Names:

**Question 1 (6 pnts)**

a. Network scheme of a brushed DC-motor.

A diagram of a circuit

AI-generated content may be incorrect.

b. Equation back EMF:

Equation motor torque:

c. Calculation of armature resistance.

d. Calculation of torque constant.

e. Calculation of efficiency

f. Relationship between the shaft speed in [rpm] and the encoder frequency in [Hz].

**Question 3 (6 pnts)**

a. Graph of CS+ at a duty-cycle of 66.7% and 50kHz switching frequency (Vin = 12V).

*A diagram of a voltage

AI-generated content may be incorrect.*

Student couples:

 Martin Marinov 2124573

Henryk Nowacki 2128013

b. Relation between average voltage on CS+ as a function of the duty-cycle at a 12V battery voltage.

c. Graphs of currents through the 4 mosfets at a 3 [A] load current (duty-cycle = 66.7%, fswitch =50kHz).

A graph paper with lines and squares

AI-generated content may be incorrect.

d. Selected mosfet:PMV20ENR

Link to datasheet:

Maximum drain-source voltage =30V Continuous drain current at 25°C = 7.6A

RDSON =21mΩ Threshold voltage (typical)= 2V

trise = 17ns tfall = 8ns

Unit price = 0.068 Dollars

e. Calculation of power losses and efficiency at: duty-cycle= 66.7%, fswitch =50kHz, Vin=12V and Iload = 3A.

) = 0.0945W

=98.9%

f. Purpose of C13 and C15:

They provide additional voltage to the upper mosfet

Connection of VS, C13 and C15:

VS must be connected to the **source of the high-side MOSFET (Q1/Q3)** and the **drain of the low-side MOSFET (Q2/Q4)**.

**Right-hand pin of C15** goes to **VS** (the switch node between Q3 and Q4).

**Right-hand pin of C13** is tied to **VS** (the switch node between Q1 and Q2).

**Question 4 (3 pnts)**

a. Relationship output voltage current sensor INA181A1IDBVT as a function of the voltages on IN+, IN- and REF.

b. Calculation of R1 and R38 in order to get VCS = 0.3V at -3A and VCS=-3.0V at +3A.

c. Calculation of C11 to get an attenuation of 100 at 50kHz.

**Question 5 (4 pnts)**

a. Calculation of R44

b. Calculation of cutoff- frequency input filter.

Calculation of cutoff- frequency output filter

c. Calculation of accuracy.

**The input bias current error is very small around 0.15µV therefore the difference is negligible**

**Question 6 (2 pnts)**

a. Calculation of R32 to obtain a 5V supply voltage.

b. Calculation power losses and temperature rise of the LMR14050S at a 12V input voltage and a 0.2A load current.

Efficiency = 90%

**Question 7 (1 pnt)**

Explanation what a pull-up resistor is.

Used to pull up the data line from 0v (0) to 3.3V(1)

Typical value of I2C pullup resistor:

4.7kΩ

**Question 8 (2 pnt)**

a. Calculation of the total current consumption of the RGB LED if all 3 leds are turned on.

b. Calculation of buzzer current at 3.3V.

Maximum current of a GPIO pin of the ESP32

20mA=max

**Question 9 (2 pnts)**

Fill in the last 3 columns of the table below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Signal | Description of signal | GPIO-pin ESP32 | Connector | Pin |
| 5.0V | Plus of 5V power supply (connected via diode, see figure 2.6) | - | J1\_ESP32 | 1 |
| 3.3V | Plus of 3.3V power supply | - | J2\_ESP32 | 1 |
| Ground | Power supply (minus) | - | J1\_ESP32 | 2 |
| Ground | Power supply (minus) | - | J2\_ESP32 | 2 |
| RX2 | Receive pin serial port connection ESP32 with high level controller | GPIO16 | J2\_ESP32 | 6 |
| TX2 | Transmit pin serial port connection ESP32 with high level controller | - | J2 | 7 |
| SCL | I2C clock signal (communication bus with module IMU MPU6050) | GPIO22 | J2 | 14 |
| SDA | I2C data signal (communication bus with module IMU MPU6050) | GPIO21 | J2 | 11 |
| enc\_A | Encoder output A | GPIO27 | J1 | 6 |
| enc\_B | Encoder output B | GPIO14 | J1 | 5 |
| pwm1 | PWM signal of half bridge 1 | GPIO25 | J1 | 8 |
| pwm2 | PWM signal of half bridge 2 | GPIO26 | J1 | 7 |
| enable | Enable signal of both half bridges | GPIO5 | J2 | 8 |
| CS | Analogue output voltage of current sensor | GPIO13 | J1 | 3 |
| VS | Analogue output voltage of voltage sensor | GPIO12 | J1 | 4 |
| button1 | Push button 1 | GPIO32 | J1 | 10 |
| button2 | Push button 2 | GPIO33 | J1 | 9 |
| button3 | Push button 3 | GPIO34 | J1 | 12 |
| green | Green LED of RGB LED | GPIO15 | J2 | 3 |
| red | Red LED of RGB LED | GPIO2 | J2 | 4 |
| blue | Blue LED of RGB LED | GPIO4 | J2 | 5 |
| buzzer | Signal that controls a buzzer | GPIO19 | J2 | 10 |

**Question 10 (1 pnts)**

What is a decoupling capacitor?

A decoupling capacitor's job is to supress high-frequency noise in power supply signals. They take tiny voltage ripples, which could otherwise be harmful to delicate ICs, out of the voltage supply.

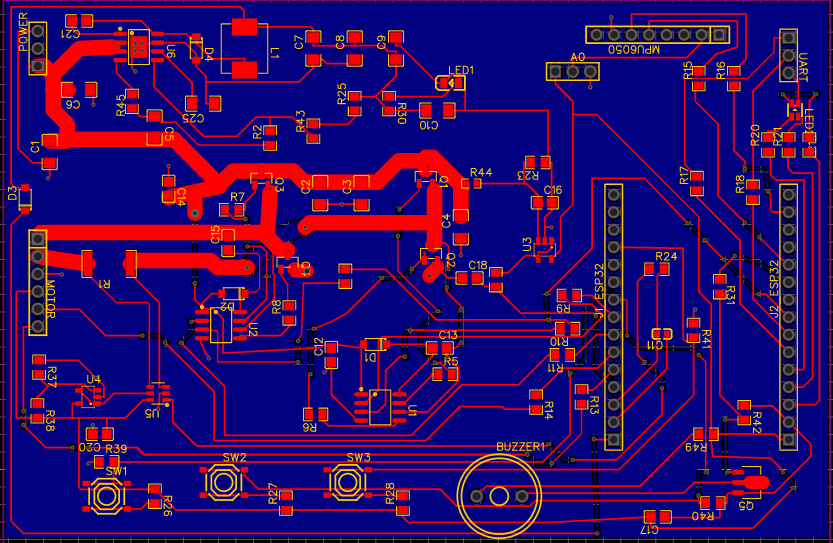
List all decoupling capacitors used in the hardware design of the motor controller.

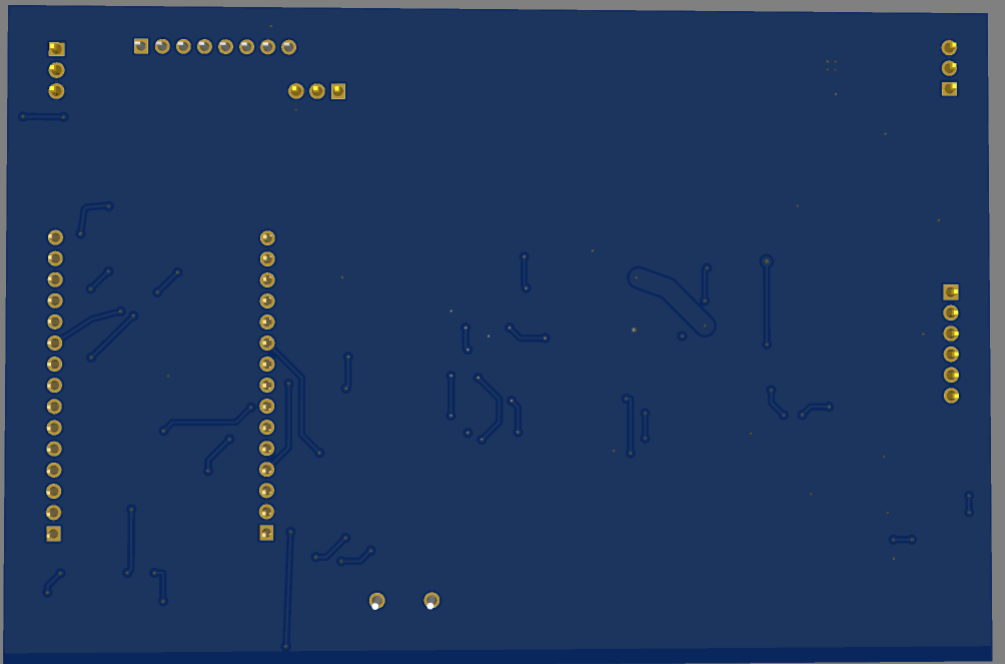
C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C12, C14, C16, C17, C20, C21, C25.

**Question 11 (20 pnts)**

Screen dump of PCB-design.

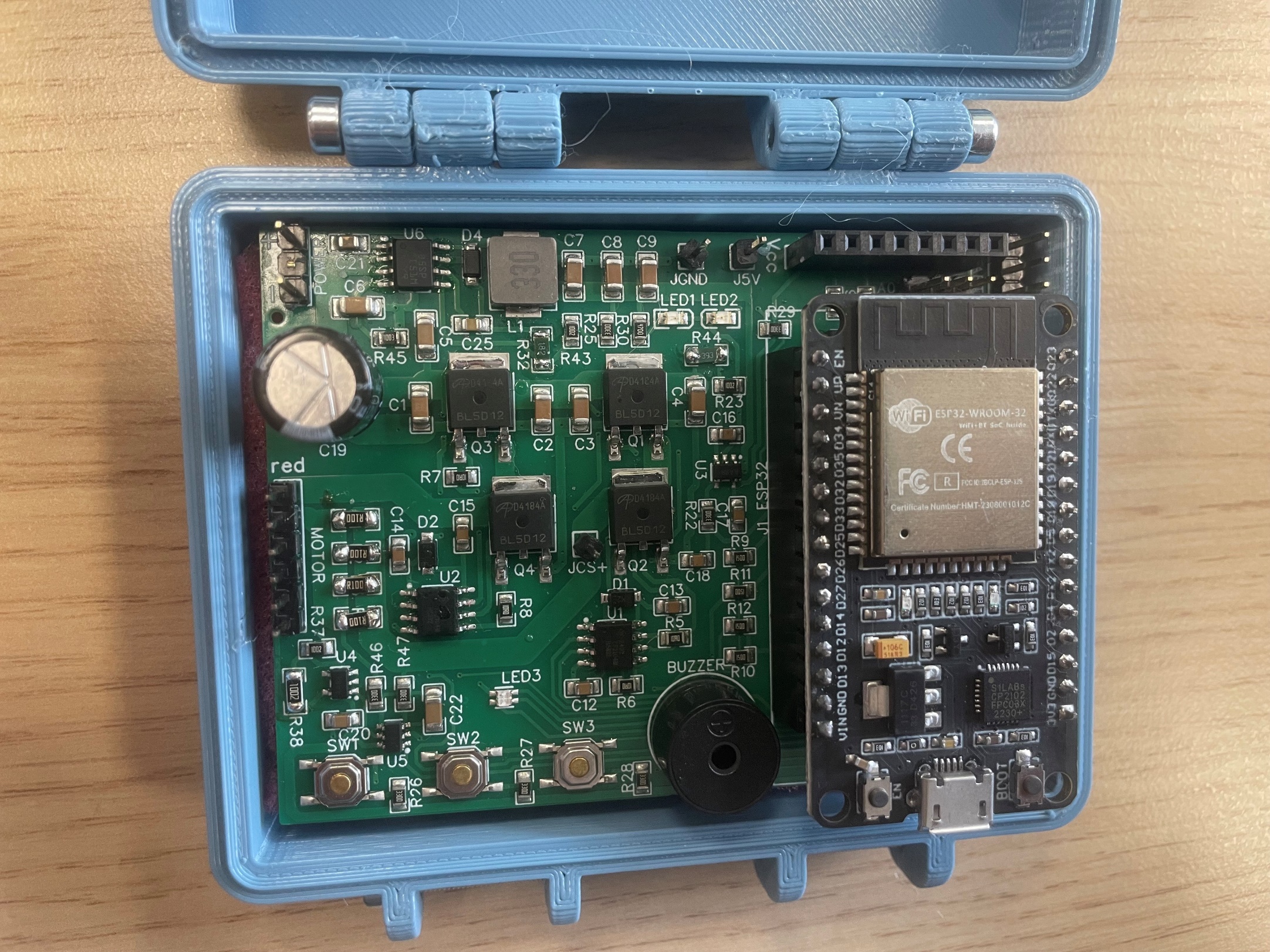
*Remark: A folder with the easyEDA project, the BOM, Ferber files and the Pick and Place file must also be turned in.*

**



**Question 12 (2 pnts)**

Picture(s) of PCB with resistors R1, R32, R38 and R44 soldered.

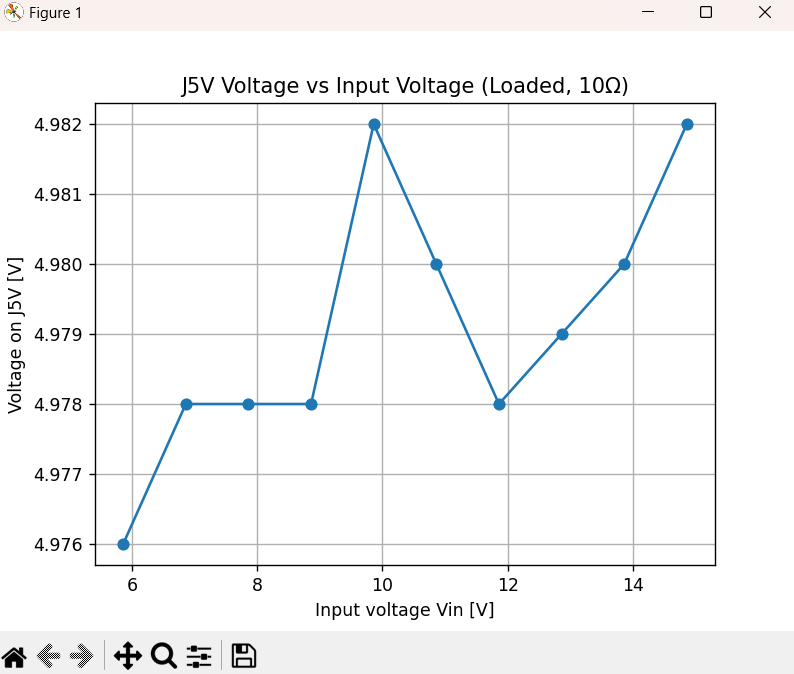
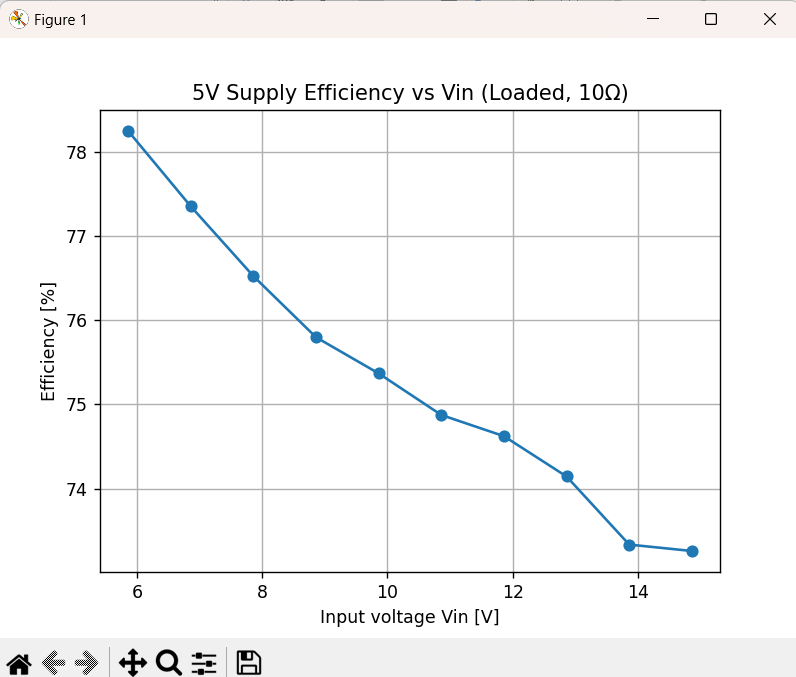


**Question 13 (4 pnt)**

a. Fill in the table below.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Supply voltage [V] (unloaded) | 5.86 | 6.86 | 7.86 | 8.86 | 9.86 | 10.86 | 11.86 | 12.86 | 13.86 | 14.86 |
| Voltage on J5V [V] (unloaded) | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 |
| Supply current [mA] (unloaded) | 95 | 83 | 74 | 67 | 60 | 55 | 52 | 48 | 45 | 43 |
| Supply voltage [V] (loaded) | 5.86 | 6.86 | 7.86 | 8.86 | 9.86 | 10.86 | 11.86 | 12.86 | 13.86 | 14.86 |
| Voltage on J5V [V] (loaded) | 4.976 | 4.978 | 4.979 | 4.978 | 4.982 | 4.98 | 4.978 | 4.979 | 4.99 | 4.982 |
| Supply current [mA] (loaded) | 540 | 467 | 412 | 369 | 334 | 305 | 280 | 260 | 244 | 228 |

b. Plot of voltage and efficiency of the 5V-supply as a function of the input voltage at a 10



Python script to make the plots. *Remark: The python script file must also be turned in a folder named Q13.*

import matplotlib.pyplot as plt  
  
# Data from the photo — LOADED situation  
vin = [5.86, 6.86, 7.86, 8.86, 9.86, 10.86, 11.86, 12.86, 13.86, 14.86] # Input voltage [V]  
vj5 = [4.976, 4.978, 4.978, 4.978, 4.982, 4.98, 4.978, 4.979, 4.98, 4.982] # J5V voltage [V]  
iin\_mA = [540, 467, 412, 369, 334, 305, 280, 260, 244, 228] # Supply current (loaded) [mA] — from photo  
  
# --- Efficiency ---  
  
iin = [x/1000 for x in iin\_mA] # A  
iout = [v/10.0 for v in vj5] # A, from 10Ω load  
efficiency = [(v\*io)/(vi\*ii) if (vi\*ii)!=0 else 0.0 for v, io, vi, ii in zip(vj5, iout, vin, iin)]  
  
# Plot J5V voltage vs input voltage  
plt.figure()  
plt.plot(vin, vj5, marker='o')  
plt.xlabel('Input voltage Vin [V]')  
plt.ylabel('Voltage on J5V [V]')  
plt.title('J5V Voltage vs Input Voltage (Loaded, 10Ω)')  
plt.grid(True)  
plt.show()  
  
# Plot Efficiency vs input voltage  
plt.figure()  
plt.plot(vin, [e\*100 for e in efficiency], marker='o')  
plt.xlabel('Input voltage Vin [V]')  
plt.ylabel('Efficiency [%]')  
plt.title('5V Supply Efficiency vs Vin (Loaded, 10Ω)')  
plt.grid(True)  
plt.show()

**Question 14 (4 pnts)**

Python script files boot.py and main.py to test the user interface.

*Remark: Both python script files must also be submitted in a folder named Q14.*

Contents of boot.py:

# boot.py — run once at boot, before main.py

import micropython

# Allow clear error messages if something goes wrong inside IRQs

micropython.alloc\_emergency\_exception\_buf(100)

print("boot.py: system ready")

Contents of main.py:

# main.py — ESP32 MicroPython (Thonny)

from machine import Pin, PWM, Timer

import micropython

import utime

# ---------------- Pin map (from assignment) ----------------

LED\_R\_PIN = 19 # Red

LED\_G\_PIN = 5 # Green

LED\_B\_PIN = 18 # Blue

SW1\_PIN = 39 # Button 1 -> Red (GPIO36/39 are input-only, no internal pulls)

SW2\_PIN = 36 # Button 2 -> Green

SW3\_PIN = 15 # Button 3 -> Blue (has internal pull-up if needed)

BUZZER\_PIN = 4

# ---------------- Configuration ----------------

INVERT\_OUTPUT = False # set True if LEDs are inverted on your PCB

DEBOUNCE\_MS = 120

SIREN\_FREQS = (375, 500)

SIREN\_PERIOD\_MS = 400 # ms

# ---------------- Safety for IRQs ----------------

micropython.alloc\_emergency\_exception\_buf(100)

# ---------------- LED helpers ----------------

\_led\_r = Pin(LED\_R\_PIN, Pin.OUT)

\_led\_g = Pin(LED\_G\_PIN, Pin.OUT)

\_led\_b = Pin(LED\_B\_PIN, Pin.OUT)

def \_led\_write(pin\_obj, on):

# If outputs are inverted (e.g., common-anode), flip the level

pin\_obj.value(1 if (on ^ INVERT\_OUTPUT) else 0)

def \_led\_read(pin\_obj):

raw = pin\_obj.value()

return (raw == 1) ^ INVERT\_OUTPUT

# start with LEDs OFF

\_led\_write(\_led\_r, False)

\_led\_write(\_led\_g, False)

\_led\_write(\_led\_b, False)

# ---------------- Buzzer (PWM) ----------------

\_buzzer\_pwm = PWM(Pin(BUZZER\_PIN))

\_buzzer\_pwm.freq(SIREN\_FREQS[0])

def \_buzzer\_on():

# duty compatibility: some ports have duty(), others duty\_u16()

try:

\_buzzer\_pwm.duty(512) # 0..1023 on ESP32 classic

except AttributeError:

\_buzzer\_pwm.duty\_u16(32768) # 0..65535

def \_buzzer\_off():

try:

\_buzzer\_pwm.duty(0)

except AttributeError:

\_buzzer\_pwm.duty\_u16(0)

\_buzzer\_off()

\_siren\_timer = Timer(0)

\_siren\_idx = 0

\_siren\_active = False

def \_siren\_timer\_cb(t):

global \_siren\_idx

\_siren\_idx ^= 1

\_buzzer\_pwm.freq(SIREN\_FREQS[\_siren\_idx])

def \_siren\_start():

global \_siren\_active, \_siren\_idx

if \_siren\_active:

return

\_siren\_idx = 0

\_buzzer\_pwm.freq(SIREN\_FREQS[\_siren\_idx])

\_buzzer\_on()

\_siren\_timer.init(period=SIREN\_PERIOD\_MS, mode=Timer.PERIODIC, callback=\_siren\_timer\_cb)

\_siren\_active = True

def \_siren\_stop():

global \_siren\_active

if not \_siren\_active:

return

\_siren\_timer.deinit()

\_buzzer\_off()

\_siren\_active = False

def \_recompute\_siren():

if \_led\_read(\_led\_r) and \_led\_read(\_led\_g) and \_led\_read(\_led\_b):

\_siren\_start()

else:

\_siren\_stop()

def \_toggle\_led\_and\_update(led\_pin\_obj):

\_led\_write(led\_pin\_obj, not \_led\_read(led\_pin\_obj))

\_recompute\_siren()

# ---------------- Buttons (interrupt + debounce) ----------------

# Expect the PCB to provide pulls for 36/39. We enable pull-up on 15.

\_sw1 = Pin(SW1\_PIN, Pin.IN)

\_sw2 = Pin(SW2\_PIN, Pin.IN)

\_sw3 = Pin(SW3\_PIN, Pin.IN, Pin.PULL\_UP)

\_last\_sw1\_ms = 0

\_last\_sw2\_ms = 0

\_last\_sw3\_ms = 0

def \_pressed(pin\_obj):

# Treat "pressed" as LOW (active-low typical). If your PCB is active-high, change to == 1.

return pin\_obj.value() == 0

# scheduler targets (avoid lambda allocations in IRQ)

def \_sched\_toggle\_r(\_):

\_toggle\_led\_and\_update(\_led\_r)

def \_sched\_toggle\_g(\_):

\_toggle\_led\_and\_update(\_led\_g)

def \_sched\_toggle\_b(\_):

\_toggle\_led\_and\_update(\_led\_b)

def \_irq\_sw1(pin):

global \_last\_sw1\_ms

now = utime.ticks\_ms()

if utime.ticks\_diff(now, \_last\_sw1\_ms) < DEBOUNCE\_MS:

return

\_last\_sw1\_ms = now

if \_pressed(\_sw1):

micropython.schedule(\_sched\_toggle\_r, 0)

def \_irq\_sw2(pin):

global \_last\_sw2\_ms

now = utime.ticks\_ms()

if utime.ticks\_diff(now, \_last\_sw2\_ms) < DEBOUNCE\_MS:

return

\_last\_sw2\_ms = now

if \_pressed(\_sw2):

micropython.schedule(\_sched\_toggle\_g, 0)

def \_irq\_sw3(pin):

global \_last\_sw3\_ms

now = utime.ticks\_ms()

if utime.ticks\_diff(now, \_last\_sw3\_ms) < DEBOUNCE\_MS:

return

\_last\_sw3\_ms = now

if \_pressed(\_sw3):

micropython.schedule(\_sched\_toggle\_b, 0)

EDGE = Pin.IRQ\_FALLING | Pin.IRQ\_RISING

\_sw1.irq(trigger=EDGE, handler=\_irq\_sw1)

\_sw2.irq(trigger=EDGE, handler=\_irq\_sw2)

\_sw3.irq(trigger=EDGE, handler=\_irq\_sw3)

print("main.py: running — press SW1/SW2/SW3 to toggle R/G/B; all ON => siren")

# keep alive; work happens in IRQs

while True:

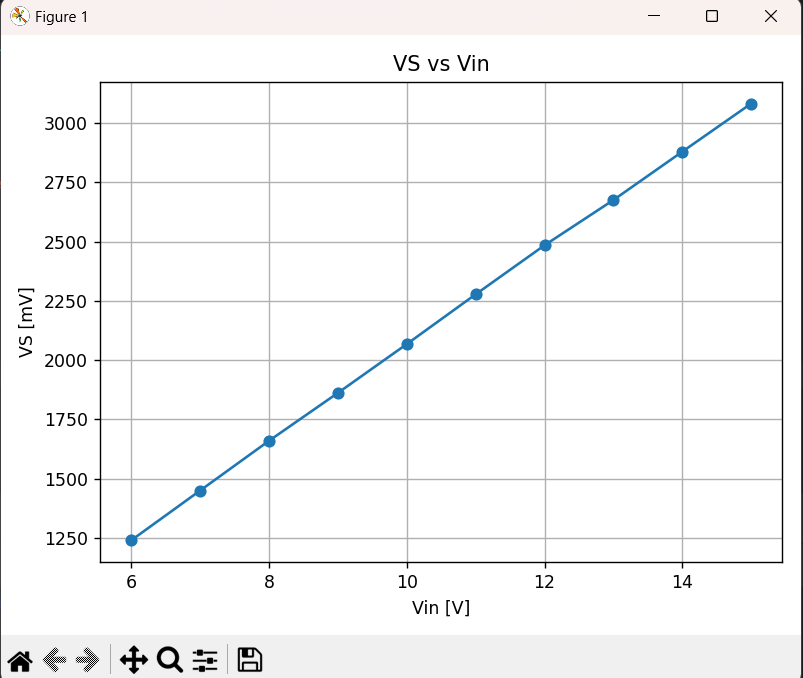
utime.sleep\_ms(250)

**Question 15 (4 pnts)**

a. Fill in the table below (measurement of VS as a function of Vin).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vin [V]  (multimeter) | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| VS [mV]  (ESP32) | 1240 | 1450 | 1660 | 1862 | 2068 | 2278 | 2485 | 2676 | 2880 | 3083 |

b. Graph of VS as a function of Vin.



c. Average proportional factor of Vin/VS determined from measurements. c1 = 4.839695

Average proportional factor of Vin/VS determined from R23 and R44. c1 = 4.9

Difference between both values in percentage = 1.23%

Given an explanation if the difference is more than 3%.

Python script to make the plot of question b and to calculate the average ratio of Vin / VS

*Remark: This file must also be submitted in a folder named Q15.py*

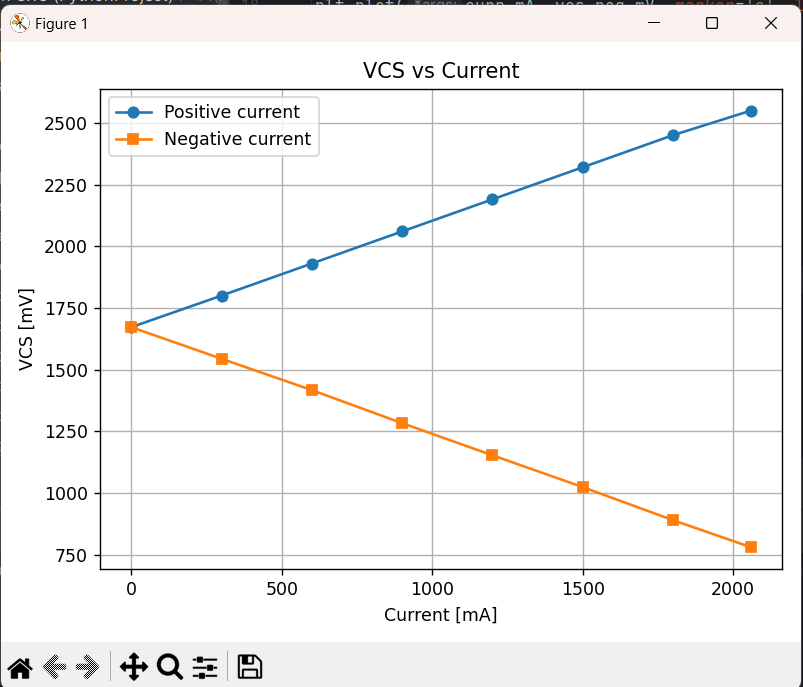
# q15\_plot\_and\_ratio.py  
# Requirements: numpy, matplotlib  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
# ---- measurements (from the table) ----  
Vin\_V = np.array([6, 7, 8, 9, 10, 11, 12, 13, 14, 15], dtype=float)  
VS\_mV = np.array([1240, 1450, 1660, 1862, 2068, 2278, 2485, 2676, 2880, 3083], dtype=float)  
  
# ---- (b) Plot VS vs Vin ----  
plt.figure()  
plt.plot(Vin\_V, VS\_mV, marker='o')  
plt.xlabel('Vin [V]')  
plt.ylabel('VS [mV]')  
plt.title('VS vs Vin')  
plt.grid(True)  
plt.tight\_layout()  
plt.show()  
  
# ---- (c) Compute c1 from measurements and from theory ----  
VS\_V = VS\_mV / 1000.0  
  
# Average proportional factor (measured): mean(Vin / VS)  
c1\_meas = np.mean(Vin\_V / VS\_V)  
  
# Theoretical c1 from divider:  
# VS = Vin \* R\_bottom / (R\_top + R\_bottom)  
# => Vin/VS = (R\_top + R\_bottom) / R\_bottom  
# Use the orientation that matches your board: top=39k, bottom=10k  
R\_top = 39\_000.0 # ohms (R23 physically at Vin side)  
R\_bottom = 10\_000.0 # ohms (R44 to GND)  
c1\_theory = (R\_top + R\_bottom) / R\_bottom  
  
# Percent difference  
diff\_pct = 100.0 \* (c1\_meas - c1\_theory) / c1\_theory  
  
print(f"Measured c1 (mean Vin/VS): {c1\_meas:.6f}")  
print(f"Theoretical c1 from R\_top={R\_top/1000:.0f}k, R\_bottom={R\_bottom/1000:.0f}k: {c1\_theory:.6f}")  
print(f"Difference (meas - theory) / theory: {diff\_pct:+.2f}%")

**Question 16 (4 pnts)**

a. Fill in the tables below (Measurement of VCS as a function of positive and negative current )

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| I [A]  (multimeter) | 0 | 0.3 | 0.6 | 0.9 | 1.2 | 1.5 | 1.8 | 2.06 |  |
| VCS [V]  (ESP32) | 1.672 | 1.8 | 1.93 | 2.06 | 2.19 | 2.32 | 2.45 | 2.55 |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| I [A]  (multimeter) | 0 | 0.3 | 0.6 | 0.9 | 1.2 | 1.5 | 1.8 | 2.06 |  |
| VCS [V]  (ESP32) | 1.672 | 1.544 | 1.417 | 1.283 | 1.153 | 1.024 | 0.89 | 0.78 |  |

b. Plot of VCS as a function of the current.

c. Values of I0 and c2 found via linear regression:

I0 = -3.8736 A c2 = 2.3182 A/V

Theoretical values of I0 and c2 based on values of R1 and R38:

I0 = -3.3 A c2 = 2 A/V

Difference between both values of I0 in percentage = 15.91%

Difference between both values of c2 in percentage = 17.38%

Given an explanation if the difference is more than 3%.

*Remark: The python script must also be submitted in a folder named Q16.py*

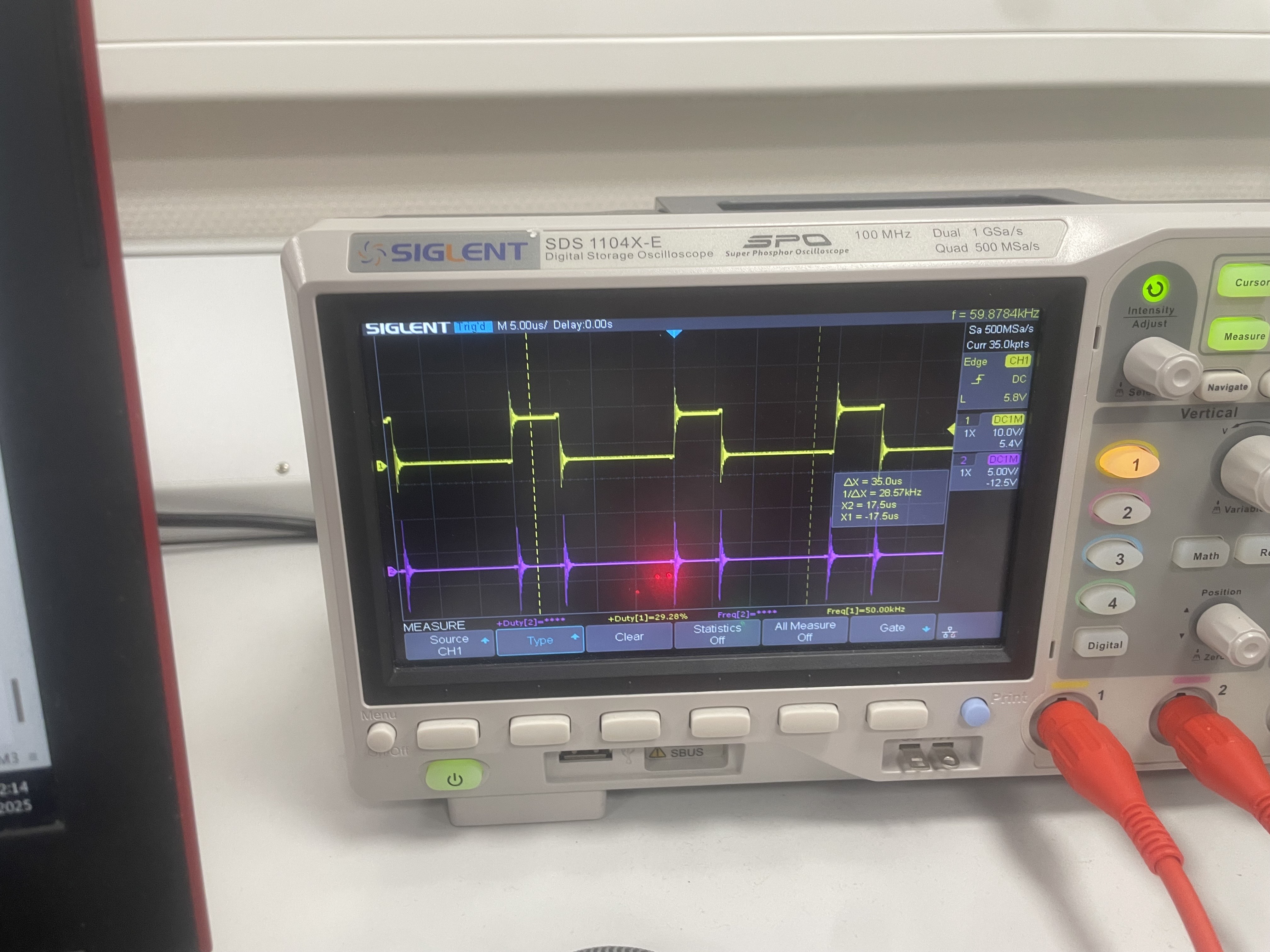
import numpy as np  
import matplotlib.pyplot as plt  
  
# -----------------------------  
# Measured data from 16b  
# -----------------------------  
# Current sweep [A] and VCS [V] for +I  
I\_pos = np.array([0, 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.06])  
V\_pos = np.array([1.672, 1.8, 1.93, 2.06, 2.19, 2.32, 2.45, 2.55])  
  
# Current sweep [A] and VCS [V] for –I (currents negated)  
I\_neg = -np.array([0, 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.06])  
V\_neg = np.array([1.672, 1.544, 1.417, 1.283, 1.153, 1.024, 0.89, 0.78])  
  
# Combine both polarities  
V = np.concatenate([V\_pos, V\_neg]) # VCS [V]  
I = np.concatenate([I\_pos, I\_neg]) # Current [A]  
  
# -----------------------------  
# Linear regression: I = I0 + c2 \* VCS  
# -----------------------------  
c2, I0 = np.polyfit(V, I, 1) # slope, intercept  
  
# -----------------------------  
# Theoretical expectations  
# -----------------------------  
# INA181A1 gain and shunt; 3.3 V rail with 10k:10k divider -> Vref = ~1.65 V  
G = 20.0 # INA181A1 gain (change if your variant differs)  
R1 = 25e-3 # 25 mΩ shunt  
Vcc = 3.3 # ESP32/board supply  
R\_top = 10e3 # divider top resistor (e.g., R37)  
R\_bottom = 10e3 # divider bottom resistor (R38 = 10 kΩ)  
Vref = Vcc \* (R\_bottom / (R\_top + R\_bottom))  
  
c2\_th = 1.0 / (G \* R1) # A/V  
I0\_th = -Vref / (G \* R1) # A  
  
# Percentage differences  
def pct(meas, th):  
 return 100.0 \* (meas - th) / th  
  
c2\_diff\_pct = pct(c2, c2\_th)  
I0\_diff\_pct = pct(I0, I0\_th)  
  
# -----------------------------  
# Print results  
# -----------------------------  
print(f"Fitted model: I = I0 + c2\*VCS")  
print(f" I0 (fit) = {I0:.4f} A")  
print(f" c2 (fit) = {c2:.4f} A/V")  
print()  
print(f"Theory (G={G:.0f}, R1={R1\*1e3:.1f} mΩ, Vref={Vref:.3f} V):")  
print(f" I0\_th = {I0\_th:.4f} A")  
print(f" c2\_th = {c2\_th:.4f} A/V")  
print()  
print(f"Percent difference:")  
print(f" ΔI0 = {c2\_diff\_pct:.2f}%")  
print(f" Δc2 = {I0\_diff\_pct:.2f}%")  
  
# -----------------------------  
# Plot in requested units (mA vs mV)  
# -----------------------------  
plt.figure()  
plt.scatter(V\*1000, I\*1000, marker='x', label='Measured points')  
V\_line = np.linspace(V.min()-0.1, V.max()+0.1, 200)  
I\_fit = I0 + c2 \* V\_line  
plt.plot(V\_line\*1000, I\_fit\*1000, label='Linear fit')  
  
plt.xlabel('VCS [mV]')  
plt.ylabel('Current [mA]')  
plt.title('I = I0 + c2·VCS (linear regression)')  
plt.grid(True)  
plt.legend()  
plt.tight\_layout()  
plt.show()

**Question 17 (3 pnt)**

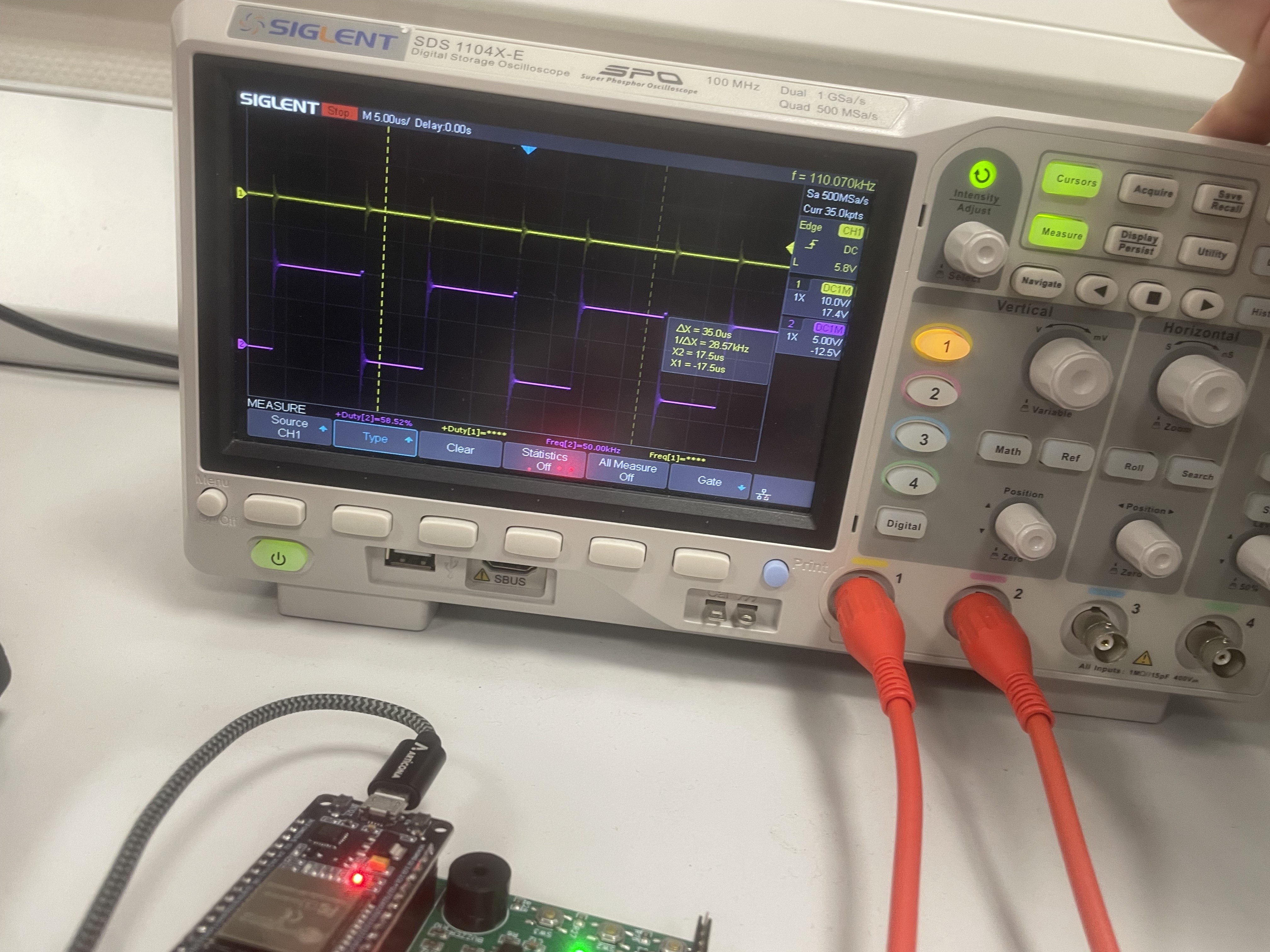
a. Fill in the table below

|  |  |  |  |
| --- | --- | --- | --- |
|  | Frequency scope- voltage [kHz] | Duty-cycle bridge [%] | Make scope image |
| pwm1 = 300, pwm2 = 0 | 50 | 29.3 | Yes |
| pwm1 = 600, pwm2 = 0 | 50 | 58.5 | No |
| pwm1 = 900, pwm2 = 0 | 50 | 87.8 | No |
| pwm1 = 0, pwm2 = 300 | 50 | 29.3 | No |
| pwm1 = 0, pwm2 = 600 | 50 | 58.5 | Yes |
| pwm1 = 0, pwm2 = 900 | 50 | 87.8 | No |

Scope-image at pwm1 = 300, pwm2 = 0.



Scope-image at pwm1 = 0, pwm2 = 600.



b. Maximum value of duty-cycle input of half-bridge 1 = 996=99.8%

Maximum value of duty-cycle input of half-bridge 2 = -997=97.3%

**Question 18 (4 pnts)**

a. Fill in the table below (encoder frequency and motor current as a function of the duty-cycle input.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Half bridge 1 active (pwm2 = 0) | | | Half bridge 2 active (pwm1 = 0) | | |
| duty-cycle | I [A] | Encoder frequency [Hz] | duty-cycle | I[A] | Encoder frequency [Hz] |
| 0 |  |  | 0 |  |  |
| 100 | 0.109 | 251.6 | -100 | -0.102 | -230 |
| 200 | 0.130 | 525.8 | -200 | -0.132 | -510 |
| 300 | 0.137 | 775.8 | -300 | -0.139 | -800 |
| 400 | 0.160 | 1002 | -400 | -0.139 | -1080 |
| 500 | 0.167 | 1434.7 | -500 | -0.153 | -1380 |
| 600 | 0.188 | 1666 | -600 | -0.153 | -1670 |
| 700 | 0.195 | 1838.2 | -700 | -0.176 | -1950 |
| 800 | 0.195 | 1930 | -800 | -0.171 | -2200 |
| 900 | 0.197 | 2105 | -900 | -0.183 | -2550 |

b. Armature resistance = Ra = 1.460445 ohm

Torque constant = km = 0.010963 V·s/rad (== 0.010963 N·m/A)

Script file used to find the armature resistance and torque constant.

*Remark: The script file must also be submitted in a folder named Q18.*

# Q18: PWMs + on-demand encoder & current readout

from time import sleep\_ms

from Encoder import GetFrequency

# Objects created in boot.py: HB\_ENABLE, pwm1, pwm2, adc\_cs

# --- Use measured model from Q16c ---

I0 = -3.8736 # [A]

c2 = 2.3182 # [A/V]

def clamp(n, lo, hi):

return lo if n < lo else hi if n > hi else n

def set\_pwm(dc):

"""Map dc (-950..950) to the two legs."""

if dc >= 0:

pwm1.duty(dc); pwm2.duty(0)

else:

pwm1.duty(0); pwm2.duty(-dc)

def read\_current\_A():

# Use accurate ADC: read\_uv -> volts

vcs\_V = adc\_cs.read\_uv() / 1\_000\_000

return I0 + c2 \* vcs\_V, vcs\_V

print("Q18 ready. Enter duty -950..950. Press ENTER (empty) to read encoder and current.")

HB\_ENABLE.value(1) # enable half-bridges (LED2 ON). :contentReference[oaicite:3]{index=3}

while True:

try:

s = input("duty or ENTER to read: ").strip()

# Empty input -> show encoder freq and motor current

if s == "":

f\_hz = GetFrequency() # encoder freq (signed by direction) :contentReference[oaicite:4]{index=4}

I\_A, vcs\_V = read\_current\_A() # current via I = I0 + c2\*VCS :contentReference[oaicite:5]{index=5}

print("Encoder = {:>6.1f} Hz | VCS = {:>5.3f} V | I = {:>+6.3f} A".format(f\_hz, vcs\_V, I\_A))

continue

# Number entered -> set duty

dc = int(s)

dc = clamp(dc, -950, 950)

set\_pwm(dc)

# small pause so terminal stays readable

sleep\_ms(30)

except KeyboardInterrupt:

print("\nStopping.")

pwm1.duty(0); pwm2.duty(0); HB\_ENABLE.value(0)

break

except Exception as e:

print("Please enter an integer in -950..950 (or just press ENTER). Error:", e)

from machine import Pin, PWM, ADC

from Encoder import InitEncoder, GetFrequency # encoder support

# --- Half-bridge enable & ADCs from Q15/16 ---

HB\_ENABLE = Pin(13, Pin.OUT) # LED2 mirrors this state

adc\_cs = ADC(Pin(32)); adc\_cs.atten(ADC.ATTN\_11DB); adc\_cs.width(ADC.WIDTH\_12BIT)

# --- PWMs from Q17 (50 kHz) ---

PWM\_FREQ = 50\_000

pwm1 = PWM(Pin(25), freq=PWM\_FREQ, duty=0) # half-bridge 1

pwm2 = PWM(Pin(26), freq=PWM\_FREQ, duty=0) # half-bridge 2

# --- Encoder init for encA=34, encB=35 (run once at boot) ---

InitEncoder(34, 35) # Appendix C says encA/encB on GPIO34/35. :contentReference[oaicite:2]{index=2}

# Start disabled for safety; main.py will enable

HB\_ENABLE.value(0)

**Question 19 (4 pnts)**

a. Which signals are involved with I2C?

SCL (clock), SDA (data), both open-drain with pull-ups.

What is an I2C address? What is its range?

Usually **7-bit (0x00–0x7F)**; some reserved. 10-bit exists.

How is the start bit and stop bit of I2C defined?

START = SDA↓ while SCL=1; STOP = SDA↑ while SCL=1.

What is the contents of the first byte of an I2C transfer?

7-bit address (MSB→LSB) + **R/W** bit (0=write, 1=read), then ACK

b. Timing diagram of SCL and SDA when an I2C master writes 1 data (150) to a slave with address 54.

SCL: ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ... (clocks)

┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─ └─┘

SDA: [ Address+W = 0b 0 1 1 0 1 1 0 0 ] [A]

0 1 1 0 1 1 0 0 0

[ Data = 0b 1 0 0 1 0 1 1 0 ] [A]

1 0 0 1 0 1 1 0 0

P: STOP (SDA↑ while SCL=1)

c. I2C address of MPU6050 =

- **0x68** when AD0 pin = **0** (default).

- **0x69** when AD0 pin = **1**.

d. How long does it take to read the 3 acceleration forces via a burst read at a baud rate of 400k bit/s.

You read 6 bytes (XH, XL, YH, YL, ZH, ZL) starting at ACCEL\_XOUT\_H:

Transaction (ideal, no stretching):

* START
* **SLA+W** (1 byte) + **ACK** → 9 bits
* **Register address** (1 byte) + **ACK** → 9 bits
* Repeated START
* **SLA+R** (1 byte) + **ACK** → 9 bits
* **Data**: 6 bytes = 6×8 = 48 bits
* **ACK/NACKs**: 6 bits (ACK after first 5, NACK after last)  
  **Total bits = 9 + 9 + 9 + (48 + 6) = 81 bits**

At **400,000 bits/s**, bit time = **2.5 µs** ⇒  
**81 × 2.5 µs ≈ 202.5 µs ≈ 0.203 ms** (ideal bus time; excludes any clock stretching or firmware overhead).

**Question 20 (3 pnts)**

b. Explain what happens when the following command is executed: i2c.writeto(self.addr, bytearray([107, 1])).

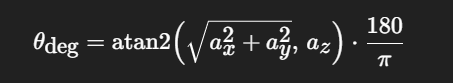
This command **wakes the MPU-6050 and selects the X-gyro PLL as its clock** by writing PWR\_MGMT\_1 = 0x01.

c. Equation to convert the 16-bits raw data of acceleration to [m/s2] (code line 41 of MPU6050\_studentversion.py).



acc\_mps2 = (raw / 16384.0) \* 9.80665 # for ±2g

d. Equation to calculate the tilt angle [°] (code line 47 of MPU6050\_studentversion.py).



tilt\_deg = math.degrees(math.atan2(math.sqrt(ax\*ax + ay\*ay), az))

e. Fill in the tables below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Angle MPU6050 board in x-direction [°]1 | Measured tilt angle [°] | acc[0] [m/s2] x | acc[1] [m/s2] y | acc[2] [m/s2] z |
| 0 |  |  |  |  |
| 15 | 11.51 | 9.62 | 0.28 | 1.93 |
| 30 | 27.41 | 8.52 | 0.29 | 4.41 |
| 45 | 45.93 | 7.05 | 0.23 | 7.28 |
| 60 |  |  |  |  |
| 75 | 72.99 | 2.63 | 0.49 | 8.59 |
| 90 | 89.85 | 0.02 | 0 | 9.16 |
| Angle MPU6050 board in y-direction [°] | Measured tilt angle [°] | acc[0] [m/s2] | acc[1] [m/s2] | acc[2] [m/s2] |
| 15 | 14.29 | 9.67 | -2.4 | -0.56 |
| 30 | 28.98 | 8.64 | -4.76 | -0.44 |
| 45 | 43.43 | 7.17 | -6.78 | -0.42 |
| 60 |  |  |  |  |
| 75 | 72.77 | 2.95 | -9.48 | -0.75 |
| 90 | 90.84 | -0.15 | -9.88 | -0.8 |

**Question 21 (2 pnts)**

a. Calculate how long it takes to transmit the string “Hello world” at a baud-rate of 115.2kbit/s, 8 data bits, 1 stop bit and no parity bit

Hello world consists of 10 characters including the space. Every character has 10 bits (1start bit, 8 data bits and 1 stop bit), so the time that it takes to transmit this message is 110/115.2 = 0.955ms

b. Code of main.py. *Remark: All python files of this program must also be submitted in a folder named Q21.*

from machine import UART, Pin, PWM

import time

def now\_ms() -> int:

return time.ticks\_ms()

class UartConsole:

"""Minimal wrapper around UART for CRLF lines and non-blocking char reads."""

def \_\_init\_\_(self, uart: UART):

self.uart = uart

self.\_buf1 = bytearray(1)

def read\_char(self):

"""Return one byte (int 0..255) if available, else None (non-blocking)."""

n = self.uart.readinto(self.\_buf1)

if n:

return self.\_buf1[0]

return None

def send\_line(self, text: str):

self.uart.write(text + "\r\n")

class DebouncedButton:

"""

Active-low button with simple debounce.

Use update(ts) on each loop; when it returns True you got a 'pressed' event (rising edge).

"""

def \_\_init\_\_(self, pin: Pin, debounce\_ms: int):

self.pin = pin

self.debounce\_ms = debounce\_ms

self.\_last\_raw = self.\_pressed\_raw()

self.\_next\_ok\_ts = 0

def \_pressed\_raw(self) -> bool:

# Active-low: logic 0 means pressed. Adjust here if hardware differs.

return self.pin.value() == 0

def update(self, ts\_ms: int) -> bool:

raw = self.\_pressed\_raw()

if raw != self.\_last\_raw:

# Edge detected

if raw and time.ticks\_diff(ts\_ms, self.\_next\_ok\_ts) >= 0:

# Rising edge into 'pressed'

self.\_next\_ok\_ts = time.ticks\_add(ts\_ms, self.debounce\_ms)

self.\_last\_raw = raw

return True

self.\_next\_ok\_ts = time.ticks\_add(ts\_ms, self.debounce\_ms)

self.\_last\_raw = raw

return False

class Led:

def \_\_init\_\_(self, pin\_no: int, initial=False):

self.pin = Pin(pin\_no, Pin.OUT, value=1 if initial else 0)

self.state = bool(initial)

def set(self, on: bool):

self.state = bool(on)

self.pin.value(1 if self.state else 0)

def toggle(self):

self.set(not self.state)

class Buzzer:

def \_\_init\_\_(self, pin\_no: int, freq\_hz: int, duty\_on: int):

self.pwm = PWM(Pin(pin\_no))

self.freq\_hz = freq\_hz

self.duty\_on = duty\_on

self.\_on = False

self.off()

def on(self):

self.pwm.freq(self.freq\_hz)

self.pwm.duty(self.duty\_on)

self.\_on = True

def off(self):

self.pwm.duty(0)

self.\_on = False

def toggle(self):

if self.\_on:

self.off()

else:

self.on()

# Application

class App:

def \_\_init\_\_(self):

# LEDs

self.led\_R = Led(PIN\_RED, initial=False)

self.led\_G = Led(PIN\_GREEN, initial=False)

self.led\_B = Led(PIN\_BLUE, initial=False)

# Buttons

# Note: 36/39 are input-only, external pulls; 15 supports internal pull-up.

self.btn1 = DebouncedButton(Pin(PIN\_SW1, Pin.IN), DEBOUNCE\_MS)

self.btn2 = DebouncedButton(Pin(PIN\_SW2, Pin.IN), DEBOUNCE\_MS)

self.btn3 = DebouncedButton(Pin(PIN\_SW3, Pin.IN, Pin.PULL\_UP), DEBOUNCE\_MS)

# Buzzer

self.buzzer = Buzzer(PIN\_BUZZER, BUZZER\_FREQ\_HZ, BUZZER\_DUTY\_ON)

# UART

self.uart = UartConsole(UART(

UART\_ID,

baudrate=UART\_BAUDRATE,

bits=UART\_BITS,

parity=UART\_PARITY,

stop=UART\_STOP,

rx=PIN\_RX2,

tx=PIN\_TX2,

timeout=UART\_TIMEOUT, # non-blocking

))

# Command map for UART single-char commands

self.\_cmd\_map = {

ord('R'): self.led\_R.toggle,

ord('G'): self.led\_G.toggle,

ord('B'): self.led\_B.toggle,

ord('A'): self.buzzer.toggle,

}

def \_handle\_uart(self):

ch = self.uart.read\_char()

if ch is None:

return

action = self.\_cmd\_map.get(ch, None)

if action:

action()

# else: ignore unknown chars

def \_handle\_buttons(self, ts\_ms: int):

if self.btn1.update(ts\_ms):

self.uart.send\_line("switch 1 is pressed")

if self.btn2.update(ts\_ms):

self.uart.send\_line("switch 2 is pressed")

if self.btn3.update(ts\_ms):

self.uart.send\_line("switch 3 is pressed")

def run(self):

# Reuse a single loop delay to stay responsive yet lightweight

while True:

ts = now\_ms()

self.\_handle\_uart()

self.\_handle\_buttons(ts)

time.sleep\_ms(LOOP\_SLEEP\_MS)

# Entry point

if \_\_name\_\_ == "\_\_main\_\_":

App().run()

**Question 22 (3 pnts)**

Code of main.py to test the tilt protection.

*Remark: All python files of this program must also be submitted in a folder named Q22.*

from machine import Pin, PWM, UART, SoftI2C

import time

from HighLevelController import HighLevelController, reg

from MPU6050\_studentversion import MPU6050

def set\_leds(green\_on: bool, red\_on: bool):

led\_green.value(1 if green\_on else 0)

led\_red.value(1 if red\_on else 0)

def buzzer\_on(on: bool):

# MicroPython ESP32 PWM: duty 0..1023 (classic API). 0 = off.

buzzer.duty(512 if on else 0)

# ---- Devices ----

mpu = MPU6050(i2c) # wakes MPU6050 in its constructor

hlc = HighLevelController(uart2, None) # TCP not used in Q22

# ---- Core function required by Q22 ----

def TiltAngle():

"""

- Read accelerations and tilt from MPU6050

- Store in reg[3..6] with correct units (Appendix E)

acc in mm/s^2 (int), tilt in degrees (int)

- Compare against reg[20] (warn) and reg[21] (error)

-> LEDs & buzzer logic per spec

"""

tilt\_deg = mpu.tiltAngle()

ax, ay, az = mpu.acc # in m/s^2 from studentversion driver

# Write registers (Appendix E):

reg[3] = int(round(ax \* 1000)) # mm/s^2

reg[4] = int(round(ay \* 1000)) # mm/s^2

reg[5] = int(round(az \* 1000)) # mm/s^2

reg[6] = int(round(tilt\_deg)) # degrees

warn = reg[20] # Tilt angle warning level [°]; 0 disables warning

err = reg[21] # Tilt angle error level [°]; 0 disables error

# --- Warning LEDs ---

if warn == 0:

# Warning system disabled -> both LEDs off

set\_leds(green\_on=False, red\_on=False)

else:

if tilt\_deg < warn:

# OK region -> green on, red off

set\_leds(green\_on=True, red\_on=False)

else:

# Warning exceeded -> red on, green off

set\_leds(green\_on=False, red\_on=True)

# --- Error (buzzer) ---

if err == 0:

buzzer\_on(False)

else:

buzzer\_on(tilt\_deg >= err)

# ---- 100 ms scheduler loop (exact period) ----

PERIOD\_MS = 100

next\_tick = time.ticks\_add(time.ticks\_ms(), PERIOD\_MS)

while True:

# Run the tilt protection function

TiltAngle()

# Service UART protocol (read/write to reg[] as needed)

hlc.readFromUart()

# Keep exactly 100 ms period

now = time.ticks\_ms()

# if we're late, jump to next slot to avoid drift

if time.ticks\_diff(next\_tick, now) <= 0:

next\_tick = time.ticks\_add(now, PERIOD\_MS)

else:

time.sleep\_ms(time.ticks\_diff(next\_tick, now))

next\_tick = time.ticks\_add(next\_tick, PERIOD\_MS)

**Question 23 (4 pnts)**

Code of main.py to test the TCP communication.

*Remark: All python files of this program must also be submitted in a folder named Q23.*

from machine import Pin

from time import sleep\_ms

from TCPHandler import TCPHandler

# WiFi network credentials

SSID = 'Kurwa' # <-- your WiFi SSID

PASSWORD = '12345679' # <-- your WiFi password

PORT = 5000

# LED on Pin 2 turns on when a client connects

tcp = TCPHandler(SSID, PASSWORD, PORT, led=Pin(2))

print("TCP square server started. Waiting for client connection...")

while True:

state = tcp.getState()

if state == 0: # not connected to WiFi

sleep\_ms(100)

continue

elif state == 1: # WiFi connected (no client)

sleep\_ms(100)

continue

elif state == 2: # client connected

data = tcp.receive() # <-- correct method name

if not data:

sleep\_ms(50)

continue

# Convert bytes -> str

try:

text = data.decode().strip()

except Exception:

text = ""

# Try to parse number and respond

try:

value = float(text)

square = value \*\* 2

tcp.transmit(str(square)) # <-- correct method name

except ValueError:

tcp.transmit("Error: invalid number")

sleep\_ms(50)

**Question 24 (14 pnts)**

Contents of main.py. *Remark: all script files of this program must also be submitted in a folder named Q24.*

#Q24

from machine import Pin, PWM, ADC, SoftI2C, UART

from time import ticks\_ms, ticks\_add, ticks\_diff, sleep\_ms

from HighLevelController1 import HighLevelController, reg

from TCPHandler import TCPHandler

from MPU6050\_studentversion import MPU6050

import Encoder

import time

SSID = "Kurwa"

PASSWORD = "12345679"

PORT = 5000

tcp = TCPHandler(SSID, PASSWORD, PORT, led=Pin(2))

hlc = HighLevelController(uart2, tcp)

# Wait for TCP server to be ready (so port 5000 opens)

print("Waiting for TCP server to start...")

tcp\_state = 0

for \_ in range(150): # ~15 s max

tcp\_state = tcp.getState() # 0: Wi-Fi connecting, 1: listening, 2: client connected

if tcp\_state >= 1:

break

time.sleep\_ms(100)

print("TCP state at start:", tcp\_state)

# ----------- Sensors -----------

mpu = MPU6050(i2c, addr=0x68, up\_axis='x', up\_sign=+1)

Encoder.InitEncoder(ENC\_A, ENC\_B)

# ----------- Motor / model constants (regs 13..19) -----------

# Encoder is 16 CPR (cycles/rev) on the MOTOR shaft; gearbox 48:1.

reg[13] = 16 # CPR at motor shaft (match Q18)

reg[14] = 21 # gear ratio ×1000 => (1/48)\*1000 ≈ 20.833 -> 21

reg[15] = 250 # rated shaft speed [rpm] (placeholder; update if you have datasheet value)

# Torque model (from your Q18b): km ≈ 10.963 mNm/A, choose Irated=200 mA -> Trated ≈ 2.193 mNm

reg[16] = 2193 # rated torque [mNm]

reg[17] = 200 # rated current [mA]

# From Q18 / Q18b

reg[18] = 160 # no-load current @ ~½ speed [mA]

reg[19] = 1460 # armature resistance [mΩ] (1.460445 Ω)

# Tilt thresholds (modifiable over HLC)

reg[20] = 15 # warn [°]

reg[21] = 30 # error [°]

# VS path: ~5:1 divider to 0..3.0 V range

VS\_MV\_PER\_COUNT = (3000 \* 5) / 4095.0 # ≈ 3.663 mV/count at battery

# Current sensor linear fit from ±I tables:

# I[mA] = I0\_MA + C2\_MA\_PER\_MV \* Vcs[mV]

I0\_MA = -3896.0

C2\_MA\_PER\_MV = 2.33

def read\_battery\_mV():

return int(VS\_ADC.read() \* VS\_MV\_PER\_COUNT)

def read\_current\_mA():

vcs\_mV = (CS\_ADC.read() \* 3000.0) / 4095.0

return int(round(I0\_MA + C2\_MA\_PER\_MV \* vcs\_mV))

def apply\_pwm\_from\_reg(pwm\_obj, cmd\_0\_1023):

"""Accept 0..1023 in reg and apply to PWM (works on duty\_u16 or duty)."""

if cmd\_0\_1023 < 0: cmd\_0\_1023 = 0

if cmd\_0\_1023 > 1023: cmd\_0\_1023 = 1023

if hasattr(pwm\_obj, "duty\_u16"):

pwm\_obj.duty\_u16(int((cmd\_0\_1023 / 1023.0) \* 65535))

else:

pwm\_obj.duty(cmd\_0\_1023)

def tilt\_protection():

"""LEDs + buzzer + EN control based on tilt thresholds; publish acc/tilt regs."""

tilt = mpu.tiltAngle()

ax, ay, az = mpu.acc # m/s^2

# publish to registers

reg[3] = int(round(ax \* 1000)) # mm/s^2

reg[4] = int(round(ay \* 1000))

reg[5] = int(round(az \* 1000))

reg[6] = int(round(tilt)) # deg

warn = reg[20]

err = reg[21]

# LEDs

if warn == 0:

LED\_G.value(0); LED\_R.value(0)

else:

if tilt < warn:

LED\_G.value(1); LED\_R.value(0)

else:

LED\_G.value(0); LED\_R.value(1)

# Error: buzzer + disable H-bridge

if err and tilt >= err:

buzzer.duty\_u16(20000) # on

EN.value(0) # disable power stage

else:

buzzer.duty\_u16(0)

EN.value(1) # allowed

def compute\_speed\_torque(current\_mA, freq\_Hz):

"""

freq\_Hz: signed cycles/sec from a 16-CPR encoder on the MOTOR shaft (like Q18)

motor\_rpm = (freq \* 60) / 16

shaft\_rpm = motor\_rpm \* (reg[14]/1000) ~ divide by 48

"""

motor\_rpm = (freq\_Hz \* 60.0) / 16.0

shaft\_rpm = motor\_rpm \* (reg[14] / 1000.0)

# no-load current sign follows direction

I0 = reg[18] if shaft\_rpm >= 0 else -reg[18]

kT = (reg[16] / float(reg[17])) if reg[17] else 0.0 # mNm/mA

Tshaft\_mNm = kT \* (current\_mA - I0)

return int(round(shaft\_rpm)), int(round(Tshaft\_mNm))

PERIOD\_MS = 100

reg[22] = 0 # pwm1 command 0..1023

reg[23] = 0 # pwm2 command 0..1023

print("Q24 starting...")

next\_t = ticks\_add(ticks\_ms(), PERIOD\_MS)

while True:

# Keep TCP state machine alive (listening / accept clients)

tcp.getState()

# sensors → regs[0..2]

reg[0] = read\_battery\_mV() # mV

reg[1] = read\_current\_mA() # mA

freq = Encoder.GetFrequency() # Hz (signed; same as Q18)

reg[2] = int(freq)

# tilt protection (also sets EN/buzzer), publish acc/tilt regs

tilt\_protection()

# derive speed & torque → regs[7..8]

rpm\_shaft, Tshaft = compute\_speed\_torque(reg[1], freq)

reg[7] = rpm\_shaft

reg[8] = Tshaft

# serve protocols

hlc.readFromTCP() # TCP client (KiTTY)

hlc.readFromUart() # UART2 client, if used

# apply PWM commands from reg[22]/[23] (0..1023)

apply\_pwm\_from\_reg(PWM1, reg[22])

apply\_pwm\_from\_reg(PWM2, reg[23])

# exact 100 ms cadence

now = ticks\_ms()

dt = ticks\_diff(next\_t, now)

if dt > 0:

sleep\_ms(dt)

next\_t = ticks\_add(next\_t, PERIOD\_MS)

else:

next\_t = ticks\_add(now, PERIOD\_MS)

**Question 25 (10 pnts)**

Fill int the tables below

|  |  |  |  |
| --- | --- | --- | --- |
| Speed setpoint [rpm] | Speed shaft [rpm] | Encoder  frequency [Hz] | Difference speed setpoint and speed shaft [%] |
| 25 | 25 | 293 | 0% |
| 50 | 51 | 565 | 2% |
| 100 | 99 | 1155 | -1% |
| 150 | 152 | 1720 | 1.3% |
| 200 | 202 | 2301 | 1% |
| 250 | 252 | 2900 | 0.8% |
| -25 | -26 | 290 | 4% |
| -50 | -50 | 570 | 0% |
| -100 | -99 | 1170 | -1% |
| -150 | -152 | 1730 | 1.3% |
| -200 | -198 | 2289 | -1% |
| -250 | -252 | 2889 | 0.8% |

|  |  |  |  |
| --- | --- | --- | --- |
| Torque limit [mNm] | Torque at shaft [mNm] | Motor current [mA] | Difference torque limit and measured torque [%] |
| Speed setpoint= 50 [rpm] | | | |
| 50 | 50 | 552 mA | 0% |
| 100 | 100 | 1009 mA | 0% |
| 150 | 150 | 1466 mA | 0% |
| 200 | 201 | 1931 mA | 0.5% |
| Speed setpoint= -50 [rpm] | | | |
| 50 | - 47 | -326 mA | -6% |
| 100 | - 97 | -789 mA | -3% |
| 150 | - 140 | -1375 mA | -6.7% |
| 200 | - 196 | -1683 mA | -2% |

The measurements are done at 0rpm (STALL zone)

Contents of main.py. *Remark: all script files of this program must also be submitted in a folder named Q25.*

from boot import \* # pins, PWM1/PWM2, EN, LEDs, ADCs, uart2, i2c, etc.

from machine import Pin

from time import ticks\_ms, ticks\_add, ticks\_diff, sleep\_ms

from HighLevelController1 import HighLevelController, reg

from TCPHandler import TCPHandler

from MPU6050\_studentversion import MPU6050

import Encoder

import time

# ---------------- Network / protocol ----------------

SSID = "Kurwa"

PASSWORD = "12345679"

PORT = 5000

tcp = TCPHandler(SSID, PASSWORD, PORT, led=Pin(2))

hlc = HighLevelController(uart2, tcp)

print("Waiting for TCP server to start.")

for \_ in range(150): # about 15 seconds

if tcp.getState() >= 1: # 0: Wi-Fi, 1: listening, 2: client connected

break

time.sleep\_ms(100)

# ---------------- Sensors ----------------

mpu = MPU6050(i2c, addr=0x68, up\_axis='x', up\_sign=+1)

Encoder.InitEncoder(ENC\_A, ENC\_B) # returns signed hertz on GetFrequency()

# ---------------- Motor / model constants ----------------

reg[13] = 16 # cycles per motor revolution at motor

reg[14] = 23 # gear ratio times one thousand (= 1/43.7)

reg[15] = 250 # rated shaft speed [rpm]

reg[16] = 318 # rated torque [mN·m]

reg[17] = 2900 # rated current [mA]

reg[18] = 100 # no-load current near half speed [mA]

reg[19] = 1460 # armature resistance [mΩ] (1.460 Ω)

# ---------------- Tilt thresholds ----------------

reg[20], reg[21] = 15, 30

# ---------------- ADC calibration ----------------

VS\_MV\_PER\_COUNT = (3300 \* 5) / 4095.0 # battery divider about five to one

# Current sensor calibration (auto-zero + selectable polarity)

C2\_MA\_PER\_MV = 2.33 # slope magnitude

CURR\_SIGN = +1 # set each loop from reg[26]

CURR\_VREF\_MV = None # learned zero (idle) voltage

current\_filt = 0.0 # filtered current (mA) for torque

def calibrate\_current\_zero(samples=64):

global CURR\_VREF\_MV

acc = 0

for \_ in range(samples):

acc += CS\_ADC.read()

time.sleep\_ms(5)

CURR\_VREF\_MV = (acc / samples) \* (3300.0 / 4095.0)

def read\_battery\_mV():

return int(VS\_ADC.read() \* VS\_MV\_PER\_COUNT)

def read\_current\_mA():

"""Read raw sensor, convert to milliampere, then apply a simple low-pass filter."""

global CURR\_VREF\_MV, current\_filt

if CURR\_VREF\_MV is None:

calibrate\_current\_zero()

vcs\_mV = (CS\_ADC.read() \* 3300.0) / 4095.0

i\_now = CURR\_SIGN \* C2\_MA\_PER\_MV \* (vcs\_mV - CURR\_VREF\_MV)

# Low-pass: 90 percent previous + 10 percent new

current\_filt = 0.9 \* current\_filt + 0.1 \* i\_now

return int(round(current\_filt))

# ---------------- Helpers ----------------

def apply\_pwm\_from\_reg(pwm\_obj, cmd\_0\_1023):

if cmd\_0\_1023 < 0: cmd\_0\_1023 = 0

if cmd\_0\_1023 > 1023: cmd\_0\_1023 = 1023

if hasattr(pwm\_obj, "duty\_u16"):

pwm\_obj.duty\_u16(int((cmd\_0\_1023 / 1023.0) \* 65535))

else:

pwm\_obj.duty(cmd\_0\_1023)

def set\_duty\_fraction(duty):

duty = max(-0.98, min(0.98, duty))

if duty >= 0:

reg[23] = 0

reg[22] = int(round(duty \* 1023))

else:

reg[22] = 0

reg[23] = int(round((-duty) \* 1023))

def get\_duty\_fraction():

if reg[22] > 0 and reg[23] == 0:

return min(0.98, reg[22] / 1023.0)

if reg[23] > 0 and reg[22] == 0:

return -min(0.98, reg[23] / 1023.0)

return 0.0

# --- Tilt protection with small hysteresis; only DISABLES the bridge

tilt\_in\_fault = False # latched fault with hysteresis

def tilt\_protection():

"""Update tilt registers, light indicators, and buzzer.

Return True when fault (bridge must be disabled), False otherwise.

This function never enables the bridge; it may only force-disable it."""

global tilt\_in\_fault

tilt = mpu.tiltAngle()

ax, ay, az = mpu.acc

reg[3] = int(round(ax \* 1000))

reg[4] = int(round(ay \* 1000))

reg[5] = int(round(az \* 1000))

reg[6] = int(round(tilt))

warn, err = reg[20], reg[21]

# Warning indicators

if warn == 0:

LED\_G.value(0); LED\_R.value(0)

else:

if tilt < warn: LED\_G.value(1); LED\_R.value(0)

else: LED\_G.value(0); LED\_R.value(1)

# Fault latch with two degree hysteresis

if err:

if tilt\_in\_fault:

if tilt < (err - 2):

tilt\_in\_fault = False

else:

if tilt >= err:

tilt\_in\_fault = True

else:

tilt\_in\_fault = False

# Buzzer and enforced disable on fault

if tilt\_in\_fault:

buzzer.duty\_u16(20000)

EN.value(0)

else:

buzzer.duty\_u16(0)

return tilt\_in\_fault

def cycles\_to\_rpm\_shaft(cycles\_hz):

cpr = reg[13] if reg[13] else 16

motor\_rpm = (cycles\_hz \* 60.0) / float(cpr)

return int(round(motor\_rpm \* (reg[14] / 1000.0)))

def compute\_torque\_mNm(current\_mA, rpm\_shaft):

"""

Return SIGNED shaft torque.

Convention used here:

- Positive torque when it assists the current direction of rotation.

- Negative torque when it opposes the rotation (braking).

Implementation:

τ = kT \* (I - sgn(ω)\*I0), then we align sign with rotation: τ\_signed = sgn(ω) \* τ

"""

# motor torque constant in mN·m per mA

kT = (reg[16] / float(reg[17])) if reg[17] else 0.0

# sign of rotation (+1 for ≥0 rpm, -1 for <0 rpm)

s\_omega = 1 if rpm\_shaft >= 0 else -1

# direction-aware no-load current

I0 = s\_omega \* reg[18]

# raw torque (can be positive or negative)

tau = kT \* (current\_mA - I0)

# torque signed with rotation direction

tau\_signed = s\_omega \* tau

# small symmetric deadband to kill tiny noise around zero

if abs(tau\_signed) < 5:

return 0

return int(round(tau\_signed))

# ---------------- Q25 controller ----------------

MODE\_OFF = 0

MODE\_TEST = 1

MODE\_CTRL = 2

# defaults

reg[9] = MODE\_OFF # start in off mode

reg[10] = 50 # speed setpoint [rpm]

reg[11] = reg[16] # torque limit [mN·m] (start at rated torque)

reg[26] = 1 # current polarity control: >0 => positive, <0 => negative

TS = 0.1 # loop period [s]

KI = 0.016 # integral gain tuned for the required ramp

RPM\_ALPHA = 0.2 # speed low-pass weight

rpm\_filt = 0.0

duty\_int = 0.0

def i\_controller\_step(rpm\_set, rpm\_meas, vin\_mV, i\_meas\_mA, enabled=True):

"""

Integral-only controller with current-based duty limiting.

Direction guard: duty sign follows setpoint sign; never crosses zero.

"""

global duty\_int

# When disabled, bleed integrator and keep sign guard

if not enabled:

duty\_int \*= 0.9

# guard against tiny sign creep

if rpm\_set >= 0 and duty\_int < 0: duty\_int = 0.0

if rpm\_set < 0 and duty\_int > 0: duty\_int = 0.0

return duty\_int

# 1) Integral step

error = rpm\_set - rpm\_meas

delta\_dc = KI \* error \* TS

# 2) Current/torque limiter (caps the permitted delta toward limit)

kT = (reg[16] / float(reg[17])) if reg[17] else 1e-6

I0 = reg[18] if rpm\_set >= 0 else -reg[18]

Imax = (reg[11] / kT) + I0 # [mA]

Ra = reg[19] / 1000.0 # [Ohm]

Uin = max(1.0, vin\_mV / 1000.0) # [V]

if rpm\_set >= 0:

# When over limit (i\_meas > Imax), limiter allows negative delta to back off,

# but the direction guard (below) will stop at zero.

delta\_dc = min(delta\_dc, (Ra \* (Imax - i\_meas\_mA)) / Uin)

else:

delta\_dc = max(delta\_dc, -(Ra \* (Imax + i\_meas\_mA)) / Uin)

# 3) Accumulate

duty\_int += delta\_dc

# 4) Direction guard + anti-windup at bound

if rpm\_set >= 0:

if duty\_int < 0.0:

duty\_int = 0.0 # do not cross into negative

if duty\_int > 0.98:

duty\_int = 0.98 # top clamp

else:

if duty\_int > 0.0:

duty\_int = 0.0 # do not cross into positive

if duty\_int < -0.98:

duty\_int = -0.98 # bottom clamp

return duty\_int

# ---------------- Telemetry (UART2) ----------------

reg[27] = 1 # 1 = telemetry on, 0 = off

TELEMETRY\_EVERY\_N = 5 # one line every 5 loops (~2 Hz at 100 ms)

telemetry\_counter = 0

def send\_telemetry(timestamp\_ms):

"""CSV to uart2: tel,<seconds>,<rpm>,<mN·m>,<mA>"""

try:

s = "tel,{:.3f},{},{},{}\n".format(

timestamp\_ms / 1000.0,

int(reg[7]), # speed (rpm)

int(reg[8]), # torque (mN·m)

int(reg[1]) # current (mA) <-- added

)

uart2.write(s)

except Exception:

pass

# -------------- Encoder sign stabilization --------------

last\_sign = +1

sign\_streak = 0

RPM\_SIGN\_MIN = 20 # do not accept sign changes below this speed (rpm)

SIGN\_HOLD\_SAMPLES = 3 # need this many consecutive samples to flip sign

# ---------------- Main loop ----------------

PERIOD\_MS = 100

reg[22] = 0

reg[23] = 0

# Ensure auto-zero happens at rest

calibrate\_current\_zero()

print("Q25 starting.")

next\_t = ticks\_add(ticks\_ms(), PERIOD\_MS)

while True:

tcp.getState() # keep server alive

# Sensors -> registers

reg[0] = read\_battery\_mV()

# current polarity from reg[26]

CURR\_SIGN = 1 if reg[26] >= 0 else -1

reg[1] = read\_current\_mA()

# Encoder frequency (signed hertz)

freq\_hz = Encoder.GetFrequency()

reg[2] = int(freq\_hz) # signed frequency

# Tilt protection (may force-disable)

tilt\_fault = tilt\_protection()

# ---------- Derived signals with sign stabilization ----------

rpm\_mag = cycles\_to\_rpm\_shaft(abs(freq\_hz)) # non-negative magnitude

cand\_sign = +1 if freq\_hz >= 0 else -1

# Only allow sign change if speed is clearly nonzero and sustained

if rpm\_mag >= RPM\_SIGN\_MIN and cand\_sign != last\_sign:

sign\_streak += 1

if sign\_streak >= SIGN\_HOLD\_SAMPLES:

last\_sign = cand\_sign

sign\_streak = 0

else:

sign\_streak = 0

rpm\_signed = rpm\_mag if last\_sign > 0 else -rpm\_mag

reg[7] = int(rpm\_signed) # signed shaft rpm

reg[8] = compute\_torque\_mNm(reg[1], reg[7]) # uses filtered current

# ------------------------------------------------------------

# Telemetry at limited rate

if reg[27]:

telemetry\_counter = (telemetry\_counter + 1) % TELEMETRY\_EVERY\_N

if telemetry\_counter == 0:

send\_telemetry(ticks\_ms())

# Filtered rpm for controller stability (on signed rpm)

rpm\_filt += RPM\_ALPHA \* (reg[7] - rpm\_filt)

# Commands from host

hlc.readFromTCP()

hlc.readFromUart()

# -------- Modes --------

m = reg[9]

if m == MODE\_OFF:

duty\_int = 0.0

reg[22] = 0; reg[23] = 0

EN.value(0) # keep disabled in OFF

elif m == MODE\_TEST:

# manual pulse width, ensure exclusivity

if reg[22] > 0: reg[23] = 0

if reg[23] > 0: reg[22] = 0

duty\_int = get\_duty\_fraction() # track for smooth switching

EN.value(0 if tilt\_fault else 1)

elif m == MODE\_CTRL:

duty = i\_controller\_step(

rpm\_set = reg[10],

rpm\_meas = rpm\_filt,

vin\_mV = reg[0],

i\_meas\_mA = reg[1],

enabled = (not tilt\_fault)

)

set\_duty\_fraction(duty)

EN.value(0 if tilt\_fault else 1)

else:

# unknown mode -> safe

duty\_int = 0.0

reg[22] = 0; reg[23] = 0

EN.value(0)

# Apply to hardware

apply\_pwm\_from\_reg(PWM1, reg[22])

apply\_pwm\_from\_reg(PWM2, reg[23])

# 100 ms cadence

now = ticks\_ms()

dt = ticks\_diff(next\_t, now)

if dt > 0:

sleep\_ms(dt); next\_t = ticks\_add(next\_t, PERIOD\_MS)

else:

next\_t = ticks\_add(now, PERIOD\_MS)

**Question 26 (10 pnts)**

Contents of main.py. *Remark: all script files of this program must also be submitted in a folder named Q26.*

from boot import \* # pins, PWM1/PWM2, EN, LEDs, ADCs, uart2, i2c, SW1/SW2/SW3, etc.

from machine import Pin

from time import ticks\_ms, ticks\_add, ticks\_diff, sleep\_ms

from HighLevelController1 import HighLevelController, reg

from TCPHandler import TCPHandler

from MPU6050\_studentversion import MPU6050

import Encoder

import time

# ---------------- Network / protocol ----------------

SSID = "Kurwa"

