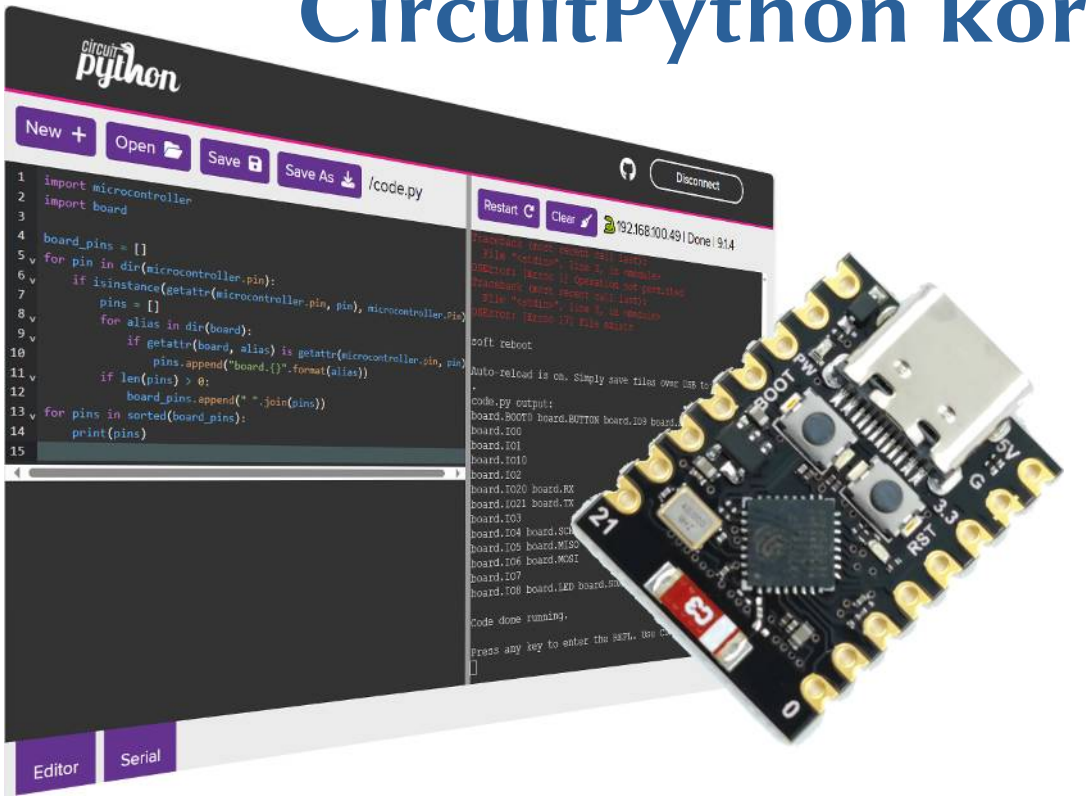
ESP32-C3 mikrovezérlők programozása CircuitPython környezetben



4. Analóg vezérlés (PWM)

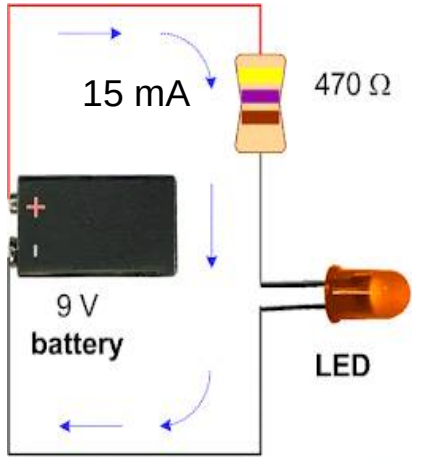
Felhasznált és ajánlott irodalom

- **Python:**
  - Mark Pilgrim/Kelemen Gábor: Ugorj fejest a Python 3-ba!
  - P. Wentworth et al. (ford. Biró Piroska, Szeghalmy Szilvia és Varga Imre): Hogyan gondolkozz úgy, mint egy informatikus: Tanulás Python 3 segítségével
- **CircuitPython:**
  - Adafruit: https://circuitpython.org/downloads
  - Learn Adafruit: Welcome to CircuitPython
  - Learn Adafruit: CircuitPython Essentials
  - Adafruit: Adafruit CircuitPython API Reference
  - Adafruit: github.com/adafruit/Adafruit CircuitPython Bundle

- **Online eszközök és támogatás:**
  - Learn Adafruit: CircuitPython on ESP32 Quick Start
  - Adafruit: Adafruit Web Serial ESPTool
  - Adafruit: CircuitPython Code Editor

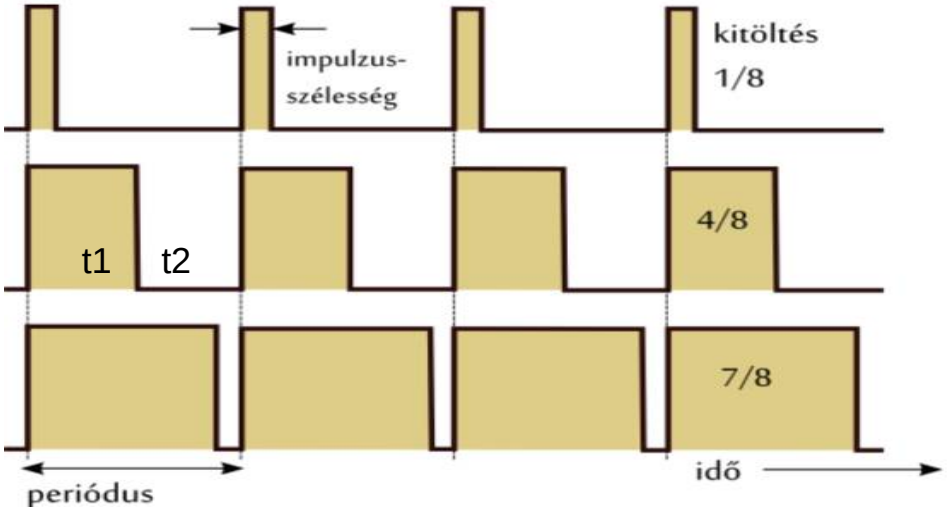
Impulzus-szélesség moduláció (PWM)

