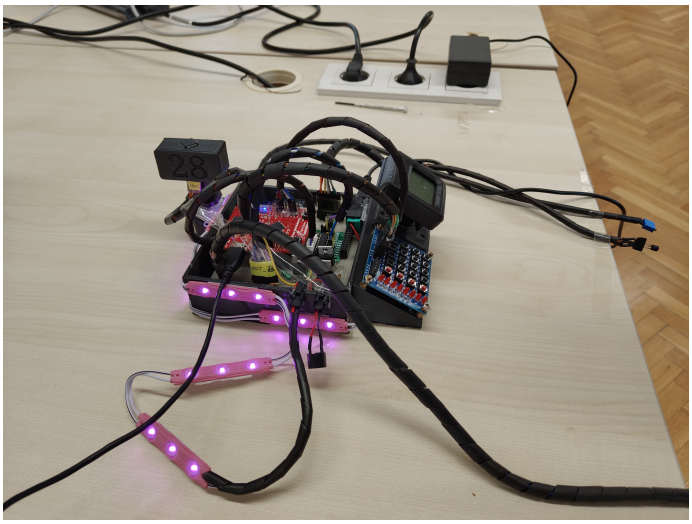
Temperature-Initiated Object Detection

EE447 Term Project 2024-2025



Group Number: 28

Student 1: Onur Karakoç 2423200

Student 2: Bora Sarıkaya 2443752

Middle East Technical University
Department of Electrical and Electronics Engineering

January 15, 2025

Temperature-Initiated Object Detection

***Abstract*—This report presents the implementation and operation of a temperature-initiated object detection system using the TM4C123G microcontroller. The system integrates analog and digital temperature sensors, an ultrasonic distance sensor, and various output modules to fulfill multi-functional tasks efficiently. Key functionalities, pseudo-codes, flowcharts, and pin diagrams are provided to illustrate the design.**

I. Introduction

The project utilizes multiple modules to achieve a robust temperature-initiated object detection system. The LM35 and BMP280 sensors are used for analog and digital temperature measurements, respectively. The HC-SR04 ultrasonic sensor enables object detection within a specified range. A Nokia 5110 LCD screen provides real-time data visualization, while a stepper motor assists in scanning the environment. Additionally, a 4x4 keypad and onboard pushbuttons allow user input for threshold adjustments. RGB LEDs and a power LED indicate system states, and a speaker adds auditory feedback. This combination ensures efficient functionality and user interaction.

