Simulation of communication in industrial networks – CAN network

Student: Mădăras Paul

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

1. Introduction

1.1 Context

In modern industrial environments, reliable and efficient communication between various control systems, sensors, and actuators is crucial for optimal operation. One widely-used protocol to achieve this communication is the Controller Area Network (CAN), a robust and flexible standard developed for real-time distributed control applications. CAN networks were originally developed for automotive systems, but their application has grown to various other industrial sectors due to their fault tolerance, real-time communication, and support for distributed control systems. Since their creation, CAN networks have become a staple in the automotive, manufacturing, robotics, and automation industries.

The CAN protocol enables multiple devices (referred to as nodes) to communicate with each other over a single shared bus, allowing for effective data transmission without the need for a central controller. As industries continue to grow in complexity, simulating and understanding the behavior of CAN networks has become increasingly important. This simulation aids in studying data flow, network bottlenecks, error handling, and overall performance, which is critical for optimizing industrial networks and ensuring seamless operation.

1.2 Objectives

The primary objective of this project is to design and implement a simulation of communication within an industrial CAN network. The simulation will focus on modeling key aspects of CAN communication, such as message transmission, bus arbitration, error detection, and fault recovery. By simulating real-world scenarios within the network, this project aims to:

- Model CAN network communication: Simulate message exchange between multiple devices (nodes) in an industrial network using the CAN protocol.
- Implement bus arbitration: Simulate CAN's priority-based message transmission, where lower-priority messages may be delayed or dropped in favor of higher-priority messages.
- Handle error detection and recovery: Implement CAN's error-handling mechanisms such as bit stuffing, CRC checks, and error frames to reflect real-world network reliability.
- GUI for Visualization: Develop a user-friendly graphical user interface to visually represent the communication process within a CAN network. This will allow users to

see data flow, message transmission, priority arbitration, and error states in real time.
- Test Scenarios and Performance Metrics: Provide different testing scenarios that demonstrate various states of the network, including error conditions, network load testing, and transmission delays.

1.3 Project Proposal

This project proposes the development of a **CAN network communication simulation** with a focus on real-time message arbitration and error handling in industrial environments. The simulation will be written in C# and will incorporate a GUI to make the underlying processes more accessible and understandable to the user.

The system will include the following components:

- **CAN Network Model:** A virtual network representing multiple CAN nodes, represented by objects of a class, each capable of sending and receiving messages in accordance with the CAN protocol**.**
- **Communication Simulation:** The transmission and reception of CAN frames will be implemented, including bit arbitration, error detection, and message prioritization.
- **Message Arbitration and Error Handling:** The simulation will demonstrate how message priority is resolved based on the identifier and how the CAN network responds to errors, such as bit errors, CRC errors, and acknowledgment failures.
- **Graphical User Interface (GUI):** A fully integrated, interactive C#-based interface that will display message flows, node status, and network states in real time. Users will have the ability to monitor and interact with the network simulation by introducing specific scenarios, such as bus overload or message collision.
- **Testing and Performance Evaluation**: The simulation will support performance evaluation under different network conditions, allowing users to observe behavior under high network loads, error conditions, and delays.

2. Bibliographic Research

2.1 CAN History

Bosch originally developed CAN in 1985 for in-vehicle networks. In the past, automotive manufacturers connected electronic devices in vehicles using point-to-point wiring systems. Manufacturers began using more and more electronics in vehicles, which resulted in bulky wire harnesses that were heavy and expensive. They then replaced dedicated wiring with in-vehicle networks, which reduced wiring cost, complexity, and weight. CAN, a high-integrity serial bus system for networking intelligent devices, emerged as the standard in-vehicle network.

2.2 How does CAN communication work?

CAN is a peer-to-peer network. This means that there is no master that controls when individual nodes have access to read and write data on the CAN bus. When a CAN node is ready to transmit data, it checks to see if the bus is busy and then simply writes a CAN frame onto the network. The CAN frames that are transmitted do not contain addresses of either the transmitting node or any of the intended receiving node(s). Instead, an arbitration ID that is unique throughout the network labels the frame. All nodes on the CAN network receive the CAN frame, and, depending on the arbitration ID of that transmitted frame, each CAN node on the network decides whether to accept the frame.

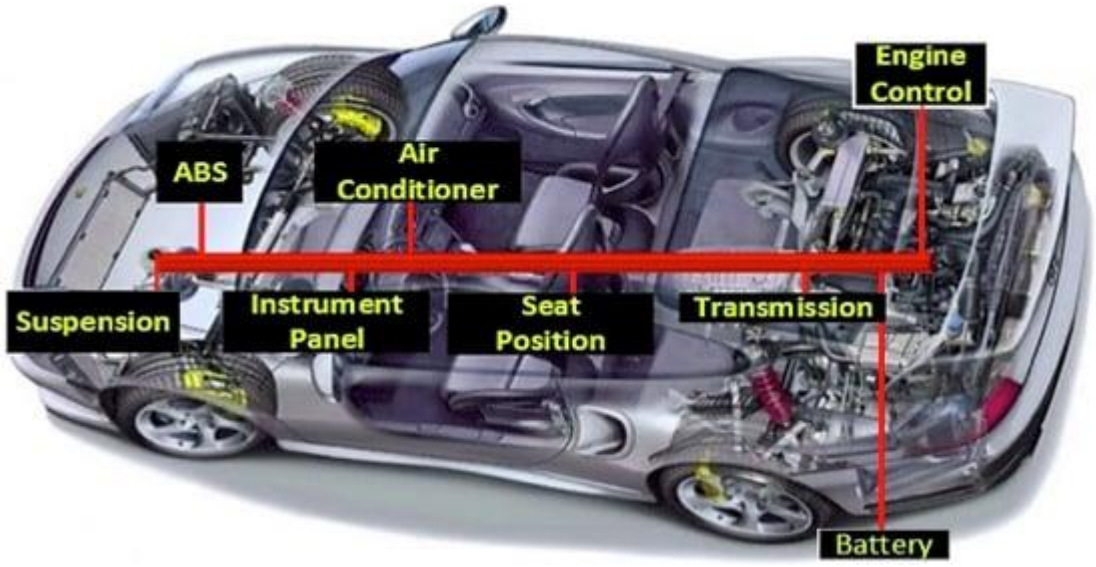


Figure 1 CAN Network representation in a car, source: *All About Circuits*

If multiple nodes try to transmit a message onto the CAN bus at the same time, the node

with the highest priority (lowest arbitration ID) automatically gets bus access. Lower-priority nodes must wait until the bus becomes available before trying to transmit again. In this way, you can implement CAN networks to ensure deterministic communication among CAN nodes.

2.3 CAN Benefits

- **Low-Cost, Lightweight Network:** CAN provides an inexpensive, durable network that helps multiple CAN devices communicate with one another. An advantage to this is that electronic control units (ECUs) can have a single CAN interface rather than analog and digital inputs to every device in the system. This decreases overall cost and weight in automobiles.

- **Broadcast Communication:** Each of the devices on the network has a CAN controller chip and is therefore intelligent. All devices on the network see all transmitted messages. Each device can decide if a message is relevant or if it should be filtered. This structure allows modifications to CAN networks with minimal impact. Additional non-transmitting nodes can be added without modification to the network.

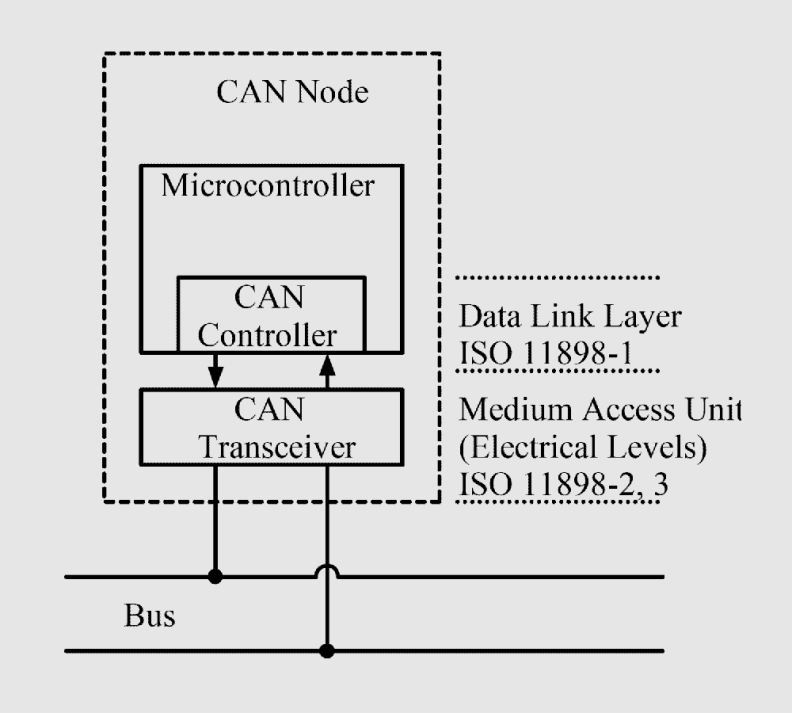


Figure 2 CAN bus network schematic, source: *DEWE Soft*

- **Priority:** Every message has a priority, so if two nodes try to send messages simultaneously, the one with the higher priority gets transmitted and the one with the lower priority gets postponed. This arbitration is non-destructive and results in non-

interrupted transmission of the highest priority message. This also allows networks to meet deterministic timing constraints.

- **Error Capabilities:** The CAN specification includes a Cyclic Redundancy Code (CRC) to perform error checking on each frame's contents. Frames with errors are disregarded by all nodes, and an error frame can be transmitted to signal the error to the network. Global and local errors are differentiated by the controller, and if too many errors are detected, individual nodes can stop transmitting errors or disconnect itself from the network completely.

2.4 CAN Applications & Examples

CAN Networks can be found in: railway applications, trains, aircraft systems, medical instruments, coffee machines, industrial automation systems, factory robotics and assembly lines, elevators and escalators, mining equipment, agricultural machinery, electric bikes and scooters, power generation systems, wind turbines, smart grid and energy management systems, building automation, home appliances, security and surveillance systems, medical devices, hospital operating equipment, prosthetics and rehabilitation devices, laboratory and diagnostic equipment, audio and broadcast systems, stage lighting and sound systems, printing and packaging machinery, vending machines, electric wheelchairs and mobility devices, CNC machines and industrial tools, military vehicles and equipment, submarines, space exploration rovers and satellites, oil and gas drilling equipment, autonomous drones, traffic and toll systems, smart parking systems, environmental monitoring systems, amusement park rides, robotic exoskeletons, fitness and rehabilitation equipment, firefighting equipment, to name a few.

2.5 CAN Protocol

CAN devices send data across the CAN network in packets called frames. A CAN frame consists of the following sections.

- **CAN Frame:** an entire CAN transmission: arbitration ID, data bytes, acknowledge bit, and so on. Frames also are referred to as messages.

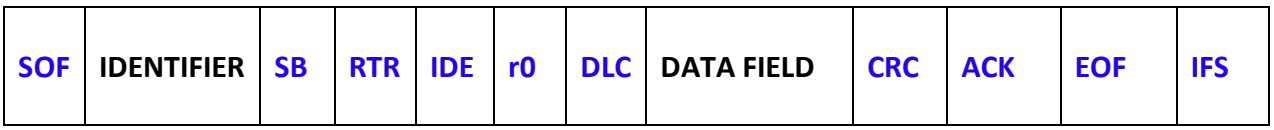


Figure 3 CAN Frame

- **SOF (start-of-frame) bit:** indicates the beginning of a message with a dominant (logic

0) bit.

- **11-bit IDENTIFIER:** identifies the message and indicates the message's priority. This represents the standard format.
- **SB bit:** stuff bit, a bit of the opposite polarity to maintain synchronization.
- **RTR (remote transmission request) bit:** serves to differentiate a remote frame from a data frame. A dominant (logic 0) RTR bit indicates a data frame. A recessive (logic 1) RTR bit indicates a remote frame.
- **IDE (identifier extension) bit:** allows differentiation between standard and extended frames. Must be dominant (logic 0) for base format with 11-bit identifiers
- **r0:** reserved bit, must be dominant (logic 0)
- **4-bit DLC (data length code):** indicates the number of bytes the data field contains.
- **Data Field:** contains 0 to 8 bytes of data.
- **16-bit CRC (cyclic redundancy check):** contains 15-bit cyclic redundancy check code and a recessive delimiter bit. The CRC field is used for error detection.
- **2-bit ACK (ACKnowledgement):** any CAN controller that correctly receives the message sends an ACK bit at the end of the message. The transmitting node checks for the presence of the ACK bit on the bus and reattempts transmission if no acknowledge is detected. Also has a recessive delimiter bit.
- **7-bit EOF:** End-of-frame, must be recessive (logic 1), this is used for error detection, if any of the bits are not logic 1, then the entire frame is compromised.
- **3-bit IFS:** Inter-frame spacing, must pe recessive (logic 1), this allows for processing time and keeps the bus idle.
- **CAN Signal:** an individual piece of data contained within the CAN frame data field. You also can refer to CAN signals as channels. Because the data field can contain up to 8 bytes of data, a single CAN frame can contain 0 to 64 individual signals (for 64 channels, they would all be binary).

3. Design I

3.1 Core Class Structure

- **ECU Class:**

    - **Purpose**: Represents individual ECUs (Electronic Control Units) that can send and receive messages on the CAN bus.
    - **Attributes**:
        - **ID** (int): Unique identifier for each ECU, used to identify messages on the CAN.
        - **MessageQueue** (Queue): Stores messages that the ECU intends to send.
        - Listeners (List<int>): A list of ECU IDs that this ECU can send messages to (e.g., temperature ECU may send alerts to the door ECU).
        - **Status** (enum or bool): Keeps track of whether the ECU is active or idle.
        - **Listeners** (List<int>): List of the listeners of an ECU
        - **NextMessage** (Message): Keeps track of the most important message at the ECU level.
    - **Methods**:
        - **SendMessage**(message): Adds a message to the **MessageQueue**.
        - **Transmit**(CAN Bus): Transmits a message on the bus
        - **ReceiveMessage**(string): Handles incoming messages in the CAN Protocol format.
        - **ProcessMessage**(): Processes and responds to received messages based on their type or importance.

- **CAN Bus Class:**

    - **Purpose**: Manages message arbitration, handling conflicts, and delivering messages to the correct ECUs based on their identifiers.
    - **Attributes**:
        - **ConnectedECUs** (List<ECU>): All ECUs connected to this CAN bus.
        - **Message** (List<Message>): Temporary holding for messages waiting for arbitration.
    - **Methods**:
        - **TransmitMessages**(): Checks **NextMessage** for the **ConnectedECUs** and determines which message has priority. Delivers the message to the appropriate recipients.
        - **AddECU** (ECU): Adds a ECU to **ConnectedECUs**.

- **Simulation Class:**

    - **Purpose**: Orchestrates the simulation by initializing the CAN bus and ECUs, simulating message passing, and controlling timing.
    - **Attributes**:
        - **CANBus** (CANBus): Instance of the CANBus class.

    - **TimeInterval** (int): Defines the clock cycle for each simulation step.
  - **Methods**:
    - **RunCycle**(): A method that simulates one "tick" of the CAN network, during which each ECU can attempt to send messages, and the CAN bus manages arbitration. Although the CAN Bus is asynchronous, for the simulation we need to know when we send a specific signal to see it's effect on the bus.
    - **SetupECUs**(): Initializes ECUs, establishes initial states, and configures message routing.

- **Message Class:**

  - **Purpose**: Encapsulates a message to be sent across the CAN bus.
  - **Attributes**:
    - **Time** (int): When the message is supposed to be sent.
    - **Data** (Dictionary or Byte Array): Contains the data payload.
    - **Priority** (int): Defines message priority for arbitration on the CAN bus.
  - **Methods**:
    - **toString**(): this method overridden will create the CAN Bus frame the ECUs will receive

3.2 Data Structures for Efficient Message Handling

- **Array/List of ECUs**: This structure can store all ECU instances for easy iteration. The simulation cycle will loop through this list to check which ECUs have messages to send.

- **Priority Queue for Message Buffer**: Since CAN uses priority arbitration, storing messages in a priority queue allows easy retrieval of the highest-priority message each cycle.

3.3 Handling CAN Bus Arbitration

- **Simulation Timing**: Since CAN is asynchronous, the simulation can handle messages as they arrive, but each simulation step (or clock cycle) will instead represent a period in which each ECU gets a chance to send a message, this will mean accessibility for the user and ensure readability. By managing timing intervals we can control when each ECU checks if it has data to send.

3.4 GUI Structure

The GUI will consist of an intuitive drag-and-drop interface that captures the dynamic interaction of ECUs over a CAN bus. The layout should be clean, with clear status indicators and visual cues for message flow, so users can easily follow message transmission and arbitration. Here are the main design elements and interactions to focus on:

ECU Representation

- Each ECU is displayed as a module with a user-defined ID label and connection indicators.
- **Status Indicators**: We use color-coded LEDs on each ECU to show real-time activities:
    - **Green LED** for sending.
    - **Blue LED** for receiving.
    - **Red LED** for arbitration.
- **Message Flow Visualization**: When an ECU sends a message, its LED briefly lights up green, and recipient ECUs' LED light up blue. Each active message state holds for a number of seconds, showing message transmission visually.

CAN Bus Display

- **Central Connection Line**: Represent the CAN bus as a horizontal line in the workspace where all ECUs connect automatically.
- **Dynamic Message Display**: Messages appear briefly as they travel on the CAN line from the sending ECU to the receiving ECUs, helping users understand transmission routes. For instance:
    - An animated pulse or line moves from the sender ECU to the CAN line, then branches to each receiving ECU.

Live Message Feed

- **Message Log Panel**: A side panel displays a real-time list of all messages sent on the CAN bus, showing important details such as:
    - **Sender ID, Receiver ID(s), Priority Level**, and **Data Payload**.
- **Description**: Each entry includes a description of the data (e.g., "Temperature Warning: 90°C"), giving context to the message.
- **Highlight Active Messages**: Each message stays highlighted in the feed for a few seconds during transmission, corresponding to its visual effect in the main workspace.

Signal Indicators

- **LED Status Indicators**: Color-coded LEDs indicate each ECU's current status:
    - **Idle** (off), **Sending** (green), **Receiving** (blue), in **Arbitration** (red).

Arbitration Visualization

- **Arbitration Mode**: When multiple ECUs attempt to send simultaneously, the system enters an “arbitration mode”:
    - An indicator (e.g., "Arbitration in Progress") appears above the CAN bus line.
    - All competing ECUs briefly light up their LEDs **Red**, allowing users to see which ECUs are vying for bus access.
- **Winning ECU Highlight**: After a few seconds, the ECU that wins arbitration turns its LED green, while the others return to idle, clearly showing which ECU won the transmission opportunity.

Additional Features

- **Drag-and-Drop Interface**:
    - Users can add ECUs by dragging elements from a toolbar onto the main workspace.
    - The ECUs will automatically connect to the CAN Bus, represented as a horizontal line in the simulation window.
    - Users will have the option of choosing from predefined ECUs to connect, or one of their own creation, where in the side panel they can add the information of the ECU and a timetable of the messages that it will send on the bus and their data.
- **Tooltips for ECU Details**:
    - **Hovering** over an ECU reveals detailed information, including ID, status, and the last message sent or received.
    - **Clicking** an ECU will let the user choose what other ECUs it transmits messages to.
    -
- **Predefined simulations**:
    - The user will have the choice of creating his own simulation, as well as selecting an already written simulation. The user will also have the choice of choosing predefined ECUs with predefined messages to add, for which they only need to choose who they transmit to (e.g. an ECU that deals with engine temperatures).
- **Pause/resume and start/stop button**:
    - After creating his own simulation or choosing a predefined one, the user will press start
    - The user has the option to stop the simulation when he pleases, this will allow them to read the information on the ECUs or understand the messages sent by the ECUs. When done, they can resume.
    - If the user wants to exit the simulation early, he can do so by pressing the stop button, which will return him to the drag and drop planification part of the interface.

3.5 GUI Design

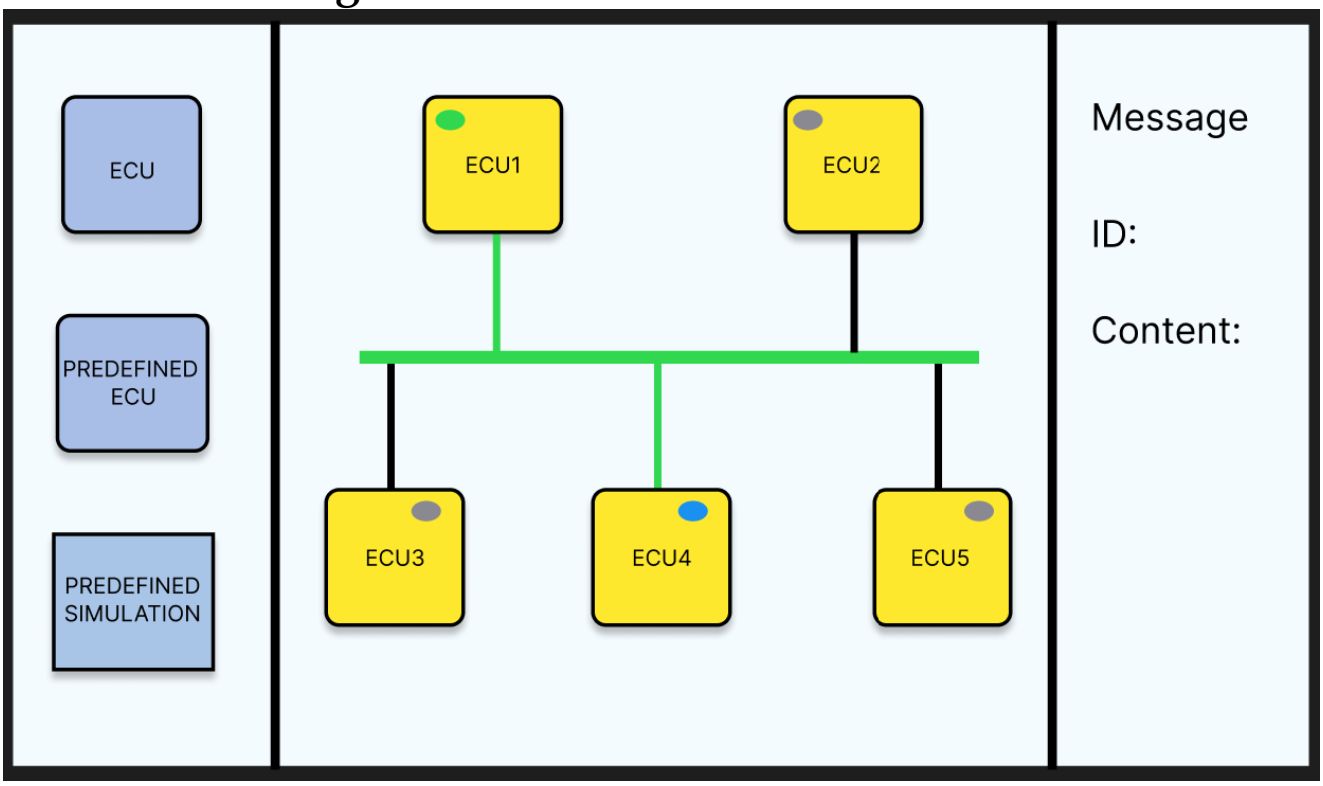


Figure 4 GUI Main Design

3.6 Block Diagram

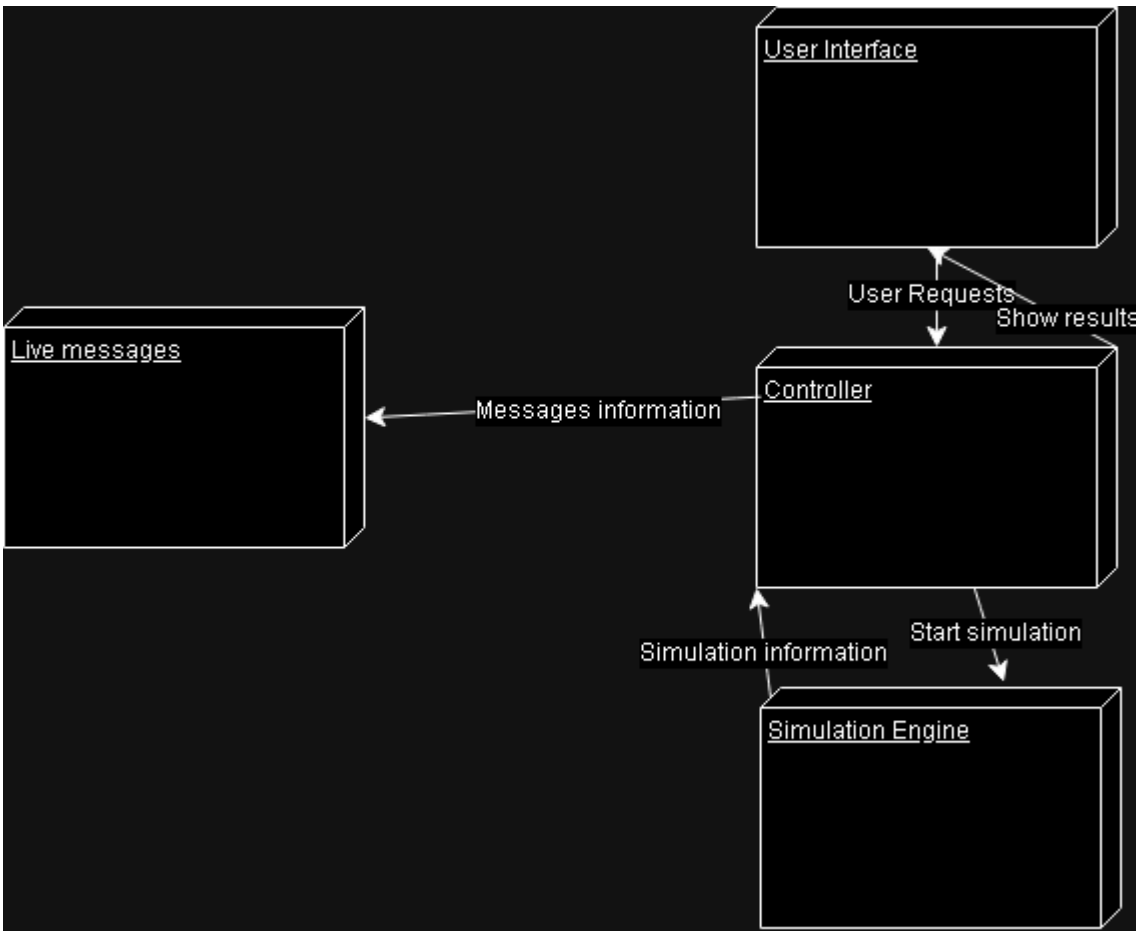
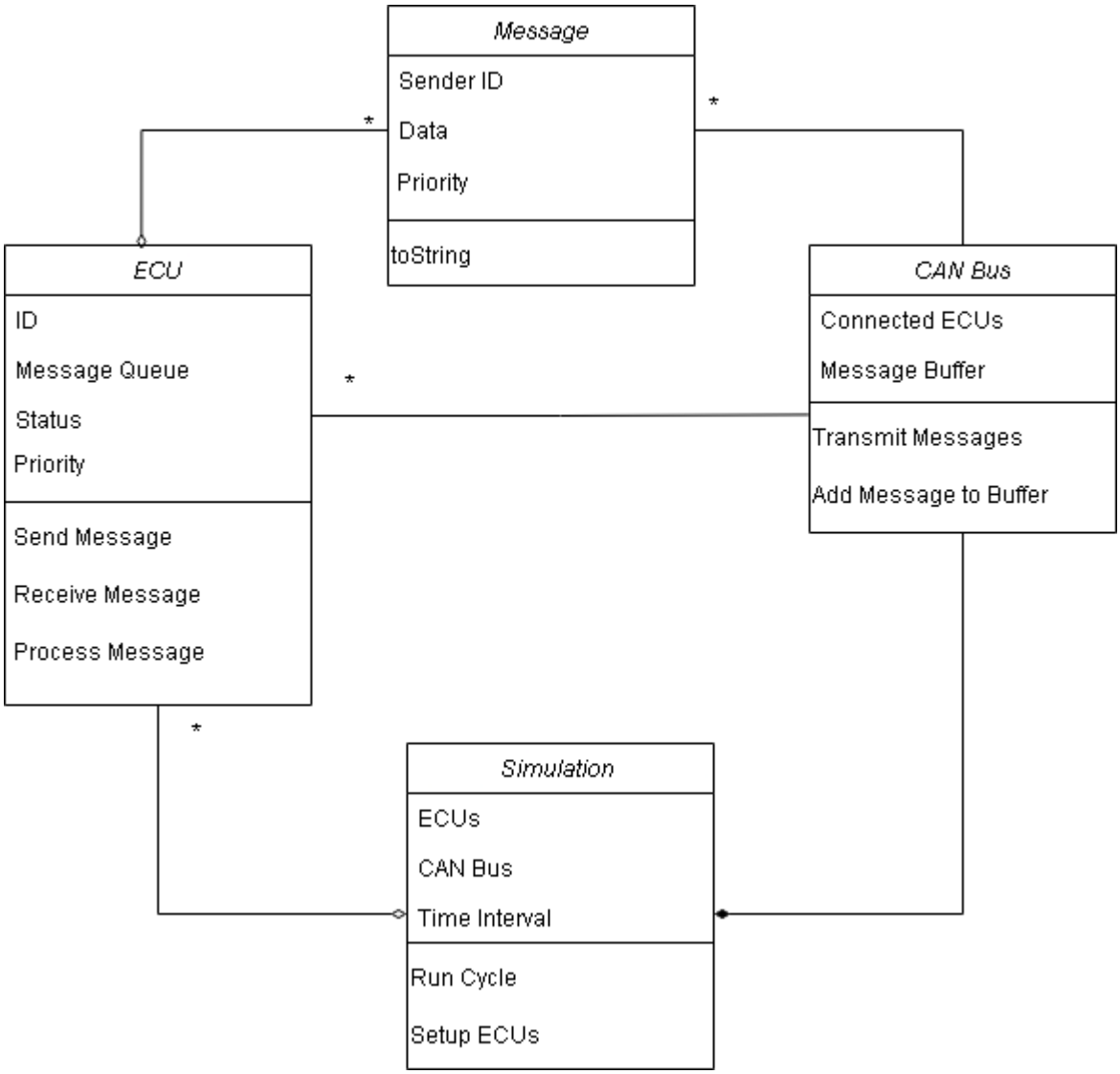


Figure 5 Block Diagram

3.7 Class diagram



4. Analysis

4.1 Use Cases

Use Case 1: Adding an ECU to the Workspace

Primary Actor: User
Goal: To add an ECU to the workspace by dragging it from the toolbar.

Preconditions:

- The simulation software is open, and the user is in the planning interface with the toolbar accessible.

Main Success Scenario:

1. The user selects an ECU icon from the toolbar.
2. The user drags the ECU icon onto the workspace and drops it in the desired location.
3. The system displays a configuration window for the ECU, prompting the user to:
    - Set a unique **ID** for the ECU.
    - Define **Listeners** (IDs of other ECUs this ECU can communicate with).
    - Set additional attributes, like **Priority** for message arbitration.
    - Select messages to be sent and when.
4. The user configures the ECU and confirms the settings.
5. The ECU appears on the workspace, with its ID label and configured settings.

Postconditions:

- The ECU is now ready to interact with other ECUs on the CAN bus when the simulation runs.

Use Case 2: Selecting a Predefined Simulation

Primary Actor: User
Goal: To quickly load and run a predefined simulation without needing to configure individual settings.

Preconditions:

- The application is open, and the user is on the planning board or main menu screen.

Main Success Scenario:

1. The user navigates to the **Predefined Simulations** section.
2. The user browses the list of available predefined simulation scenarios (e.g., "Basic CAN Communication," "Arbitration Conflict Test," "Error Handling Demo").
3. The user selects a predefined simulation suited to their needs.
4. The system loads the selected simulation, automatically configuring all parameters (e.g., ECU setup, CAN Bus properties, message flows).
5. The user clicks the **Start** button, and the simulation begins, running according to the predefined settings.

Postconditions:

- The predefined simulation runs as configured, allowing the user to observe or interact with it without manual setup.
- The user can view, pause, or stop the simulation as needed, just as with a custom simulation.

Use Case 3: Starting the Simulation

Primary Actor: User
Goal: To initiate the simulation and observe CAN bus interactions between ECUs.

Preconditions:

- The user has configured at least one ECU on the workspace.

Main Success Scenario:

1. The user clicks the "Start" button to begin the simulation.
2. The system initiates ECU actions based on their schedules and configurations, with messages sent and received as per CAN protocol.
3. The user observes:
    - **LED indicators** on each ECU (e.g., green for sending, blue for receiving, red for arbitration).
    - The **Live Message Feed** panel, showing real-time message details like sender, receiver, and message content.
4. The user can follow message flows visually as they move from sender ECUs to receiver ECUs via the CAN bus.

Postconditions:

- The simulation runs, giving the user insight into ECU interactions, message arbitration, and CAN bus operations.

Use Case 4: Pausing and Resuming the Simulation

Primary Actor: User
Goal: To temporarily pause the simulation to inspect ECU states and messages, then resume it.

Preconditions:

- The simulation is actively running.

Main Success Scenario:

1. The user clicks the "Pause" button, temporarily halting all message activity.
2. The user inspects the simulation state:
    - **LED indicators** on ECUs show their last activity (e.g., sending, receiving).
    - The **Live Message Feed** shows the recent messages and their details.
3. When ready, the user clicks the "Resume" button, and the simulation continues from the paused state.

Postconditions:

- The user can pause and resume the simulation as needed for closer observation and analysis.

Use Case 5: Stopping the Simulation

Primary Actor: User
Goal: To stop the simulation entirely, reset all activity, and return to the simulation planning board for adjustments or new configurations.

Preconditions:

- The simulation is actively running or paused.

Main Success Scenario:

1. The user clicks the "Stop" button to terminate the simulation.
2. The simulation halts completely:
    - All message activities between ECUs are stopped.
    - LED indicators on ECUs are reset to their initial states (e.g., inactive or off).
    - The Live Message Feed is cleared of any recent or past messages.
3. The system navigates the user back to the planning board screen, where:
    - The user can reconfigure simulation parameters (e.g., ECU roles, CAN Bus settings).
    - The user can review previous simulation setups or start a new simulation from scratch.

Postconditions:

- The simulation state is fully reset.

- The user is presented with the planning board, ready to set up a new simulation or modify existing parameters.

Use Case 6: Viewing Message Details in the Live Message Feed

Primary Actor: User
Goal: To monitor the content and flow of messages exchanged on the CAN bus in real-time.

Preconditions:

- The simulation is running, with messages being sent and received between ECUs.

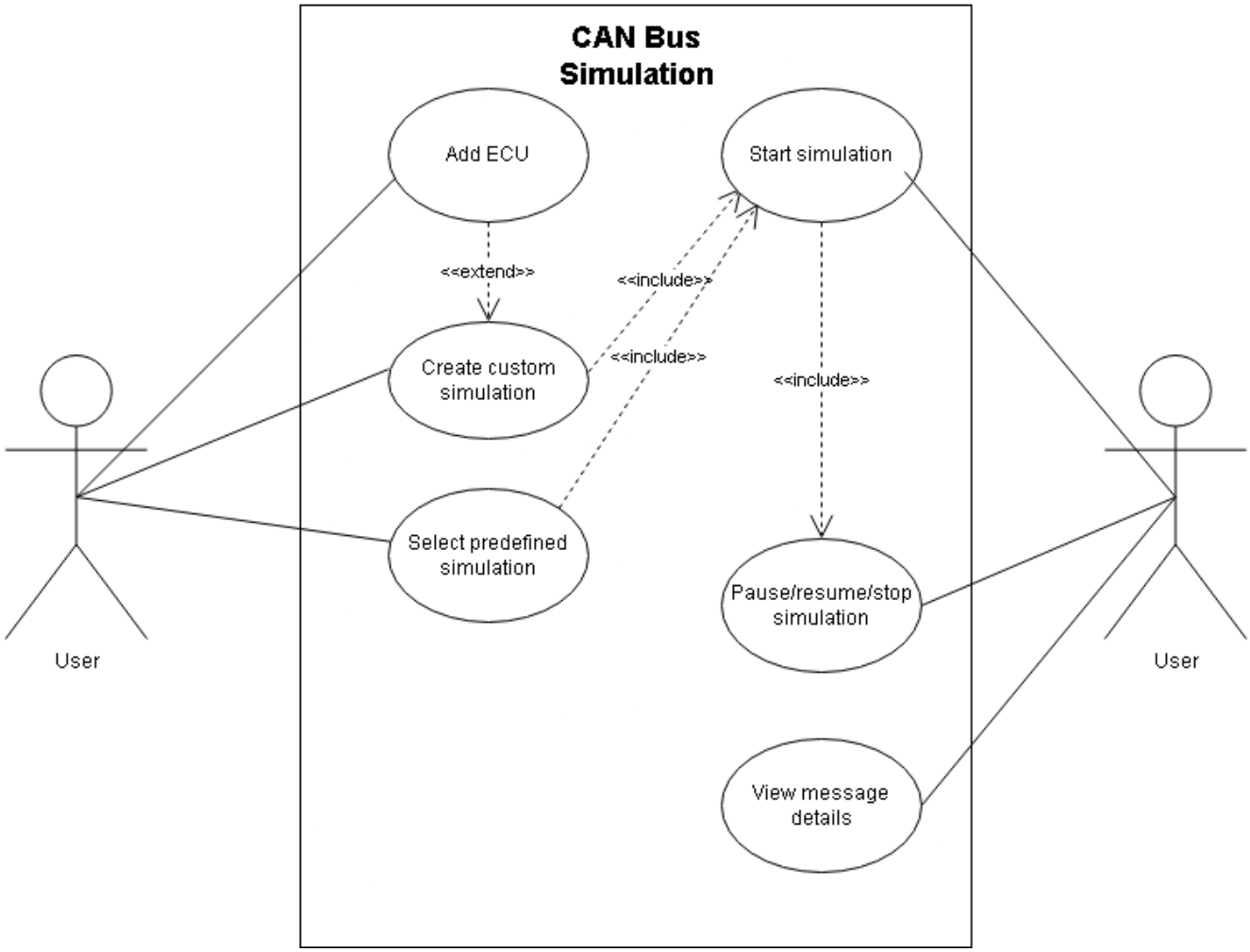
Main Success Scenario:

1. The user views the **Live Message Feed** panel, which updates with each message sent across the CAN bus.
2. For each entry, the feed displays:
    - **Sender ID** and **Receiver ID(s)** of the message.
    - **Priority Level** and **Data Payload** for message context.
    - **Description** of the message data (e.g., "Temperature Warning: 90°C").
3. Each message stays highlighted for a few seconds, giving the user time to read it before it disappears or scrolls.

