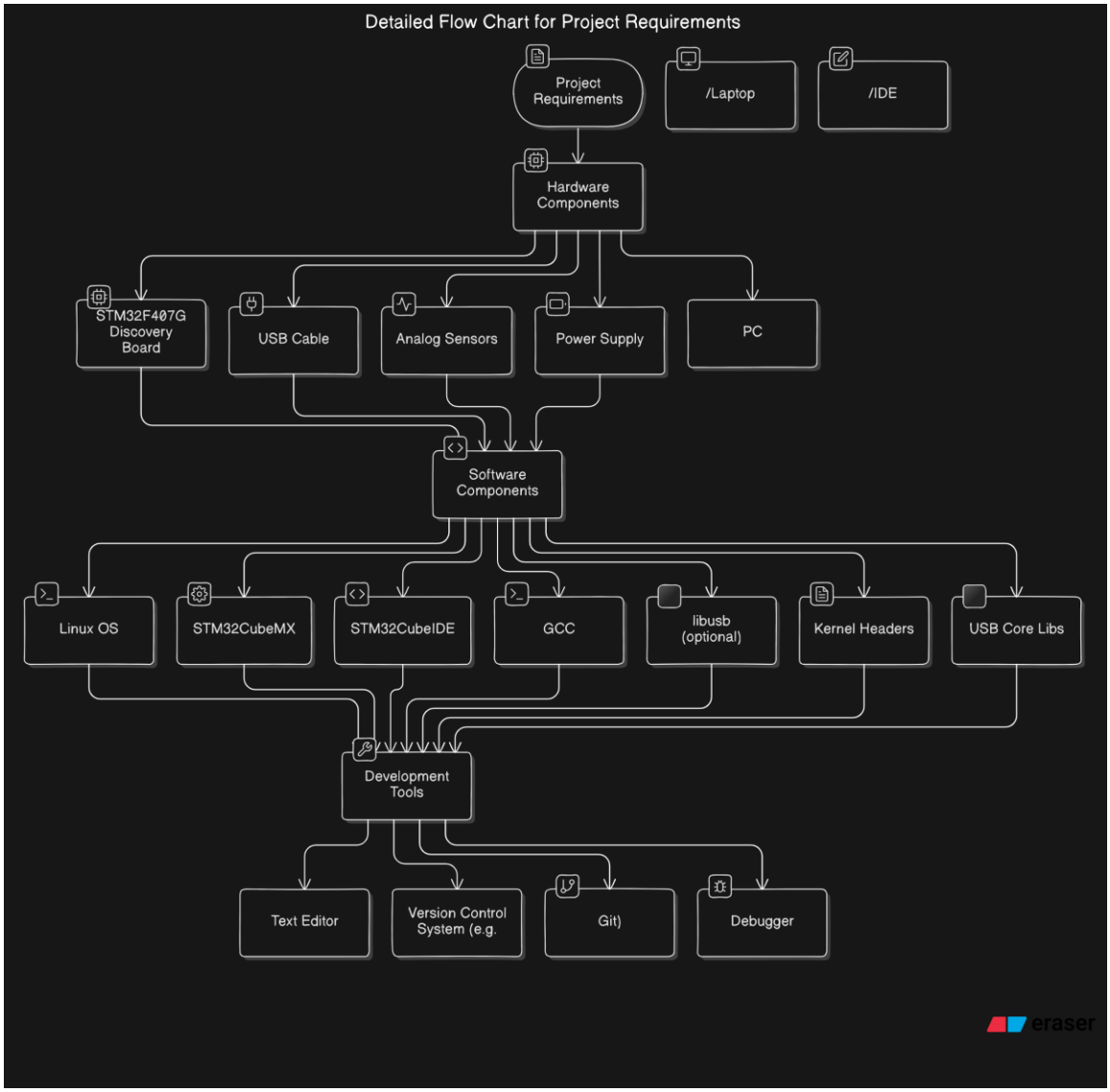
# *1.* INTRODUCTION TO PROJECT

The "**Development of a Linux USB Device Driver Using the STM32F407G Discovery Board**" project is designed to explore and implement the fundamental concepts of USB communication through a practical application. This project focuses on creating a robust USB device driver for the Linux operating system using the STM32F407G Discovery Board, a powerful microcontroller development platform.

The primary goal of this project is to facilitate seamless data transfer between a USB device and a Linux-based system. The system enables users to interact with a USB device configured as a microcontroller, allowing for efficient data acquisition and communication. The project involves designing and implementing a USB device driver that supports real-time data streaming and reliable data exchange.

By developing this driver, the project not only enhances the understanding of USB communication protocols but also provides a practical tool for real-time data acquisition and analysis, demonstrating the potential of integrating microcontroller-based systems with modern Linux environments for advanced applications.

## 2.REQUIREMENTS



## Hardware Components

### 2.1 STM32F407G Discovery Board

The STM32F407G Discovery Board is a development platform featuring the STM32F407G microcontroller, which includes a range of peripherals like GPIOs, ADCs, and USB interfaces. This board will act as the USB device in the project, allowing us to implement and test the USB device driver by providing the necessary hardware environment to interact with the Linux host system.

### 2.2 USB Cable

A standard USB cable is essential for establishing a physical connection between the STM32F407G Discovery Board and the Linux host system. This connection enables the transfer of data between the microcontroller and the host, which is crucial for the USB device driver's functionality and testing.

### 2.3 **Analog Sensors**

For data acquisition, the project uses two specific analog sensors: the LM35 temperature sensor and a Light Dependent Resistor (LDR). The LM35 sensor measures temperature and provides an analog voltage proportional to the temperature, while the LDR detects light intensity by varying its resistance. These sensors are connected to the analog-to-digital converters (ADCs) on the STM32F407G Discovery Board. The ADC converts the analog signals from the LM35 and LDR into digital data, which is then transmitted to the Linux host via USB. This setup allows for real-time monitoring and acquisition of environmental data.

### 2.4 **Power Supply**

A reliable power supply is necessary to provide stable and sufficient power to the STM32F407G Discovery Board and the connected sensors. The power supply ensures consistent operation and prevents disruptions that could affect the performance and reliability of the system.

### 2.5 PC/Laptop

A PC or laptop running a Linux-based operating system is required for the development, testing, and debugging of the USB device driver. This computer will be used to write and compile the driver code, interact with the STM32F407G Discovery Board, and verify that the driver performs correctly in a Linux environment.

**Software Components**

### 2.6 Linux OS

The Linux operating system provides the environment needed for developing and testing the USB device driver. It supports USB communication and driver integration, allowing for interaction with the STM32F407G Discovery Board and facilitating the overall development process.

### 2.7 STM32CubeIDE

STM32CubeIDE is an Integrated Development Environment (IDE) tailored for STM32 microcontrollers. It will be used to write, compile, and debug the firmware for the STM32F407G Discovery Board, ensuring that the USB communication functionalities are properly implemented.

### 2.8 GCC

The GNU Compiler Collection (GCC) provides the tools necessary for compiling C code into executable firmware for the STM32 microcontroller. GCC will be used to convert the source code into a format that the STM32F407G can execute.

### 2.9 Kernel Headers

Kernel headers are needed to build kernel modules and drivers, providing the necessary definitions and functions. They are essential for developing the USB device driver, allowing integration with the Linux kernel's USB subsystem**.**

### 2.10 USB Core Libraries

The USB core libraries in the Linux kernel handle fundamental USB communication tasks. These libraries support the development and management of USB device functionalities, ensuring proper data transfer and device interaction.

**Development Tools**

### 2.11 Integrated Development Environment (IDE)

An IDE such as STM32CubeIDE will be used for writing, compiling, and debugging both the firmware and the USB device driver code. This tool facilitates code development and management by providing an integrated environment with features like code editing, debugging, and project management

### 2.11 C Programming

Proficiency in C programming is essential for developing the firmware and USB device driver. C is used for writing the low-level code that interacts with the STM32 microcontroller and implements the USB communication protocols.

### 2.11 Version Control System (e.g., Git)

A version control system like Git is crucial for managing source code revisions, tracking changes, and collaborating with other developers. Git helps maintain a history of changes and supports code management practices.

### 2.11 Debugger

Debuggers are tools used for testing and troubleshooting firmware and driver code. They allow developers to step through code, set breakpoints, and monitor system states, which is essential for identifying and fixing issues during development.

# 3. USB Protocol

## 3.1. Introduction

The Universal Serial Bus (USB) protocol is a standardized method for connecting peripherals to computers, facilitating communication and power supply. USB is designed to be a plug-and-play interface, allowing devices to be connected and used with minimal user intervention. It supports data transfer, power delivery, and device management across a wide range of applications.

## 3.2. USB Architecture

- **Host and Device**: USB communication is based on a host-device model. The host (typically a computer or a USB hub) controls the communication and manages the devices connected to it. The device (such as a keyboard, mouse, or USB flash drive) responds to the host's commands.

- **Endpoints**: USB devices use endpoints for communication. Each endpoint is a designated data channel within the device. Endpoints are categorized into:

    - **Control Endpoints**: Used for device configuration and management.
    - **Bulk Endpoints**: Used for large data transfers.
    - **Interrupt Endpoints**: Used for time-sensitive data transfers.
    - **Isochronous Endpoints**: Used for data requiring consistent timing, such as audio or video streams.
- **Pipes**: A pipe is a logical connection between a host and a device endpoint. It defines the data transfer type and direction (IN or OUT). Pipes are used for managing data flow between the host and device.

## 3.3. USB Data Transfer Types

- **Control Transfers**: Used for device configuration and command operations. Control transfers involve a setup stage, data stage, and status stage. They are used for operations such as setting up device descriptors and configurations.

- **Bulk Transfers**: Used for transferring large amounts of data with no specific timing requirements. Bulk transfers are reliable and can handle high data throughput.

- **Interrupt Transfers**: Used for time-sensitive data transfers where data needs to be delivered within a specific timeframe. Interrupt transfers are often used for devices that require timely responses, such as keyboards or mice.

- **Isochronous Transfers**: Used for data streams requiring continuous, real-time delivery. Isochronous transfers ensure a consistent data rate but do not guarantee error-free delivery. They are commonly used for audio and video devices.

### 3.4. USB Device Classes

USB devices are organized into different classes based on their functionality. Each class defines a set of standards and protocols for device communication. Examples of USB device classes include:

- **Human Interface Device (HID)**: Includes devices like keyboards and mice.
- **Mass Storage Class (MSC)**: Includes devices like USB flash drives and external hard drives.
- **Communication Device Class (CDC)**: Includes devices like modems and network adapters.
- **Audio Class**: Includes devices like USB microphones and speakers.

### 3.5. USB Protocol Layers

The USB protocol consists of several layers, each responsible for different aspects of USB communication:

- **Physical Layer**: Defines the electrical and mechanical properties of the USB interface, including connectors and cables.
- **Data Link Layer**: Manages the data framing, error detection, and retransmission. It ensures reliable data transfer between the host and device.
- **Protocol Layer**: Defines the communication protocols, including transfer types and device management commands.
- **Application Layer**: Interfaces with higher-level applications and handles device-specific operations.

### 3.6. USB Communication Flow

- **Enumeration**: The process of identifying and configuring a USB device when it is connected to the host. Enumeration involves assigning device addresses and retrieving device descriptors.

- **Descriptors**: Structures that provide information about a USB device's capabilities and configuration. Key descriptors include:

    - **Device Descriptor**: Contains information about the device, such as vendor ID, product ID, and device class.
    - **Configuration Descriptor**: Describes the device's power requirements and configuration options.
    - **Interface Descriptor**: Defines the functionalities provided by each interface within the device.
    - **Endpoint Descriptor**: Provides details about each endpoint, including its type and size.
- **Control Transfer Protocol**: Used for device configuration and management. It involves three stages: setup, data, and status.

- **Data Transfer**: Data transfer between the host and device can occur in different modes depending on the transfer type (bulk, interrupt, isochronous).

### 3.7. USB Speeds

USB supports different speeds to accommodate various data transfer needs:

- **Low-Speed (1.5 Mbps)**: Used for devices with low data transfer requirements, such as keyboards and mice.
- **Full-Speed (12 Mbps)**: Used for devices requiring moderate data transfer, such as printers and audio devices.
- **High-Speed (480 Mbps)**: Used for devices with high data transfer needs, such as webcams and external hard drives.
- **SuperSpeed (5 Gbps and above)**: Used for very high-speed data transfer, such as USB 3.0 and USB 3.1 devices.

### 3.8. USB Error Handling

USB includes mechanisms for detecting and handling errors during data transfer. These mechanisms include error checking, retries, and acknowledgment of successful data transfer.

### 3.9. USB Driver Development

- **Driver Interfaces**: USB drivers interact with the USB subsystem of the operating system, handling device enumeration, data transfer, and device management.
- **Driver APIs**: Provide functions for interacting with USB devices, including initialization, configuration, and data transfer operations.
- **Kernel Integration**: USB drivers must be integrated with the operating system's kernel, ensuring proper communication and management of USB devices.

# 4. USB Device Driver Code

## 4.1. Header Files

- <linux/usb.h>: Provides definitions and functions for interacting with USB devices in the Linux kernel. This header file includes structures like struct usb_device, struct usb_interface, and functions such as usb_register() and usb_deregister().
- <linux/completion.h>: Defines the DECLARE_COMPLETION() macro and the complete() function used for signaling the completion of asynchronous operations.
- <linux/spinlock.h>: Provides definitions and functions for spinlocks used to protect shared data structures from concurrent access.
- <linux/module.h>: Contains macros for module information and initialization functions, such as module_init() and module_exit().
- <linux/kernel.h>: Provides kernel-specific functions like printk() for logging messages, and kmalloc() and kfree() for memory management.
- <linux/fs.h>: Defines file operations structures like struct file_operations used to implement operations like open(), read(), write(), and release().
- <linux/slab.h>: Provides memory allocation functions such as kmalloc(), kzalloc(), and kfree().
- <linux/uaccess.h>: Contains functions for user-space access, including copy_to_user() and copy_from_user().

## 4.2. Definitions

- USB_CDC_VENDOR_ID: Vendor ID for the STM32 microcontroller.
- USB_CDC_PRODUCT_ID: Product ID for the STM32 microcontroller.
- USB_CDC_EP_IN: Endpoint address for incoming data (IN direction).
- USB_CDC_EP_OUT: Endpoint address for outgoing data (OUT direction).
- USB_CDC_BUFSIZE: Buffer size for data transfers (512 bytes).

## 4.3. USB Structures

- struct usb_device *device: Pointer to the USB device structure, representing the USB device connected to the system.
- struct usb_class_driver class: Defines the USB class driver, including file operations and device name.
- struct usb_anchor *anchor: Used to anchor URBs (USB Request Blocks) for bulk transfers.
- struct urb *read_urb: URB for handling read operations from the USB device.
- struct urb *write_urb: URB for handling write operations to the USB device.
- DECLARE_COMPLETION(read_completion): Declares a completion structure for signaling read completion.
- DECLARE_COMPLETION(write_completion): Declares a completion structure for signaling write completion.
- char kdata[USB_CDC_BUFSIZE]: Kernel buffer for storing data during write operations.

### 4.4. Functions

- read_complete(): Callback function triggered when a read operation completes. It signals the completion event for read operations.

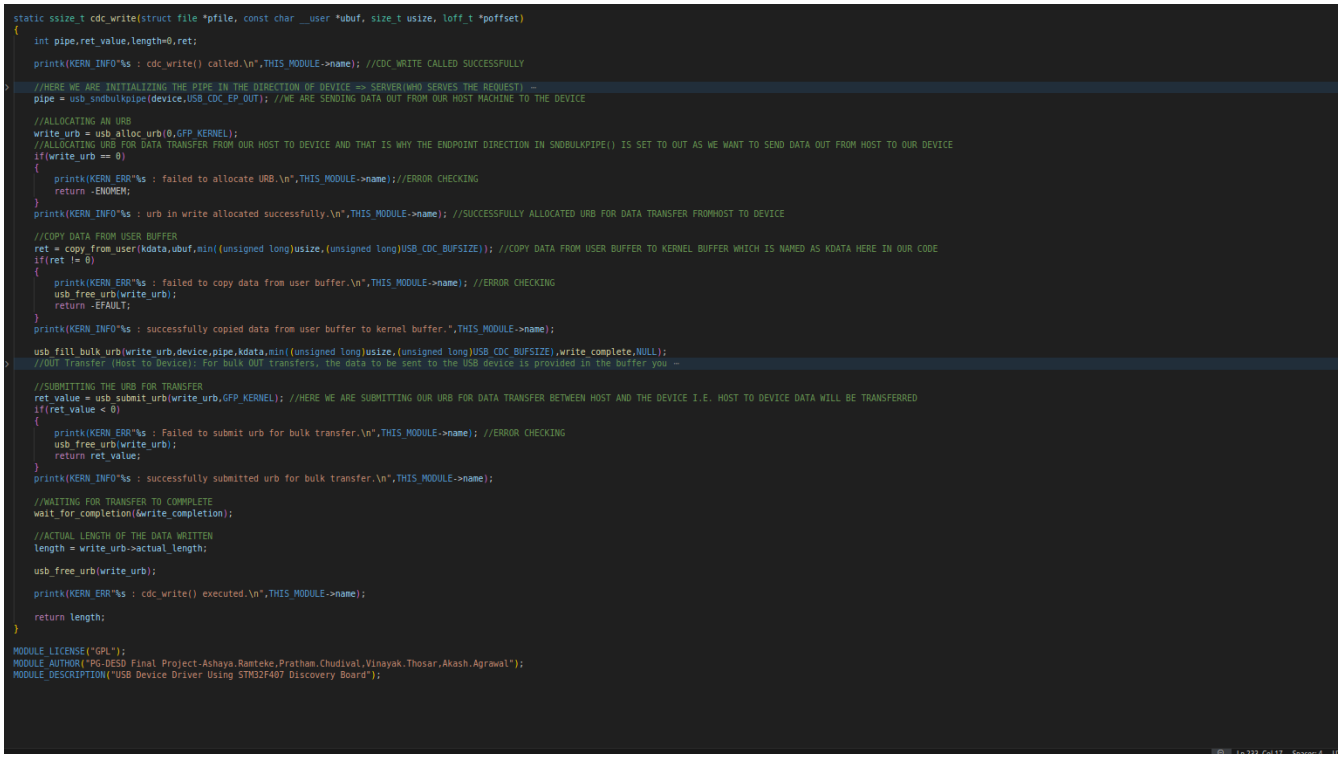
- write_complete(): Callback function triggered when a write operation completes. It signals the completion event for write operations.

### 4.5. Device File Operations

- cdc_open(): Opens the device file, called when a user-space application opens the device.
- cdc_close(): Closes the device file, called when a user-space application closes the device.
- cdc_read(): Handles read operations from the USB device. It allocates buffers, submits a URB for reading, and copies data from the kernel buffer to user-space.
- cdc_write(): Handles write operations to the USB device. It copies data from user-space to a kernel buffer, submits a URB for writing, and sends the data to the device.

### 4.6. USB Driver Functions

- cdc_probe(): Called when a USB device matching the device ID table is connected. Initializes the driver, allocates resources, and registers the device with the USB subsystem.



```
static ssize_t cdc_write(struct file *pfile, const char __user *ubuf, size_t usize, loff_t *poffset)
{
    int pipe,ret_value,length=0,ret;

    printk(KERN_INFO"%s : cdc_write() called.\n",THIS_MODULE->name); //CDC_WRITE CALLED SUCCESSFULLY

    //HERE WE ARE INITIALIZING THE PIPE_IN THE DIRECTION OF DEVICE => SERVER(WHO SERVES THE REQUEST) …
    pipe = usb_sndbulkpipe(device,USB_CDC_EP_OUT); //WE ARE SENDING DATA OUT FROM OUR HOST MACHINE TO THE DEVICE

    //ALLOCATING AN URB
    write_urb = usb_alloc_urb(0,GFP_KERNEL);
    //ALLOCATING URB FOR DATA TRANSFER FROM OUR HOST TO DEVICE AND THAT IS WHY THE ENDPOINT DIRECTION IN SNDBULKPIPE() IS SET TO OUT AS WE WANT TO SEND DATA OUT FROM HOST TO OUR DEVICE
    if(write_urb == 0)
    {
        printk(KERN_ERR"%s : failed to allocate URB.\n",THIS_MODULE->name);//ERROR CHECKING
        return -ENOMEM;
    }
    printk(KERN_INFO"%s : urb in write allocated successfully.\n",THIS_MODULE->name); //SUCCESSFULLY ALLOCATED URB FOR DATA TRANSFER FROMHOST TO DEVICE

    //COPY DATA FROM USER BUFFER
    ret = copy_from_user(kdata,ubuf,min((unsigned long)usize,(unsigned long)USB_CDC_BUFSIZE)); //COPY DATA FROM USER BUFFER TO KERNEL BUFFER WHICH IS NAMED AS KDATA HERE IN OUR CODE
    if(ret != 0)
    {
        printk(KERN_ERR"%s : failed to copy data from user buffer.\n",THIS_MODULE->name); //ERROR CHECKING
        usb_free_urb(write_urb);
        return -EFAULT;
    }
    printk(KERN_INFO"%s : successfully copied data from user buffer to kernel buffer.",THIS_MODULE->name);

    usb_fill_bulk_urb(write_urb,device,pipe,kdata,min((unsigned long)usize,(unsigned long)USB_CDC_BUFSIZE),write_complete,NULL);
    //OUT Transfer (Host to Device): For bulk OUT transfers, the data to be sent to the USB device is provided in the buffer you …

    //SUBMITTING THE URB FOR TRANSFER
    ret_value = usb_submit_urb(write_urb,GFP_KERNEL); //HERE WE ARE SUBMITTING OUR URB FOR DATA TRANSFER BETWEEN HOST AND THE DEVICE I.E. HOST TO DEVICE DATA WILL BE TRANSFERRED
    if(ret_value < 0)
    {
        printk(KERN_ERR"%s : Failed to submit urb for bulk transfer.\n",THIS_MODULE->name); //ERROR CHECKING
        usb_free_urb(write_urb);
        return ret_value;
    }
    printk(KERN_INFO"%s : successfully submitted urb for bulk transfer.\n",THIS_MODULE->name);

    //WAITING FOR TRANSFER TO COMMPLETE
    wait_for_completion(&write_completion);

    //ACTUAL LENGTH OF THE DATA WRITTEN
    length = write_urb->actual_length;

    usb_free_urb(write_urb);

    printk(KERN_ERR"%s : cdc_write() executed.\n",THIS_MODULE->name);

    return length;
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("PG-DESD Final Project-Ashaya.Ramteke,Pratham.Chudival,Vinayak.Thosar,Akash.Agrawal");
MODULE_DESCRIPTION("USB Device Driver Using STM32F407 Discovery Board");
```

- cdc_disconnect(): Called when a USB device is removed. Cleans up resources and unregisters the device.

```
//DISCONNECT FUNCTIONALITY
static void cdc_disconnect(struct usb_interface *intf)
{
    usb_deregister_dev(intf,&class);//this removes the device node from /dev directory and reverses the operations performed by use_reg
    printk(KERN_INFO"%s : USB CDC device successfully disconnected .\n",THIS_MODULE->name);
}

//F_OPS OPEN
static int cdc_open(struct inode *pinode, struct file *pfile)
{
    printk(KERN_INFO"%s : cdc_open() called .\n",THIS_MODULE->name);
    return 0;
}

//FOPS CLOSE
static int cdc_close(struct inode *pinode, struct file *pfile)
{
    printk(KERN_INFO"%s : cdc_close called .\n",THIS_MODULE->name);
    return 0;
}
```

### 4.7. Kernel Module Initialization and Exit

- cdc_init(): Initializes the USB device driver. Allocates memory for the anchor, initializes spinlocks, and registers the USB driver.
- cdc_exit(): Cleans up the USB device driver. Deregisters the USB driver and frees allocated memory.

### 4.8. Module Metadata

- MODULE_LICENSE("GPL"): Specifies the license for the module, indicating it is GPL-compliant.
- MODULE_AUTHOR("**AUTHOR NAME**"): Author information for the module.
- MODULE_DESCRIPTION("**DESCRIPTION ABOUT THE MODULE**"): Describes the purpose of the module.

### 4.9. Key Points

- **URBs (USB Request Blocks)**: Used for managing USB data transfers. They are allocated, filled with transfer details, and submitted to the USB subsystem.
- **Completion Handling**: The driver uses completion structures to signal the completion of asynchronous read and write operations.
- **Buffer Management**: Memory is allocated and freed for buffers used in read and write operations to handle data transfer between the USB device and user-space applications.

This code implements a basic USB device driver for the STM32F407 Discovery Board, handling read and write operations, managing USB requests, and interacting with user-space applications.

# 5. SENSORS : LDR & LM35

### 5.1 Introduction to LDR (Light Dependent Resistor)

An **LDR** (Light Dependent Resistor), also known as a photoresistor, is a light-sensitive device that exhibits a change in resistance when exposed to light. The resistance decreases with an increase in light intensity, making it useful in light-sensing circuits.

#### 5.1.1 Working Principle

The LDR is made of semiconductor materials with high resistance in darkness. When light falls on the semiconductor, photons give energy to electrons, making them jump to a higher energy state, reducing the material's resistance. This change in resistance can be used to detect the intensity of light in the environment.

#### 5.1.2 Applications

- **Automatic street lights**: Turns on the lights at dusk and off at dawn.
- **Light meters**: Used in photographic devices to measure light intensity.
- **Alarm systems**: Detects light levels to trigger an alarm.

#### 5.1.3 Integration with STM32F407

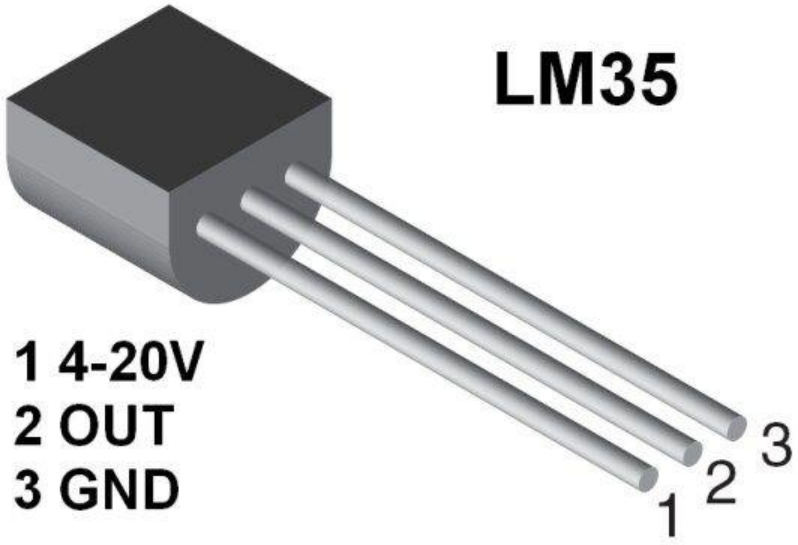
In your project, the LDR can be connected to one of the ADC (Analog-to-Digital Converter) channels of the STM32F407. The ADC converts the analog signal, which

varies with light intensity, into a digital value. This value can be processed to determine the corresponding light level.

- **Circuit Connection**: The LDR is connected in series with a resistor to form a voltage divider. The output voltage of this divider is fed to the ADC pin of the STM32F407.
- **ADC Configuration**: The multi-channel ADC of STM32F407 can be configured to continuously monitor the LDR's output, allowing real-time light intensity monitoring.

### 5.2 Introduction to LM35 (Temperature Sensor)

The **LM35** is a precision integrated-circuit temperature sensor that outputs a voltage linearly proportional to the Celsius temperature. Unlike thermistors, the LM35 does not require external calibration or trimming.




#### 5.2.1 Working Principle

The LM35 produces a linear output voltage (10 mV/°C), making it straightforward to interface with microcontrollers like the STM32F407. The sensor has an operating range from -55°C to +150°C and provides accurate temperature readings with minimal external components.

### 5.2.2 Applications

- **Thermal management**: In systems requiring precise temperature control, such as in HVAC (Heating, Ventilation, and Air Conditioning).
- **Industrial temperature monitoring**: Used in various industrial environments for monitoring and controlling temperature-sensitive processes.
- **Battery management systems**: Monitors the temperature of batteries to prevent overheating.

### 5.2.3 Integration with STM32F407

For temperature measurement, the LM35's output is an analog voltage that can be connected to another ADC channel of the STM32F407. The ADC converts this analog voltage into a digital value, which can be further processed to obtain the temperature.

- **Circuit Connection**: The LM35 has three pins: Vcc (power supply), GND (ground), and Vout (output). The Vout pin is connected to the ADC pin of the STM32F407.

- **ADC Configuration**: The ADC is configured to read the voltage from the LM35, and the temperature is calculated by converting this digital value back into Celsius using the formula:
  (°C) ValueTemperature (°C)=ADC Value×1024Vref×100

## 5.3 Multi-Channel ADC on STM32F407

The **STM32F407** microcontroller is equipped with multiple ADC channels that allow simultaneous sampling of different analog signals. This feature is particularly useful in projects where multiple sensors, like LDR and LM35, need to be monitored in real-time.

### 5.3.1 ADC Features

- **Resolution**: 12-bit resolution, which provides a digital value between 0 to 4095 for a full-scale input.
- **Channels**: The STM32F407 has up to 16 ADC channels.
- **Conversion Modes**: Supports single, continuous, scan, and discontinuous modes, allowing flexible configuration based on application requirements.

### 5.3.2 Configuration for Multi-Channel Operation

- **Step 1: Initialize ADC**: Configure the ADC peripheral by enabling the corresponding clock and setting up the resolution and data alignment.
- **Step 2: Channel Selection**: Specify the channels connected to the LDR and LM35. These channels can be selected for sequential conversion.

- **Step 3: Continuous Conversion Mode**: This mode allows the ADC to automatically start a new conversion after the last one completes, ideal for real-time monitoring.
- **Step 4: DMA Integration**: For efficient data handling, the ADC can be configured to work with DMA (Direct Memory Access), allowing the automatic transfer of conversion results to memory without CPU intervention.

## 5.4 Project Implementation

### 5.4.1 LDR and LM35 Setup

In your device driver project, the LDR and LM35 are interfaced with the STM32F407's multi-channel ADC. The LDR provides data on ambient light levels, while the LM35 monitors the system's temperature. By using multiple ADC channels, both sensors can be monitored simultaneously.

### 5.4.2 Data Processing

The STM32F407's firmware processes the digital values obtained from the ADC to interpret the light intensity and temperature. Thresholds can be set to trigger specific actions, such as turning on/off lights or activating cooling mechanisms.

### 5.4.3 Practical Applications

This setup can be used in various practical applications, such as:

- **Smart lighting systems**: Automatically adjusting lighting based on ambient conditions.
- **Environmental monitoring systems**: Keeping track of temperature and light levels in industrial or residential settings.

## 5.5 Conclusion

By integrating LDR and LM35 with the STM32F407's multi-channel ADC, you can develop robust and efficient systems for monitoring environmental conditions. The use of STM32F407 allows simultaneous and accurate data acquisition from multiple sensors, providing real-time insights into the operating environment.

# 6. STM32 USB DEVICE

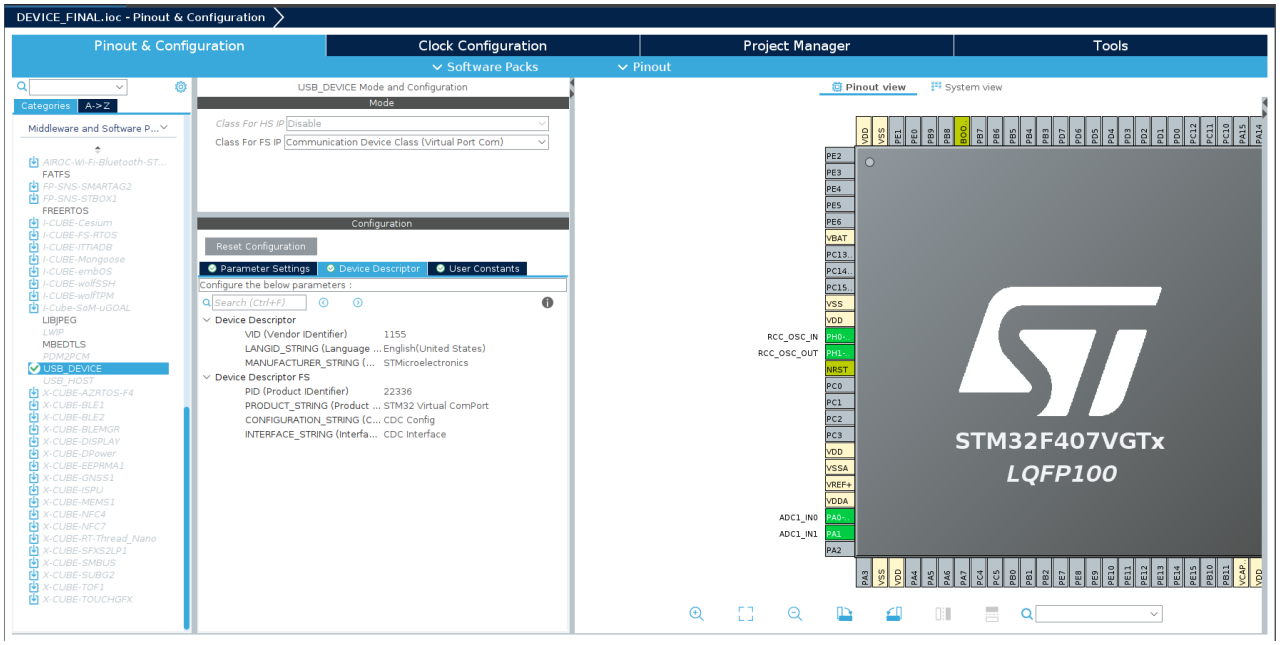
## 6.1 Introduction

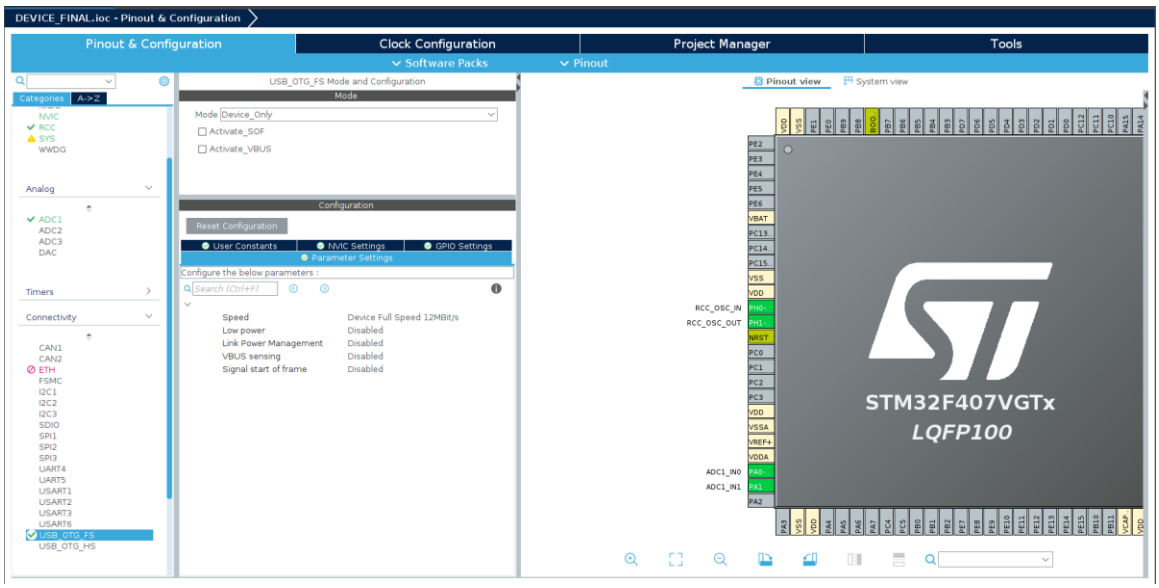
This chapter outlines the implementation of LDR (Light Dependent Resistor) and LM35 (Temperature Sensor) using the STM32F407 microcontroller. The sensors are interfaced with the microcontroller via its multi-channel ADC (Analog-to-Digital Converter), allowing the system to monitor both light intensity and temperature in real-time. The data obtained from these sensors is processed and transmitted via USB to a host computer for further analysis.

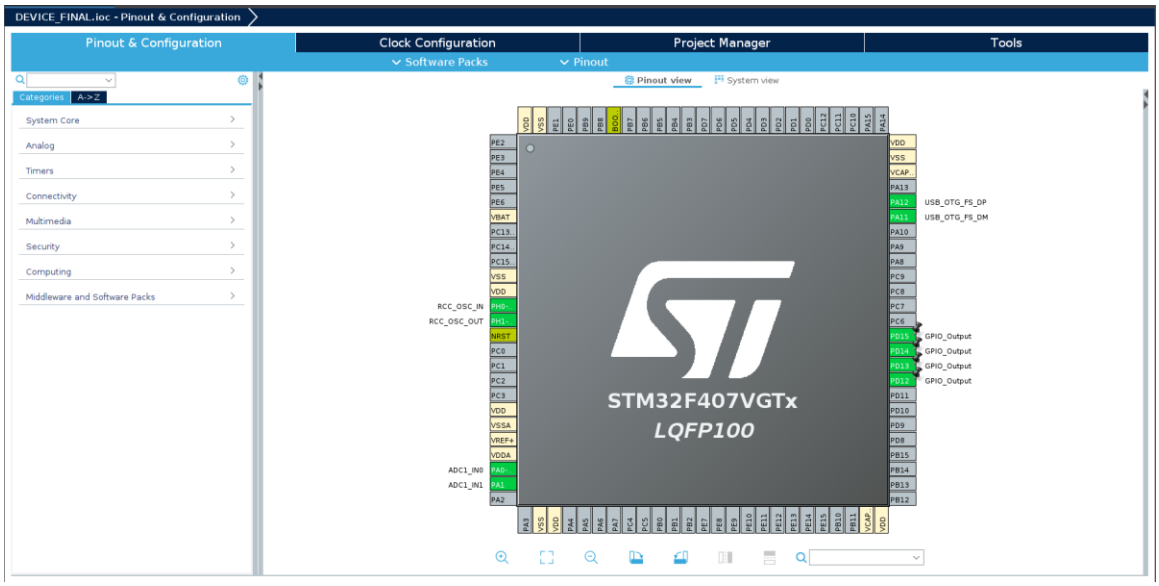
## 6.2 Hardware Setup

The hardware setup involves connecting the LDR and LM35 sensors to the STM32F407 microcontroller. The LDR is connected to ADC Channel 0, and the LM35 is connected to ADC Channel 1. Both sensors are powered by the microcontroller's 3.3V supply, with their output connected to the respective ADC channels.

- **LDR Circuit**: The LDR is connected in series with a resistor to form a voltage divider. The junction between the LDR and the resistor is connected to ADC Channel 0 of the STM32F407.
- **LM35 Circuit**: The LM35 has three pins: Vcc, GND, and Vout. The Vout pin, which provides a voltage proportional to the temperature, is connected to ADC Channel 1.


DEVICE_FINAL.ioc - Pinout & Configuration
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
USB_DEVICE Mode and Configuration
Mode
Class For HS IP
Disable
Class For FS IP
Communication Device Class (Virtual Port Com)
Configuration
Reset Configuration
Parameter Settings
Device Descriptor
User Constants
Configure the below parameters :
Search (Ctrl+F)
Device Descriptor
VID (Vendor IDentifier) 1155
LANGID_STRING (Language ... English(United States)
MANUFACTURER_STRING (... STMicroelectronics
Device Descriptor FS
PID (Product IDentifier) 22336
PRODUCT_STRING (Product ... STM32 Virtual ComPort
CONFIGURATION_STRING (C... CDC Config
INTERFACE_STRING (Interfa... CDC Interface
Pinout view
System view
STM32F407VGTx
LQFP100
RCC_OSC_IN
RCC_OSC_OUT
ADC1_IN0
ADC1_IN1


DEVICE_FINAL.ioc - Pinout & Configuration
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
USB_OTG_FS Mode and Configuration
Mode
Mode
Device_Only
Activate_SOF
Activate_VBUS
Configuration
Reset Configuration
User Constants
NVIC Settings
GPIO Settings
Parameter Settings
Configure the below parameters :
Search (Ctrl+F)
Speed Device Full Speed 12MBit/s
Low power Disabled
Link Power Management Disabled
VBUS sensing Disabled
Signal start of frame Disabled
Pinout view
System view
STM32F407VGTx
LQFP100
RCC_OSC_IN
RCC_OSC_OUT
ADC1_IN0
ADC1_IN1




## 6.3 Software Design

The software design consists of configuring the ADC to read the analog values from the LDR and LM35 sensors and then processing and transmitting these values via USB. The STM32 HAL (Hardware Abstraction Layer) library is used for ADC configuration, GPIO

```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

      HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_15);
      if(flag == 1){
          ADC_Select_CH0();
          HAL_ADC_Start(&hadc1);
          HAL_ADC_PollForConversion(&hadc1, 1000);
          ADC_Val[0] = HAL_ADC_GetValue(&hadc1);
          HAL_ADC_Stop(&hadc1);
          sprintf(str,"LDR  = %d \r\n",ADC_Val[0]);
          CDC_Transmit_FS((uint8_t *)str, strlen(str));
          flag = 0;
      }
      else if(flag == 2){
          ADC_Select_CH1();
          HAL_ADC_Start(&hadc1);
          HAL_ADC_PollForConversion(&hadc1, 1000);
          ADC_Val[1] = HAL_ADC_GetValue(&hadc1);
          ADC_Val[1] = ADC_Val[1]/10.24;
          HAL_ADC_Stop(&hadc1);
          sprintf(str,"LM35  = %d \r\n",ADC_Val[1]);
          CDC_Transmit_FS((uint8_t *)str, strlen(str));
          flag = 0;
      }
      else if(flag == 3){
          volatile uint16_t cnt=0;
          while(cnt<10){
                      HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
                      HAL_Delay(1000);
                      cnt++;
                  }
          flag = 0;
      }

      HAL_Delay(1000);
    /* USER CODE END WHILE */
```

initialization, and USB communication.

### 6.3.1 ADC Configuration

The ADC is initialized and configured to operate in multi-channel mode. The MX_ADC1_Init() function is responsible for setting up the ADC with the following parameters:

- **Clock Prescaler**: ADC clock set to PCLK/4.
- **Resolution**: 12-bit resolution for precise measurement.
- **Scan Mode**: Enabled for scanning multiple channels.
- **Continuous Conversion**: Enabled for continuous data acquisition.
- **Number of Conversions**: Set to 2, corresponding to the LDR and LM35 sensors.

Two functions, ADC_Select_CH0() and ADC_Select_CH1(), are used to select the appropriate ADC channel before starting the conversion process.

### 6.3.2 USB Communication

The data acquired from the ADC is transmitted via USB to a host computer using the USB CDC (Communications Device Class) protocol. The CDC_Transmit_FS() function is used to send the sensor data in a formatted string.

### 6.3.3 Main Loop

The main loop continuously monitors a flag variable to determine which sensor data to acquire and transmit. The flag can take three values:

- **Flag = 1**: The system reads the LDR value from ADC Channel 0, formats it, and sends it over USB.
- **Flag = 2**: The system reads the LM35 value from ADC Channel 1, converts it to temperature, and sends it over USB.
- **Flag = 3**: The system performs a simple LED blinking routine as a placeholder for additional functionality.

The HAL_Delay() function is used to introduce a delay between each operation, ensuring that the system runs smoothly.

```
  * @}
  */

/** @defgroup USBD_DESC_Private_Defines USBD_DESC_Private_Defines
  * @brief Private defines.
  * @{
  */

#define USBD_VID     0x0483
#define USBD_LANGID_STRING     1033
#define USBD_MANUFACTURER_STRING     "STMicroelectronics"
#define USBD_PID_FS     0xc1b0
#define USBD_PRODUCT_STRING_FS     "SUNBEAM PG-DESD MAR24-LINUX USB DEVICE DRIVER USING STM32"
#define USBD_CONFIGURATION_STRING_FS     "CDC Config"
#define USBD_INTERFACE_STRING_FS     "CDC Interface"

#define USB_SIZ_BOS_DESC            0x0C

/* USER CODE BEGIN PRIVATE_DEFINES */

/* USER CODE END PRIVATE_DEFINES */

/**
  * @}
  */

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/** @defgroup USBD_DESC_Private_Macros USBD_DESC_Private_Macros
  * @brief Private macros.
  * @{
  */

/* USER CODE BEGIN PRIVATE_MACRO */

/* USER CODE END PRIVATE_MACRO */

/**
  * @}
  */

/** @defgroup USBD_DESC_Private_FunctionPrototypes USBD_DESC_Private_FunctionPrototypes
  * @brief Private functions declaration.
```

## 6.4 Implementation Details

The implementation details include the following steps:

1. **Initialization**:
    - The system clock is configured for optimal performance.
    - GPIO pins are initialized for LED control.
    - The ADC is configured and initialized for multi-channel operation.
    - USB communication is set up using the MX_USB_DEVICE_Init() function.

2. **Sensor Data Acquisition**:

    - The LDR data is acquired by selecting ADC Channel 0 and starting the conversion process.
    - The LM35 data is acquired similarly by selecting ADC Channel 1.

3. **Data Processing**:

    - The LDR value is directly read and transmitted.
    - The LM35 value is scaled to obtain the temperature in degrees Celsius.

4. **USB Data Transmission**:

    - The acquired data is formatted into a string and transmitted via USB to the host computer.

```
/**
  * @brief  Data received over USB OUT endpoint are sent over CDC interface
  *         through this function.
  *
  *         @note
  *         This function will issue a NAK packet on any OUT packet received on
  *         USB endpoint until exiting this function. If you exit this function
  *         before transfer is complete on CDC interface (ie. using DMA controller)
  *         it will result in receiving more data while previous ones are still
  *         not sent.
  *
  * @param  Buf: Buffer of data to be received
  * @param  Len: Number of data received (in bytes)
  * @retval Result of the operation: USBD_OK if all operations are OK else USBD_FAIL
  */
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
{
  /* USER CODE BEGIN 6 */

    char ptr[100];
  USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
  USBD_CDC_ReceivePacket(&hUsbDeviceFS);
  //strcpy(ptr,(char *)Buf);
  //memcpy(ptr,Buf,*Len);
  strncpy(ptr, (char *)Buf, *Len);
    ptr[*Len] = '\0';
  uint32_t choice = atoi(ptr);
      if(choice == 1){
          flag = 1;
      }
      else if(choice == 2){
          flag = 2;
      }
      else if(choice == 3){
          flag = 3;
      }
      else
          flag = 4;


  return (USBD_OK);
  /* USER CODE END 6 */
}
```

**n**
**d Solutions**

During the implementation, several challenges were encountered, including:

- **ADC Configuration**: Ensuring the ADC was correctly configured for multi-channel operation required careful selection of parameters.
- **USB Communication**: Managing USB communication required attention to buffer management and ensuring data integrity during transmission.

These challenges were addressed through iterative testing and debugging, leading to a robust implementation.

### 6.6 Conclusion

The implementation of LDR and LM35 sensors using the STM32F407 microcontroller was successfully completed. The multi-channel ADC configuration allowed simultaneous monitoring of light intensity and temperature, and the USB communication facilitated easy data transfer to a host system. This project demonstrates the practical application of embedded systems in sensor interfacing and data acquisition.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{

    HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_15);
    if(flag == 1){
        ADC_Select_CH0();
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, 1000);
        ADC_Val[0] = HAL_ADC_GetValue(&hadc1);
        HAL_ADC_Stop(&hadc1);
        sprintf(str,"LDR  = %d \r\n",ADC_Val[0]);
        CDC_Transmit_FS((uint8_t *)str, strlen(str));
        flag = 0;
    }
    else if(flag == 2){
        ADC_Select_CH1();
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, 1000);
        ADC_Val[1] = HAL_ADC_GetValue(&hadc1);
        ADC_Val[1] = ADC_Val[1]/10.24;
        HAL_ADC_Stop(&hadc1);
        sprintf(str,"LM35  = %d \r\n",ADC_Val[1]);
        CDC_Transmit_FS((uint8_t *)str, strlen(str));
        flag = 0;
    }
    else if(flag == 3){
        volatile uint16_t cnt=0;
        while(cnt<10){
                    HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
                    HAL_Delay(1000);
                    cnt++;
                }
        flag = 0;
    }

    HAL_Delay(1000);
  /* USER CODE END WHILE */
```

## *7.References*

**1. Yiu, J. (2015). *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors* (3rd ed.). Newnes.**

**2. STMicroelectronics. (2024). *STM32F407VG Microcontroller Datasheet*. Available at: [STMicroelectronics](https://www.st.com/resource/en/datasheet/stm32f407vg.pdf).**

**3. STMicroelectronics. (2024). *AN2834: Using the STM32F4 ADC in multi-channel mode*. Available at: [STMicroelectronics](https://www.st.com/resource/en/application_note/an2834-using-the-stm32f4-adc-in-multichannel-mode-stmicroelectronics.pdf).**

**4. Bianchi, V., Bassoli, M., Lombardo, G., & Munari, I. (2019). "Environmental Monitoring System for Smart Cities Based on STM32 and LoRa Network." *IEEE Sensors Journal*, 19(16), 7551-7558.**

**5. Embedded Systems Academy. (2024). "Getting Started with STM32 ADC – Part 1." Available at: [Embedded Systems Academy](https://www.esacademy.com/en/getting-started-with-stm32-adc-part1.html).**