Linnæus University
Sweden

Assignment 3

Fault tolerance of Embedded System

Assignment 3

Author: Michael Daun
Supervisor: Mehdi Saman Azari
Semester: HT24
Course code: 2DT303

Table of Contents

Introduction to Fault Tolerance

Fault tolerance is about accepting that some faults can't always be avoided. We therefore must be prepared for these faults and implement solutions that can handle the faults as they occur so the system can still be able to be as functional as required of it. As described in [1] "Fault tolerance means to avoid service failures in the presence of faults."

In real world applications this means that systems that have a critical function have procedures in place for certain events or faults that can't be avoided. A common example for this is airplanes that have several engines for redundancy reasons in case one or several engines fail the plane can still be operational [1].

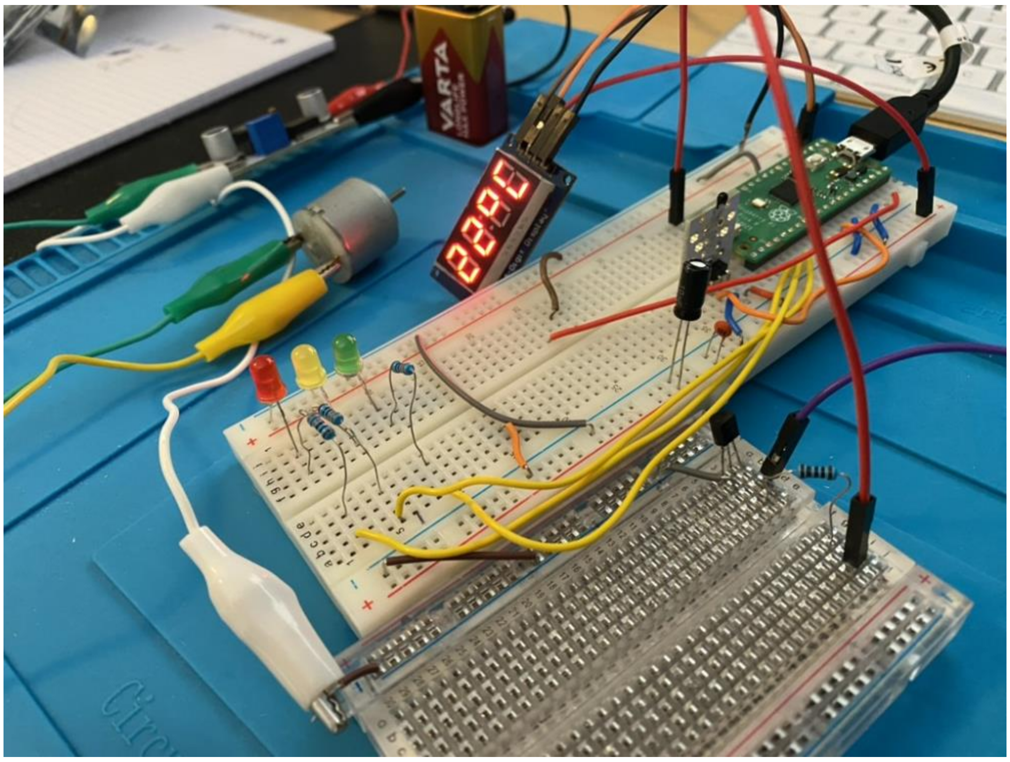
Designers of embedded systems and Raspberry Pi Pico in particular have to have a certain amount of fault tolerance in mind. Whether it be code that must avoid getting stuck in deadlocks or if individual weaker components have to be implemented with backup in case of failure. The Raspberry Pi Pico is a low-cost microcontroller with limited built-in fault tolerance. It needs to be paired with external components that can ensure that it doesn't get fed with high voltage spikes and gets a stable current source without interruptions.

Common types of faults that I can list from the top of my mind is:

1. **Transient power faults:** These include temporary power loss, voltage fluctuations, or power spikes, which can cause unexpected system behavior but are typically short-lived.
2. **Software bugs and deadlocks:** Code issues, such as logical errors, race conditions, or deadlocks, where the system becomes unresponsive or stuck in an infinite loop.
3. **Hardware degradation and component failure:** Over time, hardware components can wear out or fail, leading to malfunctions or degraded system performance.
4. **Temperature issues:** Overheating is a significant problem that can damage an embedded system, but low temperatures can also cause issues. In any case, temperature affects the performance of the system and its individual components.

