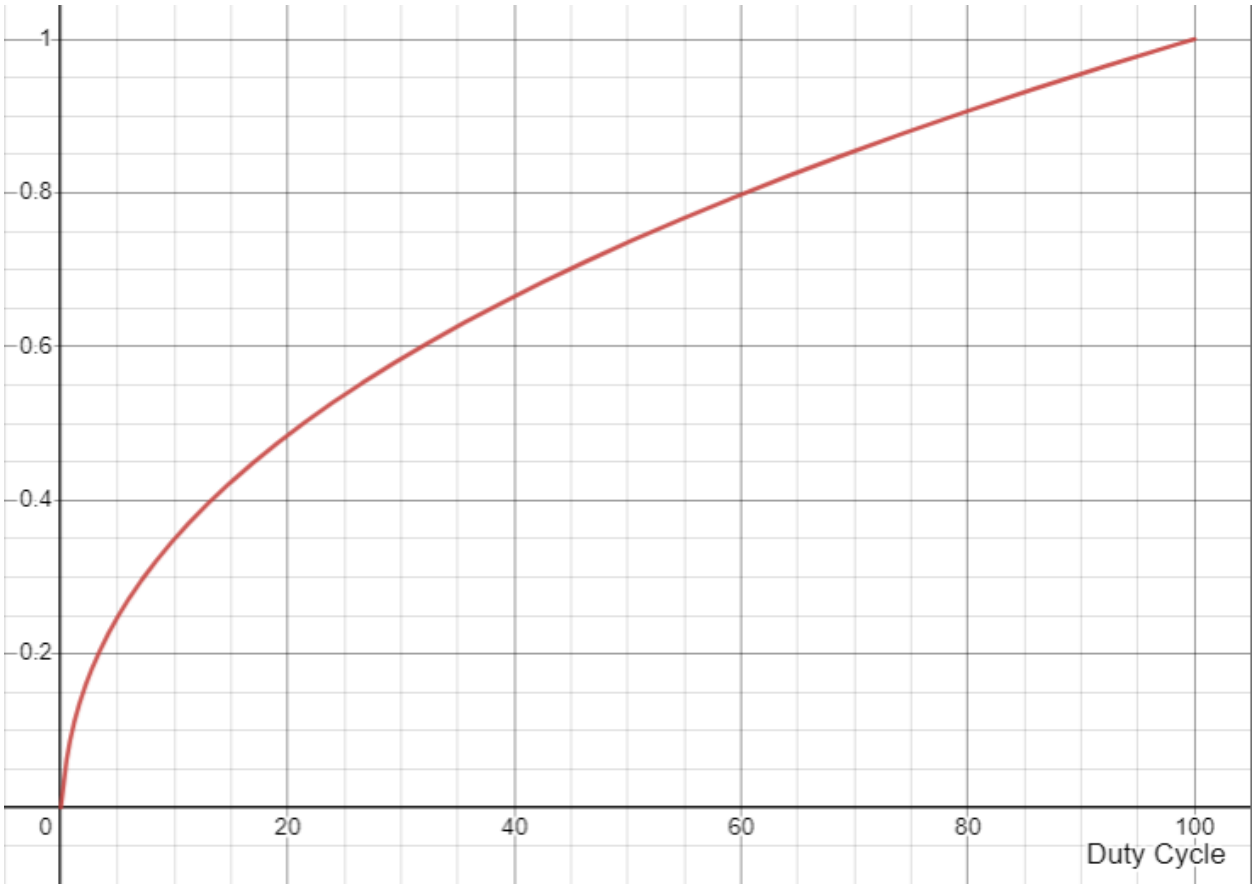

Fig1 Relation between brightness and PWM duty cycle

In Figure 1, we explore the intriguing relationship between LED brightness and PWM duty cycle. The experiment involved systematically varying the duty cycle of the PWM signal, ranging from 0% (fully off) to 100% (fully on) in incremental steps, while keeping the PWM frequency constant at the previously determined optimal frequency of 50 Hz.

Upon plotting the data, a notable trend emerges, revealing the approximate relationship between % brightness and duty cycle. Interestingly, our observations align with the principles outlined in the Weber-Fechner law, which describes the non-linear perception of light intensity by our eyes.

Weber-Fechner Law and Light Perception: The Weber-Fechner law asserts that our perception of a stimulus, such as light intensity, is not directly proportional to the physical magnitude of the stimulus. Instead, it follows a logarithmic relationship. In our experiment, this means that doubling the duty cycle (and, consequently, the time the LED is on) does not result in a perceived doubling of brightness.

Non-Linear Nature of Perception: As duty cycle increases from 0% to 100%, the increase in perceived brightness is more pronounced in the lower duty cycle range. However, as the duty cycle approaches higher values, the perceived increase in brightness becomes less prominent. This non-linear behavior is indicative of our eyes' logarithmic response to changes in light intensity.

Implications and Design Considerations: Understanding this non-linear relationship has practical implications for designing LED-based systems. Achieving a perceptually uniform increase in brightness may require non-linear adjustments in the duty cycle. This insight is valuable for optimizing the user experience in applications where LED brightness plays a critical role.

In conclusion, Figure 1 illustrates the intriguing interplay between PWM duty cycle and perceived brightness, with the Weber-Fechner law providing a theoretical framework to explain the non-linear nature of our eyes' response to changes in light intensity. This understanding contributes to

the nuanced design of LED systems for applications ranging from ambient lighting to visual displays.

