CAN Frame Sniffer with Automatic Baud Rate Detection and Configurable Filtering

Stefanos Ziakas

Abstract

This project presents a CAN Frame Sniffer, built on the NUCLEO-H755ZI-Q board, implemented as a C library for easy reuse in other projects. It supports automatic baud-rate detection or manual configuration via a user menu and allows CAN ID message filtering. The sniffer was successfully tested on a vehicle's OBD-II port, capturing live CAN data.

Contents

1 Introduction

This section briefly introduces the CAN protocol and the purpose of the CAN Frame Sniffer.

1.1 Controller Area Network (CAN)

Controller Area Network (CAN) is a robust communication protocol widely used in automotive and industrial systems. It allows multiple Electronic Control Units (ECUs) to exchange data reliably over a shared bus without requiring a central host computer. CAN is designed for real-time control and is highly resilient to electrical noise.

1.2 CAN Frame Sniffer

The CAN Frame Sniffer is a tool designed to monitor and analyze CAN bus traffic in real time. It connects directly to the CAN network and captures the data frames exchanged between nodes. Such tools are commonly used for debugging, diagnostic checks, protocol analysis, and more.

2 CAN Protocol Overview

This section provides key definitions and operational concepts of the CAN protocol, along with the specific configurations used in this project. For a more detailed explanation of the protocol, refer to the listed references.

2.1 Classical CAN vs. CAN FD

Classical CAN, the original protocol, supports data frames up to 8 bytes and bit rates up to 1 Mbps. CAN FD (Flexible Data-Rate) extends this with frames up to 64 bytes and higher transmission speeds. This project focuses on Classical CAN.

2.2 Standard vs. Extended Frames

CAN messages can use either a Standard or Extended frame format. Standard format uses an 11-bit identifier, while Extended format uses a 29-bit identifier, allowing for a larger number of unique message IDs. This project utilizes the standard frame format.

2.3 Bit Timing Parameters and Calculation Formulas

Bit timing in CAN defines how the nodes on the bus synchronize and interpret each bit, ensuring reliable communication even in the presence of delays and electrical noise. Each bit on the CAN bus is divided into time quanta (TQ), which are the smallest units of time used for sampling and synchronization.

The total number of TQs per bit is determined by the sum of the following segments:

- **Synchronization Segment (SyncSeg):** The first segment of each bit, marks the beginning of the bit and is used to synchronize the node on a signal edge. This segment is always 1 TQ.

- **Propagation Segment (PropSeg):** Compensates for physical propagation delay across the bus. Its value depends on cable length and transceiver characteristics.
- **Phase Segment 1 (PhaseSeg1):** Can be lengthened or shortened during resynchronization.
- **Phase Segment 2 (PhaseSeg2):** Can also be adjusted during resynchronization.

Therefore, the total number of Time Quanta per bit is:

$$N_{\text{TQ}} = 1 + \text{PropSeg} + \text{PhaseSeg1} + \text{PhaseSeg2}.$$

The nominal bit rate is determined by the CAN controller clock frequency f_{clk}, the prescaler, and the number of TQ per bit:

$$\text{Baud Rate} = \frac{f_{\text{clk}}}{\text{Prescaler} \times N_{\text{TQ}}}.$$

When the propagation segment is assumed to be zero (e.g., for short physical distances), this reduces to:

$$\text{Baud Rate} = \frac{f_{\text{clk}}}{\text{Prescaler}\,(1 + \text{PS1} + \text{PS2})}.$$

Based on the above segments we can also calculate the Sample Point. The sample point is the time within the bit at which the bus level is read. It occurs at the end of the SyncSeg + PS1 interval:

$$\text{Sample Point} = \frac{\text{SyncSeg} + \text{PS1}}{1 + \text{PropSeg} + \text{PS1} + \text{PS2}}.$$

With SyncSeg = 1 and PropSeg = 0, this simplifies to:

$$\text{Sample Point} = \frac{1 + \text{PS1}}{1 + \text{PS1} + \text{PS2}}.$$

Selecting appropriate values for these parameters is critical to ensure proper synchronization across all nodes on the network and to meet the required communication speed. In this project, the bit-timing configurations were derived assuming a 40 MHz FDCAN peripheral clock and a short physical bus length, where propagation delays are negligible. Under these conditions, the propagation segment was omitted from the calculations, and the timing values were chosen to maintain a nominal sample point of 87.5% for most baud rates.

The resulting configurations are summarized in Table 2.3. These values have been experimentally verified and provide stable communication for the target setup.
These timing values probably will not be valid if:

- the bus length or physical characteristics differ from the assumed setup,
- a different sample point is desired,
- the peripheral clock frequency changes, or
- the number of time quanta per bit is reduced to lower internal hardware load.

In any such case, the bit-timing parameters must be recalculated.

Baud Rate (bps)	Prescaler	TSEG1	TSEG2
5 000	200	34	5
10 000	100	34	5
20 000	50	34	5
50 000	20	34	5
100 000	10	34	5
125 000	8	34	5
200 000	5	34	5
250 000	4	34	5
400 000	4	19	5
500 000	2	34	5
800 000	2	19	5
1 000 000	1	34	5

Table 1: Supported CAN Bit-Timing Configurations

2.4 Filters and Masking

CAN controllers use acceptance filters and masks to reduce the processing load by ignoring messages that are not relevant to the node. A filter selects the message identifiers that the CAN controller will accept, and the mask indicates which bits of the identifier must match the filter for a message to be received. In this project, the setup supports a single filter and a single mask.

3 Hardware Setup

This section describes the hardware components used by the CAN Frame Sniffer and their interconnections. The system consists of a NUCLEO-H755ZI-Q microcontroller board, a Waveshare SN65HVD230 CAN transceiver module, a Waveshare USB-CAN-A adapter, and a PC. Figure 1 shows the overall hardware setup.

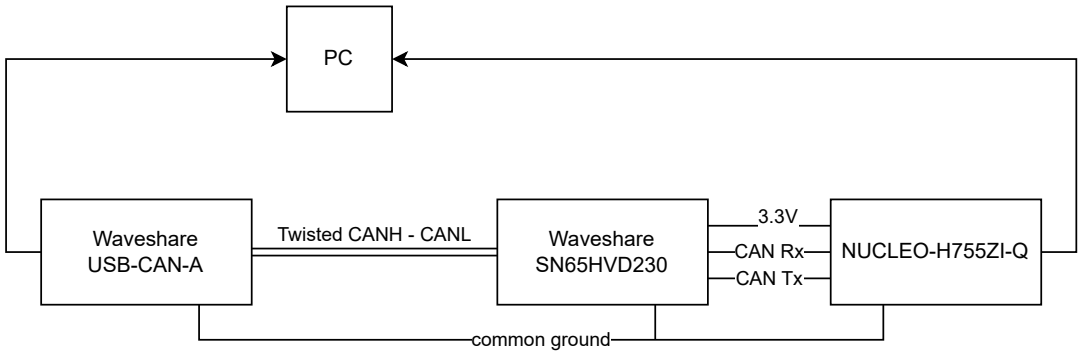

Figure 1: Hardware Setup of the CAN Frame Sniffer.

3.1 System Overview

The NUCLEO-H755ZI-Q board acts as the core processing unit of the system. It communicates with the CAN network through the SN65HVD230 transceiver, which handles the physical layer conversion between the differential CAN bus signals (CAN_H and CAN_L) and the MCU's logic-level CAN RX/TX pins. Captured CAN frames are transmitted to the PC via UART over the same USB interface used for programming the board, allowing analysis and visualization. The Waveshare USB-CAN-A adapter, connected to the same CAN bus, functions as a transmitting node controlled by the PC, that sends CAN frames onto the bus to be detected by the sniffer.

3.2 NUCLEO-H755ZI-Q Development Board

The NUCLEO-H755ZI-Q is a dual-core STM32H7-based development board that provides high processing performance and flexible connectivity options.
In this project, the FDCAN1 peripheral operates in Classical CAN, Bus Monitoring Mode. The USART3 peripheral operates Asynchronously and is responsible for transmitting data to the PC.

3.3 Waveshare SN65HVD230 CAN Transceiver Module

The SN65HVD230 is a 3.3V CAN transceiver based on the TI SN65HVD230 chip. It supports the transmission of both Standard (11-bit) and Extended (29-bit) Classical CAN frames at speeds up to 1 Mbps.

3.4 Waveshare USB-CAN-A Interface Adapter

The USB-CAN-A adapter serves as a bridge between the CAN bus and the PC. It provides a USB interface for monitoring and transmitting CAN frames using dedicated software. The adapter supports bit rates from 5 kbps to 1 Mbps and again both Standard and Extended Classical CAN frames.