Baseline Project

My baseline project is a temperature-controlled DC motor with a feedback display. A **Raspberry Pi Pico** is used in conjunction with an analog temperature sensor (**NTC thermistor**). The Pico reads the analog signal from the sensor and converts it to Celsius using the **Steinhart-Hart equation**. The temperature is then displayed on an external **TM1637 display**. Depending on the measured temperature, the Pico may activate an **S8050 NPN transistor**, which completes the circuit to allow an external battery to power the DC motor. Additionally, three LEDs (red, yellow, and green) act as status indicators for the motor, displaying the current power level based on the PWM signal sent to the motor. The external battery's voltage is regulated using an **LM2596S buck converter**. Two capacitors were added to provide a more stable electrical signal to the thermistor and to lower electrical disturbances from the DC-motor.



Hardware Setup 1. My Baseline Project as Described.

Identifying Fault Scenarios

Scenario	Description	Potential Impact	Priority (1 - 5)
Power Outage	A loss of power supply to the Raspberry Pi Pico due to voltage fluctuations or battery failure.	Pico will shut down, stopping all system operations, including motor control, temperature sensing, and display updates.	5

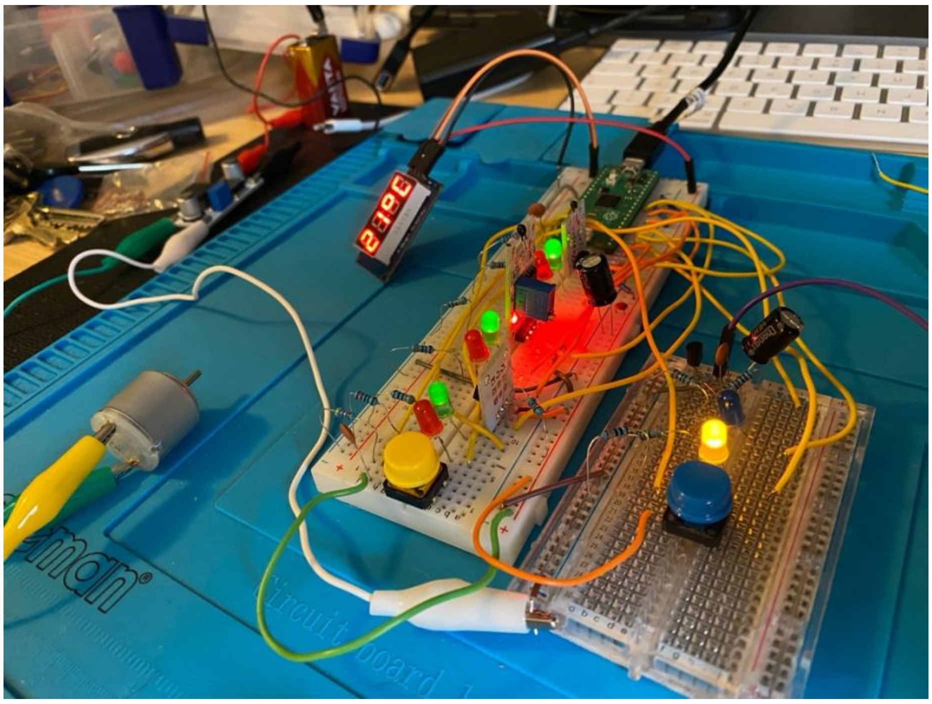
Voltage Fluctuations	Inconsistent voltage from one of the power sources (USB or 9V battery).	The motor may run erratically, sensor readings may become unstable, and the system may behave unpredictably.	5
Sensor Malfunction (NTC Thermistor)	The temperature sensor might fail due to broken connections, shorts or incorrect voltage supplied.	Incorrect temperature readings will lead to faulty motor control. It could activate when it shouldn't and operate at the wrong RPM.	4
Transistor Failure (S8050)	The transistor fails to switch properly, either due to insufficient base current or faulty connections.	The motor might not turn on or could stay on continuously, causing system malfunction and potential motor damage.	4
Motor Overload	The size of the DC-motor makes it more prone to damage from excessive current draw. A motor will also degrade by time making it more likely to fail.	Overloading may cause the motor to overheat or burnout disabling a vital part of the system.	3
Wrong PWM Settings	The PWM signal needs to be tailored for its purpose in the system, a faulty setting could potentially harm the system.	A too high signal might damage the motor and a low signal could make the performance uneven.	2
Software Crash or Infinite Loop	Errors in the code causing the Pico to crash or get stuck in an infinite loop.	The system becomes "soft locked" and stuck in its current state. Potentially harming components and making the system non-functional.	5
Communication Failures with Display	The TM1637 display may lose connection with the Pico, resulting in incorrect or missing temperature data.	Users will not be able to monitor temperature in real-time.	2
Environmental Factors	Extreme temperatures or moisture could affect the operation of both the Pico and the motor.	Temperatures outside of the manufacturers recommendations may result in failure of individual components. Moisture could result in short circuits which could severely damage the system.	1
Loose or Broken Connections	Physical connections between components	The system may lose connection between components causing inconsistent or incorrect operation.	3

