Logbook For Microcontroller

Student name: Son Cao

Student number: 570135

Class: ETI2V.IA-2

Week 1: IO & Interrupts

Goal:

- This week, I learned how to control multiple GPIO pins and handle external interrupts on the Raspberry Pi Pico.

Step 2: Picotool

- **picotool load *.uf2 -f** means:
  - load = upload the UF2 file to the Pico over USB.
  - -f = "force" overwrite, even if the Pico is busy.
- **picotool info** can show device info and firmware status.

Step 3: Scanner Light (Mask Functions)

Using GPIO Mask Functions on the Raspberry Pi Pico:

- I learned about GPIO Mask Functions, which are useful for controlling multiple pins at once instead of one at a time with gpio_put(). This makes the code cleaner and faster.

1. gpio_init_mask(mask)

- Initializes all pins specified in the mask at once.

- mask is a bit pattern where each 1 represents a pin to initialize.

- Example: gpio_init_mask((1u << 15) | (1u << 14)); initializes pins 15 and 14.

2. gpio_set_dir_all_bits(mask, direction)

- Sets the direction (input/output) for all pins in the mask.

- direction = true for output, false for input.

3. gpio_put_all(value)

- Sets all pins at once according to the bit pattern in value.

- Example: gpio_put_all(0xFFFF) turns all pins high.

4. gpio_put_masked(mask, value)

- the state of only the pins in the mask. Other pins remain unchanged.

- Example: gpio_put_masked((1u << 15) | (1u << 14), (1u << 15)) turns pin 15 high and pin 14 low, leaving others unchanged.

5. gpio_set_dir_masked(mask, direction)

- Sets the direction of multiple pins at once, only affecting pins in the mask.

Step 4: External Interrupt Example

How the Program Works:

1. **Pin Initialization:**
   - The LED is configured as an output pin.
   - The button is configured as an input with an internal pull-up resistor (gpio_pull_up()), meaning the pin is normally HIGH and goes LOW when pressed.
2. **Interrupt Setup:**

```
gpio_set_irq_enabled_with_callback(BUTTON_STOP, GPIO_IRQ_EDGE_FALL, true, &gpio_callback);
```

   - enables an **interrupt** on the falling edge (button press).
   - When the button is pressed (signal goes from HIGH -> LOW), the interrupt triggers the callback function.
3. **Interrupt Service Routine (ISR):**
   - The function gpio_callback() runs automatically when the button is pressed.
   - gpio_xor_mask(1 << LED);
     - toggles the LED pin (XOR flips its current state => if on -> off, if off -> on).
4. **Main Loop:**
   - The while(true) loop continuously runs tight_loop_contents(), which keeps the system active but idle.

Question: What does the expression: (1<<LED) | (1<< BUTTON) actually mean ?

- This creates a **bitmask** with bits for both LED and BUTTON set to 1.
- Example: if LED=10 and BUTTON=2:
  - (1<<10) = 0b0000010000000000
  - (1<<2) = 0b0000000000000100
  - The | combines them => both pins are included.

Question: why is there one general gpio callback ?

- Because the RP2040 **uses one shared interrupt** handler for all GPIOs.
  The callback gets the GPIO number that triggered it, so you can handle multiple buttons in one function.

Question: why is it advisable to use the function: tight_loop_contents() ?

- tight_loop_contents() is a hint to the compiler and debugger that your program is idle here.
- It keeps the CPU busy in a power-efficient idle loop, allowing background tasks like USB I/O to continue and avoiding watchdog resets.
    - Allows the debugger (like Picotool) to show that the CPU is in the loop and not hung.
    - Gives the SDK a hook to optimize power use in the future.
    - Keep the program neat instead of just writing an empty while(1); loop
    - => It is advisable because it shows the program is idle, helps debugging, and can save power instead of just spinning in an empty loop.

Step 5: Scanner Light + Interrupts

- **BUTTON_START** starts scanning on rising edge (button release).
- **BUTTON_STOP** stops scanning on falling edge (button press).

```
//set interrupt callback
//The interrupt allows the program to respond immediately to button events, without constantly checking them in the main lo
gpio_set_irq_enabled_with_callback(BUTTON_START, GPIO_IRQ_EDGE_RISE, true, &gpio_callback); //release -> trigger interrupt
gpio_set_irq_enabled_with_callback(BUTTON_STOP, GPIO_IRQ_EDGE_FALL, true, &gpio_callback); //press -> trigger interrupt (cu
```

- **volatile bool scanning** ensures ISR updates are seen in the main loop.
- Bitmask handling is clean and correct.
- use **tight_loop_contents()** while idle

Week 2: Timers

Step 1: Timer

Goal:

- Blink one LED at 2 Hz, i.e. on for 0.5 s, off for 0.5 s, using a hardware timer instead of a delay function.
- Also, print the current system time using printf("%lld\n", time_us_64());.