PASSWORD = "12345679"

PORT = 5000

tcp = TCPHandler(SSID, PASSWORD, PORT, led=Pin(2))

hlc = HighLevelController(uart2, tcp)

print("Waiting for TCP server to start.")

for \_ in range(150): # about 15 seconds

if tcp.getState() >= 1: # 0: Wi-Fi, 1: listening, 2: client connected

break

time.sleep\_ms(100)

# ---------------- Sensors ----------------

mpu = MPU6050(i2c, addr=0x68, up\_axis='x', up\_sign=+1)

Encoder.InitEncoder(ENC\_A, ENC\_B) # returns signed hertz on GetFrequency()

# ---------------- Motor / model constants ----------------

reg[13] = 16 # cycles per motor revolution at motor

reg[14] = 23 # gear ratio ×1000 (= 1/43.7)

reg[15] = 250 # rated shaft speed [rpm]

reg[16] = 318 # rated torque [mN·m]

reg[17] = 2900 # rated current [mA]

reg[18] = 100 # no-load current [mA]

reg[19] = 1460 # armature resistance [mΩ] (1.460 Ω)

# ---------------- Tilt thresholds ----------------

reg[20], reg[21] = 30, 45

# ---------------- ADC calibration ----------------

VS\_MV\_PER\_COUNT = (3300 \* 5) / 4095.0 # battery divider about five to one

# Current sensor calibration

C2\_MA\_PER\_MV = 2.33

CURR\_SIGN = +1

CURR\_VREF\_MV = None

current\_filt = 0.0

def calibrate\_current\_zero(samples=64):

global CURR\_VREF\_MV

acc = 0

for \_ in range(samples):

acc += CS\_ADC.read()

time.sleep\_ms(5)

CURR\_VREF\_MV = (acc / samples) \* (3300.0 / 4095.0)

def read\_battery\_mV():

return int(VS\_ADC.read() \* VS\_MV\_PER\_COUNT)

def read\_current\_mA():

"""Read current sensor and filter."""

global CURR\_VREF\_MV, current\_filt

if CURR\_VREF\_MV is None:

calibrate\_current\_zero()

vcs\_mV = (CS\_ADC.read() \* 3300.0) / 4095.0

i\_now = CURR\_SIGN \* C2\_MA\_PER\_MV \* (vcs\_mV - CURR\_VREF\_MV)

current\_filt = 0.9 \* current\_filt + 0.1 \* i\_now

return int(round(current\_filt))

# ---------------- Helpers ----------------

def apply\_pwm\_from\_reg(pwm\_obj, cmd\_0\_1023):

cmd\_0\_1023 = max(0, min(1023, cmd\_0\_1023))

if hasattr(pwm\_obj, "duty\_u16"):

pwm\_obj.duty\_u16(int((cmd\_0\_1023 / 1023.0) \* 65535))

else:

pwm\_obj.duty(cmd\_0\_1023)

def set\_duty\_fraction(duty):

duty = max(-0.98, min(0.98, duty))

if duty >= 0:

reg[23] = 0

reg[22] = int(round(duty \* 1023))

else:

reg[22] = 0

reg[23] = int(round((-duty) \* 1023))

def get\_duty\_fraction():

if reg[22] > 0 and reg[23] == 0:

return min(0.98, reg[22] / 1023.0)

if reg[23] > 0 and reg[22] == 0:

return -min(0.98, reg[23] / 1023.0)

return 0.0

# --- Tilt protection ---

tilt\_in\_fault = False

def tilt\_protection():

"""Read tilt, drive LEDs, buzzer, and disable on fault."""

global tilt\_in\_fault

tilt = mpu.tiltAngle()

ax, ay, az = mpu.acc

reg[3] = int(round(ax \* 1000))

reg[4] = int(round(ay \* 1000))

reg[5] = int(round(az \* 1000))

reg[6] = int(round(tilt))

warn, err = reg[20], reg[21]

if warn == 0:

LED\_G.value(0); LED\_R.value(0)

else:

if tilt < warn: LED\_G.value(1); LED\_R.value(0)

else: LED\_G.value(0); LED\_R.value(1)

# Fault latch with hysteresis

if err:

if tilt\_in\_fault:

if tilt < (err - 2):

tilt\_in\_fault = False

else:

if tilt >= err:

tilt\_in\_fault = True

else:

tilt\_in\_fault = False

if tilt\_in\_fault:

buzzer.duty\_u16(20000)

EN.value(0)

else:

buzzer.duty\_u16(0)

return tilt\_in\_fault

def cycles\_to\_rpm\_shaft(cycles\_hz):

cpr = reg[13] if reg[13] else 16

motor\_rpm = (cycles\_hz \* 60.0) / float(cpr)

return int(round(motor\_rpm \* (reg[14] / 1000.0)))

def compute\_torque\_mNm(current\_mA, rpm\_shaft):

"""Compute signed shaft torque."""

kT = (reg[16] / float(reg[17])) if reg[17] else 0.0

s\_omega = 1 if rpm\_shaft >= 0 else -1

I0 = s\_omega \* reg[18]

tau = kT \* (current\_mA - I0)

tau\_signed = s\_omega \* tau

if abs(tau\_signed) < 5:

return 0

return int(round(tau\_signed))

# ---------------- Q25 controller ----------------

MODE\_OFF = 0

MODE\_TEST = 1

MODE\_CTRL = 2

reg[9] = MODE\_OFF

reg[10] = 50

reg[11] = reg[16]

reg[26] = 1

TS = 0.1

KI = 0.016

RPM\_ALPHA = 0.2

rpm\_filt = 0.0

duty\_int = 0.0

def i\_controller\_step(rpm\_set, rpm\_meas, vin\_mV, i\_meas\_mA, enabled=True):

global duty\_int

if not enabled:

duty\_int \*= 0.9

if rpm\_set >= 0 and duty\_int < 0: duty\_int = 0.0

if rpm\_set < 0 and duty\_int > 0: duty\_int = 0.0

return duty\_int

error = rpm\_set - rpm\_meas

delta\_dc = KI \* error \* TS

kT = (reg[16] / float(reg[17])) if reg[17] else 1e-6

I0 = reg[18] if rpm\_set >= 0 else -reg[18]

Imax = (reg[11] / kT) + I0

Ra = reg[19] / 1000.0

Uin = max(1.0, vin\_mV / 1000.0)

if rpm\_set >= 0:

delta\_dc = min(delta\_dc, (Ra \* (Imax - i\_meas\_mA)) / Uin)

else:

delta\_dc = max(delta\_dc, -(Ra \* (Imax + i\_meas\_mA)) / Uin)

duty\_int += delta\_dc

if rpm\_set >= 0:

duty\_int = min(0.98, max(0.0, duty\_int))

else:

duty\_int = max(-0.98, min(0.0, duty\_int))

return duty\_int

# -------------- Encoder sign stabilization --------------

last\_sign = +1

sign\_streak = 0

RPM\_SIGN\_MIN = 20

SIGN\_HOLD\_SAMPLES = 3

# -------------- Buttons (use SW1/SW2/SW3 from boot.py) --------------

# Active-low convention: 1 = released, 0 = pressed

prev\_sw1 = SW1.value()

prev\_sw2 = SW2.value()

prev\_sw3 = SW3.value()

DEBOUNCE\_MS = 30

last\_sw1\_time = 0

last\_sw2\_time = 0

last\_sw3\_time = 0

def \_pressed(prev, now):

return (prev == 1) and (now == 0) # falling edge on active-low

last\_active\_mode = MODE\_TEST # remembered for SW1 toggle

# ---------------- Main loop ----------------

PERIOD\_MS = 100

reg[22] = 0

reg[23] = 0

calibrate\_current\_zero()

print("Q25 starting.")

next\_t = ticks\_add(ticks\_ms(), PERIOD\_MS)

while True:

tcp.getState()

# Sensors

reg[0] = read\_battery\_mV()

CURR\_SIGN = 1 if reg[26] >= 0 else -1

reg[1] = read\_current\_mA()

freq\_hz = Encoder.GetFrequency()

reg[2] = int(freq\_hz)

tilt\_fault = tilt\_protection()

rpm\_mag = cycles\_to\_rpm\_shaft(abs(freq\_hz))

cand\_sign = +1 if freq\_hz >= 0 else -1

if rpm\_mag >= RPM\_SIGN\_MIN and cand\_sign != last\_sign:

sign\_streak += 1

if sign\_streak >= SIGN\_HOLD\_SAMPLES:

last\_sign = cand\_sign

sign\_streak = 0

else:

sign\_streak = 0

rpm\_signed = rpm\_mag if last\_sign > 0 else -rpm\_mag

reg[7] = int(rpm\_signed)

reg[8] = compute\_torque\_mNm(reg[1], reg[7])

rpm\_filt += RPM\_ALPHA \* (reg[7] - rpm\_filt)

# ---------------- Button handling (debounced) ----------------

now\_ms = ticks\_ms()

sw1 = SW1.value()

sw2 = SW2.value()

sw3 = SW3.value()

# SW1: toggle OFF <-> last\_active\_mode (if no tilt fault)

if \_pressed(prev\_sw1, sw1) and ticks\_diff(now\_ms, last\_sw1\_time) > DEBOUNCE\_MS:

last\_sw1\_time = now\_ms

if reg[9] == MODE\_OFF and not tilt\_fault:

reg[9] = last\_active\_mode

else:

if reg[9] != MODE\_OFF:

last\_active\_mode = reg[9]

reg[9] = MODE\_OFF

reg[22] = 0; reg[23] = 0

EN.value(0)

# SW2: speed up / bias forward

if \_pressed(prev\_sw2, sw2) and ticks\_diff(now\_ms, last\_sw2\_time) > DEBOUNCE\_MS:

last\_sw2\_time = now\_ms

if reg[9] == MODE\_TEST:

if reg[23] == 0:

reg[22] = min(970, reg[22] + 50)

else:

reg[23] = max(0, reg[23] - 50)

elif reg[9] == MODE\_CTRL:

step = max(1, int(round(0.05 \* reg[15]))) # +5% of rated

max\_sp = int(round(1.20 \* reg[15])) # +120% cap

reg[10] = min(max\_sp, reg[10] + step)

# SW3: speed down / bias reverse

if \_pressed(prev\_sw3, sw3) and ticks\_diff(now\_ms, last\_sw3\_time) > DEBOUNCE\_MS:

last\_sw3\_time = now\_ms

if reg[9] == MODE\_TEST:

if reg[22] == 0:

reg[23] = min(970, reg[23] + 50)

else:

reg[22] = max(0, reg[22] - 50)

elif reg[9] == MODE\_CTRL:

step = max(1, int(round(0.05 \* reg[15]))) # -5% of rated

min\_sp = -int(round(1.20 \* reg[15])) # -120% cap

reg[10] = max(min\_sp, reg[10] - step)

prev\_sw1, prev\_sw2, prev\_sw3 = sw1, sw2, sw3

# -------------------------------------------------------------

# Commands from host

hlc.readFromTCP()

hlc.readFromUart()

# Modes

m = reg[9]

if m == MODE\_OFF:

duty\_int = 0.0

reg[22] = 0

reg[23] = 0

EN.value(0)

elif m == MODE\_TEST:

if reg[22] > 0: reg[23] = 0

if reg[23] > 0: reg[22] = 0

duty\_int = get\_duty\_fraction()

EN.value(0 if tilt\_fault else 1)

elif m == MODE\_CTRL:

duty = i\_controller\_step(

rpm\_set = reg[10],

rpm\_meas = rpm\_filt,

vin\_mV = reg[0],

i\_meas\_mA = reg[1],

enabled = (not tilt\_fault)

)

set\_duty\_fraction(duty)

EN.value(0 if tilt\_fault else 1)

else:

duty\_int = 0.0

reg[22] = 0

reg[23] = 0

EN.value(0)

apply\_pwm\_from\_reg(PWM1, reg[22])

apply\_pwm\_from\_reg(PWM2, reg[23])

now = ticks\_ms()

dt = ticks\_diff(next\_t, now)

if dt > 0:

sleep\_ms(dt)

next\_t = ticks\_add(next\_t, PERIOD\_MS)

else:

next\_t = ticks\_add(now, PERIOD\_MS)