II. Methodology

A. *Schematics*

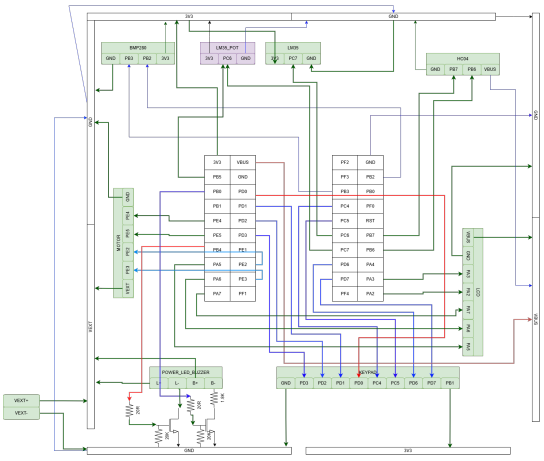


Fig. 1: Pin Diagram of TM4C123G Connections

B. *System Workflow and Flowcharts*

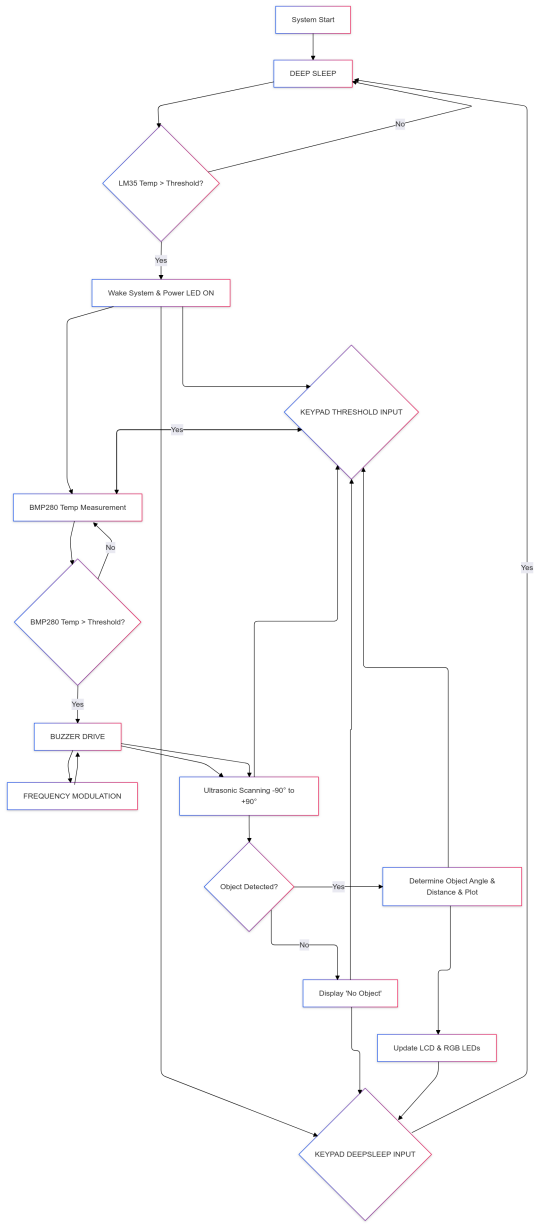


Fig. 2: Flow Chart of the Designed System

C. *Deep Sleep Mode*

Functionality: The system enters a low-power state, monitoring only essential functions.

```
1. Read the LM35 temperature threshold.
2. Clear the Nokia LCD display and display the
    threshold value.
3. Turn off the power LED.
```

```
4. De-initialize peripherals such as:
   - Keypad
   - Radar
   - HC-SR04 ultrasonic sensor
   - BMP280 temperature sensor
   - Onboard LEDs
5. Configure the system for deep sleep mode:
   - Enable deep sleep mode in system control block
       (SCB).
   - Keep only the Analog Comparator clock active.
6. Re-initialize the Analog Comparator for
    monitoring temperature.
7. Enter a loop to wait for an interrupt:
   - If the LM35 temperature rises above the
       threshold, exit the loop.
8. Upon waking up, reinitialize necessary
    peripherals using the WAKE\_UP function.
```

D. Onboard LED

Functionality: Control onboard LEDs for feedback.

```
1. Initialize GPIO pins for onboard LEDs.
2. Configure pins as output and enable digital
    functionality.
3. To turn on an LED:
   - Set the corresponding GPIO pin high.
4. To turn off an LED:
   - Set the corresponding GPIO pin low.
5. To turn off all LEDs:
   - Set all GPIO pins low.
```

E. Buzzer and Timer

Functionality: Control buzzer for give the information of the radar scanning begin.

```
1. Initialize GPIO pin and Timer1 for the buzzer.
2. Use pull down resistors for the power switches
    pin.
3. Configure Timer1 in periodic mode.
4. To activate the buzzer:
   - Map temperature to a timer interval.
   - Set the interval in Timer1.
   - Enable the buzzer by setting the GPIO pin high
       (controlled via an N-type MOSFET).
5. To deactivate the buzzer:
   - Set the GPIO pin low.
   - Disable Timer1 and its interrupts.
```

