



**UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA
FILHO” FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA**

TRABALHO DE GRADUAÇÃO

PROJETO DE UM SISTEMA DE TELEMETRIA PARA VEÍCULOS FÓRMULA SAE UTILIZANDO PROTOCOLO CAN E WI-FI

Aluno: João Victor Franco Fernandes

Orientador: Prof. Dr. Carlos Antonio Alves

Ilha Solteira
2022

JOÃO VICTOR FRANCO FERNANDES

**PROJETO DE UM SISTEMA DE TELEMETRIA PARA VEÍCULOS FÓRMULA SAE
UTILIZANDO PROTOCOLO CAN E WI-FI**

Trabalho de graduação apresentado à
Faculdade de Engenharia do Campus de
Ilha Solteira – UNESP como parte dos
requisitos para obtenção do grau de
engenheiro eletricista.

Orientador: **Prof. Dr. Carlos Antonio Alves**

Ilha Solteira

2022

JOÃO VICTOR FRANCO FERNANDES

**PROJETO DE UM SISTEMA DE TELEMETRIA PARA VEÍCULOS FÓRMULA SAE
UTILIZANDO PROTOCOLO CAN E WI-FI**

Trabalho de graduação apresentado à
Faculdade de Engenharia do Campus de
Ilha Solteira – UNESP como parte dos
requisitos para obtenção do grau de
engenheiro eletricista.

Orientador: **Prof. Dr. Carlos Antonio Alves**

BANCA EXAMINADORA

Orientador: Prof. Dr. Carlos Antonio Alves

Departamento de Engenharia Elétrica
Faculdade de Engenharia de Ilha Solteira
UNESP

Prof. Dr.

Departamento XX
Faculdade de Engenharia de Ilha Solteira
UNESP

Prof. Dr.

Departamento XX
Faculdade de Engenharia de Ilha Solteira
UNESP

AGRADECIMENTOS

Agradeço a toda minha família, em especial ao meu pai Luis Carlos Fernandes, à minha mãe Sheila Regina Pfeifer Franco Fernandes e a minha irmã Mariana Franco Fernandes, por todo apoio durante minha jornada acadêmica.

Aos meus grandes amigos da República B.O. cum Fumo, pelo acolhimento e parceria durante todos esses anos de graduação.

A todos da equipe Fênix Racing FSAE – em especial aos colegas da eletrônica, que me auxiliaram enormemente no desenvolvimento deste trabalho.

Aos meus companheiros de sala de aula, em especial ao Evandro, Hugo, João Pedro, Paulo, Pedro, Theo, Thiago e Vinícius, pelo apoio nas mais diversas noites em claro estudando e nas boas risadas, inclusive nos momentos difíceis.

À meus amigos Filipe, Gabriela, Gustavo e Pedro, que sem o suporte e companheirismo, não atingiria metade das conquistas que tenho hoje, saibam que poderão contar para sempre comigo, assim como sei que poderei contar para sempre com vocês.

À todos os professores que fizeram parte da minha educação e aqueles que ainda o farão, - em especial ao meu orientador, Prof. Dr. Carlos Antonio Alves, pela confiança depositada em minha pessoa.

À Deus, por permitir tudo o que vivi até aqui.

RESUMO

Neste trabalho de Graduação promoveu-se o desenvolvimento de um sistema de telemetria voltado a veículos de categoria fórmula SAE, com o intuito de permitir o acesso em tempo real a aquisição de dados desenvolvida pelo time de eletrônica do grupo de extensão Fênix Racing da UNESP de Ilha Solteira. O projeto utiliza duas placas microcontroladoras – ESP32 e Arduino UNO - para captar dados gerados pelos mais diversos sensores de velocidade, aceleração e temperatura através do protocolo CAN (*Controller Area Network*). Após a captação de dados, o ESP32 é responsável por enviá-los via conexão Wi-Fi 3G de um celular à um servidor *No-SQL Firebase* para que então seja possível a leitura e análise em tempo real dos dados, permitindo um *feedback* mais rápido para ações voltadas à segurança do piloto e a otimização do projeto veicular, e a expansão do projeto teórico com uma maior quantidade de informações a serem analisadas e estudadas pelo time de desenvolvimento. A máxima taxa de transferência obtida foi de 35 Hz, porém, com uma média de 34 Hz devido a limitações de velocidade das bibliotecas utilizadas. A taxa de atualização mostra que o sistema é parcialmente funcional, considerando os sensores de temperatura, GPS, velocidade, RPM e curso da suspensão. Ao final do projeto são sugeridas recomendações para otimização do sistema de transferência de dados.

