Elaboration of a Dummy Engine for Controller Testing

F. Castillo Rodríguez[1], S. A. Sanoja Hernández[2], G. S. Hernández Carrillo[3] , A. Saavedra Ricalde[4] and M. A. Quintana Silva[5].
Engineering and Sciences School, Monterrey Institute of Technology and Higher Education, Mexico State, Mexico.
[1] A01750126@tec.mx, [2] A01749472@tec.mx, [3] A01750124@tec.mx, [4] A01750971@tec.mx,
[5] A01750161@tec.mx.

***Abstract*--In the assignment established by the Training Partner GENERAC, we were required to do a "Dummy Engine" to test a controller that is going to be implemented in an electric generator powered by a combustion engine. The purpose of the "Dummy Engine" is to emulate the signals that the real engine is going to interact with, by sending and receiving the proper messages, so that the controller can be tested without the real engine. Our "Dummy Engine" must be a device that can be connected to a controller by the CAN protocol, with the two wires required (CAN High and CAN Low) through SAE J1939 standard.**

I. Introduction

The base of the project was the CAN protocol to communicate the devices, this protocol only requires a pair of wires to communicate all the messages. The messages in the CAN protocol travel in packs that contain the direction of the transmitter and the code of the message. And we implemented the CAN protocol with Arduino (both its libraries and modules).

The Forming Partner GENERAC provided us with a list of variables and their ranges, that we should be able to modify with our device, variables such as RPM, oil pressure, oil level, etc,

GENERAC also provided us with some "Deep Sea" Controllers so that we could test our projects. These controllers can communicate by various methods with the available ports, but the one necessary for the class was the CAN protocol that only required the ports CAN High and CAN Low.

Finally our solution must have a controller connected to some interface so that we can modify the variables and a CAN module to communicate with the Deep Sea controller.

II. Introduction to CANBUS

The CAN protocol (Controller Area Network) is a communication bus that was originally developed by BOSCH for the automotive industry to replace complex wiring systems with just two wires. The reduced complexity of the wiring paired with its high immunity to electrical interference, and its ability to self-diagnose and repair data errors have led to the implementation of the CAN protocol in many more industries than just the automotive, including automation and manufacturing.

It operates at data rates of up to 1 Mb/s and can transmit a maximum of 8 bytes per message frame. Identification of nodes is not supported but message identification and priority is, this means that a message that is transmitted through the network will arrive to every single node. Another great advantage of the CAN protocol is that message collision is not experienced. If two receivers send a package at the same time, the one with the highest priority will proceed and the other one will wait until the message is received and then it will be sent again.

The two wires that conform the bus, called CAN high and CAN low can be in two states, dominant and recessive, in the dominant state, there is a voltage differential between the two of approximately 1.5 - 3 V, and when the recessive state is present, the voltage differential goes away. This change of state is the core of CAN communication as it is used to synchronize the nodes and to transmit information in itself, the dominant state being a logical 1 and the recessive state being a logical 0.

III. Introduction to SAE J1939

J1939 is a set of standards defined by SAE built on CAN, it is used mainly on heavy-duty vehicles and its made up of the following layers:

The physical layer, which describes the electrical interference to the bus. The data link layer, which describes the general rules for constructing a message, accesign the bus, and detecting transmission errors and finally, the application layer, which describes the specific data contained within each message sent across the network.

The data transmission rate most of the time is of 250 kb/s or 500 kb/s, and on the contrary of the standard CAN protocol, a specific destination address can be included within the message identifier in order to direct a particular device.

J1939 uses a 29 bit identifier that is structured as it can be seen in Table 1.

TABLE 1
STRUCTURE OF A 29-BIT IDENTIFIER. [3]

Priority	3 bits
Reserved	1 bit
Data Page	1 bit
PDU Format	8 bits
PDU Specific	8 bits
Source Address	8 bits

The priority of the message goes as follows, a value of 0 has the highest priority, and is usually given to time sensitive variables like the torque control message from the transmission to the motor, on the other hand, lower priority messages are not as time critical, for example, the vehicle road speed.

The reserved bit in the identifier should be set to 0 in transmitted messages.

The PDU format determines if the message is broadcasted to the whole network or to a specific device.

The PDU specific or PS interpretation varies according to the PDU format, if the format is between 0 - 239, the PS contains the destination address, if the format is between 240 - 255, the message will be broadcasted and the PS contains a specific group extension.

The term PGN represents the combination of the Reserved, Data Page, PDU Format and PDU Specific values into a single 18 bit package.

IV. SOLUTION DESIGN

For the elaboration of the solution, it was required one Arduino UNO controller, an Arduino CAN module MCP2515, 10 N/O push buttons, 10kΩ resistors, one 10kΩ potentiometer, black and red wire, a 16x2 LCD display, an I2C Arduino module for the display, and it was manufactured a wooden box to hold and protect everything. The power for the device was supplied by one computer to the Arduino UNO with an USB type B to A cable. All these components were required so that it was created an easy to operate and intuitive product.

A. Electric Design

For the electric design, the solution used 10 normally open push buttons connected directly to the digital inputs of the Arduino Uno. Each push button has their respective 10kΩ pulldown resistor to prevent parasite signals and each one of them allows modification of specific parameters. It also used a LCD display with an I2C integrated module. The SCL pin is connected to the analog input 5 and the SDA pin to the analog input 4. The 10kΩ potentiometer is connected to the analog input 0 of the Arduino and the CS, SO, SI and SCK pins of the MCP2515 module are connected to the pin 10, 12, 11 and 13 respectively. The INT pin of the MCP2515 module is not used. The CAN Low and CAN High wires of the MCP2515 module and controller are connected, High with High, Low with Low. Finally, 2 120Ω resistors are connected at each end of the CAN wires to control and synchronize the reception of the signal in every terminal. The magnitude is arbitrary, but the CAN BUS protocol indicates it must be of 120Ω.

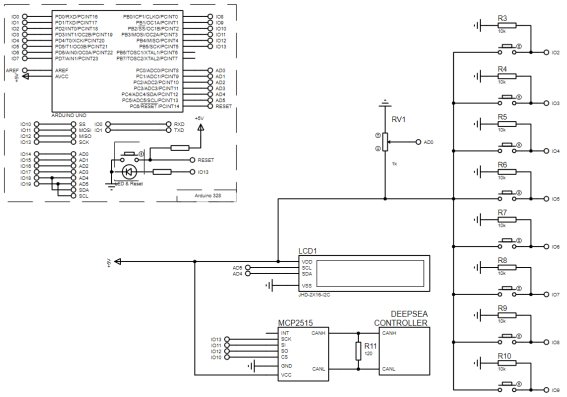

Fig. 1 Electric diagram of the solution.

B. Physical Design

The physical design is meant to be compact and resistant, we made it in the way that the Arduino UNO, the Arduino CAN module and the electronic assembly fit inside the wooden box and also it should be able to hold the 10 push buttons, the LCD display and the potentiometer on the lid.

The box is completely made out of wood and it is assembled with screws, inside the box the arduino and the CAN module are holded with screws and on the lid, the LCD display, as well as the buttons and the potentiometer are pasted with glue. Finally the wooden surface of the box is painted with black spray paint, for a more aesthetic finish. Additionally some labels were pasted to indicate the function of each button (one number on each button and one table with the description in front of the box).

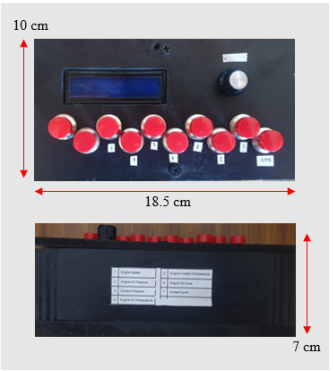

Fig. 2 Illustration with the dimensions of the final device.

V. Methodology for the Solution

Taking into account the requirements of the training partner, the solution was delimited to simulate only certain parameters of an engine. Some of this parameters were further discarded and the final ones were:

- Coolant Level
- Engine Oil Level
- Engine Coolant Temperature
- Engine Oil Temperature
- Coolant Pressure
- Engine Oil Pressure
- Engine Speed

To achieve the modification of all these parameters the first thing to take into account are their PGNs within SAE J1939. If all the parameters above were listed by their PGNs, the list would look like this:

- *PGN 65262:* Engine Coolant Temperature and Engine Oil Temperature.
- *PGN 65263:* Coolant Level, Engine Oil Pressure, Engine Oil Level and Coolant Pressure.
- *PGN 61444:* Engine Speed.

After this, the position of each parameter within the PGNs was found. Table 2 specifies this and also helps identify the amount of bytes each parameter has.

TABLE 2
Position of Parameters within their PGNs

Parameter	Amount of bytes	Byte within PGN	PGN
Engine Coolant Temperature	1	1	65262
Engine Oil Temperature	2	3 & 4	
Coolant Level	1	7	65263
Engine Oil Pressure	1	4	
Engine Oil Level	1	2	
Coolant Pressure	1	6	
Engine Speed	2	4 & 5	61444

With help of an online PGN - CAN ID converter created by CSS Electronics [7], the CAN ID in hexadecimal was found:

- *PGN 65262*: 0xCFEEE00
- *PGN 65263*: 0xCFEEF00
- *PGN 61444*: 0xCF00400

Finally, the transmission rate for each PGN was found:

- *PGN 65262*: 1000ms
- *PGN 65263*: 500ms
- *PGN 61444*: 100ms

VI. Integration of the Methodology into Arduino Code

After obtaining all the specific requirements of each parameter, the integration of the standard J1939 into an Arduino Uno was made with the help of the MCP2515 module. For this, an Arduino library compatible with CAN was used; specifically the one made by Cory Fowler [8].

Later on, in the Arduino IDE, the MCP2515 was initialized running at 8Mhz with a baud rate of 250 kb/s.

In every moment of the operation of the dummy engine, all 3 PGNs are being sent to the controller, this means all parameters are always being sent.

When a button is pressed, the specific parameter for the button can be modified, and a map of the value of a potentiometer determines this value. For the mapping of every parameter, it was necessary to visualize the upper and lower limits of it, as well as the resolution and offset. An example is shown in figure 3.

spn110 - Engine Coolant Temperature - Temperature of liquid found in engine cooling system.

Data Length:	1 byte
Resolution:	1 deg C/bit , -40 deg C offset
Data Range:	-40 to 210 deg C
Type:	Measured
Suspect Parameter Number:	110
Parameter Group Number:	[65262]

Fig. 3. Example of required values for an optimal parameter mapping.

In this case, the lower limit would be 0 and the upper limit would be 250 because the offset is indicated as -40 deg C. In the coding this would look like this:

```
ECTempBasic = map(analogRead(Pot), 0, 1023, 0, 250);
ECTemp = ECTempBasic - 40;
```

Being *ECTempBasic* the mapping and *ECTemp* the real value of the parameter.

In the case of our solution, the parameter in the Arduino IDE that maps the potentiometer, *ECTempBasic*, is the one that is being replaced in the PGN and sent to the controller, and the variable with the real value, *ECTemp*, is the one that is being shown in the LCD display.

In the case of the parameters with 2 bytes (like Engine Speed), the methodology was a little bit different. In the case of the mapping, that was done exactly like a 1 byte parameter, nevertheless, it was necessary to divide the mapped value into the 2 bytes. This was done with the help of an Arduino function called *lowByte* and *highByte* that can automatically divide the variable into 2 bytes. In the coding this would look like this:

```
ESpeedBasic = map(analogRead(Pot), 0, 1023, 0, 64000);
NESpeed1 = lowByte(ESpeedBasic);
NESpeed2 = highByte(ESpeedBasic);
```

After mapping the selected variable, the code modifies the specific byte of the parameter in the PGN. In the coding this would look like this:

```
ECTempBasic = map(analogRead(Pot), 0, 1023, 0, 250);
NECTemp = ECTempBasic;
PGN_65262[0] = NECTemp;
```

And finally, as said before, every PGN is sent to the controller through its specific direction and with the value of the parameter modified. In Arduino IDE, it is needed to specify if it is Standard CAN (0) or Extended CAN (1), and as the solution is using Extended CAN, then it is required to write "1" when sending the message. Also, it is required to specify the length of the message, and as the solution is always sending PGNs (8 bytes of information), it is required to write "8" when sending the message. The solution is also taking into account the transmission rate of each PGN, so they are being sent at different times. In the coding this would look like this:

```
if (millis() >= TiempoPGN_65262 + 1000) {
  sndStat = CAN0.sendMsgBuf(0xCFEEE00, 1, 8, PGN_65262);
  TiempoPGN_65262 += 1000;
}
if (millis() >= TiempoPGN_65263 + 500) {
  sndStat = CAN0.sendMsgBuf(0xCFEEF00, 1, 8, PGN_65263);
  TiempoPGN_65263 += 500;
}
if (millis() >= TiempoPGN_61444 + 100) {
  sndStat = CAN0.sendMsgBuf(0xCF00400, 1, 8, PGN_61444);
  TiempoPGN_61444 += 100;
}
```

The whole code can be found in the appendix 1.

VII. RESULTS

Fig. 4 Top view of the final device.

Fig. 5 Front view of the final device.

Due to the limitations of time and resources only 7 variables out of the total 10 were used, however, the device was able to send the information to the Deep Sea controller and keep it stored, so the 7 variables are capable of sending information and keeping it saved at the same time, which is quite useful since it is possible to monitor different signals within the controller without the disadvantage that when the variable is changed the information disappears.

The proposed solution has solid advantages, it is relatively compact and with the help of the labels and different buttons it's easy and intuitive to use. The display helps visualize the information being modified and its value, so it is really easy to corroborate the correct functioning of the dummy motor. But, even though it can transmit the information in a satisfying way it can still be improved upon. As mentioned before, a lack of resources in the Arduino UNO, specifically, digital inputs, prevented the implementation of more parameters so an expansion in the number of inputs in the microcontroller would solve that issue.

Furthermore, replacing the buttons with a couple of 10 position rotary switches can make the design more user friendly, by reducing the apparent number of inputs but also increasing the possible inputs.

And finally, for the product to reach a greater audience, the implementation of language selection could also be done.

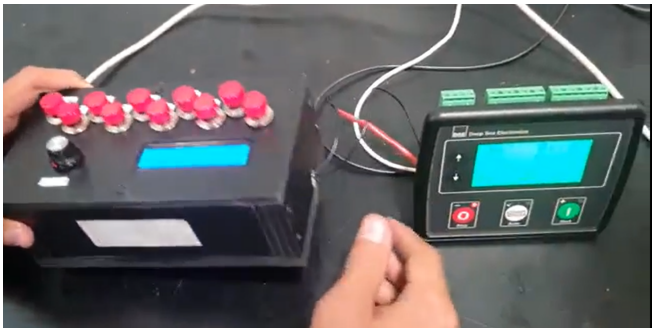

Fig. 6 Illustration of the final device connected to the controller.

For a more visual representation of the functioning of the device, a video can be found in appendix 2.

VIII. Conclusions

The programming to realize the CAN BUS SAE J1939 protocol was challenging because a profound investigation had to be conducted to know how to make the code, for example, how to determine the bit to use in each SPN to send data as well as how to use the PGN (Parameter Group Number) in the code and more information that was needed to correctly communicate the dummy engine with the controller.

Also, an understanding of the CAN communication had to be implemented in the solution so the information was sent in the way it was intended to.

The objective was achieved but the solution can be further improved upon as talked about in the previous section.

IX. References

[1] Corrigan S. (2016) *Introduction to the Controller Area Network (CAN)*. [Online]. Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1655478995114&ref_url=https%253A%252F%252Fwww.google.com%252F

[2] W. Voss. (2014) *Controller Area Network (CAN) Prototyping with Arduino*. [Online]. Available: http://www.prometec.net/wp-content/uploads/2015/07/Controller-Area-Network-Prototyping-With-Arduino-Wilfried-Voss.pdf

[3] (2022) *J1939 Introduction*. [Online]. Available: https://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction/

[4] (2018) *SAE J1939 Programming with Arduino - ARD1939 Sample Application*. [Online]. Available: https://copperhilltech.com/blog/sae-j1939-programming-with-arduino-ard1939-sample-application/

[5] W. Voss. (2018) *Guide To SAE J1939 - Parameter Group Numbers (PGN)*. [Online]. Available: https://copperhilltech.com/blog/guide-to-sae-j1939-parameter-group-numbers-pgn/

[6] W. Voss. (2018) *SAE J1939 Programming with Arduino - Suspect Parameter Numbers (SPN)*. [Online] Available: https://copperhilltech.com/blog/sae-j1939-programming-with-arduino-suspect-parameter-numbers-spn/

[7] (N. D.) *CAN ID to PGN Converter.* [Online]. Available: https://docs.google.com/spreadsheets/d/10f7-TFU9oViSQZYGFYVPDia2w1hd5eOPMlgJXmx31Lg/edit#gid=1130918092

[8] (2021) *MCP_CAN_lib*. [Online]. Available: https://github.com/coryjfowler/MCP_CAN_lib

X. Appendix

[1] Complete Arduino Code
[2] Video of the operating device (Time stamp: 3:38)