# 

**Motorvehicle University of Emilia-Romagna**

Master Degree in Advanced Automotive Electronic Engineering

Academic year 2018-2019

**Laboratory experience: CAN network modeling**

Supervisors: Prof. Carlo Concari &

Prof. Alessandro Soldati

Davide Di Maria

Eugenio Serra

Filippo Marcucci

**Introduction**

CANopen is a higher layer protocol based on the top of can bus, which serves as basis for communication (layers 1 and 2). CANopen, for its part, specifies layers 6 and 7 (session and application) of the ISO/OSI model.

CANopen communication protocol includes the following types of messages:

* SDO (service data object)
* PDO (process data object)
* SYNC
* NMT
* EMCY

Service Data Objects (SDOs) are used for direct access to CANopen devices: in particular, to read and write object dictionary entries.

The communication emulates a closed frames exchange between two nodes, that are in a peer to peer relationship, referred as SDO client and SDO server: the latter is the one which provides or accepts the data via its object dictionary and the former requests (reads) or transfers (writes) the data.

The client always initiates the communication and the transfer continues as a request-response sequence of messages which varies according to a specific protocol identified with the first data byte of the service data object.

An SDO message is always 8 bytes long: even if some bits of the data field are not used, they are set equal to 0.

Since only rarely all the bytes contain useful information, the SDO transfer can actually be shortened adopting the so called “Expedited SDO transfer” in which it is possible to transmit up to 4 bytes of data in the initialization phase (the first phase of the communication).

Considering the CANopen protocol, we can distinguish four different SDO services that are:

* Initiate SDO upload
* Upload SDO segment
* Initiate SDO download
* Download SDO segment.

In order to simulate the real messages transmitted by the client and by the server during this phase we considered a communication structured as follow:

* At the beginning the SDO client start the communication with an Initiate SDO download service message as follows:



* After this the server acknowledges with a messages that has a predefined command byte fixed at the value 0x60 and is structured in this way:

For what concerns SDO upload service, used to read an object dictionary entry of a node, we have more or less the same structure with just few differences:

* The communication always begins from the client that sends a message with the same structure but with a command byte fixed at value 0x40:



* After the reception of the client request, the SDO server responds with a message structured in this way:

For our project, we decided not to consider the possibility to have Expedited SDO transfer and the presence of the toggle bit because this would have led to have messages of variable length (aspect difficult to manage during our simulation).

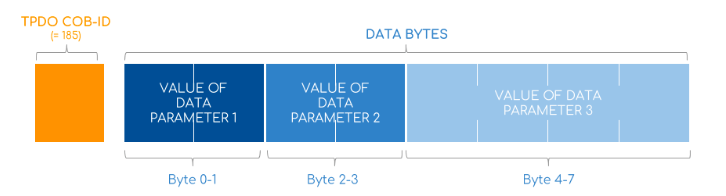
The Process Data Object (PDO) message is used to process real time data among various nodes and can transfer up to 8 bytes of data with a single frame from or to the device. One PDO can contain multiple object dictionary entries.

There are two kinds of PDOs: transmit and receive PDOs (TPDO and RPDO).

On one hand, RPDO can send data to a device and, on the other, TPDO can read data from it.

PDOs can be sent synchronously or asynchronously: synchronous PDOs are sent after the SYNC message (you can request to a device to transmit TPDO that contains data you need by sending an empty TPDO with the RTR flag) whereas asynchronous after internal or external triggering events.

In our project in order to simulate the transmission of the PDO by a node we considered a frame structure in which the length of the data field can be modified during the simulation to consider the possibility of sending one or more object dictionary entries:



In our project, we also considered the transmission of messages necessary for the configuration of the devices and to control the communication state of each node (NMT messages).

According to the CANopen specification, the network manager node can control the communication state of the others and decide to change their states (individually or all collectivly) with the transmission a single message.

NMT messages are transmitted with the highest priority and they have a structure like this 

where the first byte is the state in which the target node must go and the second byte represents the address of the node. Moreover, by setting an address equal to zero, all the devices on the bus should go to the indicated state. In particular, we can differentiate among the following five different states (each of them has a specific NMT code):



In our project, we decided to consider the transmission of the bootup message, that each node must send to switch automatically from the initialization mode to the pre-operational mode (state in which the exchange of PDO is not possible, but only the communication via SDO messages, emergency, synchronization and NMT control messages).

So, we planned to simulate the so called initial heartbeat protocol: a protocol used to monitor the nodes in the network and verify that they are alive. So, we programmed each node to initially send a message with COB-ID 0x700 + node ID followed by a data bytes that indicates its status.

In general, the bootup messages can have the following structure, containing an NMT state code according to what is represented in the following picture:



In our simulation, we considered that at the beginning each node sends one of this message considering boot up as represented state.

The last possible message that we modeled is the Sync-Producer which provides the synchronization-signal for the Sync-Consumer nodes.

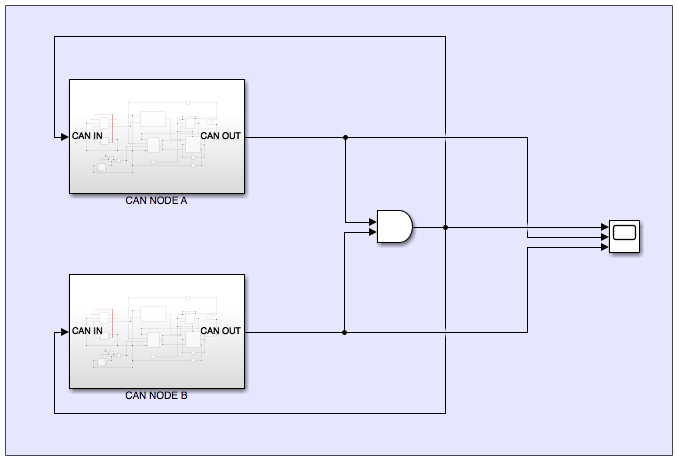
When a Sync-Consumer receives the signal, it starts performing some kind of synchronous tasks.

A typical case where it is used is together with synchronous PDO messages to guarantee that sensors and that actuators devices work periodically and in a coordinated way.

**PROJECT DESCRIPTION**

We start our project from the analysis of the requirements we were given and then we continue with the definition of our architecture. We decide to adopt Simulink as our IDE (as suggested) because we believe that, at the end, it will guarantee us very high-quality results.

First of all, we start the design of the network from the creation of the structure belonging to a parametric node, highlighting its main inputs and outputs (CAN IN and CAN OUT). Then we choose to connect multiple can nodes (through their CAN OUT ports) to the inputs of an AND logic gate. The resulting output of the AND can be considered as the value present on a CAN bus (CAN bus in fact is also said to be a “wire-AND” protocol for its characteristic dominance of 0 over 1). The value present on the CAN bus is returned as input (through CAN IN PORTS) to every node attached to the network.

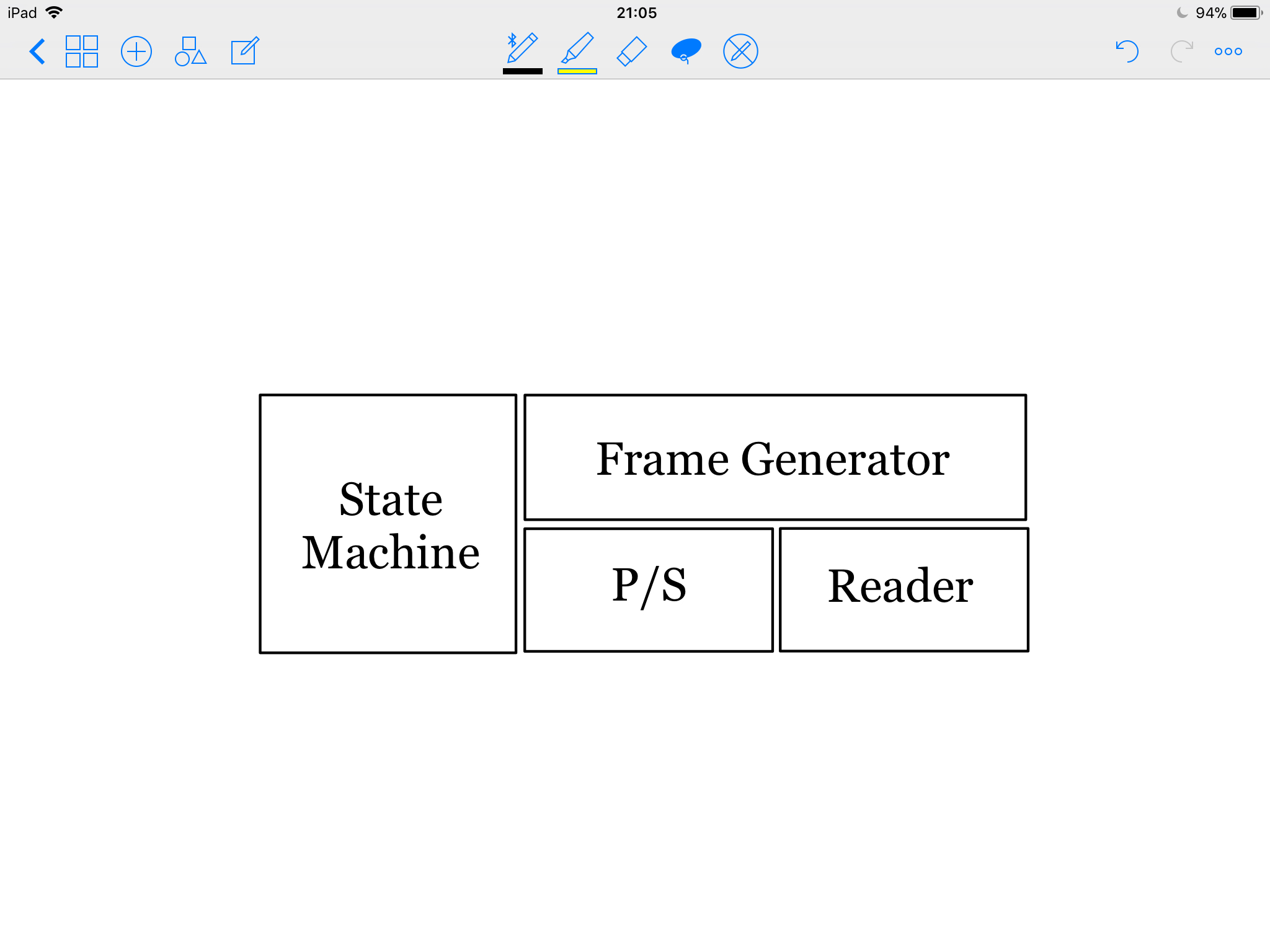


Simulink model of a CAN network with two nodes connected through an AND gate.

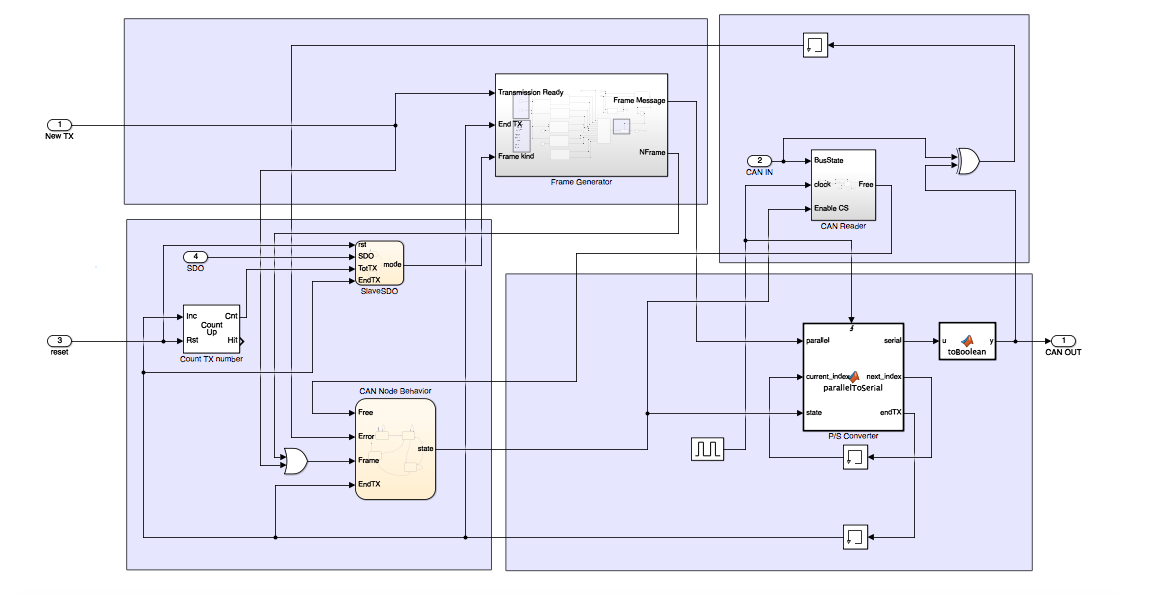
**INTERNAL STRUCTURE TO THE CAN NODE**

Secondly, we continue defining the internal structure of one CAN node: in particular, our idea is to design the framework of a parametric CAN node that can be later specialized to emulate the behavior of a certain CANopen node (sending proper frames in a predefined order).

Therefore, all the nodes present in our model share the same main structure. The only two exceptions are a custom ID field (used during the bitwise arbitration phase) and state machine (that describes which kind of frame each node has to send).



Functional model of a CAN Node.

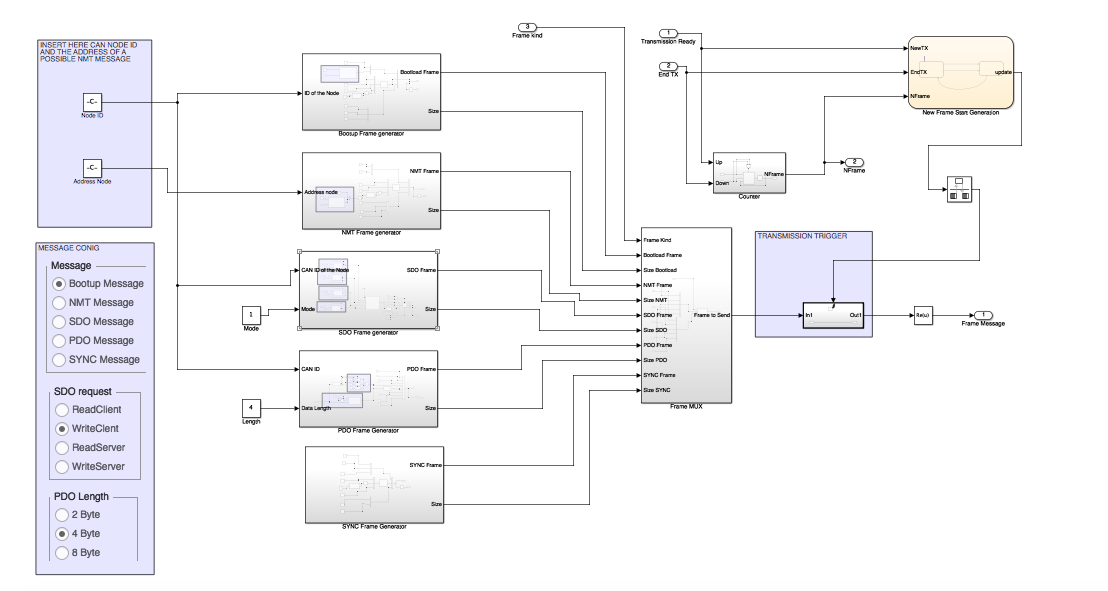


CAN Node internal model.

The Frame Generator generates a binary vector structured according to CAN standard that a node will later start to transmit. It expects to receive two data as input: the ID field of the node (set by the user) and the kind of data field, among those described in CANopen standard, we desire to have at its output; this latter value comes from the State Machine block.

Internally, the Frame Generator structure resembles to a sort of MUX: in fact there a five sub-blocks which generate a SDO, PDO, NMT, BOOTUP and SYNC message, and according to the value present as input propagates the correct one at the output.

Each of the five sub-blocks (SDO, PDO, NMT, BOOTUP and SYNC) has more or less the same structure: it generates a CAN frame containing a SOF, ID, RTR, Control, Data, CRC, ACK and EOF fields in a well-structured way aiming to emulate a typical CANopen bus.

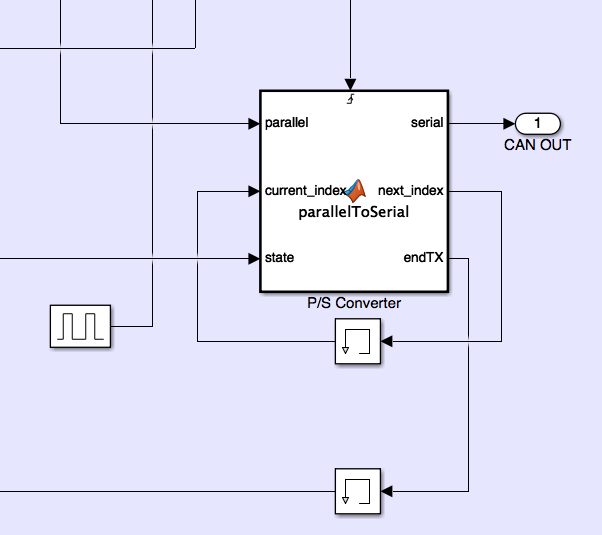


Frame Generator Block internal structure.

Fields that are not fixed by the standard (as the actual information contained in the frame) are randomly generated and CRC calculation with a General CRC Generator block (already-made by Simulink) uses the real CAN polynomial generator (). Moreover, we implemented a Stuffing algorithm for our messages, a variable size PDO data field (of 2, 4 or 8 byte selectable by the user) and four possible SDO structures (similar to the four possible messages of Read Client, Write Client, Read Server and Write Server).

The Parallel to Serial Converter block, when enabled by the State Machine, tries to push one on the bus (through its CAN OUT port) one bit every clock cycle; when disabled it simply leaves the CAN OUT value to 1 (recessive). Moreover, it is responsible of a very delicate task: notifying to the node when the frame has been completed transmitted.

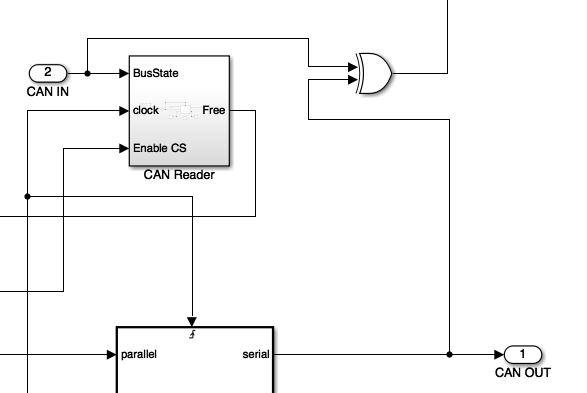
It is realized through a simple MATLAB Function block.



Parallel to Serial Converter Block.

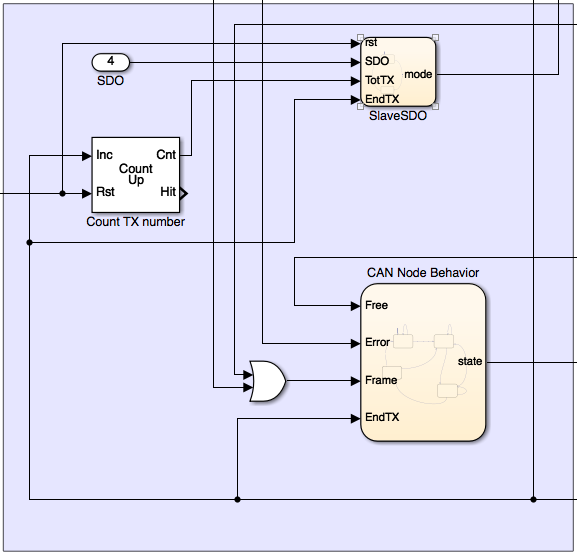
The Bus Reader Block, when activated by the State Machine, starts sensing the bus (by checking the value present on its CAN IN port) checking whether another node is already transmitting or not. In particular, it starts counting the number of consecutive 1s that are on the bus and, if they are greater than a predefined threshold, it notifies to the State Machine the idleness of the bus giving the chance to start a transmission.

In addition to this, the block also contains a subsystem that is responsible of generating an error message in case of lost bitwise during the arbitration phase to notify to the State Machine to stop the message transmission.



Bus Reader Block.

Finally, the State Machine Block is the brain of a CAN node: it takes car of switching on and of the previous three blocks according to the CAN rules. Specifically, the State Machine block is actually divided into two separate state machines: one emulates the behavior of a CAN node at layer 2 (deciding when the node must transmit, sense the bus, doing a bitwise arbitration and stop transmission) and the other that takes care of the layer 7 (application layer) deciding the kind of frame that the node should send each time (ex. first BOOTUP and then PDO).

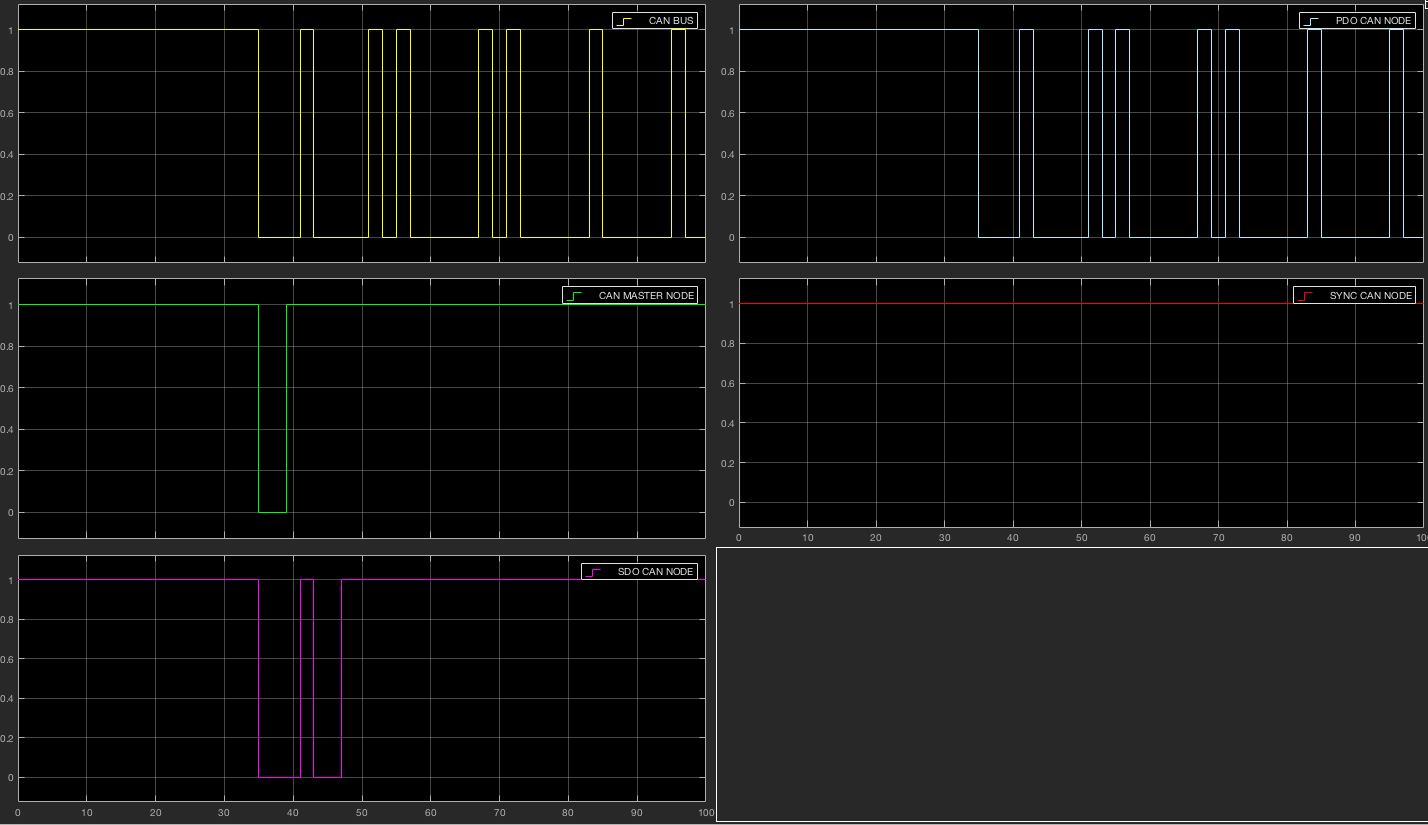


State Machine Block.

**TESTS**

We carried out a complete set of tests, starting from the single functional units (unit testing) starting from the inner sub-blocks. Once verified their correct functioning we moved on to test their integration and verify the correct behavior of the four main blocks and eventually of the entire node.

As last step, we connected multiple nodes to the same bus via an AND gate and we verified the correct functioning of the whole system (bitwise arbitration correctly performed according to the ID assigned to the different nodes).



Bitwise arbitration phase on CAN bus with four nodes.

**RESULTS**

By simulating three different situations we have obtained the following results:

|  |  |
| --- | --- |
| Network ready (bootup) | 396 |
| Simulation time | 5000 |  |  |
| SYNC period | 256 |  |  |
| N frame | 5 |  |  |
| N nodes | 4 |  |  |
|  |  |  |  |
| Latency (**μ**s) | MEAN | MAX | MIN |
| Master | 159,6 | 277 | 65 |
| SDO | 328,2 | 727 | 123 |
| PDO | 114,6667 | 296 | 74 |
| SYNC | 56 | 56 | 56 |
|  |
|  | Jittermax | Tx completed | First message | |
| Master | 212 | 905 | 540 | |
| SDO | 604 | 1748 | 1144 | |
| PDO | 222 | 4976 | 1000 | |
| SYNC | 0 | 4881 | 761 | |

Case 1: 5000

|  |  |
| --- | --- |
| Network ready (bootup) | 396 |
| Simulation time | 10000 |  |  |
| SYNC period | 256 |  |  |
| N frame | 10 |  |  |
| N nodes | 4 |  |  |
|  |  |  |  |
| Latency (**μ**s) | MEAN | MAX | MIN |
| Master | 149 | 277 | 65 |
| SDO | 349,2 | 1619 | 123 |
| PDO | 142,4118 | 1015 | 74 |
| SYNC | 56 | 56 | 56 |
|  |
|  |
|  | Jittermax | Tx completed | First message | |
| Master | 212 | 1702 | 540 | |
| SDO | 1496 | 3704 | 2036 | |
| PDO | 941 | 9862 | 1797 | |
| SYNC | 0 | 9767 | 761 | |

Case 2: 10000s

|  |  |
| --- | --- |
| Network ready (bootup) | 396 |
| Simulation time | 20000 |  |  |
| SYNC period | 256 |  |  |
| N frame | 20 |  |  |
| N nodes | 4 |  |  |
|  |  |  |  |
| Latency (**μ**s) | MEAN | MAX | MIN |
| Master | 143,7 | 277 | 65 |
| SDO | 359,7 | 3575 | 123 |
| PDO | 159,8788 | 2686 | 74 |
| SYNC | 56 | 56 | 56 |
|  |
|  | Jittermax | Tx completed | First message | |
| Master | 212 | 3296 | 540 | |
| SDO | 3452 | 7616 | 3992 | |
| PDO | 2612 | 19508 | 3468 | |
| SYNC | 0 | 19413 | 761 | |

Case 3: 20000s

Observing the data, we note that in all three cases the network response time is constant because the order of the bootup messages remains unchanged, even increasing the number of frames on the channel.

The noteworthy fact is that the average latency of the master decreases as frames increases, since having more frames to send and having a lower priority only to the sync node but having precedence over the PDO and SDO nodes, the latency contribution of each extra frame will surely be less than the average latency of the master and will therefore contribute to lowering the average latency level.

The minimum latency of the master is in fact equal to 65 microseconds, which means that for each frame added to each simulation and up to the value of at least 65, the average latency of the master will tend to decrease.

No less important is the result of the sync: in fact in this case average latency, maximum latency and minimum latency remain unchanged in each case. This fact is due to the highest priority of the sync node.

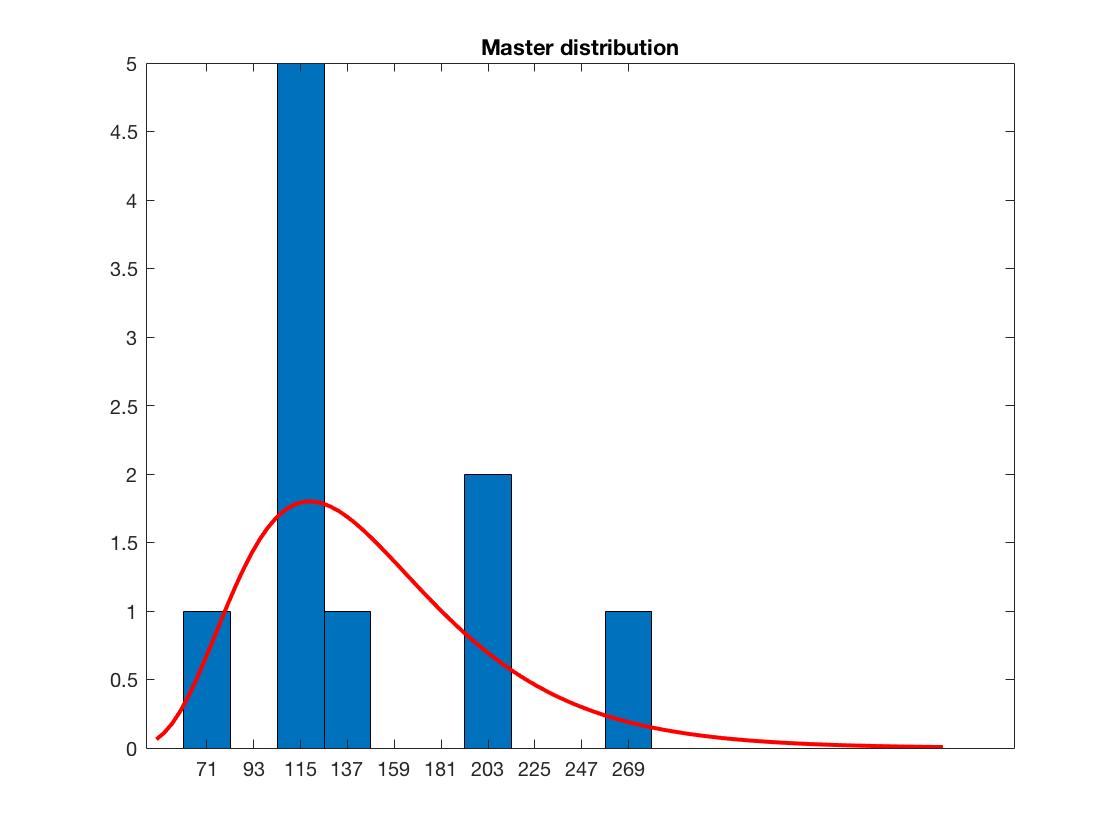
As for the PDO and SDO nodes, since they are the least priority nodes, their average latency will tend to increase because the master messages will certainly be sent before the can with respect to theirs, and this is also shown by the data in the table.

It should be noted that the SDO node has a maximum latency that increases in the third case up to more than four times compared to the first case, which leads to a significant delay in requests made via SDO.

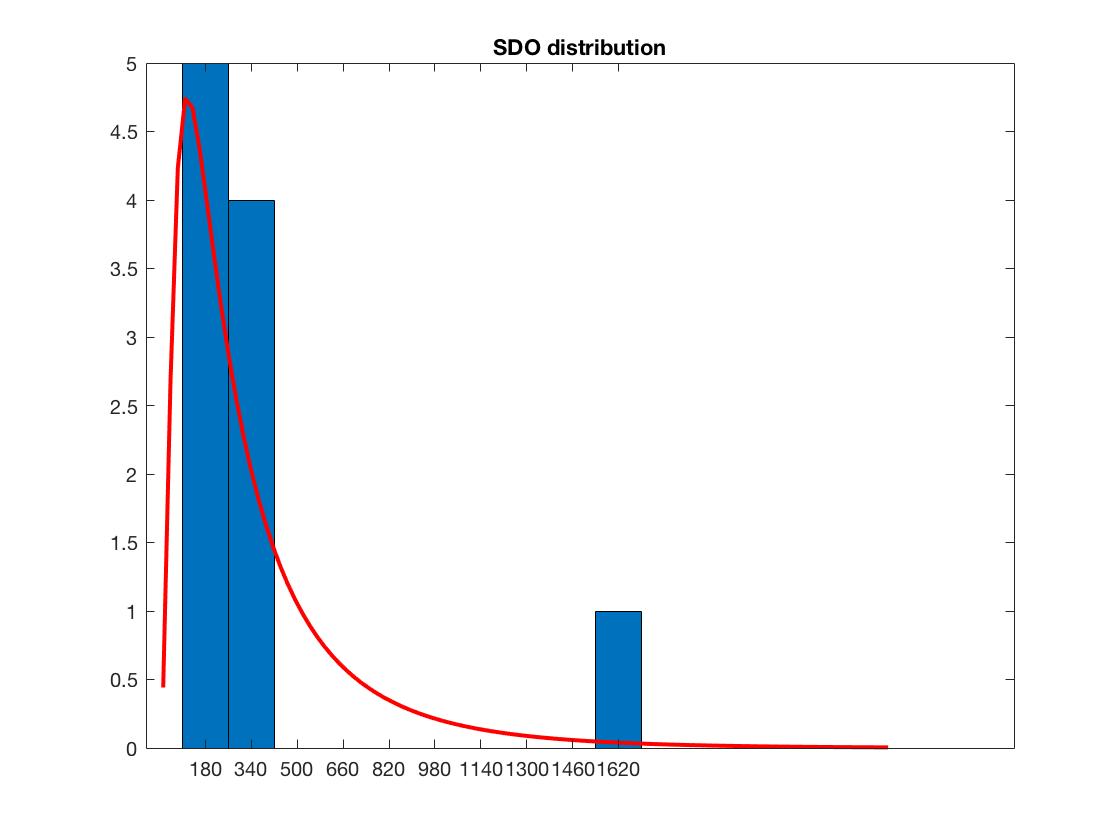
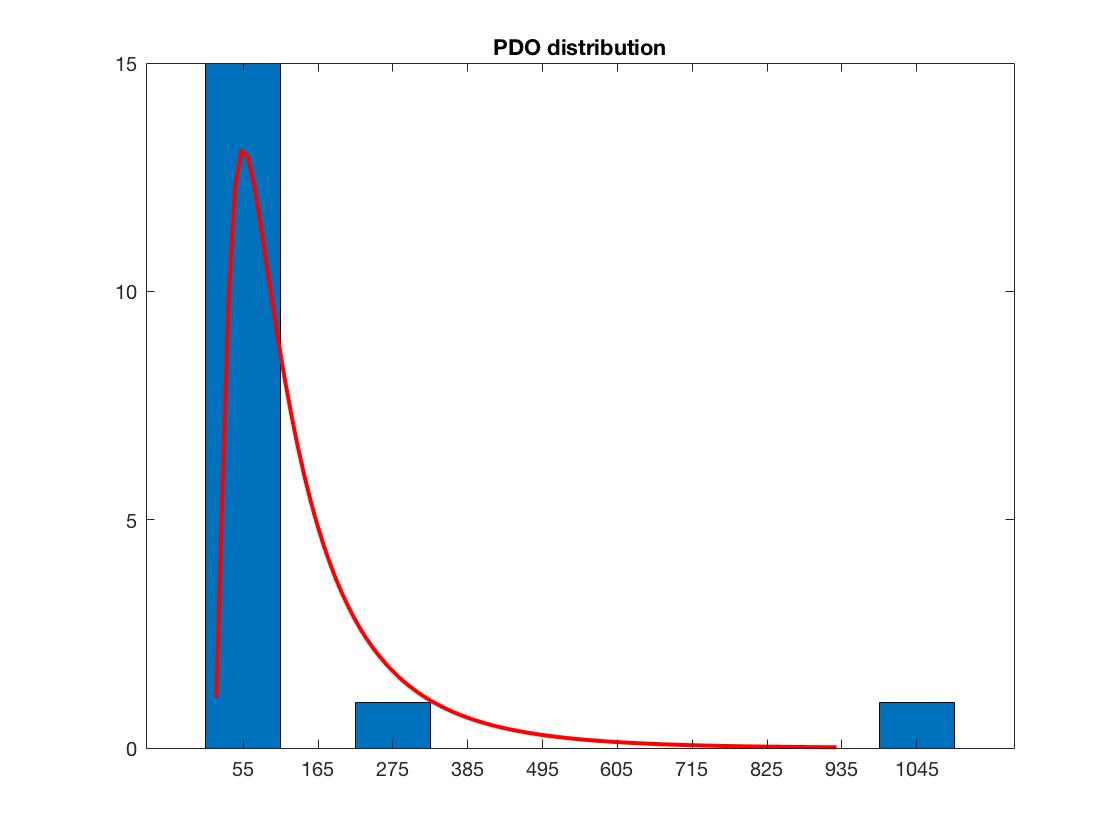
Last parameter is the jitter: as we may notice from the tabs, jitter increases as frames increases; this is due to the fact that MASTER node has more frame to send and has an higher priority than SDO and PDO nodes.

Distribution Function estimate

From the latency vectors it is possible to obtain an estimate of the probability distributions through the "histfit" command of the MATLAB program.



Analyzing the case with 10 frames for each node, as we can see from the graphs, the latency of the master is more likely to be around its average value, i.e. 115 microseconds.



On the other hand, by comparing PDO and SDO nodes, we note that the maximum probability distribution of the PDO node is more displaced towards the left than that of the SDO node, denoting a better latency as already mentioned above.

The distribution of the SYNC node is not reported because the latency value remains constant.