Palavras-chave: Aquisição de dados, ESP32, Arduino, Processamento de sinais

ABSTRACT

This undergraduate thesis promoted the development of a telemetry system aimed at vehicles of the Formula SAE category, in order to allow real-time access to the data acquisition developed by the electronics team of the Fenix Racing extension group at UNESP Ilha Solteira. The project uses two microcontroller boards - ESP32 and Arduino UNO - to capture data generated by the speed, acceleration and temperature sensors, through the CAN (Controller Area Network) protocol. After capturing data, ESP32 is responsible for sending them via a 3G Wi-Fi connection from a smartphone to a No-SQL Firebase server, so that it is possible to read and analyze the data in real time, allowing faster feedback for actions aimed at pilot safety, vehicle design optimization and expansion of the theoretical design with a greater amount of information to be analyzed and studied by the development team. The maximum transfer rate obtained was 35 Hz, however, with an average of 34 Hz due to speed limitations of the libraries used. The refresh rate shows that the system is partially functional, considering the sensors for temperature, GPS, speed, RPM and suspension travel. At the end of the project, recommendations are suggested for optimizing the data transfer system.

Palavras-chave: Data acquisition, ESP32, Arduino, Signal processing

LISTA DE ABREVIações

CAN – Controller Area Network

CRC – Cyclic Redundancy Check

CS – Chip Select

CSS – Cascading Style Sheets

ECU – Electronic Control Unit

FX9 – Fênix 9 – Veículo desenvolvido no ano de 2019 pela equipe Fênix Racing

FX – Fênix X ou Fênix 10 – Décimo veículo desenvolvido pela equipe Fênix Racing

GND – Ground

HTML – Hypertext Markup Language

I/O – Input/Output

I2C – Inter-Integrated Circuit

IDE – Integrated Development Environment

JSON – JavaScript Object Notation

MAP – Manifold Absolute Pressure

MISO – Master Input Slave Output

MOSI – Master Output Slave Input

SAE – Society of Automotive Engineers

SCK – Serial Clock

SD Card – Secure Digital Card

SPI – Serial Peripheral Interface

TPS - Throttle Position Sensor

UART – Universal asynchronous receiver/transmitter

USB – Universal Serial Bus

VCC – Voltage Common Collector

LISTA DE FIGURAS

Figura 1 - Equipe Fênix Racing FSAE na competição SAE 2019.....	12
Figura 2 - Componentes principais de um sistema de aquisição de dados de um veículo de corrida.....	14
Figura 3 - Interface do <i>MegaLogViewer</i>	17
Figura 4 - Barramento físico do protocolo CAN.....	18
Figura 5 - Protocolo de mensagens CAN.....	18
Figura 6 - Ambiente de desenvolvimento integrado Arduino.....	19
Figura 7 - Arduino UNO.....	20
Figura 8 - Placa de desenvolvimento ESP-32.....	21
Figura 9 - Injeção eletrônica MegaSquirt MS3/MSX instalada no veículo FX9.....	22
Figura 10 - <i>Software</i> TunerStudio.....	23
Figura 11 - Interface <i>Realtime Database</i>	24
Figura 12 - Exemplos de gráficos através da biblioteca ApexCharts.js.....	25
Figura 13 - Página do GitHub para upload do <i>app</i>	26
Figura 14 - Interfaces do projeto de telemetria.....	27
Figura 15 - Módulo MCP2515 TJA1050.....	30
Figura 16 - Página de visualização em tempo real dos dados coletados.....	32
Figura 17 - Página de visualização em tempo real, compactada.....	32
Figura 18 - Implementação de recursos visuais para a criação dos gráficos.....	33
Figura 19 - Página de análise de informações.....	33
Figura 20 - Página para seleção e <i>download</i> das informações no banco de dados....	34
Figura 21 - Variação dos dados gravados pela injeção eletrônica.....	38
Figura 22 - Dados armazenados no servidor <i>firebase</i>	39
Figura 23 - Visualização em tempo real dos dados no servidor:.....	39
Figura 24 - Visualização completa dos dados no servidor.....	40
Figura 25 – Comparativo entre os eixos da primeira e da segunda versão.....	42
Figura 26 – Esquemático do projeto.....	55