Postconditions:

- The user gains insight into real-time message exchanges, their data, and priorities on the CAN network.

4.2 Use Case Diagram



5. Implementation I

Introduction

This chapter provides an in-depth discussion on the implementation of the designed system. It translates the analyzed requirements and design specifications into functional components, ensuring that the application adheres to the objectives. The system simulates a Controller Area Network (CAN) with multiple Electronic Control Units (ECUs) and supports message arbitration, faulty ECUs, and a realistic simulation environment.

5.1 Implementation Details

1. Core Components

The system is divided into the following key components:

1. **ECU Class**
   Represents an Electronic Control Unit (ECU) with functionalities to send, receive, and manage messages. ECUs also handle message arbitration through the CAN bus.
2. **FaultyECU Class**
   Extends the ECU class to simulate faulty behavior by overriding the ReceiveMessage method.
3. **Message Class**
   Encapsulates message data, including priority, payload, and cyclic redundancy check (CRC). The Message class also converts data into a binary format for transmission.
4. **CANBus Class**
   Simulates a CAN bus, mediating communication between connected ECUs and enforcing priority-based message transmission.
5. **Simulation Class**
   Orchestrates the simulation by managing ECUs, running transmission cycles, and controlling the simulation flow based on predefined conditions.

2. Key Classes and Their Implementation

1. ECU Implementation

The ECU class serves as the foundation of the system, representing a single Electronic Control Unit. Each ECU is responsible for managing its own message queue, transmitting messages to the CAN bus, and receiving messages from it.

Key functionalities include:

- **Message Queue Management**: ECUs maintain a queue to prioritize messages based on their scheduling times and predefined priorities. Messages are dynamically ordered to ensure correct arbitration.
- **Transmission**: ECUs convert outgoing messages into binary strings, simulating real-world CAN data frames. The transmission is synchronized with the CAN bus to ensure correct timing.

- **Reception**: Each ECU listens for messages on the CAN bus, validates incoming data, and sends acknowledgment signals based on its behavior.

The ECU also manages its state transitions, such as moving between idle, transmitting, or receiving states.

2. FaultyECU Class

The FaultyECU class extends the functionality of the standard ECU class to simulate erroneous behavior. By overriding the default message acknowledgment process, this class introduces faults into the system. For example:

- Faulty ECUs send incorrect acknowledgments, disrupting message validation.
- They simulate real-world scenarios where certain devices may malfunction, allowing the system to assess its resilience against such conditions.

3. Message Class

The Message class encapsulates all the critical details of a CAN message, such as:

- **Priority**: Defines the arbitration level for message transmission, ensuring higher-priority messages are transmitted first.
- **Data Length Code (DLC)**: Indicates the size of the payload, aligning with CAN specifications.
- **Payload**: Represents the actual data being transmitted, which is converted to binary format for transmission.
- **CRC (Cyclic Redundancy Check)**: Ensures the integrity of transmitted data by attaching a checksum, which is verified upon reception.

The implementation also includes bit-stuffing mechanisms, adding robustness to the simulation by preventing consecutive identical bits that could cause synchronization issues.

4. CAN Bus Functionality

The CAN bus is a central component that connects all ECUs, facilitating communication and enforcing message arbitration. Its implementation focuses on:

- **Message Arbitration**: Resolves conflicts when multiple ECUs attempt to transmit simultaneously. The bus prioritizes messages with the lowest priority value, ensuring a deterministic communication pattern.
- **Transmission Management**: The CAN bus ensures that only one ECU transmits at a time. It also propagates messages to all connected ECUs for acknowledgment.
- **Error Handling**: The system monitors the behavior of ECUs during transmission and reception, identifying faulty acknowledgments and simulating corrective actions when necessary.

5. Simulation Orchestration

The simulation component integrates all the system elements, managing the lifecycle of the CAN bus and connected ECUs. Its responsibilities include:

- **Initialization**: Configures ECUs with unique identifiers, listeners, and message queues. Each ECU is preloaded with messages to simulate real-time operations.
- **Cycle Management**: Runs the simulation in discrete cycles, evaluating message readiness based on timestamps and controlling the transmission order.
- **Pause and Stop Controls**: Allows dynamic interruption of the simulation, enabling testing under paused conditions or early termination when all messages are processed.

5.2 Challenges and Solutions

The implementation involved addressing several challenges:

- **Message Arbitration**: Ensuring correct arbitration required careful management of message priorities and synchronized transmission across ECUs.
- **Fault Simulation**: Adding faulty ECU behavior without disrupting the core simulation necessitated the use of inheritance and method overrides.
- **Scalability**: The modular structure of the system ensures that additional ECUs or extended CAN bus functionalities can be integrated seamlessly.

5.3 Implementation Summary

The implementation effectively translates the design into a functional simulation of a CAN bus system. Each component works cohesively to emulate realistic communication scenarios, allowing for robust testing and analysis of CAN protocols. The modular and extensible

design ensures the system can be adapted for future enhancements or additional functionalities.

6. Design II

6.1 New Classes

- **ECU Configuration Class**

    - **Purpose:** Represents the configuration of an Electronic Control Unit (ECU) for a CAN Bus simulation. It defines the parameters for message generation, error rates, priorities, and listeners.

    - **Attributes**
        - **ID (int):** Unique identifier for the ECU.
        - **Frequency (string):** Specifies the message generation rate in messages per minute (e.g., "60 messages/min").
        - **ErrorRate (string):** Represents the error rate as a fraction (e.g., "1/25"), determining the likelihood of generating an invalid message.
        - **MinPriority (int):** Defines the minimum priority value for messages generated by the ECU.
        - **MaxPriority (int):** Defines the maximum priority value for messages generated by the ECU.
        - **Listeners (List<int>):** A list of ECU IDs that this ECU can send messages to.
        - **ECU (ECU):** A reference to the specific ECU object associated with this configuration.

        - **Methods**
            - **GenerateRandomMessages():** Generates a list of random messages based on the configuration's frequency, error rate, and priority range.
            - **Returns:** A List<Message> containing generated messages, potentially with errors as per the defined error rate.
            - **ShuffleMessages(List<Message> messages):** Randomly shuffles a list of messages to introduce variability in transmission order.
            - **Parameters:** messages - The list of Message objects to shuffle.
            - **ParseFraction(string fraction):** Parses a fraction string (e.g., "1/25") and extracts the denominator as an integer to represent the error rate.

- **Parameters:** fraction - A fraction in the form "numerator/denominator".
- **Returns:** The denominator as an integer (default is 0 if parsing fails).

6.2 Changes To Classes

- **ECU Class**

  - **Purpose:** Represents individual ECUs (Electronic Control Units) that can send and receive messages on the CAN bus, while managing message priority, transmission, and reception states.

  - **Attributes**
    - **ID (int):** Unique identifier for each ECU, used to identify messages on the CAN bus.
    - **Name (string):** A descriptive name for the ECU.
    - **MessageQueue (Queue<Message>):** A queue storing messages the ECU intends to send.
    - **NextMessage (Message):** Represents the highest-priority message currently at the ECU level, ready for transmission.
    - **Listeners (List<int>):** A list of ECU IDs that this ECU is configured to send messages to.
    - **Status (ECUStatus):** Enum representing the current state of the ECU, such as Sending, Receiving, Arbitration, or Idle.
    - **TE (int):** Transmission error counter.
    - **RE (int):** Reception error counter.

  - **Methods**
    - **ECU(int id, string name, List<int> listeners, Queue<Message> messageQueue):**
      Constructor to initialize an ECU with its ID, name, listeners, and message queue.
      - If the queue contains messages, it sets the first message as NextMessage.
      - Default state is set to Idle.

    - **Message Handling**
      - **SendMessage(CANBus canBus):**
        Transmits the current NextMessage onto the CAN bus.
        - **Parameters:** canBus - The CAN bus instance.
      - **RemoveMessage():**
        Removes the current NextMessage after it has been transmitted and loads the next message from the MessageQueue.

- **ReceiveMessage(CANBus canBus):**
  Handles the reception of a message from the CAN bus. Verifies the message validity using the Message.IsValidMessage() method and sends an acknowledgment (ACK).
  - **Parameters:** canBus - The CAN bus instance.
  - **Returns:** A string representing the acknowledgment (ACK).
- **SendACK(bool isValid):**
  Sends an acknowledgment (ACK) for a received message based on its validity.
  - **Parameters:** isValid - Boolean indicating if the received message is valid.
  - **Returns:** A string representing the acknowledgment.

- **State Management**
  - **EnterArbitrationMode():**
    Switches the ECU to Arbitration mode and updates the simulation's visual representation.
    - **Returns:** A string message indicating the mode change.
  - **EnterIdleMode():**
    Switches the ECU to Idle mode and updates the simulation's visual representation.
    - **Returns:** A string message indicating the mode change.
  - **EnterSendingMode():**
    Switches the ECU to Sending mode, updating the visual simulation to indicate binary transmission (e.g., sending 0 or 1).
    - **Returns:** A string message indicating the mode change.
  - **Sending1():**
    Updates the simulation to show the ECU transmitting a binary 1.
  - **Sending0():**
    Updates the simulation to show the ECU transmitting a binary 0.
  - **EnterReceivingMode():**
    Switches the ECU to Receiving mode and updates the simulation's visual representation.
    - **Returns:** A string message indicating the mode change.
  - **EnterPassiveMode():**
    Activates the Passive mode when the transmission error count reaches the threshold, updating the simulation's visual representation.
    - **Returns:** A string message indicating entry into passive mode.

7. Implementation II

7.1 Changes

ECU Class

Overview of Changes

- Enhanced functionality and additional features to improve the management of message transmission and reception.
- Improved error handling mechanisms and state management.
- Better integration with the CAN bus simulation for visual and functional updates.

Key Improvements in Attributes

1. **Name Attribute:**
    - Added a Name attribute for better identification and debugging in simulation environments.
2. **Enhanced MessageQueue:**
    - Refined message queue functionality to dynamically manage pending messages and set NextMessage.
3. **Error Counters:**
    - Added TE (Transmission Errors) and RE (Reception Errors) counters to monitor the ECU's reliability.
4. **Listeners Attribute:**
    - Retained but expanded usage to clearly define inter-ECU communication.

Key Improvements in Methods

1. **State Transition Methods:**
    - **New State Modes:**
        - EnterArbitrationMode()
        - EnterIdleMode()
        - EnterSendingMode() (with visual differentiation for binary 1 and 0)
        - EnterReceivingMode()
        - EnterPassiveMode()
    - Enhanced visual simulation support by integrating mode changes with CanBusSimulator.
2. **Error Handling:**
    - **ReceiveMessage():**
        - Validates incoming messages using Message.IsValidMessage().
        - Acknowledges messages with an ACK string.
        - Tracks errors with the RE counter.
    - **Passive Mode Activation:**
        - Automatically enters Passive Mode when the transmission error count (TE) crosses a threshold, ensuring safer ECU operation under errors.
3. **Dynamic Visual Feedback:**
    - Methods like Sending1() and Sending0() provide real-time updates to the

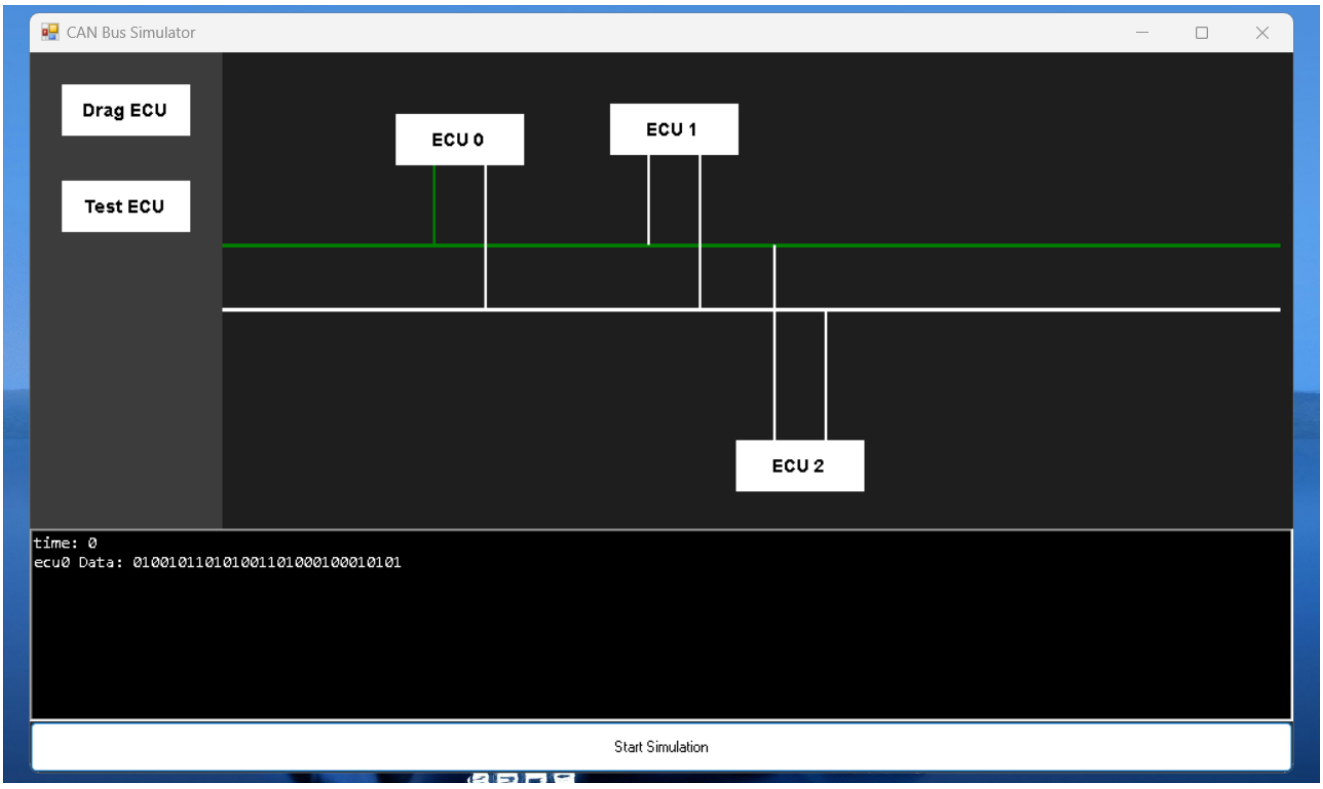
visual representation of the CAN bus and ECU connection states.

4. **Message Handling Workflow:**
    - Added robustness to the RemoveMessage() and SendMessage() methods to ensure smooth operation with the MessageQueue.

Purpose of Changes in Implementation 2

1. **Improved Fault Tolerance:**
    - Introduction of error counters (TE and RE) and passive mode ensures safer and more reliable operation in case of faults.
2. **Enhanced Debugging and Monitoring:**
    - Clearer state transitions and Name attributes provide better insights during simulation and testing.
3. **Improved Simulation Experience:**
    - Integration with CanBusSimulator enables real-time visual updates, making the system more user-friendly for education and debugging.
4. **Scalability and Maintainability:**
    - Well-defined state management and modular methods make the ECU class easier to extend and maintain.

7.2 GUI



8. Testing

For testing, a predefined simulation with errors is used. The test 2 tests.

8.1 Test 1

The setup consists of three ECUs:

1. **ECU0** (highest priority) has an erroRate of 1/1 and sends 60 messages per minute. This means it will be excluded from writing to the CAN bus after its first 2*error Threshold messages. To avoid repeating erroneous messages, each message will be transmitted twice—once with the error and once corrected (since its error rate is 1/1, each message will be sent twice: first incorrect, then correct).
2. **ECU1** (medium priority) has an error rate of 1/10 and transmits 25 messages per minute. This results in three erroneous messages every two minutes for a threshold of 5, meaning ECU1 will stop after 4 minutes.
3. **ECU2** (lowest priority) has an error rate of 1/25 and sends 60 messages per minute. After ECU0's messages are excluded, ECU2 will transmit 35 messages per minute. One of these will be erroneous every minute, causing ECU2 to remain the last to leave when no other ECUs are left.

Expected Output:

- ECU0 (highest priority): Sends 10 messages, but because of its high error rate (1/1), it is removed from the bus quickly.
- ECU1 (medium priority): Transmits messages for about 4 minutes, with occasional errors (3 errors every 2 minutes).
- ECU2 (lowest priority): Continues transmitting until all other ECUs are removed. It occasionally has errors (1 per minute).
- The system demonstrates proper handling of errors, priority enforcement, and gradual exclusion of faulty ECUs.

8.2 Test 2

The second test introduces more dynamic and varied ECU configurations compared to the first test. It primarily tests the following aspects of the CAN bus system:

1. **Handling of Higher Message Frequencies:**
    - Ensures the system can prioritize ECUs with a higher message frequency, as they transmit more often and require more arbitration.
2. **Dynamic Listener Relationships:**
    - Tests whether the acknowledgment and reception processes work seamlessly when listeners are assigned dynamically in a circular pattern (e.g., ECU0 → ECU1 → ECU2 → ECU3 → ECU0).
3. **Arbitration with Mixed Error Rates:**
    - Validates that ECUs with varying error rates (e.g., 1/5, 1/15, 1/20) can still participate in arbitration effectively until they exceed their error limits.
4. **Simulation of Moderate Load:**
    - Verifies the system's robustness under a moderate load of ECUs (4 in this case) with diverse configurations, including error rates, message frequencies, and priority ranges.
5. **Visual Feedback:**

- Confirms that ECUs' statuses (Idle, Arbitration, Sending, Receiving) are visually and programmatically updated correctly during the simulation.

Expected Output:

- When multiple ECUs attempt to transmit simultaneously, arbitration is resolved based on priority (lowest arbitration ID wins).
- Each ECU with a lower arbitration ID continues transmitting, while higher arbitration ID ECUs are placed in idle mode.
- Only one ECU should successfully transmit at a time, and the system handles collisions effectively without disruption.

9. Validation

1. Validation Tools and Resources

- **Simulation Logs:** Analyze detailed logs for message content, timestamps, and error statuses.
- **Error Tracking:** Utilize error counters to monitor the rate and number of erroneous messages per ECU.

2. Validation Output

The validation will produce:

1. Error rate compliance reports for each ECU.
2. Message transmission logs, showing prioritization and exclusion details.
3. Summary of ECU statuses and exclusion times.
4. Comparative results across multiple simulation runs to confirm consistency.

3. Success Criteria

The validation will be deemed successful if:

- The error rates match the predefined rates for each ECU.
- Exclusion occurs as specified after reaching error thresholds.
- The prioritization system correctly orders messages.
- The simulation achieves the expected results consistently in multiple runs.

4. Validation Image

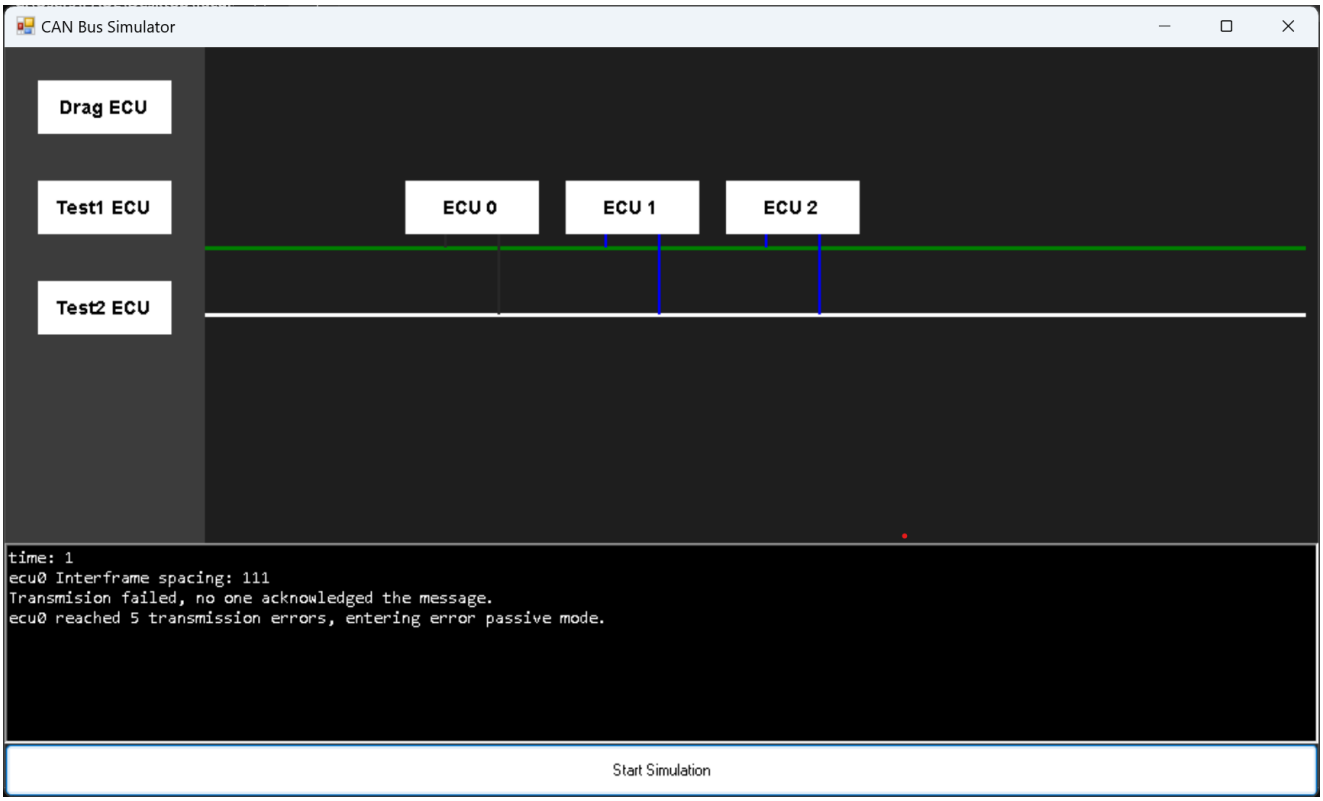


Figure 6 An ECU entering Passive Mode in Test1 after going over the error threshold

After testing multiple times the simulation gives the desired output.

Bibliography

[1] Falch Martin, "CAN Bus Explained - A Simple Intro [2024]" *CSS Electronics*, last modified July 2024. https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial

[2] Michael St. Stephen, "Introduction to CAN (Controller Area Network)" *All About Circuits*, 19 February 2019. https://www.allaboutcircuits.com/technical-articles/introduction-to-can-controller-area-network/

[3] Smith Maloy Grant, "What Is CAN Bus (Controller Area Network) and How It Compares to Other Vehicle Bus Networks" *DEWE Soft*, 13 February 2024. https://dewesoft.com/blog/what-is-can-bus