3.5 Hardware Interconnections

The USB-CAN adapter, the CAN transceiver module and the NUCLEO microcontroller are connected as follows:

USB-CAN adapter CAN_L	$<->$	Transceiver CAN_L
USB-CAN adapter CAN_H	$<->$	Transceiver CAN_H
USB-CAN adapter GND	$<->$	NUCLEO GND
Transceiver 3.3V	$<->$	NUCLEO 3.3V
Transceiver CAN_TX	$<->$	NUCLEO PD1
Transceiver CAN_RX	$<->$	NUCLEO PD0
Transceiver GND	$<->$	NUCLEO GND

The CAN bus consists of a twisted pair cable, and both the SN65HVD230 transceiver and the USB-CAN-A adapter include 120 Ω termination resistors to minimize signal reflections and maintain signal integrity. The total bus length in this setup is less than 1 m, which is well within the recommended range for reliable testing at 500 kbps.

4 Software Setup

This section presents the software environment and firmware structure used in the implementation of the CAN Frame Sniffer. While the hardware provides the physical interface to the CAN bus, the software defines the system's behavior by managing data acquisition, communication with the host PC, and user interaction through a serial terminal.

4.1 Development Environment

The CAN Frame Sniffer firmware was developed and tested using a combination of embedded and host-side tools. The complete setup is summarized below:

- **STM32CubeIDE:** Primary Integrated Development Environment (IDE) for writing, compiling, and debugging the firmware via the integrated ST-Link debugger of the NUCLEO-H755ZI-Q board. STM32CubeIDE provides full support for STM32 microcontrollers and integrates STM32CubeMX for peripheral configuration, clock setup, and code generation.
- **Waveshare USB-CAN V2.12:** Host-side software tool for the Waveshare USB-CAN-A adapter. It allows the PC to transmit and receive CAN frames over the connected CAN bus. During testing, the adapter was operated in loopback mode at various baud rates to verify that the sniffer correctly captured and decoded CAN frames.
- **Tera Term:** Serial terminal interface used to communicate with the NUCLEO board via UART. It provides access to the firmware's text-based user menu, allowing the user to configure parameters such as the CAN baud rate and filter masks.

All tools were run on a Windows-based PC environment.

4.2 Firmware Overview

The firmware running on the NUCLEO-H755ZI-Q board is organized into modular components to simplify development, improve readability, and allow future reuse. The main modules include CAN handling, UART communication, and the user interface menu. This architecture uses higher-level logic to wrap and hide register-dependent code, which simplifies debugging, testing, and future expansion. Figure 2 illustrates the firmware structure and the interactions between its modules.