- Hogyan szabályozzuk egy fogyasztó teljesítményét ha az analóg módszer nem hatékony?
  Ebben a példában az ellenálláson 3,5-ször nagyobb teljesítmény disszipálódik, mint amennyi a LED-re jut
- Megoldás lehet a fogyasztó periodikus be- és kikapcsolása, ahol az átlagos teljesítményfelvételt a be- és kikapcsoltsági idő aránya szabja meg
- A **PWM** lényege: állandó periódusidejű jelet keltünk, a szabályozás pedig a jel kitöltési tényezőjének változtatásával történik



470 Ω
15 mA
U_R = 7 V
105 mW

LED
Uf ≈ 2 V
15 mA
30 mW



kitöltési tényező = impulzusszélesség/periódusidő

vagy:

$$kitöltési\ tényező = \frac{t_1}{t_1 + t_2}$$

PWM kimenetek

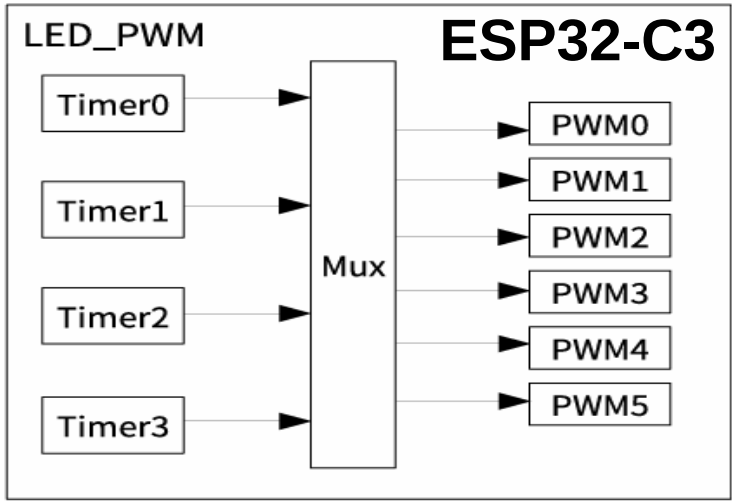
- Bármelyik GPIO láb lehet PWM kimenet, de egyidejűleg csak legfeljebb 6

PWM kimenetek kezelése CircuitPythonban

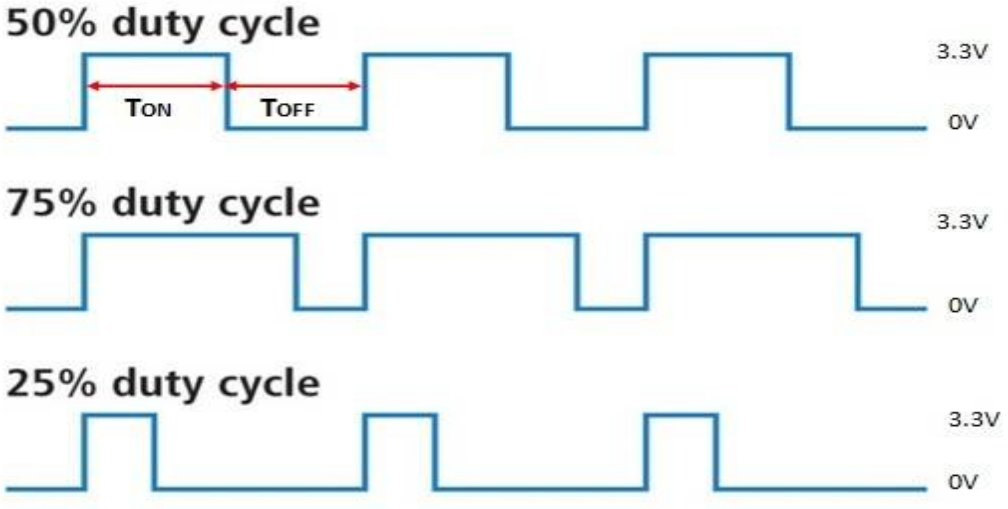
- **PWM** kimenetek kezelése: **pwmio** modul **PWMOut** osztály
- A konstruktor:

```
PWMOut(pin:Pin,frequency:int,duty_cycle:int,var_freq:bool)
```

pin – a választott kivezetés,
frequency – a frekvencia (Hz),
duty_cycle – a kitöltés (0 – 65 535),
var_freq – változó frekvencia jelzése



Bármelyik GPIO kivezetés lehet PWM kimenet, de egyidejűleg csak 6 PWM jelet tudunk előállítani



TON: Time requires for pulse to remain ON i.e. HIGH State
TOFF: Time requires for pulse to remain OFF i.e. LOW State

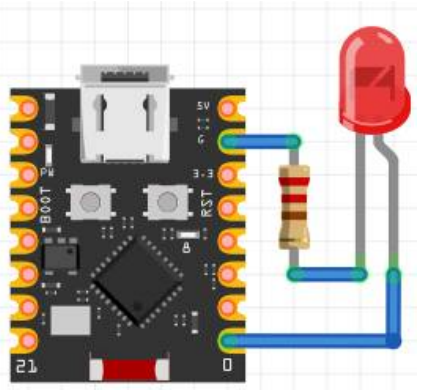
led_fade1.py

- Legyen **PWMOut** kimenet a **GPIO0** kivezetés, amelyre egy LED-et kötünk, és fokozatosan növeljük, majd a maximum elérése után csökkentsük a kitöltést!
- A **PWM** frekvenciát 1000 Hz-re állítottuk be
- A kitöltés (duty_cycle) 0 – 65 535 közötti szám lehet
- Az ***i*** ciklusszámláló értéke 0 – 127 között változik, ezt 512-vel szorozva tudjuk a kívánt tartományba transzformálni
- Az 512-vel történő szorzást a balra léptető operátorral a legkönnyebb elvégezni:

  x << 9 = x * 512

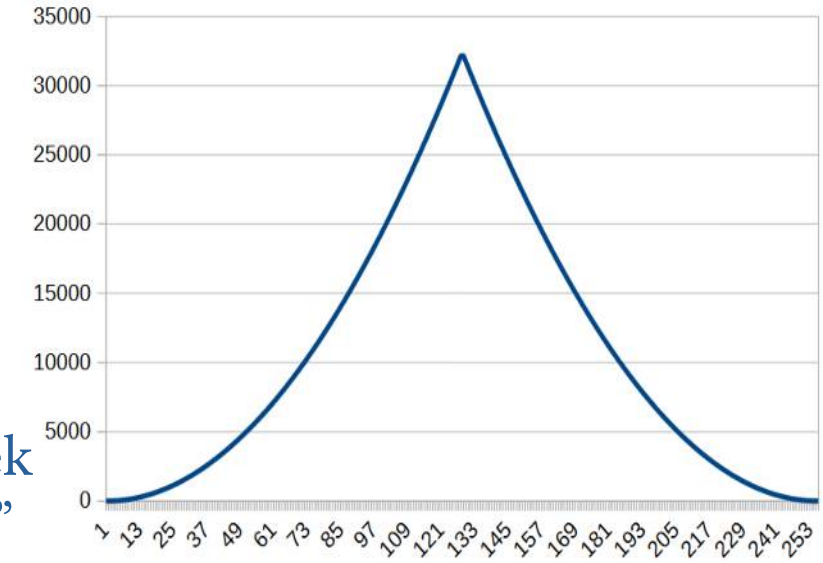
```
import pwmio
import board
import time

#--- Initialize PWM output on IO0 for the LED
led=pwmio.PWMOut(board.IO0,frequency=1000, duty_cycle=0)

while True:
    #--- Fade in --------
    for i in range(128):
        led.duty_cycle = i<<9
        time.sleep(0.015)
    #--- Fade out -------
    for i in range(128):
        led.duty_cycle = (127-i)<<9
        time.sleep(0.015)  # 15ms delay
```

Szorzás 512-vel



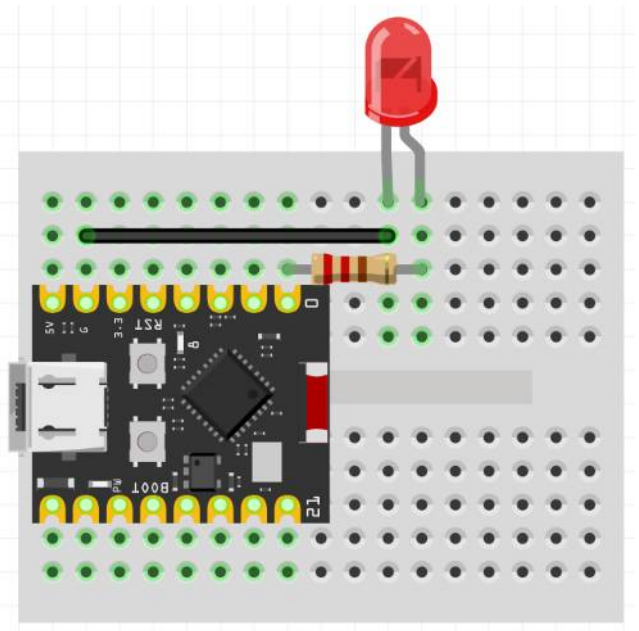
led_fade2.py – a „lélegző” LED

- A szemünk fényérzékelése nem lineáris, vagyis az *n*-szer nagyobb fényerő nem fog arányosan *n*-szer nagyobb fényérzetet okozni, ezért az előző program nem kelt olyan látványt, mint szeretnénk
- Jobb hatást érünk el, ha a fényerő egyre meredekebben emelkedik és egyre lassabban csökken. A matematikai nehézségek elkerülésére egy egyszerű algoritmust használunk:
  - bevezettünk egy ***increment*** nevű változót, amelynek az értékét minden lépésben 4-gyel megnöveljük/lecsökkentjük
  - A fényerő *(**brightness**)* értékét minden lépésben az ***increment*** változó új értékével növeljük/csökkentjük
- A fenti algoritmussal a szemünk egyenletesebbnek látja a fényerő változását, a LED szinte „lélegezik”



led_fade2.py – a „lélegző” LED

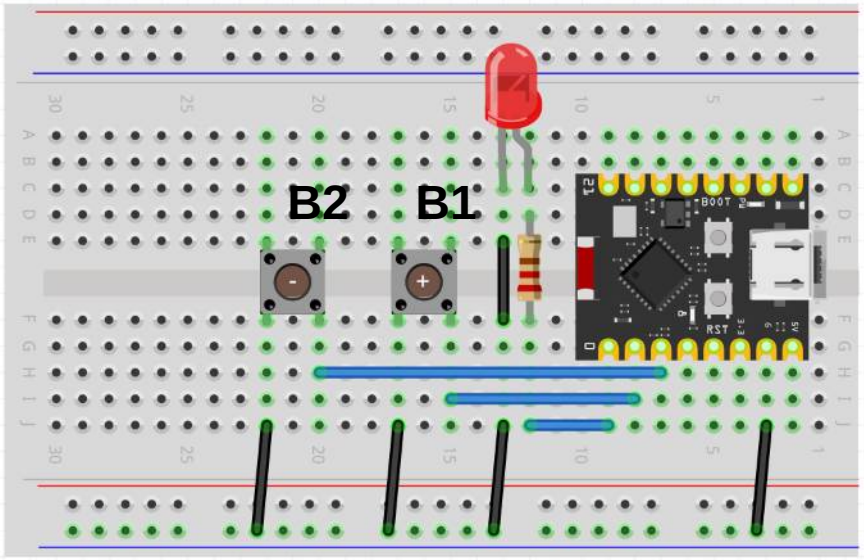
```
import pwmio
import board
import time
#--- Initialize PWM output on IO0 for the LED
led = pwmio.PWMOut(board.IO0, frequency=1000, duty_cycle=0)
while True:
    brightness = 0
    increment = 0          # Pre-calculated value
    #--- Fade in --------
    for i in range(128):
        led.duty_cycle = brightness
        brightness += increment
        increment += 4     # Gradual increase
        time.sleep(0.015) # 15ms delay
    #--- Fade out -------
    for i in range(128):
        increment -= 4     # Gradual decrease
        brightness -= increment
        led.duty_cycle = brightness
        time.sleep(0.015) # 15ms delay
```



Csak ebben a sorrendben végrehajtva tudunk ugyanazokon az értékeken lépkedni visszafelé!

led_control.ino – egy gyakorlati alkalmazás

- Tervezzük meg egy szabályozható LED fényforrás vezérlését:
  - két nyomógomb rövid lenyomása be- és kikapcsolja a LED-et
  - hosszú lenyomásuk pedig fokozatosan növeli/csökkenti a fényerőt
- A **LED**-et *(melynek fényerejét **PWM**-mel szabályozzuk)* a **GPIO0** kimenetre kötjük
- A **B1** nyomógombot *(bekapcsolás, fényerő növelés)* a **GPIO1** bemenetre kötjük
- A **B2** nyomógombot *(kikapcsolás, fényerő csökkentés)* a **GPIO2** bemenetre kötjük
- **Használat:**
  - **B1** rövid nyomásra bekapcsolja a LED-et a legutóbbi fényerő beállítással
  - **B1** hosszan nyomva tartva (min. 0.5s) fokozatosan növeli a LED fényerejét (0.5s időközzel 15 lépésben duplázza a PWM kitöltést)
  - **B2** rövid nyomásra kikapcsolja a LED-et
  - **B2** hosszan nyomva tartva (min. 0.5s) fokozatosan növeli a LED fényerejét (0.5s időközönként 15 lépésben felezi a PWM kitöltést)



led_control.ino – 4/1.

```
import pwmio
import board
import digitalio
import time

led = pwmio.PWMOut(board.IO0, frequency=1000, duty_cycle=0)

button1 = digitalio.DigitalInOut(board.IO1)
button1.direction = digitalio.Direction.INPUT
button1.pull = digitalio.Pull.UP
button2 = digitalio.DigitalInOut(board.IO2)
button2.direction = digitalio.Direction.INPUT
button2.pull = digitalio.Pull.UP

button1_press_time = 0
button2_press_time = 0
button1_counter = 0
button2_counter = 0

# Brightness control variables
brightness_levels = [int(65535 / (2 ** (15 - i))) for i in range(15)]
brightness_index = 0
led.duty_cycle = brightness_levels[brightness_index]
```

led_control.ino – 4/2.

```
# Check button press function
def check_button_press():
    global button1_press_time, button2_press_time, button1_counter, button2_counter
    current_time = time.monotonic()

    # Check button1 (B1)
    if not button1.value:              # Button1 pressed?
        if button1_press_time == 0:
            button1_press_time = current_time
        button1_counter += 1
        if button1_counter >= 10:      # Check every 500 ms (50 ms * 10)
            handle_long_press(button1)
            button1_counter = 0
    else:
        if button1_press_time != 0:
            press_duration = current_time - button1_press_time
            if press_duration < 0.5:
                handle_short_press(button1)
            button1_press_time = 0
            button1_counter = 0
```

led_control.ino – 4/3.

```
# Check button2 (B2)
    if not button2.value: # Button2 pressed?
        if button2_press_time == 0:
            button2_press_time = current_time
        button2_counter += 1
        if button2_counter >= 10:  # Check every 500 ms (50 ms * 10)
            handle_long_press(button2)
            button2_counter = 0
    else:
        if button2_press_time != 0:
            press_duration = current_time - button2_press_time
            if press_duration < 0.5:
                handle_short_press(button2)
            button2_press_time = 0
            button2_counter = 0

# Handle short press
def handle_short_press(button):
    if button == button1:
        turn_on_led()
    elif button == button2:
        turn_off_led()
```

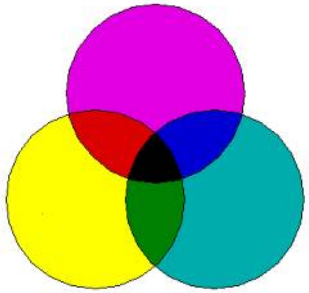
```
# Handle long press
def handle_long_press(button):
    if button == button1:
        increase_brightness()
    elif button == button2:
        decrease_brightness()
```

led_control.ino – 4/4.

