

HOME APPLICATION CONTROL

Introduction to embedded systems project



December 25, 2024

**Team members**

|  |  |
| --- | --- |
| Malak Mohamed Salem | 21P0277 |
| Mariam Mahmoud Abdou | 21P0214 |
| Siveen Said Sayed | 21P0149 |
| Hager Hesham Mohamed | 21P0297 |
| Hamsa Ahmad Abdlmgeed | 20P1874 |

# **Project Overview**

The Home Appliances Control System is a comprehensive solution that enables users to monitor and control various home appliances through a mobile application. The system implements a layered architecture approach, consisting of three main layers:

## 1. Application Layer (main)

- Android mobile application providing user interface for control and monitoring

- Handles user interactions and displays system status

- Implements temperature alarm logic and door status logging

- Manages Bluetooth communication with the microcontroller

## 2. Hardware Abstraction Layer (HAL)

(magnetic\_switch.c/h, temp.c/h, lamp\_plug.c/h, bluetooth\_module.c/h)

- Provides abstract interface between hardware and middleware

- Implements device drivers for:

- Lamp and plug switches

- Temperature sensor

- Magnetic door switch

- Bluetooth module

- Physical alarm

- Handles hardware initialization and configuration

## 3. Middleware/Mechanical Layer

( tm4c123gh6pm.h, DIO.c/h, adc.c/h, uart.c/h , Systick.c/h, utils.h )

- Implements control logic for appliance switching

- Initializes ports and pins

- Manages state transitions and system timing

# **System Architecture Overview**

The system utilizes a TM4C123GH6PM microcontroller as its core processing unit, interfacing with various peripheral devices and sensors.

# **Hardware Implementation**

## Core Components:

### 1. Processing Unit

- TM4C123GH6PM microcontroller

- Operating at 16MHz

- Utilizing GPIO, ADC, and UART peripherals

### 2. Power Management

- 5V DC power supply for microcontroller and sensors

- 220V AC handling through isolated relay modules

- Voltage regulation and protection circuits

### 3. Sensing Elements

- Temperature sensor for room monitoring

The LM35 (temperature sensor) is a precision temperature sensor that provides an analog output proportional to the temperature in °C. The sensor has three pins: VCC, GND, and OUTPUT.

* VCC (Power): The VCC pin of the LM35 is connected to the 3.3V pin of the Tiva C microcontroller to supply power to the sensor.
* GND (Ground): The GND pin of the LM35 is connected to the GND pin of the Tiva C to complete the electrical circuit.
* OUTPUT (Analog Signal): The OUTPUT pin of the LM35 is connected to the PE3 pin of the Tiva C microcontroller. This pin will read the analog voltage signal, which corresponds to the temperature.

The buzzer is used to provide audible alerts based on temperature threshold. It is connected to the Tiva C microcontroller as follows:

* Ground Connection: One terminal of the buzzer is connected directly to the GND pin of the Tiva C microcontroller.
* Signal Connection: The other terminal of the buzzer is connected to PE1 of the Tiva C through a 1 kΩ resistor. This resistor limits the current flowing through the buzzer, protecting both the buzzer and the microcontroller.

