Simulation of Communication in Industrial Networks

CAN Network

Student: Mureșan Andrei-Ioan

Structure of Computer Systems Project

Technical University of Cluj-Napoca

January 15, 2025

Contents

Introduction

1.1 Context

The **CAN** (Controller Area Network) communication protocol is widely used in industries such as automotive, aerospace, and industrial automation for robust and efficient communication between various electronic devices. It operates as a multi-master, broadcast communication system, allowing multiple Electronic Control Units (ECUs) to communicate over a shared bus. CAN is favored for its noise immunity, real-time data handling, and fault tolerance, making it an essential part of modern embedded systems. [1]

However, while the **CAN protocol** provides the framework for low-level communication, higher-level protocols like CANopen are necessary to manage more complex operations, such as real-time data exchange, device configuration, and network management. CANopen builds on top of CAN, providing the necessary structure for distributed control systems by organizing data, commands, and system states using standardized messages such as **PDOs**, **SDOs**, and **NMT** commands.

In the automotive industry, for example, ECUs responsible for tasks such as engine control, anti-lock braking, and power steering communicate through the CAN bus. CANopen further organizes this communication by providing a standardized framework to manage real-time data, configuration parameters, and device states.

1.2 Objective

The objective of this project is to develop a simulation of a CAN communication system using **Python** and **PyQt** (UI). The simulation will model the behavior of multiple **ECUs** communicating over a CAN bus, implement message arbitration, and integrate CANopen protocol features for real-time data exchange, network management, and device configuration. The simulated environment will be a basic car.

1.3 Project proposal

This section states what features will be implemented in the final simulation of the CAN protocol. The details of each feature will be presented in future sections.

List 1.3.1: Specific goals

1. Simulating the low-level CAN bus communication protocol, including bit-level arbitration.

2. Implementing the CANopen application layer protocol which defines real-time data exchange via PDOs (Process Data Objects), configuration and diagnostics via SDOs (Service Data Objects), and network management using NMT (Network Management) messages, by the help of the Object Dictionary.

3. Creating a GUI using PyQt that will enable the user to:
    - visualize the message transmission
    - display the state of the bus
    - display the states of the ECU
    - enabling users to interact by sending commands
    - configurating devices and adding new devices

Bibliographic Research

2.1 What is CAN?

CAN (Controller Area Network) is a serial multi-master, message broadcast communication protocol that was developed by BOSCH [1].

Its domain of application ranges from high-speed networks to low-cost multiplex wiring. In automotive electronics, engine control units, sensors, anti-skid-systems, etc. are connected using CAN with bitrates up to 1 Mbit/s. At the same time, it is cost effective to build into vehicle body electronics, e.g. lamp clusters, electric windows etc. to replace the wiring harness otherwise required. [2]

CAN operates at the **Physical Layer** and **Data Link Layer** of the OSI model, managing the transmission of messages and ensuring the integrity of data through mechanisms such as arbitration and error detection.

Remark 2.1.1: CAN vs CANopen

CANopen, is a higher-layer protocol built on top of CAN, providing a framework for how data is organized, exchanged, and managed in a network. It operates at the **Application Layer** and defines how devices interact. So, it is not the same thing as the **CAN** bus protocol, which operates at a lower level. [4]

2.2 The CAN bus

The bus in the CAN protocol refers to the physical connection that links multiple devices (ECUs) together, allowing them to communicate and exchange data. It consists of two wires: **CANH** (CAN High) and **CANL** (CAN Low), which form a twisted-pair cable.

The CAN bus operates using differential signaling, where the data is transmitted by the difference in voltage between the CANH and CANL wires, rather than an absolute voltage level. [2] This differential method makes the system highly resistant to external noise because

any noise that affects both wires equally will be canceled out when the voltage difference is measured.

List 2.2.1: Bus states

- **Dominant State** (Logical 0): When the bus is in the dominant state, CANH is driven to a higher voltage (around 3.5V), while CANL is driven to a lower voltage (around 1.5V). This voltage difference (approximately 2V) represents a dominant bit, which is interpreted as a logical 0. [1]
- **Recessive State** (Logical 1): In the recessive state, both CANH and CANL are at the same voltage (around 2.5V), creating a 0V differential, which is interpreted as a logical 1. [1]

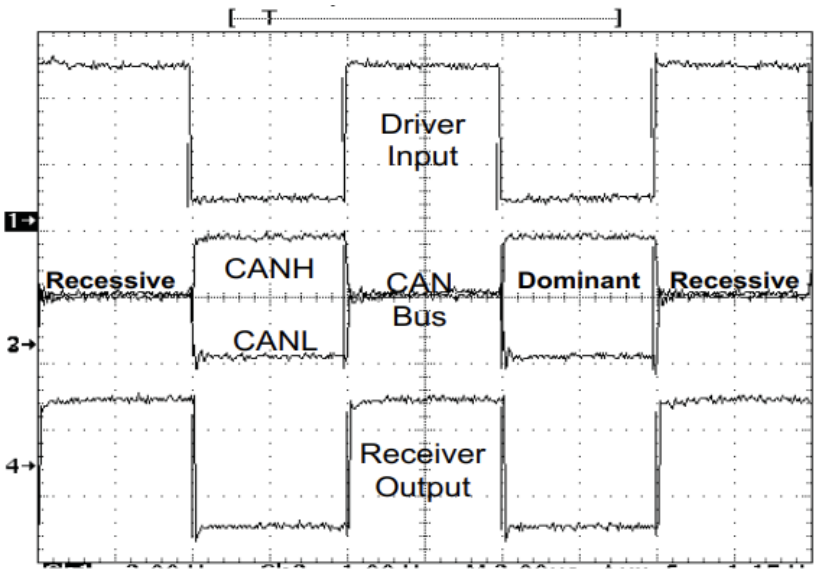

Figure 1. States of the bus under an oscilloscope [1]

The system defaults to the recessive state when no data is being transmitted. When an ECU sends data, it drives the bus into the dominant state for logical 0s, and the other ECUs can detect this change on the bus.

When two ECUs try to write a 1 and a 0 at the same time, the bus will be taken to the dominant state, the 0 winning the bus. This helps in arbitration.

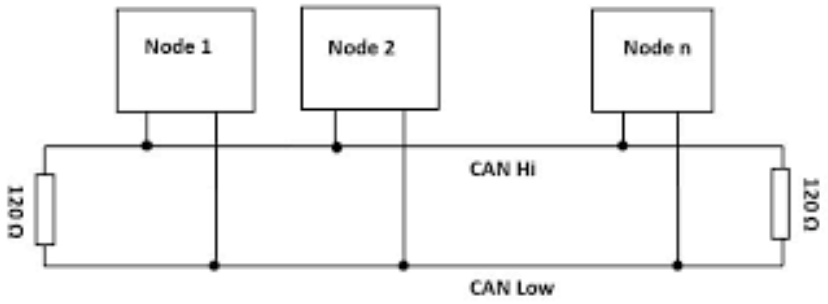

Figure 2. CAN bus [3]

2.3 Transceiver, CAN controller and DSP

A CAN device is made from multiple components, each with specific purposes.

The **CAN transceiver** is the component that interfaces between the **CAN controller** (operating at the Data Link Layer) and the physical bus, operating at the Physical Layer. Its main job is to convert the digital signals from the CAN controller into electrical signals that are transmitted over the CAN bus and to convert electrical signals from the bus back into digital data for the controller to process. Its key functionalities are signal conversion, bus monitoring, error handling and physical isolation. [1]

The **CAN controller** is the core component responsible for managing the **Data Link Layer** of the CAN protocol. It handles all communication-related tasks between the **DSP** (or microcontroller) and the **CAN transceiver**, ensuring that messages are correctly formatted, transmitted, and received according to the CAN protocol. Its key functionalities are message framing, message arbitration, error detection and message buffering. [1]

A **Digital Signal Processor (DSP)** is a specialized microprocessor optimized for performing high-speed mathematical operations, particularly useful in real-time applications. In the context of a CAN system, the DSP typically handles the higher-level processing tasks of an ECU, such as:

- Signal processing for sensors and actuators.
- Control algorithms for systems like engine control, motor drives, or industrial automation.

The **DSP** ensures that these real-time tasks are executed quickly and efficiently, without introducing delays into the system. It can process signals from sensors, apply complex control algorithms, and determine outputs that need to be sent to other devices on the CAN network. Although the DSP does not directly handle CAN message framing or transmission,

it works alongside the **CAN controller** and **transceiver** to ensure smooth and efficient communication with other nodes in the network.

In *Figure 3* we can observe such configuration.

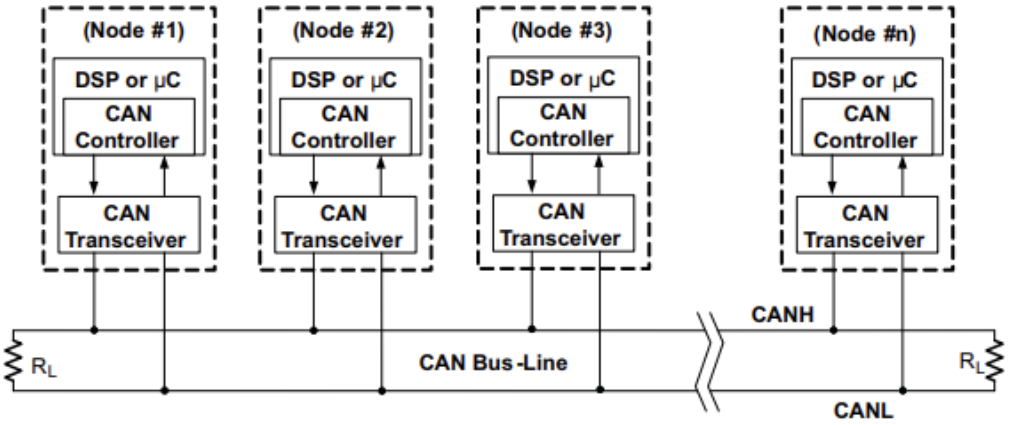

Figure 3. CAN system [1]

2.4 Arbitration

Arbitration in a **CAN** system is the process used to determine which Electronic Control Unit gets to send its message when multiple ECUs try to transmit simultaneously. CAN uses a non-destructive bitwise arbitration mechanism, meaning that the message with the highest priority (lowest **CAN-ID**) always wins without causing data loss.[2]

Remark 2.5.1: Low level explanation

During arbitration, each ECU monitors the bus while transmitting its message. If an ECU detects that another ECU is sending a dominant bit (0) while it is sending a recessive bit (1), it stops transmitting, as the bus has been taken by a higher-priority message. This process ensures that only one message is transmitted at a time and that the message with the highest priority is always sent first.[1]

2.5 CANopen

CANopen is a higher-layer communication protocol built on top of the CAN protocol, designed for embedded systems requiring structured data exchange, configuration, and network management. While CAN handles the low-level transmission of messages, CANopen adds organization and control at the Application Layer. [4]

At the core of CANopen is the **Object Dictionary**, a table that defines how each device's data and settings are structured. It enables standardization across the network, making it easier for devices to communicate and be configured.

CANopen defines two key message types:

- **PDOs** (Process Data Objects): Used for real-time data exchange, allowing devices to send and receive operational data like sensor readings and control commands quickly and efficiently.
- **SDOs** (Service Data Objects): Used for configuration and diagnostics, allowing one device to read or write to another device's Object Dictionary entries, enabling dynamic updates to device settings.

Additionally, NMT (Network Management) messages are used to control the operational states of devices, ensuring synchronized operation within the network.

CANopen is widely used in industries such as automotive, industrial automation, and medical devices, where real-time communication and device configuration are critical. It enhances CAN's capabilities by providing a flexible, structured approach to data management and device coordination. [4]

Bibliography

[1] Texas Instruments ™, *"Introduction to the Controller Area Network"*, 2002 – revised 2016.

[2] Robert Bosch GmbH., *"CAN specification"*, 1991

[3] www.picotech.com, *https://www.picotech.com/library/knowledge-bases/oscilloscopes/can-bus-serial-protocol-decoding*

[4] CSSElectronics, *https://www.youtube.com/@CSSElectronics-CAN-Logger-X000*