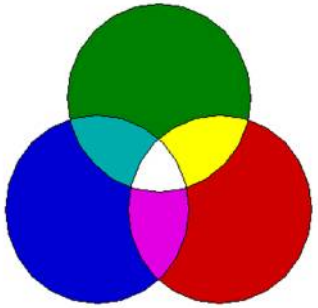
```
def turn_on_led():
    global led
    led.duty_cycle = brightness_levels[brightness_index]

def turn_off_led():
    global led
    led.duty_cycle = 0

def increase_brightness():
    global brightness_index, led
    if brightness_index < 14:
        brightness_index += 1
        led.duty_cycle = brightness_levels[brightness_index]

def decrease_brightness():
    global brightness_index, led
    if brightness_index > 0:
        brightness_index -= 1
        led.duty_cycle = brightness_levels[brightness_index]

while True:
    check_button_press()
    time.sleep(0.05)  # 50 ms delay for button sampling
```

Szín, színlátás, színkeverés, színtér modellek

- Az Arduino tanfolyam keretében a **2020. április 30-i** előadásban (videofelvétel, előadásvázlat, mintaprogramok) már részletesen beszéltünk a színről, a színlátásról, a színkeverésről és különféle színtér modellekről, illetve az ezek között konverzióról, az alábbiakban ezeknek csak néhány részletét elevenítjük fel
- **A színkeverésnek** két módszere ismert:
  - **szubtraktív színkeverés** az emberi szemtől függetlenül, a fények természetes spektrális módosulása útján jön létre (pl. nyomdászat)
  - **additív színkeverés** az emberi látórendszerben alakul ki (pl. színes TV)



- Szubtraktív színkeverés (**CYMK modell,** a tinta gyengíti a fényvisszaverést bizonyos hullámhosszon)

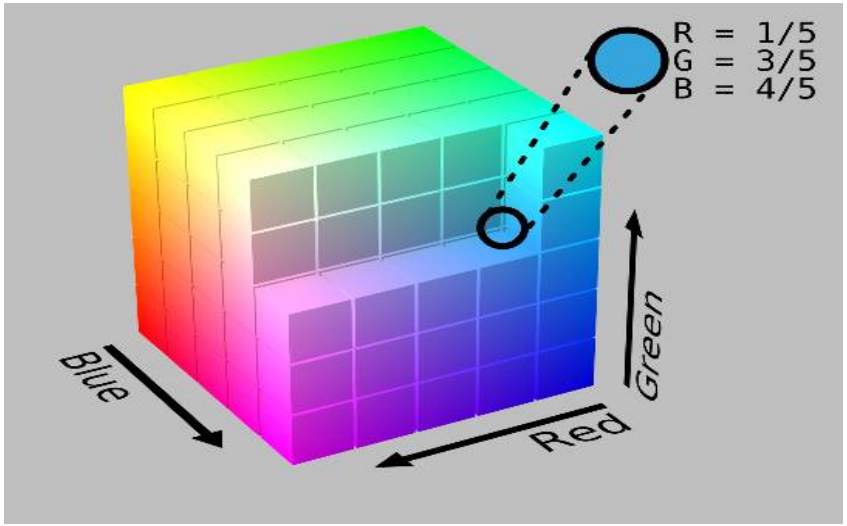


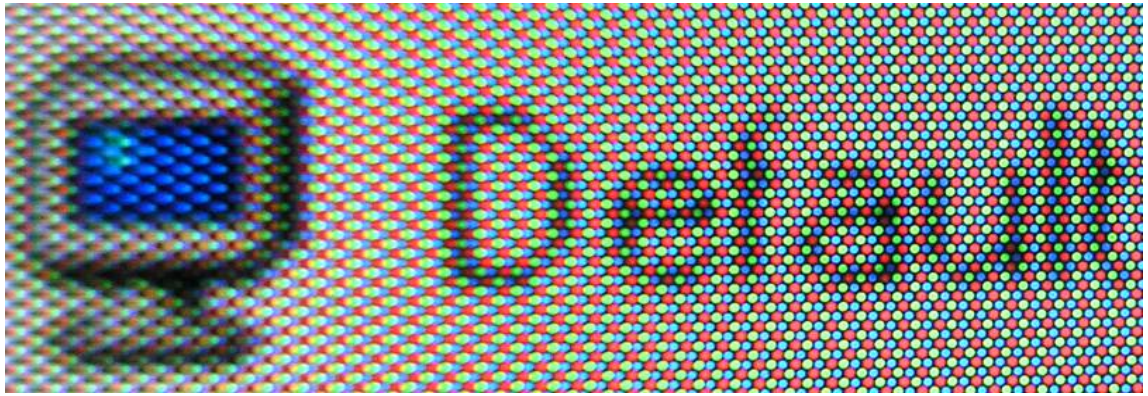
A mai előadásban ezzel kísérletezünk

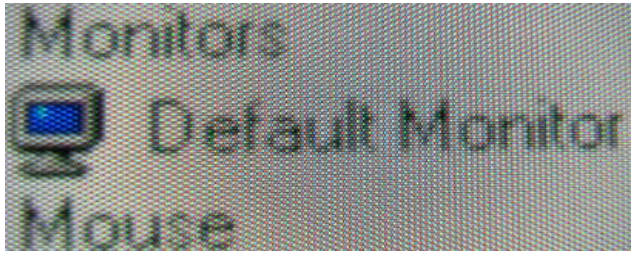
- Additív színkeverés (**RGB modell,** különböző hullámhosszúságú fénysugarakkal)

