1. **Abstract**

Modern vehicles use the Electronics Control Unit (ECU) to control and monitor all the activities within the vehicle. The number of ECUs are increasing as the complexity of vehicles increases. All the ECUs present in the vehicles are communicated with each other via CAN protocol. Any malfunction in the ECU or abnormal behaviour of ECU is detected or understood by diagnostic services. CAN Protocol does not have advanced features like Diagnostic. The CAN protocol covers only the Physical and Data link layer of the OSI model. There is a need for a standardised diagnostic protocol which can use CAN as underlying technology. Standardised diagnostic protocols used in the automotive domain are On Board Diagnostics (OBD) and Unified Diagnostic services (UDS). UDS protocol is defined under the ISO 14229 standard and provides a standardized framework for in-vehicle communication and fault diagnosis. This project focuses on the **design and implementation of the UDS protocol** on an embedded system using **STM32F407 microcontrollers**. The project involves developing a diagnostic communication system between two Electronic Control Units (ECUs), a **Body Control Module (BCM)** and a **Light Control Module (LCM)** connected over a **CAN bus**.

**Keyword:** Unified Diagnostic Services (UDS),ISO 14229,Electronic Control Unit (ECU),Body Control Module (BCM),Light Control Module (LCM), Controller Area Network (CAN),Diagnostic Trouble Code (DTC),Routine Control Identifier(RID) , Data Identifier (DID),ISO-TP (Transport Protocol)

1. **Introduction**

In modern automotive systems, the ability to diagnose and troubleshoot Electronic Control Units (ECUs) efficiently is critical for ensuring vehicle reliability and performance. The **UDS** protocol, defined by **ISO 14229**, is widely adopted in the automotive industry to facilitate standardized communication between diagnostic tools and vehicle ECUs over the **CAN network**. UDS enables functions such as **fault detection, software updates, parameter tuning, and remote ECU diagnostics**, making it an essential part of vehicle maintenance and repair processes.

This paper presents the **design and implementation of a UDS-based diagnostic communication system using STM32F4 microcontrollers**. The project involves two distinct ECUs **Body Control Module (BCM)** and **Light Control Module (LCM)** that communicate via **CAN bus** and respond to UDS diagnostic requests. The system supports essential UDS services, including **Read Data by Identifier (0x22), Write Data by Identifier (0x2E), Routine Control (0x31), ECU Reset (0x11), and Read DTC Information (0x19)**.

To validate the robustness of the system, **fault injection techniques** are utilized, simulating real-world ECU failures such as **Overvoltage and Under-Voltage conditions,** **CAN bus disconnection, ECU power loss, and LED circuit failures**. The diagnostic response to these failures is analyzed using **a Waveshare USB-to-CAN module and Python-based UDS test scripts**. By implementing UDS on embedded automotive ECUs, this project aims to **demonstrate the practical application of standardized diagnostic services in real-world vehicle systems, improving the efficiency of vehicle fault detection and maintenance processes**.

1. **System Overview**

**UDS Protocol Overview**

Unified Diagnostic Services (UDS) is a high-level diagnostic communication protocol defined by the ISO 14229 standard. It is extensively used in the automotive industry to facilitate standardized communication between diagnostic tools and vehicle Electronic Control Units (ECUs). UDS operates over the Controller Area Network (CAN), which serves as the underlying low-layer protocol, providing the physical and data link layers necessary for communication.

**Client Server Architecture**

UDS protocol follows Client-Server architecture. In Client-Server architecture diagnostic tool acts as the client, and the ECU acts as the server. The client initiates diagnostic requests. client sends commands to the ECU for actions like reading data, clearing diagnostic trouble codes (DTCs), or performing ECU reprogramming. The client is typically a diagnostic scanner or testing equipment. The server, which is the Electronic Control Unit (ECU), processes the client’s requests and responds accordingly. It provides data or executes actions as specified by the client’s UDS commands, such as responding with vehicle status or resetting the ECU.

**UDS Message Structure**

UDS protocol is request Based protocol. The fig shows UDS messages structure.

A diagram of a message structure

AI-generated content may be incorrect.

CAN ID: The CAN ID is a unique identifier used in the CAN protocol to distinguish between different messages on the network. In UDS, CAN ID used to identify the source and destination of diagnostic requests and responses. CAN IDs can be either standard (11-bit) or extended (29-bit), depending on the network configuration.

**Protocol Control Information (PCI):** The PCI ﬁeld can be 1-3 bytes long and contains information related to the transmission of messages that do not ﬁt within a single CAN frame. PCI helps with message type identification, segmentation and reassembly, flow control, and sequence numbering.

**Service Identifier (SID)**: The SID is a unique code that specifies the type of diagnostic service being requested. Each service has a predefined SID, such as 0x22 for Read Data by Identifier or 0x2E for Write Data by Identifier. The SID is crucial for the ECU to understand the nature of the request.