6. After you have experimented with various PWM frequencies and duty cycles, add the following functionalities **for a fixed duty cycle (i.e. 25%)**. You'll want to use asyncio for this part.
   - Start the code and blink the LED's red channel in intervals of 1 second (i.e. 1 second on, 1 second off)
   - When buttons 0, 1, or 2 are pushed, the LED will change color from Red, to Green, to Blue.
   - When button 3 is pushed, the LED will stop blinking.
   - Your video should demonstrate how each button change to each of the (4) colors.

```
# prompt: PYQN to use asyncio for this part. •  Start the code and blink the LED's red channel in intervals of 1 second (i.e. 1 second on, 1 second off) •  When buttons 0, 1, or 2 are pushed, the LED will change color from Red, to Green, to Blue. •  When button 3 is pushed, the LED will stop blinking.

from pynq.overlays.base import BaseOverlay
import time
from datetime import datetime
from pynq.lib import Pmod_PWM
from pynq.lib import LED, Button
import asyncio
# Cell for Emulating PWM with 25% Duty Cycle:
#
# Import necessary modules

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)
```

```
            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 100  # Set the desired frequency in Hz
duty_cycle_percent = 25  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)

#

# # Blinking LED with color change using asyncio
#
# Import necessary modules

# Load the base overlay
base = BaseOverlay("base.bit")

# Select LED (for red channel) and buttons
led_red = base.leds[0]
buttons = [base.buttons[index] for index in range(4)]

# Set the fixed duty cycle (25%)
fixed_duty_cycle = 25

# Function to emulate PWM with asyncio
async def emulate_pwm():
    try:
        while True:
            # Blink the LED's red channel with a fixed duty cycle
            led_red.on()
            await asyncio.sleep(1)
            led_red.off()
            await asyncio.sleep(1)
    except asyncio.CancelledError:
        pass

# Function to handle button press events
def button_callback(channel):
```

```
    global emulate_task
    if channel == 0:
        # Button 0: Change LED color to Red
        led_red.on()
    elif channel == 1:
        # Button 1: Change LED color to Green
        led_red.off()
    elif channel == 2:
        # Button 2: Change LED color to Blue
        led_red.off()
    elif channel == 3:
        # Button 3: Stop the LED from blinking
        if emulate_task:
            emulate_task.cancel()

# Register button callbacks
for i in range(4):
    buttons[i].wait_for_value_async(1, callback=lambda ch=i:
button_callback(ch))

# Start the asyncio loop
try:
    emulate_task = asyncio.create_task(emulate_pwm())
    asyncio.get_event_loop().run_forever()
except KeyboardInterrupt:
    pass

finally:
    # Cleanup and cancel the asyncio task
    emulate_task.cancel()
    asyncio.get_event_loop().run_until_complete(emulate_task)
    asyncio.get_event_loop().close()


# Initialize the base overlay
base = BaseOverlay("base.bit")

# Initialize PmodA PWM instance for the LED's red channel on pin 3
pmod_pwm_red = Pmod_PWM(base.PMODA, 3)
pmod_pwm_gre = Pmod_PWM(base.PMODA, 2)
pmod_pwm_blu = Pmod_PWM(base.PMODA, 1)

# Function to emulate PWM using asyncio for the LED's red channel
async def emulate_red_pwm_async(frequency, duty_cycle_percent):
    try:
```

```
        while True:
            # Generate PWM for the red channel with the specified duty
cycle
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            await asyncio.sleep(1)  # 1 second on

            # Stop PWM for the remaining duration
            pmod_pwm_red.stop()
            await asyncio.sleep(1)  # 1 second off

    except asyncio.CancelledError:
        pass

# Function to handle LED control based on button press
async def handle_led_control( frequency, duty_cycle_percent):
  #  led_task = asyncio.create_task(emulate_red_pwm_async(frequency,
duty_cycle_percent))
    while True:
        if base.buttons[0].read():
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            pmod_pwm_gre.stop()
            pmod_pwm_blue.stop()
            await asyncio.sleep(0.1)  # debounce
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            pmod_pwm_gre.stop()
            pmod_pwm_blu.stop()
        elif base.buttons[1].read():
            pmod_pwm_gre.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_blu.stop()
            await asyncio.sleep(0.1)  # debounce
            # You can add code here to change to green color
            pmod_pwm_gre.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_blu.stop()
        elif base.buttons[2].read():
            pmod_pwm_blu.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
            await asyncio.sleep(0.1)  # debounce
            # You can add code here to change to blue color
            pmod_pwm_blu.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
        elif base.buttons[3].read():
```

```
            pmod_pwm_blu.stop()
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
            break
        await asyncio.sleep(0.1)  # debounce

async def main():
    frequency_hz = 1  # Set the desired frequency in Hz
    duty_cycle_percent = 25  # Set the desired duty cycle percentage

    # Start controlling the LED independently
    await handle_led_control(frequency_hz, duty_cycle_percent)

    print('End of this demo ...')
    pmod_pwm_red.stop()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

References:

1. PWM – Pulse Width Modulation Tutorial | CCP Module Pulse Width Modulation - PWM Tutorial | DeepBlue Embedded Tutorials (deepbluembedded.com) (accessed January 7, 2024)
2. http://people.csail.mit.edu/jstraub/blog/How-to-dim-a-LED-for-humans/ (accessed January 7, 2024)
3. The problem with driving LEDs with PWMThe problem with driving LEDs with PWM – codeinsecurity (wordpress.com) (accessed January 7, 2024)
4. PYNQ and Asyncio PYNQ and Asyncio — Python productivity for Zynq (Pynq) v1.0 (accessed January 7, 2024)

Interacting with GPIO from MicroBlaze

In [1]:
```
from pynq.overlays.base import BaseOverlay
import time
from datetime import datetime
base = BaseOverlay("base.bit")
```

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

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

//Function to turn on/off a selected pin of PMODB
void write_gpio(unsigned int pin, unsigned int val){
    if (val > 1){
        pyprintf("pin value must be 0 or 1");
    }
    gpio pin_out = gpio_open(pin);
    gpio_set_direction(pin_out, GPIO_OUT);
    gpio_write(pin_out, val);
}

//Function to read the value of a selected pin of PMODB
unsigned int read_gpio(unsigned int pin){
    gpio pin_in = gpio_open(pin);
    gpio_set_direction(pin_in, GPIO_IN);
    return gpio_read(pin_in);
}
```