Az RGB színtér modell

- Az **RGB** additív színtér modellben vörös, zöld és kék összetevők keverésével próbáljuk leírni, vagy előállítani a színeket
- Minden színárnyalat az R, G és B ”koordinátákkal” jellemezhető
- Az alábbi képen egy színes monitor fényképe látható kinagyítva







Wikipedia: RGB color model, Color spaces w RGB primaries

A HSV (vagy HSI) színtér modell

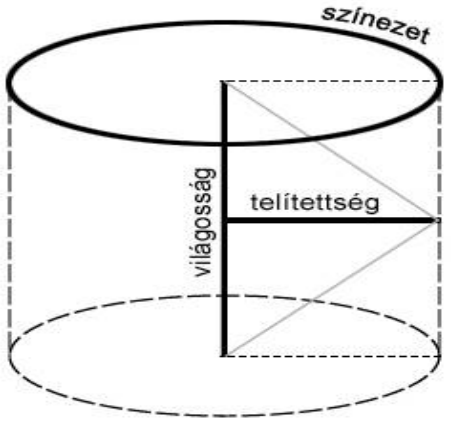
- A színérzetek jellemzőit hengerkoordináta rendszerben is ábrázolhatjuk, ahol a Φ szög a színezet (*Hue*, 0 – 360°), a tengelytől való távolság a telítettség (*Saturation*, 0 – 100), a tengelymenti távolság pedig a világosság (*Value, Intensity*, 0 – 100)
- A henger inkább kettőskúp, mert fekete, ill. fehér szín esetében nincs értelme telítettségről, vagy színezetről beszélni
- Összefüggés a HSI és az RGB értékek között:

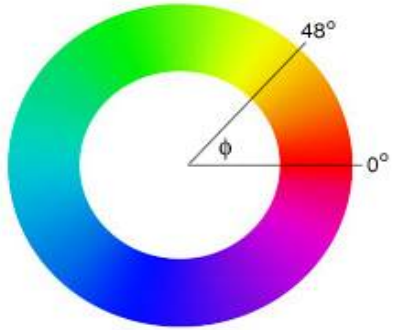
$$I = \frac{R+G+B}{3}$$

$$S = 1 - \frac{3}{(R+G+B)} \min(R,G,B)$$

$$H = \cos^{-1}\left(\frac{(R-G)+(R-B)}{2\sqrt{(R-G)^2+(R-B)(G-B)}}\right) \quad \text{assuming } G > B$$

If $B > G$, then $H = 360 - H$.





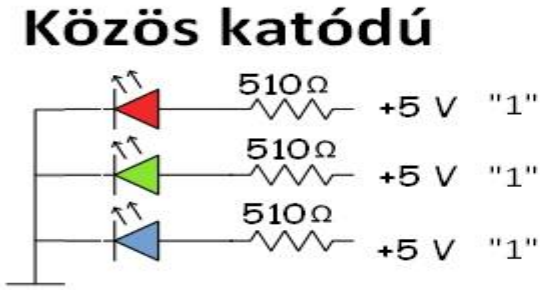
Források:

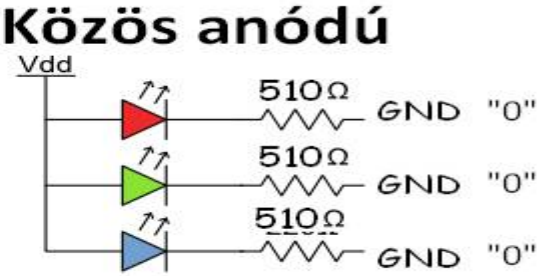
Földvári Melinda: Szín + Kommunikáció

Wikipedia: RGB color model

(analóg) RGB LED-ek

- Az RGB LED három, különböző színű (vörös, zöld, kék) LED-et tartalmaz egy közös tokban, s vagy az anódok, vagy a katódok össze vannak kötve
- Az áramkorlátozásról nekünk kell gondoskodni (pl. ellenállás)
- **Közös katód** esetén az anódokat magas szintre kell húzni
- **Közös anód** esetén a katódokat lefelé kell húzni (inverz logika)
- Kísérleteinknél elég lesz színenként 5 – 10 mA

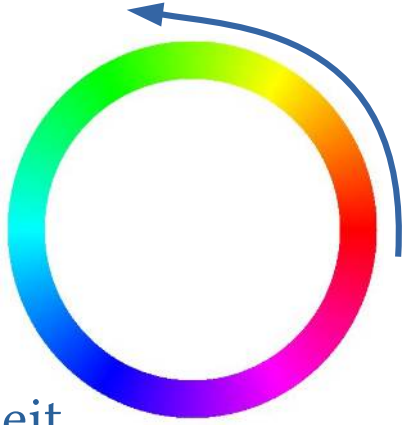


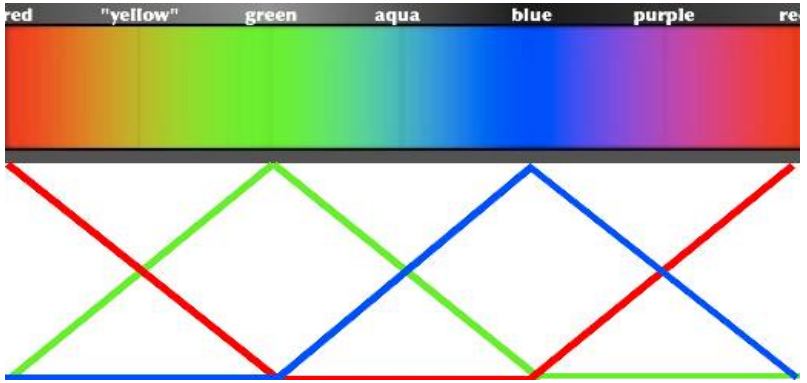


+5V helyett most +3,3V lesz a magas szint

Hogyan keverjük a színeket?

- Az **RGB LED** lábait **PWM**-mel vezérelve színátmeneteket hozhatunk létre
- A színspektrum lefedését három színátmenettel oldhatjuk meg:
  - piros ⟶ zöld átmenet
  - zöld ⟶ kék átmenet
  - kék ⟶ piros átmenet
- A **CircuitPython rainbowio** beépített moduljának **colorwheel** függvénye a színkereket a nyíl szerinti irányba haladva 256 részre osztja, s a kiválasztott szín R, G, B komponenseit egy 24 bites számba tömörítve adja vissza pl.
  `rainbowio.colorwheel(200)` ⟶ `0x5A00A5`





`0101 1010` `0000 0000` `1010 0101`

R G B

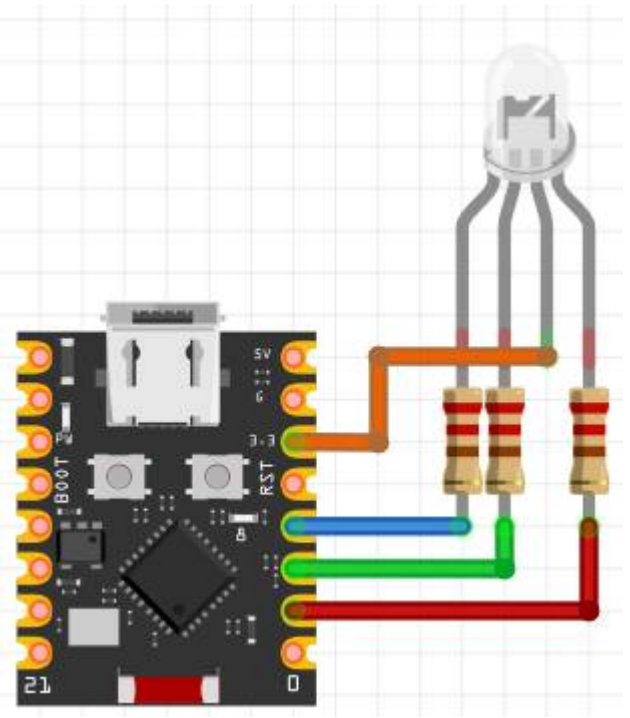
RGB_colorwheel.py – kevergetjük a színeket

```
import time
import board
import pwmio
from rainbowio import colorwheel
red = pwmio.PWMOut(board.IO1, frequency=1000)
green = pwmio.PWMOut(board.IO2, frequency=1000)
blue = pwmio.PWMOut(board.IO3, frequency=1000)

def set_rgb_color(r, g, b):
    red.duty_cycle = 65535 - (r * 257)
  green.duty_cycle = 65535 - (g * 257)
   blue.duty_cycle = 65535 - (b * 257)

while True:
    for i in range(256):
        color = colorwheel(i)  # 24-bit color code
        r = (color >> 16) & 0xFF
        g = (color >> 8) & 0xFF
        b = color & 0xFF
        set_rgb_color(r, g, b)
        time.sleep(0.05)
```

Invertálás közös anód esetéhez

Közös katód esetén ezeket töröljük ki!

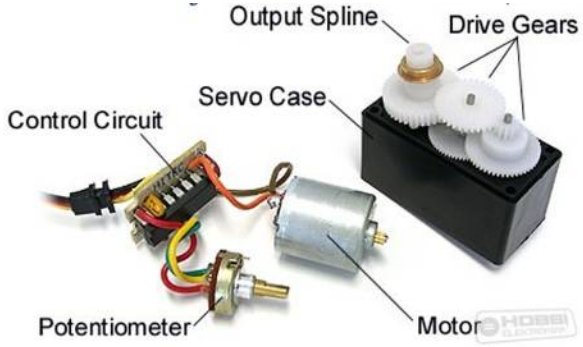
r, g, b: 0 – 255 közötti színkódok



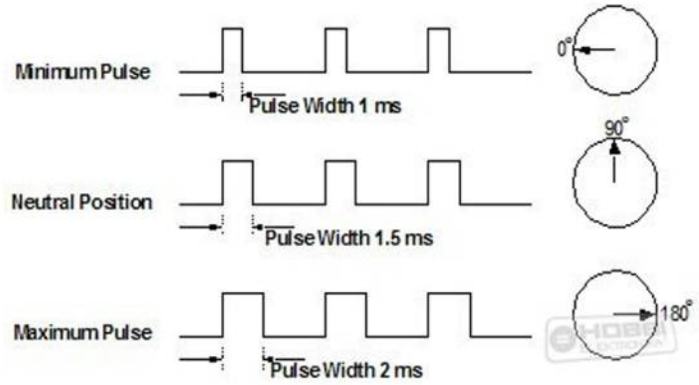
Közös anódú RGB LED bekötése

Szervo motorok

- A szervo egy pozícionálható motor, amely „ismeri” az aktuális pozícióját, és a cél pozíciót. Feladata, hogy az aktuális pozícióból a kívánt pozícióba álljon
- Felépítése: vezérlő áramkör, mechanikusan összekapcsolt motor és potenciométer (a potméterrel leosztott feszültség jelzi a pozíciót)
- Általában 180 °-os tartományban mozgatható, 1 – 2 ms szélességű, 50 Hz-es jellel vezérelhető (PWM