F. Nokia 5110 LCD

Functionality: Display system status and sensor readings. The Nokia 5110 LCD library by Prof. Valvano has been adapted for this project with custom modifications to enable the plotting of graphs in polar coordinates. The library provides robust SPI-based communication and includes utilities for clearing the display buffer, setting individual pixels, and rendering data on the screen. The key enhancement in this project is the implementation of a custom function to display angle-versus-distance data.

```
1. Clear the display buffer using
    Nokia5110_ClearBuffer().
2. Define the screen boundaries:
   - X-axis: [0, 83]
   - Y-axis: [0, 47] (inverted coordinate system).
3. Draw axes on the screen:
   - Loop through X pixels to draw a horizontal axis
       .
   - Loop through Y pixels to draw a vertical axis.
4. Add tick marks for reference:
   - Place tick marks every 10 pixels on both axes.
5. Plot data points:
   - For each angle and distance pair:
     a. Convert polar coordinates to Cartesian:
        x = distance * cos(angle)
        y = distance * sin(angle)
     b. Map Cartesian coordinates to screen pixels.
     c. Use Nokia5110_SetPxl(x, y) to set the pixel.
6. Render the updated buffer on the screen using
    Nokia5110_DisplayBuffer().
```

G. 4x4 Keypad

Functionality: The 4x4 keypad allows user input for various operations such as threshold adjustments and function triggering.

- Pseudo-code for KEYPAD.h and KEYPAD.c

```
1. Define KEYPAD_Struct:
   - Attributes: key, number, activate,
       deepSleepActivate, nokiaOutput.
   - Function pointers: init(), read(),
       deinit().
2. Initialize GPIO pins:
   - Enable clocks for rows and columns.
   - Set rows as outputs, columns as
       inputs with pull-up resistors.
3. Read keypad input:
   - Activate a row, scan columns for a
       pressed key.
   - If a key is detected, convert it to
       its value and store.
   - Display the pressed key on the
       Nokia LCD.
4. Manage keypad state:
   - Use interrupts to toggle activation
       mode.
   - Implement debounce logic.
5. Deinitialize keypad:
   - Reset GPIO configurations.
   - Disable interrupts.
```

- Pseudo-code for KEYPAD.s

```
1. Initialize GPIO Pins:
   - Set rows as outputs, columns as
       inputs with pull-up resistors.
2. Scan Keypad:
   - Set one row high, others low.
   - Check columns for low signal
       indicating a pressed key.
   - Wait to check if debouncing occurs
   - Check columns again for is a low
       signal indicating a pressed key
   - Repeat for all rows.
3. Decode Keypress:
   - Map the active row and column to a
       key value.
4. Return Key Value:
   - If no key is pressed, return 0xFF.
   - Otherwise, return the detected key
       value.
```

- Pseudo-code for ChangeGraphButton

```
1. Initialize GPIO for the button:
   - Enable clock for the GPIO port.
   - Set the pin as input with pull-up
       resistors.
2. Handle Button Press:
   - Check for a falling edge (button
       press).
   - Apply debounce logic with a delay.
   - Change display state based on the
       button press.
3. Update Display:
   - Case 0: Show thresholds.
   - Case 1: Display temperatures and
       thresholds.
   - Case 2: Plot Graph 1.
   - Case 3: Plot Graph 2 or display "No
        Object" if no data available.
4. Reinitialize Peripherals:
   - Reinitialize BMP280 sensor to
       refresh values.
   \\
```

- Pseudo-code for Decimal Number Entry

```
1. Initialize State Variables:
   - counter for keypress count.
   - dot_pressed flag to handle decimal
       point.
   - number to store the final parsed
       value.
   - fraction_part and
       fraction_multiplier for
       fractional calculations.
2. Keypad Read Loop:
   - Read a key using KEYPAD_Read_ASM().
   - If a numeric key is pressed (0-9):
     - Append it to nokiaOutput.
     - Update integer or fractional part
         based on dot_pressed.
     - Display the current number on the
         Nokia LCD.
   - If the dot key is pressed:
     - Set dot_pressed flag.
     - Update nokiaOutput and LCD
         display.
   - If "Enter" key is pressed:
     - Save the number.
     - Exit the loop.
3. Update BMP280 Threshold:
   - Combine integer and fractional
       parts.
   - Set BMP280 thresholdTemperature.
```

