PROFILE

1. Introduction

- **Name**: Tirumala Rama Kiran ELUGUBANTI.
- **Current Role**: Software Engineer BMS in Volocopter GmbH, Bruchsal.
- **Education**: Bachelor of Technology in Electronics and Communication Engineering, Acharya Nagarjuna University, INDIA.

2. Professional Summary

- **Experience**: Worked for below organizations as Embedded Software Engineer.
  - Bosch Global Software, India
  - ABB India Limited, India
  - Black Pepper Technologies, India
  - Rockwell Automation Southeast Asia private Limited, Singapore
  - AIRBUS Group India Private Limited (sub-contractor, HCL Technologies)
  - Volocopter GmbH, Bruchsal, Germany (current employer)
- **Key Skills**: Embedded Systems (18 years of experience) | C (18) | C++ (10) | Python (3) | Firmware (10) | Linux (8) | Windows (15) | Object Oriented Programming OOP (10) | Design Patterns (6) | Algorithms (10) | gdb (11) | Perl (1) | XML (8) | Jenkins (2) | SDLC (15) | Code Coverage & Code Review (10) | IPC (8) | SPI (8) | Clearcase (9) | Source Insight (4) | DO-178C (5) | HLR (15) | LLR (15) | DOORS (2) | RTRT (4) | Canalyzer (4) | TAXI (4) | MATLAB/Simulink (3) | MQTT (3) | Unix (10) | STOOD (3) | MobaXterm (3) | Eclipse (8) | Marcel (3) | Impact (3) | Analog Devices BF526 (1) | Inforce (2) | CC2500 radio (2) | VxWorks (3) | Power Electronics Controller (PEC) (4) | UDP (4) | TCP/IP (4) | Modbus TCP (2) | PLC programming (4) | Ethernet (4) | Micorcontrollers, ARM Cortex-M, Atmel AVR e.g. (10) | RTOS (10) | CAN, I2C, Uart, Rs232, RS485, SPI (8) | CodeWright (8) | SDOM (6) | UDE (4) | INCA (5) | ASCET (6) | Git (5) | Jira (6) | Agile (8)

3. Professional Experience

- **Project Highlights**: Worked on various projects in Automotive, Industrial automation, medical, aerospace domains.
  - Automotive component design and development – Diesel particulate filter, Accelerator pedal, Dosing valves.
  - Industrial automation PLCs like PEC800 communicating with slave devices like Profinet, Wago using TCP/IP and UDP.
  - HiSIB (Hemo immune Signal-input-box) development capturing patient parameters like ECG, SpO2, Invasive blood pressure.
  - Flight warning system of A350 aircraft composing messages and display on cockpit monitor.
  - German eVTOL Volocity Battery Management system (BMS) software development.

4. Strengths

- Strong leadership and team-building skills, with managing teams of 5-10 people.

5. Motivation

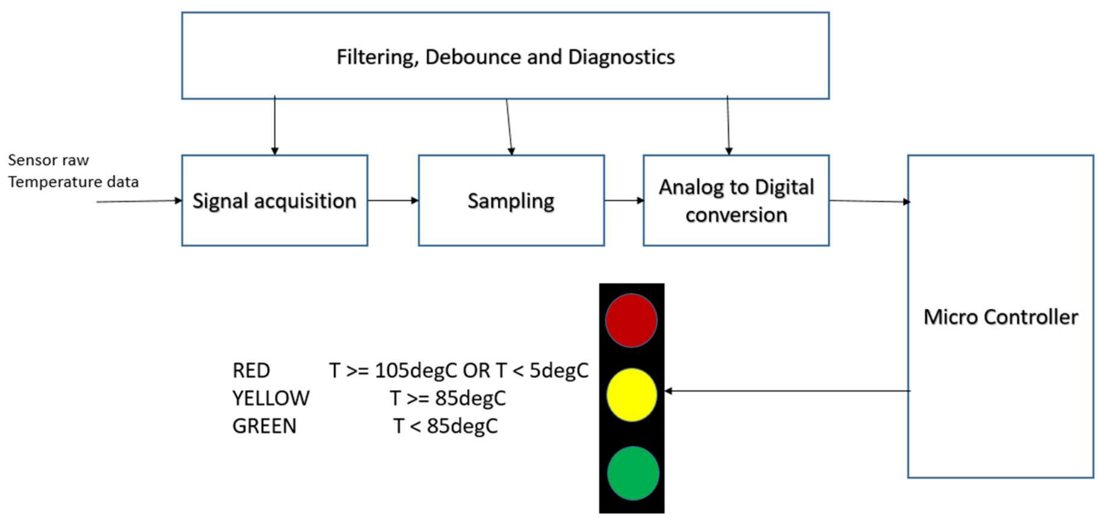
- I am excited about this role because it combines my passion for embedded system design and development, allowing me to drive results while ensuring alignment with the company's broader goals.
- I admire company's commitment to innovation and sustainability, and I am eager to contribute to the mission.

6. Future Goals

- In the next years, I aim to take on a Senior Engineer role in the project, contributing to the organization's overall growth and success while continuing to learn from experts.

TEMPERATURE MONITOR

Block Diagram



Temperature Monitoring strategy uses signal acquisition from the sensor. The raw data collected is filtered, debounced for certain period of time. Then checked for errors like Short circuit to Battery(SCB), Short circuit to ground(SCG), Open load(OL) and Over Temperature(OT) errors using Diagnostics System Management(DSM). Periodic sampling is done based on Nyquist's rate, which is more than twice of the input highest signal frequency. Sampled data is converted to Digital form using Analog-to-Digital conversion methods.

The microcontroller recognizes the temperature values from the data output connected to ADC converter. Based on the temperature values, the corresponding GPIO pins are activated for RED, YELLOW and GREEN LEDs.

<u>Implementation in C language:</u>

<u>Assumptions:</u>

1. ready_count and not_ready_count were added in get_port() API to mimic the delay from the multiplexers to select the data.
2. write_data() and read_data() APIs were implemented with integer variables as port address instead of pointers to ease assignments.

Simulation of Temperature monitoring is implemented in two sub-modules.

- ADC.c
- TempMon.c

<u>ADC.c</u>

This C code simulates and interact with an analog-to-digital converter (ADC) system, for embedded hardware or a device interface.

1. Header files and global variables

```
#include <stdio.h>
#include <malloc.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
```

These headers provide standard C libraries for input/output, memory allocation, time, string manipulation, and boolean data types.

```
unsigned eoc;
extern unsigned base;
int AD_Enabled = 0;
bool FilterDebounceDiagnosticsDone = false;
int CurrentChannel = 0;
unsigned int base = 0x2000;
```

- `eoc`: Represents the "End of Conversion" (EOC) register address.
- `base`: The base memory address for the ADC hardware.
- `AD_Enabled`: A flag indicating if the ADC is enabled.
- `FilterDebounceDiagnosticsDone`: A flag indicating if filtering, debounce and diagnostics are completed.
- `CurrentChannel`: Tracks the current ADC channel being read.

2. Analog channel structure and state definitions

```
struct anachan {
  int data;
  int status;
} AnalogChannel [8];
```

- This defines an array of structures (`AnalogChannel`) to hold data and status for each of the 8 analog channels.

```
enum anastat {
  INACTIVE,
  START_CONVERSION,
  DATA_READY
};
```

- An enum `anastat` is defined to represent the different states of an analog channel:
  - `INACTIVE`: The channel is not active.
  - `START_CONVERSION`: The conversion process has started.
  - `DATA_READY`: The conversion is complete, and the data is available.

3. Functions

a. `write_data` and `read_data` functions

```
void write_data (int x, int data) {
    x = data;
}

int read_data (int x) {
    return x;
}
```

- `write_data`: Writes data to the specified port (simplified here to just assign `data` to `x`).
- `read_data`: Returns data from the specified port (simplified here to just return data).

b. `get_port`

```
/* find hardware port if one exists */
unsigned get_port(void)
{
  int x;
  static unsigned local_port;
  int portAddress;
  unsigned int base, not_ready_count, ready_count;

  if(local_port == 32767)
    return 0;

  for(x=0x200; x<0x3c0; x+=0x40)    {
      not_ready_count = 32767;
      ready_count = 32767;

      /* start conversion */
      write_data(x,0);
      while((read_data(x+0x18) & 0x80) && --not_ready_count); /* wait for
not ready */
      while(!(read_data(x+0x18) & 0x80) && --ready_count); /* wait for
ready */
      if(ready_count < 32767 && ready_count > 0)
    {
      local_port = base = x;
      portAddress = local_port + 0x18;

      return portAddress;
```

```
    }
  }
}
```

- This function finds a hardware port by checking the ports in a range (`0x200` to `0x3c0`) and waiting for a response from each. If it finds a valid port, it returns the port address. “Ready count”, “Not ready count” and mask 0x80 were simulated to depict the delay occurred in selection of the port based on multiplexer connected to the channel and other latency time in fetching port address.

c. `InitializeAnalog`

```
unsigned InitializeAnalog(void) {
  base = get_port();
  eoc = base + 0x18;
  if(!base)
    return 0;
  memset(&AnalogChannel, 0, sizeof(AnalogChannel));
  CurrentChannel = 0;
  return base;
}
```

- This function initializes the ADC by getting the hardware port, setting the base address, and clearing the `AnalogChannel` structure array if garbage value or previous value persists.

d. `TurnOnAnalog` and `TurnOffAnalog`

```
int TurnOnAnalog(int channel) {
  if (channel < 0 || channel > 7)
    return -1;
  AnalogChannel[channel].status = START_CONVERSION;
  AD_Enabled = 1;
  return channel;
}

int TurnOffAnalog(int channel) {
  if (channel < 0 || channel > 7)
    return -1;
  AnalogChannel[channel].status = INACTIVE;
  return AnalogChannel[channel].data;
}
```

- `TurnOnAnalog`: Sets a channel to the `START_CONVERSION` state and enables the ADC.
- `TurnOffAnalog`: Sets the channel state to `INACTIVE` and returns the last fetched data.

e. `new_timer`

```
void new_timer(int timer_counter)
{
  int x;

  /* Is ADC channel enabled? */
  if(AD_Enabled)
  {
    /* look for start conversion or data ready status */
    while(AnalogChannel[CurrentChannel].status != INACTIVE
              && (timer_counter != 0))
    {
       /* Fetch data from channel with DATA_READY status  */
```

```
        switch(AnalogChannel[CurrentChannel].status)
        {
          case START_CONVERSION:
                /* will be ready at next interrupt */
                AnalogChannel[CurrentChannel].status = DATA_READY;
          break;

          case DATA_READY:
           /* check eoc even though it's probably already ready */
           while(!(read_data(eoc) & 0x80));
            /* load data into structure */
           AnalogChannel[CurrentChannel].data = read_data(base);

           /* set up for next */
           AnalogChannel[CurrentChannel].status = START_CONVERSION;
                break;
          case INACTIVE:
                break;
        } /* end switch(AnnalogChannel[CurrentChannel].status) */

        /* check the next channel for data */
        CurrentChannel++;

        /* decrement counter */
        timer_counter--;
      } /* end while loop */
    } /* end if(AD_Enabled) */
  }
```

- This function simulates a timer, checking for data ready on analog channels and reading data when it's available. It works by iterating over channels and updating their status based on the timer.

f. SampleData

```
void SampleData(void)
{
  int x;
  new_timer(100.0);

  x = (int)InitializeAnalog();

  printf("init ana = %X\n",x);

  x = TurnOnAnalog(CurrentChannel);
  printf("TurnOnAnalog(CurrentChannel); = %d\n",x);

  x = GetChannelData(CurrentChannel);
  printf("%4d",x);

  /* Perform filtering, debounce and Diagnostics on raw data */
  FilterDebounceDiagnosticsDone = true;
}
```

- This function calls `new_timer` to simulate ADC sampling and data collection. It also initializes the analog system, turns on the current channel, and performs filtering, debounce, and diagnostics.

g. ADC_Raw

```
char* ADC_Raw()
{
  unsigned ADC_Chan0,dac1,eoc;
  int count;
  char data[300];

  ADC_Chan0 = get_port();

  if(ADC_Chan0 == 0)
  {
    printf("no hardware found\n");
    return 0;
  }

  SampleData();

  dac1 = ADC_Chan0 + 8;
  eoc = ADC_Chan0 + 0x18;

  printf("ADC Channel 0 after get_port = %X\n",ADC_Chan0);

  for(count=0; count<300; count++)
  {
        /* wait for ready and collect data */
        while(!(read_data(eoc) & 0x80))
          data[count] = (char)read_data(dac1);
  }

    return data;
}
```

- This function collects raw data from an ADC channel by initiating sampling data, read data, and then storing the results in a `data` array.

h. ADC_Output

```
int ADC_Output() {
  if (FilterDebounceDiagnosticsDone == true) {
    if (ADC_Raw()!= '\0')
      return 0;
    else
      return ADC_Raw();
  }
}
```

- This function is responsible for processing the output of the ADC. If filtering, debounce and diagnostics are done, it attempts to get raw data from the ADC and returns converted digital data.

Summary

ADC.c code simulates and manage interactions with an ADC system. It allows to:

1. Initialize the ADC system and configure channels.
2. Start and stop conversions for each analog channel.
3. Collect raw data from the ADC.
4. Perform diagnostic and filtering operations on the data.

5. Handle the timing of ADC conversions with a custom `new_timer` function.

<u>TempMon.c</u>

Constants and Definitions:

1. **LED Pin Definitions**:
    - `RED_LED_PIN`, `YELLOW_LED_PIN`, `GREEN_LED_PIN`: These are the pin addresses associated with the three LEDs. They are represented using hexadecimal values.
2. **Hardware Serial Number**:
    - `HARDWARE_SERIAL_NUMBER`: Defines the serial number of the hardware.
3. **HIGH and LOW**:
    - These macros are used to represent high and low states (1 and 0), typically for controlling GPIO pins (high means turning the pin on, low means turning it off).

Enumerations and Variables:

- `hardware_revision`: An enum to track the hardware revision type, either `REV_A` (revision A), `REV_B` (revision B), or `NONE` if no revision is detected.
- `temperature` and `adcOutput`: `temperature` is the fetched temperature value, and `adcOutput` stores the raw ADC value.

Functions:

1. **`GPIO_Write (int PIN, int status)`**:
    - This function simulates the writing of a high or low value to a GPIO pin.

```
void GPIO_Write (int PIN, int status)
{
        /* Write data into GPIO port PIN */
        PIN = &status;
}
```

2. **`GPIO_Init (int PIN)`**:
    - This function simulates the initialization of a GPIO pin by setting its value to 0 (LOW).

```
void GPIO_Init (int PIN)
{
        /* Initialize 0 to the address PIN */
        int data = 0;
        PIN = &data;
}
```

3. **`ADC_Init ()`**:
    - Initializes the ADC system. Here, it just sets `adcOutput` to 0, but in a real implementation, this function would configure the ADC hardware.

```
void ADC_Init ()
{
```

```
        adcOutput = 0;
}
```

4. **`EEPROM_Init ()`**:
    - Initializes the EEPROM settings and sets the `hardware_revision` to `NONE` initially. In practice, this would typically involve configuring the hardware for EEPROM communication.

```
void EEPROM_Init ()
{
        hardware_revision = NONE;
}
```

5. **`EEPROM_Read ()`**:
    - This function reads the hardware revision from the EEPROM based on a predefined memory address (`0x6000` for REV_A or `0x7000` for REV_B). It returns either `REV_A` or `REV_B` based on the address value.

```
int EEPROM_Read()
{
        if (HARDWARE_SERIAL_NUMBER == 0xABC1234)
            return REV_A;
        else
            return REV_B;
}
```

6. **`delay(int counter)`**:
    - A simple delay function. It contains a loop or timer code to introduce a delay in execution.

```
void delay (int counter)
{
        int i;
        for (i = 0; (i < counter); i++)
                i++;
}
```

7. **`TempMon_ISR ()`** (Interrupt Service Routine):
    - This is the core function of the system, simulating an interrupt service routine. It reads the temperature from the ADC output, processes it according to the hardware revision, and updates the LED states:
        - If the temperature is below 5°C or above 105°C, the Red LED is turned on.
        - If the temperature is between 85°C and 105°C, the Yellow LED is turned on.
        - Otherwise, the Green LED is turned on.

```
void TempMon_ISR(void) {
    /* Get ADC output from the port */
    adcOutput = ADC_Output ();
    int temperature;

    if (hardware_revision == REV_A) {
        temperature = adcOutput;
    } else {
```

```
        temperature = adcOutput / 10;
    }

    /* Update LED based on temperature */
    if (temperature < 5 || temperature >= 105) {
        GPIO_Write (RED_LED_PIN, HIGH);
        GPIO_Write (YELLOW_LED_PIN, LOW);
        GPIO_Write (GREEN_LED_PIN, LOW);
    }
    else if (temperature >= 85) {
        GPIO_Write (RED_LED_PIN, LOW);
        GPIO_Write (YELLOW_LED_PIN, HIGH);
        GPIO_Write (GREEN_LED_PIN, LOW);
    }
    else {
        GPIO_Write (RED_LED_PIN, LOW);
        GPIO_Write (YELLOW_LED_PIN, LOW);
        GPIO_Write (GREEN_LED_PIN, HIGH);
    }
}
```

Main Function (`main()`):

- **GPIO Initialization**: Initializes the GPIO pins for the three LEDs (Red, Yellow, Green).
- **ADC Initialization**: Initializes the ADC to read temperature data.
- **EEPROM Initialization and Hardware Revision Read**: Initializes the EEPROM system and reads the hardware revision address to determine the type of hardware.
- **Main Control Loop**: A continuous loop where the interrupt service routine (`TempMon_ISR ()`) is called to update the LEDs based on the temperature, followed by a delay of 100 microseconds.

```
int main(void) {

    /* Initialize GPIO pins for LEDs */
    GPIO_Init(GREEN_LED_PIN);
    GPIO_Init(YELLOW_LED_PIN);
    GPIO_Init(RED_LED_PIN);

    /* Initialize ADC for temperature sensor reading */
    ADC_Init ();

    /* Initialize EEPROM and read configuration */
    EEPROM_Init ();
    int hardware_revision = EEPROM_Read(HARDWARE_SERIAL_NUMBER);

    /* Main control loop */
    while (1) {
        /* Call Interrupt service routine to get ADC data */
        TempMon_ISR ();
        /* Wait for next sample (100µs) */
        Delay (100);
    }
}
```

Implementation in C++ language:

Assumptions:

1. ready_count and not_ready_count were added in get_port() API to mimic the delay from the multiplexers to select the data.
2. write_data() and read_data() APIs were implemented with integer variables as port address instead of pointers to ease assignments.

Simulation of Temperature monitoring is implemented in two sub-modules.

- ADC.cpp
- TempMon.cpp

ADC.cpp

This C++ code simulates the operation of an **Analog-to-Digital Converter (ADC)**, managing the ADC hardware and interacting with multiple analog channels for data conversion.

1. Global Variables:

- **eoc:** End-Of-Conversion address, an unsigned integer.
- **base:** The base address for ADC hardware, initialized to `0x2000`.
- **AD_Enabled:** A boolean variable indicating whether the Analog-to-Digital conversion is enabled.
- **CurrentChannel:** Tracks the currently active analog channel (ranging from 0 to 7).

```
/* AD structures, prototypes & variables */
unsigned eoc;                 // End of conversion
unsigned int base = 0x2000;   // Base address
bool AD_Enabled = false;      // Analog to Digital conversion enable flag
// Filtering of raw signal, debounce and Diagnostics flag
bool FilterDebounceDiagnosticsDone = false;
int CurrentChannel = 0;      // current active channel
```

2. `Anachan` Structure:

- **data:** Holds the converted data for an analog channel.
- **status:** Represents the current state of the analog channel.

There are a total of 8 analog channels, stored in the array `AnalogChannel[8]`.

```
/* Analog Channel definition */
struct Anachan {
    int data;
    int status;
};

// Define 8 Analog channels
Anachan AnalogChannel [8];
```

3. `Anastat` Enum:

Defines three possible states for each analog channel:

- **INACTIVE:** The channel is not active and not performing any conversion.
- **START_CONVERSION:** The channel is starting the conversion process.
- **DATA_READY:** The data from the conversion is available.

```
/* Analog Channel states */
enum Anastat {
    INACTIVE,
    START_CONVERSION,
    DATA_READY
};
```

4. `ADC` Class:

The main class that simulates ADC operations, such as initialization, enabling channels, starting conversions, and retrieving data.

Constructor:

- Initializes the `base` address to `0`, sets `AD_Enabled` to `false`, and `CurrentChannel` to `0`.
- Uses `memset` to initialize all `AnalogChannel` entries to zero.

```
ADC () {
    base = 0;
    AD_Enabled = false;
    CurrentChannel = 0;
    memset(&AnalogChannel, 0, sizeof(AnalogChannel));
}
```

Methods:

- **`InitializeAnalog` Method:** Initializes the analog channel by calling `get_port`, which retrieves a port address for ADC communication. If no hardware is detected (port is `0`), it returns `0`. Otherwise, it sets up the `eoc` and returns the base address of the ADC.

```
// Initialize analog channel and set the base address
unsigned InitializeAnalog () {
    base = get_port();
    eoc = base + 0x18;

    if (base == 0)
        return 0;

    // return base address
    return base;
}
```

- **`TurnOnAnalog` Method:** Activates a specific analog channel (between 0 and 7). The channel's status is set to **START_CONVERSION**. If the provided channel number is invalid (outside the range 0 to 7), it returns `-1`.

```
// Set the active channel to START_CONVERSION and get the channel
int TurnOnAnalog (int channel) {
    if (channel < 0 || channel > 7)
```

```
        return -1;

    AnalogChannel[channel].status = START_CONVERSION;
    AD_Enabled = true;

    return channel;
}
```

- **`TurnOffAnalog` Method:** Deactivates a specific analog channel by setting its status to **INACTIVE**. It checks if the channel is indeed inactive. If it is, the method returns `-1`. If the channel data is valid, it returns the channel's data.

```
// Set the inactive channel to INACTIVE and get the last active data
int TurnOffAnalog(int channel) {
    if (channel < 0 || channel > 7)
        return -1;

    AnalogChannel[channel].status = INACTIVE;

    return AnalogChannel[channel].data;
}
```

- **`new_timer` Method:** Simulates a periodic timer interrupt. This method is responsible for checking and handling the status of the analog channels. It:
  - Finds the next active channel (if any) to process.
  - If the channel status is **START_CONVERSION**, it changes the status to **DATA_READY**.
  - If the status is **DATA_READY**, it reads data from the ADC hardware and then moves to the next channel.
  - It uses `read_data ()` (hardware interaction function) to read data from port.

```
    // Periodic timer interrupt. Checks and handles the status of the
analog channels
    void new_timer (int timer_counter) {
       // check if Analog to digital conversion is enabled
        if (AD_Enabled) {
            // look for start conversion or data ready status
            while (AnalogChannel[CurrentChannel].status != INACTIVE
                      && (timer_counter != 0)) {
                // update the data and status of Analog channels
                switch (AnalogChannel[CurrentChannel].status) {
                case START_CONVERSION:
                   AnalogChannel[CurrentChannel].status = DATA_READY;
                break;

                case DATA_READY:
                   while (!(read_data(eoc) & 0x80));
                   // Get data from Analog channel
                   AnalogChannel[CurrentChannel].data = read_data(base);
                   AnalogChannel[CurrentChannel].status = START_CONVERSION;
                break;

                case INACTIVE:
                break;
                }
            // increment to next channel
            CurrentChannel++;
```

```
            // decrement the counter until it becomes 0
            timer_counter--;
          }
        }
      }
```

- **`SampleData` Method:** Simulates the process of collecting data:
  - Initializes the analog system and turns on current channel.
  - Retrieves the value of current channel.
  - Perform filtering, debounce and Diagnostics on raw data.

```
void SampleData () {
    // Capture samples based on timer counter 100
    new_timer (100);

    unsigned x = InitializeAnalog ();
    cout << "init ana = " << hex << x << endl;

    x = TurnOnAnalog(CurrentChannel);
    cout << "TurnOnAnalog(CurrentChannel); = " << x << endl;

    x = GetChannelData(CurrentChannel);
    cout << "GetChannelData(CurrentChannel); = " << x << endl;

    // Perform filtering, debounce and Diagnostics on raw data
    FilterDebounceDiagnosticsDone = true;
}
```

- **`ADC_Raw` Method:** Simulates reading raw ADC data:
  - It checks for a valid port address using `get_port`.
  - Calls `SampleData` to perform a sample.
  - Reads data from the ADC channel and stores it in a `data` array.
  - Returns the `data` array containing the digitized values.

```
char* ADC_Raw () {
    unsigned ADC_Chan0, dac1, eoc;
    int count;
    static char data[300];

    // Open the port and get the data
    ADC_Chan0 = get_port();

    if (ADC_Chan0 == 0) {
        cout << "no hardware found" << endl;
        return nullptr;
    }

    // Sample signals based on the timer counter
    SampleData ();

    // set base and end of conversion address and offset
    dac1 = ADC_Chan0 + 8;
    eoc = ADC_Chan0 + 0x18;

    cout << "ADC Channel 0 after get_port = " << hex << ADC_Chan0 <<
endl;

    // capture the data from the channel and update buffer
```

```
    for (count = 0; count < 300; count++) {
        while (!(read_data(eoc) & 0x80));
        data[count] = (char) read_data(dac1);
    }

    return data;
}
```

- **`ADC_Output` Method:** Checks if the `ADC_Raw` method returns invalid data. If it does, it returns `0`; otherwise, digital converted data is returned. This sequence of steps occurs only if Filtering, Debounce and Diagnostics are done.

```
// Get the data output from ADC
int ADC_Output () {
    if (FilterDebounceDiagnosticsDone == true)
    {
        if (ADC_Raw ()! = nullptr)
            return 0;
        else
            return reinterpret_cast<int>(ADC_Raw());
    }
}
```

Private Methods:

- **`get_port` Method:** Searches for a valid ADC port in a given range (from `0x200` to `0x3c0`). It sends out a `write_data()` command and waits for the ADC to respond. If it finds a ready port, it returns the port address; otherwise, it returns `0`.

```
// Get the port address
unsigned get_port () {
    static unsigned local_port;
    int x, portAddress;
    unsigned int not_ready_count, ready_count;

    if (local_port == 32767)
        return 0;

    for (x = 0x200; x < 0x3c0; x += 0x40) {
            not_ready_count = 32767;
                ready_count = 32767;
                write_data(x, 0);

                // Wait until the port address is updated. Polling
                while ((read_data(x + 0x18) & 0x80) && --
not_ready_count);
                while (!(read_data(x + 0x18) & 0x80) && --
ready_count);
                if (ready_count < 32767 && ready_count > 0) {
                        local_port = base = x;
                        portAddress = local_port + 0x18;
                        return portAddress;
                }
        }
    }
```

- **`get_frequency` Method:** Returns a fixed value of `200.0`, simulating the frequency of the ADC.

```
double get_frequency () {
        return 200.0;
}
```

- **`read_data` Method:** A simulation of reading data from hardware I/O, returning the input value passed to it. (simplified here to just return data)

```
int read_data (int x) {
        return x;
}
```

- **`write_data` Method:** A simulation of writing to hardware I/O, where the value passed in is assigned to `x`. (simplified here to just assign data to x)

```
void write_data(int x, int data) {
        x = data;
}
```

- **`GetChannelData` Method:** Returns the data value stored in the specified analog channel.

```
int GetChannelData(int channel) {
        return AnalogChannel[channel].data;
}
```

5. `ADC_main` Function:

- The entry point of the ADC simulation.
- It creates an `ADC` object and calls `SampleData` to perform ADC operations.
- Fetches ADC output and returns the same.

```
// main function to update ADC output
int ADC_main () {
        ADC adc;
        adc.SampleData();
        int output = adc.ADC_Output();
        return output;
}
```

Summary:

This code simulates the operation of an **Analog-to-Digital Converter (ADC)** system in a hardware abstraction layer. It provides functionality for:

- Initializing ADC channels and setting their states (active/inactive).
- Managing the conversion process and storing results.
- Reading and writing data using simulated hardware I/O functions (`read_data` and `write_data`).

The system operates on 8 analog channels, with a fixed sampling frequency. The `ADC` class handles both the control of these channels and the conversion process.

<u>TempMon.cpp</u>

1. Header and Constants:

- **#include <iostream>:** Includes standard input-output stream functionality for printing to the console.
- **#include <thread> and #include <chrono>:** These headers would typically be used for threading and timing.
- **#include "TempMon.hpp":** Assumed to be a custom header file containing additional declarations.
- **LED Pins:**
  - RED_LED_PIN, YELLOW_LED_PIN, GREEN_LED_PIN: Defined as hexadecimal values to represent GPIO pin addresses for three LEDs.
- **Hardware Serial Number and Revision:**
  - **Hardware Serial Number**: A constant representing the serial number of hardware.
  - **Hardware Revision Enum:** Defines different hardware revision types (REV_A, REV_B, NONE).

```
#include <iostream>
#include <thread>
#include <chrono>
#include "TempMon.hpp"

#define RED_LED_PIN    0x1000
#define YELLOW_LED_PIN 0x2000
#define GREEN_LED_PIN  0x3000

#define HARDWARE_SERIAL_NUMBER 0xABC1234

#define HIGH 1
#define LOW  0

enum HardwareRevision { REV_A = 0, REV_B, NONE };
```

2. TempMon Class:

This class represents the temperature monitoring system. It contains several methods and member variables for managing the GPIO pins, reading from ADC, handling hardware revisions, and updating the LED status based on the temperature.

- **Member Variables:**
  - adcOutput: Stores the ADC output value.
  - temperature: Stores the calculated temperature based on ADC output.
  - hardware_revision: Stores the hardware revision type (either REV_A, REV_B, or NONE).

```
class TempMon {
private:
    int adcOutput;
    int temperature;
    HardwareRevision hardware_revision;
```

- **Constructor:** The constructor initializes adcOutput, temperature, and hardware_revision to default values.

```
TempMon (): adcOutput(0), temperature(0), hardware_revision(NONE) {}
```

- **`GPIO_Write` and `GPIO_Init`:**
  - These methods are placeholders for controlling GPIO pins.
  - `GPIO_Write` simulates setting a GPIO pin to a specified status (HIGH or LOW).
  - `GPIO_Init` simulates initializing a GPIO pin by setting its data to zero.

```
    void GPIO_Write(int PIN, int status) {
        // Write data into GPIO port PIN
        // Since we're using C++, we assume this is just a placeholder
function.
        std::cout << "GPIO " << std::hex << PIN << " set to " << status <<
std::dec << std::endl;
    }

    void GPIO_Init(int PIN) {
        // Initialize 0 to the address PIN
        int data = 0;
        std::cout << "GPIO " << std::hex << PIN << " initialized to " <<
data << std::dec << std::endl;
    }
```

- **`ADC_Init` and `EEPROM_Init`:**
  - `ADC_Init`: Initializes the ADC output (though in this case, it sets `adcOutput` to zero).
  - `EEPROM_Init`: Initializes hardware revision from the EEPROM (though it sets `hardware_revision` to `NONE` initially).

```
    void ADC_Init () {
        adcOutput = 0;
    }

    void EEPROM_Init () {
       // sample code for initialize
        hardware_revision = NONE;
    }
```

- **`EEPROM_Read`:**
  - This function simulates reading the hardware revision from an EEPROM. It returns different revision types based on the address provided (`0x6000` returns `REV_A`, `0x7000` returns `REV_B`).

```
    int EEPROM_Read() {
        if (HARDWARE_SERIAL_NUMBER == 0xABC1234)
            return REV_A;
        else
            return REV_B;
    }
```

- **`delay`:**
  - This method is a placeholder for introducing a delay in the main control loop. It increments a counter until the counter decrements to zero.

```
// dummy delay function
void delay (int counter) {
    int i;
    for(i = 0; (i < counter); i++)
            i++;
}
```

- **`ADC_Output`:**
  - A placeholder method that simulates getting the ADC output (returns `adcOutput`).

```
int ADC_Output() {
    // Placeholder for actual ADC output function
    return adcOutput;
}
```

- **`TempMon_ISR`:**
  - This is the interrupt service routine for handling the temperature monitoring process.
  - Reads the ADC output.
  - Calculates the temperature based on the hardware revision (for REV_A, it uses the raw ADC value; for REV_B, it divides by 10).
  - Based on the temperature, it controls the LEDs:
    1. If the temperature is below 5 or above 105, it lights up the red LED.
    2. If the temperature is between 85 and 105, it lights up the yellow LED.
    3. Otherwise, the green LED is turned on.

```
void TempMon_ISR() {
    /* Get ADC output from the port */
    adcOutput = ADC_Output();

    if (hardware_revision == REV_A) {
        temperature = adcOutput;
    } else {
        temperature = adcOutput / 10;
    }

    /* Update LED based on temperature */
    if (temperature < 5 || temperature >= 105) {
        GPIO_Write(RED_LED_PIN, HIGH);
        GPIO_Write(YELLOW_LED_PIN, LOW);
        GPIO_Write(GREEN_LED_PIN, LOW);
    } else if (temperature >= 85) {
        GPIO_Write(RED_LED_PIN, LOW);
        GPIO_Write(YELLOW_LED_PIN, HIGH);
        GPIO_Write(GREEN_LED_PIN, LOW);
    } else {
        GPIO_Write(RED_LED_PIN, LOW);
        GPIO_Write(YELLOW_LED_PIN, LOW);
        GPIO_Write(GREEN_LED_PIN, HIGH);
    }
}
```

- **`MainLoop`:**
  - This is the main control loop of the program, where:
    1. The GPIO pins for the LEDs are initialized.

2. The ADC is initialized.
3. The EEPROM is initialized, and the hardware revision is read.
4. The system continuously reads temperature data, processes it, and updates the LEDs every 100 microseconds.

```
void MainLoop () {
    /* Initialize GPIO pins for LEDs */
    GPIO_Init(GREEN_LED_PIN);
    GPIO_Init(YELLOW_LED_PIN);
    GPIO_Init(RED_LED_PIN);

    /* Initialize ADC for temperature sensor reading */
    ADC_Init ();

    /* Initialize EEPROM and read configuration */
    EEPROM_Init();
    hardware_revision = static_cast<HardwareRevision>(EEPROM_Read());

    /* Main control loop */
    while (true) {
        /* Call Interrupt service routine to get ADC data */
        TempMon_ISR();

        /* Wait for next sample (100µs) */
        Delay (100);
    }
}
```

3. Main Function:

- The `main` function creates an instance of the `TempMon` class and calls the `MainLoop` method to start the monitoring process.

```
int main() {
    TempMon tempMon;
    tempMon.MainLoop();
    return 0;
}
```

Summary of Operation:

- The system continuously monitors temperature data through the ADC, processes the value based on the hardware revision, and uses three LEDs (red, yellow, green) to visually represent different temperature ranges. The temperature monitoring loop runs indefinitely with a 100-microsecond delay between each cycle.