Explanation of How It Works:

1. **Timer Setup**

- **add_repeating_timer_ms()** creates a repeating callback that runs automatically every 500 ms.

```
add_repeating_timer_ms(-500, toggle_led_callback, NULL, &timer);
```

    - Run the function toggle_led_callback() every 500 ms
    - Because the interval is negative, start the timer immediately and align it to system time boundaries.

2. **Callback Function**

```
bool toggle_led_callback(struct repeating_timer* t){
    static bool led_state = false;
    led_state = !led_state;
    gpio_put(LED, led_state);

    printf("%lld\n", time_us_64());  //Returns the current time in microseconds since the board started running.
    return true;
}
```

- It toggles the LED (on <-> off) using gpio_put(LED, led_state).
- Prints the current timestamp in **microseconds** via time_us_64().

3. **Main Loop**

- while (1) tight_loop_contents(); keeps the CPU awake and responsive while the hardware timer runs in the background.

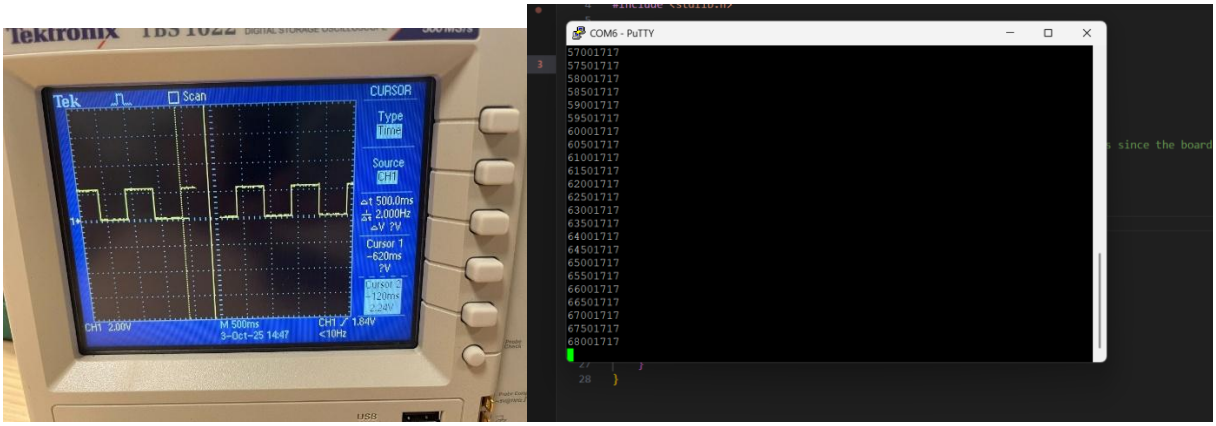
Question: What does the negative delay mean, and why is it handy ?

- A **negative delay** means the timer will fire its callback at a **regular time interval aligned to system time** instead of waiting for the first delay to pass.
- For example:
  - add_repeating_timer_ms(500, ...) waits **500 ms** before the first callback.
  - add_repeating_timer_ms(-500, ...) calls the first callback **immediately**, and then repeats every **500 ms**.
- **This is handy because it:**
  - Makes the timer start instantly.
  - Keeps the timing aligned to exact intervals (e.g. every half-second mark of the system clock).
  - Ensures stable, synchronized timing -- ideal for periodic tasks like LED blinking or sampling sensors.

Question: What is the (small) disadvantage of using a positive delay ?

- A **positive delay** waits one full period before the first callback.
  - The LED won't blink immediately (it starts after 500 ms).
  - The timer may drift slightly from exact system time boundaries, leading to small timing inaccuracies over long runs.

Step 2: Oscilloscope



Observation:

- When using a negative delay, the LED toggles immediately and stays perfectly aligned to 0.5 s intervals.
- When using a positive delay, there is a small initial delay before blinking starts, and the timing may drift slightly due to scheduling differences.
- On the oscilloscope, the negative-delay signal is consistent and periodic, while the positive-delay signal may show minor offset or jitter.

Week 3: ADC & UART

Step 1: UART communication

Goal:

- Control 4 LEDs through UART.
- The value received (0x0 -> 0xF) determines which LEDs are ON (binary display).
  UART uses:
  - 9600 baud
  - no flow control
  - FIFO disabled
  - RX interrupt to set LEDs immediately when data arrives.