H. I2C Communication

Functionality: The I2C protocol is utilized for communication with peripherals like BMP280.

```
1. Initialize I2C:
   - Enable I2C0 and GPIOB clock.
   - Configure PB2 (SCL) and PB3 (SDA) for alternate
       I2C functionality.
   - Enable I2C Master mode and set clock speed to
       100kHz.

2. Write a Byte:
   - Wait for the I2C bus to be idle.
   - Send slave address and specify the register
       address.
   - Send the data byte to the slave.
   - Issue a stop condition to complete the
       transaction.

3. Read a Byte:
   - Wait for the I2C bus to be idle.
   - Send slave address and register address to read
       from.
   - Issue a repeated start condition.
   - Read the data byte from the slave.
   - Issue a stop condition to complete the
       transaction.

4. Read Multiple Bytes:
   - Send the starting register address to the slave
       .
   - Read bytes sequentially from the slave.
   - Send an ACK after each byte except the last one
       .
   - Send a NACK and issue a stop condition after
       the last byte.
```

I. BMP280 Sensor

Functionality: The BMP280 sensor is used for temperature and pressure measurements. It operates over the I2C protocol and includes functionality for calibration, temperature averaging, and threshold-based operations.

```
1. Initialize BMP280:
   - Initialize I2C communication.
   - Soft reset the sensor.
   - Read calibration data from registers (0x88 to 0
       x9F).
   - Configure control and measurement registers for
       normal mode.

2. Read Temperature:
   - Read raw temperature data (20-bit value across
       3 registers).
   - Apply temperature compensation formulas to
       convert to  C .

3. Read Pressure:
   - Read raw pressure data (20-bit value across 3
       registers).
   - Use the compensated temperature (t_fine) for
       pressure calculation.
   - Apply pressure compensation formulas to
       calculate pressure in Pa.

4. Calculate Temperature Average:
   - Loop for a predefined number of samples (e.g.,
       130).
   - Read raw temperature data and accumulate.
   - Divide the total by the number of samples to
       get the average.
   - Convert to  C .

5. Wait for Temperature Overshoot:
   - Continuously read average temperature.
   - Compare it with the threshold temperature.
   - Allow user input (via keypad) to modify the
       threshold dynamically.

6. Deinitialize BMP280:
   - Put the sensor into sleep mode.
   - Deinitialize I2C to reduce power consumption.
```

J. HCSR04 Sensor

Functionality: The HCSR04 sensor is used for distance measurements using ultrasonic waves. It measures the time delay between triggering an ultrasonic pulse and receiving the echo to calculate the distance.

```
1. Initialize HCSR04:
   - Initialize GPIO for trigger and echo pins.
   - Initialize Timer for edge-time capture mode.
   - Configure timer for interrupt-driven trigger
       toggling.

2. Trigger Ultrasonic Pulse:
   - Toggle trigger pin high for 10  s .
   - Wait for the pulse to complete.

3. Measure Echo Pulse:
   - Wait for rising edge using timer capture mode.
   - Record the timestamp of the rising edge.
   - Wait for falling edge using timer capture mode.
   - Record the timestamp of the falling edge.
   - Calculate pulse width based on timestamps.

4. Calculate Distance:
   - Compute sound speed using the given temperature
       .
   - Use the formula: \(\text{Distance (cm)} = \frac
       {\text{Pulse Width ( s )} \times \text{Sound
       Speed (cm/ s )}}{2}\).

5. Deinitialize HCSR04:
   - Disable GPIO and Timer interrupts.
   - Revert GPIO pins to safe states.
   - Power down the Timer module if no longer needed
       .
```

K. RADAR System

Functionality: The RADAR module controls a stepper motor and utilizes the HC-SR04 ultrasonic sensor for scanning and object detection. Below is the pseudo-code for its implementation.

```
1. Initialize RADAR GPIO:
   - Enable clocks for Port E.
   - Configure PE2, PE3, PE4, and PE5 as outputs for
       the stepper motor.
   - Disable analog functionality and alternate
       functions for these pins.

2. Run Motor:
   - Based on the step number, activate
       corresponding GPIO pins.
   - Deactivate all other pins to achieve stepping
       motion.

3. RADAR Enable:
   - Initialize GPIO for RADAR.
   - For each step (0 to 1024):
     - Wait for the step flag to be set (via SysTick
         ).
     - Call Run Motor to move the stepper motor.
     - Every 5th step (RENDER_RATE):
       - Measure distance using HC-SR04.
       - Record angle and distance.
       - Accumulate valid distances and angles for
           averaging.
     - Reverse the motor to reset its position.
   - Calculate average distance and angle:
     - If an object is detected:
       - Compute average distance and angle.
       - Update LED indicators based on distance
           thresholds.
     - Otherwise, set average distance to -1.
   - Disable RADAR GPIO.

4. SysTick Handler:
   - Increment the millisecond counter.
   - Set the step flag every RENDER_RATE * 2 ms.

5. RADAR Disable:
   - Deactivate all stepper motor GPIO pins.
```

III. Results and Conclusion

A. System Images and Debugging

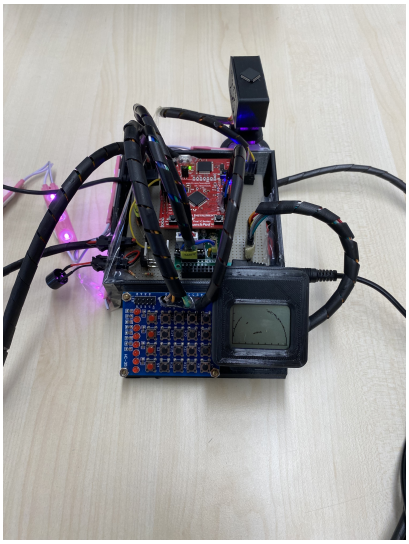


Fig. 3: Polar Coordinates Distance Graph

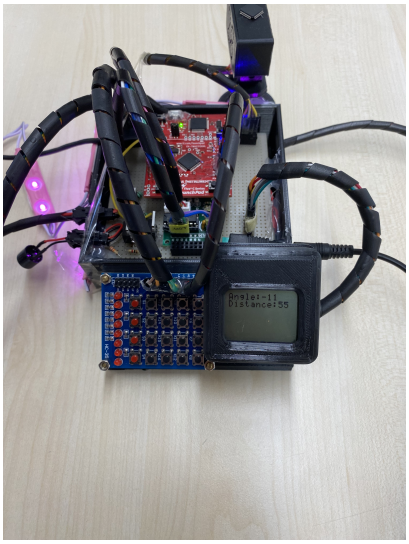


Fig. 4: Average Detected Object Distance and Angle

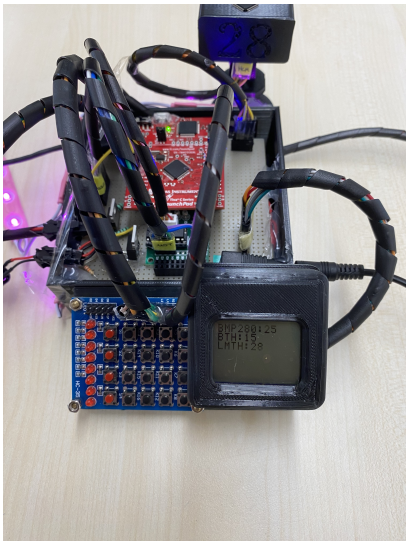


Fig. 5: Real-Time Digital Temperature Readings and Thresholds

During the development and testing of the system, several challenges were encountered and systematically addressed to ensure the proper functioning of the project. These challenges and their corresponding solutions are detailed below:

B. Keypad Input Reading and Decimal Conversion

Problem: Reading decimal and floating-point numbers from the keypad posed challenges. Interrupts could not be used during the scanning process because they would interfere with the sequential detection of button presses. Additionally, multiple button presses were required to construct decimal and floating-point numbers.

Solution: A dedicated button was implemented to toggle interrupt functionality. When this button is pressed, a flag is set, and the program enters a secondary endless loop to allow uninterrupted keypad scanning. Once the flag is reset by another interrupt, the program resumes normal operation. This approach enabled the system to accurately read multiple button presses and convert them into decimal numbers.

C. Keypad Debouncing

Problem: Button debouncing caused false detections during keypad operation, leading to unreliable input readings.

Solution: Minimal delays were introduced to filter out noise and stabilize button detection. By implementing this delay, the program could confirm a valid key press when the button remained pressed consistently.

D. Keypad and HCSR04 Interference

Problem: The keypad and HCSR04 sensor interfered with each other due to a 0-ohm connection between PD0 (keypad pin) and PB6 (HCSR04 trigger pin) on the Tiva board. This resulted in incorrect sensor readings.

Solution: The 0-ohm dismounted to prevent interference between two pins. During testing, however, this oversight led to the corruption of one Tiva board and the debugger of another. To address this, the debugger from one faulty microcontroller was exchanged with another functioning unit, restoring operability. One can also see the faulted M4 microprocessor on top of the HCSR04 ultrasonic distance sensor.

E. Memory Constraints and `math.h` Library Usage

Problem: The inclusion of the `math.h` library for converting Cartesian coordinates to polar coordinates resulted in the compiled program size exceeding the 32 KB limit imposed by the Keil software under the -O0 compiler optimization level. This issue rendered the program incompatible with the available memory constraints.

Solution: To address this limitation, a lookup table was implemented to replace runtime computations for trigonometric functions. The table was populated with pre-calculated values for cosine and sine functions over integer angles ranging from -180° to 180°. The arrays were defined as follows:

$$\text{cos\_values}[i] = \sum_{j=-180}^{180} \cos(j), \quad \text{sin\_values}[i] = \sum_{j=-180}^{180} \sin(j).$$

This approach significantly reduced memory usage by eliminating the need for real-time trigonometric computations while maintaining the required accuracy for the operations.

F. Noise in LM35 Temperature Sensor

Problem: The LM35 analog temperature sensor was highly susceptible to noise. Physical contact with its cables or the sensor itself introduced resistance and voltage fluctuations, causing the microcontroller to erroneously wake up from deep sleep.

Solution: The connections were re-soldered and secured on a perfboard to improve stability. Additionally, the cables were separated and insulated using composite materials (e.g., heat shrink tubing) to prevent noise interference and ensure reliable sensor operation.

Conclusion

The resolution of these issues significantly improved the system's reliability and performance. The implemented solutions not only addressed the immediate problems but also enhanced the robustness of the design for future applications. This iterative process underscores the importance of systematic debugging and creative problem-solving in embedded systems development.

IV. NOTES

This project will be made open-source following the completion of the grading process at Middle East Technical University. The GitHub repository link is provided below. All project-related materials, including source codes, 3D models, and other necessary resources, will be available in the repository.

https://github.com/onurkarakoc79/
EE447-Temperature-Initiated-Object-Detection

REFERENCES

[1] Texas Instruments, "Tiva TM4C123GH6PM Microcontroller Data Sheet," Available: http://www.ti.com/lit/ds/spms376e/spms376e.pdf.

[2] Texas Instruments, "Tiva C Series TM4C123G LaunchPad Evaluation Board User's Guide," Available: https://www.ti.com/lit/ug/spmu296/spmu296.pdf.

[3] Department of Electrical and Electronics Engineering, Middle East Technical University, "EE447 Term Project 2024-25: Temperature-Initiated Object Detection," Available: https://github.com/onurkarakoc79/EE447-Temperature-Initiated-Object-Detection/blob/main/Project%20Documents/ee447_Term_Project_2024_25%20(1).pdf.