1 Introduction

The CAN bus permits commucation between other important devices, like the Raspberry Pi (used for tracking) and Windows (used for robot simulation, developed in Unreal Engine). However, some devices used in the robot don't support CAN, like some controllers (Spark and VictorSP) and most sensores (NavX or limit switches, for example). That's why we developed our own CAN bus protocol, by RoboRIO's Java programming.

2 Objective

The objective of this document is to present a scientific background behind Brazilian Storm #6404's CAN bus protocol development and explain it for use in external devices and prevenctive maintence.

3 Data Structure

Each data receives informations from ID, byte position (0 to 7) and bit. The bits can inform the absolute value (for sensors with values between 0 to 255), ternary state (in the case of controllers that are slowing down or accelerating) and boolean state (in the case of boolean sensors, like limit switches).

The data is separated into types. The main types of data are data coming from sensors and controllers. They have a shared ID bases: for example, every sensor starts with ID "1F 64" and every controller starts with the ID "2F 64". The structure for shared IDs can be found in the beginning of each section.

Alguns sensores serão virtualizados para simulação e para isso, precisam receber inputs. Há uma programação separada para uso nos simuladores do time e ela pode ser encontrada no GitHub. De acordo com as normas de controle e configuração, a programação para simulação estará dentro do repositório do robô simulado (BS-03 ou BS-04), sob a branch "simulation".

Some sensores are virtualized for simulation and for that, they need to receive inputs. There's a separated codebae for simulation use in the team's robots and it can be found in GitHub. According to the control and configuration norms, the codebase can be found inside the repository of the robot you want to simulate (BS-03 or BS-04), under the "simulation" branch.

4 Use of CAN Data

4.1 Raspberry Pi

The Raspberry Pi (RPi, for short) is used to send tracking information (angle and distance from the robot to the vision target). In 2019, these informations were sent by the Shuffleboard (Ethernet connection with the radio). However, the FMS (Field Management System) used by FIRST during the competition refused the RPi connection with the RoboRIO.

Figure 1 – Raspberry Pi 3 Model B, used for tracking

The FMS doesn't have the right to refuse a CAN connection, because the CAN bus works in a local network. That's why the CAN bus use is essential for ensuring the data reliabilty between the RPi and the RoboRIO.

O RPi não possui funcionalidade CAN por padrão. Por isso, é usada a placa MCP2515, que implementa a funcionalidade de interface CAN para qualquer dispositivo que tenha comunicação SPI.

The RPi doesn't have CAN bus functionality by default. That's why we use the MCP2515 board, that implements the CAN interface to any board that has SPI communication.

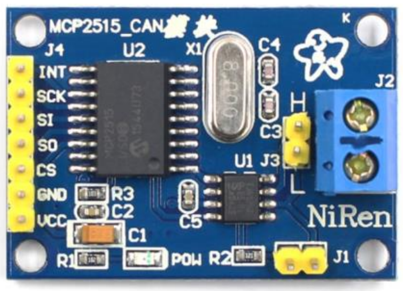

Figure 2 – MCP2515

4.2 Arduino

The Arduino is used to store the information sent by the robot during the match. With that, it's possible to use the information sent to create graphs of acceleration, speed, robot movement and many other data that can be modelled by the robot information: battery voltage, the current that each motor draws, force applied in each controller, gyroscope rotations, encoder pulses and much more.

The Arduino's task is to receive: he receives everything the robot sends, in every possible ID. For that, we need to use a CAN controller (MCP2515) and connect it with the RoboRIO. After receiving the data, he stores it in a ".txt" file in a separate SD card. He's also able to communicate with the RoboRIO by Ethernet, with the W5100 shield.

Figure 3 – Shield Arduino W5100

Há um sistema separado que lê esse arquivo ".txt" e transforma isso em gráficos sobre o robô, como aceleração, velocidade média, posicionamento na quadra em função do tempo, quantidade de pontos marcados, dentre outros dados importantes. Ele roda em Windows, é programado em C# e é executado com base no arquivo gerado em cada partida.

There's a separated system that reads that ".txt" file and transforms that in robot graphs, like acceleration, positioning in the field (related to time) and other important data. It runs in Windows, is programmed in C# and considers each match file.

4.2 Simulator

The simulator is a system developed in Unreal Engine 4 that reads the CAN bus data to register every important data (drvetrain, climbing and other subsystems) and transforms that in robot movement inside the virtual field. It's able to test autonomous programming and every functionality used in the robot can be ran inside it.

To communicate the RoboRIO with the simulator, we need to communicate the RoboRIO with the computer. That communication is made with a device that transforms CAN data to Serial, like the PCAN-USB. To read the IDs and send the data to the Unreal Engine, we developed a way to receive serial data from UE4. The UE4 doesn't receive serial data by default.

5 Sensors

Device ID: 30

Java ID: 1F 64

5.1 Digital Sensors

Java ID: 1F 64 04

5.1.2 Limit Switches (Claw and Tower)

Java ID: 1F 64 04 FF

5.1.2.3 Tower Limit Switch - Superior

Java ID: 1F 64 04 FF

Byte: 0

Bit: (byte) 1

Type: Output

5.1.2.4 Tower Limit Switch - Inferior

Java ID: 1F 64 04 FF

Byte: 0

Bit: (byte) 2

Type: Output

5.1.2.6 Claw Limit Switch - Superior

Java ID: 1F 64 04 FF

Byte: 1

Bit: (byte) 2

Type: Output

5.1.2.7 Claw Limit Switch - Inferior

Java ID: 1F 64 04 FF

Byte: 1

Bit: (byte) 3

Type: Output

5.2 Analog Sensors

Java ID: 2F 64 04 DD

5.2.1 Drivetrain Encoders

Java ID: 2F 64 04 DD

5.2.1.1 Encoder absoolute value

Java ID: 2F 64 04 DD

Byte: 0

Bit: Absolute encoder value, from 0 to 255, in centimeters

Type: variable (input for the simulator codebase, output for the logging codebase)

5.2.1.2 Encoder direction

Java ID: 2F 64 04 DD

Byte: 1

Bit: 1 for positive and anything else for negative

Type: variable (input for the simulator codebase, output for the logging codebase)

5.2.2 Virtual NavX

Java ID: 2F 64 04 DD

5.2.2.1 NavX absolute value

Java ID: 2F 64 04 DD

Byte: 2

Bit: 0 to 180

Type: variable (input for the simulator codebase, output for the logging codebase)

5.2.2.2 NavX Direction

Java ID: 2F 64 04 DD

Byte: 3

Bit: 1 for positive and anything else for negative

Type: variable (input for the simulator codebase, output for the logging codebase)

6 Controllers

Device ID: 31

Java ID: 2F 64

6.1 Subsystem Controllers

Java ID: 2F 64 04

6.1.1 Tower Controllers

Java ID: 2F 64 04 FF

6.1.1.1 Tower Speed

Java ID: 2F 64 04 FF

Byte: 0

Bit: Speed, from 0% to 100% (decimal)

Type: Output

6.1.2 Claw Controlers

Java ID: 2F 64 04 AA

6.1.2.1 Cargo Controller - Enabled

Java ID: 2F 64 04 AA

Byte: 0

Bit: (decimal) 1

Type: Output

6.1.2.2 Cargo Controller - Disabled

Java ID: 2F 64 04 AA

Byte: 0

Bit: (decimal) 2

Type: Output

6.1.2.1 Claw Mode Controller - Moving Up

Java ID: 2F 64 04 AA

Byte: 1

Bit: (decimal) 1

Type: Output

6.1.2.1 Claw Mode Controller - Disabled

Java ID: 2F 64 04 AA

Byte: 1

Bit: (decimal) 2

Type: Output

6.1.2.1 Claw Mode Controller – Moving Down

Java ID: 2F 64 04 AA

Byte: 1

Bit: (decimal) 3

Type: Output

6.1.3 Drivetrain Controllers

Java ID: 2F 64 04 BB

6.1.3.1 Speed Controller Group - Left

Java ID: 2F 64 04 BB

Byte: 0

Bit: Speed, from 0 to 100 (in decimal)

Type: Output

6.1.3.2 Speed Controller Group - Right

Java ID: 2F 64 04 BB

Byte: 1

Bit: Speed, from 0 to 100 (in decimal)

Type: Output

6.1.3.3 Speed Controller Group – Left

Java ID: 2F 64 04 BB

Byte: 2

Bit: Speed, from 0 to 100 (in decimal)

Type: Output