**Sub-Function Byte:** The subfunction byte is a single byte value that follows the SID in the UDS message structure. It indicates specific actions or modes within the main service. For example, in the Routine Control service (0x31), the subfunction byte can specify different routines to be executed. In case the response is positive, the tester may want to suppress the response (as it may be irrelevant). This is done by setting the 1st bit to 1 in the sub function byte. Negative responses cannot be suppressed.

**Data Identifier (DID)**: The DID identifies specific data elements within the ECU that are being accessed or modified. For example, a DID might refer to a particular sensor reading or configuration parameter. The DID ensures that the correct data is targeted by the diagnostic service.

**Data Parameters:**  Data Parameters are additional data elements required for the service. Parameters can include values to be written, conditions to be met, or specific instructions for the ECU. They provide the necessary context for the SID and DID to perform the requested operation.

**UDS Response**

In UDS protocol client trigger request and server will respond to this request as per configuration. The response are of two type Positive response and Negative response.

**Positive response**

A positive response indicates that the requested diagnostic service has been successfully executed by the ECU. The structure of a positive response typically mirrors the request, including the Service Identifier (SID) with an added offset of 0x40 to distinguish it as a response. For example, if the request SID is 0x22 (Read Data by Identifier), the positive response SID will be 0x62. Positive responses may also include additional data, such as the requested information or confirmation of the action performed.

**Negative response**

A negative response indicates that there was an issue with the requested diagnostic service. Negative responses include a Negative Response Code (NRC) that specifies the type of error encountered. The structure of a negative response includes the original SID, followed by the NRC. Common NRCs include

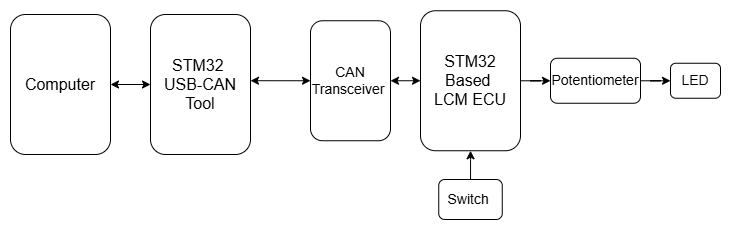
**0x10 (General Reject):** The request was rejected for a general reason not covered by other NRCs.

**0x11 (Service Not Supported):** The requested service is not supported by the ECU.

**0x12 (Subfunction Not Supported):** The specified subfunction is not supported by the ECU.

**0x13 (Incorrect Message Length or Invalid Format):** The message length or format is incorrect.

**Block Diagram**

The block diagram illustrates the architecture of the diagnostic communication system implemented using the UDS protocol on STM32 microcontrollers. The system consists of several key components connected in sequence to facilitate efficient communication and control.

**Computer**: The computer acts as the diagnostic tool. Waveshare USB-CAN-A software tool is installed on computer. Computer is interfacing with the STM32 USB-CAN tool. It sends diagnostic requests and receives responses, enabling the monitoring and control of the ECUs.

**Waveshare USB-CAN-A Tool:** This tool serves as the interface between the computer and the CAN network. It converts USB signals from the computer into CAN signals and vice versa, allowing seamless communication between the diagnostic tool and the ECUs. The Waveshare USB-CAN-A supports various CAN baud rates and provides a reliable connection for diagnostic operations.

**CAN Transceiver**: The CAN transceiver is responsible for transmitting and receiving CAN signals between the Waveshare USB-CAN-A tool and the ECUs. It ensures reliable data transmission over the CAN bus.

**STM32 Based LCM ECU**: The Light Control Module (LCM) ECU is based on the STM32 microcontroller. It processes diagnostic requests related to lighting control and responds accordingly. The LCM ECU is connected to input and output devices to perform its functions.

**Potentiometer**:  The potentiometer is connected between the LED and an output GPIO pin of the STM32 Based LCM ECU. It is used to simulate conditions such as under-voltage and over-voltage to create faulty conditions

**LED**: The LED is connected to GPIO Pin of ECU via Potentiometer. It provides visual feedback based on the potentiometer's settings, allowing for real-time monitoring of light control and fault

**Switch**: The switch is connected to the STM32 Based LCM ECU and serves as an input device. It is used to turn on and off LED.

**Software Component**

**STM32CubeIDE:** STM32CubeIDE is an integrated development environment (IDE) provided by STMicroelectronics. It is Eclipse-based development environment, offering a comprehensive platform for developing, debugging, and testing applications on STM32 microcontrollers. STM32CubeIDE supports code generation, project management, and debugging features.

**STM32CubeMX:** STM32CubeMX is a graphical tool that simplifies the configuration and initialization of STM32 microcontrollers. It allows developers to configure peripherals, middleware, and pin assignments through an intuitive interface. STM32CubeMX generates initialization code that can be directly used in STM32CubeIDE, streamlining the development process.

**Waveshare** USB-CAN-A: It is software tool by Waveshare. This tool is used to send and receive the CAN message on computer. This tool can configure baud rate of CAN protocol. This is the interface for UDS request and Response.

**Working and Flowchart**

1. **Read Data by Identifier (0x22)**

**0xF100: Status of LED**

**0xF200: Software Version Number**

1. **Write Data by Identifier (0x2E)**

**0xF200: Software Version Number**

1. **Routine Control (0x31)**

**Blinking LED with 1 sec Delay**

1. **Read DTC (0x19)**

**Read stored DTC from Memory**

1. **Clear DTC (0x14)**

**Clear DTC from Memory**

**Expected Result:**

1. Read Status of LED

The current status of LED can be retrieved using service read Data by Identifier (0x22) and DID (0xF100)

|  |  |  |  |
| --- | --- | --- | --- |
| Request | 0x22 | 0xF1 | 0x00 |
| Interpretation | Read Data by Identifier (SID) | LED status (DID 1st Byte) | DID 2nd Byte |

| **Response** |
| --- |
| Positive | 0x62 | 0xF1 | 0x00 | 0x01/0x02 |
| Interpretation | SID + 0x40 | DID 1st Byte | DID 2nd Byte | status Data |
|  | | | | |
| Negative | 0x7F | 0x22 | 0x10 |  |
| Interpretation | NRC SID | SID | General Reject NRC |  |

1. Read Software Version Number

The current software version number can be retrieved using Service Read Data by Identifier (0x22) and DID (0xF200)

|  |  |  |  |
| --- | --- | --- | --- |
| Request | 0x22 | 0xF2 | 0x00 |
| Interpretation | Read Data by Identifier (SID) | SW Version Number (DID 1st Byte) | DID 2nd Byte |

| **Response** |
| --- |
| Positive | 0x62 | 0xF2 | 0x00 | 0xA1 |
| Interpretation | SID + 0x40 | DID 1st Byte | DID 2nd Byte | SW Version Number |
|  | | | | |
| Negative | 0x7F | 0x22 | 0x10 |  |
| Interpretation | NRC SID | SID | General Reject NRC |  |

1. Write Software Version number

The updated SW version number can be written into the Software using Write Data by identifier (0x2E) and DID (0xF200)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Request | 0x2E | 0xF2 | 0x00 | 0xA2 |
| Interpretation | Read Data by Identifier (SID) | SW Version Number (DID 1st Byte) | DID 2nd Byte | New SW version Number |

| **Response** |
| --- |
| Positive | 0x6E | 0xF2 | 0x00 | 0xA1 |
| Interpretation | SID + 0x40 | DID 1st Byte | DID 2nd Byte | SW Version Number |
|  | | | | |
| Negative | 0x7F | 0x2E | 0x10 |  |
| Interpretation | NRC SID | SID | General Reject NRC |  |

1. Blinking LED routine for Testing

To check weather the LED is working correct or not dedicated blinking led routine can be used with Routine control SID (0x31).

To start routine sub–Function SID (0x01) and for stop routine sub function SID (0x02) is used. For Blinking led with 1 sec delay routine is used 0x2020.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Request | 0x31 | 0x01/0x02 | 0xB1 | 0xB1 |
| Interpretation | Routine Control (SID) | Start/ Stop Routine (Sub Function SID) | RID 1st Byte | RID 2nd Byte |

| **Response** |
| --- |
| Positive | 0x71 | 0x01/0x02 | 0xB1 | 0xB1 |
| Interpretation | SID + 0x40 | Start/Stop Routine (Sub Function SID) | RID 1st Byte | RID 2nd Byte |
|  | | | | |
| Negative | 0x7F | 0x31 | 0x10 |  |
| Interpretation | NRC SID | SID | General Reject NRC |  |

1. Read DTC to detect Fault

The fault occurred for over voltage and under voltage are stored in system with unique number called as Diagnostic Trouble Code (DTC). The fault in system can be understood by reading the stored DTCs using Read DTC service (0x19) and Sub Functional SID DTC by Status Mask (0x01) and status mask value (0xFF). This request will read all the DTC regardless of Status.

|  |  |  |  |
| --- | --- | --- | --- |
| Request | 0x19 | 0x01 | 0xFF |
| Interpretation | Read DTC (SID) | Read DTC by Status Mask (Sub Function SID) | Status Mask Value |

| **Response** |
| --- |
| Positive | 0x59 | 0x01 | 0xA0 | 0xB0 |
| Interpretation | SID + 0x40 | Read DTC by Status Mask (Sub Function SID) | Undervoltage DTC | Overvoltage DTC |
|  | | | | |
| Negative | 0x7F | 0x19 | 0x10 |  |
| Interpretation | NRC SID | SID | General Reject NRC |  |

1. **Clear DTC to Resolve Fault**

Once the faults have been diagnosed and resolved, it is essential to clear these stored DTCs to reset the system and ensure accurate future diagnostics. The Clear DTC service (0x14) is used for this purpose.

|  |  |  |
| --- | --- | --- |
| Request | 0x14 | 0xFF |
| Interpretation | Clear DTC (SID) | Read DTC by Status Mask (Sub Function SID) |

| **Response** |
| --- |
| Positive | 0x54 |
| Interpretation | SID + 0x40 |
|  | | | | |
| Negative | 0x7F | 0x14 | 0x10 |  |
| Interpretation | NRC SID | SID | General Reject NRC |  |