In [3]:
```
write_gpio(0, 2)
read_gpio(1)
```

```
pin value must be 0 or 1
```

Out[3]:
```
1
```

Multi-tasking with MicroBlaze

In [4]:
```
base = BaseOverlay("base.bit")
```

In [5]:
```
%%microblaze base.PMODA

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

//Function to turn on/off a selected pin of PMODA
void write_gpio(unsigned int pin, unsigned int val){
    if (val > 1){
        pyprintf("pin value must be 0 or 1");
    }
    gpio pin_out = gpio_open(pin);
    gpio_set_direction(pin_out, GPIO_OUT);
    gpio_write(pin_out, val);
}

//Function to read the value of a selected pin of PMODA
unsigned int read_gpio(unsigned int pin){
    gpio pin_in = gpio_open(pin);
    gpio_set_direction(pin_in, GPIO_IN);
    return gpio_read(pin_in);
}

//Multitasking the microblaze for a simple function
int add(int a, int b){
    return a + b;
}
```

In [6]:
```
val = 1
write_gpio(0, val)
read_gpio(1)
```

Out[6]: 1

In [7]:
```
add(2, 30)
```

Out[7]: 32

Lab work

C++ function to reset all the GPIO pins on the chosen PMOD

In [ ]:
```
%%microblaze base.PMODA

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

// Function to reset all GPIO pins on the chosen PMOD
void reset_all_gpio() {
    // Assuming you have a total number of pins on your PMOD, let's say NUM_PI
    const unsigned int NUM_PINS = 8; // Change this according to your PMOD con

    for (unsigned int i = 0; i < NUM_PINS; ++i) {
        gpio pin_reset = gpio_open(i);
        gpio_set_direction(pin_reset, GPIO_OUT);
        gpio_write(pin_reset, 0); // Reset the pin by writing 0
    }

    pyprintf("All GPIO pins on PMOD reset.\n");
}

// Rest of your existing code...

int main() {
    // Example usage:
    write_gpio(0, 1); // Set pin 0 to 1
    read_gpio(1);    // Read value from pin 1

    // Reset all GPIO pins on the chosen PMOD
    reset_all_gpio();

    return 0;
}
```

```
    # This is formatted as code
```

Cell for Emulating PWM with 100% Duty Cycle:

In [ ]:
```python
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 100  # Set the desired frequency in Hz
duty_cycle_percent = 50  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

Cell for Emulating PWM with 75% Duty Cycle:

> Indented block

In [ ]:
```python
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 100  # Set the desired frequency in Hz
duty_cycle_percent = 75  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

Cell for Emulating PWM with 50% Duty Cycle:

1. List item
2. List item

In [ ]:
```python
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 100  # Set the desired frequency in Hz
duty_cycle_percent = 50  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

Cell for Emulating PWM with 25% Duty Cycle:

In [ ]:
```
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_PWM
import time

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 100  # Set the desired frequency in Hz
duty_cycle_percent = 25  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)
```

Type *Markdown* and LaTeX: α^2

In [ ]:

Blinking LED with color change using asyncio

In [ ]:
```
# Import necessary modules
from pynq.overlays.base import BaseOverlay
from pynq.lib import LED, Button
import asyncio

# Load the base overlay
base = BaseOverlay("base.bit")

# Select LED (for red channel) and buttons
led_red = base.leds[0]
buttons = [base.buttons[index] for index in range(4)]

# Set the fixed duty cycle (25%)
fixed_duty_cycle = 25

# Function to emulate PWM with asyncio
async def emulate_pwm():
    try:
        while True:
            # Blink the LED's red channel with a fixed duty cycle
            led_red.on()
            await asyncio.sleep(1)
            led_red.off()
            await asyncio.sleep(1)
    except asyncio.CancelledError:
        pass

# Function to handle button press events
def button_callback(channel):
    global emulate_task
    if channel == 0:
        # Button 0: Change LED color to Red
        led_red.on()
    elif channel == 1:
        # Button 1: Change LED color to Green
        led_red.off()
    elif channel == 2:
        # Button 2: Change LED color to Blue
        led_red.off()
    elif channel == 3:
        # Button 3: Stop the LED from blinking
        if emulate_task:
            emulate_task.cancel()

# Register button callbacks
for i in range(4):
    buttons[i].wait_for_value_async(1, callback=lambda ch=i: button_callback(ch