How It Works

1. **UART setup**

```
uart_init(UART_ID, BAUD_RATE);
gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART); //Configures GPIO0 as the TX (transmit) pin for UART0.
gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART); //Configures GPIO1 as the RX (receive) pin for UART0.

uart_set_hw_flow(UART_ID, false, false); //Controls hardware flow control (CTS/RTS) for UART.
                                         // CTS (Clear to Send) => input signal from another device telling your Pico it's allowed to send data. (disable)
                                         //RTS (Request to Send) => output signal from Pico telling the other device it's ready to receive. (disable)

uart_set_fifo_enabled(UART_ID, false); //Disables FIFO. Each byte triggers the RX interrupt immediately for real-time processing.

//An IRQ stands for Interrupt Request
//Without interrupts, your program would have to constantly check (poll) for events like incoming UART data – wasting CPU time
//With an IRQ, the CPU can do other work, and when data arrives, the UART automatically triggers an interrupt, which runs your function (on_uart_rx()) right away
irq_set_exclusive_handler(UART0_IRQ, on_uart_rx);   //=> defines the function to run
irq_set_enabled(UART0_IRQ, true);                   //=> allows hardware to generate interrupts
uart_set_irq_enables(UART_ID, true, false);         //=> only RX triggers the interrupt.
```

- Initializes UART0 (pins 0 and 1).
- **Flow control disabled** (no CTS/RTS).
- **FIFO disabled** => every received byte instantly triggers the interrupt.
- The **RX interrupt** calls your handler on_uart_rx() whenever new data arrives.

2. **Interrupt handler**

```
//UART RX interrupt handler
void on_uart_rx(){
    while(uart_is_readable(UART_ID)){    //checks if there is any data waiting in the UART receive buffer
        uint8_t ch = uart_getc(UART_ID);
        set_leds(ch & 0x0F);  //Takes the lower 4 bits of the received byte. 0x0F = 00001111 (binary mask). ch & 0x0F strips away everything except the bottom 4 bits.
    }
}
//ch = 0xAF   = 1010 1111 (in binary)
//0x0F        = 0000 1111 (bit mask)
// => ch & 0x0F   = 0000 1111 = 0x0F  => removes the upper 4 bits, keeping only the last 4 bits => so keep only 1111
//Each bit is compared one by one – only bits where both are 1 remain 1.
```

- Reads each incoming byte.
- Uses ch & 0x0F to keep only the lower 4 bits.
- Updates LEDs via set_leds() accordingly.

3. LED update

```
void set_leds(uint8_t value){
    for(int i = 0; i < 4; i++){            // (value >> i) & 1: Shift the bits of value right by i positions and check whether the lowest bit is 1
        gpio_put(LED_PINS[i], (value >> i) & 1);   //shift 1 bit to the right and only keep the least significant bit so 1 0 1 0 (on off on off)
                                                   //The bits that fall off the right side are discarded, and zeros are added on the left.
    }
}
```

- Displays the binary pattern of the 4 lowest bits.
- Example: 0x0F (1111) => all LEDs ON, 0x05 (0101) => LED0 & LED2 ON.

4. Loopback test

- Connect TX -> RX with a jumper wire.
- In main(), send a test pattern:

```
for (uint8_t i = 0; i < 16; i++) {
    uart_putc(UART_ID, i);
    sleep_ms(200);
}
```

LEDs counting in binary 0000 -> 1111.

5. Peer-to-peer test

- Connect TX->neighbour's RX and RX->neighbour's TX, with common GND.
- Send values to each other and watch LEDs update live.

Question: what is the default parity, databits and stopbits setting? How can you change this?

- The default UART configuration is:
  - Data bits: 8
  - Parity: None
  - Stop bits: 1
- commonly written as 8N1
- How to change these setting:
  - You can change UART frame format using:
    - uart_set_format(UART_ID, data_bits, stop_bits, parity);

- Example:
  - uart_set_format(UART_ID, 7, 2, UART_PARITY_EVEN);
  - Parity options available:
    - UART_PARITY_NONE
    - UART_PARITY_EVEN
    - UART_PARITY_ODD

Step 2: AD conversion

Goal:

- Read a potentiometer (connected to GPIO 26 / ADC0) and display the MSB 4 bits on 4 LEDs.

How it works:

```
//init ADC
adc_init();
adc_gpio_init(26);  //GPIO26 -> ADC0
adc_select_input(0);

while(1){
    uint16_t result = adc_read();   //Read
    uint8_t msb = result >> 8;  //Shifts t
    //result = 0b1010 1100 1111 (12 bits)
    //result >> 8 → 0b0000 1010 (8 bits)
    set_leds(msb & 0x0F);
    sleep_ms(500);
}
```

- As you rotate the pot, the LEDs show a binary pattern of the top 4 bits of the ADC value.

Step 3: ADC & UART

- The ADC reads the potentiometer value every 500 ms.
- The upper 4 bits are sent through UART using uart_putc().
- The RX interrupt on the receiving Pico (or loopback) updates the LEDs.

This allows:

- Real-time analog-to-digital indication over UART.
- Two Picos to exchange live potentiometer data.