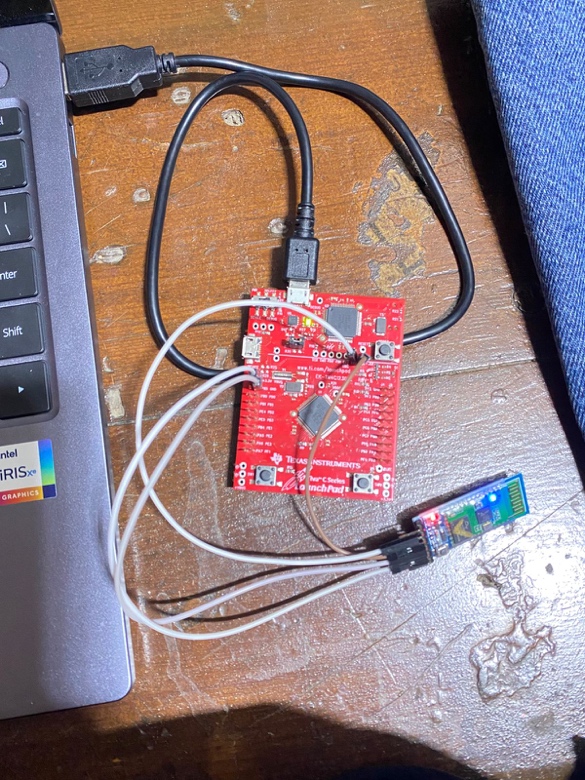
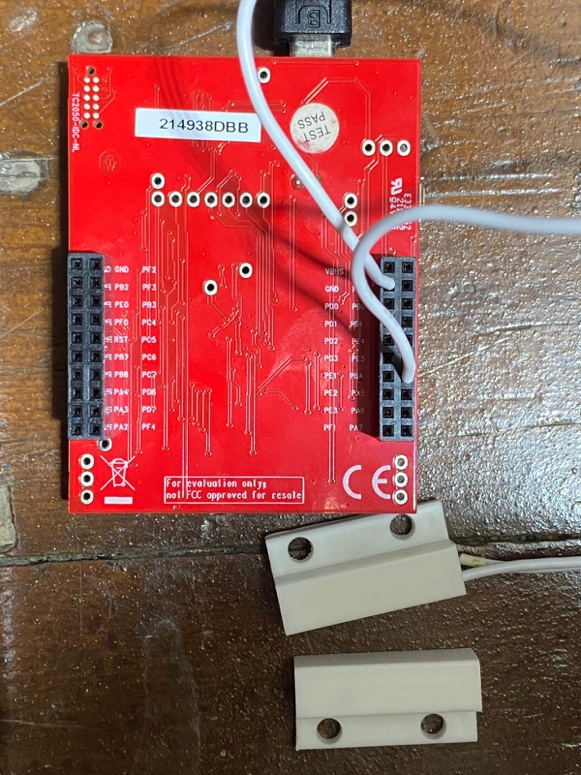
- Magnetic reed switch for door status detection

### 4. Control Elements

- Switches for lamp and plug control

### 5. Communication

- Bluetooth module for mobile app connectivity



# **Software Architecture**

## Android Application Structure

- User interface for appliance control

- Real-time temperature monitoring

- Door status logging system

- Bluetooth communication handler

- Alarm for high temperature detection

- Switches for controlling lamp and plug remotely.

## Microcontroller Firmware

- Main control loop

- Interrupt handlers

- Serial communication implementation

# **Software Implementation Details**

## Driver Layer Implementation

### 1. Utility file (utils.h)

The system implements fundamental bitwise operations through macro definitions, These operations form the foundation for all hardware interactions in the system.

- **SET\_BIT:** Sets a specific bit in a register

**- CLEAR\_BIT:** Clears a specific bit in a register

**- TOGGLE\_BIT:** Toggles a specific bit in a register

**- GET\_BIT:** Reads the value of a specific bit

**- WRITE\_BYTE:** Writes a complete byte to a register

#ifndef BITWISE\_OPERATIONS\_H

#define BITWISE\_OPERATIONS\_H

#define SET\_BIT(REG, BIT) ((REG) |= (1U << (BIT)))

#define CLEAR\_BIT(REG, BIT) ((REG) &= ~(1U << (BIT)))

#define TOGGLE\_BIT(REG, BIT) ((REG) ^= (1U << (BIT)))

#define GET\_BIT(REG, BIT) (((REG) >> (BIT)) & 1U)

#define WRITE\_BYTE(REG, VALUE) ((REG) = VALUE))

#endif // BITWISE\_OPERATIONS\_H

## 2. Digital I/O Module (DIO)

The DIO module provides a hardware abstraction layer for **GPIO** operations:

**Key Features:**

- Port initialization and configuration

- Pin direction control (INPUT/OUTPUT)

- Digital/Analog mode selection

- Pull-up/Pull-down configuration

- Port and pin-level read/write operations

**Implementation Highlights:**

#include "DIO.h"

/\*\*

\* Function to get the base address of the specified port.

\* port: The port identifier (e.g., PORT\_A, PORT\_B, etc.).

\* return: The base address of the GPIO port, or 0xFF if the port is invalid.

\*/

uint32\_t get\_port\_base(uint8\_t port) {

switch (port) {

case PORT\_A: return GPIO\_PORTA\_BASE;

case PORT\_B: return GPIO\_PORTB\_BASE;

case PORT\_C: return GPIO\_PORTC\_BASE;

case PORT\_D: return GPIO\_PORTD\_BASE;

case PORT\_E: return GPIO\_PORTE\_BASE;

case PORT\_F: return GPIO\_PORTF\_BASE;

default: return 0xFF; // Invalid port identifier

}

}

/\*\*

\* Function to initialize a GPIO pin.

