Politecnico di Torino

Master's Degree in Mechatronic Engineering
Collegio di Ingegneria Informatica, del Cinema e Meccatronica

A.A. 2022/2023

Development of an Electric Vehicle Arduino-like Open-source Platform Evduino

Supervisor
Prof. Stefano CARABELLI

Candidate
Giuseppe Jose FERRARA

INDEX

Abstract

During the last decade the usage of Embedded System in automotive applications is increased extraordinarily and even in a chip shortage period the trend is to replace classical mechanical system. With this vision, in the following work of thesis will be discussed the development of EVduino, an open-source Vehicle Management Unit with its environment, for automotive applications.

The basic idea is to provide the users the possibility of developing applications with the flexibility of model-based design through Simulink with the capability to embed software in an Arduino-like fashion. The fundamental starting point is the ISO 26262 standard for functional safety in road vehicles, in which is described the development of a product in each phase.

The final goal is to improve the functionality range of the EVduino itself in order to include a library that can manage CAN protocol, that is a vehicle bus designed to implement the communications between the microcontroller and other devices without a host computer.
The custom board used was developed by Ideas&Motion S.r.l equipped with Aurix Infineon TC277TP processor, specifically designed for safety critical applications.

The CAN Application software C code written starting from Infineon's low-level drivers, keeps the advantages of Arduino, also it is not only limited to single task execution, but thanks to Erika Enterprise RTOS, more tasks can be executed at different frequencies, moreover its integration with HighTec IDE provides necessary features such as Rate Monotonic Scheduling for the task execution.
At this point, there is the need of provide a visual approach to the functions, this leads to the previous mentioned library adaptation for the purpose of creating Simulink blocks for EVduino input and outputs using MATLAB Legacy Code Tool. To sum up, the user is now capable of designing custom Simulink models with CAN communication blocks, tune parameters of the model and log CAN messages, for example with an external tool used as a receiving or transmitting node.

The final step is to automatically generate a C code thanks to the Embedded Coder (that is an extension of Simulink and MATLAB coder), which can be imported in the IDE so that the custom board can perform the execution.

This kind of approach is especially efficient due to the fact that reduces the introduction of bugs of human nature and, most of all, accelerates the software development process, for instance giving the possibility of integrating new functions into previous version of the code.

Introduction

In the last few decades, the automotive industry has undergone deep changes, especially concerning the importance and the role of the electronic and the software.
Both helped improving performances and, most of all, safety, for instance the introduction of ABS, the management of the engine etc.

In this context, Controller Area Network (CAN bus) has been developed starting from 1983 at Robert Bosch GmbH.[1]

The Controller Area Network (CAN) protocol has become the standard for communication between electronic control units (ECUs) in modern automobiles, as well as in many other industrial applications.
The CAN protocol offers many advantages over traditional communication methods, such as speed, reliability, and efficiency. As the complexity and sophistication of modern vehicles and industrial systems continue to increase, the need for robust and efficient communication networks becomes increasingly important.

Technically speaking, CAN bus is a vehicle bus standard to allow the communication between the microcontroller and the peripherals' applications without a host computer.

It is a message oriented protocol, where the information to be exchanged is decomposed into messages, which are assigned an identifier and encapsulated in frames for transmission.

The data in the frames are transmitted serially but in a priority built system, so if more than one device transmit, the highest priority one continues while the other stop.

Frames are received by all devices, including by the transmitting device.

Although initially applied in the automotive sector, as a vehicle bus, it is currently used in many applications in the embedded sector.

Here will be first designed a CAN application library to enable the communication between the chosen board (EVduino), equipped with an Infineon Aurix TC277 microcontroller.

Then will be developed the Simulink blocks, related to this library, in order to enable the user to build and generate C code in a Model Based approach, giving also to the user the capability of logging CAN messages.

Library Development

The Aurix board was designed and developed by Ideas & Motion S.r.l, it is based on the Infineon Aurix TC277 microcontroller, it is equipped with a triple Tricore processor, with 200 MHz, 4Mb of flash memory and a Powerful Generic Timer Module (GTM).[2]

The peripherals available and already developed are ADC, PWM, GPIO ports, what was missing is CAN and here will be developed.

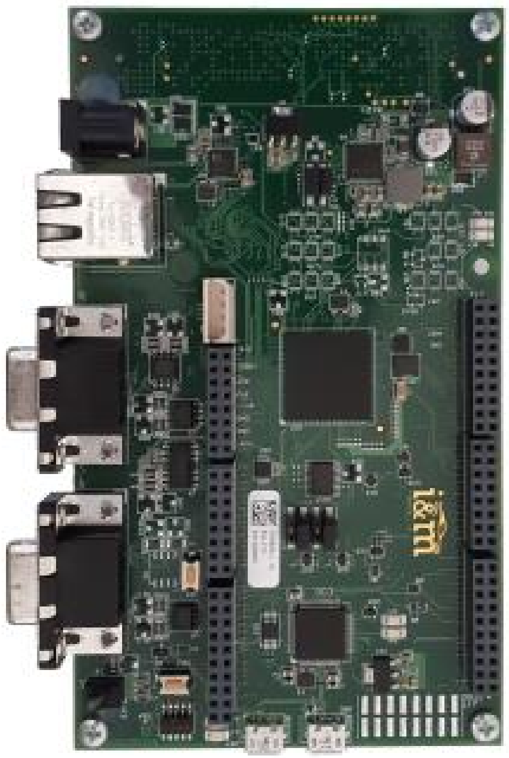


Figure 1 EVduino board

The *'Project Starting Folder'* contains all the tools and configuration files for ***Erika Enterprise RTOS***, as well as the Infineon's drivers and the Arduino-Like library. For this reason, it is the starting point of every application and will be analysed in detail in the next Section.

It has a complex structure, as can be deduced from the figure below, therefore only the most relevant sub-folder content will be analysed.

First of all, all the files will be written in the subfolder "*Library*" of the source code under *AppSw/Tricore* as can be seen in Figure 2 and are based on the *IfxMultican_Can.c* and *IfxMultican_Can.h* located in *Multican* folder under *BaseSw*, which contains the *Infineon Low Level Drivers (ILLD)*, a set of all the functions needed to interface with the peripherals.

WARNING: Do not modify the drivers.

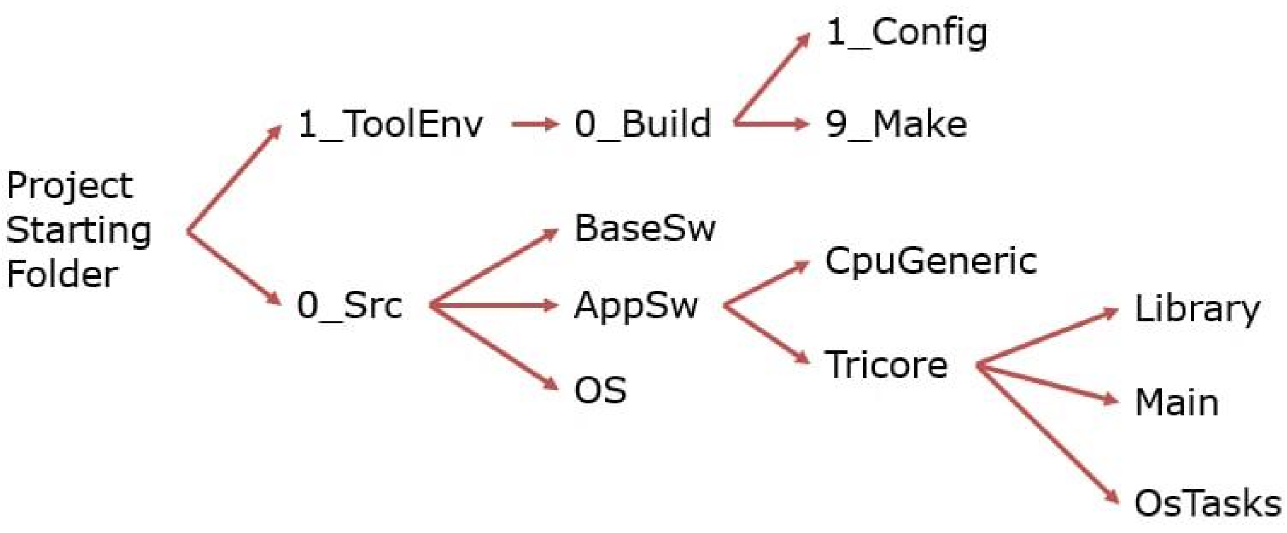


Figure 2 Application and Baseline software scheme

Here two main files have been created or modified:

- *EVduino_if.h*: Header file in which are defined and linked all the prototype functions for an easier use of every peripheral including CAN, below it is shown the example.

  ```
  void CANinit(void);
  void CAN_transmit(void);
  void CAN_receive(void);
  ```

- *CAN_lib.c*: Main file of the library useful for CAN communication, here the previously defined functions have been coded, but first are configured the needed variables and most important: the input and output pins of the core assigned to CAN0_RX and CAN0_TX.

NOTE: Only CAN node 0 is available, so the user must use only port J3 of the board.
Moving forward in this file, the first written function is:

1) `void CANinit(void);`

It is needed to initialize multiple things, not only the port.
A problem encountered and solved here was the fact that to enable the sending and receiving capability of CAN port, two extra pins should be defined in Output Mode Push Pull, they serve for enabling CAN port 0 and disabling its standby.

(Only one for the other nodes, if the developing is needed, information is available on the board datasheet)

Resuming in brief the whole function, first it enables and sets the mode of the pins of the CAN connector, then it configurates the CAN node both for transmitting and receiving.

NOTE: To change the *baudrate* you have to modify the following line of code `canNodeConfig.baudrate = 1000000;`
Default baurate is set to 1 MHz and sample rate set to 80%.
Slower baudrate decreases the speed of data passing through the bus, it can cause conflicts if multiple datas are handled.

Lastly, the function configures and assigns separately CAN message object for transmitted messages and for received messages, especially it set the acceptance mask, the kind of data frame (*IfxMultican_Frame_transmit* or *IfxMultican_Frame_receive*) and the length of the message, here it imposed to 8 bytes since we only have to deal with basic CAN messages.

2) `void CAN_transmit(void);`

This function, first, initializes the message structure to be sent in this form:

- `msg` – The message which should be initialized.
- `id` – The message ID
- `dataLow` – The lower part of the 64-bit data value
- `dataHigh` – The upper part of the 64-bit data value

- `lengthCode` – number of bytes (data length code) which should be transmitted (0…8), that will always be fixed to 8 in our case.

Then, after checking that the bus is not busy, it sends the CAN message.

NOTE: If a user, wants to send a basic CAN message here to be read via monitoring tools, it is sufficient to impose to `id`, `dataLow` and `dataHigh` the wanted values as uint32.

```
3) void CAN_receive(void);
```

This function, first, initialize the message structure to be received as in the previous form, but instead of the variables, it places temporary dummy values, these will be substituted by the real values afterwards.

Then it waits until the Node receives a new message, i.e., the RX pending flag of a message object is set to false, after this it properly reads the CAN message.

If no new data has been received, the code gets stuck in a loop, to signal the error.

Significant Issues

The *CAN_lib.C* file has to be rearranged according to the meet both the requirement of the Infineon Low Level Drivers and the configuration of the board.

In fact, in the first tests, a CAN message could not be handled by the microprocessor since there was a mismatch between PIN configurations.
To overcome this problem there is the need to set in the proper way and in the right modality all the PINs such as one for the *Standby* of the connector and its relative channel and, also, the *Enable*.

Moreover, the CAN high and CAN low PINs must be modified according to the one selected in the datasheet, from where all these information can be derived.

Real Time Operating System

A Real-Time Operating System (RTOS) is a specialized type of operating system designed to manage and run applications with strict timing and reliability requirements. Unlike traditional operating systems, an RTOS guarantees deterministic timing behaviour, meaning that it can respond to external events within a predictable and precise timeframe.

The used RTOS is ERIKA (Embedded Real-Time Kernel Architecture) which is an open-source Real-Time Operating System (RTOS) specifically designed for deeply embedded systems. It is a preemptive, priority-based RTOS that supports multiple concurrent threads and provides deterministic timing behaviour.[3]

ERIKA is based on the OSEK/VDX standard, which defines a common set of interfaces for automotive embedded systems.

To properly set the operating system in *'Project Starting Folder'*, between all the folders and files contained inside, it is important to look under the path *1_ToolEnv/0_build/1_Config* at the file 'CfgErikaOS.oil'.

It is the *.oil* file containing the instructions (synchronous and asynchronous tasks, alarms, events, etc.) for the configuration of the Operating System.
Oil stands for 'OSEK Implementation Language', and it is typical of any OSEK-compliant RTOS.

Only periodic tasks (and their relative alarms) are configured. The following table list all the configured tasks with their frequency of execution and the assigned priority.

Period of execution	Frequency of execution	Priority
1 ms	**1 KHz**	**64**
5 ms	**200 Hz**	**32**
10 ms	**100 Hz**	**16**
20 ms	**50 Hz**	**8**
50 ms	**20 Hz**	**4**
100 ms	**10 Hz**	**2**
Background		**1**

Figure 3 Periodic task and relative priority

A higher value of priority means that the associated task has a higher priority. Therefore, the 1ms task is the one with highest priority. If two or more tasks have the same execution starting time, the one with highest priority will be executed first, and then proceed in order of priority.

If a task with higher priority must be executed while a task with slower priority is running, a rate monotonic scheduling with preemption is adopted for the scheduling of the tasks.

This behaviour can be better appreciated in *Figure 4*.

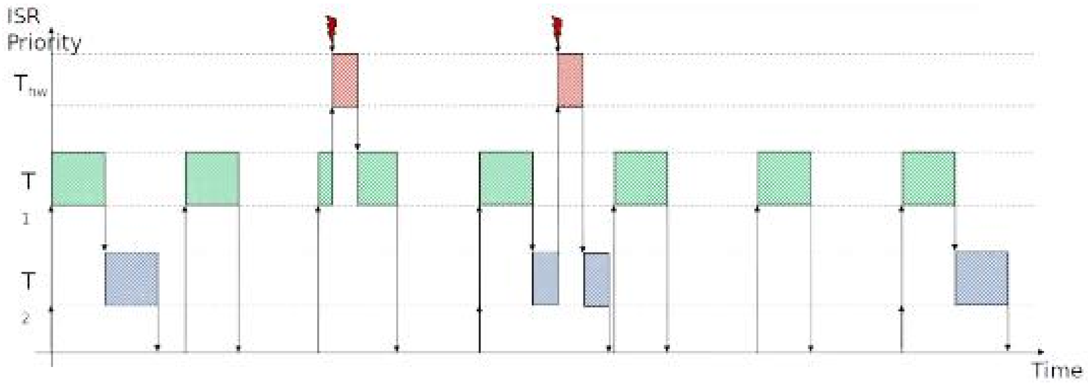
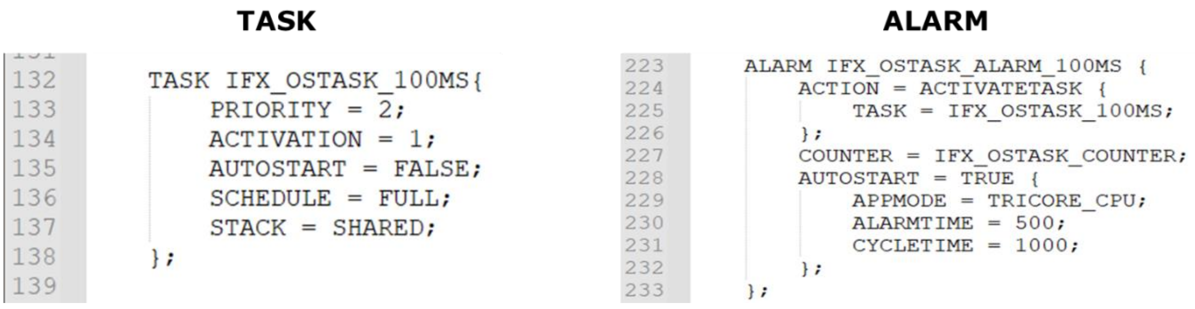


Figure 4 Rate monotonic scheduling with preemption

Each task is activated by its relative alarm, in which are configured the timings at which the call of the task happens.
In the picture below is shown how the 100ms task and its relative alarm are configured inside the provided oil file.



TASK

```
TASK IFX_OSTASK_100MS{
    PRIORITY = 2;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
    SCHEDULE = FULL;
    STACK = SHARED;
};
```

ALARM

```
ALARM IFX_OSTASK_ALARM_100MS {
    ACTION = ACTIVATETASK {
        TASK = IFX_OSTASK_100MS;
    };
    COUNTER = IFX_OSTASK_COUNTER;
    AUTOSTART = TRUE {
        APPMODE = TRICORE_CPU;
        ALARMTIME = 500;
        CYCLETIME = 1000;
    };
};
```

Figure 5 Example of .oil file

Inside the task can be identified the `PRIORITY` and the scheduling strategy (`SCHEDULE`), that is set to FULL which stands for full preemptive. The task is executed at each activation of the alarm.
The alarm is activated for the first time after 50 ms (`ALARMTIME = 500`) and then each 100 ms (`CYCLETIME = 1000`).

IDE

HighTec is the Integrated Development Environment suggested by Infineon and it is particularly suitable for multitask applications thanks to the possibility to integrate Erika Enterprise RTOS.

Refer to the Chapter 2 of *"EVduino MultiTask IDE – User Manual*" guide for the details on the procedure developed and used.

Basically, the tasks can be placed in a file known as *OSTasks.C*, according to their cycle time, after having defined them in other two files.

In the next chapter, the performed test follows this procedure.

Library Functions Test

The working directory here will be ***OSTask*** under the path *0_Src/AppSW/Tricore* of the Project Starting Folder.

To test the developed functions on *CAN_lib.C,* the idea was to set the environment using three tasks defined in `Application.c` and `Application.h` in a structure as follows:

- Initialization task, which will only contain the `CANinit` function and will be placed after the OS start call.

- Two periodic tasks which will respectively include only `CAN_transmit` and `CAN_receive` functions, these tasks will then be placed in two different periodic calls in the file `OSTasks.C,`

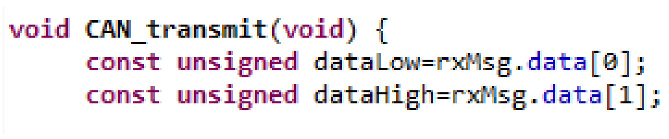
NOTE: Use a faster cycle time for the receiving task, while place the transmission in a slower one.

The general setting description can be found in ***Section 1.6*** of ***"EVduino MultiTask IDE – User Manual"***.

To build, upload on the executable file on the board, run the project and properly select the setting of the IDE, refer to ***Section 2*** of the same document.

Moreover, few more modifications are needed to perform the test of CAN library which are:

- In *CAN_lib.c*, in the function void `CAN_transmit(void)` add the lines in *Figure 6*, to send a previously received message.



```
void CAN_transmit(void) {
    const unsigned dataLow=rxMsg.data[0];
    const unsigned dataHigh=rxMsg.data[1];
```

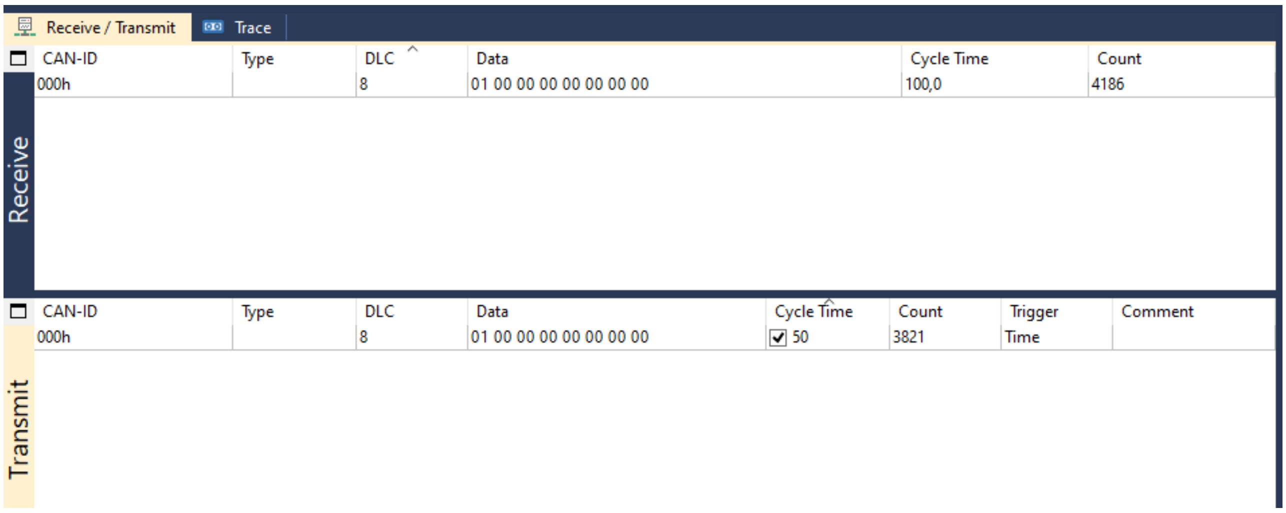
Figure 6 Lines to add/modify in CAN_lib.C

Then:

- Write a CAN message using a CAN monitoring tool (such as PCAN-View with a PCAN-USB and a termination), set the cycle time of the sent message in sync with the cycle time selected for the transmitting task.

- Finally send the message via the monitoring tool, you should be able to see the same message you sent.

Summarizing, two CAN nodes of a network are present (PC and EVduino), a message will be sent at a certain cycle time (in *Figure 7*, 50 ms is chosen) via an external node (the PC) that simulates the presence of a CAN network, the same message is received by the EVduino microcontroller with the frequency set by the receiving task and sent back to the PC at the cycle time selected for the transmitting task (100 ms in *Figure 7*).



CAN-ID	Type	DLC	Data	Cycle Time	Count
000h		8	01 00 00 00 00 00 00 00	100,0	4186

CAN-ID	Type	DLC	Data	Cycle Time	Count	Trigger	Comment
000h		8	01 00 00 00 00 00 00 00	✓ 50	3821	Time	

Figure 7 PCAN view of a CAN message sent from PC and received back.

Simulink Block Generation

MATLAB Legacy Code Tool

After having tested the library software developed, the following step was to modify according to our need and build the Simulink EVduino library blocks.

The MATLAB Legacy Code Tool is a tool that is able to integrate external C and C++ code with MATLAB. [4]

It is used for interfacing existing legacy code with the MATLAB environment, allowing users to reuse and integrate code written in C or C++ within MATLAB.

The Legacy Code Tool can be used to create a MATLAB interface or a Simulink block to an external C or C++ function or library, which can then be used directly from the software.

The tool supports both static and dynamic libraries and provides an easy-to-use graphical interface for configuring and building the interface code.

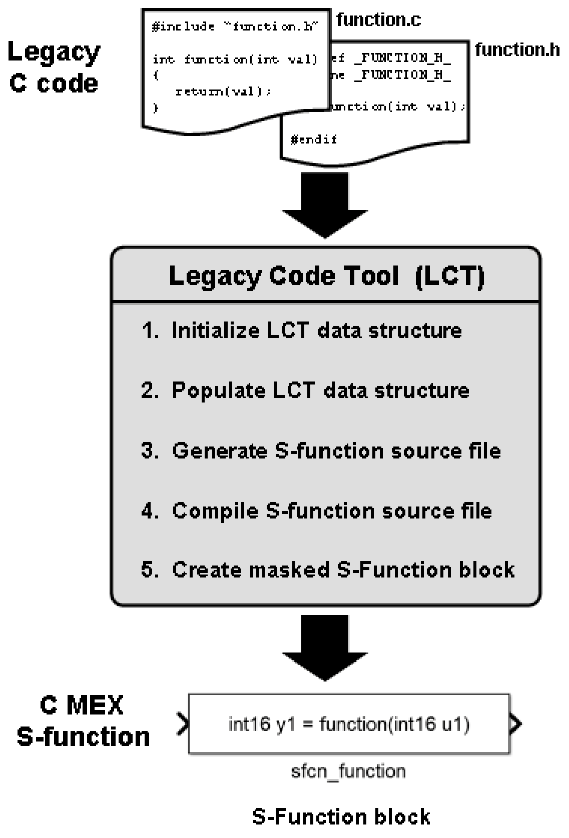

Legacy C code
function.c
function.h
Legacy Code Tool (LCT)
1. Initialize LCT data structure
2. Populate LCT data structure
3. Generate S-function source file
4. Compile S-function source file
5. Create masked S-Function block
C MEX S-function
int16 y1 = function(int16 u1)
sfcn_function
S-Function block

Figure 8 MATLAB Legacy Code Tool general procedure

EVduino Simulink Library

In order to develop and generate the Simulink blocks, the *CAN_lib.C* code needs to be adapted to meet our requirement.

In fact, each function corresponds to a block, and, apart from the `CANinit` the other two were modified in their declaration, to add the input/output signals that must be interfaced with the model, the new functions are:

- `void CAN_transmit(uint32 tx_id, uint32 datatx_low, uint32 datatx_high, uint8 N);`

  The block behaves the same way as described in the chapter *Library Development,* apart from the fact that here the user must select different indexes N from a drop-down list in the block if multiple CAN messages are being sent simultaneously (output of the same task).

  In the declaration are written the input port of the block, that will be interfaced with the other Simulink blocks.

  The first input is the id of the message itself as a uint32, the other two are the two halves of 64-bit message as two uint32, DataLow and DataHigh.

  NOTE (1): To use the *CAN PACK* block from *Vehicle Network Toolbox* and *.dbc* files a conversion is needed in between this and *CAN_transmit* block.

  NOTE (2): Before using the blocks with *CAN Pack/Unpack* blocks write this command `canMessageBusType` in MATLAB's command window or in a .m file in which the parameter of the Simulink model may be defined.
  This command creates a CAN type bus useful for the previous mentioned conversion.

- `void CAN_receive(uint32 *rxid, uint32 *dataLow, uint32 *dataHigh, uint8 *length, uint8 *extended, uint8 *error, uint8 *remote, double *timestamp);`

  Even this block behaves almost in the same way as described in the chapter *Library Development.*

In the declaration are written the output port of the block, that will be interfaced with the other Simulink blocks, and since C does not allow to return multiple values from a function, this has been done using *call by reference* method.

The first output is the id of the message itself as a uint32, the other two are the two halves of 64-bit message as two uint32, DataLow and DataHigh. The other outputs are needed for the conversion, but they do not provide useful data.

NOTE: To use the *CAN UNPACK* block from *Vehicle Network Toolbox* and *.dbc* files a conversion is needed in between *CAN_receive* block and the unpack.
The order of the Outputs is the one in the declaration of the function, it differs from the order of the CAN bus selected in the *Bus Creator* block needed for the conversion.

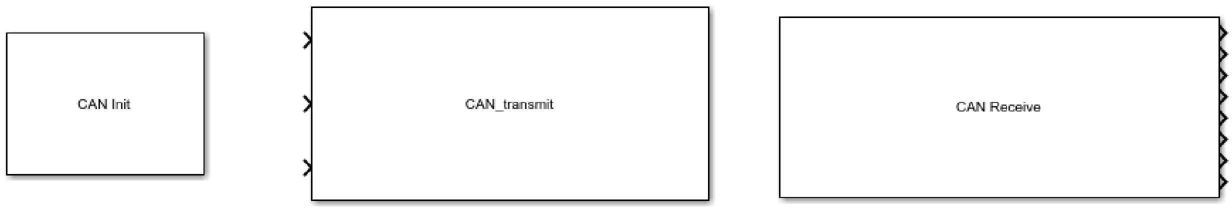


Figure 9 EVduino Simulink blocks generated with MATLAB.

Digital Filter

To show the added functionality of the library, it has been built a digital filter as described in the manual ***EVduino Simulink Support Package - Digital Filters Tutorial.***

The filter is a first order low-pass filter, and, in *z-transform* domain has a transfer function of the form: $H\ (z) = K\frac{1-e^{-\frac{T}{\tau}}}{z-e^{-\frac{T}{\tau}}}$

Where *K* is the filter's gain, *T* is the sampling time and τ is the time constant.

Below, in *Figure 10* can be seen the schematic of the circuit built for the test:

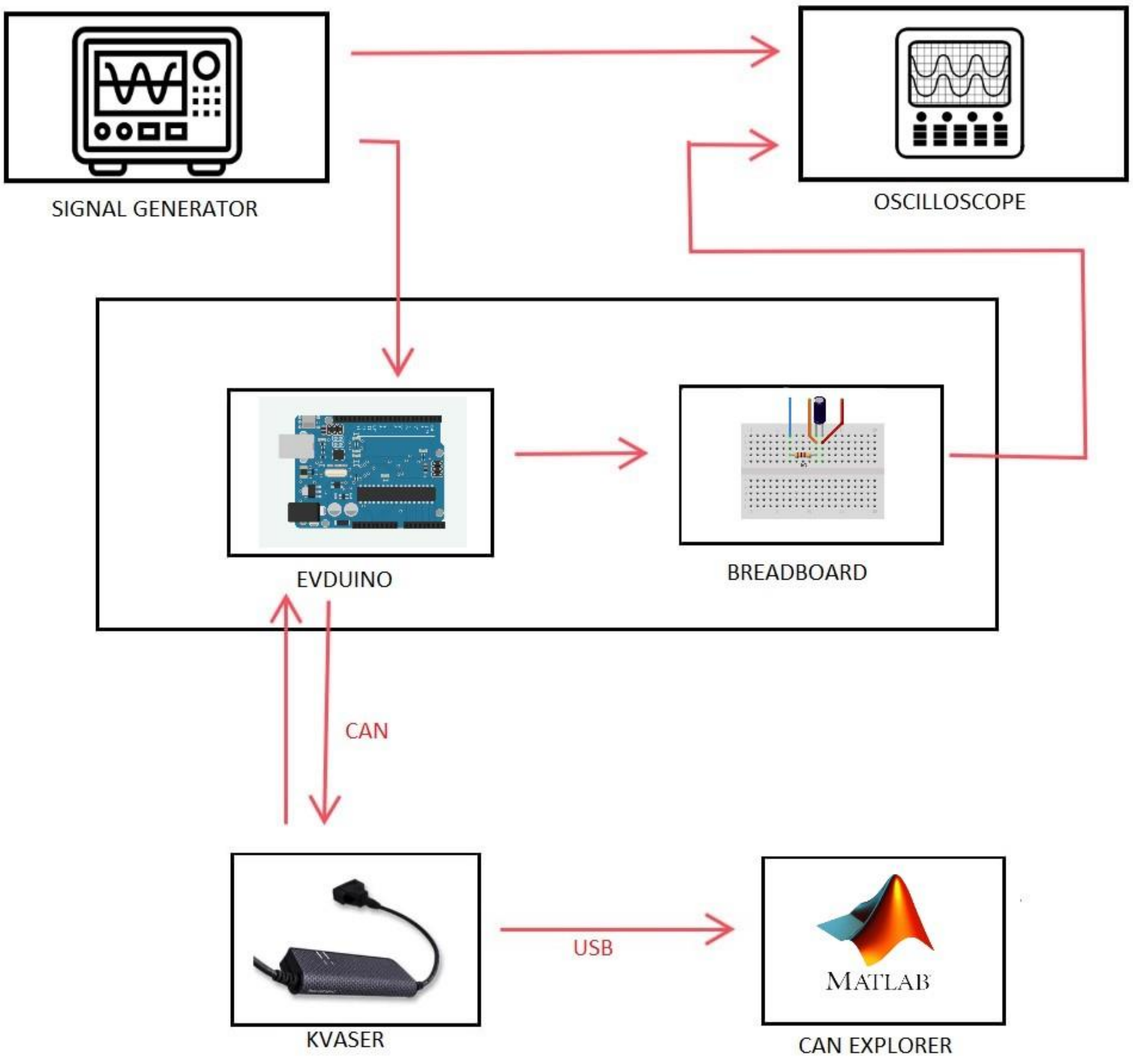


Figure 10 Scheme of the Digital Filter test environment.

Moreover, the filter Simulink model is composed by three main cyclic tasks plus one initialization task (*Initialize Function* for the peripherals initialization):

- CHECK ON OFF: a 5 ms (200 Hz) task that checks the digital state of the switch and implements the debouncing logic (for each task, lower execution time → higher priority).

- FAST TASK: here it is defined the transfer function of the filter, the task has a 20ms cycle time (50 Hz) and the filter also has same sampling time T_{fast}.

- SLOW TASK: same of the previous task but it is slower (50ms cycle time, 20 Hz) and its sampling time T_{slow} is also 50 ms.

Below in *Figure 11* is represented the Simulink Model scheme of the filter.

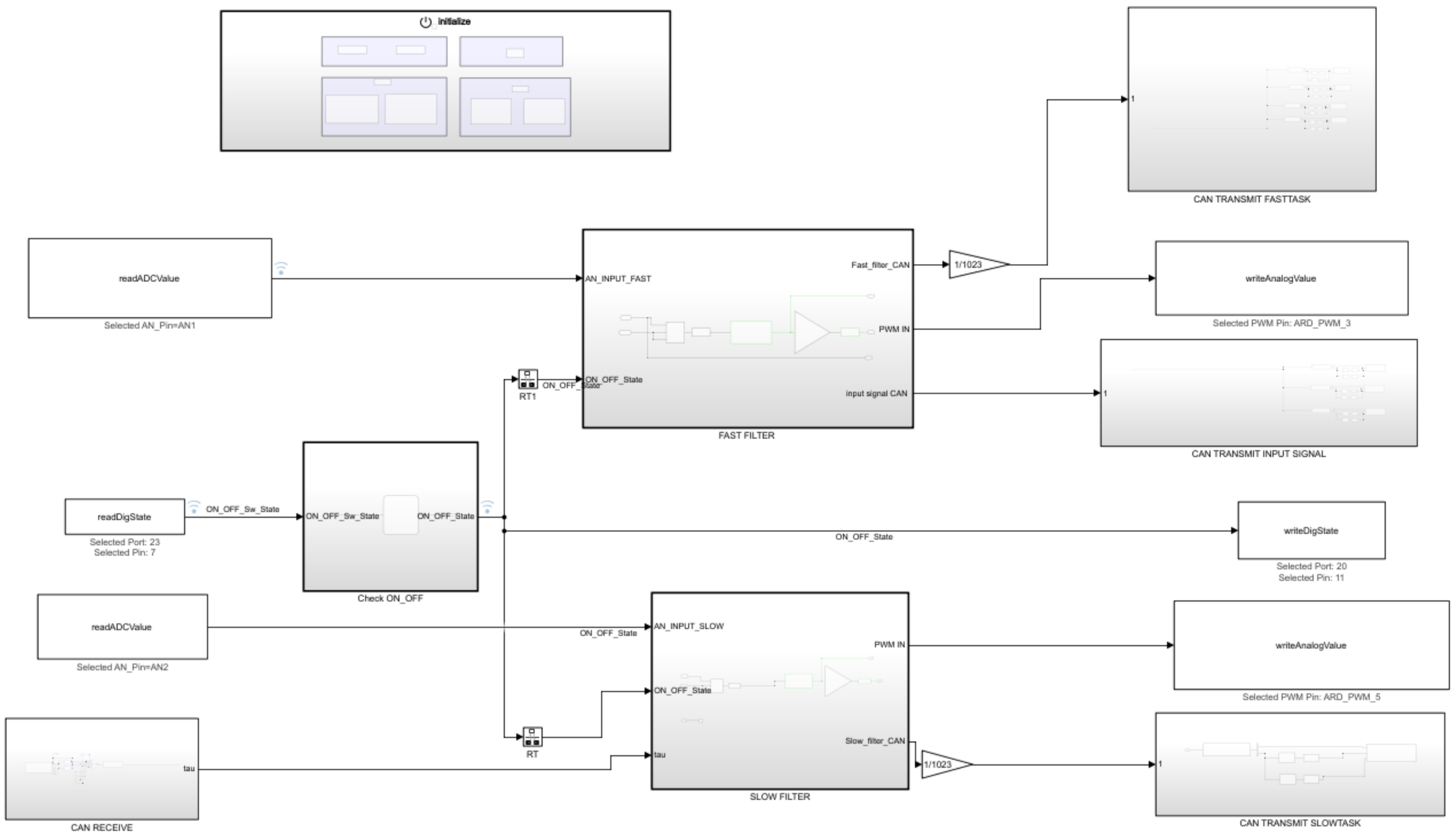


Figure 11 Simulink model of the filter

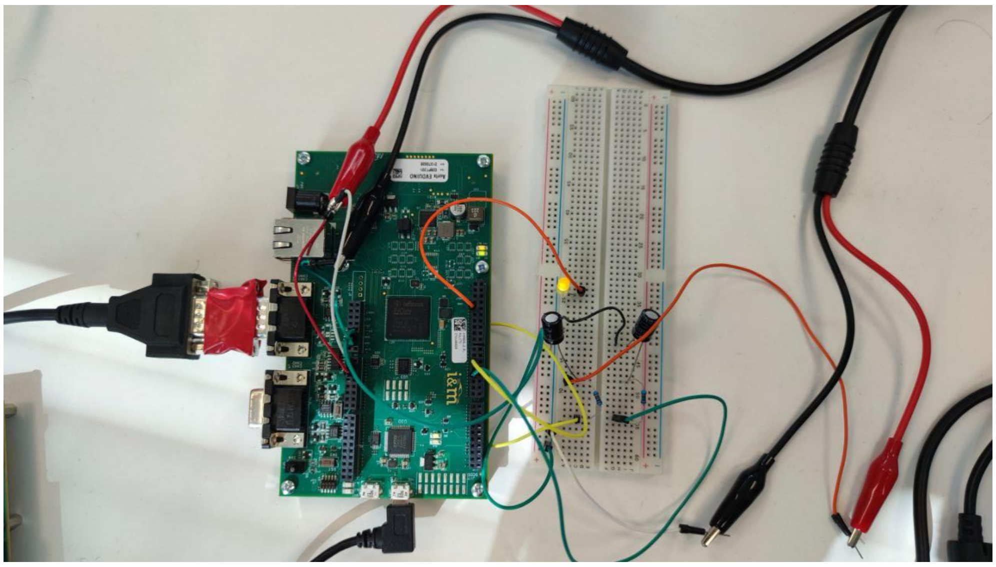


Figure 12 EVduino board with CAN monitoring tool connected (left).

Test

The test set was to log via CAN some of the data of the filter, in particular the Voltage of the Input Signal, SLOWTASK and FASTTASK and the Boolean value of the ON_OFF.

The signals were stored in a *.dbc* file (CAN Database, where the data are stored and structured), that can be selected in CAN PACK and, after a conversion, sent via CAN Transmit blocks.

They were performed using MATLAB CAN Explorer interface and Kvaser Leaf Light v2 as monitoring tool, a glimpse of the test result while switching ON and OFF the filter can be seen below in *Figure 13.*

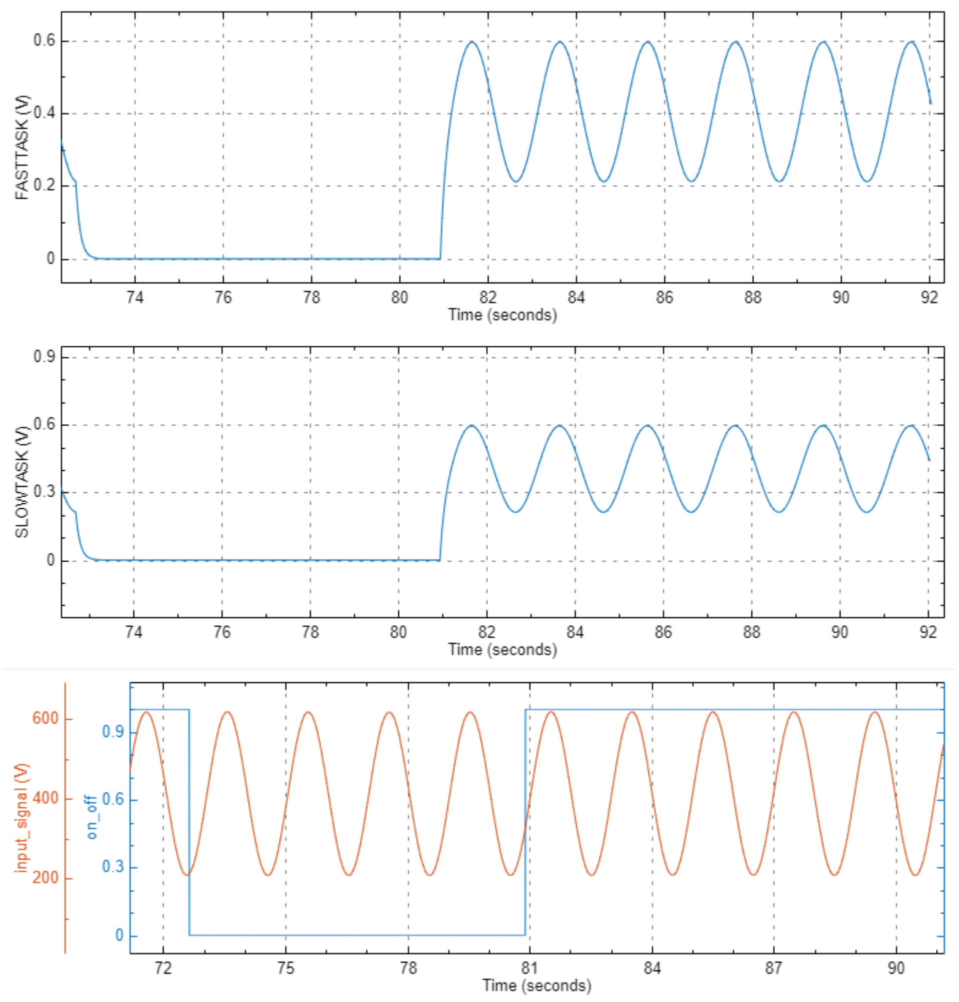
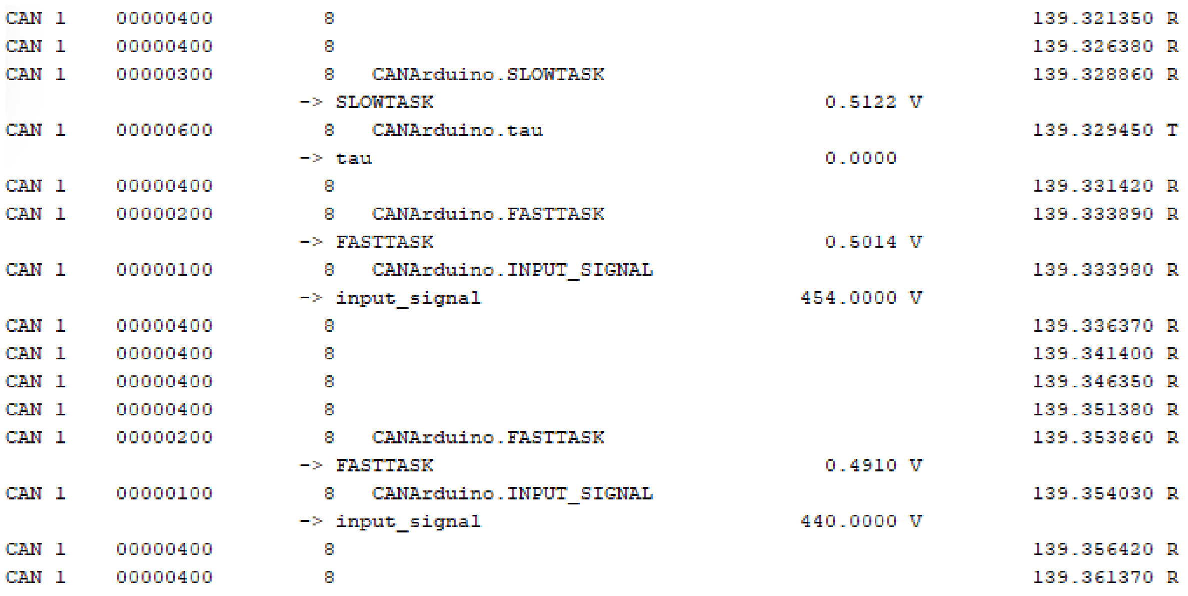


Figure 13 CAN Explorer signals interface view

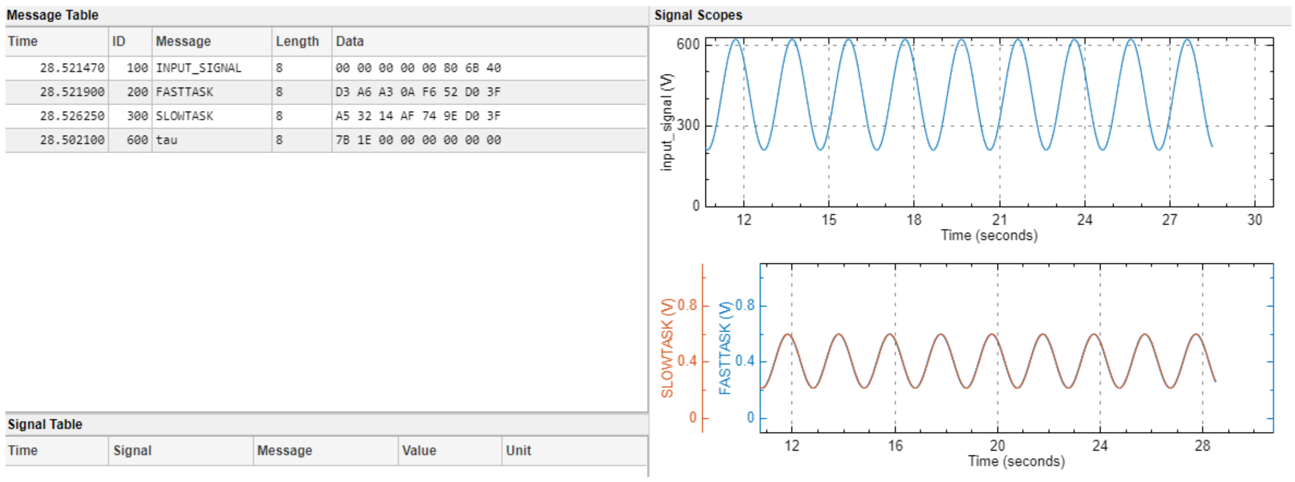
Also, the receiving block has been tested in the same way, and since the CAN message could not be sent via MATLAB, another free software can be used, KVASER CanKing.

In *Figure 14* are displayed both the message *CANArduino.tau*, transmitted via tool to the board (marked with T in the last column), and the other messages in the opposite direction (R).



```
CAN 1     00000400       8                                                   139.321350 R
CAN 1     00000400       8                                                   139.326380 R
CAN 1     00000300       8   CANArduino.SLOWTASK                             139.328860 R
                       -> SLOWTASK                         0.5122 V
CAN 1     00000600       8   CANArduino.tau                                  139.329450 T
                       -> tau                              0.0000
CAN 1     00000400       8                                                   139.331420 R
CAN 1     00000200       8   CANArduino.FASTTASK                             139.333890 R
                       -> FASTTASK                         0.5014 V
CAN 1     00000100       8   CANArduino.INPUT_SIGNAL                         139.333980 R
                       -> input_signal                   454.0000 V
CAN 1     00000400       8                                                   139.336370 R
CAN 1     00000400       8                                                   139.341400 R
CAN 1     00000400       8                                                   139.346350 R
CAN 1     00000400       8                                                   139.351380 R
CAN 1     00000200       8   CANArduino.FASTTASK                             139.353860 R
                       -> FASTTASK                         0.4910 V
CAN 1     00000100       8   CANArduino.INPUT_SIGNAL                         139.354030 R
                       -> input_signal                   440.0000 V
CAN 1     00000400       8                                                   139.356420 R
CAN 1     00000400       8                                                   139.361370 R
```

Figure 14 Output view of Kvaser CanKing.



Message Table

Time	ID	Message	Length	Data
28.521470	100	INPUT_SIGNAL	8	00 00 00 00 00 80 6B 40
28.521900	200	FASTTASK	8	D3 A6 A3 0A F6 52 D0 3F
28.526250	300	SLOWTASK	8	A5 32 14 AF 74 9E D0 3F
28.502100	600	tau	8	7B 1E 00 00 00 00 00 00

Signal Table

Time	Signal	Message	Value	Unit

Figure 15 Same Output view in CAN Explorer.

Conclusion and Future Works

In conclusion, this kind of approach is especially efficient due to the fact that reduces the introduction of bugs of human nature and, most of all, accelerates the software development process, for instance giving the possibility of integrating new functions into previous version of the code.

The user will be able to model multitask application and generate code, with some limitation linked to the IDE configuration.

In fact, one of the next steps of developing process of EVduino could be to run the code directly on the board without the need of a PC.

Another important step is to design via Simulink a way to tune some parameters via CAN messages.

After the completion of these passages, the user could be capable of implementing a complete control logic with CAN network, in order to use a "rugged" version of EVduino as a Vehicle Management Unit.

Bibliography

[1] "*CAN bus*", Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/CAN_bus

[2] Infineon Technologies AG, *"AURIX™ TC270 / TC275 / TC277 Data Sheet"*, Munich, Germany, 2017.

[3] Evidence S.r.l, *"ERIKA Enterprise Manual, Real-time made easy"*, Version 1.45, Pisa, Italy, 2012.

[4] The Mathworks Inc., "Integrate C Functions Using Legacy Code Tool", The Mathworks.