	may become loose or break.		

Developing Fault Tolerance Strategies

Tripple Modular Redundancy (TMR)

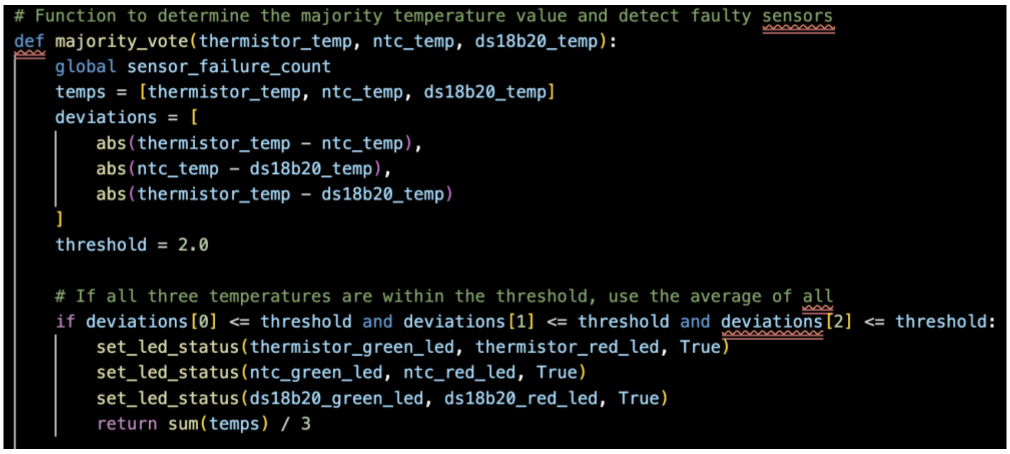
To increase the system's reliability and fault tolerance, I implemented a TMR system. This aims to safeguard against sensor related faults. I used a **DS18B20** digital temperature sensor, an **NTC-MF52** thermistor, and the already mentioned basic NTC Thermistor sensor. Since these sensors operate differently, they were manually calibrated to align with each other. After calibration, their readings rarely deviated by more than 0.5 degrees Celsius. To indicate the operational status of each sensor, I added both a red and a green LED. The green LED lights up if the sensor is functioning correctly, while the red LED indicates a fault. I replaced the previous LED used for indicating system status with a yellow LED, which now shows if the system is running continuously. It also blinks during system reboot sequences. Additionally, a blue LED is used to indicate the motor's power. This LED mirrors the motor's intensity, utilizing PWM from the Pico to provide a visual representation of the motor's output.



Hardware Setup 2. Fault Tolerant System with TMR.

Code for TMR

The Raspberry Pi Pico continuously monitors the temperature from all three sensors. The temperature displayed is the result of a majority vote, averaging the outputs of the sensors. If one sensor deviates significantly from the other two, it is suspended and excluded from the vote until its readings return within an acceptable threshold. If a sensor persistently reports values outside the acceptable range, the system will reboot itself to attempt recalibration and restore sensor accuracy.

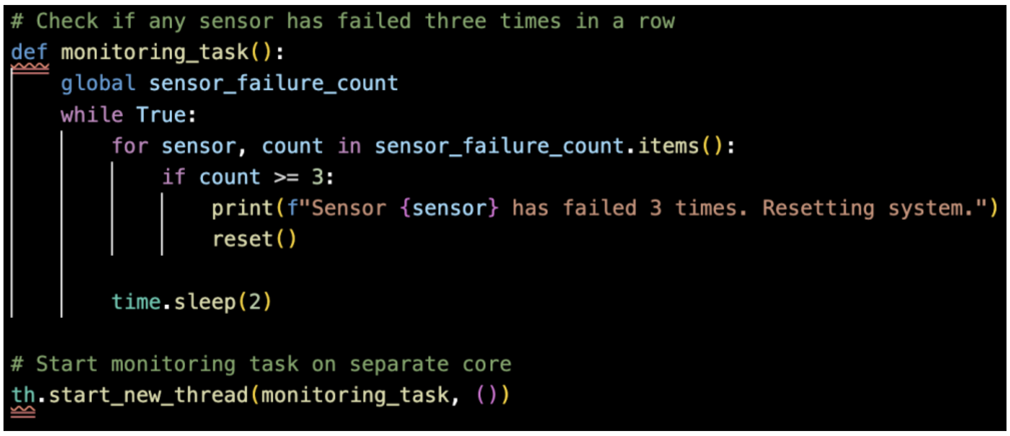


```
# Function to determine the majority temperature value and detect faulty sensors
def majority_vote(thermistor_temp, ntc_temp, ds18b20_temp):
    global sensor_failure_count
    temps = [thermistor_temp, ntc_temp, ds18b20_temp]
    deviations = [
        abs(thermistor_temp - ntc_temp),
        abs(ntc_temp - ds18b20_temp),
        abs(thermistor_temp - ds18b20_temp)
    ]
    threshold = 2.0

    # If all three temperatures are within the threshold, use the average of all
    if deviations[0] <= threshold and deviations[1] <= threshold and deviations[2] <= threshold:
        set_led_status(thermistor_green_led, thermistor_red_led, True)
        set_led_status(ntc_green_led, ntc_red_led, True)
        set_led_status(ds18b20_green_led, ds18b20_red_led, True)
        return sum(temps) / 3
```

Code 1. Part of Majority Vote System.

A separate thread is responsible for checking if any of the sensors are faulty.



```
# Check if any sensor has failed three times in a row
def monitoring_task():
    global sensor_failure_count
    while True:
        for sensor, count in sensor_failure_count.items():
            if count >= 3:
                print(f"Sensor {sensor} has failed 3 times. Resetting system.")
                reset()

        time.sleep(2)

# Start monitoring task on separate core
th.start_new_thread(monitoring_task, ())
```

Code 2. Separate Thread Checks Sensor Reliability.

Software Fault Tolerance Strategies

To improve the software's reliability, I implemented a watchdog timer to monitor the system's responsiveness and ensure it functions as intended. I used the Pico's internal watchdog timer, configuring it with an 8-second timeout. This setup ensures that if the system doesn't respond within that period the watchdog timer will reset the system. This approach safeguards against potential software crashes, infinite loops, or unresponsive states.

```
while True:
    wdt.feed()
    read_and_validate_temperatures()
    time.sleep(5)
```

Code 3. Main Loop of The System with Watchdog Timer.

Testing

Sensor Manipulation

To validate the TMR system, I manually manipulated the sensors by warming them with my fingers to simulate a fault scenario. This test created conditions where one sensor deviated significantly from the others, triggering the system's response. As designed, the sensor was quickly disabled when its temperature readings fell outside the acceptable threshold compared to the other two sensors. This procedure was repeated for all temperature sensors to ensure consistent system behavior. Additionally, prolonged manipulation of a sensor resulted in the monitoring core detecting the persistent fault, prompting the system to reboot, as intended.

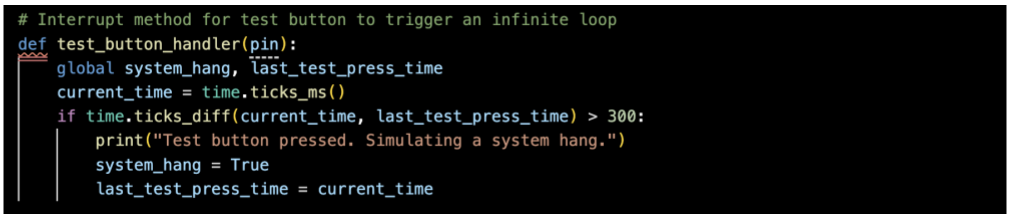
Core Functionality and Watchdog Tests

With the TMR system functioning effectively, testing the motor and other system components required a creative approach. The motor is programmed to activate at specific temperature thresholds, making it challenging to manually adjust the temperature across all sensors simultaneously due to the TMR system's fault tolerance. To facilitate testing of the system's core functionality and the watchdog timer, I installed two push buttons. The first button applies a temperature offset, allowing me to bypass the TMR system and trigger the motor activation manually.

```
# Interrupt method for button press, apply temperature offset with each press
def button_handler(pin):
    global temp_offset, last_press_time
    current_time = time.ticks_ms()
    if time.ticks_diff(current_time, last_press_time) > 300:
        temp_offset += 2
        if temp_offset > 10:
            temp_offset = 0
        print(f"Button pressed, temperature offset: +{temp_offset} C")
        last_press_time = current_time
```

Code 4. Button Implementation For Temperature Offset.

The second button forces the system into an infinite loop, effectively simulating a system hang and triggering the watchdog timer.



```python
# Interrupt method for test button to trigger an infinite loop
def test_button_handler(pin):
    global system_hang, last_test_press_time
    current_time = time.ticks_ms()
    if time.ticks_diff(current_time, last_test_press_time) > 300:
        print("Test button pressed. Simulating a system hang.")
        system_hang = True
        last_test_press_time = current_time
```

Code 5. Button Implementation For Testing The Watchdog Timer.

Analysis

The implemented solutions have proven to be highly effective. While the overall complexity of this system is relatively low, numerous factors can still cause malfunctions. The TMR (Triple Modular Redundancy) system provides robust fault tolerance, but the main challenge lies in integrating the sensors so they work effectively together. Once this is achieved, the probability of a faulty sensor reading is significantly reduced.

Ensuring fault tolerance in the software is just as critical as in the hardware. The watchdog timer serves as a straightforward and effective solution for preventing temporary system soft locks due to unexpected errors. Additionally, utilizing both cores of the Pico was a priority. One core is dedicated to monitoring sensor performance, which allows the system to reboot if persistent faults are detected. Alternatively, a different approach could involve permanently disabling any sensor that the monitoring core identifies as faulty. This strategy allows the system to perform its own health checks, enabling early fault detection and, if possible, preventing them from affecting overall functionality.

In summary, the fault tolerance strategies implemented proved to be effective for this system.

Sources

[1]: J. Knight, *Fundamentals of Dependable Computing for Software Engineers*, Boca Raton, FL: CRC Press, Taylor & Francis Group, 2012.

Appendix

Code

```
import math
import time
import _thread as th
from machine import Pin, ADC, PWM, WDT, reset
import onewire, ds18x20, tm1637

# Setting up hardware
display = tm1637.TM1637(clk=Pin(0), dio=Pin(1))
adc_thermistor = ADC(Pin(26))
adc_ntc = ADC(Pin(27))
ds_pin = Pin(20)
ds_sensor = ds18x20.DS18X20(onewire.OneWire(ds_pin))
# Motor
motor_pin = Pin(15)
pwm_motor = PWM(motor_pin)
pwm_motor.freq(500)
# LED Pins
blue_led = PWM(Pin(19))  # Blue LED for motor intensity
blue_led.freq(500)
yellow_led = Pin(18, Pin.OUT)  # Yellow LED for system status
thermistor_green_led = Pin(28, Pin.OUT)
thermistor_red_led = Pin(22, Pin.OUT)
ntc_green_led = Pin(16, Pin.OUT)
ntc_red_led = Pin(17, Pin.OUT)
ds18b20_green_led = Pin(10, Pin.OUT)
ds18b20_red_led = Pin(8, Pin.OUT)
# Button Pins
button = Pin(12, Pin.IN, Pin.PULL_DOWN)
test_button = Pin(14, Pin.IN, Pin.PULL_DOWN)
# Steinhart-Hart coefficients for Thermistor sensors
A = 0.001129148
B = 0.000234125
```

```
C = 0.0000000876741
# Resistor values, manually calibrated
R0 = 10680
R0_ntc = 87000
# Button debounce
last_press_time = 0
last_test_press_time = 0
# Global variables
temp_offset = 0
sensor_failure_count = {"thermistor": 0, "ntc": 0, "ds18b20": 0}
system_hang = False
# Initialize Watchdog Timer with a 8-second timeout
wdt = WDT(timeout=8000)

# Interrupt method for button press, apply temperature offset with each press
def button_handler(pin):
    global temp_offset, last_press_time
    current_time = time.ticks_ms()
    if time.ticks_diff(current_time, last_press_time) > 300:
        temp_offset += 2
        if temp_offset > 10:
            temp_offset = 0
        print(f"Button pressed, temperature offset: +{temp_offset} C")
        last_press_time = current_time

# Interrupt method for test button to trigger an infinite loop
def test_button_handler(pin):
    global system_hang, last_test_press_time
    current_time = time.ticks_ms()
    if time.ticks_diff(current_time, last_test_press_time) > 300:
        print("Test button pressed. Simulating a system hang.")
        system_hang = True
        last_test_press_time = current_time

# Interrupt instructions for buttons
```

```
button.irq(trigger=Pin.IRQ_RISING, handler=button_handler)
test_button.irq(trigger=Pin.IRQ_RISING, handler=test_button_handler)

# Method that calculates temperature for thermistor sensors
def read_temperature(adc_pin, R):
    adc_value = adc_pin.read_u16()  # Read 16-bit value
    adc_value_12bit = adc_value >> 4  # Scale down to 12-bit
    voltage_measured = adc_value_12bit * 3.3 / 4095  # Convert 12-bit ADC reading to voltage
    resistance = (voltage_measured * R) / (3.3 - voltage_measured)
    logR = math.log(resistance)
    inv_T = A + B * logR + C * (logR ** 3)
    T_kelvin = 1 / inv_T
    T_celsius = T_kelvin - 273.15
    return T_celsius

# Function to read the DS18B20 sensor
def read_ds18b20():
    try:
        roms = ds_sensor.scan()
        if roms:
            ds_sensor.convert_temp()
            time.sleep_ms(750)
            temperature = ds_sensor.read_temp(roms[0])
            return temperature - 2  # Manual calibration offset
        else:
            print("No DS18B20 sensor found!")
            return None
    except Exception as e:
        print(f"Error reading DS18B20: {e}")
        return None

# Function to determine the majority temperature value and detect faulty sensors
def majority_vote(thermistor_temp, ntc_temp, ds18b20_temp):
    global sensor_failure_count
    temps = [thermistor_temp, ntc_temp, ds18b20_temp]
```

```
deviations = [
    abs(thermistor_temp - ntc_temp),
    abs(ntc_temp - ds18b20_temp),
    abs(thermistor_temp - ds18b20_temp)
]
threshold = 2.0

# If all three temperatures are within the threshold, use the average of all
if deviations[0] <= threshold and deviations[1] <= threshold and deviations[2] <= threshold:
    set_led_status(thermistor_green_led, thermistor_red_led, True)
    set_led_status(ntc_green_led, ntc_red_led, True)
    set_led_status(ds18b20_green_led, ds18b20_red_led, True)
    return sum(temps) / 3

# If thermistor and NTC agree, use their average, mark DS18B20 as faulty
elif deviations[0] <= threshold:
    set_led_status(thermistor_green_led, thermistor_red_led, True)
    set_led_status(ntc_green_led, ntc_red_led, True)
    set_led_status(ds18b20_green_led, ds18b20_red_led, False)
    sensor_failure_count["ds18b20"] += 1
    sensor_failure_count["thermistor"] = 0
    sensor_failure_count["ntc"] = 0
    return (thermistor_temp + ntc_temp) / 2

# If NTC and DS18B20 agree, use their average, mark Thermistor as faulty
elif deviations[1] <= threshold:
    set_led_status(thermistor_green_led, thermistor_red_led, False)
    set_led_status(ntc_green_led, ntc_red_led, True)
    set_led_status(ds18b20_green_led, ds18b20_red_led, True)
    sensor_failure_count["ds18b20"] = 0
    sensor_failure_count["thermistor"] += 1
    sensor_failure_count["ntc"] = 0
    return (ntc_temp + ds18b20_temp) / 2

# If Thermistor and DS18B20 agree, use their average, mark NTC as faulty
```

```
    elif deviations[2] <= threshold:
        set_led_status(thermistor_green_led, thermistor_red_led, True)
        set_led_status(ntc_green_led, ntc_red_led, False)
        set_led_status(ds18b20_green_led, ds18b20_red_led, True)
        sensor_failure_count["ds18b20"] = 0
        sensor_failure_count["thermistor"] = 0
        sensor_failure_count["ntc"] += 1
        return (thermistor_temp + ds18b20_temp) / 2

    # If none agree, default to the Thermistor and mark all as faulty
    # + 3 failure count to force system reboot
    else:
        set_led_status(thermistor_green_led, thermistor_red_led, False)
        set_led_status(ntc_green_led, ntc_red_led, False)
        set_led_status(ds18b20_green_led, ds18b20_red_led, False)
        sensor_failure_count["ds18b20"] += 3
        sensor_failure_count["thermistor"] += 3
        sensor_failure_count["ntc"] += 3
        print("All sensors disagree significantly, no reliable reading")
        return 0

# Function to set LED status for sensors
def set_led_status(green_led, red_led, reliable):
    green_led.value(1 if reliable else 0)
    red_led.value(0 if reliable else 1)

# Function to control motor speed and blue LED intensity based on temperature
def control_motor_and_led(temp):
    if temp <= 23:
        pwm_motor.duty_u16(0)
        blue_led.duty_u16(0)
    elif 23 < temp <= 24:
        pwm_motor.duty_u16(32768)
        blue_led.duty_u16(16384)
    elif 24 < temp <= 30:
        pwm_motor.duty_u16(49152)
        blue_led.duty_u16(32768)
    elif temp > 30:
        pwm_motor.duty_u16(65535)
        blue_led.duty_u16(65535)

# Function to read and validate temperatures from all sensors
def read_and_validate_temperatures():
    global system_hang
    if system_hang:
        print("System hang simulated. Stopping temperature validation.")
        while True:
            pass  # Infinite loop to simulate hang
    else:
        thermistor_temp = read_temperature(adc_thermistor, R0)
        ntc_temp = read_temperature(adc_ntc, R0_ntc)
        ds_temp = read_ds18b20()

        if ds_temp is None:
            ds_temp = float('inf')

        majority_temp = majority_vote(thermistor_temp, ntc_temp, ds_temp)
        adjusted_temp = majority_temp + temp_offset
        control_motor_and_led(adjusted_temp)
        display.temperature(round(adjusted_temp))

        print(f"Thermistor Temperature: {thermistor_temp:.2f} C")
        print(f"NTC Sensor Temperature: {ntc_temp:.2f} C")
        if ds_temp != float('inf'):
            print(f"DS18B20 Temperature: {ds_temp:.2f} C")
        else:
            print("DS18B20 Temperature: Error reading sensor")
        print(f"Majority Vote Temp: {adjusted_temp:.2f} C \n")

# System reboot sequence: Yellow LED blinks 5 times
```

```
def reboot_sequence():
    for _ in range(5):
        yellow_led.on()
        time.sleep(0.2)
        yellow_led.off()
        time.sleep(0.2)
    yellow_led.on()

# Check if any sensor has failed three times in a row
def monitoring_task():
    global sensor_failure_count
    while True:
        for sensor, count in sensor_failure_count.items():
            if count >= 3:
                print(f"Sensor {sensor} has failed 3 times. Resetting system.")
                reset()

        time.sleep(2)

# Start monitoring task on separate core
th.start_new_thread(monitoring_task, ())

# Run the reboot sequence
reboot_sequence()

while True:
    wdt.feed()
    read_and_validate_temperatures()
    time.sleep(5)
```