ADVANCED AUTOMOTIVE ELECTRONIC ENGINEERING

Compliance Design of Automotive Systems

Modelling of Traffic on a CAN Network

Presented by: Enrico Marini

Emiliano Tomaro

Massimo Toscanelli

Supervisors: Prof. Carlo Concari

Prof. Alessandro Soldati

Academic Year 2018/2019

Index

1. Introduction to Controller Area Network(CAN)

The CAN is a protocol largely used in automotive that allows a robust communication between several ECUs and sensors on board a vehicle. Each ECU (e.g. engine control unit, airbags, automatic transmission, ESP, audio system) represents a node connected to the CAN bus. Nowadays, cars have around 70 ECUs.

CAN system is not a point-to-point communication system, it is instead a broadcast transmission typology, making it a smarter, cheaper and lighter solution than point-to-point.

The main purpose of the CAN is to allow any ECU to communicate with the entire system without causing an overload to the controller computer. The ECUs communicate through a single CAN interface and the system allows for central error diagnosis and configuration among all ECUs. Every node receives all transmitted messages, decides relevance and chooses whether it needs the data on the bus or not.

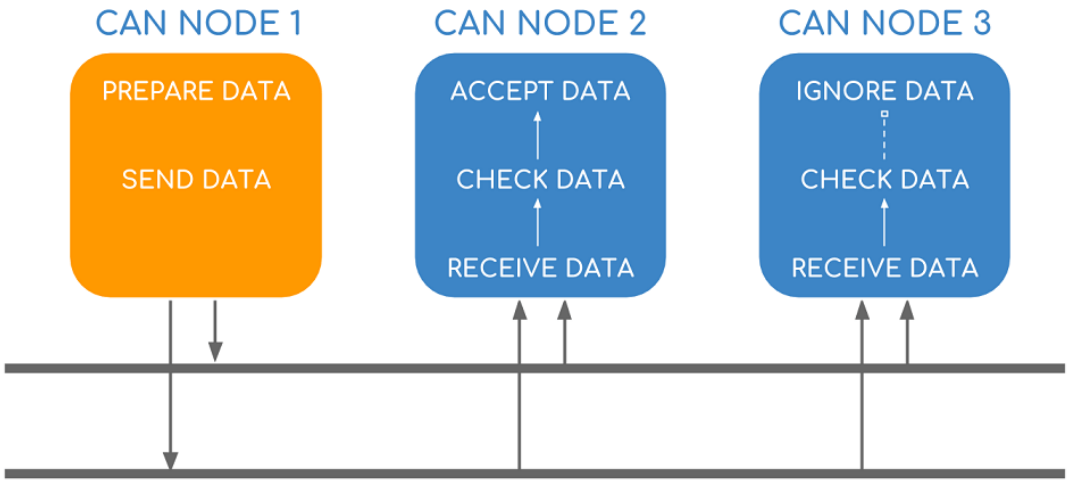

Figure 1: CAN Bus

CAN messages are prioritized according to each node's ID (lower ID means higher priority), making it a very efficient system. Furthermore, it is a flexible protocol since it is possible to include additional nodes over time.

Each ECU reads the bus through a buffer and each ECU writes on the bus through a transistor. The bus is also called a "wired AND" because the 0 level (0 V ground, logical bit value 0) is the dominant level: if one ECU writes a 0 on the bus, the logical value on the bus will be a 0 regardless of what other ECUs are writing.

If two different nodes are waiting for another to end transmission, they will probably start transmitting together once the bus is free. To avoid collisions on the bus, bitwise arbitration is used: all the ECUs with a transmission request start simultaneously to send the identifier of their respective CAN message to be transmitted, bitwise from the most significant to the least significant bit. Knowing that a 0 bit is dominant on the bus, each ECU compares the value in the bus with the value it sent (bit monitoring) and if it reads a 0 when it pushed a 1 it means that there is another ECU with higher priority that is willing to transmit - this is

why lower IDs have higher priority. Lower priority ECU will then stop transmitting and start listening to the higher priority ECU transmitting.

The whole message sent by a node could be subdivided into 8 different blocks:

- Start Of Frame (SOF): a dominant 0 to tell the other ECUs that a message is coming;
- Identifier (ID): gives each message a priority;
- Remote Transmission Request (RTR): a forced transmission from other ECUs;
- Control: informs the length of the data expressed in Bytes (0 to 8 B);
- Data: the actual information;
- Cyclic Redundancy Check (CRC): check used to ensure data integrity (i.e. detects errors);
- Acknowledgment (ACK): a bit that indicates if the CRC process is correct;
- End Of Frame (EOF): marks the end of a CAN message.

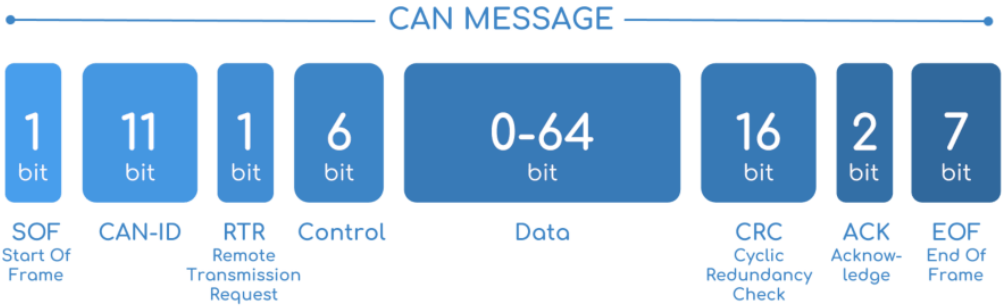

Figure 2: CAN Logical Frame

At the end of a message, 3 bits of interframe are used as well.

Not to lose synchronization among ECUs in long consecutive bit sequences, bit stuffing is performed by the transmitting node from the beginning of a message up to the CRC (not the CRC delimiter). Stuffing adds an opposite bit after five consecutive equal bits (purple bits in the following figure).

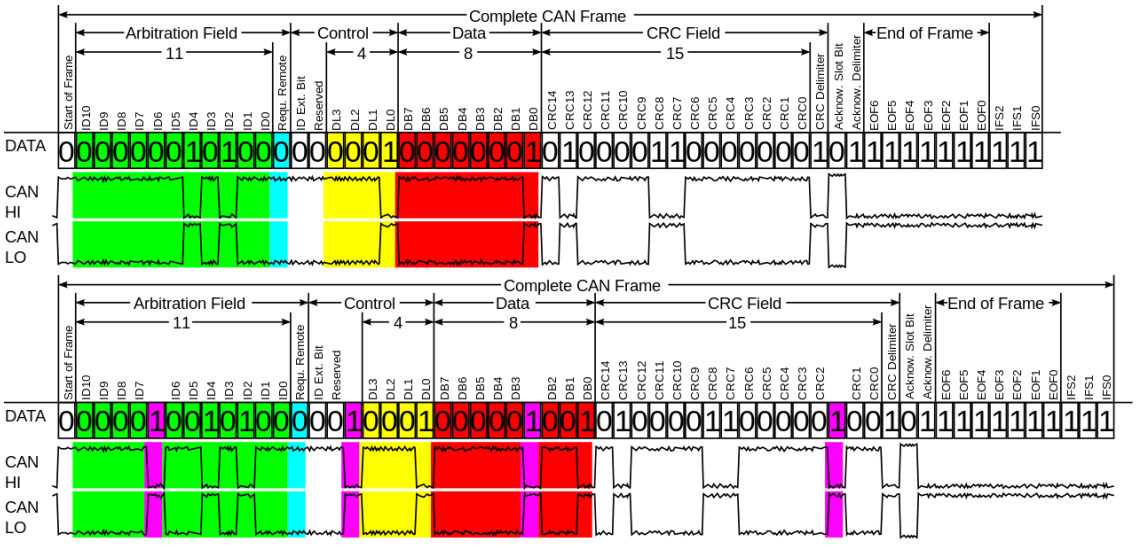

Figure 3: CAN Physical Frame

CAN bus's speed is 1 Mb/s for a 40 m long bus. The speed decreases with longer buses to 40 Kb/s in a 1 Km long bus.

1.1 CAN Open

The push for increased vehicle functionality may require changes to the core CAN technology. Increased functionality could for example mean an increase in the data load or a way not to leave precious data unharvested.

The former is allowed by CAN-FD (i.e. Flexible Data-rate) which increases the payload by a factor 8 and allows for a higher data bit rate (i.e. up to 8 Mb/s with a with a payload of 64 B).

The latter is allowed by CANopen, used to optimize standard CAN applications. CANopen is extensively used in industrial robotics, production machinery, medical equipment and speciality vehicles because they all need a kind of memory built-in. The collection of data is very important - even if you don't really know how to use those data - and CANopen, built on CAN protocol, can, in addition, record data: configurations and processed data are stored in a table called Object Dictionary.

In CANopen the 11-bit CAN ID is split into two parts: a 4-bit function code and a 7-bit CANopen node ID (this limitation restricts the amount of devices on the network to 127). These two blocks, followed by a 1-bit RTR, form the COB-ID.

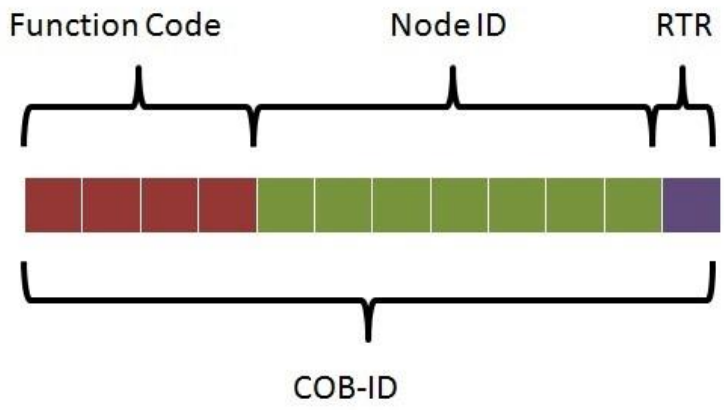

Figure 4: CANopen, COB-ID

All COB-IDs must be unique to prevent conflicts on the bus. Functions of the COB-ID include:

- Service Data Objects (SDO): it handles read/write requests allowing a master (the SDO client) to grab data from a node;
- Process Data Objects (PDO): it represents data that could be changing in time such as inputs from sensors;
- SYNC: periodical messages used to send PDOs synchronously;
- Network Management (NMT): used to change the configuration of a slave node between initializing, pre-operational, operational and stopped;
- EMCY: it communicates node's status and information about errors that may have occurred.

Also, the CAN data section is different in CANopen since the frame is split into three parts: 1 B for the specifier (it indicates what type of message is being transferred, what's its length and the type of transfer), 3 B for the node index and sub-index and 4 B for the actual data transfer.

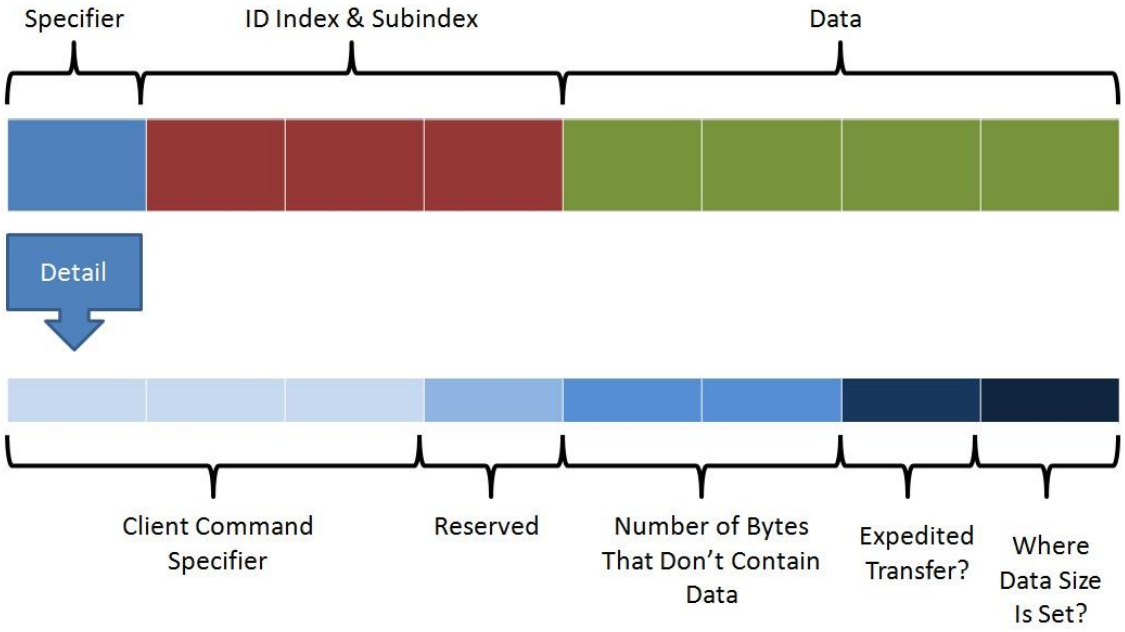

Figure 5: CANopen payload

SDO is the mechanism for which a node will send a request to the network and the node of interest will respond with the data requested. When the SDO client wants to request some information, it sends a request using a COB-ID of 600h + the slave node ID. The server will respond using a COB-ID of 580h + its node ID. SDO communication only allows access to one Object Dictionary at a time. If the data that need to be transferred do not fit a single message, a segmented transfer (i.e. opposite of expedited) is selected and data is transferred using multiple messages.

PDO is the mechanism for which data is sent as soon as it is ready. There are two types of PDOs which are transfer PDOs (TPDO, produced by the node) and receive PDOs (RPDO, coming to the node). There are different methods through which a TPDO could start such as time driven events and polling: TPDO is initiated at a fixed time interval (using a pre-configured number of SYNC signals) or when the process data in the node changes. TPDOs and PRDOs' COB-IDs are multiple in order not to have a lot of overhead for accessing continually changing data. They are:

TPDO	180h + node ID	280h + node ID	380h + node ID	480h + node ID
RPDO	200h + node ID	300h + node ID	400h + node ID	500h + node ID

The SYNC protocol provides the basic network synchronization mechanism, since it triggers the synchronization periodically, compensating the indeterminism of time in CAN. The transmission period of SYNC is configurable. The COB-ID used by SYNC is 080h.

2. CAN Node Model

We wanted to obtain an implementation of the CAN network that was as modular as possible. That is why we decided to implement single CAN nodes that have been reused in our model to simulate different nodes of the same network.

A single node is made of three main sub-blocks: Transmission, Node Logic and Receiver.

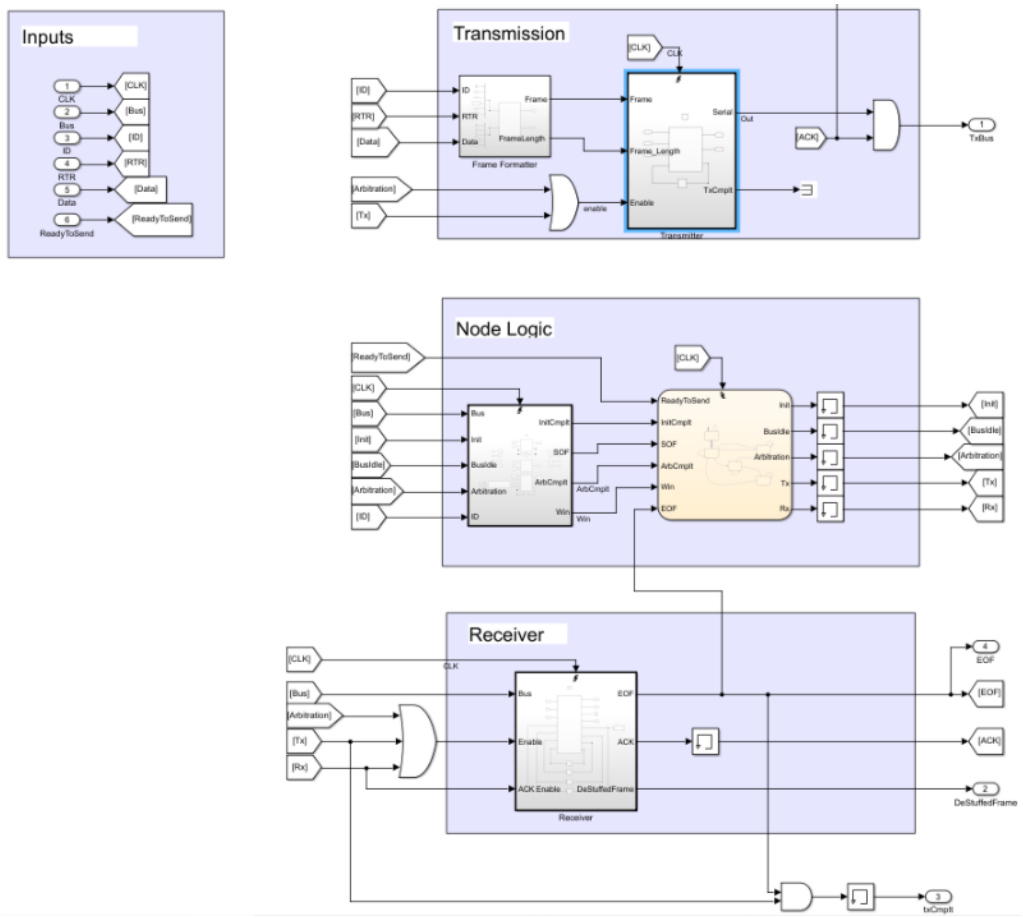

Figure 6: CAN Node

The first one is needed to describe the general transmission of the node, form its logic (frame creation) to its physical implementation. The second one acts as controller of the node functionalities. The third one represents the physical receiver.

2.1 Transmission

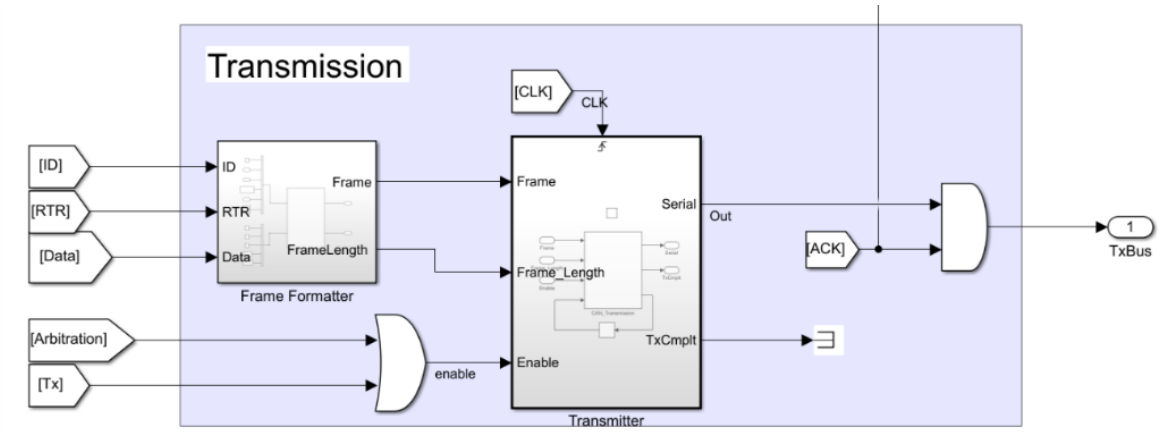

Figure 7: CAN Node, Transmission

In this part of the logic we have the FrameFormatter. It is used to generate the frame that our CAN node will be able to transmit. Inside this block we find a FrameStuffer that receives as input two vectors: the former contains the part of the frame that must be stuffed, the latter just the fixed part that will directly flow in the bus.

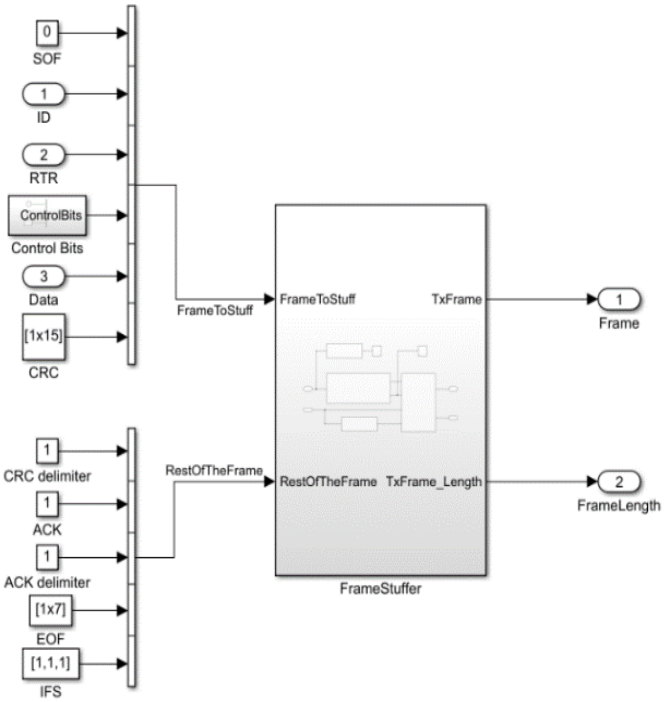

Figure 8: Frame Formatter

The output of the entire FrameFormatter block enters in the Transmitter one, where we have the physical transmission implementation of the node. This block is enabled only when we are arbitrating or transmitting ("or" of arbitration and transmission states of the state machine). When di enable is turned off the transmitter's index of the buffer goes back to the initial position, thus, in case of arbitration lost, at the next try, it will start back from the first position. Its output is not directly connected to the bus, but it is in "and" with the ACK signal generated by the Receiver block (that is the only one who knows exactly when to send it).

2.2 Node Logic

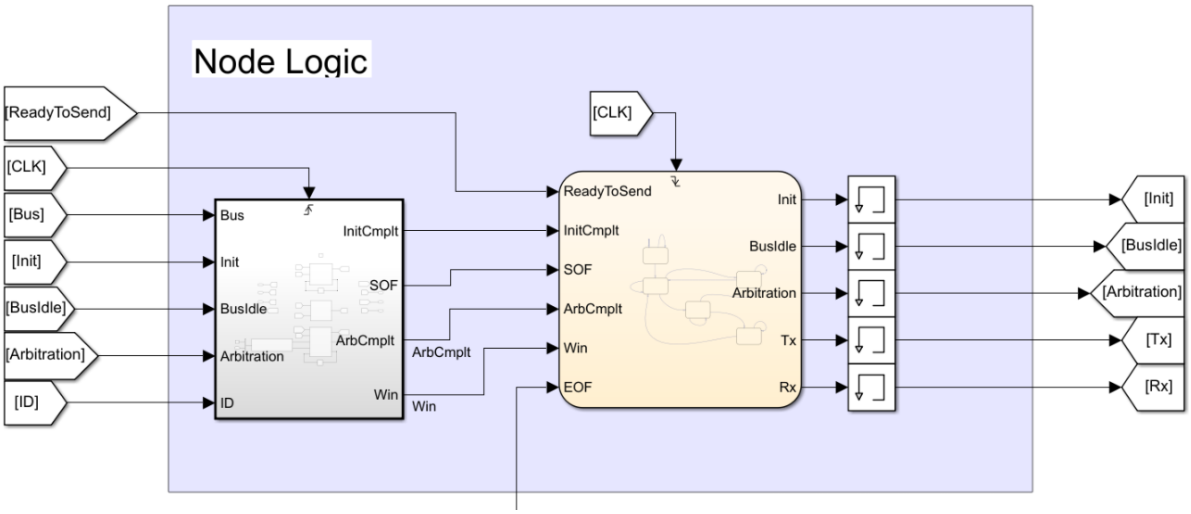

Figure 9: CAN Node Logic and FSM

The logic of the node is controlled by a finite state machine that uses parameters (i.e. ReadyToSend, SOF, ArbCmplt, Win, EOF) as flags raised to understand when it is time to move from a state to another. Combinatory logic of the state machine is implemented in the block on the left side.

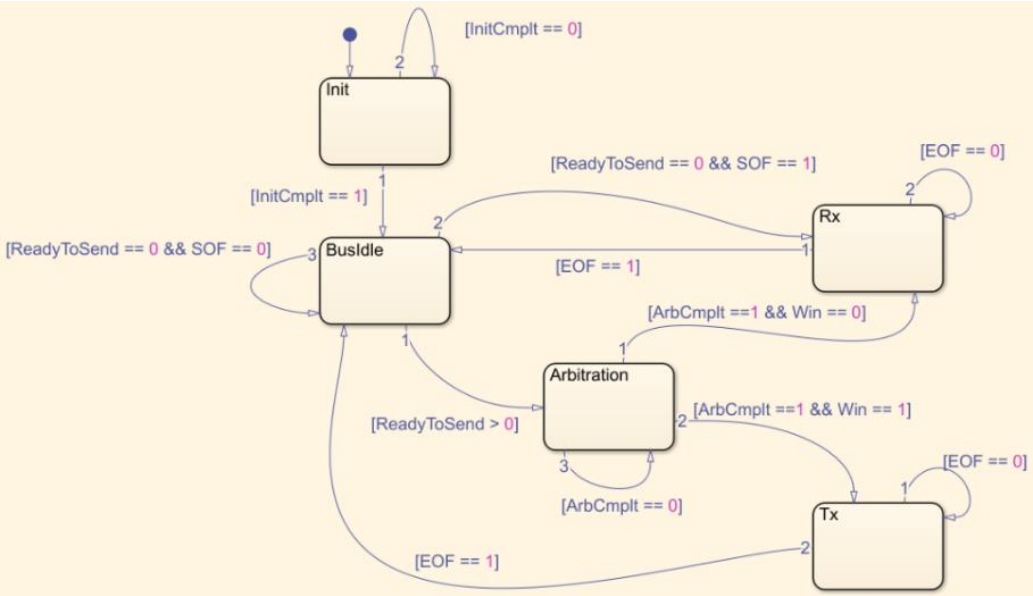

Figure 10: FSM State Diagram

The state machine starts in the Init (Initialization) phase and then, as soon as the bus is idle, it goes in BusIdle state. Here it waits for raise of two parameters: ReadyToSend or SOF. If SOF becomes 1, it means that there is an incoming packet and the machine goes in Rx state to read it, otherwise, if the value of ReadyToSend is greater than 0 (we want to transmit one or more packets), it goes in Arbitration phase. From Rx we can only go back to BusIdle only if the frame is completely received (not to lose synchronization with other nodes). From arbitration phase instead, we can move in Rx if we lose the arbitration with other nodes, or we can go in Tx (transmission) state to start our frame transmission and then come back to the BusIdle state.

2.3 Receiver

Reasoning on this machine structure we understood that the node would have needed a physical receiver implementation enabled in three states: Rx, Tx, Arbitration.

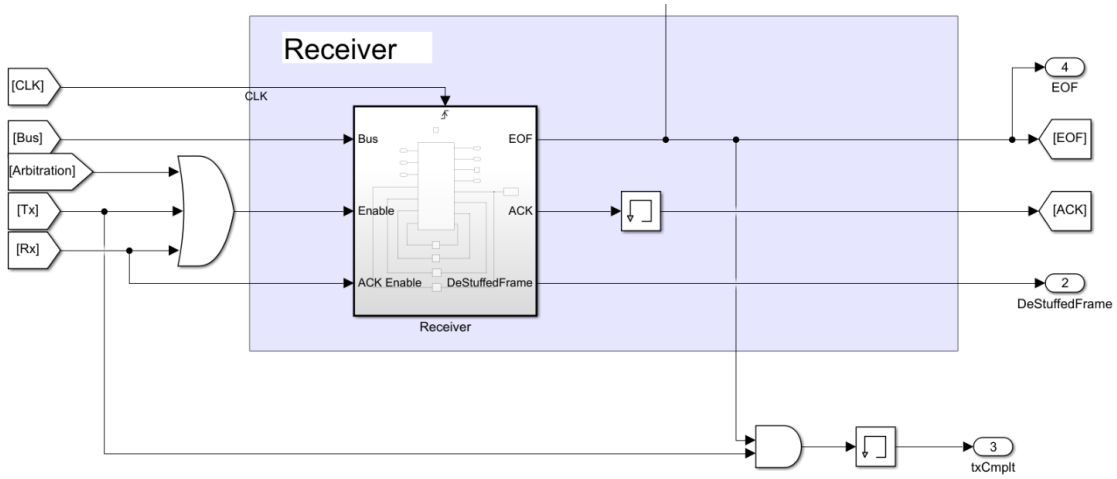

Figure 11: CAN Node, Receiver

On the Receiver side we have just a block that represents the physical receiver implementation. It must keep trace of everything is flowing through the bus, that is why we enabled it during Arbitration, Tx and Rx phases. The Receiver has three main jobs: DeStuffedFrame generation, ACK sending at a very precise position inside the frame and EOF signal generation. EOF must not be confused with the EOF bits of a CAN frame, EOF signal is a flag used to change the state of our state machine notifying that the bus is no more busy. Its “and” with the Tx state of the machine is used to generate a transmission completed signal.

This block was the most difficult to implement because a receiving node does not know the length of a stuffed frame in advance, so it cannot simply have a buffer that stores a vector with predefined length and then destuff it. Considering that, we decided to start receiving packets destuffing them on every cycle until we obtain a 98-frame length, after that we simply continue to append bits inside the vector knowing exactly where to send the EOF and eventually the ACK signals. When EOF is triggered DeStuffedFrame is ready to be read from the receiving node.

3. CAN Network Model

The creation of the network starts with the design of one single node, which is then replicated numerous times in order to simulate the functioning of hundreds of nodes all connected and communicating with the network.

Each node has the clock (CLK), the data that is on the bus (Bus) and a unique ID as input. The output of each node is connected to a AND port ("wired AND"), which output is what is found on the bus.

The model is structured to analyze:

- Typical Response time
- Delay between the logical and the physical transmission
- Jitter of periodic signals

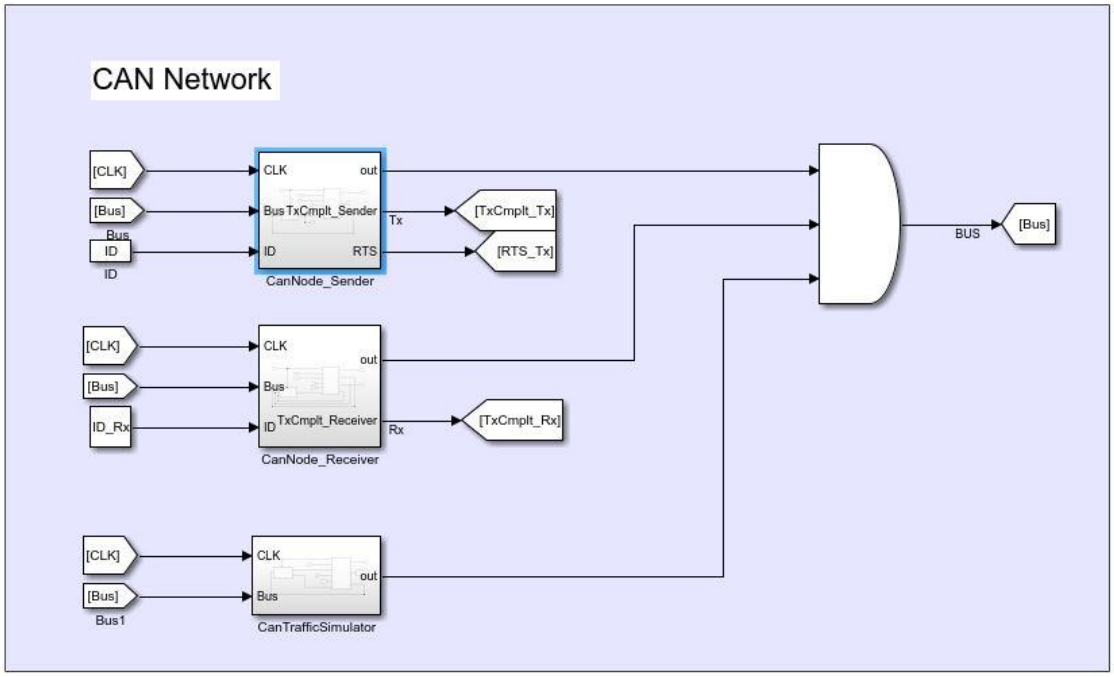

Figure 12: CAN Network

The nodes use for this purpose are three: one Sender, one Receiver and one Traffic Simulator.
These are all structured with the same low-level structure (physical and control layers), with the addition of different logical parts in order to manage them.

- <u>CAN Node Sender:</u>
  Takes care of transmitting periodic packets according certain parameters modifiable in Matlab.
  It consist in a counter which increments the ReadyToSend flag of the Can Node, this value is then decremented at the end of the transmission. In case of congested network, where the node's ID does not allow the transmission, this value keeps increasing.

- CAN Node Receiver:
  Is always in receiving mode, waiting for a request by means of the Transmitting Node. When this packet arrives, it tries to reply transmitting a certain payload on the bus.
  This node has been used to estimate the typical response time.

- CAN Traffic Simulator:
  This node tries to simulate a generic Can traffic on a certain Network, given as input the BusLoad (0% to 100%) and a list of Can IDs.
  This nodes logic rises a ReadyToSend flag depending on the BusLoad value, it is a proportional equation related on the number of packets on the bus in a certain simulation time.
  Every time it succeeds in transmitting a packet, the ID changes randomly among the ones in the list.

4. GUI - Graphical User Interface

In CAN protocol, the ID of each node is buried inside each ECU. Instead, we decided to give the ID as input, through a graphic user interface (GUI), in order to simulate as many cases as possible using two nodes: one that simulates the node we want to study and test, while the other simulates the traffic of the network by changing its ID every time a transmission is completed – it simulates all the other nodes of the network. This allows us to consider both higher and lower priorities and a customizable number of nodes forming the network.

First, the user will enter the ID of the node that has to be tested in terms of transmission delay, due to priority and traffic congestion of the network. ID must be entered with a blank space in between each bit (e.g. '0 0 0 1 0 0 1 0 1 1 1').

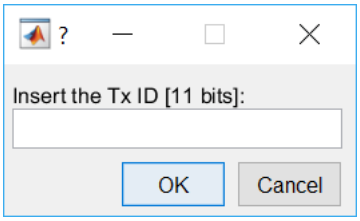

Figure 13: GUI, ID request

More dialog boxes will guide the user through, who is able to insert the other nodes' IDs manually to study a condition or to let the interface generate them randomly. The number of additional nodes is asked (e.g. '35'). The user is also able to choose how many higher priorities nodes he wants the network to have (e.g. '4'), the remaining will be lower priorities or random.

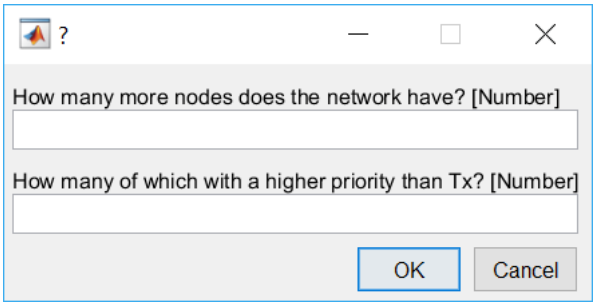

Figure 14: GUI, Number of Nodes

If the inserted numbers are not consistent with what declared in previous steps, error dialogs will appear.

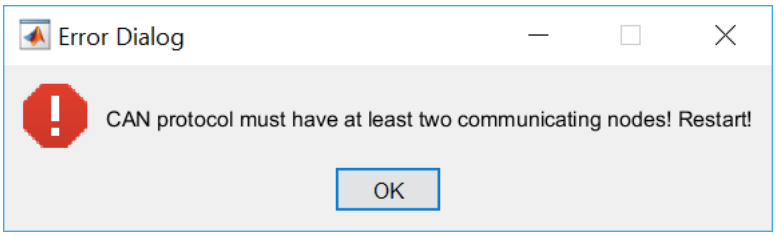

Figure 15: GUI, Warning

5. Timing Analysis

In order to analyze the Typical Response Time and the transmission delay, different functions were implemented: using a counter triggered by the network's clock and strategical flags in the low-level CAN node, we were able to fill in a vector the time instants under analysis.
These vectors are brought into the workspace, at the end of every simulation, and subsequently elaborated via Matlab.

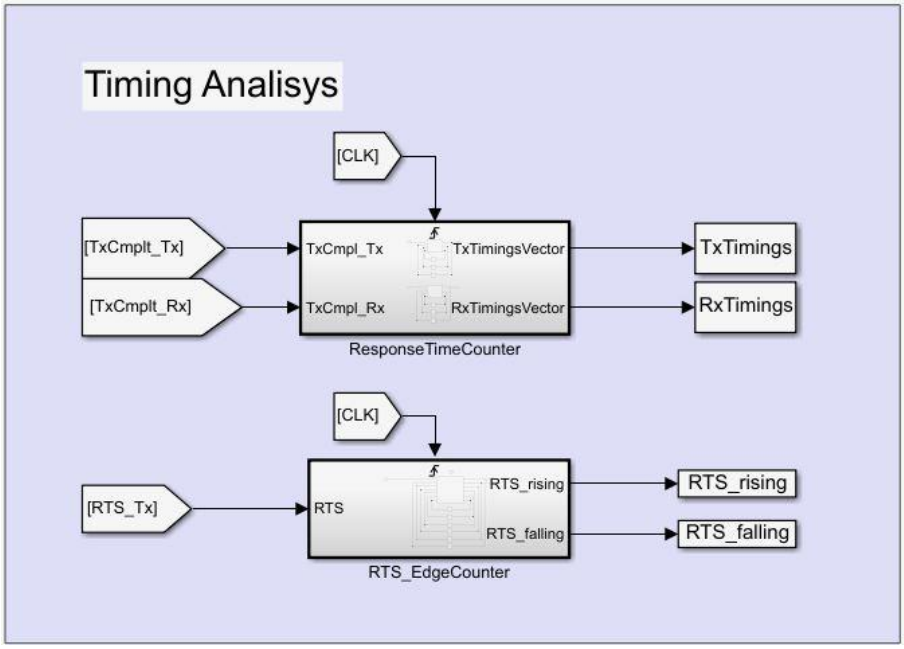

Figure 16: Timing Analysis

The analysis we are going to show are obtained with these settings:

- CLK freq. = 500KHz
- TPDO period = 10 mS
- SimTime = 0.2 s
- ID_Tx = [1,1,1,1,1,0,0,1,1,1,1];
- ID_Rx = [1,1,1,1,0,0,1,1,1,1,1];

The Can Traffic Simulator is fed by 20 pseudo-random IDs, chosen with priorities around ID_Tx and ID_Rx.

5.1 Typical response time distribution

Typical response time distribution is the average response waiting time of a node that sends a request frame to another node, conditioned by the traffic on the network. The nodes directly in this analysis are the transmitter and the receiver, we used their *TxCmplt* flag to evaluate the response delay.

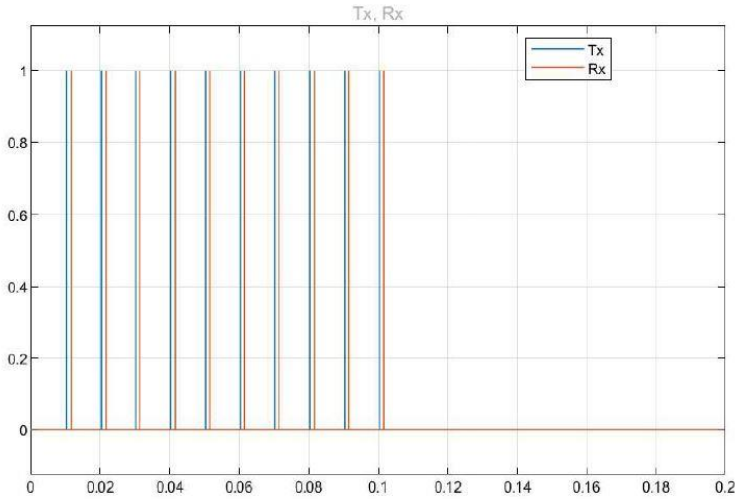

Figure 17: Typical Response Time

In this picture we can see the activation of the *TxCmplt* flags of the two nodes, it was simulated with a Bus Load of 80%. Elaborating these timing result with Matlab, repeating the simulation with different Bus Loads, we obtained the following result:

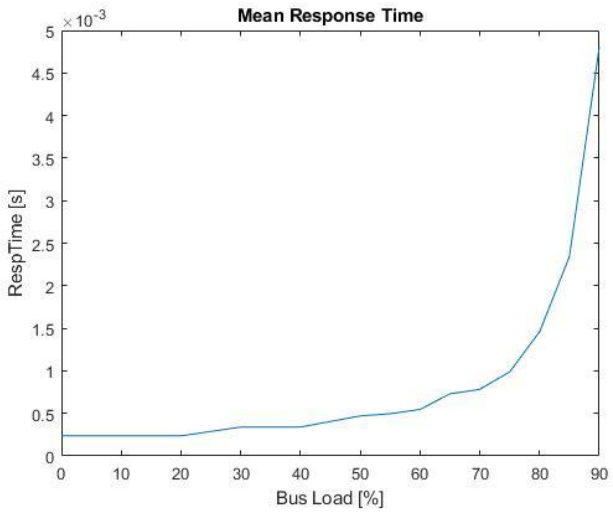

Figure 18: Mean Respose Time

This plot shows the variation of the average response time depending on the Bus Load of the network. Above the 90% Bus Load the receiver is no more able to access the bus.

5.2 Delay (Logical & Physical Transmission)

Delay is the difference in time between the logical sending of the message at the application level and the completion of its sending on the bus, conditioned by the traffic of the network.
As in the previous analysis, we chose flags of the Can Node to identify this specific event. The counter is enabled by the rising edge of the ReadyToSend flag and is disabled at its falling edge. This operation is repeated in time and then averaged, for different Bus Loads.

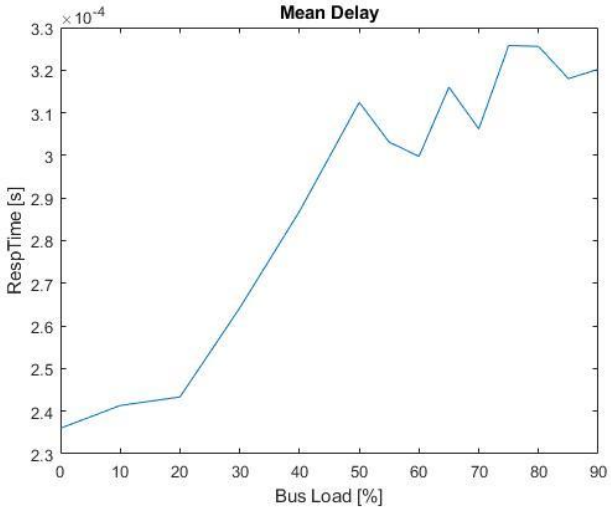

Figure 19: Mean Delay

What we expected was a monotonous trend, which is not consistent with the result obtained. It may be due to pseudo-random choice for the ID and payloads, reflecting in frame stuffed differently.
Nevertheless, the trend of the delay above 50% Bus Load is contained in 15 Clock cycles (300 uS).

5.3 Jitter

Jitter is the delay deviation from true periodicity of a periodic signal, conditioned by the traffic of the network. The only node involved in this analysis is the transmitting one, a periodic TPDO with period 10 mS. The flag used to trigger this event is again the *TxCmplt* one, we considered just the first 20 frames.
The inter-times have been saved in the following matrix (times are in second):

Bus Load	t1	t2	t3	t4	t5	t6	t7	t8	t9
0 %	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10 %	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
20%	0.0099	0.0099	0.01	0.01	0.01	0.01	0.01	0.01	0.01
30%	0.0101	0.0099	0.01	0.01	0.01	0.0101	0.0099	0.01	0.01
40%	0.0099	0.0099	0.01	0.01	0.01	0.01	0.01	0.0102	0.0099
50%	0.0101	0.0099	0.01	0.0102	0.0098	0.0101	0.0102	0.0098	0.0101
55%	0.01	0.01	0.01	0. 0.01	0.0101	0.0101	0.01	0.0098	0.01
60%	0.0099	0.0099	0.01	0.01	0.0102	0.0099	0.0099	0.009	0.01
65%	0.0099	0.0101	0.0099	0.0102	0.0099	0.0099	0.0101	0.0099	0.0102
70%	0.0101	0.0099	0.0100	0.0101	0.0099	0.0101	0.0101	0.0098	0.0101
75%	0.0101	0.0101	0.0101	0.0098	0.0100	0.0101	0.0101	0.0101	0.0101
80%	0.0099	0.0101	0.0099	0.0101	0.0099	0.0101	0.0099	0.0101	0.0099
85%	0.0099	0.0101	0.0099	0.0101	0.0099	0.0101	0.0099	0.0101	0.0099
90%	0.0101	0.0099	0.0101	0.0101	0.0098	0.0101	0.0101	0.0098	0.0101

Figure 20: Jitter matrix

What we see is constant Jitter just until the 10% Bus Load, increasing this value we notice a slight chance in its trend. Jitter remains always in the range of a fraction of half millisecond, it seems not to be significantly dependent on the bus load.
This could probably be a limit of the developed model. Nevertheless, we are confident that, in case of further studies, this problem could find a straight forward solution.