Week 4 - 5: SPI & I2C

Goal:

- Control LEDs using the MCP23S17 SPI I/O Expander.
- Output analog voltages using the SPI DAC.

- Read back DAC voltage using the Pi Pico's ADC.
- Combine both SPI devices to show data on LEDs and output voltage.
- Compare SPI with I2C in theory and practice.

Step 1: SPI I/O Expander (MCP23S17)

How it works

- The MCP23S17 adds 16 extra GPIO pins via SPI.
- Each group of 8 pins (Port A, Port B) is controlled by internal **registers**.

Registers Used:

Register	Address	Purpose	Data Written
IODIRA	0x00	Data direction for Port A (1 = input, 0 = output)	0x00 (all outputs)
GPIOB	0x0A	Controls output state for Port B LEDs	0–255 (LED binary display)

Question: Explain in your own words which control, register and data bytes to use, and how and where you found this information.

- In datasheet (8-bit I/O Expander) page 8, I found the SPI addressing structure as below:

FIGURE 1-5: SPI ADDRESSING REGISTERS

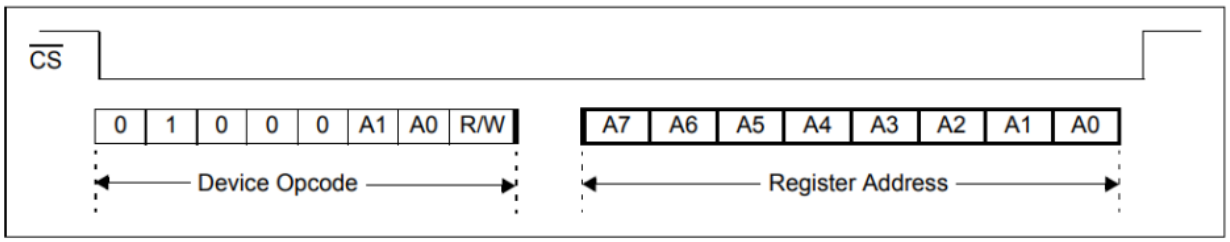
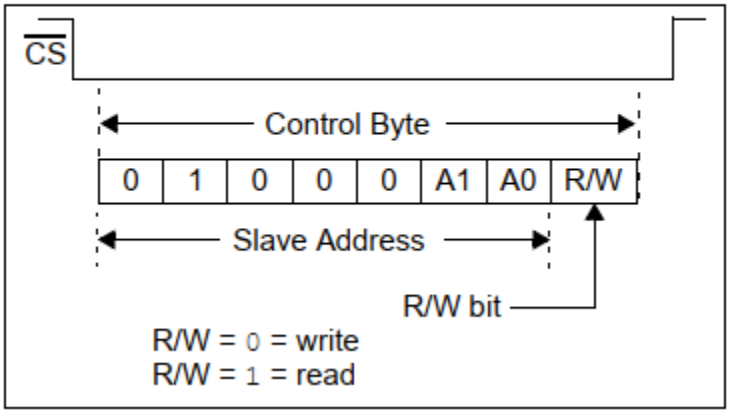


FIGURE 1-3: SPI CONTROL BYTE FORMAT



- The device op code is set to 0b01000000, since the chip is only going to send write operations to control the LEDs (bit 0 = 0)
- Secondly, as shown in page 9, I used 2 control registers (IODIR and GPIO):

TABLE 1-3: CONFIGURATION AND CONTROL REGISTERS

Register Name	Address (hex)	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	POR/RST value
IODIR	00	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0	1111 1111
IPOL	01	IP7	IP6	IP5	IP4	IP3	IP2	IP1	IP0	0000 0000
GPINTEN	02	GPINT7	GPINT6	GPINT5	GPINT4	GPINT3	GPINT2	GPINT1	GPINT0	0000 0000
DEFVAL	03	DEF7	DEF6	DEF5	DEF4	DEF3	DEF2	DEF1	DEF0	0000 0000
INTCON	04	IOC7	IOC6	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0	0000 0000
IOCON	05	—	—	SREAD	DISSLW	HAEN*	ODR	INTPOL	—	--00 000-
GPPU	06	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	0000 0000
INTF	07	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INTO	0000 0000
INTCAP	08	ICP7	ICP6	ICP5	ICP4	ICP3	ICP2	ICP1	ICP0	0000 0000
GPIO	09	GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0	0000 0000
OLAT	0A	OL7	OL6	OL5	OL4	OL3	OL2	OL1	OL0	0000 0000

* Not used on the MCP23008

- Page 10 shows me more about how to use IODIR to set the state of the LEDs, as all outputs (register bits are 0x00, or IODIR, data bits are 0x00, for all the LEDs are output)

```
// Set MCP23S17 port A as output -> drives LEDs
io_exp_write(0x00, 0x00); // IODIRA = 0 (all outputs)
```

- Finally, I use GPIO (reg bits 0x09) to control the state of the LED mask, as shown in page 19 of the datasheet:

1.6.10 PORT (GPIO) REGISTER

The GPIO register reflects the value on the port. Reading from this register reads the port. Writing to this register modifies the Output Latch (OLAT) register.