\* port: The port identifier (e.g., PORT\_A).

\* pin: The pin number within the port (0-7).

\* direction: The pin direction (1 for output, 0 for input).

\* mode: The mode of the pin (1 for digital, 0 for analog).

\*/

void dio\_init(uint8\_t port, uint8\_t pin, uint8\_t direction, uint8\_t mode) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return; // Exit if the port is invalid

// Enable the clock for the specified port

SET\_BIT(SYSCTL\_RCGCGPIO\_R, port);

while (GET\_BIT(SYSCTL\_PRGPIO\_R, port) == 0); // Wait until the port is ready

// Unlock mechanism for locked pins

GPIO\_PORT\_LOCK\_R(port\_base) = GPIO\_LOCK\_KEY; // Unlock the port using the key

SET\_BIT(GPIO\_PORT\_CR\_R(port\_base), pin); // Allow changes to the pin configuration

// Set the pin direction (input or output)

if (direction)

SET\_BIT(GPIO\_PORT\_DIR\_R(port\_base), pin); // Configure as output

else

CLEAR\_BIT(GPIO\_PORT\_DIR\_R(port\_base), pin); // Configure as input

// Configure the pin mode (digital or analog)

if (mode)

SET\_BIT(GPIO\_PORT\_DEN\_R(port\_base), pin); // Enable digital functionality

else

CLEAR\_BIT(GPIO\_PORT\_DEN\_R(port\_base), pin); // Enable analog functionality

}

/\*\*

\* Function to configure the internal pull-up resistor for a pin.

\* port: The port identifier (e.g., PORT\_A).

\* pin: The pin number within the port (0-7).

\* pull: Enable or disable pull-up (1 to enable, 0 to disable).

\*/

void dio\_pull(uint8\_t port, uint8\_t pin, uint8\_t pull) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return; // Exit if the port is invalid

if (pull)

SET\_BIT(GPIO\_PORT\_PUR\_R(port\_base), pin); // Enable pull-up resistor

else

CLEAR\_BIT(GPIO\_PORT\_PUR\_R(port\_base), pin); // Disable pull-up resistor

}

/\*\*

\* Function to read the value of a specific pin.

\* port: The port identifier (e.g., PORT\_A).

\* pin: The pin number within the port (0-7).

\* return: The value of the pin (1 for high, 0 for low, or 0xFF if invalid).

\*/

uint8\_t dio\_readpin(uint8\_t port, uint8\_t pin) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return 0xFF; // Return 0xFF for invalid port

return GET\_BIT(GPIO\_PORT\_DATA\_R(port\_base), pin); // Read and return pin value

}

/\*\*

\* Function to read the value of an entire port.

\* port: The port identifier (e.g., PORT\_A).

\* return: The value of the port (8-bit), or 0xFF if invalid.

\*/

uint8\_t dio\_readport(uint8\_t port) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return 0xFF; // Return 0xFF for invalid port

return (uint8\_t)GPIO\_PORT\_DATA\_R(port\_base); // Read and return port value

}

/\*\*

\* Function to write a value to a specific pin.

\* port: The port identifier (e.g., PORT\_A).

\* pin: The pin number within the port (0-7).

\* value: The value to write (1 for high, 0 for low).

\*/

void dio\_writepin(uint8\_t port, uint8\_t pin, uint8\_t value) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return; // Exit if the port is invalid

if (value)

SET\_BIT(GPIO\_PORT\_DATA\_R(port\_base), pin); // Set pin high

else

CLEAR\_BIT(GPIO\_PORT\_DATA\_R(port\_base), pin); // Set pin low

}

/\*\*

\* Function to write a value to an entire port.

\* port: The port identifier (e.g., PORT\_A).

\* value: The 8-bit value to write to the port.

\*/

void dio\_writeport(uint8\_t port, uint8\_t value) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return; // Exit if the port is invalid

GPIO\_PORT\_DATA\_R(port\_base) = value; // Write the value to the entire port

}

## Hardware Abstraction Layer (HAL)

### Magnetic Switch Implementation:

#include "magnetic\_switch.h"

// Pointer to the interrupt callback function

static void (\*interrupt\_callback)(void) = 0; // Callback function for the interrupt

// Global variables to store the port and pin configuration

static uint8\_t g\_port;

static uint8\_t g\_pin;

/\*\*

\* Initialize the magnetic switch with the given port and pin.

\* Configures the pin as an input with a pull-up resistor and sets up the interrupt.

\* port: The GPIO port identifier (PORT\_B).

\* pin: The pin number within the port (4).

\* callback: Pointer to the callback function to execute on interrupt.

\*/

void magnetic\_switch\_init(uint8\_t port, uint8\_t pin, void (\*callback)(void)) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return; // Exit if the port is invalid

// Initialize the pin as input with pull-up resistor

dio\_init(port, pin, IN, DIGITAL);

dio\_pull(port, pin, UP);

// Store global interrupt configuration

g\_port = port;

g\_pin = pin;

// Configure GPIO interrupt settings

CLEAR\_BIT(GPIO\_PORT\_IM\_R(port\_base), pin); // Disable interrupt for the pin during setup

CLEAR\_BIT(GPIO\_PORT\_IS\_R(port\_base), pin); // Set to edge-sensitive interrupt

SET\_BIT(GPIO\_PORT\_IBE\_R(port\_base), pin); // Enable interrupt on both edges (rising and falling)

SET\_BIT(GPIO\_PORT\_ICR\_R(port\_base), pin); // Clear any prior interrupt flags

SET\_BIT(GPIO\_PORT\_IM\_R(port\_base), pin); // Enable interrupt for the pin

// Enable the corresponding NVIC interrupt for the port

switch (port) {

case PORT\_A: SET\_BIT(NVIC\_EN0\_R, 0); break;

case PORT\_B: SET\_BIT(NVIC\_EN0\_R, 1); break;

case PORT\_C: SET\_BIT(NVIC\_EN0\_R, 2); break;

case PORT\_D: SET\_BIT(NVIC\_EN0\_R, 3); break;

case PORT\_E: SET\_BIT(NVIC\_EN0\_R, 4); break;

case PORT\_F: SET\_BIT(NVIC\_EN0\_R, 30); break;

}

// Assign the user-defined callback function

interrupt\_callback = callback;

}

/\*\*

\* Get the current state of the magnetic switch.

\* port: The GPIO port identifier (e.g., PORT\_A).

\* pin: The pin number within the port (0-7).

\*return: The current state of the pin (1 for high, 0 for low, or 0xFF if invalid).

\*/

uint8\_t get\_magnetic\_switch\_state(uint8\_t port, uint8\_t pin) {

uint32\_t port\_base = get\_port\_base(port);

if (port\_base == 0xFF) return 0xFF; // Return 0xFF for invalid port

return dio\_readpin(port, pin); // Return the current state of the pin

}

/\*\*

\* Interrupt Service Routine (ISR) for the magnetic switch.

\* Handles the GPIO interrupt, clears the interrupt flag, and calls the user-defined callback function.

\*/

void magnetic\_switch\_handler(void) {

// Clear the interrupt flag for the specific pin

uint32\_t port\_base = get\_port\_base(g\_port);

if (port\_base == 0xFF) return; // Exit if port\_base is invalid

SET\_BIT(GPIO\_PORT\_ICR\_R(port\_base), g\_pin); // Clear the interrupt flag

// Execute the callback function if defined

if (interrupt\_callback != 0) {

interrupt\_callback(); // Call the user-defined interrupt handler

}

}

**Key Features:**

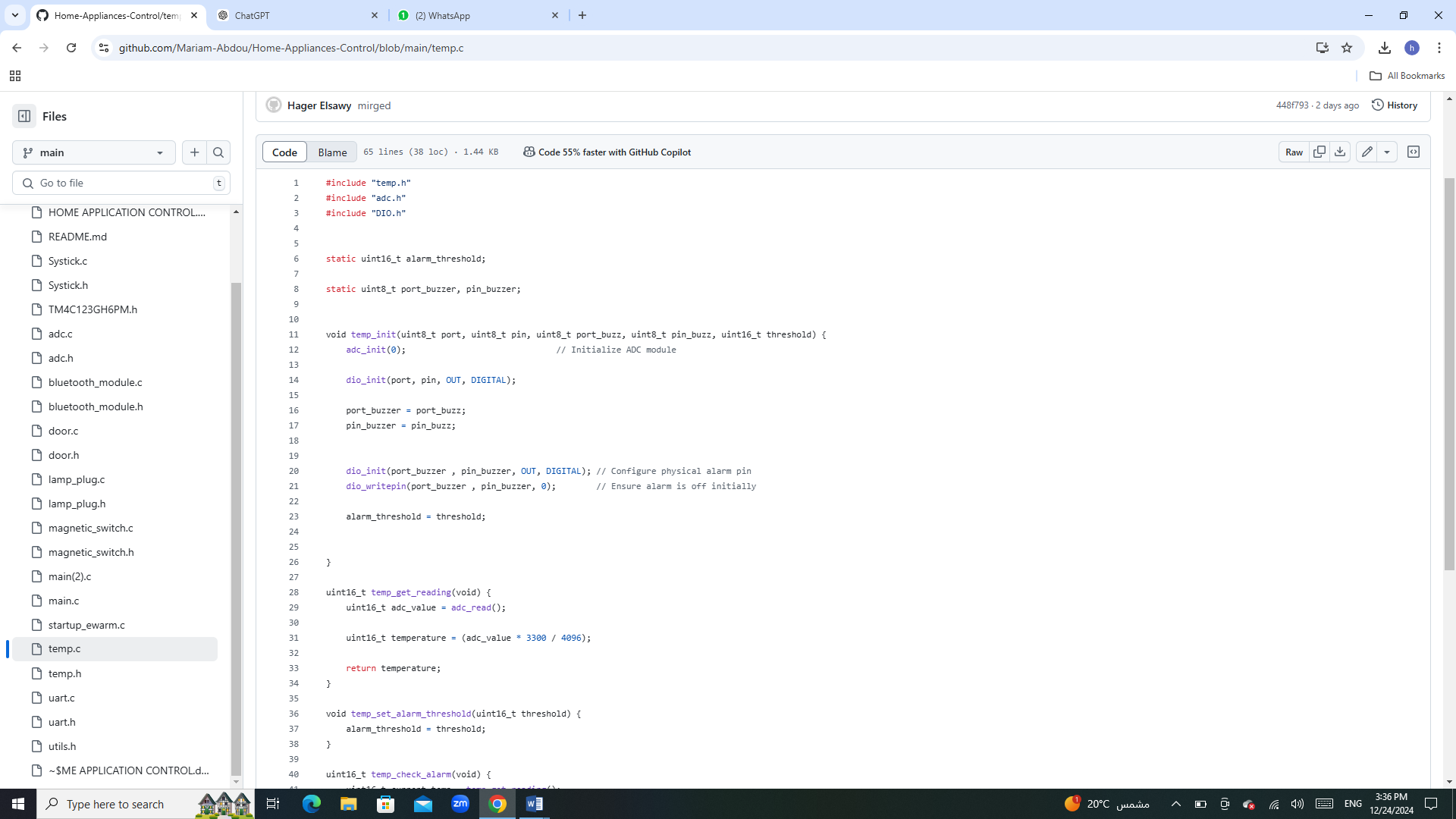
- Interrupt-driven door state detection

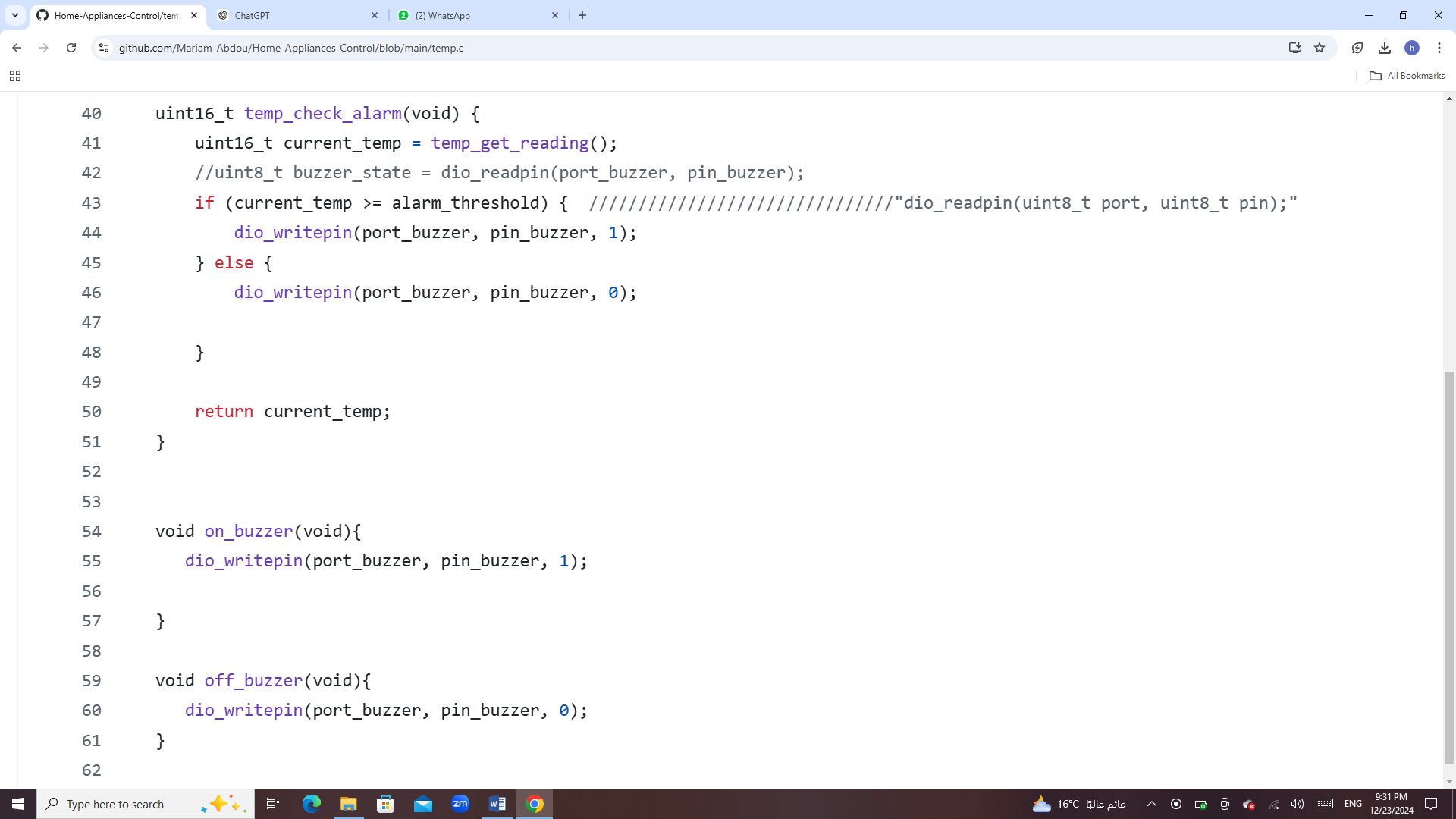
- Callback mechanism for state changes

- Built-in debouncing through hardware configuration

- Real-time status reporting through Bluetooth

### Temperature and Buzzer Implementation:





**Key Features:**

- The temp\_get\_reading function reads the ADC (Analog-to-Digital Converter) value, converts it to a temperature value using a scaling factor for a 12-bit ADC (0–4096 range), and returns the temperature.

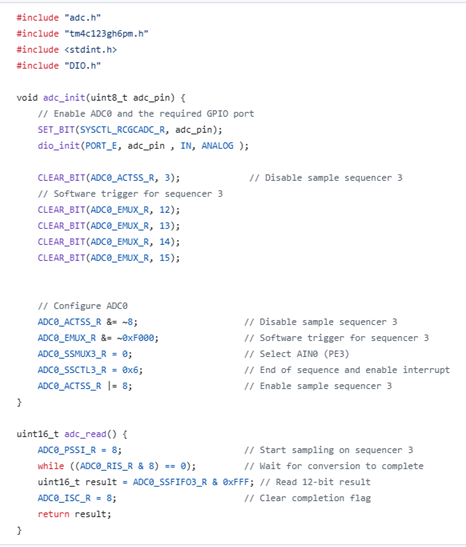
- The temp\_set\_alarm\_threshold function allows dynamic configuration of the alarm temperature threshold.

- The temp\_init function initializes the temperature module, ADC, and a buzzer output pin. This setup includes ensuring the alarm buzzer is off at startup.

- Handles pin assignments for ADC input and buzzer output to ensure modular and reusable code for different hardware configurations.

## Hardware Abstraction Layer (HAL)

### ADC Implementation:



**Key Features:**

- Configures the GPIO pin as an analog input.

- Sets up ADC sequencer 3 for single-sample conversion using a software trigger.

- Starts a conversion on sequencer 3.

- Waits for the conversion to complete and returns the 12-bit result.