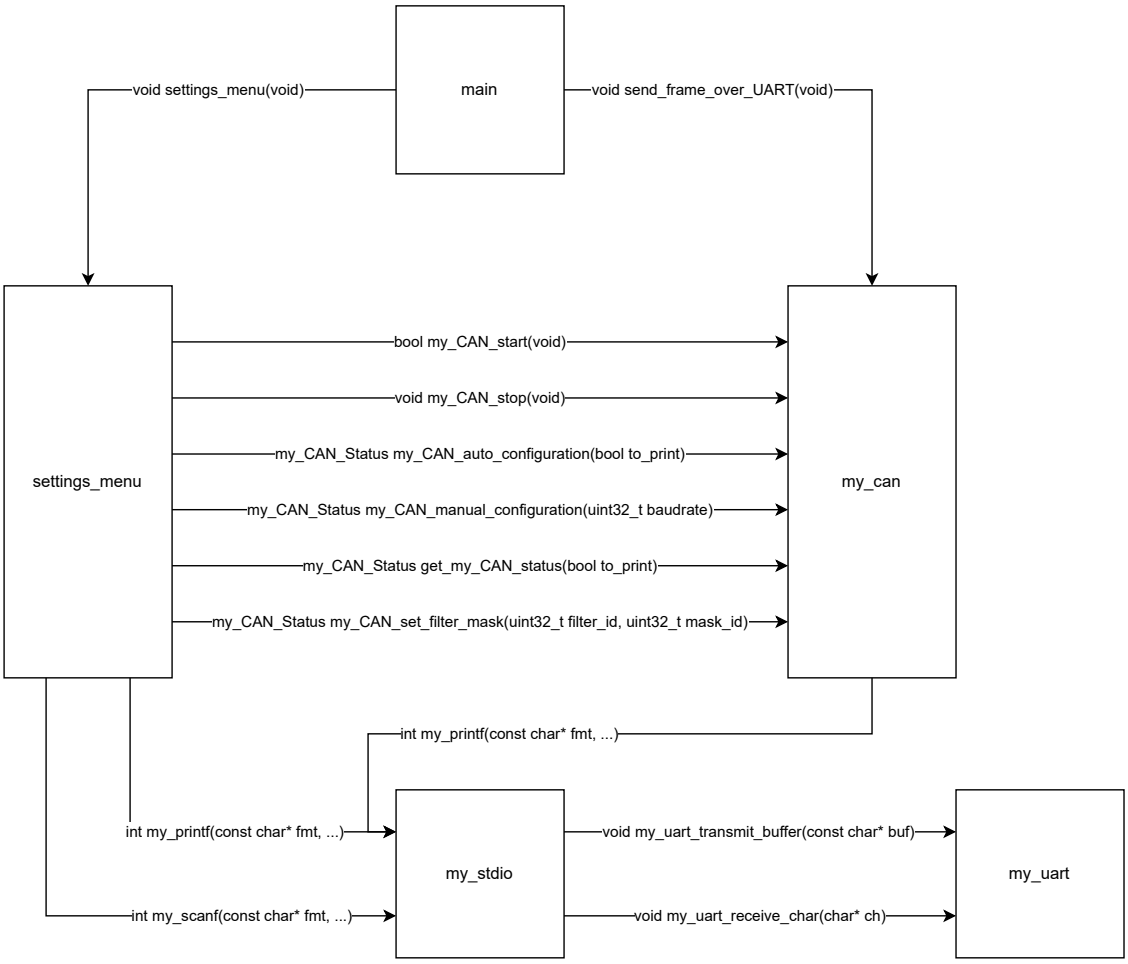

Figure 2: Firmware Structure of the CAN Frame Sniffer.

4.3 CAN Configuration

The CAN module provides all functionality related to the FDCAN1 peripheral. This includes initialization and deinitialization of the CAN interface, configuration of the baud rate either manually or automatically, setup of acceptance filters, and control of CAN communication through start and stop functions. The module operates in interrupt-driven mode, capturing received frames and storing them in a software ring buffer for later processing. An overview of the primary functions provided by this module, along with a brief documentation, follows:

```
/** @brief Configure CAN manually using a requested baud rate. */
my_CAN_Status my_CAN_manual_configuration(uint32_t baudrate);

/** @brief Try all supported baud rates until bus activity is detected. */
my_CAN_Status my_CAN_auto_configuration(bool to_print);

/** @brief Set the filter and mask values. */
my_CAN_Status my_CAN_set_filter_mask(uint32_t filter_id, uint32_t mask_id);
```

```
/** @brief Get the current CAN configuration status. */
my_CAN_Status get_my_CAN_status(bool to_print);

/** @brief Initialize CAN, configure filters, and start the CAN peripheral. */
bool my_CAN_start(void);

/** @brief Stop CAN activity, disable interrupts, and reset software buffers. */
void my_CAN_stop(void);
```

Listing 1: Main functions of CAN Module.

4.4 UART Communication

The UART module is responsible for managing serial communication with the host PC via the USART3 peripheral, which operates over the USB interface provided by the ST-Link debugger. Configured to support standard I/O functions, this module allows the user to interact with the firmware through the terminal menu interface. A summary of the UART and I/O functions, along with brief explanatory comments, is presented below:

```
/** @brief Transmit a null-terminated string buffer over USART3. */
void my_uart_transmit_buffer(const char* buf);

/** @brief Receive a single character from USART3. */
void my_uart_receive_char(char* ch);

/** @brief Formatted UART output similar to printf. */
int my_printf(const char* fmt, ...);

/** @brief Formatted UART input similar to scanf. */
int my_scanf(const char* fmt, ...);

/** @brief Read frames from the software buffer and print them over UART. */
void send_frame_over_UART(void);
```

Listing 2: Main functions of UART Module.

4.5 User Interface

The User Interface module provides a text-based menu accessible via Tera Term or any other serial terminal. This module enables the user to configure the CAN peripheral's baud rate, set frame filter and mask, start the sniffer, and query the current CAN status. The following, outlines the structure and options of the menu interface:

```
/** @brief Prints the settings menu options over UART. */
static void print_menu(void) {
  my_printf("************************************\r\n");
  my_printf("* CAN Sniffer - Settings menu      *\r\n");
  my_printf("*                                  *\r\n");
  my_printf("* a: Auto Configure CAN Baud Rate  *\r\n");
  my_printf("* m: Manual Configure CAN Baud Rate *\r\n");
  my_printf("* s: Set CAN Filter-Mask           *\r\n");
```

```
  my_printf("* g: Get CAN Sniffer status         *\r\n");
  my_printf("* q: Quit and Start CAN Sniffer     *\r\n");
  my_printf("************************************\r\n\n");
}

/** @brief Blocking menu used to configure the CAN sniffer.
  * @detail Manages the configuration of the above options. */
void settings_menu(void);
```

Listing 3: Options of the User Menu Interface.

For brevity, only the key components of the software are shown. The complete source code is available for a detailed view at: CAN-Sniffer.

5 Testing and Validation

This section outlines the methods used to validate the CAN Frame Sniffer. The integrated hardware–firmware system was tested to ensure proper operation of all components, including CAN communication, UART interface, and user interaction via the terminal. Initial bench tests were done by emulating CAN bus traffic using the USB–CAN adapter to verify basic functionality. After that, the system was tested in a real-world automotive environment by connecting it directly to the vehicle's OBD-II port.

5.1 Bench Testing

Initial tests were performed to ensure proper operation of the sniffer under various configurations. The baud rate detection algorithm, filter mask configuration, and UART communication were verified using the USB-CAN adapter as a CAN frame generator that transmitted predefined frames at different baud rates. The firmware correctly identified and adapted to each configuration, while all frames were successfully received, stored, and displayed through the terminal interface.

5.2 Automotive Validation

To evaluate the sniffer in an actual CAN network, the device was connected to a vehicle's OBD-II port (2005 Mitsubishi Colt). Prior to connection, the electrical characteristics of the bus were verified to ensure safe operation. The onboard 120 Ω termination resistor of the transceiver module was desoldered, as the vehicle's network already included two terminating resistors, maintaining the correct overall impedance of approximately 60 Ω. A short CAT 6 cable (less than 1 m) was used to preserve signal integrity, as its twisted-pair design provides controlled impedance and reduces interference. Upon connection, the automatic baud rate detection correctly identified the network speed as 500 kbps, which is the standard for most conventional vehicles. The sniffer was initially configured with no active filters, allowing all CAN frames on the bus to be captured and logged for subsequent analysis.

The interconnection between the OBD-II port of the 2005 Mitsubishi Colt, the CAN transceiver module, and the NUCLEO microcontroller is arranged as follows:

OBD-II port pin14 CAN_L	$<->$	Transceiver CAN_L
OBD-II port pin6 CAN_H	$<->$	Transceiver CAN_H
OBD-II port pin5 GND	$<->$	NUCLEO GND
Transceiver 3.3V	$<->$	NUCLEO 3.3V
Transceiver CAN_TX	$<->$	NUCLEO PD1
Transceiver CAN_RX	$<->$	NUCLEO PD0
Transceiver GND	$<->$	NUCLEO GND

5.3 Log Analysis and Reverse Engineering

Since the CAN Frame Sniffer operates passively and does not implement any OBD-II request/response protocol, it cannot query the vehicle for specific data, such as vehicle or engine speed. Instead, collected log files were analyzed offline to identify relevant frames. Through examination and comparison of CAN messages, the frame corresponding to the vehicle speed was successfully determined.

5.4 Data Visualization

To demonstrate practical use of the captured data, a simple graphical speedometer was implemented in `PyQt`. The speedometer interfaced with the sniffer via the serial connection and displayed the real-time vehicle speed.

6 Knowledge Gained

This project provided several valuable insights and practical skills:

- Acquired foundational knowledge of the CAN protocol, including frame reception, filtering, and bit timing configuration.
- Familiarized with the STM IDE, including project setup, peripheral and clock configuration, and the use of debugging tools.
- Gained hands-on experience in simple embedded system design by writing custom firmware, initializing peripherals, and handling interrupts.
- Valued the importance of studying board and MCU manuals to understand how STM32 predefined header and source files interface with the hardware.
- Performed simple reverse engineering to identify relevant CAN frames in a real vehicle network.
- Recognized the value of modular code design for maintainability, scalability, and ease of use.
- Developed skills in reading schematics, verifying connections, and performing safe electrical measurements to protect components.
- Designed practical testing and validation workflows, from lab bench to real-world automotive experiments.

7 Future Work

Several extensions can be considered to enhance the functionality and applicability of the developed CAN sniffer system:

- **Full OBD-II Protocol Implementation:** Integrate standard OBD-II PID requests and responses to actively query vehicle parameters such as speed, RPM, and fuel level. This would move the system from passive frame monitoring to a comprehensive vehicle diagnostic tool.
- **Support for CAN FD and Extended Frames:** Extend the firmware and hardware capabilities to handle CAN FD frames with data payloads up to 64 bytes and higher transmission speeds, as well as support for extended 29-bit identifiers. This would allow compatibility with modern automotive networks.
- **Embedded Data Visualization:** Implement a real-time graphical dashboard directly on the microcontroller using a display library such as `LVGL`. Techniques such as DMA could be leveraged to reduce CPU load and improve refresh rates for real-time data plotting.
- **Wireless Data Transmission:** Add wireless connectivity (e.g., via an ESP module) to transmit captured CAN data to a remote PC.

These are just a few potential enhancements that could broaden the system's applicability in vehicle diagnostics and monitoring.

References

[1] Texas Instruments, *Introduction to the Controller Area Network (CAN)*, Application Report, August 2002 - Revised May 2016.

[2] KVASER, *The CAN Bus Protocol Tutorial*

[3] NXP, *CAN Bit Timing Requirements*, Application Note

[4] OTIS ISRC PVT LTD, *Computation of CAN Bit Timing Parameters Simplified*, iCC 2012.

[5] Texas Instruments, *SN65HVD230-CAN-Board-Datasheets*

[6] Waveshare, *USB-CAN-A Wiki*

[7] ST, *RM0399 MCU Reference manual*

[8] ST, *H755ZIQ Board Schematic*

[9] ST, *UM2408 Board User manual*

[10] ARM, *Arm v7-M Architecture Reference Manual*

A Appendix

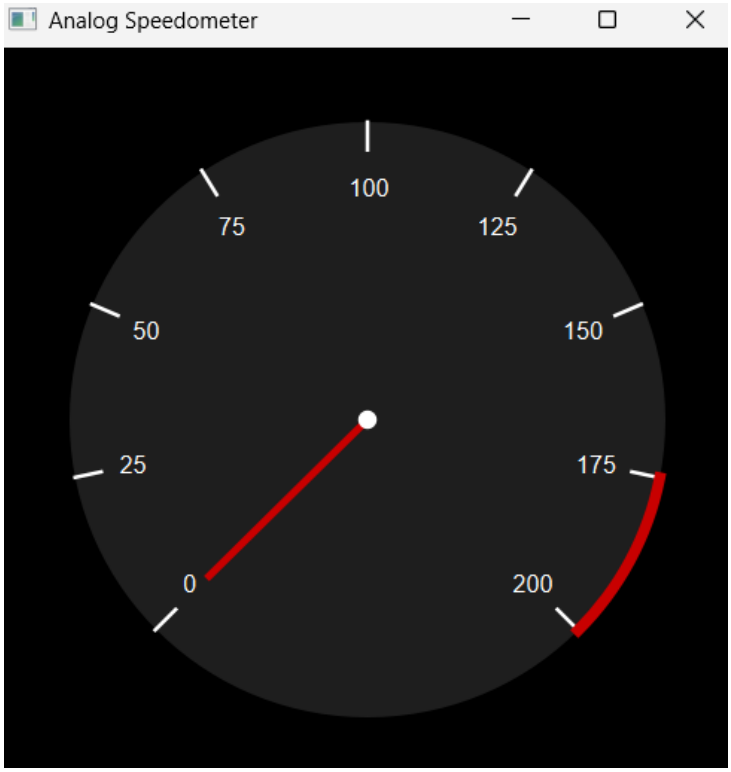

Figure 3: Simple Analog Speedometer used for Data Visualization

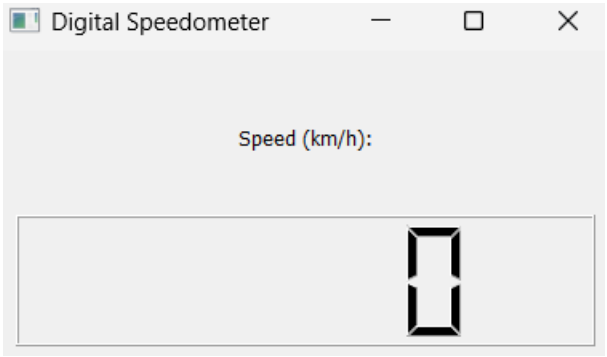

Figure 4: Simple Digital Speedometer used for Data Visualization