REGISTER 1-10: GPIO – GENERAL PURPOSE I/O PORT REGISTER (ADDR 0x09)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-0 **GP7:GP0:** These bits reflect the logic level on the pins <7:0>
1 = Logic-high.
0 = Logic-low.

- There are 8 data bits (or 1 byte), which come after register bits, each of them represents the state of the LED. This is how I send write operation to I/O expander

```c
//Write byte to MCP
void io_exp_write(uint8_t reg, uint8_t data)
{
    uint8_t buf[3] = {MCP_ADDR, reg, data}; // SPI write opcode
    cs_select();
    spi_write_blocking(spi0, buf, 3); //This sends the 3-byte array through the SPI0 peripheral to the MCP23S08
    cs_deselect();
    sleep_ms(10);
} //Ex:
//0x40 => Control byte (write command)
//0x09 => Register address (GPIO)
//0xFF => Data to write (set all pins high)
```

Question: Why are SPI Chip Select (CS) pins "low-active"?

- SPI uses an active-low Chip Select (CS) line to enable the correct device.
  - When CS = LOW, the selected chip's SPI input becomes active.
  - When CS = HIGH, the chip ignores the SPI clock and data lines. This prevents multiple devices on the same SPI bus from responding at the same time.
- This design is **hardware-safe** because:
  - Devices are inactive by default (CS HIGH).
  - Prevents multiple chips from driving the MISO line at once.
  - Saves power (chip stays idle when not selected).
- Example: If you pull both DAC and IO Expander CS LOW at the same time, both devices will respond and data will collide.

Step 2: SPI DAC

How it works:

- The DAC converts a 10-bit digital value (0–1023) into an analog voltage (0–Vref).

```c
// Write a value to DAC
void dac_write_spi(uint16_t value)
{

    //spi_set_format(spi0, 8, SPI_CPOL_1, SPI_CPHA_1, SPI_MSB_FIRST);

    // Set SPI for falling edge (Mode 1) => data is read when the clock goes from H to L
    //"The contents of the serial input register are transferred to the DAC register on the sixteenth falling edge of SCLK"
    //=> stable voltage output
    spi_set_format(spi0, 8, SPI_CPOL_0, SPI_CPHA_1, SPI_MSB_FIRST);

    value = value << 2; //moves the 10-bit DAC data into bits 11–2 of the 16-bit register
    uint8_t buf[2];

    buf[0] = (value >> 8) & 0xFF;   //High byte(control + top DAC bits)
                                    // (shift upper 8 bits of 16-bit register (bits 15-8) down into bits 7–0 of the byte)
                                    //& 0xFF ensures only 8 bits are sent via SPI

    buf[1] = value & 0xFF;  //Low byte (bottom DAC bits + padding)
                            //(Keeps lower 8 bits of value (bits 7–0))

    cs_select_dac(); //SYNC (CS) low => start of write
                                //16 falling edges of SCLK clock all bits into the DAC's serial input register.
    spi_write_blocking(spi0, buf, 2);   // DIN (Serial data input) + SCLK (Serial clock) (Sends 16 bits (control + 10-bit data) over DIN/SCLK)
    cs_deselect_dac();  //SYNC high => DAC latches value
    sleep_us(10);

    //"Once SYNC is low, the data on DIN is clocked into the 16-bit serial input register on the falling edges of SCLK"
    //"Normally, the SYNC line is kept low for at least 16 falling edges of SCLK and the DAC is updated on the 16th SCLK falling edge"
}
```

Question: What is the input and output range of the DAC chip?

- Digital input: 0-1023 (10-bit digital number)
- Analog output: 0 V -> Vref (usually 0–3.3V if powered at 3.3V)
- **Why:** The DAC converts digital signals into a smooth voltage for things like sensors, actuators, or LED brightness control.

Question: Besides the normal operation mode, what are the other modes used for? And how ?

- DACs typically have 3 modes (controlled by configuration bits):
  - Normal mode (DAC output enabled): Output follows the input value => Generate analog voltage
  - Shutdown mode (Output buffer off, DAC idle): Reduces current, disables output, or connects output via a resistor to GND => Save power
  - Gain mode (Adjust reference scaling (x1 or x2)): Change voltage range
- How: You set special bits in the 16-bit command sent over SPI.
- Why useful: You might want to turn off the DAC temporarily without cutting power or save battery in a low-power device.

