SIMULATION OF CAN-BUS USING RT-DEVS

|  |
| --- |
| Maaz Jamal (101127351) |
|  |
| Department of Systems And Computer Engineering |
| Carleton University |
| 1125 Colonel By Dr  Ottawa, ON K1S 5B6 |

ABSTRACT

CAN bus is a networking protocol that was developed for vehicles but has now found use in other applications like robotics and smart buildings. In real-life applications sensors are distributed in the system and are connected over a networking protocol to exchange data. To more closely model systems and allow simulation of nodes independently while at the same time being able to communicate over the network we have built a DEVS model of a CAN controller on the Mbed-OS platform and implemented it in hardware using RT-DEVS. To demonstrate the usage we have built a simple system modeling a vehicle with 3 nodes and communication between them.

# INTRODUCTION

The Controller Area Network (CAN) bus is a standard (ISO 11898-1) that was designed to allow for easier communication between the different nodes or sensors in a vehicle over a robust and simple network. Over the years the number of sensors in a vehicle has increased tremendously and the CAN network has become the de facto standard for vehicle manufacturers to use and allow communication between the nodes in the vehicle.

The CAN network is a broadcast-based network that allows all the nodes to have access to the packets being exchanged on the network. This allows simpler wiring as all the nodes can be connected to a single BUS and allow the Vehicle manufacturers to include an On-Board Diagnostics Port (OBD). Using the OBD all of the sensors can be tested and verified for proper functionality.

CAN bus has also found uses in other applications like building automation. (Manuel Ortiz et al 2011) implemented an HVAC and alarm automation system using CAN bus, due to the low cost and the long-range of the CAN network. (Ashtekar Shweta et. al 2011) implemented a lighting control system using the CAN bus a the backbone network.

In all of these applications, the nodes are distinct entities and they communicate with each other using CAN Bus. Simulating and testing each node with its dependency on the network is not possible without modeling and incorporating a model for the CAN controller. Modeling a CAN controller would allow each node to be tested individually. It will also allow simulation of the node with the network to be gauge the response of the node to inputs and check the outputs to the network. This can ensure the node can be connected to a physical network and still function properly.

Currently, the CAN controller is not modeled in the DEVS formalism. This prevents the modeling of a system with multiple nodes using the CAN network effectively. In this project, we will implement a CAN controller using the RT-DEVS formulism and create a model of a vehicle with acceleration and braking units connected to a motor controller over a CAN bus.

# CAN Overview And hardware

## CAN Properties

The CAN network is a broadcast-based CSMA/ RD network. The CAN network sends messages in the CAN DATA frame that is shown in figure 1. Each frame has an identifier field that allows the nodes to identify and filter the messages they need. The network assigns priority to the messages based on the identifier field. Lower id numbers have a higher priority and are allowed to be transmitted in case multiple nodes communicate at the same time.

The payload can be between 0-8 bytes (or 0-64 bytes for CAN FD) and a CRC field ensures the integrity of the data. The network is specified to be able to run at a maximum speed of 1 Mbit/s although lower speeds are also possible.

These properties provide the following benefits to a CAN network

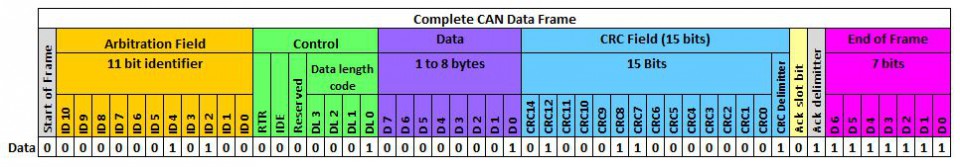
* Broadcast nature allows all the nodes to be connected to a single connection wire. In addition, this also allows a node to monitor all the nodes on the network easily.
* Reducing the speed of the network allows a longer distance between the connected nodes.
* Assigning priority based on the message ID field can allow more important and critical nodes to always send their messages without interruption.

Figure 1. CAN Data Frame. ( From Pico Technologies)

When a message is transmitted over the CAN bus it is transmitted in big-endian format. The identifier field is also called the arbitration field as this decides which message will go first. As discussed above if two messages are sent at the same time. The message with a lower id is sent first. The CAN bus consists of two lines CAN High (CANH) and CAN Low(CANL) and a differential voltage is applied across them. The line is designed such that the 0 bit is dominant during transmission and will cause any high bit to be forced to a low bit. This allows arbitration to take place over the line.

For each message sent, there is also an acknowledgment from the receiver. Any receiver node can set the acknowledgment bit to high and confirm successful transmission. If the acknowledgment is not received the transmitter asks all the other nodes to discard the previous message. The Data length code informs the receivers about the size of data in the message frame, so the frame size is variable.

The CAN bus allows the system designer to choose what they want and how they want to represent their data. The 11-bit identifiers allow the designer to choose from 2048 ids and can use the built-in arbitration to assign priority to each message. The identifiers are not assigned to a CAN port, but rather each node can choose the id for each message that it sends. This allows the designer to connect multiple sensors or actuators to a single node and then send out their messages under a different id.

# model definitions

## Complete System Model

The complete model consists of three coupled models. Each model is implemented on each node. The three models are.

1. Acceleration-Brake-Can coupled model.
2. CAN-Motor coupled model.
3. CAN-Display coupled model.

### Acceleration-Brake-CAN model

This model takes analog inputs and passes them into the network using the CAN controller. The inputs are taken from analog pins of the board and passed onto a model that converts the floating-point values to unsigned bytes as required by the CAN controller. The coupled model is defined as follows.

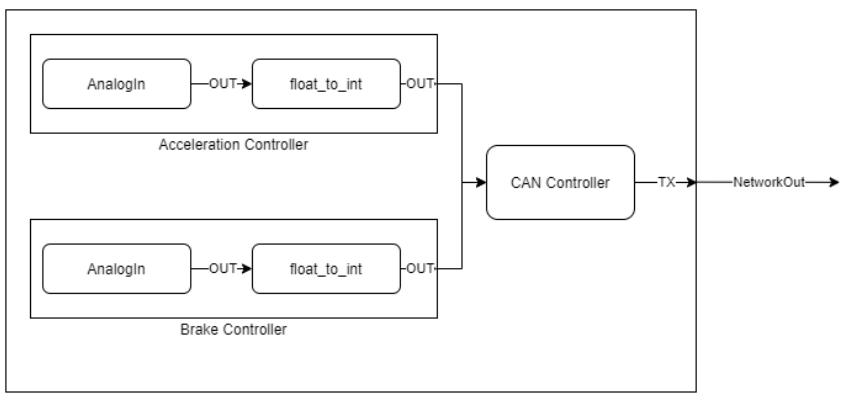
The model is shown in figure 2.

Figure 2. Accelerator-Brake-CAN coupled model.

### CAN-Motor Coupled Model

This model receives the acceleration and brake inputs from the network and passes them unto the motor which adjusts its internal parameters according to the inputs. The motor also outputs the speed of the motor to the network. The coupled model is defined as follows.

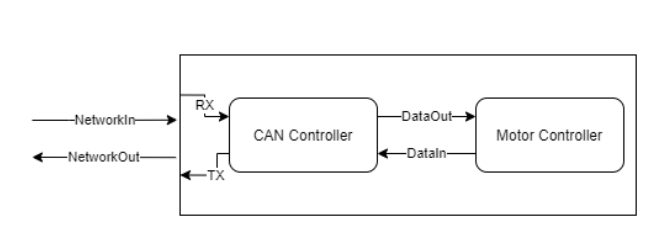
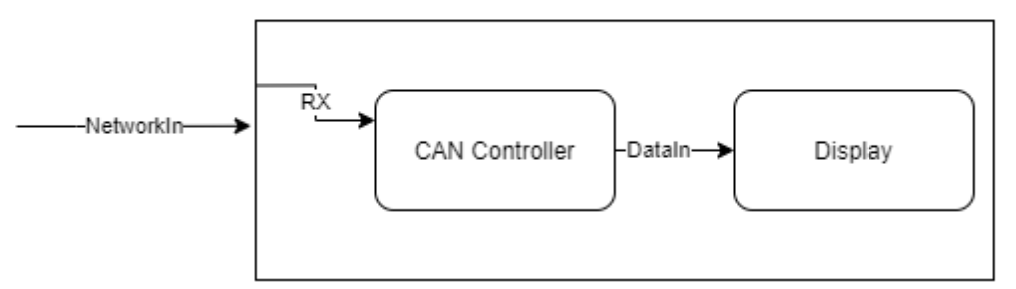
A visual description of the model is shown in figure 3.

Figure 3. CAN-Motor Coupled model.

### CAN-Display coupled model.

This model displays the values of the motors' speed, acceleration input, and brake input as received from the CAN network. The display is an LCD connected to the board, The coupled model is given as follows.

Figure 4. CAN-Display coupled model.

## Atomic Models

In this section, we will provide the formal specification of the atomic models that are used in the coupled models. The important models for this system are the

* CAN Controller model.
* Float To internal model.
* Motor Controller.

### CAN Controller Model

The CAN controller Model is responsible for reading messages from the CAN network and passing the messages to the node. The model converts the message from type CANMessage as defined in Mbed CAN API to a user-defined internal message. The internal message retains the ID and data of the message and strips the rest components. The ID is retained so that the models connected to the CAN controller can filter the messages. The Tx port is used for outputs from the network and the DataIn and DataOut ports are inputs and outputs from the rest of the atomic models on the node.. The formal specifications are given below.

if mode == 0

{

read(message)  
 if receive == true

{

Convert\_to\_internal(msg)  
 }  
else:

ta(state) = 0

}

elif mode == 1

{

set receive = false

set send = false

passivate()

}  
elif mode == 2

{

if internaldone == true

{

read(message)  
 if receive == true

{

Convert\_to\_internal(message)

set internaldone = false

}

}

elif receive = true

{

receive = false

passivate()

}

}

if x.port.DataIn

{

set internaldone = true

message = get\_message\_DataIn()

message = convert\_to\_CAN\_Message.

}

if receive = true

{

output( message, Y.DataOut)

}

elif transmit = true

{

output(message. Y.Tx)

}

The CAN controller state mode decides what the CAN controller does. If the mode is set to 0 the Controller is in read-only mode and only reads from the network and outputs the results through the DataOut port. The network is polled to read the message. If the mode is 1 then the controller is in transmit-only mode and will only transmit the internal data to the Network using the Network port. If the mode is set to 2 the network will both read and transmit on the network.

The state “internaldone” is only used during mode 2. During this mode, the model after receiving the message and passing it onto the other models in the device is passivated. The state “internaldone” represents if the other atomic models have processed the data and returned a message to transmit through the node. Hence, this state is set to false after a message is read from the network. When the model receives input from the other models in the coupled model the “internaldone” is set to true and the CAN port can start reading messages from the network again. This is done to prevent the CAN port from using up all of the CPU time.

It is important to note the CAN controller does not set the ID of the message. Neither does it strip ID information from the message. This is important as it allows the other atomic models to decide what ID they will use and how will they react to different ID messages and the information contained in them.

### Motor Atomic Model:

The Motor Atomic model is built to simulate a motor attached to and powering a vehicle. The inputs that the motor receives are acceleration and braking input from the CAN network. To simulate a vehicle in motion, the model needs to increase its velocity in increments when an accelerating input is given. Slow down when braking or decelerating input is given. Finally, if an accelerating input is given and then removed the velocity should decrease over time and reach zero after some time.

When a vehicle is in motion it is affected by a drag force due to the presence of air resistance. We can use the drag force equation from fluid dynamics to model air resistance as air itself is fluid. The drag equation is where the symbols stands for the density of the fluid, velocity of the object, drag coefficient, and cross-sectional area respectively. In this model, we can simplify the drag equation by combining into one term and keeping the velocity of the vehicle as the only variable. The drag force then becomes where is the drag weight of the model.

To complete the model we need to add the acceleration and braking inputs to the equations. Since the acceleration and the braking are forces applied to the vehicle, we can add or subtract them with the drag force on the vehicle. Since the braking force has a different intensity than the accelerating force we need to add weights to the forces. The final iterative equation then becomes ). The equation is iterative so that the speed updates gradually as happens in real-world settings. The atomic model is then.

speed = prevSpeed + ( \*accel - brake – \*prevSpeed/2

message.id = id

message.data = speed

ta(state) = 60ms

if X.DataIn.id == 1100

{

brake = x.DataIn.data

}

elif x.DataIn.id == 1200

{

accel = x.DataIn.data

}

output(message, Y.DataOut,)

The weights have been set to . The accelerating input is weighted such that more acceleration is needed to overcome the force of friction generated by the brakes. The drag coefficient is dependent on the previous velocity of the vehicle. This means that the car can speed up and the drag force will increase until the input acceleration equals the drag force. Thus constant velocity is achieved.

### float\_to\_internal Atomic Model:

This atomic model takes the floating-point inputs from the analog\_pins and converts them into an internal message format that the CAN controller can read and later send out over the can network. The float to internal sets the ID of the message. This allows it to be reused by other analog inputs and send data that can be distinguished by the other components.

transmit = false.

passivate()

data = X.DataIn.data

data = float\_to\_unsigned\_char(data)

message.len = len(data)

message.data = data

ta(state) = 10ms

output(message, Y.DataOut,)

The model expects input from the “analogInput” and waits until it receives an input from the atomic model. The model processes the data and passes this to the next model. It then passivates and waits for the next input.

### LCD\_display Atomic Model

The LCD\_display atomic model acts as a wrapper for TextLCD library. The model accepts input from the CAN bus and deciphers the message source from the ids.

write = false.

passivate()

msg = X.DataIn.data

msg.data = convert\_to\_percentage(msg.data)

if (msg.id == 1100)

{

message.brake = msg.data

}

else if (msg.id == 1200)

{

message.accel = msg.data

}

else if (msg.id == 1300)

{

message.speed = msg.data

}

clear\_screen()

output(message, Y.DataOut,)

The model clears the screen before sending out a new message to replace the text on the screen. The text is preformatted and only the values of the relevant fields are changed to express an update.

# Simulation Results

The simulations are carried out on software and hardware to verify proper working. To test for proper working the following tests were carried out:

* CAN testing in software.
* Motor simulation in software
* Hardware simulation of the complete system

## CAN Testing:

This simulation tests the CAN transmission and receiving of the models. This scenario also tests the float\_to\_internal atomic models as they are used in the simulation as well. The test scenario is set up like the coupled model Acceleration-Brake-CAN model discussed in 3.2.1.

The model receives inputs from AnalogIn and passes them on to float\_to\_internal atomic model which passes them out to the CAN atomic model which passes them to the CAN bus. The inputs to the model sweep from 1.0 to 0 and then back to 1.0 in increments of 0.1. The results are displayed in table 1.

| Analog Input | Float\_to\_internal | | CAN1 TX output | | CAN2 DataOut | |
| --- | --- | --- | --- | --- | --- | --- |
| ID | Data | ID | Data | ID | Data |
| 1.0 | 1100 | 255 | 1100 | 255 | 1100 | 255 |
| 0.9 | 1100 | 229 | 1100 | 229 | 1100 | 229 |
| 0.8 | 1100 | 204 | 1100 | 204 | 1100 | 204 |
| 0.7 | 1100 | 178 | 1100 | 178 | 1100 | 178 |
| 0.6 | 1100 | 153 | 1100 | 153 | 1100 | 153 |
| 0.5 | 1100 | 127 | 1100 | 127 | 1100 | 127 |
| 0.4 | 1100 | 102 | 1100 | 102 | 1100 | 102 |
| 0.3 | 1100 | 76 | 1100 | 76 | 1100 | 76 |
| 0.2 | 1100 | 51 | 1100 | 51 | 1100 | 51 |
| 0.1 | 1100 | 25 | 1100 | 25 | 1100 | 25 |
| 0.0 | 1100 | 0 | 1100 | 0 | 1100 | 0 |
| 0.1 | 1100 | 25 | 1100 | 25 | 1100 | 25 |
| 0.2 | 1100 | 51 | 1100 | 51 | 1100 | 51 |
| 0.3 | 1100 | 76 | 1100 | 76 | 1100 | 76 |
| 0.4 | 1100 | 102 | 1100 | 102 | 1100 | 102 |
| 0.5 | 1100 | 127 | 1100 | 127 | 1100 | 127 |
| 0.6 | 1100 | 153 | 1100 | 153 | 1100 | 153 |
| 0.7 | 1100 | 178 | 1100 | 178 | 1100 | 178 |
| 0.8 | 1100 | 204 | 1100 | 204 | 1100 | 204 |
| 0.9 | 1100 | 229 | 1100 | 229 | 1100 | 229 |
| 1.0 | 1100 | 255 | 1100 | 255 | 1100 | 225 |

Table 1. Simulation output

The results show that the analog inputs are converted from floating point to unsigned char. The CAN bus sends and transmits data by dividing it into byte-sized chunks, so the most obvious data type to use is the unsigned char data type. This data type represents 1 byte and has a value ranging from 0-255. The float\_to\_internal converts the float to unsigned char and adds an id to the message. The message is stored in an internal data structure that contains the id, data, and length of the data. This is passed to the CAN1 atomic model using the DataIn port. Which sends the data out through the TX output port. The CAN2 atomic model receives this input and then sends it out using the CAN2 DataOut, where it can be used by subsequent atomic models to do something with the data.

We can see the float values are converted to 1 byte unsigned char values and passed to the CAN1 model which then passes them out to the CAN2 model, which receives the data as it was sent out by the CAN1 model.

## Motor Simulation

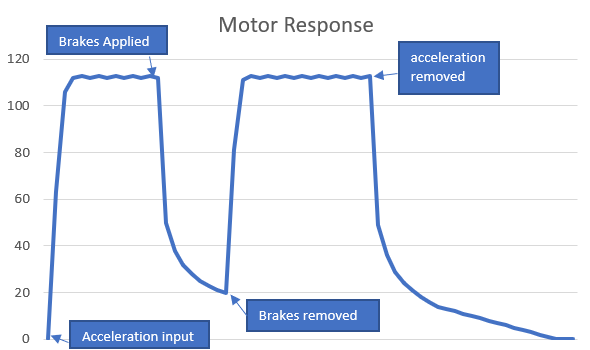
For the motor simulation scenario, we test a node with a similar setup as in section 3.1.2. Here we are testing the ability of the motor to speed up incrementally if an accelerating input is required, slow down incrementally if a decelerating input is applied, and slow down if accelerating input is removed. We also check if the motor is returning the speed it has calculated.

To check this we apply an input at the full acceleration in the beginning and hold it there. We then apply a braking input at half the intensity of the accelerating input and then remove the braking input. We finally remove the accelerating input and check the resulting speed.

The resultant graph of this simulation can be found in figure 5. We can see that when the accelerating input is applied the motor reaches the maximum speed and stay there. When braking input is applied the speed falls incrementally. We then remove the braking input and see that the speed increases back to its top speed. Finally, the accelerating input is removed and we can see that the speed falls due to drag force until eventually reaching zero.

If we analyze the log files we can see that the motor receives the input calculates the speed and then return that message to the CAN controller for further transmission on the CAN bus. Since the CAN controller is in mode 2 the CAN controller receives a message and then stops transmitting until the other model returns data through the DataIn port, in this case, the motor atomic model. The first few outputs from the logs are shown in table 2 with extra information removed.

The ID 1200 represents the ID of accelerating input. The ID 1300 is reserved for the Motor data. The input for acceleration is applied once and then received by the CAN port, the motor upon receiving the input starts calculating its speed iteratively. Hence why there is only one entry for the Input and CAN DataOut and multiple entries for Motor Output and CAN TX.

Figure 5. Motor speed output graph.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Input | | CAN DataOUT | | Motor Output | | CAN TX | |
| ID | data | ID | Data | ID | Data | ID | Data |
| 1200 | 255 | 1200 | 255 | 1300 | 0 | 1300 | 0 |
| 1300 | 63 | 1300 | 63 |
| 1300 | 106 | 1300 | 106 |

Table 2. Selected motor log outputs.

## Hardware Simulation:

The hardware simulation was carried using the setup shown in figure 7. The log outputs were recorded over serial USB into a computer. Three log files were obtained one from each of the nodes:

* Accelerator and Brake unit.
* Console/Display node.
* Motor node.

The accelerator and brake unit CAN port is running in mode 1, meaning the node only transmits Data. The console/display unit is running in mode 0 and only receives inputs from the network and finally, the motor node is running in mode 2 and receives and sends information.

Analyzing the log file from the accelerator unit we can see the log shows the input from the two analog pins and the outputs from the float\_to\_internal and CAN port. However, we do not see any input from the CAN port as it should be as this node is in mode1. The log for the CAN port message does not show any values as in this case it uses the Mbed OS representation of CANMessage and the logger cannot decipher the message. Table 3 shows some of the outputs from this node.

| Analog Input Acceleration | Acceleration Output | | Analog Input Brake | Brake Output | |
| --- | --- | --- | --- | --- | --- |
| ID | Data | ID | Data |
| 0.165079 | 1200 | 42 | 0 | 1100 | 0 |
| 0.388278 | 1200 | 99 | 0.0019536 | 1100 | 0 |
| 0.784615 | 1200 | 200 | 0 | 1100 | 0 |
| 1 | 1200 | 255 | 0.0981685 | 1100 | 25 |
| 0.999756 | 1200 | 255 | 0.610745 | 1100 | 155 |
| 1 | 1200 | 255 | 0.999512 | 1100 | 254 |

Table 3. Logged data from Accelerator and Brake node.

Table 3 represents twisting the potentiometer to a minimum resistance for the accelerator pin and then doing the same for the brake analog potentiometer. The analog input shows the intensity of analog input at the pin while the other column shows the converted values. If we look at entries 4 and 5, we can see that 0.98 gives us 25, which is approximately 1/10 of 1.0 and 255. In the same manner, the 0.610 is converted to 155 which is approximately 60% of 1.0 and 255 so we can conclude this node is working as intended.

Moving onto the motor node log files ( a small selected subset is shown in table 4). We can see the inputs from the accelerator and brake node are successfully reaching the node over the CAN bus. The motor is calculating the speed and then transmitting the values over the CAN bus.

|  |  |  |  |
| --- | --- | --- | --- |
| Motor model Output | | CAN DataOut | |
| ID | Data | ID | Data |
| 1300 | 0 | 1200 | 254 |
| 1300 | 62 | 1100 | 0 |
| 1300 | 105 | 1200 | 254 |
| 1300 | 113 | 1100 | 0 |
| 1300 | 113 | 1100 | 0 |
| 1300 | 112 | 1200 | 23 |
| 1300 | 55 | 1100 | 0 |
| 1300 | 45 | 1200 | 0 |
| 1300 | 34 | 1100 | 0 |
| 1300 | 28 | 1100 | 0 |

Table 4. Log output of the motor node. Some data points have been removed for sake of brevity

We can see the CAN port is receiving both accelerating and braking inputs. In this subset, the braking input is set to zero while the accelerating input is set to max and then set to zero again. The motor calculates this speed and sends it back to the CAN port which transmits it with ID 1300. Due to limitations of the buffer size of the logging software a complete log of the output of the motor was not achieved however if we plot the data that we have as shown in figure 6. We can see the plot resemble what we obtained in figure 5 for maximum acceleration and then removing the accelerating input.

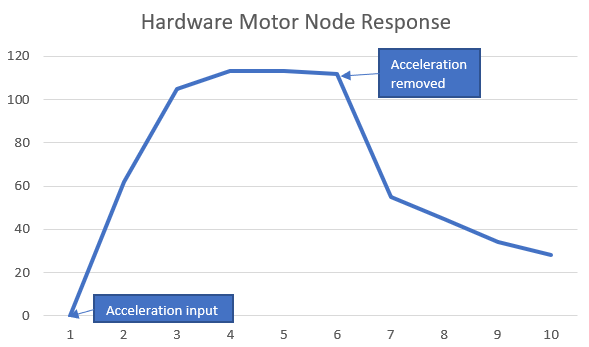


Figure 6. Hardware motor node simulation.

Finally, we move to the console unit logs. This node receives inputs from the other two nodes and tells us that the network is working properly. A small subset of the logs is displayed in table 5. We can see the console node is receiving all the outputs from the other two nodes and hence we can conclude that the network is working properly.

|  |  |
| --- | --- |
| CAN DataOUT | |
| ID | Data |
| 1300 | 66 |
| 1200 | 154 |
| 1300 | 70 |
| 1100 | 24 |
| 1300 | 72 |

Table 5. Console log.

# Hardware Overview

To make a CAN network we need a CAN controller and a CAN transceiver IC. The transceiver IC converts the digital serial inputs from the CAN controller into physical voltage levels. The physical CAN network is composed of two wires with a differential voltage applied across them. In this project, we used an MCP2551 transceiver IC as it has low RFI emissions and high noise immunity while supporting up to 1Mbit/s communication. The low RFI emissions reduce the introduction of noise into the network or the CAN controller attached to the IC.

The CAN controller used in this project is a NUCLEO-F207ZG board running Mbed OS. The board contains two CAN port and large ROM and SRAM allowing larger more complex models to be loaded onto the system The board also uses Mbed OS, which allows us to use the RT-CADMIUM library to model the system using DEVS and port the models over to the hardware.

The Mbed OS takes care of low-level details like arbitration and sending acknowledgments and allows the programmer to focus on sending and receiving to the CAN Bus.

## Hardware Setup

In this project, we use three boards to form nodes and exchange information between them. The three boards represent a console unit, a motor unit, and the accelerator and brake unit.

The console unit is used to process and display the information being passed through the CAN bus from the other units. The motor unit simulates a motor operating in a vehicle and simulates the effects of acceleration, drag, and braking forces. The accelerator and brake unit converts analog inputs to CAN messages and passes them on to the motor node.

The accelerator and brake units take analog inputs from analog pins and can be changed using a potentiometer. The inputs have separate ids and are sent over the CAN network, allowing the motor unit to receive and at the same time be able to distinguish between the two inputs. The motor unit uses this information to compute the speed and send this information back over the CAN network. The three nodes are shown in figure 7.

|  |  |
| --- | --- |
| a) | b) |
| c) | |

Figure 7. Pictured above, a) Motor node b) Accelerator & Brake node c) Console Unit

|  |
| --- |
|  |

Figure 3. The complete system with 3nodes and connections between them

In figure 3 we can see the complete system. The nodes are connected over the CAN network. The boards have two pins TX and RX pins for the CAN port. The TX pin is used to send data to the CAN transceiver, the RX pin receives data from the transceiver. Both of these pins are digital outputs and serial.

The TX and RX are connected to the respective pins on the MCP2551 IC. The two wires leaving the IC are CANH and CANL wires and together they carry each message. As stated above the CANH and CANL have differential voltages applied to them si=o even though they may be two wires they still carry data serially. The wires can be twisted to form a twisted pair wire, this can reduce noise in the wires and allow the signal to be carried over long distances and at faster speeds. However, that is not necessary in this case.

# Discussion

The simulation and modeling of the CAN controller allow us to distribute a system model over multiple nodes and more closely model the system with actual hardware implementation. CAN Bus was chosen due to its long-standing use in vehicles and in recent years its adoption in smart buildings is growing.

The implementation of the model in DEVS presented a few challenges. The CAN controller should ideally have an RX port from which it receives input from the CAN bus. However, modeling this on hardware proves difficult as the CAN bus cannot be coupled with the CAN controller model. This is because the CAN transceiver is a separate physical entity. It will not provide any external inputs to the system that can be routed to the input ports.

One solution to this problem is to read the CAN bus in the internal transition function and set the time advance function to 0. This causes the program to repeatedly check the line and if an input is received we can change the time advance function to a finite value and be able to send the output to the other models. While this works, with certain limitations, on hardware. During software simulations the fact that the time advance function is set to 0 the model does not proceed and is stuck at that point forever.

It causes certain problems in hardware as well, the NUCLEO-F207 board has two CAN port per board. If one port is set to read the line in this fashion the other port becomes useless and cannot be used for any other purpose. In internal testing, the coupled model was not able to proceed as it was waiting for input from the line. This then means that using this method the model can only be used to read from a line and any internal action can only be triggered only if something is read from the line.

The different modes were created to tackle this problem so that each node can only perform the function that it has to do. Mode 1 transmit the only mode was easy to implement and did not cause any significant problems.

Mode 2, however, requires some thought. If we are to receive and send from the same node then we need to add a delay after input is received. If we add a finite delay, the model will wait for some time and perform an output. The model will then go back to polling the line and prevent the other models from proceeding and using the CAN port to transmit data. The solution is to passivate the model after it has performed an output and then wait until the rest of the models have advanced and returned what they want to send. The CAN port is then allowed to transmit the data and then move back to polling the input line.

This method however still prevents the node from generating its output on its own and is dependent on input from the CAN bus to generate an output. This is not a problem in this scenario but maybe a problem in other cases.

The motor model iteratively computes the speed by taking into account the acceleration, braking, and drag force on the vehicle. The steep velocity curve can be reduced by reducing the weight of the accelerator (and braking) coefficients so that for a given input a smaller amount is added to the previous velocity. The model can be improved by adding the effect of gears on top speed by changing the coefficients of the model. In the same vein, a more accurate equation for the drag force and its coefficients can more closely model a vehicle in motion.

# COnclusion

In this report, we implemented a CAN controller using DEVS formalism in software and then simulating the CAN controller on RT-DEVS. The models were implemented on NUCLEO-F207Z boards running Mbed-OS and CADMIUM simulation tool. Implementing the CAN bus now allows us to simulate boards independently and still allow communication between the boards on actual hardware. A vehicle with an accelerator, brake unit, a console unit, and a motor unit was simulated and the results showed that the CAN controller can successfully run in each of the three modes (receiving, transmitting, and both). The accelerator brake unit ran in sending mode, the console unit ran in receiving mode, and the motor node in both mode. The logs for the console unit showed us that the other two nodes were successfully receiving and sending data over the CAN bus and that the console unit was able to receive the data. The models can be further improved by solving some issues. The current method of polling the line prevents the model from proceeding and makes the other model dependent on the input from an external source to proceed with their simulation. Using an interrupt based model that use interrupt signals from the internal registers can allow the rest of the models on a node to proceed with their simulation and not be reliant on receiving a message from the node.

# REFERENCES

1. *Road vehicles — Controller area network (CAN)*, ISO 11898-1, 1993
2. Ortiz M., Diaz M., Bellido F., Saez E., Quiles F. (2011) Smart Home Automation Using Controller Area Network. In: Abraham A., Corchado J.M., González S.R., De Paz Santana J.F. (eds) International Symposium on Distributed Computing and Artificial Intelligence. Advances in Intelligent and Soft Computing, vol 91. Springer, Berlin, Heidelberg. <https://doi-org.proxy.library.carleton.ca/10.1007/978-3-642-19934-9_21>
3. Shweta S.A., Mukesh D.P., Jagdish B.N. (2011) Implementation of Controller Area Network (CAN) Bus (Building Automation). In: Unnikrishnan S., Surve S., Bhoir D. (eds) Advances in Computing, Communication and Control. ICAC3 2011. Communications in Computer and Information Science, vol 125. Springer, Berlin, Heidelberg. <https://doi-org.proxy.library.carleton.ca/10.1007/978-3-642-18440-6_64>
4. P. (n.d.). CAN and CAN FD bus decoding. Retrieved January 10, 2021, from https://www.picotech.com/library/oscilloscopes/can-bus-serial-protocol-decoding