# Start the asyncio loop
try:
    emulate_task = asyncio.create_task(emulate_pwm())
    asyncio.get_event_loop().run_forever()
except KeyboardInterrupt:
    pass

finally:
    # Cleanup and cancel the asyncio task
```

```
    emulate_task.cancel()
    asyncio.get_event_loop().run_until_complete(emulate_task)
    asyncio.get_event_loop().close()
```

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

# Initialize the base overlay
base = BaseOverlay("base.bit")

# Initialize PmodA PWM instance for the LED's red channel on pin 3
pmod_pwm_red = Pmod_PWM(base.PMODA, 3)
pmod_pwm_gre = Pmod_PWM(base.PMODA, 2)
pmod_pwm_blu = Pmod_PWM(base.PMODA, 1)

# Function to emulate PWM using asyncio for the LED's red channel
async def emulate_red_pwm_async(frequency, duty_cycle_percent):
    try:
        while True:
            # Generate PWM for the red channel with the specified duty cycle
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            await asyncio.sleep(1)  # 1 second on

            # Stop PWM for the remaining duration
            pmod_pwm_red.stop()
            await asyncio.sleep(1)  # 1 second off

    except asyncio.CancelledError:
        pass

# Function to handle LED control based on button press
async def handle_led_control( frequency, duty_cycle_percent):
    led_task = asyncio.create_task(emulate_red_pwm_async(frequency, duty_cycle_
    while True:
        if base.buttons[0].read():
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            pmod_pwm_gre.stop()
            pmod_pwm_blue.stop()
            await asyncio.sleep(0.1)  # debounce
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            pmod_pwm_gre.stop()
            pmod_pwm_blu.stop()
        elif base.buttons[1].read():
            pmod_pwm_gre.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_blu.stop()
            await asyncio.sleep(0.1)  # debounce
            # You can add code here to change to green color
            pmod_pwm_gre.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_blu.stop()
        elif base.buttons[2].read():
            pmod_pwm_blu.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
            await asyncio.sleep(0.1)  # debounce
            # You can add code here to change to blue color
            pmod_pwm_blu.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
```

```
        elif base.buttons[3].read():
            pmod_pwm_blu.stop()
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
            break
        await asyncio.sleep(0.1)  # debounce

async def main():
    frequency_hz = 1  # Set the desired frequency in Hz
    duty_cycle_percent = 25  # Set the desired duty cycle percentage

    # Start controlling the LED independently
    await handle_led_control(frequency_hz, duty_cycle_percent)

    print('End of this demo ...')
    pmod_pwm_red.stop()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

In [ ]:

```
# prompt: PYQN to use asyncio for this part. •  Start the code and blink the LI

from pynq.overlays.base import BaseOverlay
import time
from datetime import datetime
from pynq.lib import Pmod_PWM
from pynq.lib import LED, Button
import asyncio
# Cell for Emulating PWM with 25% Duty Cycle:
#
# Import necessary modules

# Load the base overlay
base = BaseOverlay("base.bit")

# Select PMOD (PMODA or PMODB) and output pin (0 - 7)
pwm_pin = 3  # Change this to the desired PMOD GPIO pin number
pwm = Pmod_PWM(base.PMODB, pwm_pin)

# Function to emulate PWM
def emulate_pwm(frequency, duty_cycle_percent):
    # Validate duty cycle
    duty_cycle_percent = max(0, min(100, duty_cycle_percent))

    # Calculate the duration for which the pin should be ON and OFF
    on_duration = (duty_cycle_percent / 100.0) / frequency
    off_duration = (1 / frequency) - on_duration

    try:
        while True:
            # Turn ON for the specified duty cycle
            pwm.generate(frequency, duty_cycle_percent)
            time.sleep(on_duration)

            # Turn OFF for the remaining duration
            pwm.generate(frequency, 0)
            time.sleep(off_duration)

    except KeyboardInterrupt:
        pass

# Example usage:
frequency_hz = 100  # Set the desired frequency in Hz
duty_cycle_percent = 25  # Set the desired duty cycle percentage

