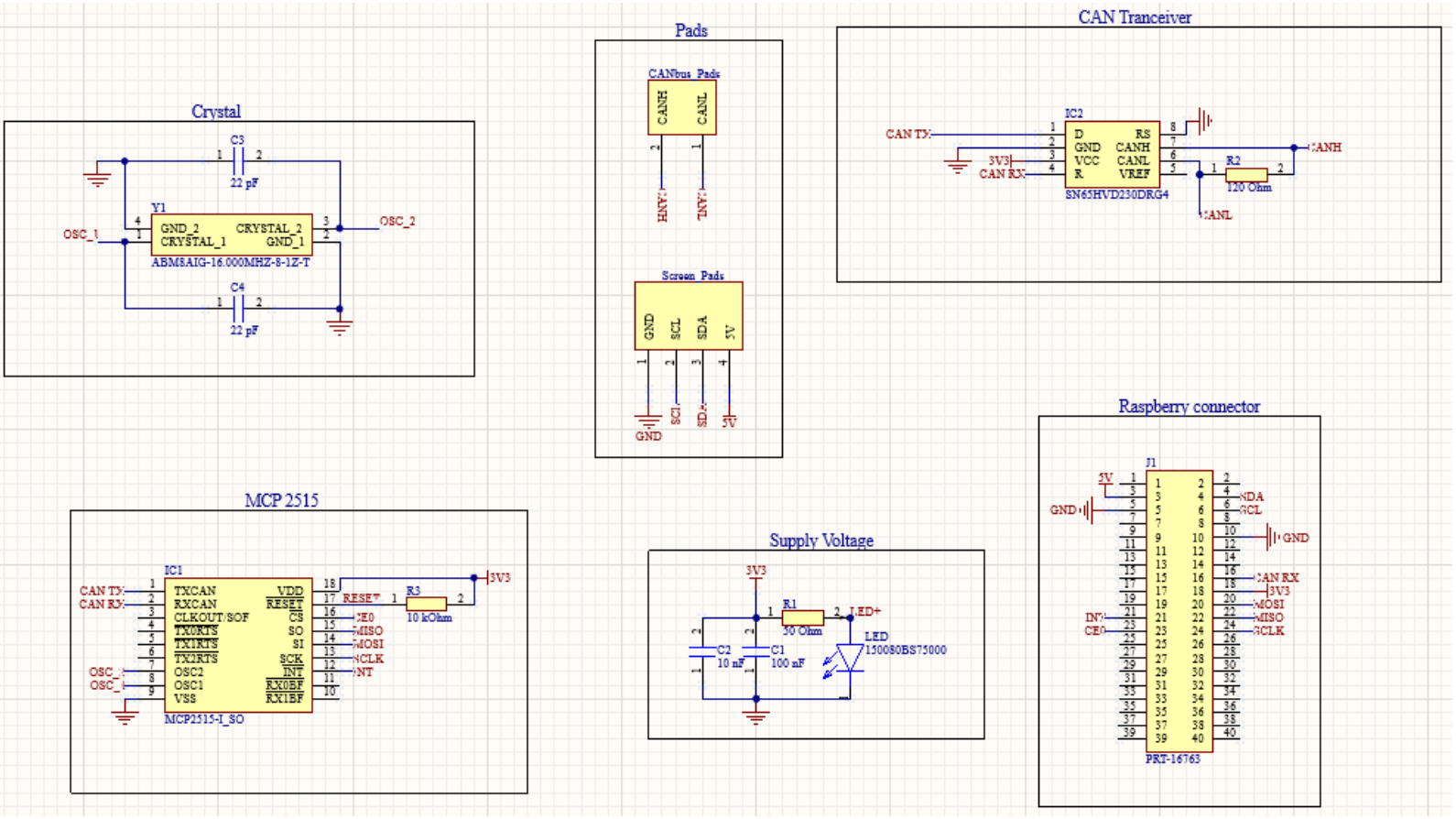
CAN_Diagnostics App

Hardware:

The device consists of a Raspberry Pi 3, a touchscreen and a custom designed PCB. The PCB is a CAN shield and transforms the CAN protocol to SPI through which it can communicate with the RPi (Raspberry Pi has no ability to communicate directly with CAN bus). CAN High and CAN Low wires are soldered on specific pads on the board and utilizing the IC SN65HVD230DRG4, the differential signal of CAN bus is transformed to digital from 0 to 3V3 (the supply of the board is 3V3). Then MCP2515-I_SO with a 16MHz crystal sends the final message through SPI to the RPi, where the software takes over. The board has 4 additional pads where jumper wires are soldered to connect the pins 5V, GND, SDA, SCL of the RPi to the touchscreen (because the PCB connector covers all of the RPi pins). The schematic is shown below:



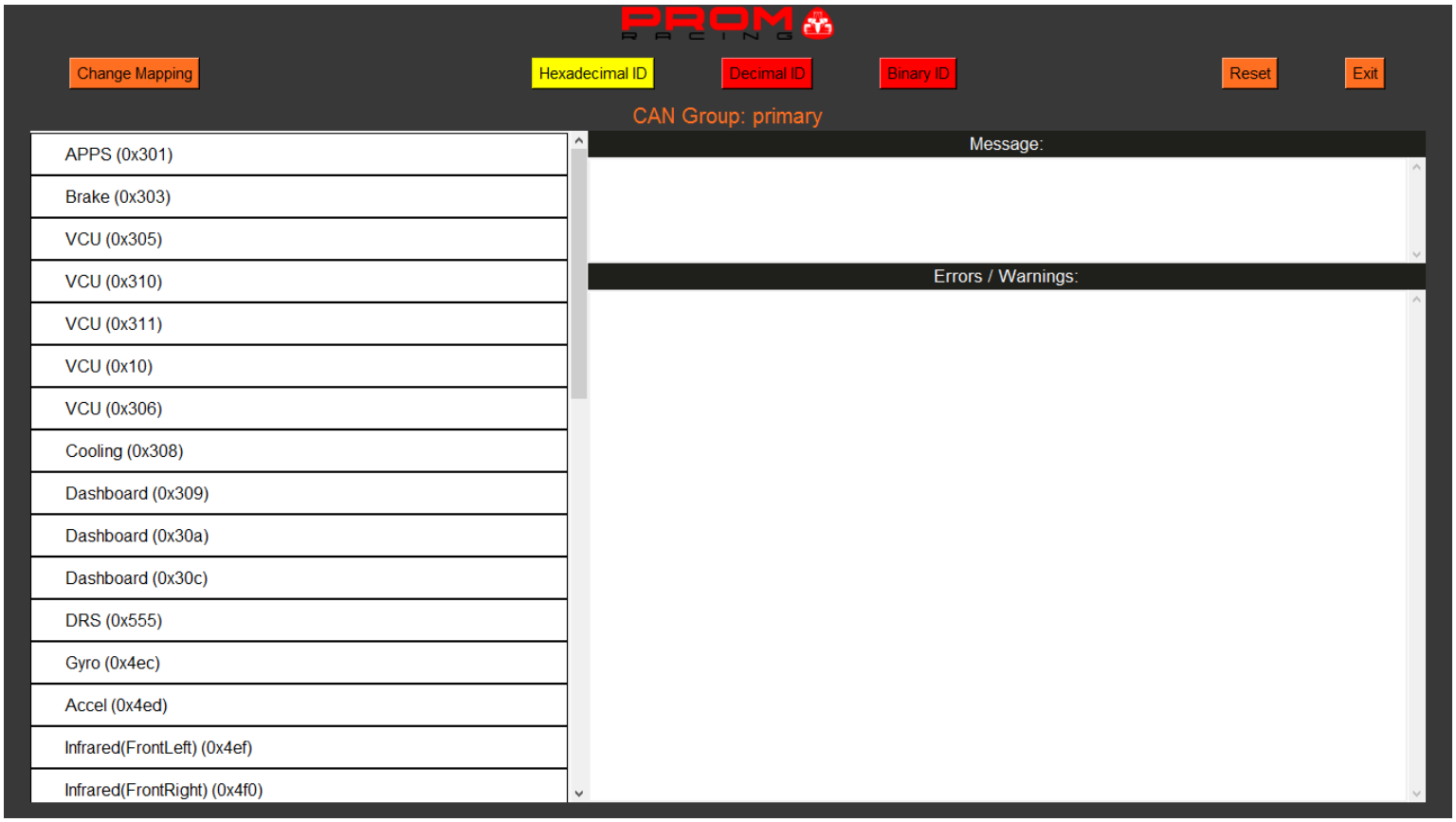
Software:

As far as the software is concerned, an object-oriented approach has been followed using the programming language Python. The app is able to read all the data in CAN bus, categorize them according to their ID, display them in a way the user can read the actual message and not just numbers, and finally find and display errors in transmission or in the messages themselves. The app can find the following errors:

- Nodes sending messages with a delay or not sending at all
- Nodes sending messages while they shouldn't (either their ID is not mapped or they belong to a different CAN group, e.g. primary while all other messages are from sensory)
- Invalid message format (wrong byte size, values out of bounds)
- Error in any check that the programmer does on a specific node

The app gives the ability to run general checks on all nodes (like making sure that they have no delay transmitting a message), but also check each sensor independently (since the format of the messages vary).

In order to run the CAN Diagnostics app, the user must run the following line on the terminal to establish the CAN bus communication at speed 1 Mbps:
sudo ip link set can0 up type can bitrate 1000000
Then, the app starts if the user runs the Python code of *main_app.py*.
Image of the app when it starts:



When *main_app.py* runs, it imports all other files needed for the execution, namely:

top.py
can_reader.py
id_panel.py
data_panel.py
error_panel.py
scrollable_frame.py
general_programmer_class.py
mapping.py
options_panel.py

CAN_Node_Handler_Classes	folder containing node handler classes (e.g. APPS, Brake etc)
mapping_txt.txt	txt file that defines the parameters and IDs of all the sensors
prom_racing_logo_resized.png	picture of our logo appearing on top of the screen

- ➔ *top.py* file is responsible for the appearance of the app on top of the screen (including the team's logo and name)

- ➔ *can_reader.py* creates an object for every message that arrives from CAN bus and contains its ID and a list of integers for every byte of data. Whenever a message arrives, it is sent to *main_app.py* for processing.

- ➔ *id_panel.py* displays all the IDs included in the mapping file as well as the ones that are transmitting messages but are not found in the mapping file (in this case, an error is generated but the ID and its data are displayed normally). The ID can be shown in hexadecimal, decimal or binary form and on the left of the ID there is the name of the node.

- ➔ *data_panel.py* is the panel under the label "Message: " and displays the last message of the sensor chosen from the ID panel. If no sensor has been chosen, the data panel is empty. Each message is followed by a timestamp.

➔ *error_panel.py* is the panel under the label “Errors / Warnings: ” and displays the last n errors / warnings of the sensor chosen from the ID panel. If no sensor has been chosen, the error panel is empty. When a node has at least one error / warning, its color turns from white to red in the ID panel. In the error panel, the errors are red and the warnings orange, and new errors / warnings are stacked on top. Errors are mostly used when something has gone wrong in the transmission of the message or in case a sensor was stopped working (eg no messages or messages taking too long to arrive), while warnings are used when the bus is working properly but the value of a byte in a message indicates a problem in the car (e.g. battery voltage is under minimum).

➔ *scrollable_frame.py* provides scrollbars for the ID, data and error panels

➔ *general_programmer_class.py* is the base class of all classes dealing with the messages of the nodes. This class and all other classes derived from it, have as input the data (an array of integers) and give as output a readable message. During this process they can create errors and warnings. Gereral_programmer_class gives out a message identical to the data received (since it is not made for a specific node) and does basic error checking that has to do with time delay.

➔ *mapping.py* is called when the app is executed. It has access to the text file *mapping_txt.txt* where it reads line by line all sensors and their properties. Firstly, it reads the id (in hexadecimal form), and then the name of the sensor (they are separated by “:”). Then, it reads the time delay (maximum time interval between 2 consecutive messages). If a message takes more time to arrive than the time delay, an error is generated in the respective ID. If a sensor doesn’t send messages periodically (like DRS), time delay is followed by a dash “-” and in that case, no time checking occurs. Finally, it reads the depth (integer indicating the n last errors / warnings to be stored) and the CAN group (primary, sensory, motor etc). All parameters are separated by comma, even at the end of line. Hashes “#” can be used to make comments, so the mapping class ignores whatever follows. After a change in mapping, the user can either click on “Exit” where no changes will be saved or “Apply”. In this case, a syntax

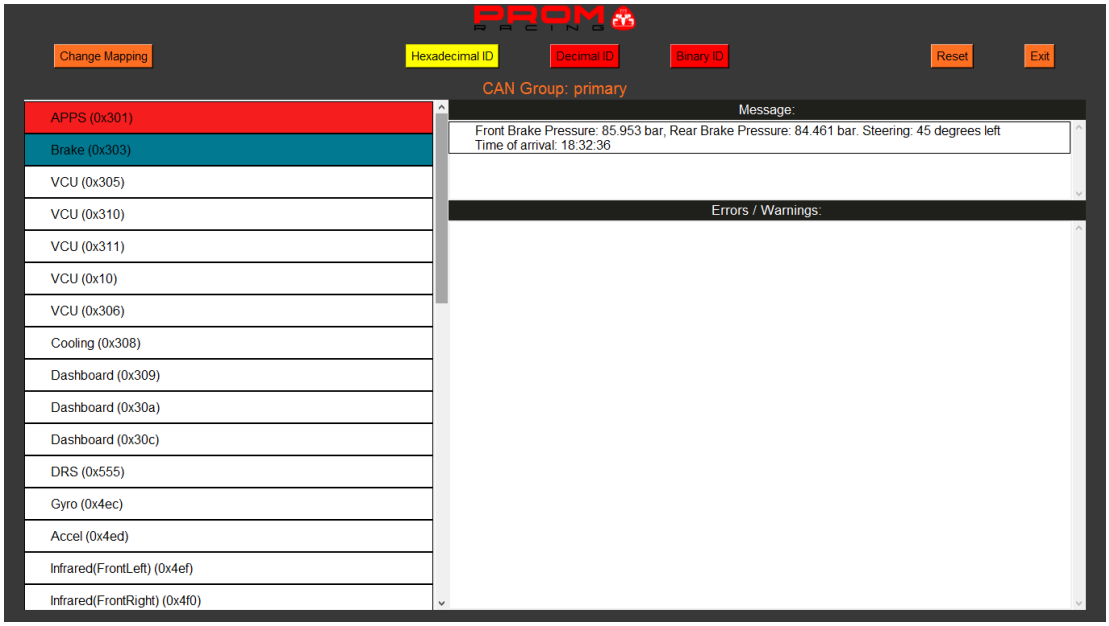
check is done and the changes cannot be saved unless no syntax errors are present.

➔ *options_panel.py* creates the buttons below the logo. “Change Mapping” option gives access to the mapping file, “Hexadecimal ID”, “Decimal ID” and “Binary ID” change the number system of the IDs displayed in the panel, “Reset” deletes all messages and errors / warnings from all sensors and turns their color back to white, and “Exit” exits fullscreen. After exit, the button is named “Fullscreen”. Esc button from keyboard has the same function of exiting and entering fullscreen.

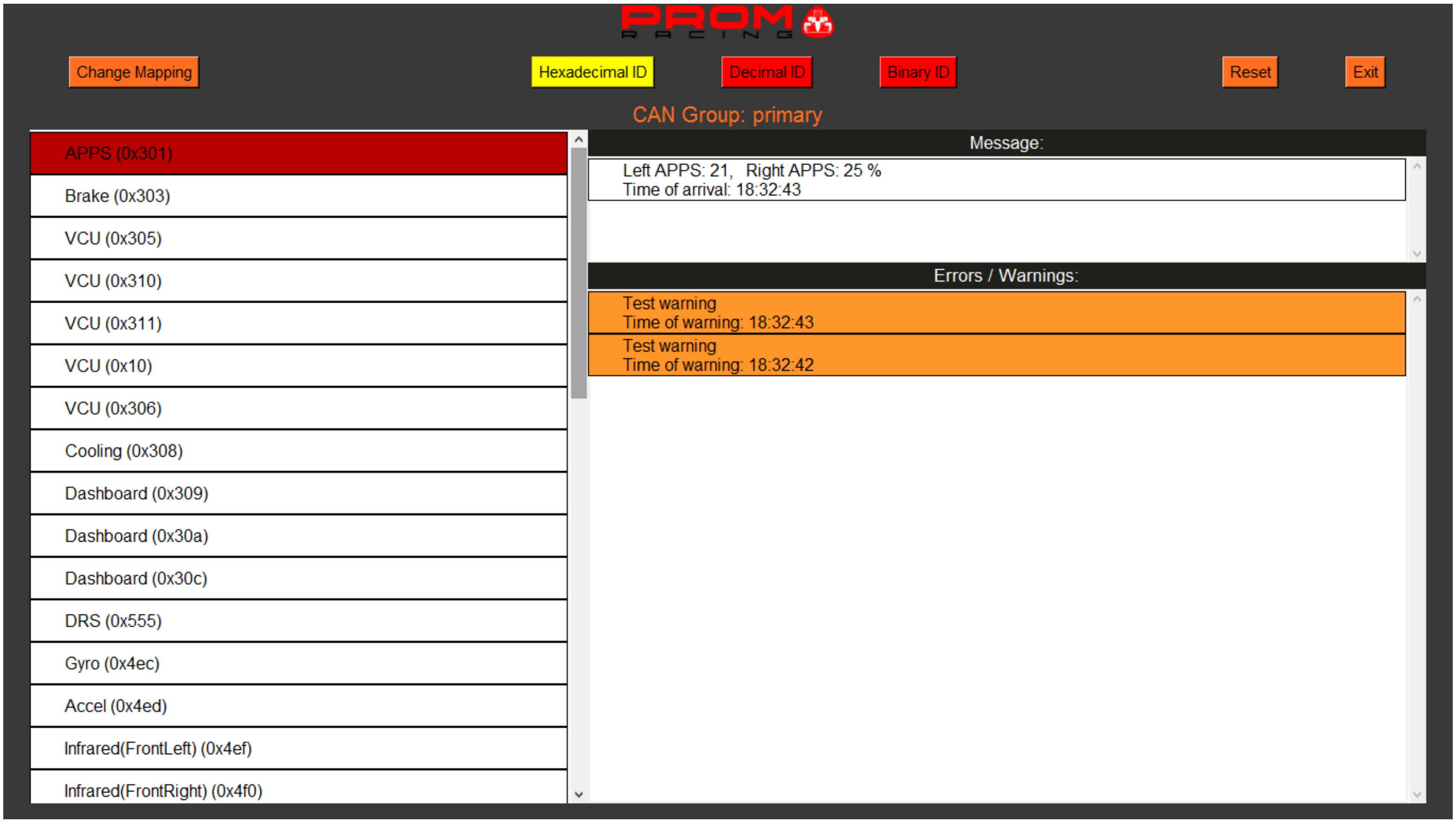
➔ *CAN_Node_Handler_Classes* folder contains all the classes made specifically for a node (e.g. Gyro, DRS, Infrared FR). These classes make a more thorough check, since the messages they process have a specific length and format. They check whether the data has the correct size, expected values, or whatever the programmer wishes to implement. Time checking is done by the base class. When a new sensor is added, the programmer should add it to the mapping text (including its parameters), create a class especially for it and place it in the folder, and update *main_app.py* so as to send the messages of the new ID to this class.

Under the buttons, the app displays the CAN Group where it is connected. This is defined by the CAN group of the sensor of the first message read by the program.

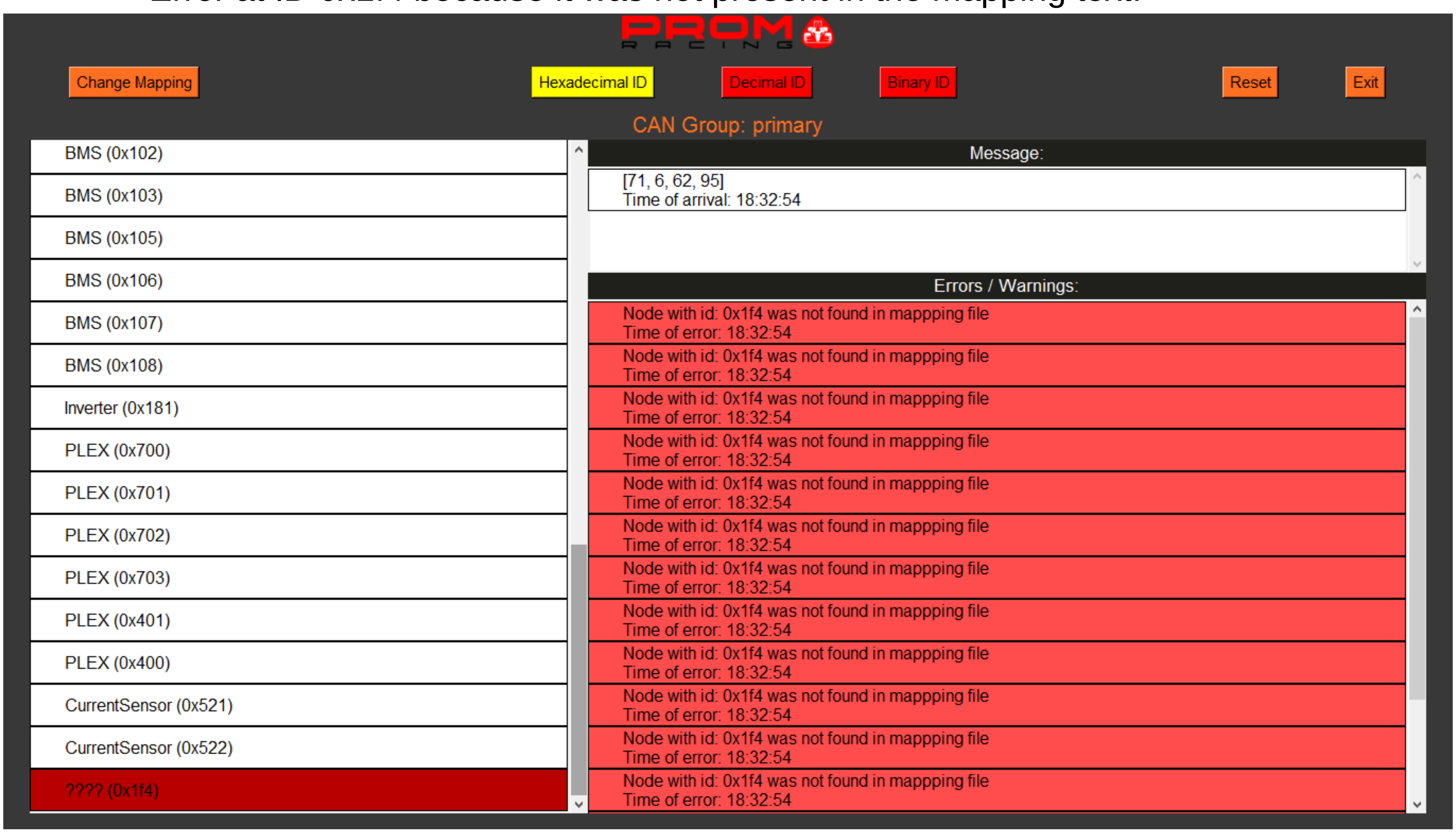
Below, screenshots from the app are shown:
APPS has an error, and Brake Sensor is chosen:



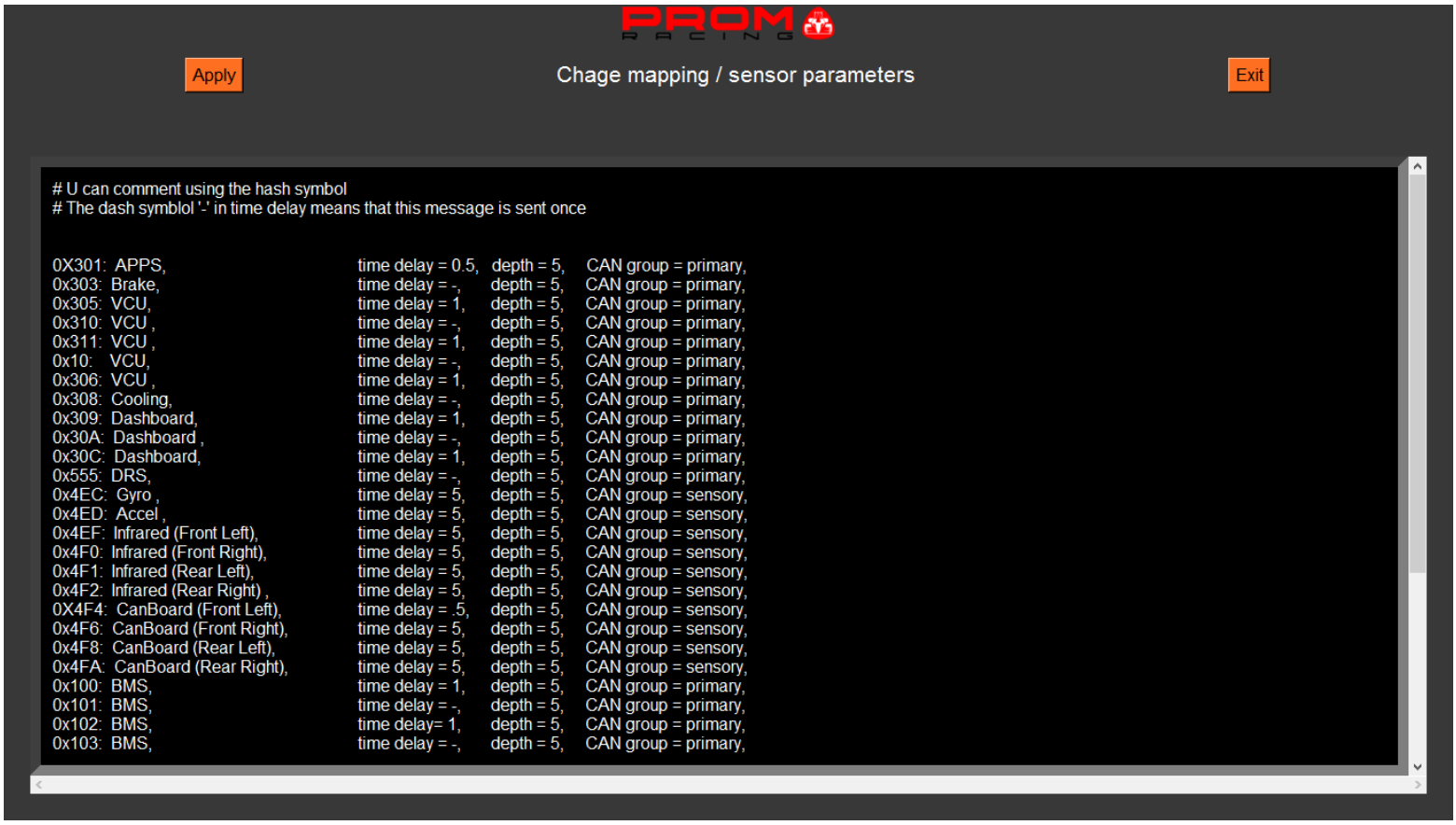
Warnings in APPS:



Error at ID 0x1f4 because it was not present in the mapping text:



Appearance of mapping text:



Appearance of mapping text when a syntax error has been made and “Apply” button has been clicked: