Department of Computer Science
Technical University of Cluj-Napoca

CAN Communication Simulation

Name: Zdrenghea Iulia
Group: 30433
Email: zdrengheaiulia01@gmail.com

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this project is to develop a simulation environment for CAN (Controller Area Network) communication. This simulator will be a valuable tool for testing and validating CAN-based systems and applications without the need for physical hardware.

1.2 Objectives

Simulate a virtual CAN network with multiple nodes to accurately represent real-world scenarios. Allow users to define and configure CAN messages. Display a real-time visualization of CAN bus traffic, including message IDs, data content, and error status.

1.3 Plan

1st Meeting: choose a project.

2nd Meeting: Introduction, Bibliographic Study, Bibliography. Research on CAN communication protocols and how they work.

3rd Meeting: Analysis and Design of the project. Start designing the software architecture and simulation logic.

4th Meeting: Implementation and Test scenarios. Write code for the project and implement it, also create a UI for better visualization of the simulation. Test edge case scenarios and solve errors.

5th Meeting: Conclusions. Think about future improvements.

6th Meeting: Write the documentation, make sure everything works.

7th Meeting: Present the project.

Chapter 2

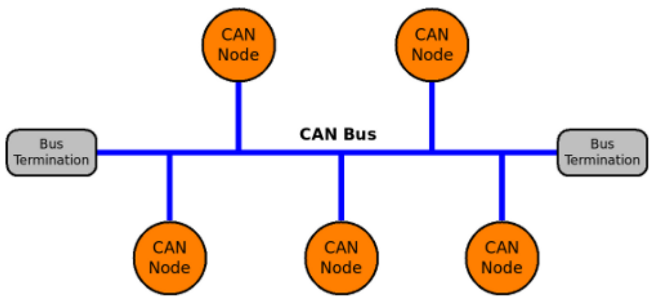
Bibliographic Study

A Controller Area Network communication is a serial bus protocol that is used to connect electronic control units and other devices in real-time applications. CAN buses and devices are common components in automotive and industrial systems.

In an automotive CAN bus system, ECUs can be the engine control unit, airbags, audio system. A modern car may have up to 70 ECUs and each of them may have information that needs to be shared with other parts of the network. The CAN bus system enables each ECU to communicate with all other ECUs – without complex dedicated wiring. Is usually implemented using a twisted pair cable. The two in the cable are designated CAN high and CAN low. Each device on the bus is connected to the CAN high and CAN low wires through a CAN transceiver. The CAN transceiver converts the digital signals from the device to different voltage signals that are transmitted on the bus.

2.1 How CAN communication look like?

We use a broadcast type of bus for this communication protocol, which means that in CAN communication, data is not sent point-to-point from node A to node B under the supervision of a central bus master, instead all devices can ”hear” the transmission. Consequently, there is no method to transmit data specifically to a node by its address. Therefore, all nodes will receive the traffic on the bus, this characteristic results in a communication network that links all the nodes connected to the bus.



2.2 Transceiver functional block

The CAN communication consists of two main components: a CAN controller and a CAN transceiver. The CAN controller handles the data link layer of CAN communication, whereas the CAN transceiver handles the physical layer.

To communicate with another node, differential signals are needed, which can be obtain by using a CAN transceiver. The CAN transceiver converts the single-ended signal to differential signals. The new differential signals created by the CAN transceiver are called CANH and CANL.

When transmitting a logical '1', the CAN transceiver sets the CANH signal to 2.5V, and the CANL signal is also set to 2.5V. This results in a 0V difference between the two, which is why the logical '1' is also known as the recessive state of the CAN bus.

Conversely, when transmitting a logical '0', CANH will be set to 3.5V, and CANL will be set to 1.5V, resulting in a 2V difference. This state of the CANH is referred to as the dominant state.

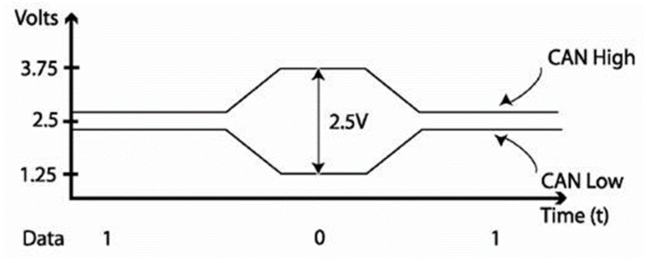


Figure 2.1: CAN transciever

2.3 CAN Fundamentals

There are four types of CAN frames: Data Frame, Remote Frame, Error Frame, Overload Frame. For this project we will use CAN 2.0B message with 11 bit (standard) identifier.

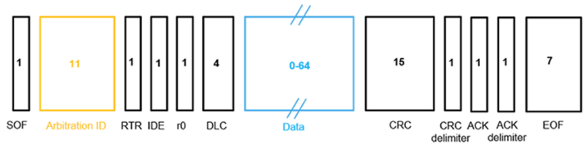


Figure 2.2: CAN Frame

- SOF Start of Frame – 1 bit - must be dominant
- Arbitrary ID – 11 bits - unique identifier indicates priority
- RTR Remote Transmission Request – 1 bit - dominant in data frames; recessive in remote frames
- IDE – 1 bit - must be dominant
- R0 – 1 bit - must be dominant
- DCL Data Length Code – 0 – 8 bytes - Contains the length of the data in bytes
- Data - Contains the actual data values; length is dictated by DLC field
- CRC Cyclic Redundancy Check – 15 bits - Error-detecting code that checks the data integrity

- CRC delimiter – 1 bit - Must be a recessive ‘1’.
- ACK Acknowledgement slot – 1 bit - transmitter sends recessive, receiver asserts dominant
- ACK delimiter bit – 1 bit - Must be a recessive ‘1’.
- EOF End of Frame – 7 bits - end of frame is indicated with a recessive ‘1’.

2.3.1 CAN Data Frame

The CAN data frame has seven parts: Start of Frame (SOF), Arbitration, Control, Data, CRC (a kind of double-check), Acknowledge (ACK), and End of Frame (EOF). In CAN, bits are either seen as ”more important” (0) or ”less important” (1). The SOF starts with an ”important” bit.

When many devices want to send messages on the network, they all get ready when they see the SOF and start sending at the same time. An arbitration plan helps decide which device gets to send its message first.

2.3.2 CAN Error Frame

If a node that’s sending or receiving data notices a mistake, it will stop right away and send out an error message. This message has a group of six important bits called an error flag, followed by eight less important bits called an error flag delimiter. Since this series of bits breaks the usual pattern, all the other nodes will also start sending error flags. If enough errors keep happening, a node might eventually shut itself down.

Nodes in a CAN network keep track of the number of errors they encounter. Depending on this count, a node can be in one of three modes: Error Active Mode (Error count between 0 and 128): The node is considered ”error active,” meaning it’s still fully functional, but at least one error has been detected, Error Passive Mode (Error count between 128 and 255): In this mode, the node is considered ”error passive.” It will transmit at a slower rate by sending 8 less important bits before sending important ones or waiting for the bus to be idle, Bus Off Mode (Error count less then 255): If the error count exceeds 255, the node will go into ”bus off” mode, effectively taking itself offline.

2.3.3 CAN Remote Frame

A node that requires data from another node on the network can request a transmission by sending a Remote Frame. A remote frame is the same as a data frame, without the data field (with the RTR bit recessive).

2.3.4 CAN Overload Frame

When a CAN node receives messages at a pace faster than it can handle, it will create an Overload Frame. This frame adds extra time between consecutive Data or Remote frames. Just like an Error Frame, the Overload Frame is made up of two parts: an overload flag with six important bits, and an overload delimiter with eight less important bits. Unlike error frames, the counters for tracking errors are not increased.

Chapter 3

Analysis

3.1 Introduction

A CAN Communication Simulator is a software application designed to mimic the functionalities and behaviors of a Controller Area Network bus. This technology is widely used across various industries. A simulator of this kind proves invaluable for tasks such as application development and testing, troubleshooting, as well as training CAN bus engineers.

3.2 Requirements

- Accuracy: The simulator must replicate the working of a genuine CAN bus. This means precise implementation of critical protocols such as arbitration and error detection.
- Efficiency: The simulator should be capable of emulating numerous CAN devices and messages with optimal performance.
- Usability: User-friendliness is crucial for the success.
- Configurability: The simulator should offer flexibility in configuring various parameters, including node properties, message content and network characteristics.
- Error Handling and Reporting: The simulator should be equipped to detect and appropriately respond to errors that may occur during simulation, providing informative feedback to users.
- Real-time Monitoring: Real-time visualization of CAN bus traffic can be a valuable feature for users to monitor the behavior of the simulated network.

3.3 Key Features

3.3.1 Virtual CAN Nodes

When defining this devices we have to decide on the characteristics of each virtual node such as: Node ID (assigning a unique ID to each node to differentiate them on the bus), Message Rate (determine how frequently each node will transmit messages), Message Content (specify the data payload of the messages each node will send). Next we have to create a data structure to represent a virtual CAN node. Also the user should be able to create and configure multiple CAN Nodes. Here we should also handle the message reception as well as Arbitration, for detecting which node gain control over the bus.

3.3.2 Message Generation and Transmission

First we have to define the structure of the CAN message for all 4 types of frames, then we can add predefined random messages or the user could define and configure the messages. Here we should also have a validation operation to ensure that the data given by the user is correct and error free. Implement a message queue for each virtual node to manage the messages waiting to be transmitted. When a message is ready for transmission, it should be placed in the queue. It is needed to implement an arbitration scheme.

3.3.3 Arbitration Handling

This part includes understanding how message IDs are used for arbitration and how the highest priority message gains bus access. Write code to handle the arbitration process. The logic should compare message IDs of nodes attempting to transmit and determine which message has the highest priority. Based on the arbitration result, establish the order in which messages will be transmitted. The message with the highest priority (lowest ID value) should gain access tot he bus first.

3.3.4 Error Simulation

Understand the CAN bus error types and identify specific points in the simulator where errors can be encountered. Write code to simulate the occurrence of errors at the designated injection points. Implement logic to detect errors within the simulation, checking for error flags in transmitted or received messages, as well as monitoring error counters. Determine how virtual nodes will reponde to detected errors.

3.3.5 Real-time Monitoring

Monitor message transmission, message reception, error detection, error handling, bus utilization. Identify specific parameters that should be tracked in real-time such as: message IDs, data content, error flags, error counters, bus load, and others. Visualize the data through a GUI and text-based console output.

3.4 High-Level Components

3.4.1 CAN Node Manager

The CAN Node Manager is a pivotal component, it's primary responsibility is to oversee the creation, configuration and management of virtual CAN nodes. Once created, the CAN Node Manager allows users to configure various attributes of each virtual CAN node. The Node Manager assigns a unique identifier (Node ID) to each virtual CAN node. The manager enables users to activate or deactivate individual virtual nodes. he CAN Node Manager provides functionality for removing virtual nodes when they are no longer needed. The CAN Node Manager interfaces with the CAN Bus Simulator component to integrate the created and configured virtual nodes into the overall simulation environment.

3.4.2 Message Generator

The Message Generator is a vital component, its primary responsibility is to produce CAN messages according to user-specified parameters. It allows users to specify various parameters that dictate the content, structure, and timing of CAN messages. The Message Generator allocates unique Message IDs to each generated message.

3.4.3 Arbitration Logic

The Arbitration Logic is like a traffic manager in a virtual CAN bus. Its job is to decide which message gets sent first. Each message has a special number called a Message ID, which helps the logic figure out which message is more important. If many messages want to go at once, the logic looks at their IDs and lets the one with the lowest ID go first. The rest of the messages wait their turn in line. The logic keeps checking to see if a new message should go ahead. This helps make sure that messages are sent in a fair and organized way.

3.5 Used Language

For this project I am going to use Java because it can run on any platform that has a Java Virtual Machine installed. I can catch errors early in the development process. It also has a built-in garbage collector, which helps manage the memory and prevent memory leaks. Java is designed around the principles of object-oriented programming, which promotes modularity, reusability, and maintainability of code. It can also suppoert multithreading and concurent programming, which is crucial for simulating real-time systems. A GUI is also provided by libraries like Swing.

Chapter 4

Design

4.1 Architecture

4.1.1 CAN Bus Model

At the center of everything is the CAN Bus Simulation Core. It's like the brain that manages all the actions of the simulated CAN bus, like sending messages, deciding which message goes first, and handling any errors. The Arbitration Protocol Module is an important part of this core. It's like the referee that makes sure messages compete fairly to get onto the bus, based on their special IDs. The Message Generation and Processing part creates the messages that will be sent on the simulated CAN bus. It works closely with the Arbitration Protocol Module to figure out which message should go first. The CAN Bus Model also works together with the CAN Node Manager to bring in virtual CAN nodes. This makes sure that messages from these virtual nodes follow the rules and get sent on the simulated bus.

4.1.2 CAN Device Model

The CAN Device Model is designed to imitate the behavior of a real-world CAN device. It has different components that work together seamlessly. At the center of it all is the Device Controller, which acts like the brain of the simulated device. It takes care of sending and receiving messages, handling special events, and managing error situations. The Message Sending Module is responsible for creating and sending CAN messages. It talks to the virtual CAN bus and decides which messages should go first based on their importance. On the other hand, the Message Receiving Module captures incoming CAN messages, processes them, and figures out what to do with them. Additionally, the Message Queue organizes messages, ensuring they are sent or received in the right order. All of these components collaborate to make the simulated CAN device behave just like a real one, effectively managing message flow and responding to events.

4.1.3 CAN Message Model

The CAN Message Model is like a blueprint that shows how different types of messages work on a CAN bus. There are four main types of messages: Standard Data Frames, Error Frames, Remote Frames and Overload Frames. Each type has its own special features. Every message has something called a ”Message ID” that makes it unique. This ID helps decide which message gets sent first on the bus. The model also pays attention to whether a message is a regular one, a special request, or an error. It works together with the CAN Bus Model to make sure messages act just like they would on a real CAN bus.

4.1.4 User Interface

The User Interface will consist of a Configuration Panel which allows users to set parameters and options for the simulation. Here the user can choose to configure a Device or a Message. After the configuration is done the user can see the incoming CAN messages in real-time, the bus status and how the messages flow through the Bus. Also the user will benefit of a user input validation mechanism to ensure that user inputs are within acceptable ranges and conform to the CAN communication protocol standards.

4.2 Data Structure

- Node Structure: This structure represents a virtual CAN node in the simulation. It may include fields like Node ID, status, message queue, error count, and other relevant attributes.

- Message Structure: This structure defines the characteristics of a CAN message. It includes fields like Message ID, data length, data content, frame type, and other relevant information.

- Message Queue Structure: This structure manages the queue of messages for a particular CAN node. It may include fields like front, rear, message array, and queue size.

- Timing and Synchronization Structure: This structure manages timing aspects of the simulation, including message transmission intervals, synchronization points, and response times.

- CAN Bus Structure: This structure represents the virtual CAN bus in the simulation. It may include fields like message array, bus status, and other relevant attributes.

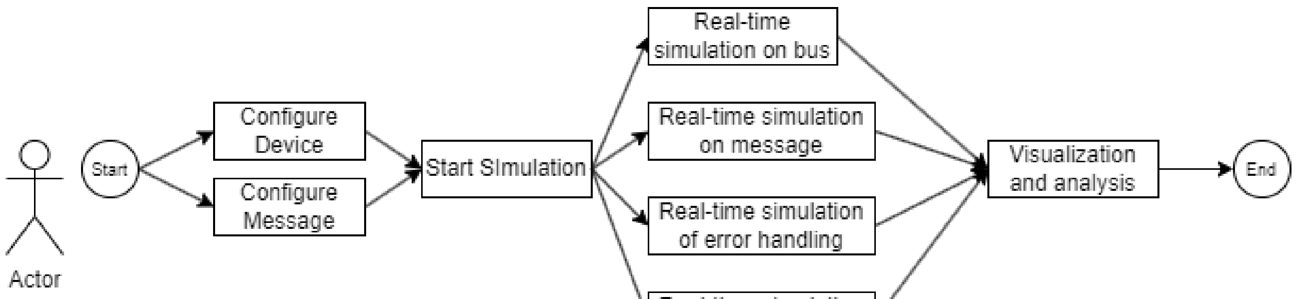

Actor
Start
Configure Device
Configure Message
Start SImulation
Real-time simulation on bus
Real-time simulation on message
Real-time simulation of error handling
Real-time simulation of arbitration handling
Visualization and analysis
End

Figure 4.1: Use-case diagram

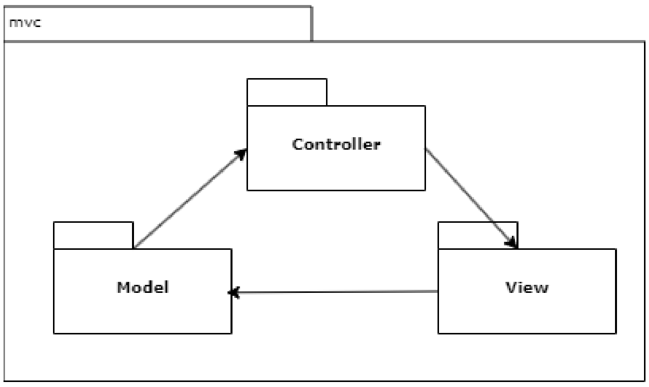


Figure 4.2: Package diagram

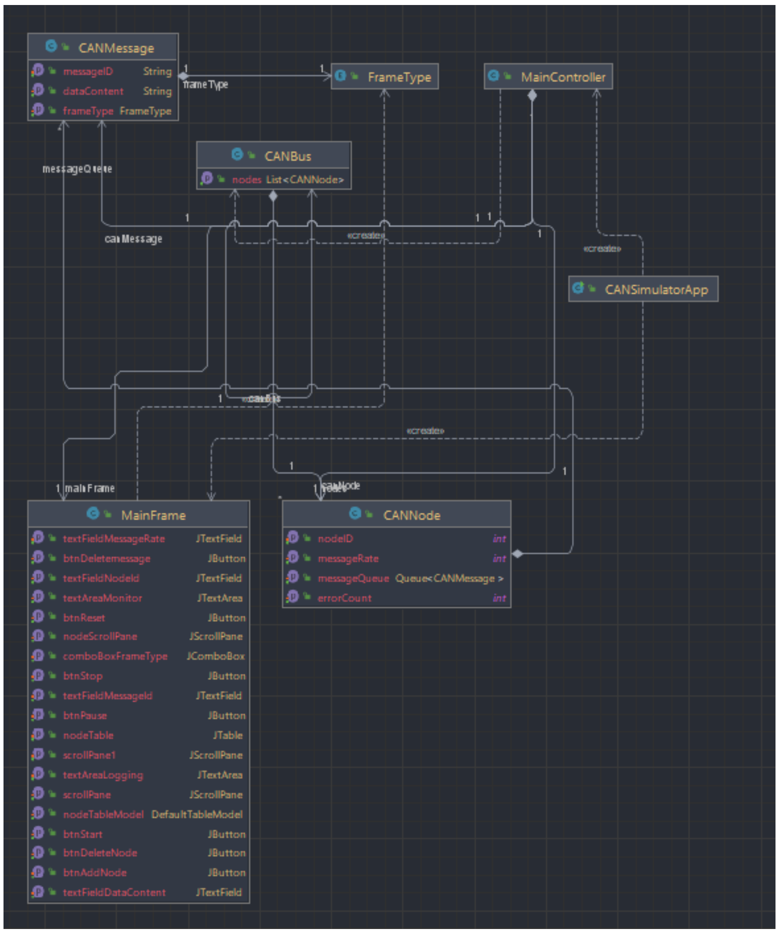


Figure 4.3: Enter Caption

Chapter 5

Implementation

When implementing this project, I first thought about how the GUI should look like, and how it will be effecting the user. So I came with a design that is simple to work with. An important thing about the GUI is that its just one part of a large application. The application also includes models and controller classes which handle the simulation logic and interaction with the GUI components.

5.1 GUI

This Java code defines the graphical user interface (GUI) for a CAN (Controller Area Network) communication simulator using the Swing framework. The GUI is designed to control and monitor the simulation of CAN bus communication, allowing users to add nodes, start, pause, stop, and reset the simulation.

The MainFrame class includes various Swing components such as buttons, labels, text fields, tables, and text areas, which collectively form the GUI. It also includes instances of the DefaultTableModel class to manage data in the tables.

Various methods provide functionality for appending text, clearing text areas, getting and setting component values, adding ActionListeners, and displaying error messages.

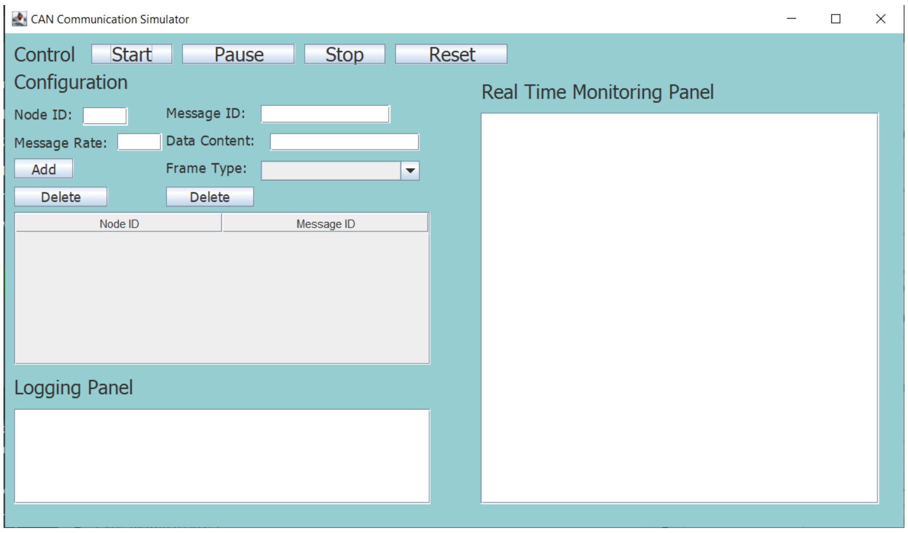
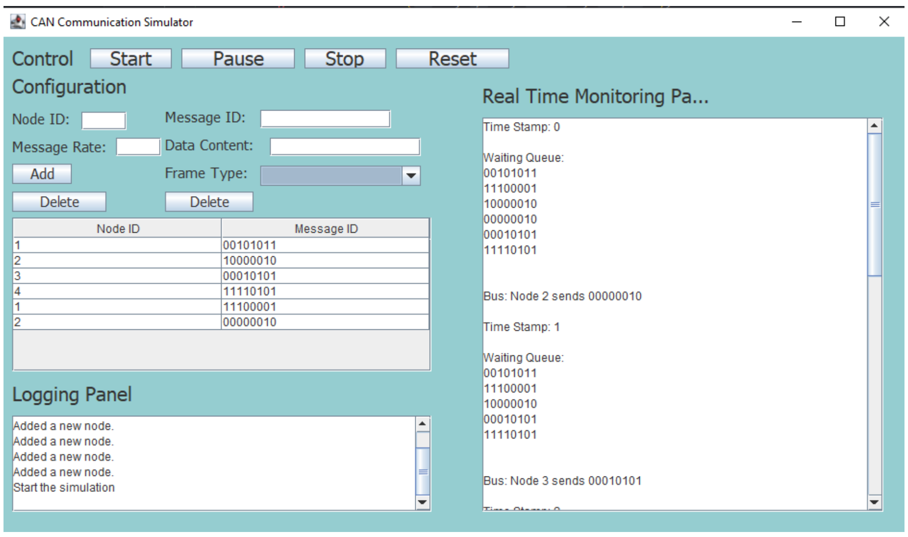


Figure 5.1: Enter Caption



Node ID	Message ID
1	00101011
2	10000010
3	00010101
4	11110101
1	11100001
2	00000010

```
Added a new node.
Added a new node.
Added a new node.
Added a new node.
Start the simulation
```

```
Time Stamp: 0

Waiting Queue:
00101011
11100001
10000010
00000010
00010101
11110101


Bus: Node 2 sends 00000010

Time Stamp: 1

Waiting Queue:
00101011
11100001
10000010
00010101
11110101


Bus: Node 3 sends 00010101
```

Figure 5.2: Enter Caption

5.2 Controller

For the controller the most important part is to simulate the real time monitoring of the CAN Messages on the Bus. It also manages the user interactions and a series of action listeners govern user-triggered events, ensuring a responsive and user-friendly experience.

The 'BtnStartListener' class triggers the initiation of the simulation. Also other listeners like AddNode class handles the addition of nodes to the simulation, incorporating input validation mechanisms. It ensures that Node ID and Message Rate are valid integers, while Message ID and Data Content adhere to the specified format of strings consisting solely of 0s and 1s. Error messages are prominently displayed on the logging panel in case of incorrect input, enhancing user feedback.

The simulateCANBusOperation method encapsulates the dynamic aspect of the simulation, iterating through source nodes, displaying pertinent information, and orchestrating the arbitration process. Real-time information, including waiting queues and arbitration outcomes, is presented on the monitoring panel, providing users with a comprehensive insight into the ongoing simulation.

Chapter 6

Testing and Validation

Input set

```
NodeID: 1
MessageRate: 2
MessageID: 00101011101
DataContent: 1101010110100011011110101011111001011100010101100111001101110110
NodeID: 2
MessageRate: 1
MessageID: 10000010101
DataContent: 1010101100111000100101011100101100111110100010101011100111100011
NodeID: 3
MessageRate: 2
MessageID: 00010101000
DataContent: 1110010011100101110100100101101001011100011110111100010111110100
NodeID: 4
MessageRate: 5
MessageID: 11110101010
DataContent: 1001110001010101010001111100110001001111010100100011101000101011
NodeID: 1
MessageRate: 2
MessageID: 11100001011
DataContent: 0101011011101101100010010000111010101101011101101011010011110000
NodeID: 2
MessageRate: 1
MessageID: 00000010111
DataContent: 0011101110110010111011101100010010100111101110101010111101101101
```

Output set

```
Time Stamp: 0
Waiting Queue:
00101011101
11100001011
10000010101
00000010111
00010101000
11110101010
Bus: Node 2 sends 00000010111
Time Stamp: 1
Waiting Queue:
00101011101
11100001011
10000010101
00010101000
11110101010
Bus: Node 3 sends 00010101000
Time Stamp: 2
Waiting Queue:
00101011101
11100001011
10000010101
11110101010
Bus: Node 1 sends 00101011101
Time Stamp: 3
Waiting Queue:
11100001011
10000010101
11110101010
Bus: Node 2 sends 10000010101
Time Stamp: 4
Waiting Queue:
11100001011
11110101010
Bus: Node 1 sends 11100001011
Time Stamp: 5
Waiting Queue:
11110101010
Bus: Node 4 sends 11110101010
```

Chapter 7

Conclusions

Working on this project was a bit challenging because it was the first time when I found out how a CAN bus works and how to make a simulation for it. Throughout the development, we encountered challenges related to the synchronization of the simulation and ensuring the accurate representation of the CAN bus dynamics. Debugging and refining the code were essential steps in achieving a functional and reliable simulation.

The project successfully demonstrates the fundamental principles of a CAN bus system, showcasing message transmission, arbitration, and node interactions. As with any simulation, there is room for further enhancements and optimizations, such as incorporating additional features, refining the user interface, and ensuring scalability for large networks.

In conclusion, the CAN Bus Simulator project provides a solid foundation for understanding and experimenting with the behavior of a CAN bus in a controlled environment. It serves as a valuable educational tool for individuals learning about embedded systems, communication protocols and real-time systems.

Chapter 8

Bibliography

- https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/controller-area-network–can–overview.html
- https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial
- https://community.nxp.com/t5/Blog/101-Controller-Area-Network-CAN-standard/ba-p/1217054
- https://www.kvaser.com/can-protocol-tutorial/
- https://copperhilltech.com/blog/controller-area-network-can-bus-tutorial-message-frame-format/
- https://www.analog.com/en/technical-articles/understanding-can-transceiver-how-validate-multinode-can-system-performance.html
- https://www.eecs.umich.edu/courses/eecs461/doc/CANnotes.pdf