Step 3: Combined SPI DAC + IO Expander

Goal:

- To output a voltage using the DAC (Digital-to-Analog Converter) (0 – 1023 => 0 – 3.3V) and simultaneously display the same value in binary on LEDs connected to the IO expander
- The IO Expander lights up LEDs to represent that same number in binary (0 = all off, 255 = all on).

Question: How do you "switch" between using one SPI chip and another ?

- The Chip Select (CS) pin chooses which device listens.

```
cs_select();
spi_write_blocking(spi0, buf, 3);
cs_deselect();
```

```
cs_select_dac(); //SYNC (CS) low =

spi_write_blocking(spi0, buf, 2);
cs_deselect_dac();  //SYNC high =>
```

- Each device has its own CS pin.
- Process:
  - Pull DAC CS LOW => send data => pull HIGH
  - Pull IO Expander CS LOW => send data => pull HIGH
- At no point are both CS LOW; this avoids bus collisions.

- You never pull both CS pins low simultaneously -> that's how you "switch" between devices.

Question: What is strange about the IO Expander's behavior?

- When switching rapidly between the DAC and IO Expander(if you don't wait or pulse CS properly), the LEDs may flicker or lag slightly.
- This happens because SPI mode settings differ:
  - IO Expander => Mode 0 (CPOL=0, CPHA=0)
  - DAC => Mode 1 (CPOL=0, CPHA=1)
- Each mode change reconfigures the SPI controller mid-loop, introducing tiny timing glitches or flickers.

Step 4: I2C vs SPI Comparison

Question: What is the difference between half-duplex and full-duplex? How does this relate to SPI ?

- Half-duplex: Send OR receive, not both at once (I2C, UART).
- Full-duplex: Send and receive simultaneously (SPI).
- SPI: Full-duplex (MOSI sends while MISO receives simultaneously).
- I2C: Half-duplex (master sends, then slave responds).

Question: Why does I2C need resistors? And why doesn't SPI need resistors? Explain the difference.

- I2C uses open-drain pins (outputs): devices can only pull the line LOW, not HIGH.
- Pull-up resistors are needed to bring the line HIGH when idle.
- SPI actively drives both HIGH and LOW signals (uses **push-pull** outputs), so no resistors are needed.

Question: When connecting multiple devices to the "bus" how do I2C and SPI "select" the correct device? Explain the difference.

- I2C: each device has a unique address (7-bit or 10-bit); the master sends the address before reading/writing.
- SPI: each device has its own CS pin; pulling it LOW selects that device.

Question: How is decided who the "master" is and who the "slave" in both I2C and SPI? Explain the difference.

- I2C: the master is the device that generates the clock. Slaves follow.
- SPI: the master always generates the clock and controls CS; slaves follow commands.

- Usually, Pico is master, DAC and IO Expander are slaves.

Question: Explain in your own words the difference between a NACK and an ACK in I2C.

- ACK (Acknowledge): Receiver pulls SDA LOW after each byte => the slave successfully received a byte (data).

- NACK (Not Acknowledge): SDA left HIGH => the slave did not receive it or can't accept more data.

- It's a way to confirm successful communication.

Question: How is the byte data send in I2C and SPI: MSB or LSB first ? and can this be changed in the Pi Pico?

- By default, MSB first (most significant bit sent first) for both I2C and SPI.

- On the Pi Pico, SPI allows changing to LSB first using spi_set_format().

- I2C is typically fixed as MSB first.

- Example: spi_set_format(spi0, 8, SPI_CPOL_0, SPI_CPHA_0, SPI_LSB_FIRST);

- Value: 0b11001010

- MSB first => sends 1 first, then 1, then 0,...

- LSB first => sends 0 first, then 1, then 0,...

WEEK 6-10: FINAL ASSIGNMENT

Question: what is the highest precision?

- For temperature sensors: Resolution (precision): 0.125 °C (11-bit ADC)
- For ambient light sensors: up to 20 bits resolution.

In your logbook you describe your design and how to "control" the chips, i.e. which bytes should go over the line, how you implemented and tested your setup.

- Spi: control LEDs, default pins
- I2C: control temperature and light sensor (LM75B and APDS respectively)
    - SDA: GPIO 4
    - SCL: GPIO 5
    - OS: (temperature interrupt) GPIO 6 (input, pull-up)
    - INT: (light interrupt) GPIO 7 (input, pull-up)

LM75B: Temperature sensor

Son Cao – 570135

- I2C address: 0x49

- Control protocol: Send read/write request/command from master to chip through

- SDA line. For example, reading temperature measurements implemented as below:

  - Write TEMP register (0x00) then read 2 bytes, combine into 16-bit and shift to the left side 5 bits (11-bit), multiply with resolution (0.125°C per LSB)

```
//2 bytes, use only the 11 MSBs
float read_temp(void){
    uint8_t reg = TEMP; //Temperature register address
    uint8_t buf[2];     //8-bit
    i2c_write_blocking(i2c_default, TEMP_ADDR, &reg, 1, true);
    i2c_read_blocking(i2c_default, TEMP_ADDR, buf, 2, false); /

    int16_t raw = ((int16_t)buf[0] << 8) | buf[1]; //Combine th
    raw >>= 5;                                     //11-bit  //
    if(raw & 0x0400) raw |= 0xF800;     //check the sign bit la
    return raw * 0.125f;                //Tempurature resolutio
```

Registers:

Table 7. Pointer value

B1	B0	Selected register
0	0	Temperature register (Temp)
0	1	Configuration register (Conf)
1	0	Hysteresis register (Thyst)
1	1	Overtemperature shutdown register (Tos)

APDS: Ambient light sensor

- I2C address: 0x52

- Register:

```
//light reg list
#define MAIN_CTRL        0x00
#define ALS_MEAS_RATE    0x04
#define ALS_GAIN         0x05
#define MAIN_STATUS      0x07
#define ALS_DATA_0       0x0D
#define INT_CFG          0x19
#define INT_PERSIST      0x1A
#define ALS_THRES_UP_0   0x21
#define ALS_THRES_UP_1   0x22
#define ALS_THRES_UP_2   0x23
#define ALS_THRES_LOW_0  0x24
#define ALS_THRES_LOW_1  0x25
#define ALS_THRES_LOW_2  0x26
```

- Control protocol:
    - Initialize: set measurement rate, resolution, gain and then turn ALS on to start measurement, using “i2c_write_blocking” with corresponding register address and data value (specified in datasheet)
    - Read measurements: send a write operation of register “ALS_DATA_0” and then send a read operation, reading 3 bytes, and then process it:

```
uint32_t read_light(void){
    uint8_t reg = ALS_DATA_0; //ALS reg address
    uint8_t buf[3];  //0x0D[7:0], 0x0E[15:8], 0x0F[19:16]
    i2c_write_blocking(i2c_default, LIGHT_ADDR, &reg, 1, true); //sets the internal pointer to reg
    i2c_read_blocking(i2c_default, LIGHT_ADDR, buf, 3, false); //read and save to buf

    int32_t raw = ((uint32_t)(buf[2] & 0x0F) << 16) | ((uint32_t)buf[1] << 8) | (uint32_t)buf[0];
    return raw;
}
```

Implementation and testing: print the data with current mode into the serial monitor, as well as display I/O expander LEDs accordingly to reading values

Optional Assignment – Threshold Interrupts

Both sensors support interrupt generation:

LM75B:

- **OS (Overtemperature Shutdown) pin** goes LOW when temperature > Tos (24 °C in my case).
- Configure:
    - Write threshold to Tos register.
    - Write hysteresis value to Thyst.
    - OS pin can trigger LED

APDS-9306:

- **INT pin** triggers when light exceeds or drops below set thresholds.

- Enable interrupt:
    - Set ALS_INT_EN = 1
    - Set upper and lower threshold registers.
- Can use interrupt to blink LED
- Then clear previous interrupt by reading MAIN_STATUS

Test plan:

- Use bright light to trigger APDS interrupt.
- Heat sensor (with finger) to trigger LM75B threshold interrupt.
- Observe LED or serial output confirming interrupt event.

HOW MY CODE WORKS:

Pin Setup

- SDA (GP4) and SCL (GP5) are used for I2C to send and receive data between the Pico and the sensors.
- SPI pins (GP18, 19, 16) and CS (GP17) are used for the LED expander.
- OS (GP6) is the output pin from the temperature sensor that goes LOW when it's too hot.
- INT (GP7) is the output pin from the light sensor that goes LOW when the light level is too high or too low.
- BUTTON (GP15) is used to switch between showing temperature or light data.

```
#define PIN_SCK 18  //c
#define PIN_MOSI 19 //c
#define PIN_MISO 16 //c
#define PIN_CS 17 //or
#define BUTTON_PIN 15

#define SDA 4  //send
#define SCL 5   // cloc
#define OS 6    //inter
#define INT 7   //inter
```

MCP23S08 (SPI I/O Expander)

- This chip controls the LEDs.
- I send 3 bytes through SPI: [control byte][register][data].
- Example:
    - [0x40][0x09][0xFF] turns all LEDs ON.
    - [0x40][0x09][0x00] turns them OFF.
- In the setup, I set all its pins as **outputs**, so I can send the sensor data to the LEDs easily.

LM75B (Temperature Sensor)

- The LM75B measures temperature in steps of **0.125°C** and works automatically after power-up.
- To read it, I write the register address 0x00 and then read two bytes of data.
- These two bytes are combined into an 11-bit value, which is converted into Celsius.
- The function temp_set_thresholds() sets the **temperature limit** to 24°C.
- If the temperature goes higher, the **OS pin** goes LOW and triggers a warning.

```
void temp_set_thresholds(){
    uint16_t raw = (uint16_t)(TEMP_THRESHOLD / 0.5) << 7; //shift

    uint8_t thyst[3] = {THYST, (raw >> 8) & 0xFF, raw & 0xFF}; //
    i2c_write_blocking(i2c_default, TEMP_ADDR, thyst, 3, false);

    uint8_t tos[3] = {TOS, (raw >> 8) & 0xFF, raw & 0xFF};
    i2c_write_blocking(i2c_default, TEMP_ADDR, tos, 3, false);
}
```

APDS-9306 (Light Sensor)

- The APDS-9306 measures ambient light with very high accuracy (20-bit).
- I enable it with [0x00][0x02] and set it to high precision mode [0x04][0x04].
- I also set gain = 3x with [0x05][0x01].
- Then I set two light thresholds (low = 100, high = 16000).
- If the light goes outside that range, the sensor sends an **interrupt** on the INT pin.
- The Pico reads three bytes of data from the sensor, combines them into one 20-bit value, and shows it on the LEDs and serial monitor.

```
void amb_light_init_interrupt(void){
    //EN ALS engine
    uint8_t cmd1[2] = {MAIN_CTRL, 0x02};
    i2c_write_blocking(i2c_default, LIGHT_ADDR, cmd1, 2, false);

    // ALS bit width: 20-bit 400ms integration time, 500ms mesure
    uint8_t cmd2[2] = {ALS_MEAS_RATE, 0x04};
    i2c_write_blocking(i2c_default, LIGHT_ADDR, cmd2, 2, false);

    // Gain = 3 (0x01)
    uint8_t cmd3[2] = {ALS_GAIN, 0x01};
    i2c_write_blocking(i2c_default, LIGHT_ADDR, cmd3, 2, false);

    // lower and upper thresholds (20-bit values)
    uint32_t low = 100;  // test value
    uint32_t high = 16000;
```

```
uint8_t low_buf[4] = {ALS_THRES_LOW_0,
    (uint8_t)(low & 0xFF),  //0xFF 8-bit, LSB
    (uint8_t)((low >> 8) & 0xFF), //intervening byte
    (uint8_t)((low >> 16) & 0x0F)}; //0x0F 4bit, MSB
i2c_write_blocking(i2c_default, LIGHT_ADDR, low_buf, 4, false);

uint8_t high_buf[4] = {ALS_THRES_UP_0,
    (uint8_t)(high & 0xFF),
    (uint8_t)((high >> 8) & 0xFF),
    (uint8_t)((high >> 16) & 0x0F)};
i2c_write_blocking(i2c_default, LIGHT_ADDR, high_buf, 4, false);
```

```
// Interrupt persistence = trigger after 2 consecutive threshold
uint8_t persist[2] = {INT_PERSIST, 0x10};
i2c_write_blocking(i2c_default, LIGHT_ADDR, persist, 2, false);

// Configure interrupt: ALS source, threshold mode, active low
uint8_t int_cfg[2] = {INT_CFG, 0x14}; //00010100 enable interrupt
i2c_write_blocking(i2c_default, LIGHT_ADDR, int_cfg, 2, false);

// Clear any pending interrupt flag (old interrupts => Makes sure
uint8_t reg = MAIN_STATUS; //Sometimes the sensor might already h
uint8_t st;
i2c_write_blocking(i2c_default, LIGHT_ADDR, &reg, 1, true);
i2c_read_blocking(i2c_default, LIGHT_ADDR, &st, 1, false);
```

Button and Interrupts

- All interrupts are handled in one function called gpio_callback().
- If I press the button, it changes between showing **temperature** or **light** mode.
- If the **OS pin** goes LOW, it means the temperature is too high => the Pico flashes the LEDs.
- If the **INT pin** goes LOW, it means the light is too strong => the LEDs also flash.
- The function flash_warning() makes the LEDs blink three times to show the alert.

```
void gpio_callback(uint gpio, uint32_t events){
    static uint64_t last_us = 0; //last time the button was pre
    uint64_t now = time_us_64(); //current time in microseconds
    if(gpio == BUTTON_PIN && (events & GPIO_IRQ_EDGE_FALL)){
        if(now - last_us > 200000){ //200ms debouncing (ignore
            mode_switched = !mode_switched;
            last_us = now; //update timestamp
        }// Detect that the button was pressed, prevent button
    // Toggle the display mode (switch between temperature and
    } else if (gpio == OS && (events & GPIO_IRQ_EDGE_FALL)){
        hot = true;
    } else if (gpio == INT && (events & GPIO_IRQ_EDGE_FALL)) {
        light_alert = true;
    }
}
```

Son Cao – 570135