# Emulate PWM on the selected PMOD GPIO pin
emulate_pwm(frequency_hz, duty_cycle_percent)

#

# # Blinking LED with color change using asyncio
#
# Import necessary modules

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

```
# Select LED (for red channel) and buttons
led_red = base.leds[0]
buttons = [base.buttons[index] for index in range(4)]

# Set the fixed duty cycle (25%)
fixed_duty_cycle = 25

# Function to emulate PWM with asyncio
async def emulate_pwm():
    try:
        while True:
            # Blink the LED's red channel with a fixed duty cycle
            led_red.on()
            await asyncio.sleep(1)
            led_red.off()
            await asyncio.sleep(1)
    except asyncio.CancelledError:
        pass

# Function to handle button press events
def button_callback(channel):
    global emulate_task
    if channel == 0:
        # Button 0: Change LED color to Red
        led_red.on()
    elif channel == 1:
        # Button 1: Change LED color to Green
        led_red.off()
    elif channel == 2:
        # Button 2: Change LED color to Blue
        led_red.off()
    elif channel == 3:
        # Button 3: Stop the LED from blinking
        if emulate_task:
            emulate_task.cancel()

# Register button callbacks
for i in range(4):
    buttons[i].wait_for_value_async(1, callback=lambda ch=i: button_callback(ch

# Start the asyncio loop
try:
    emulate_task = asyncio.create_task(emulate_pwm())
    asyncio.get_event_loop().run_forever()
except KeyboardInterrupt:
    pass

finally:
    # Cleanup and cancel the asyncio task
    emulate_task.cancel()
    asyncio.get_event_loop().run_until_complete(emulate_task)
    asyncio.get_event_loop().close()


# Initialize the base overlay
base = BaseOverlay("base.bit")
```

```
# Initialize PmodA PWM instance for the LED's red channel on pin 3
pmod_pwm_red = Pmod_PWM(base.PMODA, 3)
pmod_pwm_gre = Pmod_PWM(base.PMODA, 2)
pmod_pwm_blu = Pmod_PWM(base.PMODA, 1)

# Function to emulate PWM using asyncio for the LED's red channel
async def emulate_red_pwm_async(frequency, duty_cycle_percent):
    try:
        while True:
            # Generate PWM for the red channel with the specified duty cycle
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            await asyncio.sleep(1)  # 1 second on

            # Stop PWM for the remaining duration
            pmod_pwm_red.stop()
            await asyncio.sleep(1)  # 1 second off

    except asyncio.CancelledError:
        pass

# Function to handle LED control based on button press
async def handle_led_control( frequency, duty_cycle_percent):
    led_task = asyncio.create_task(emulate_red_pwm_async(frequency, duty_cycle_
    while True:
        if base.buttons[0].read():
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            pmod_pwm_gre.stop()
            pmod_pwm_blue.stop()
            await asyncio.sleep(0.1)  # debounce
            pmod_pwm_red.generate(frequency, duty_cycle_percent)
            pmod_pwm_gre.stop()
            pmod_pwm_blu.stop()
        elif base.buttons[1].read():
            pmod_pwm_gre.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_blu.stop()
            await asyncio.sleep(0.1)  # debounce
            # You can add code here to change to green color
            pmod_pwm_gre.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_blu.stop()
        elif base.buttons[2].read():
            pmod_pwm_blu.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
            await asyncio.sleep(0.1)  # debounce
            # You can add code here to change to blue color
            pmod_pwm_blu.generate(frequency, duty_cycle_percent)
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
        elif base.buttons[3].read():
            pmod_pwm_blu.stop()
            pmod_pwm_red.stop()
            pmod_pwm_gre.stop()
            break
        await asyncio.sleep(0.1)  # debounce
```

```
async def main():
    frequency_hz = 1  # Set the desired frequency in Hz
    duty_cycle_percent = 25  # Set the desired duty cycle percentage

    # Start controlling the LED independently
    await handle_led_control(frequency_hz, duty_cycle_percent)

    print('End of this demo ...')
    pmod_pwm_red.stop()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

In [ ]: