

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

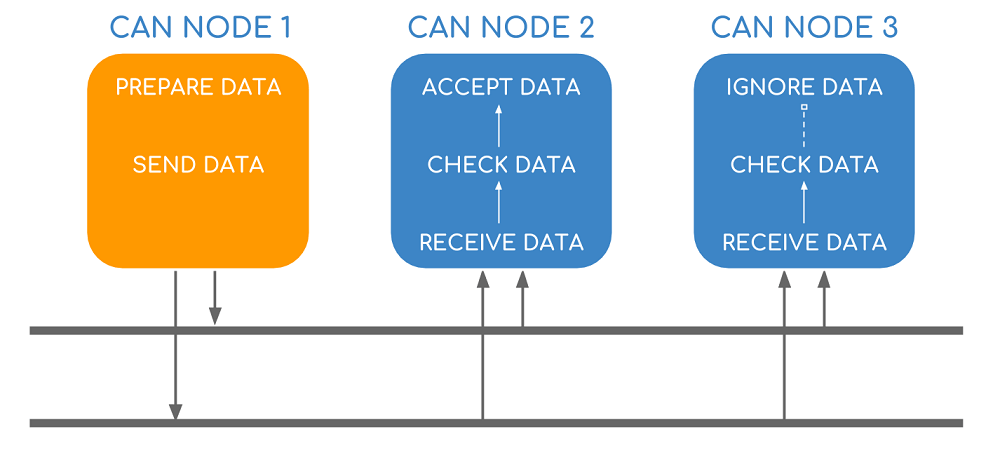
Prof. Alessandro Soldati

Academic Year 2018/2019

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.



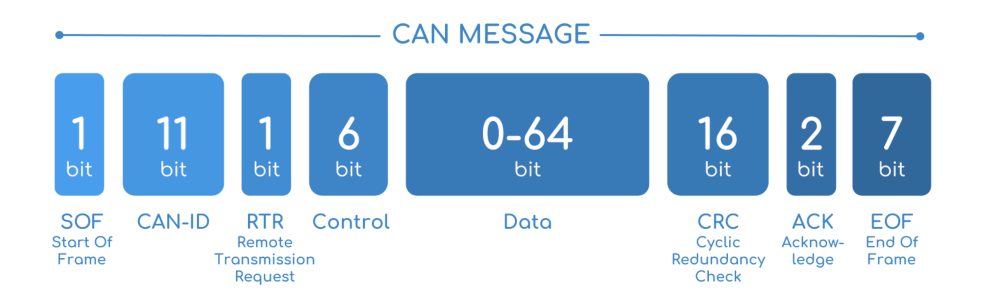
Every node receives all transmitted messages, decides relevance and chooses whether it needs the data on the bus or not. CAN messages are prioritized according to each node’s ID (lower ID means higher priority), making it a very efficient system. Furthermore, it is a flexible protocol since it is possible 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.

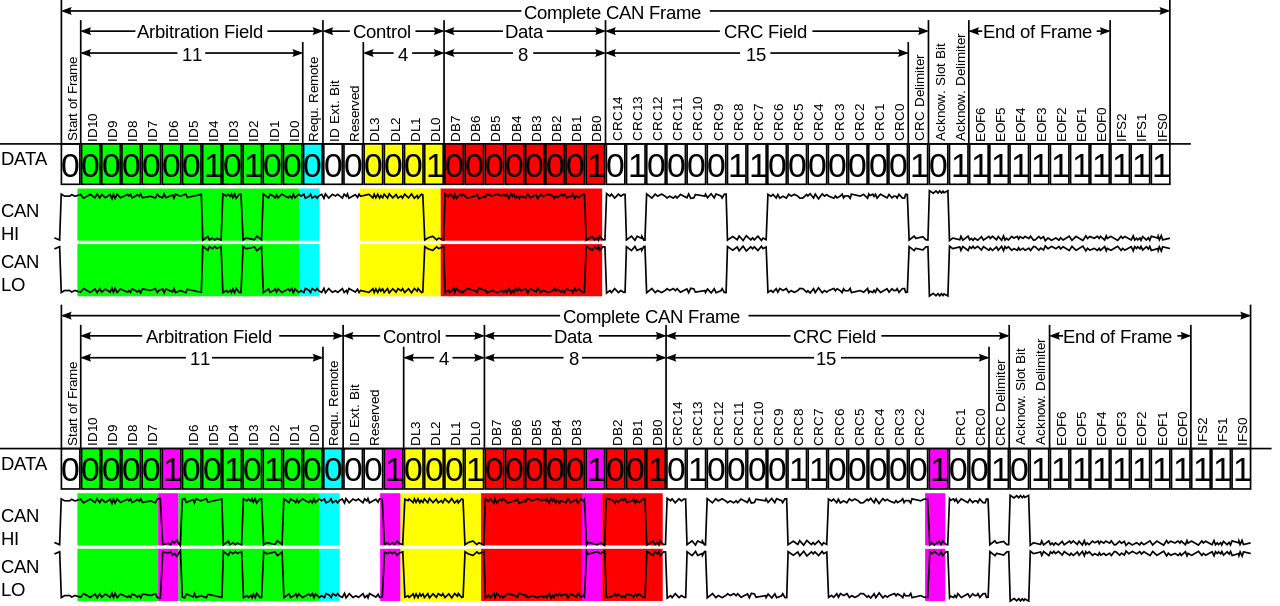
The whole message sent by a node could be subdivided into 8 different blocks:

* 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.



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).



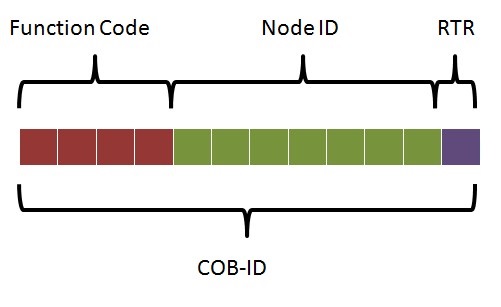
CAN bus’s speed is 1 Mb/s for a 40 m long bus. The speed decreases with longer buses to 40 Kb/s in a 1 Km long bus.

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.

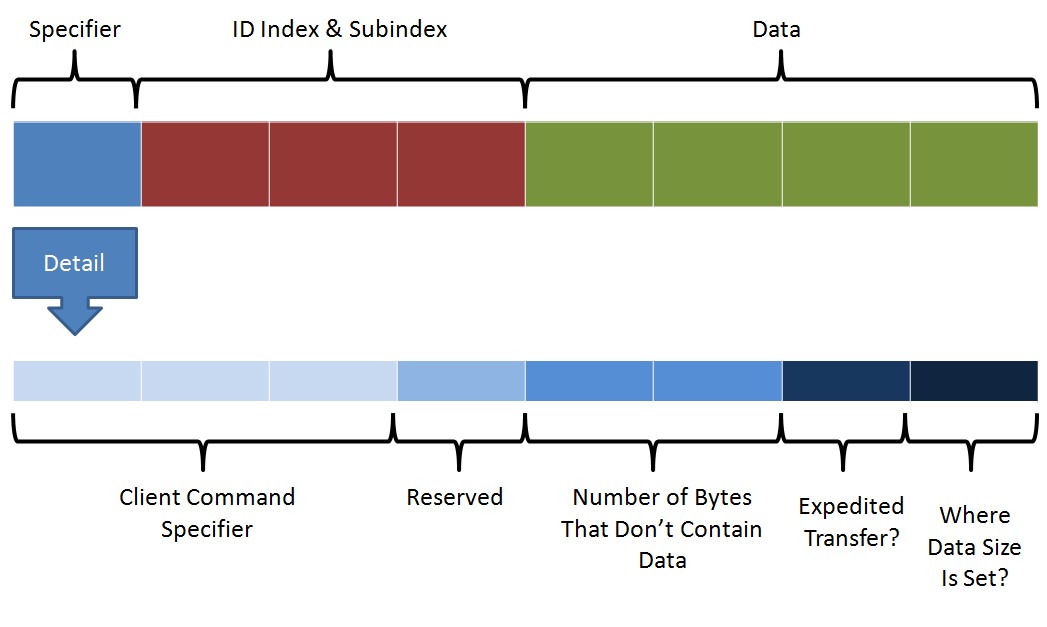
In CANopen the 11-bit CAN ID is split into two parts: a 4-bit function code and a 7-bit CANopen node ID (this limitation restricts the amount of devices on the network to 127). These two blocks followed by a 1-bit RTR form the COB-ID:



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 particular 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 subindex and 4 B for the actual data transfer.



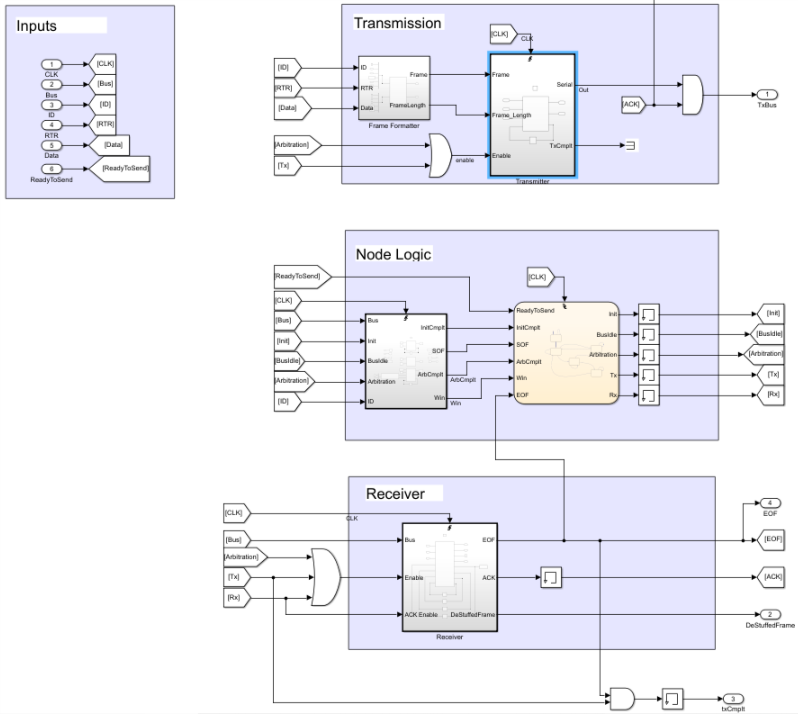
SDO is the mechanism for which a node will send a request to the network and the node of interest will respond with the data requested. When the SDO client wants to request some information it sends a request using a COB-ID of 600h + the slave node ID. The server will respond using a COB-ID of 580h + 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.

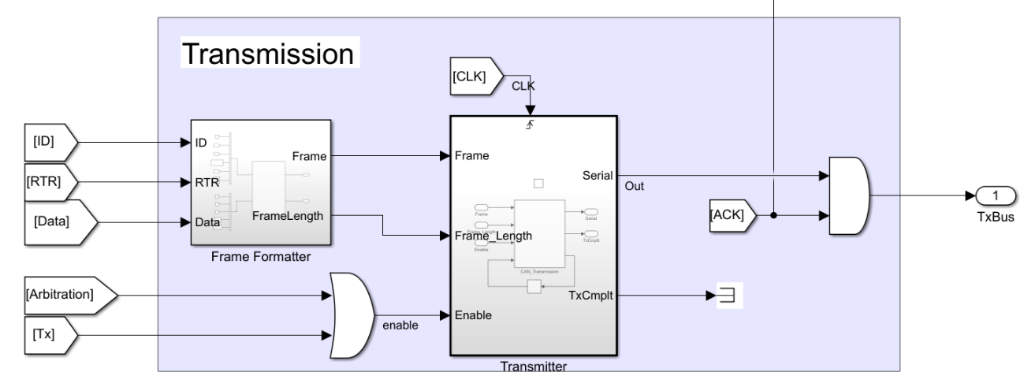
# HOW WE IMPLEMENTED A CAN NODE

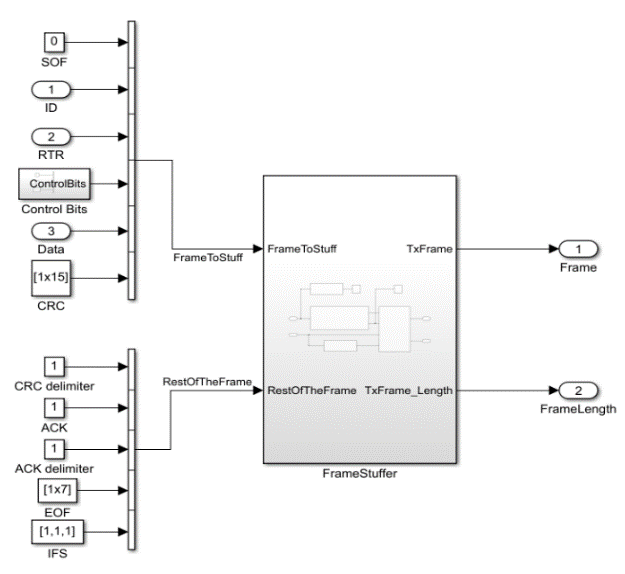
We wanted to obtain an implementation of the CAN network that was as modular as possible. That is why we decided to implement single CAN nodes that have been reused in our model in order to simulate different nodes of the same network.

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

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.

## Transmission

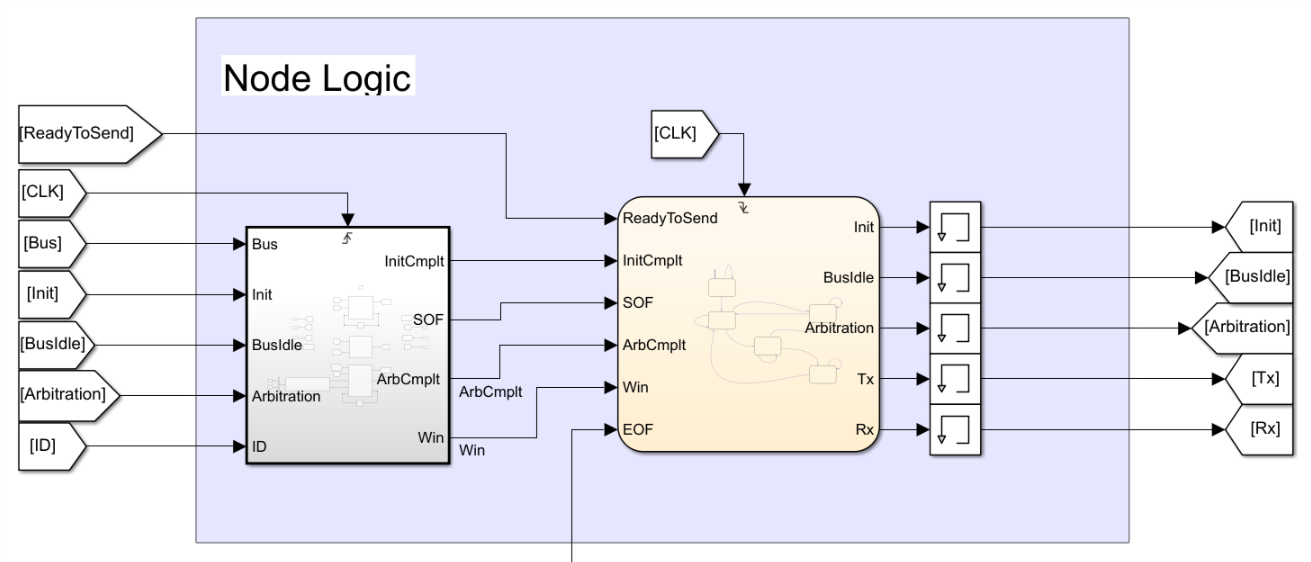


In this part of the logic we have the FrameFormatter. It is used to generate the frame that our Can node will be able to transmit. Inside this block we find a FrameStuffer that receives as input two vectors: the former contains the part of the frame that must be stuffed, the latter just the fixed part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

