Input Drivers

Class: PowerButton

- Interface (Functions):
    - def __init__(self, pin_number):
        - Stores the pin this button is connected to.
    - def check_press(self):
        - Checks if the button was just pressed (and released).
        - Includes "debouncing" (to ignore electrical noise).
        - Returns: True if a single, clean press just happened, False otherwise.

Class: ModeButton

- Interface (Functions):
    - def __init__(self, pin_number):
        - Stores the pin this button is connected to.
    - def check_press(self):
        - Same as PowerButton's check_press.
        - Returns: True if a single, clean press just happened, False otherwise.

Class: ToFSensor

- Interface (Functions):
    - def __init__(self, i2c_bus):
        - Sets up the connection to the sensor.
    - def read_distance(self):
        - Asks the sensor for a measurement.
        - Returns: The distance in millimeters (e.g., 50).

Class: HallSensor (Trach Tube Sensor)

- Interface (Functions):
    - def __init__(self, pin_number):

        - Stores the pin this sensor is connected to.
    - def get_status(self):
        - Reads the sensor's value.
        - Returns: A simple string: "TUBE_IN" or "TUBE_OUT".

Output Drivers

Class: ModeLEDs

- Interface (Functions):
    - def __init__(self, pin_obstruction, pin_dislodge, pin_mystery):
        - Stores the pin for each of the three mode LEDs.
    - def set_mode(self, mode_name):
        - Receives a string: "OBSTRUCTION", "DISLODGEMENT", or "MYSTERY".
        - If "OBSTRUCTION", turn on obstruction LED and turn off the other two.
        - etc.
    - def turn_all_off(self):
        - Turns off all three mode LEDs.

Class: SevenSegmentDisplay

- Interface (Functions):
    - def __init__(self, i2c_bus):
        - Sets up the connection to the display.
    - def show_text(self, text):
        - Receives a string like "SEL", "WIN", "LOSE".
        - Figures out which segments to light up to show this text.
    - def show_number(self, number):
        - Receives a number like 60 or 59.
        - Shows that number on the display.
    - def turn_off(self):

        - Clears the display.

Class: Buzzer

- Interface (Functions):
    - def __init__(self, pin_number):
        - Stores the pin this buzzer is connected to.
    - def play(self, sound_name):
        - Receives a string: "WIN", "LOSE", "WARN", or "STOP".
        - If "WIN", play a happy series of beeps.
        - If "WARN", play a repeating pulse.
        - If "STOP", stop all sound.

Class: FeedbackLEDs (Win/Loss)

- Interface (Functions):
    - def __init__(self, green_pin, red_pin):
        - Stores the pins for the win/loss LEDs.
    - def show_feedback(self, feedback_type):
        - Receives a string: "WIN", "LOSE", or "OFF".
        - If "WIN", flash the green LED.
        - If "LOSE", flash the red LED.
        - If "OFF", turn both LEDs off.

Class: Servo

- Interface (Functions):
    - def __init__(self, pin_number):
        - Stores the pin for the servo.
    - def move_to(self, position_name):
        - Receives a string: "HOME", "OBSTRUCT", or "DISLODGE".
        - Moves the servo arm to the correct angle for that position.

Software "Driver"

Class: GameTimer

- Interface (Functions):
    - def __init__(self, duration_in_seconds):
        - Stores the total time for the game.
    - def start(self):
        - Records the current system time as the start time.
    - def get_time_left(self):
        - Checks current time vs. start time.
        - Returns: Number of seconds left.
    - def is_finished(self):
        - Checks if the time left is 0 or less.
        - Returns: True if time is up, False otherwise.

3. Main Game Logic Pseudocode

This is the "Brain" class that runs the whole show.

Class: TrachTrainerGame

- def __init__(self): (The setup function)
    - --- Create all driver objects ---
    - self.power_button = PowerButton(pin=...)
    - self.mode_button = ModeButton(pin=...)
    - self.tof_sensor = ToFSensor(bus=...)
    - self.hall_sensor = HallSensor(pin=...)
    - self.mode_leds = ModeLEDs(pin1=..., pin2=..., pin3=...)
    - self.display = SevenSegmentDisplay(bus=...)
    - self.buzzer = Buzzer(pin=...)
    - self.feedback_leds = FeedbackLEDs(green=..., red=...)

- self.servo = Servo(pin=...)
- self.timer = GameTimer(duration_in_seconds=60)
- --- Define game modes and states ---
- self.game_modes = ["OBSTRUCTION", "DISLODGEMENT", "MYSTERY"]
- self.current_mode_index = 0
- self.game_state = "OFF" # Other states: "IDLE", "GAME_ACTIVE", "GAME_OVER"
- --- Game-specific variables ---
- self.tube_was_removed = False
- self.game_over_start_time = 0

- def run_game_loop(self):
  - while True:
    - 1. check the power button
    - if self.power_button.check_press():
      - if self.game_state == "OFF":
        - self.game_state = "IDLE"
        - # (Set servo to HOME, turn on default mode LED, show "SEL" on display)
        - self.servo.move_to("HOME")
        - self.mode_leds.set_mode(self.game_modes[self.current_mode_index])
        - self.display.show_text("SEL")
      - else:
        - self.game_state = "OFF"
        - # (Turn everything off)
        - self.mode_leds.turn_all_off()
        - self.display.turn_off()

            - self.buzzer.play("STOP")
            - self.feedback_leds.show_feedback("OFF")
            - self.servo.move_to("HOME")
        - 2. Run logic based on the current game state
        - if self.game_state == "IDLE":
            - self.run_idle_logic()
        - elif self.game_state == "GAME_ACTIVE":
            - self.run_game_active_logic()
        - elif self.game_state == "GAME_OVER":
            - self.run_game_over_logic()
        - # Wait a tiny bit to prevent CPU from running at 100%
        - sleep(0.01)
- def run_idle_logic(self):
    - Check for mode change
    - if self.mode_button.check_press():
        - self.current_mode_index = (self.current_mode_index + 1) % len(self.game_modes)
        - current_mode_name = self.game_modes[self.current_mode_index]
        - self.mode_leds.set_mode(current_mode_name)
    - Check for game start
    - distance = self.tof_sensor.read_distance()
    - if distance < 50: # (e.g., 50mm = "suction tool is present")
        - self.start_game()
- def run_game_active_logic(self):
    - Check for timer loss
    - if self.timer.is_finished():

        - self.end_game(did_win=False)
        - return # (Exit this function early)
    - Show time left
    - time_left = self.timer.get_time_left()
    - self.display.show_number(time_left)
    - Run logic for the specific mode
    - current_mode_name = self.game_modes[self.current_mode_index]
    - if current_mode_name == "OBSTRUCTION":
    - distance = self.tof_sensor.read_distance()
    - if 30 < distance < 40: # (e.g., 30-40mm = "perfect depth")
    - self.end_game(did_win=True)
    - elif distance < 50: # (e.g., < 50mm = "suction present but wrong depth")
    - self.buzzer.play("WARN")
    - else:
    - self.buzzer.play("STOP")
    - elif current_mode_name == "DISLODGEMENT":
    - status = self.hall_sensor.get_status()
    - if status == "TUBE_OUT":
    - self.tube_was_removed = True
    - self.buzzer.play("WARN")
    - elif status == "TUBE_IN" and self.tube_was_removed == True:
    - # (They put it back in AFTER taking it out!)
    - self.end_game(did_win=True)
- def start_game(self):
    - self.game_state = "GAME_ACTIVE"
    - self.buzzer.play("STOP")

    - # (Reset game variables)
    - self.tube_was_removed = False
    - # (Move servo to "problem" position)
    - current_mode_name = self.game_modes[self.current_mode_index]
    - if current_mode_name == "OBSTRUCTION":
    - self.servo.move_to("OBSTRUCT")
    - elif current_mode_name == "DISLODGEMENT":
    - self.servo.move_to("DISLODGE")
    - # (Start timer)
    - self.timer.start()
- def end_game(self, did_win):
    - self.game_state = "GAME_OVER"
    - self.buzzer.play("STOP")
    - # (Move servo back home)
    - self.servo.move_to("HOME")
    - if did_win:
    - self.display.show_text("WIN")
    - self.feedback_leds.show_feedback("WIN")
    - self.buzzer.play("WIN")
    - else:
    - self.display.show_text("LOSE")
    - self.feedback_leds.show_feedback("LOSE")
    - self.buzzer.play("LOSE")
    - # (Record time to know when to go back to IDLE)
    - self.game_over_start_time = current_system_time()
- def run_game_over_logic(self):

- # (Wait for 5 seconds before going back to IDLE)
- if current_system_time() - self.game_over_start_time > 5:
- self.game_state = "IDLE"
- # (Reset all feedback)
- self.buzzer.play("STOP")
- self.feedback_leds.show_feedback("OFF")
- self.mode_leds.set_mode(self.game_modes[self.current_mode_index])
- self.display.show_text("SEL")