SER 17/18
INTERCONNECTING MBED NODES WITH CAN: EMULATING A CAR CAN NETWORK

Authors: Duarte Botelho (77965), João Rodrigues (81843) & Mariana Martins (80856)

1. INTRODUCTION

The goal of this project is to interconnect Mbed boards with a Controlled Area Network (CAN) bus in order to evaluate this type of network when concerning real-time systems and reliability. To make this possible, we will be implementing an automotive CAN network with different sensors and actuators connected to several Mbeds (the nodes of the network).

Modern cars are equipped with hundreds of sensors all spread across them, creating a need to connect them through a reliable in-vehicle communication network. We wanted to solve the problem with a realistic approach, thus we will be implementing this network with three Mbed nodes spread across an emulated car. One will be in the front bumper, one next to the dashboard inside the car, displaying information collected by the sensors to the driver, and one in the rear bumper. Each node will have a key part in the network since each sensor collects and detects different environment situations that need to trigger an effect across the vehicle (for example, the light sensor in the front bumper must turn the front and rear lights on, when the car enters a tunnel).

In section 2, we discuss in bigger depth the characteristics of the network, what was implemented, including the sensors and actuators used, the node behaviour, how the communication was made inside the network and finally, how the energy management was handled.

In section 3, there is a description about the physical connections that must be used in order to reproduce the network, all the nodes, the sensors and actuators.

Finally, in section 4, the results of the implementation are compared to what was expected, and some conclusions and further steps of the project are taken into account.

All our code is available at
https://github.com/Mrrvm/SER/tree/master/Project.

2. IMPLEMENTATION

As said before, we created a network with three Mbeds spread across the vehicle. Fig. 1 and 2 show, respectively, the sensors and actuators connected to each of the Mbeds as well as how they are connected to each other through the CAN Bus. A more detailed description of this hardware will be given in subsection 2.2.

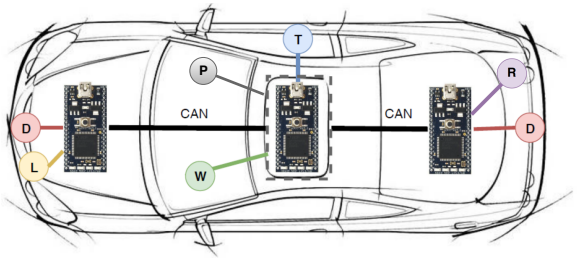

Fig. 1: Representation of the sensors used
D - distance sensor, L - light sensor, P - potentiometer, T - temperature sensor, W - Weight Sensor.

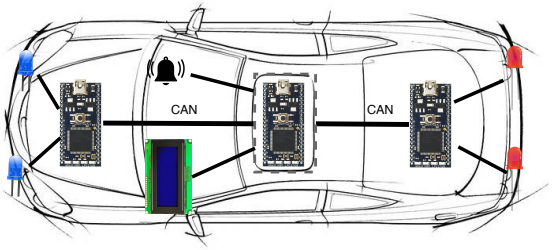

Fig. 2: Representation of the actuators used.

The Controlled Area Network (CAN) Bus was created by Robert Bosch GmbH and released at the Society of Automotive Engineers in 1986 as a more convenient solution to the complex problem of point-to-point wiring between nodes and the **electronic control units** (ECU). Before CAN, all sensors and actuators - or nodes - where individually connected to the electronic control unit (ECU) through wires. The ECU did all the processing based on data collected from the sensors and controlled the vehicle accordingly through the actuators. This method worked well while the number of nodes was small but as the number of sensors, actuators and therefore number of cables raised, this implementation led to overload in the single ECU, more weight to the vehicle, higher system complexity and higher overall cost. The solution was to replace the single ECU with multiple ECUs, each collecting, processing and controlling specific parts of the vehicle (through their own sensors and actuators). This new set of ECU and the respective sensors/actuators is called *module*. Each module has a connection to a shared bus that only required two cables (**CAN High and CAN Low**) across the entire vehicle and

could send as well as receive information from all the other ECUs. This corresponds to our implementation. Each Mbed represents an ECU, that collects information from the sensors at its disposal, processes it and broadcasts to the network a message containing the data and an identifier. The identifier distinguishes the messages and establishes their priority on the network. In subsection 2.4 will overlook this aspect better.

2.1. Mbed

Mbed [1] is a platform for IoT devices based on 32-bit ARM Cortex-M microcontrollers. We used the board NXP LPC1768, with a Cortex-M3 running at 96 MHz, 32 KB RAM, 512KB FLASH storage. It also has 40 General-Purpose Input/Output (GPIO) pins, 6 of which are ADC and is powered by 4.5-9V.

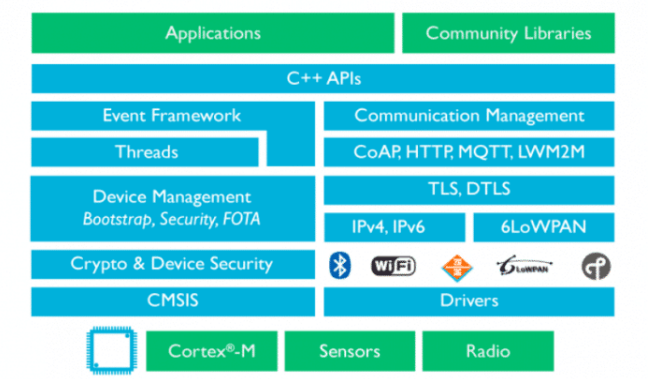

Fig. 3: Layers of the Mbed OS.

To control the board we used the Mbed OS (Fig. 3), which is a Real Time Operating System (RTOS) built specially for this devices, based on C++. This allows us to use, for example, threads, interrupts, semaphores and mutexes to approach to the problem more efficiently.

Mbed was created around the idea of building a collaborative tool for IoT. Thus the platform provides an online development environment that builds the programs directly to binary, ready to deploy on the board. There is a strong community that provide numerous resources available online. In our project we used this resources to get information from the sensors. Information for this can be found in the next subsection when we talk more about sensors.

Currently there are two versions available of the Mbed OS, 2.0 and 5.0. We started to build our project around the newer version to take advantage of the improvements made by the developers, however we found that some libraries did not work as expected with it, so we used remained with the older version.

2.2. Sensors and Actuators

Multiple sensors and actuators were used, all of them somehow related with a specific car feature. A list and short explanation about the sensors and actuators used is provided in this section and each sensor and actuator's detailed characteristics are presented in their respective datasheets referenced along the text.

2.2.1. Ultrasonic Sensor - HC-SR04

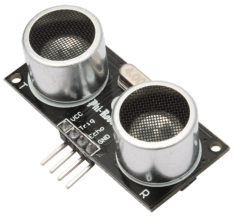

Fig. 4: Ultrasonic Sensor.

The goal of Ultrasonic sensor [2], represented in Fig. 4, is to measure the distance of the nearest obstacle. To achieve it, it emits an ultrasound at $40\,000$ Hz that travels through the air and reflects in the object in its path, if there is one. As we know the speed of the sound is $340\,\mathrm{m/s}$,so we can estimate the distance, d in m, that the sound wave traveled by applying (1), where t is the period of time the wave took to travel from the sensor to the object detected and back to the sensor.

$$d = \frac{t \times 340}{2} \tag{1}$$

Besides VCC and GND, this board has an Echo pin and a Trigger pin. The Trigger pin is an input pin. This pin has to be kept high for $10\,\mu$s for the ultrasonic wave to be sent. The Echo pin is an output pin, that remains *HIGH* for the period of time the ultrasonic wave took to return to the sensor.

The library HC-SR04 [3] was used to access the sensor data. By calling the method *getCm()* we get a float with the current distance value which is then converted to int to be transmitted over the CAN bus. The purpose of this was to determine the distance between the front or rear bumper of the car and a possible object in its path.

2.2.2. Light Dependent Resistor - LDR

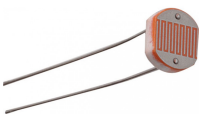

Fig. 5: Light Dependant Resistor.

A photo-conductive light sensor, represented in Fig. 5 consists on a light dependant resistor [4], which doesn't produce electricity but simply changes its physical properties when subjected to light energy. The electrical resistance changes in response to changes in the light intensity, as represented in Fig. 6. Photo-conductivity results from light hitting a semiconductor material which controls the current flow through it. Thus, more light increases the current for a given applied voltage.The LDR requires an external pull-up resistor. The goal of this sensor is detecting light above or below a certain threshold, turning the car's front and rear lights on or off.

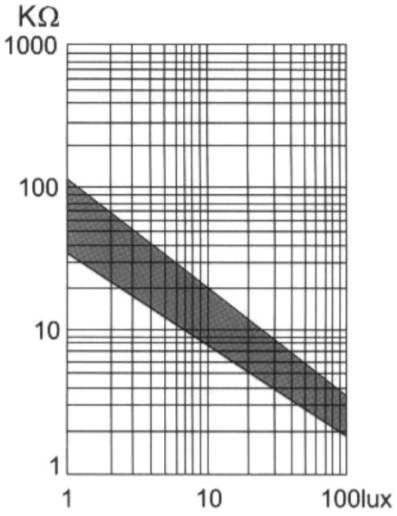

Fig. 6: LDR resistance varying with light intensity.

2.2.3. *Rain Sensor - SN-RAIN-MOD*

In the Rain sensor [5], represented in 7, the resistance of the collector board varies accordingly to the amount of water on its surface. When the board is wet, the resistance decreases, and so does the output voltage. The rain sensor has a built-in potentiometer for sensitivity adjustment, a digital output, an analog output, a power LED that lights up when the sensor is turned on and digital output LED. The analog signal is sent to the Mbed and read as a raw value, similarly to what happened with the LDR. By comparing certain thresholds, the intensity of the rain can be determined and the speed of the windshield wipers defined.

Fig. 7: Rain Sensor.

2.2.4. *Weight Sensor: Load cell with ADC - HX711*

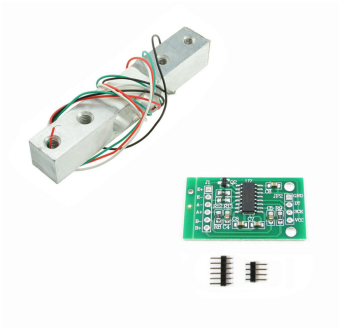

Fig. 8: Weight Cell.

A load cell [6], represented in Fig. 8, is a transducer that is used to create an electrical signal whose magnitude is directly proportional to the force being measured in a bar strain gauge. In bar strain gauge load cells, the cell is set up in a "Z" formation so that torque is applied to the bar and the four strain gauges on the cell will measure the bending distortion, two measuring compression and two tension. When these four strain gauges are set up in a Wheatstone bridge formation, it is easy to accurately measure the small changes in resistance from the strain gauges.

In a Wheatstone bridge, represented in Fig. 9, where V_{in} is a known constant voltage and V_G is the output voltage, we can see that, if $\frac{R1}{R2} = \frac{R3}{R4}$ then V_G is 0, but if there is a change to the value of one of the resistors, V_G will have a resulting change that can be measured and is governed by (2), using ohms law.

This way, a voltage measurement can be directly associated with the force being applied, and to implement this sensor it was used a library [7] that makes such a function available, converting the voltage output to the mass imposed

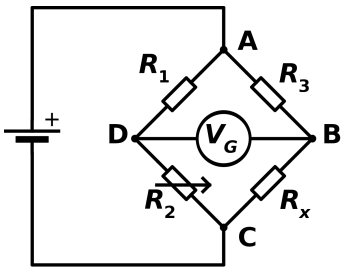

Fig. 9: Representation of a Wheatstone bridge disposition.

on the sensor. This sensor is responsible for detecting when someone entered or left the car.

$$V_G = V_{in} \cdot \left(\frac{R3}{R3 + R4} - \frac{R2}{R1 + R2} \right) \quad (2)$$

2.2.5. *LED's*

The LED [15] is the well known two-lead semiconductor light source. A p–n junction diode that emits light when activated i.e., when a suitable current is applied to the leads, electrons are able to recombine with electron holes within the device, releasing energy in the form of photons. This effect is called electroluminescence, and the color of the light is determined by the energy band gap of the semiconductor. The white LED represents the front lights of the car, while the red LED represents the rear lights of the car.

2.2.6. *Temperature Sensor*

The LM75B [12] is implemented in the Mbed Application Board via a I2C bus. It's a temperature-to-digital converter using an on-chip band gap temperature sensor and Sigma-Delta A-to-D conversion technique with an overtemperature detection output. Delta-sigma modulation converts the analog voltage into a pulse frequency and can be understood as pulse-density modulation or pulse-frequency modulation depending on implementation. In general, frequency may vary smoothly in infinitesimal steps, as may voltage, and both may serve as an analog of an infinitesimally varying physical variable such as acoustic pressure, light intensity, and in this case, temperature.

2.2.7. *Potentiometer*

The Potentiometer [13] is, similarly with the Teamperature Sensor, integrated in the Mbed Application Board. The usual implementation consists in a simple three-terminal resistor with a sliding or rotating contact that forms an adjustable voltage divider. If only two terminals are used, it acts as a variable resistor. The potentiometer is used as emulation of the car key rotation, using a threshold to define when the car is being turned on or off.

2.2.8. *LCD*

A liquid-crystal display (LCD) is a flat-panel display which uses the light-modulating properties of liquid crystals. These do not emit light directly, instead using a backlight or reflector to produce images in color or, in our case, monochrome. LCDs are available to display arbitrary images (as in a general-purpose computer display) or fixed images with low information content, which can be displayed or hidden, such as preset words, digits, and seven-segment displays, as in a digital clock. They use the same basic technology, except that arbitrary images are made up of a large number of small pixels, while other displays have larger elements. This is the type of LCD that we have in our project. We used the library C12832 [11] to facilitate the communication with the 128x32 LCD.

2.2.9. *Buzzer*

A buzzer or beeper is an audio signalling device, which may be mechanical, electromechanical, or piezoelectric. The last is the one used in this project. This element may depend on an oscillating electronic circuit or other audio signal source, driven with a piezoelectric audio amplifier, and also on the acoustic cavity resonance in its build. The piezo [8] was used to warn the driver that something could be wrong. For instance, if the car is close to an object, the piezo starts ringing and as the distance shortens, the beep frequency increases for the driver to take action. PWM Tone Library [9] was used to produce sounds when the central node received some commands.

2.3. Node Behaviour

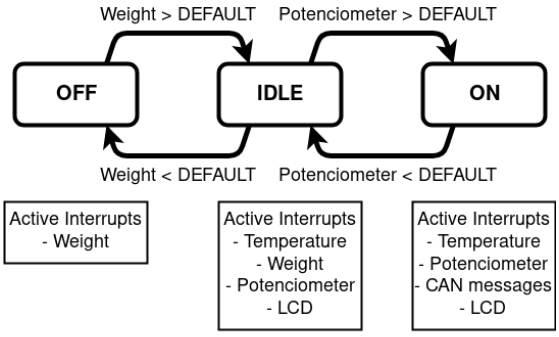

Fig. 10: State machine of the central node.

The central node is the most important one, because it controls the peripheral nodes main behaviour. To better understand how the system works, we will give an example based on Fig. 10. When the driver unlocks the car, its state is OFF and the weight interrupt is on, if he enters the car, then *IDLE* state is activated, because the weight sensor senses at least the a default value, and the interrupts to measure the temperature and the potenciometer and to display messages on the LCD are initialized. If the weight sensor stops detecting weight (driver leaves the vehicle), the node enters the *OFF* state again, stopping all the interrupts except the weight one. On the other hand, if the weight sensor continues to sense at least its default value, the vehicle stays in *IDLE* mode until the potentiomenter is rotated more than its default value

(emulation of the car key). When this happens the node enters *ON* mode, activating the CAN message handler interrupt, stopping the weight interrupt and sending a wakeup message to the peripheral nodes. When the potentiometer value goes below its default value, the node goes back to *IDLE* mode and sends a sleep message to the peripheral nodes, stopping the CAN messages interrupt and turning back on the weight interrupt.

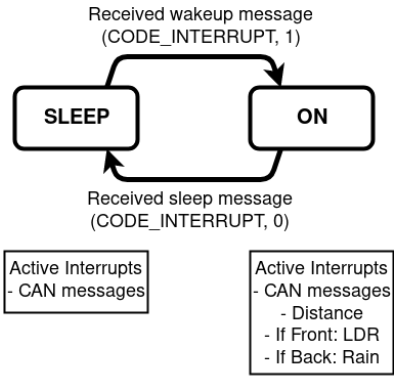

Fig. 11: State machine of the peripheral nodes.

As for the flow of the peripheral nodes (Fig. 11), they are initialized in *SLEEP* mode, where the only interrupt active is the CAN messages handler. Then when the central node sends them a wake up message, the nodes enter the *ON* state and activating the interrupts that measure the distance and luminosity (in the front node) or rain intensity (back node). Then, if the center node sends a sleep message, the peripheral nodes enter *SLEEP* mode again, disabling the all interrupts except the CAN messages one.
To explain the interrupts further, we can take a look at figures 12 and 13. Each one of them is a timer, set to check a certain aspect from a defined time to time. This time was chosen according to the importance of that aspect for the vehicle/driver. To implement this, we used the class RTOSTimer from the RTOS library [16], since these timers are handled in a dedicated thread and the callback functions run under its control, making the interrupts more reliable, specially given the circumstances.

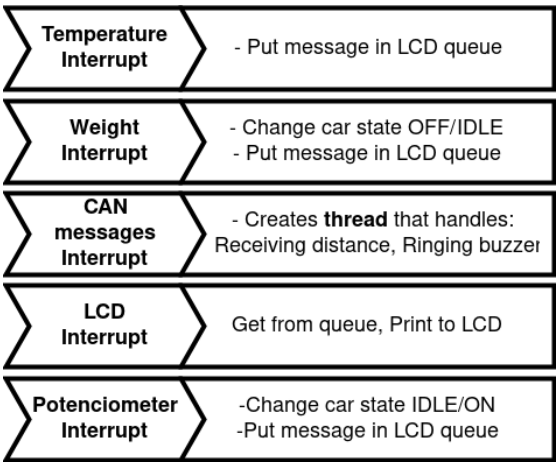

Fig. 12: Interrupts on the central node summary.

- **Temperature Interrupt**
(TEMP_CHECK_TIME=5000ms)
When the handler for this interrupt (tempHandler) is called, if the temperature is different from what the user set as default, a message with the temperature difference is pushed to the LCD queue. This queue was also created with the RTOS library, using the class Queue, in order for the LCD to consume the messages. This class does not allow priority, and so that is not taken into account.

- **Weight Interrupt** (WEIGHT_CHECK_TIME=5000ms)
When the handler is called, if the weight is above a certain default, the car changes state to IDLE, otherwise goes to OFF. As well as the temperature interrupt, it pushes a message to the LCD queue. Both this and the temperature's are not urgent and so 5 seconds are good enough.

- **CAN messages Interrupt**
(MSG_CHECK_TIME=500ms)
Here when the handler is called, we verify if there is any message to read on the CAN network. In case there is, a thread is created to handle it. The central node only receives distance messages from the peripheral nodes, which require a certain amount of CPU time to be treated since they trigger a loop ringing the buzzer and also need a Mutex protecting it (front and back distance can trigger ringing at the same time - parking in parallel for example). Therefore, in order to not compromise the system reliability, the spawn of a thread each time it receives a message is essential. The Mutex class is also from RTOS library. Due to the importance of the distance awareness when driving, this interrupt has a time lower than the others.

- **LCD Interrupt**
(LCD_CHECK_TIME=4000ms)

When triggered, a message is read from the queue and displayed on the LCD and remains on it, till the timer triggers the interrupt again. Here the time was chosen regarding what we thought was better for the driver to have time to read the message.

- **Potenciometer Interrupt**
  (POT_CHECK_TIME=500ms)
  When the driver turns the key on, all the other system features that were sleeping (distance, lights and windshield cleaner) that are set on the peripheral nodes, must wake as fast as possible, thus the time is low since there is still the added time of sending a message to CAN, of the peripheral node's interrupt being triggered and of receiving the message. As explained before, this defines the state IDLE/ON.

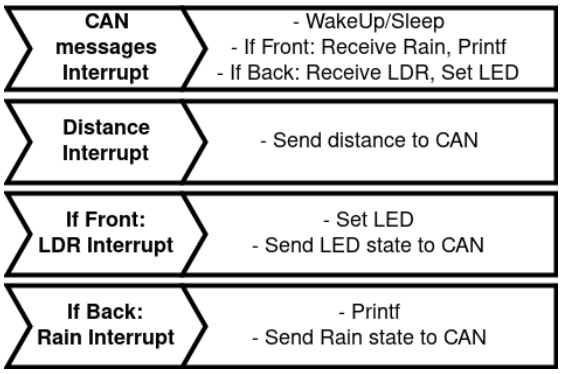

Fig. 13: Interrupts on peripheral nodes summary.

- **CAN messages Interrupt**
  (MSG_CHECK_TIME=500ms)
  On trigger, 2 different kind of messages can be received, a wake up/sleep message or a ldr/rain message. In case it's the first, the state of the node changes accordingly and the consequences of that are later managed. If it's the latter, the lights turn on/off according to the content of the message if we are on the back node, otherwise a message saying the windshield cleaners state is printed. The time was chosen like this for the same reasons stated above.

- **Distance Interrupt**
  (DIST_CHECK_TIME=500ms)
  Because it's a priority feature, the time is low since a message must still be sent and received on the central node for the buzzer to ring.

- **LDR Interrupt**
  (LDR_CHECK_TIME=2000ms)
  When triggered, the LDR analog value is checked, and if above or below a certain default, it turns on or of the lights and sends a message saying LIGHTS_OFF or LIGHTS_ON. Because this also a priority feature, the time was set what we thought would be suitable to have the lights turning on, once the maximum time is 3000ms ($LDRtime_{max} + CANtime_{max} * 2$) plus the time spent reading and writing to the sensors and actuators.

- **Rain Interrupt**
  (RAIN_CHECK_TIME=2000ms)
  This is the same as the LDR, taking also a maximum of 3000ms.

All the default values talked about were tested and defined experimentally and are presented, as well as all this time periods on interface.h code file.
Before talking about the maximum time paths, there are still three times to have into account, the time the program takes to realize it changed state (1000ms in all nodes, STATE_CHECK_TIME), the time the peripheral takes to trigger when sleeping (2000ms, SLEEP_CHECK_TIME) and that same time but when they are awake (500ms, MSG_CHECK_-TIME). When the peripheral nodes are sleeping, they take longer to trigger a CAN message receive event, and so the maximum time it would take for the system to become fully ON is

$$PotenciometerTime_{max} + CANtime_{max} + SleepingCANtime_{max} + StateTime_{max} * 2 = 6000ms \quad (3)$$

, which is enough since a driver takes around that time between rotating the key and start driving.
It was important for the interrupts to not have too much workload, so that crashes between interrupts avoided. Using interrupts instead of threads can help optimize the system since they provide low overhead and good latency at low load, which is the case.

2.4. Network Communications

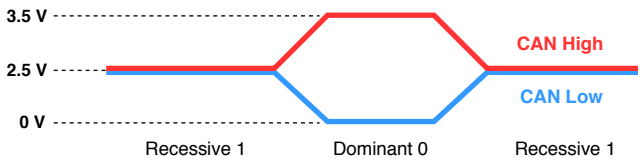

Fig. 14: CAN Low and CAN High signals.

Each CAN message frame consists on a series of **dominant 0's** and **recessive 1's**, created by the two wires (Fig. 14), grouped in the format exemplified in figure 15, and at the end of each wire there is a resistor of 120 Ω. One of the most important fields is the identifier or arbitration field, due to being what establishes the message priority and what identifies

it. The identifier with the lowest number is the most import and has priority over the rest. What if two devices try to send a message at the exact same time? Fig. 16 exemplifies this problem. They both start to send at the same time but because *A* is lower than *F*, *device 2* owns the rights to sending its message and *device 1* stops sending and leaves the bus. Table 1 shows the types of messages on our protocol and below we present the respective description of each message.

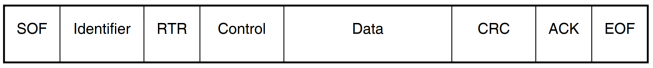

Fig. 15: CAN message frame.

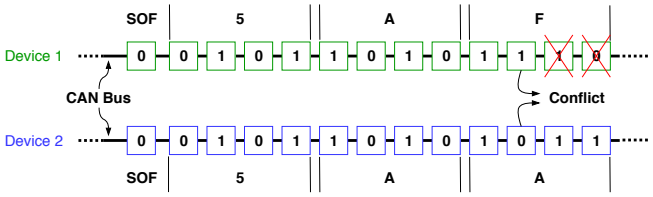

Fig. 16: Example of message conflict between two devices.

Table 1: Protocol.

Identifier	Message Type	Description
0x200	INTERRUPT	Central node sends OFF signal to other nodes if driver leaves the vehicle or ON signal otherwise
0x201	DISTANCE_FRONT	Front node sends the current distance value
0x202	DISTANCE_BACK	Back node sends the current distance value
0x203	LDR	Front node sends LIGHTS_ON signal if external luminosity changed to low or LIGHTS_OFF otherwise
0x204	RAIN	Back node informs of the state the windshield cleaners

2.5. Energy Management

Energy management is an important aspect of our project. Because this system was built to be deployed in a vehicle, it will always receive power from the car battery, which is a fairly stable and reliable power source. Nevertheless, it should be a goal to save as much energy as possible if, for example, the system is to be scaled to more then three nodes having the same core implementation. Another reason is that by reducing the processor load to save power we are increasing the lifespan of the hardware. This is important because car owners need to put the vehicle in service less often, reducing costs. One big benefit from a smart energy management is that if this system is deployed in an electric vehicle we may increase the autonomy range in several kilometers (knowing that cars nowadays have more then 60 ECUs).

In that spirit, we did several things to lower the energy spent, inclusive using the PowerControl library [17]. Citing [19], the consumed power equals

$$Power = CapacitiveLoad \cdot Voltage^2 \cdot ClockFrequency \quad (4)$$

- **PHY_PowerDown()**
  Since we don't need and Ethernet connection, we used this function from the PowerControl library to power it down - "Powers down the PHY and LPC1768 EMAC" [18]. This helps reducing the capacitive load. The more I/O units turned on in the hardware, the more capacitive load.

- **Sleep state**
  According to [18], turning the Mbed boards to sleep mode saves only about 8.4% energy, which is very little compared to the 30.3% when in deep sleep. However, when in deep sleep we cannot wake up the peripheral nodes by CAN. Still this is better than using wait(), since sleep does not clock in new instructions, it turns off the clock on the processor core and waits for a timer interrupt. Thus helps reduce the clock frequency.

- **Dynamic timers**
  As we are using interrupts instead of threads, we can also reduce the clock frequency, since it avoids the Mbed from continuously running in loops that do nothing or wait() functions. In our implementation, the timers are never all on at the same time. Depending on the state each node is, there are certain timers that will be stopped, which is a further improvement on this kind of build.

3. SETUP

To reproduce our current project version, three LPC1768 Mbeds are required (front, central and rear). The board pinout is represented in Fig. 17. To make connection to the CAN bus, each Mbed is connected to one MCP2561 transceiver (p9 - RD, p10 - TD). The front Mbed is also connected to one ultrasonic sensor HC-SR04, where p7 is trigger and p8 is echo, one white LED in pin p5 and to the LDR in pin 20. The back sensor is also connected to one ultrasonic sensor and to one red LED over the same pins and to the rain sensor with pin p20. As for the central node, it is connected to the

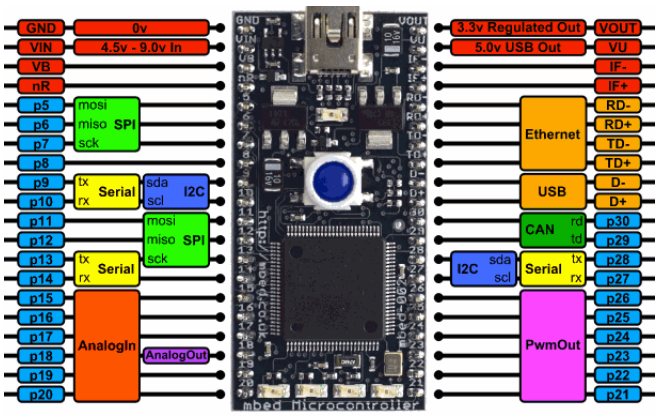

Fig. 17: Mbed pinout

Mbed application board (Fig. 18), which provides built-in temperature sensor (p28 - SDA and p27 - SCL), potentiometer (p19) and display (p5 - MOSI, p6 RESET, p7 - SCK, p8 - A0, p11 - NCS). We also have the central connected to the weight sensor (p12 - DT, p13 - SCK) and to the buzzer (p21). Finally we also use two 120 Ω and two 220 Ω resistors, for the CAN connection and for the LEDs, respectively.

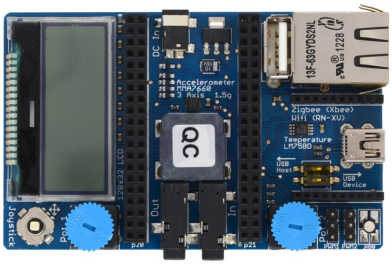

Fig. 18: Mbed application board

4. DISCUSSION

The solution presented in the report implements an emulation of an automotive vehicle network, using Mbed modules as nodes of this network. It establishes a defined relationship between what is seen as a central node, with multiple and different concurrent tasks, which interact with the user (driver), and two decentralized nodes, one located in the front and the other in the rear, which interact only with the CAN network in order to know in what state they should be.

The CAN network was tested successfully with all three nodes, and the network shared values were sent and received by all of them. All the sensors and actuators were also tested separately with success, with all the respective dependant libraries integrated.

As the project was close to completion, some improvements started being implemented, more precisely the energy dependencies and definition of different states for all nodes, depending on the actions of the user and the data perceived in the sensors. These changes, however, weren't tested at their full length, for which the code submitted may present some unexpected behaviour. Also because of this, the report to our frustration does not present any data relating to the energy comsumption of the system nor to the response times.

Although the access to the laboratory was available during the week days (on a limited schedule), it is noted as a disadvantage to the development of a project of this complexity that most of the sensors and actuators used weren't provided, neither the Mbed modules were provided for testing outside the limited hours that the laboratory was opened. The usage of a single node during approximately half of the course of this project hardened the implementation of the fully fledged code in these latter stages, as the time for testing was even less than the usual.

Nevertheless, we think the report succeeds in showing the advantages of a CAN network configuration, specially compared to the previous existent network architectures, and its behaviour in a multi-node, concurrent system. The sensors and actuators used in this implementation cover a great length of different physical properties of the real world that are being processed by the system, as well as different data gathering techniques. The energy management section studies the impact of this topic on the implementation subject, as we cannot in any circumstance forget the importance of saving as many energy as possible, specially in a system like the one here in question, where there is no connection to the main energy network.

In a further development to what was done in this project, one could test the fully working system and look for the best fitted parameters in a real life situation, more precisely the values of the thresholds that allow a better experience for the driver as well as a more efficient functioning of the actuators presented. A better energy optimization could be defined, by understanding in what states and functions, the system spends more energy. And finally, a further study into failure situations should be done, in order to understand if the interrupt-like implementation proves problematic.

5. REFERENCES

[1] Ultrasonic Sensor
tinyurl.com/ser1718-mbed

[2] Ultrasonic Sensor
tinyurl.com/ser1718-ultrasonic

[3] Library HC-SR04
tinyurl.com/ser1718-HC-SR04

[4] LDR
tinyurl.com/ser1718-ldr

[5] Rain Sensor Sensor
tinyurl.com/ser1718-rain

[6] Load Cell Sensor
tinyurl.com/ser1718-weight

[7] HX711 Library
tinyurl.com/ser1718-HX711

[8] Buzzer
tinyurl.com/ser1718-buzzer

[9] PWM Tone Library
tinyurl.com/ser1718-PWM-Tone-Library

[10] LCD
tinyurl.com/ser1718-lcd

[11] Library C12832
tinyurl.com/ser1718-C12832

[12] Temperature Sensor
tinyurl.com/ser1718-temperature

[13] Potentiometer
tinyurl.com/ser1718-potentiometer

[14] Transciever MCP2561
tinyurl.com/ser1718-transciever

[15] LED's
tinyurl.com/ser1718-led

[16] RTOS Library
https://os.mbed.com/handbook/RTOS

[17] PowerControl Library
https://os.mbed.com/users/no2chem/code/PowerControl/

[18] mbed Power Control/Consumption
https://os.mbed.com/users/no2chem/notebook/mbed-power-controlconsumption/

[19] Power Management
https://os.mbed.com/cookbook/Power-Management