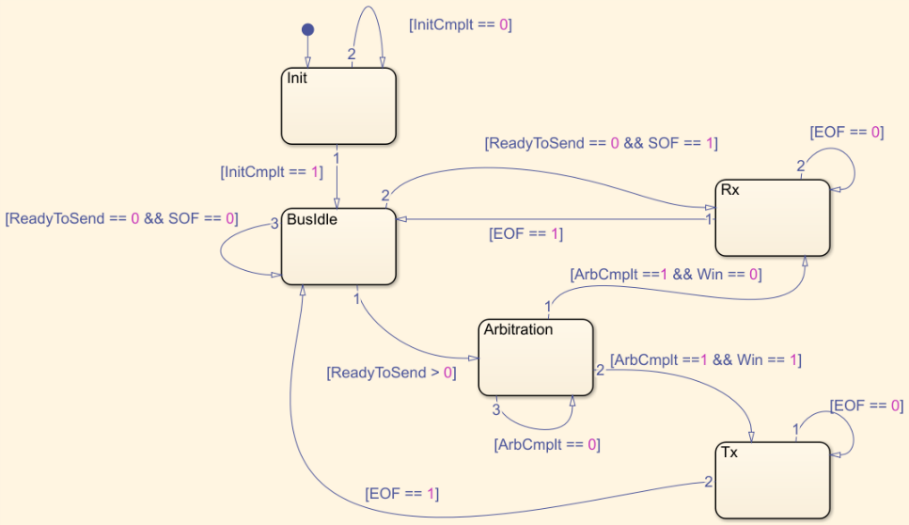
The output of the entire FrameFormatter block enters in the Transmitter one, where we have the physical transmission implementation of the node. This block is enabled only when we are arbitrating or transmitting (“or” of arbitration and transmission states of the state machine). Its output is not directly 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).

FrameFormatter

## Node Logic



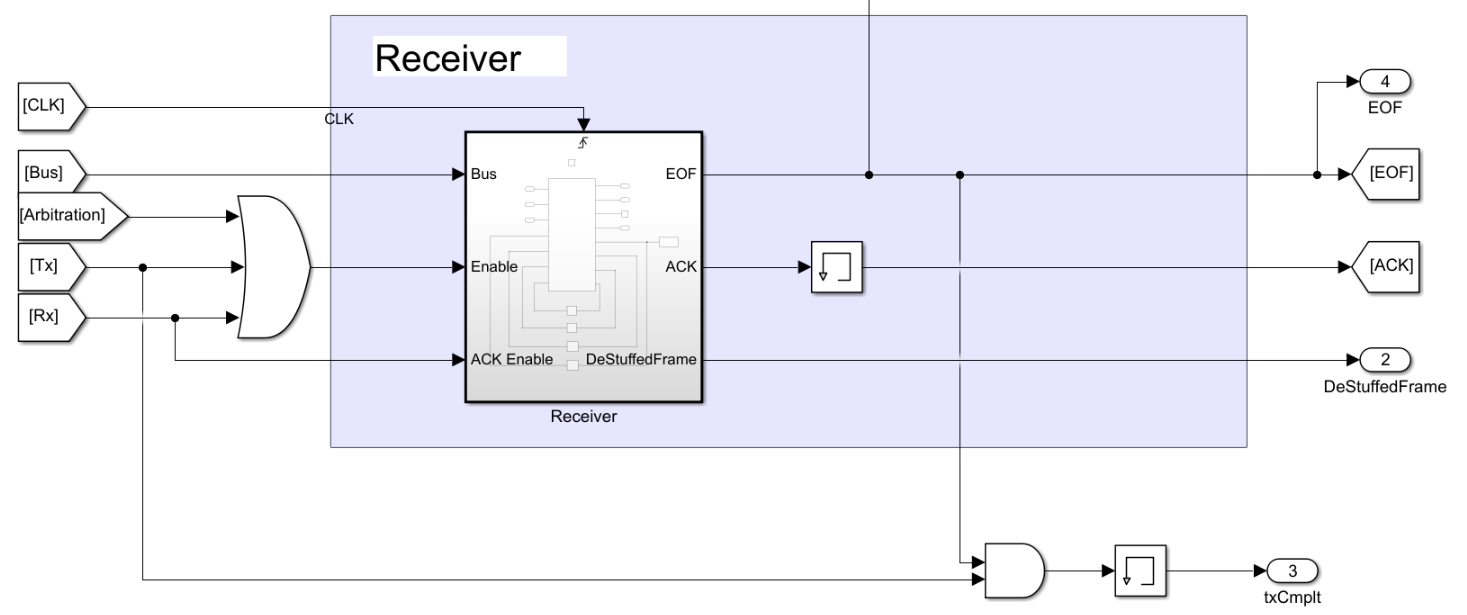
The logic of the node is controlled by a finite state machine that uses parameters (i.e. ReadyToSend, SOF, ArbCmplt, Win, EOF) as flags raised to understand when it is time to move from a particular state to another. Combinatory logic of the state machine is implemented in the block on the left side.



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 and 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 loose synchronization with other nodes). From arbitration phase instead, we can move in Rx if we loose 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.

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

## Receiver

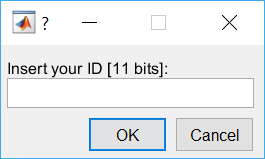


On the Receiver side we have just a block that represents the physical receiver implementation. It must keep trace of everything is flowing through the bus, that is why we enabled it during Arbitration, Tx and Rx phases. The Receiver has three main jobs: DeStuffedFrame generation, ACK sending at a very 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.

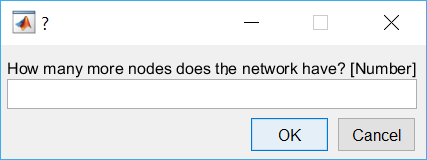
This block was the most difficult to implement because a receiving node does not know the length of a stuffed frame in advance, so it cannot simply have a buffer that stores a vector with predefined length and then destuff it. Considering that, we decided to start receiving packets destuffing them until 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.

In CAN protocol, the ID of each node is buried inside each ECU. Instead, we decided to give the ID as input, through a user interface, in order to simulate as many cases as possible only using two nodes: one simulates the node we want to study and test, 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.

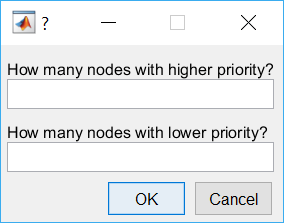
First, the user will enter the ID of the node that as to be tested in terms of transmission delay due to priority and traffic congestion of the network. ID has to be entered with a blank space in between each bit (e.g. ‘0 0 0 1 0 0 1 0 1 1 1’).



The number of additional nodes is asked (e.g. ‘35’). Since CANopen is implemented in our analysis, the number of total nodes is restricted to 127 due to the smaller number of available bits for identification.



More dialog boxes will guide the user through, who is able to insert the other nodes’ IDs manually to study a particular condition or to let the interface generate them randomly. The user is also able to choose how many higher and lower priorities nodes he wants the network to have (e.g. ‘4’ and ‘20’). If the numbers are not consistent with what declared in the beginning, error dialogs will appear.



In the end, the user will be able to choose the bus load through a dialog box as well.

TIMING ANALYSIS

We modelled nodes and traffic on a CAN communication network to analyze typical response time distribution, delay and jitter.

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 of the network.

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.

Jitter is the delay deviation from true periodicity of a periodic signal, conditioned by the traffic of the network.