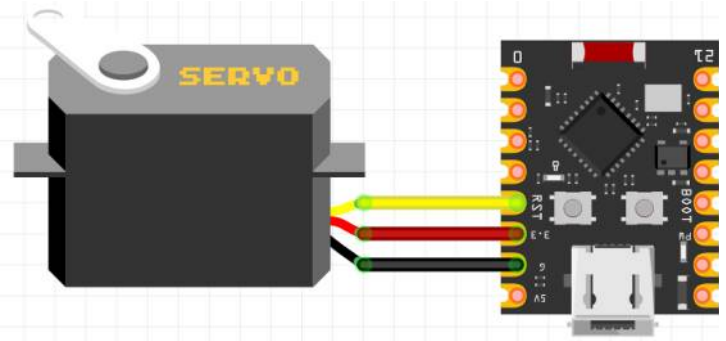
Felépítés



Jelalak

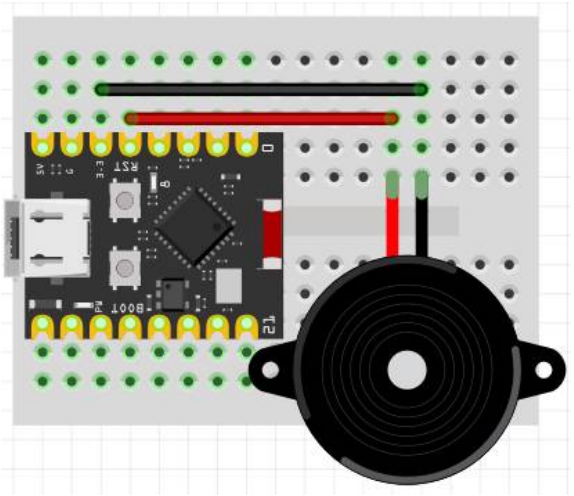
pwm_servo.py

```
import time
import board
import pwmio
# Create a PWMOut object on pin IO4
pwm = pwmio.PWMOut(board.IO4, frequency=50)
# Function to convert angle to duty cycle
def angle_to_duty_cycle(angle):
    # Convert angle (0-180) to pulse width (0.5-2.5 ms)
    pulse_width_ms = 0.5 + (2.0 * angle / 180)
    # Convert pulse width to duty cycle (0-65535)
    duty_cycle = int((pulse_width_ms / 20) * 65535)
    return duty_cycle

while True:
    for angle in range(0, 181, 10):  # Move servo from 0 to 180 degrees
        pwm.duty_cycle = angle_to_duty_cycle(angle)
        time.sleep(1)
    for angle in range(180, -1, -10):  # Move servo from 180 to 0 degrees
        pwm.duty_cycle = angle_to_duty_cycle(angle)
        time.sleep(1)
```



pwm_frequency.py

- Ha a program futása során meg akarjuk változtatni a PWM frekvenciát, akkor a PWMOut osztály példányosításakor jeleznünk kell ezt a ***variable_frequency*** paraméter ***True*** értékével (az alpértelmezett érték: False)
- Az alábbi példában két, hallható frekvenciájú jelet keltünk, amit egy piezo csipogóval meg is szólaltathatunk



```
import pwmio
import board
import time

pwm = pwmio.PWMOut(board.IO4, duty_cycle=2**15, frequency=440, variable_frequency=True)
time.sleep(1)
pwm.frequency = 880
time.sleep(1)
```

boci-boci.py – dallam lejátszása

```python
import pwmio
import board
import time

pwm = pwmio.PWMOut(board.IO4, duty_cycle=2**15, frequency=440, variable_frequency=True)

melody = [
    ("C", 2), ("E", 2), ("C", 2), ("E", 2),("G", 4), ("G", 4),
    ("C", 2), ("E", 2), ("C", 2), ("E", 2),("G", 4), ("G", 4),
    ("c", 2), ("H", 2), ("A", 2), ("G", 2),("F", 4), ("A", 4),
    ("G", 2), ("F", 2), ("E", 2), ("D", 2),("C", 4), ("C", 4)
]

tempo = 72
beat_duration = 60 / tempo / 4       # Duration of a 1/16 note

for note, duration in melody:
    frequency = NOTE_FREQUENCIES[note]
    pwm.frequency = int(frequency)
    pwm.duty_cycle = 2**15  # 50% duty cycle
    time.sleep(beat_duration * duration)
    pwm.duty_cycle = 0  # Turn off the buzzer between notes
    time.sleep(beat_duration * 0.1)  # Short pause between notes
```

```python
NOTE_FREQUENCIES = {
    "C": 261.63,  # C4
    "D": 293.66,  # D4
    "E": 329.63,  # E4
    "F": 349.23,  # F4
    "G": 392.00,  # G4
    "A": 440.00,  # A4
    "H": 493.88,  # H4
    "c": 523.25   # C5
}
```