LISTA DE TABELAS

Tabela 1 – Taxa de atualização mínima e recomendada para sinais

Tabela 2 – Lista de instrumentos a serem enviados para o ESP-32

Tabela 3 – Velocidade (*step*) e frequência de transmissão de um único dado entre os microcontroladores

Tabela 4 – Velocidade (*step*) e frequência de transmissão de pacotes entre os microcontroladores

SUMÁRIO

1. INTRODUÇÃO.....	11
1.1. Fórmula SAE	11
1.2. Aquisição de dados em um carro de corrida	12
1.3. Telemetria	15
1.4. Protocolo CAN.....	17
1.5. MICROCONTROLADOR ARDUINO	19
1.6. MICROCONTROLADOR ESP-32	20
1.7. INJEÇÃO ELETRÔNICA MEGASQUIRT MS3/MSX.....	21
1.8. BASE DE DADOS EM TEMPO REAL	23
1.9. Front-end Interface	25
2. IMPLEMENTAÇÃO DO SISTEMA DE TELEMETRIA	27
2.1. Interface veicular	28
2.2. Interface <i>Back-end</i>	30
2.3. Interface Front-end	31
2.4. Testes.....	34
3. SIMULAÇÃO DO SISTEMA.....	36
4. CONCLUSÕES.....	43
5. REFERÊNCIAS BIBLIOGRÁFICAS	45
6. ANEXO I – Código Arduino.....	47
7. ANEXO II	51
8. ANEXO III	55

1. INTRODUÇÃO

Nos dias de hoje, a telemetria é um componente fundamental para a obtenção de um *feedback* rápido e confiável em um veículo de alta desempenho. O objetivo da telemetria para veículos de categoria Fórmula SAE é tornar o projeto mais confiável e aprimorá-lo, entregando informações de alto valor aos engenheiros envolvidos no projeto veicular. Além disso, torna todo o desenvolvimento do projeto mais seguro, com respostas em tempo real ao que acontece nos testes em pista e na competição anual realizada pela SAE.

1.1. Fórmula SAE

A categoria Fórmula SAE é um projeto iniciado oficialmente em 1979 com a criação de uma competição de veículos de corrida desenvolvidos por graduandos em engenharia. A primeira edição da competição aconteceu em 1981 nos Estados Unidos, substituindo a versão anterior conhecida como Mini-Indy (SAE, 2022). O projeto foi desenvolvido pela SAE que serve como mediadora e jurada nas competições realizadas por todo o globo. As diretrizes a serem seguidas consistem em um livro de regras desenvolvido pela própria SAE, além da organização dos próprios participantes em times que representam a universidade (SAE, 2022).

No Brasil há anualmente uma competição nacional onde as universidades concorrem entre si nos três dias de evento, em provas estáticas e dinâmicas que avaliam a desempenho de cada projeto na pista, além das apresentações técnicas de cada equipe. A primeira edição da competição no Brasil ocorreu no ano de 2004 (SAE, 2022).

Dentro de uma equipe Fórmula SAE há uma subdivisão de áreas para foco em sistemas do veículo, entre elas, a área da eletrônica, a qual é responsável pelo sistema de eletrônica de potência e aquisição de dados do veículo. É muito importante para o projeto um sistema confiável e seguro, e para realizar essa garantia, a eletrônica realiza a coleta de dados de sensores de diversos sistemas, como a suspensão e direção, acionamentos, *powertrain*, motor e aerodinâmica. Através dos dados coletados as outras áreas recebem um *feedback* do comportamento de seu sistema no veículo e podem realizar a otimização do mesmo.

Figura 1. Equipe Fênix Racing FSAE na competição SAE 2019



Fonte: Equipe Fênix Racing, 2019

1.2. Aquisição de dados em um carro de corrida

Em um veículo voltado à competições, um dos sistemas mais importantes para otimização dos resultados é a aquisição de dados. A aquisição de dados de um veículo de corrida é uma memória eletrônica que armazena parâmetros em função do tempo definidos pelo usuário com o objetivo de serem analisados através de um computador (SEGERS, 2014).

De acordo com o livro *Analysis Techniques for Racecar Data acquisition* a aquisição de dados pode ser subdividida em categorias, sendo elas a análise de desempenho do veículo, a análise de desempenho do piloto, o desenvolvimento do veículo, a confiabilidade e segurança, a disposição dos parâmetros do veículo e o registro de dados históricos do carro (SEGERS, 2014).

Os sinais coletados pela equipe dependem principalmente do orçamento relacionado à atividade, mas as possibilidades são inúmeras. Esses sinais podem ser divididos entre sinais básicos e suplementares, os sinais básicos são necessários para o correto funcionamento do veículo, sendo eles:

- RPM do motor;
- Tensão da bateria;
- Posição da borboleta da admissão;
- Aceleração lateral e longitudinal do veículo;
- Posição do volante;
- Velocidade do veículo;
- Temperatura do motor;
- Sensor de oxigênio do escapamento;

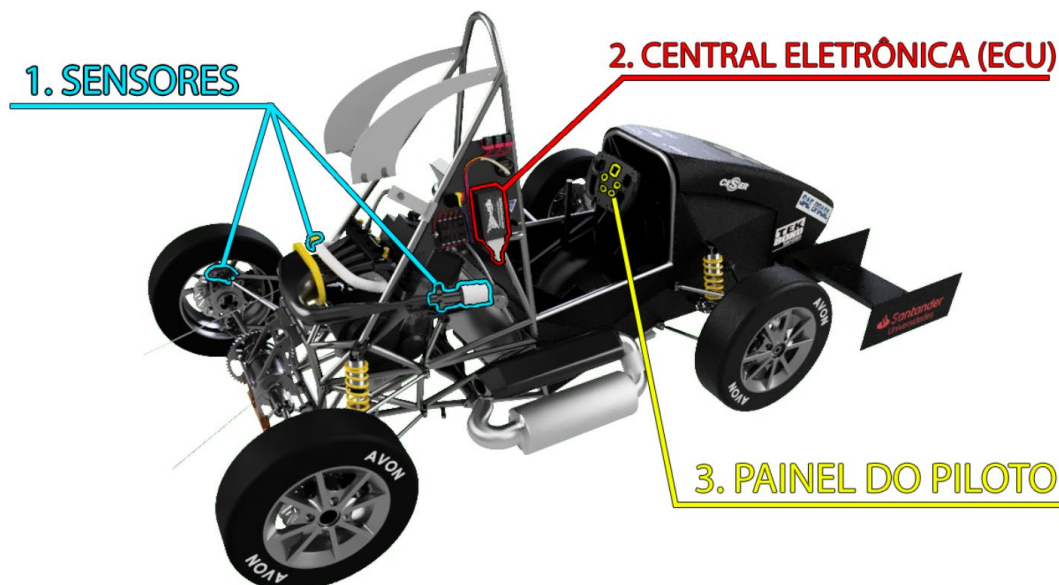
Já os sinais suplementares auxiliam no desenvolvimento do veículo, entre eles estão:

- Pressão da linha de freio;
- Movimento da suspensão;
- Temperaturas de roda;
- Velocidade de cada roda;
- Pressões aerodinâmicas;
- Posição da marcha, entre outros.

Essa lista possui outros inúmeros sinais possíveis, que entregarão dados relacionados a cada problema ou questão que os engenheiros envolvidos no projeto terão ao longo do desenvolvimento de cada sistema do veículo. Além da coleta dos dados, deve-se lembrar que para cada sensor relacionado aos sinais apresentados há um custo de cabos para alimentação do sistema e sinal, memória eletrônica no sistema de armazenamento, correto processamento para que não exista perda de dados durante o recebimento das informações, entre outras questões relacionadas ao *hardware* utilizado (SEGERs, 2014).

Além da escolha dos sinais que serão incluídos no sistema de aquisição de dados, há ainda a escolha de um *hardware* compatível com o que se espera, esses componentes podem variar de sistema para sistema, mas geralmente possuem os mesmos componentes principais, conforme Figura 2:

Figura 2. Componentes principais de um sistema de aquisição de dados de um veículo de corrida.



Fonte: Elaborado pelo autor

Todos os sinais vitais para o correto funcionamento do veículo vão para a ECU, que pode ou não possuir um sistema de registro de dados próprio, no FXX a ECU utilizada é o modelo *Megasquirt MS3/MS3X*, a qual será discutida mais à frente neste trabalho.

Ainda de acordo com o livro *Analysis Techniques for Racecar Data Acquisition*, a frequência de aquisição de um sinal pode variar de acordo com o que se pretende retirar de informação do mesmo (SEGERS, 2014). Para exemplificar, sensores de temperatura do motor não precisam de uma taxa de atualização tão alta pois essa grandeza física não possui uma variação alta, diferente por exemplo da variação do curso de uma suspensão, onde é necessário uma taxa de atualização maior. Na Tabela 1 é possível visualizar a taxa de atualização mínima e recomendada para alguns dados.

Tabela 1. Taxa de atualização mínima e recomendada para sinais

Sensor	Taxa mínima de atualização (Hz)	Taxa recomendada de atualização (Hz)
Posição de GPS	5	20
Movimento da suspensão	200	500
Temperatura de ar e fluidos	1	5
Atividade do <i>Chassis</i> e do piloto	50	50
Pressão do ar e fluidos	10	10

Fonte: *Analysis Techniques for Racecar Data Acquisition*, 2014

Outro ponto importante é a resolução do dado a ser analisado e coletado, para isso, é necessário a realização de um estudo para cada instrumento utilizado na leitura dos dados e qual é a informação de saída esperada. Nos próximos tópicos será possível visualizar que a resolução do Arduino UNO, utilizado neste trabalho, é de 10-bit, ou seja, em uma conversão analógico-digital, o mesmo pode entregar até $2^{10} = 1024$ passos discretos. Para uma alimentação de 5000 mV, tem-se a menor diferenciação de leitura a cada $5000/1024 = 4,88$ mV. Como exemplo, utiliza-se então dessas propriedades para a leitura de potenciômetro linear de 30mm conectado nos mesmos 5 Volts de alimentação, tem-se então uma necessidade de variação de aproximadamente $30/1024 = 0,0293$ mm para tornar o dado mensurável. Cada sensor será selecionado pela equipe de acordo com suas necessidades, e conectado ao sistema de aquisição de dados, para então ser enviado a telemetria projetada por este trabalho.

Há ainda outras questões relacionadas à aquisição de dados como o envio dos dados tratados para o visor utilizado pelo piloto, porém não serão discutidas neste trabalho.

1.3. Telemetria

Como mostrado no tópico anterior a aquisição de dados é um passo muito importante em um veículo de corrida, porém, ainda há um passo além, que é a entrega desses dados para o usuário final em tempo real. Essa entrega de dados em tempo real é conhecida como telemetria e tem como objetivo diminuir o tempo para análise dos dados e também aumentar a segurança do piloto e do projeto, já que a equipe em

base recebe dados importantes como a temperatura do motor ou dos pneus, e pode agir com muito mais velocidade em caso de alguma informação fora do *range* esperado.

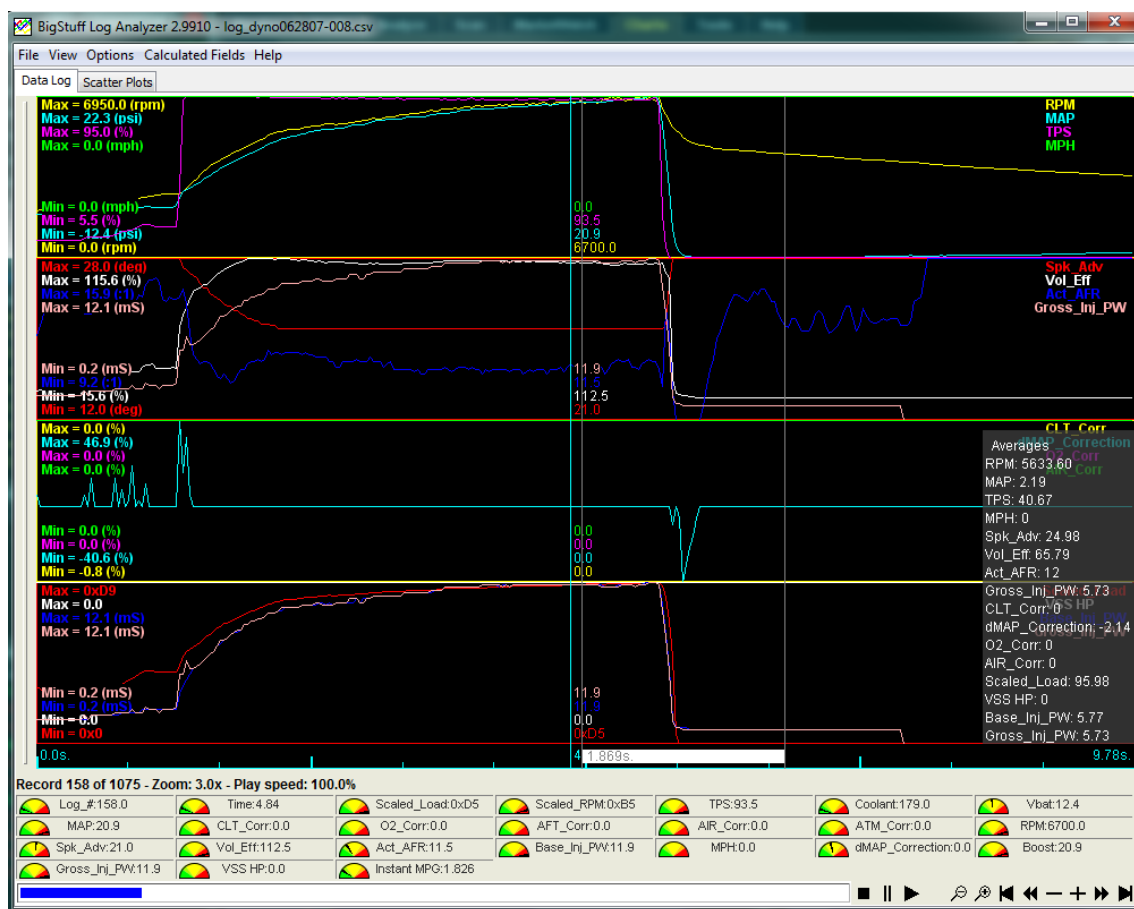
Para que a telemetria exista, é necessário antes que a aquisição de dados esteja em funcionamento. Após o processo de aquisição de dados estar completo, realiza-se a implementação da telemetria que pode ser feita com uma infinidade de ferramentas e processos diferentes, transmissões utilizando protocolos de comunicação de rádio, internet, servidores para armazenamento paralelo e *softwares* finais para entrega legível dos dados coletados, processados e enviados pela aquisição original.

O primeiro passo para a criação da telemetria é a escolha de um *hardware* apropriado para o envio dos dados coletados, para isso leva-se em consideração a velocidade de processamento, o barramento de dados para recebimento e envio dos dados, os protocolos disponíveis para conexão, a velocidade de transmissão, a segurança e o custo de implementação. Para esse trabalho, selecionou-se dois microcontroladores, sendo um Arduino e um ESP-32, os quais serão introduzidos nos próximos tópicos.

Após o sistema de envio selecionado, deve-se escolher um sistema para o recebimento dos dados e os protocolos que serão utilizados para envio do mesmo, o que inclui qualquer tipo de dispositivo que transforme os dados recebidos em informações legíveis para o usuário final, além disso, pode-se adicionar uma etapa de armazenamento para consultas históricas. A utilização de um servidor cumpre bem esse objetivo, assim, é possível enviar os dados para um computador com conexão à internet e que será preparado para receber os dados.

O último passo, caso necessário, é a seleção de um *software* que transforme os dados recebidos em uma visualização simples e informativa para que os engenheiros trabalhem em cima. Há vários *softwares* que cumprem com essa funcionalidade, como por exemplo o *MegaLogViewer* representado na Figura 3, que cria gráficos de acordo com os dados para rápida e fácil visualização do usuário. Além disso, é possível criar o próprio *software* para leitura em tempo real, como uma página na *Web* ou um aplicativo para celular.

Figura 3. Interface do MegaLogViewer



Fonte: *EFIAnalytics*, 2020

A Figura 3 mostra alguns exemplos de sinais de veículos que podem ser lidos utilizando o *MegaLogViewer*, como rotações por minuto, MAP, TPS e velocidade.

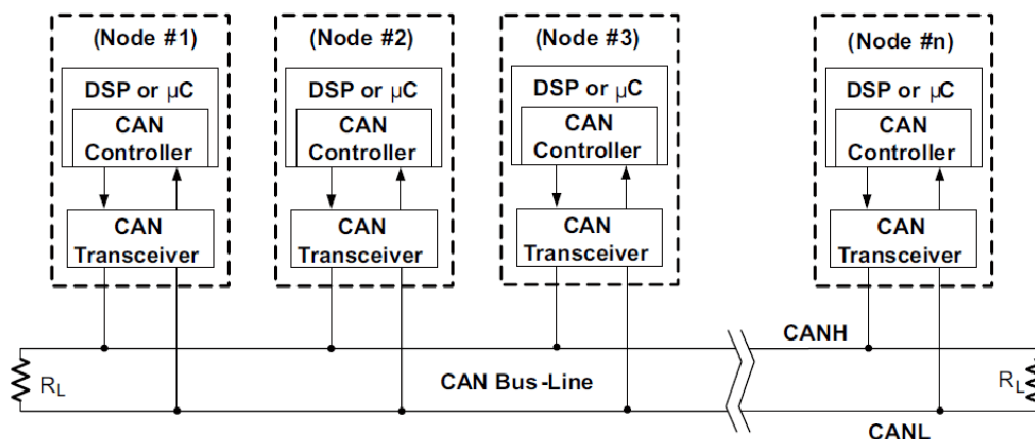
1.4. Protocolo CAN

O CAN é um protocolo projetado por Robert Bosch para microcontroladores e dispositivos, baseado nas normas ISO 11898 e ISO 11519-2 para utilização em redes de comunicação serial para veículos automotivos. A velocidade de transferência pode chegar a até 1 Mbit/s (BOSCH, 1991). O meio físico da comunicação CAN é a utilização de dois fios no arranjo de par trançado, nomeados de CAN-H e CAN-L, e a comunicação é realizada através da diferença de potencial entre esses dois fios. (BOSCH, 1991)

O protocolo CAN é utilizado neste projeto para trocar informações entre a injeção eletrônica e o microcontrolador Arduino, que serão utilizados no desenvolvimento da telemetria do veículo. As informações são transferidas através de pacotes denominados *frames*, cada *frame* representa uma mensagem com campos que definem as propriedades da mensagem, o primeiro campo denomina-se

“*Arbitration ID*” que refere-se ao endereço para o qual a mensagem será enviada, isso é necessário pois o protocolo permite vários dispositivos conectados de uma vez conforme a Figura 4 mostra:

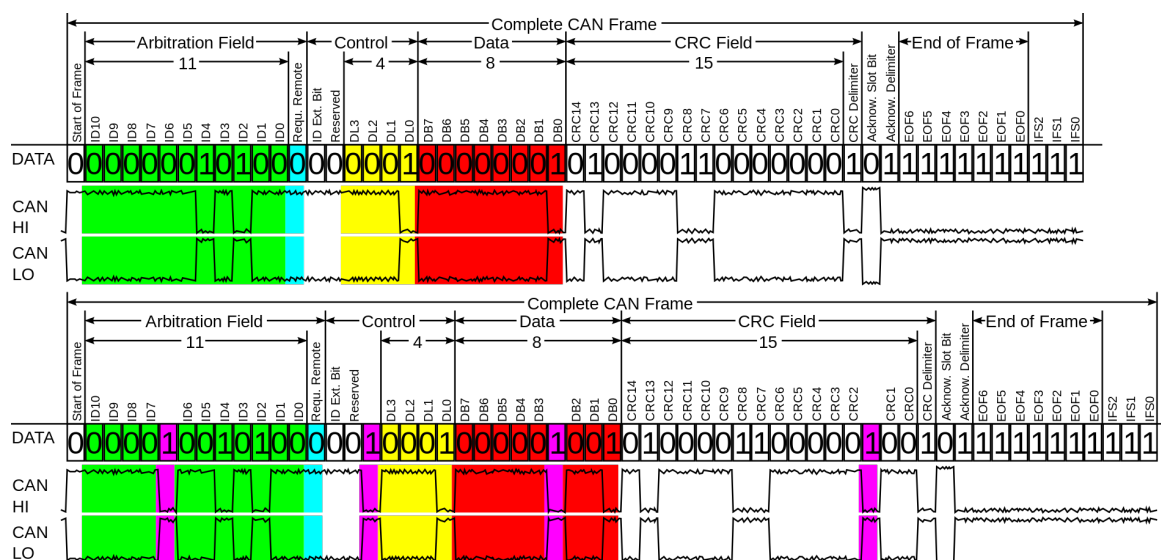
Figura 4. Barramento físico do protocolo CAN



Fonte: Texas Instruments, 2016

A Figura 5 mostra o protocolo de mensagens CAN em detalhe conforme explicado anteriormente:

Figura 5. Protocolo de mensagens CAN



Fonte: WikiCommons, Wikipedia

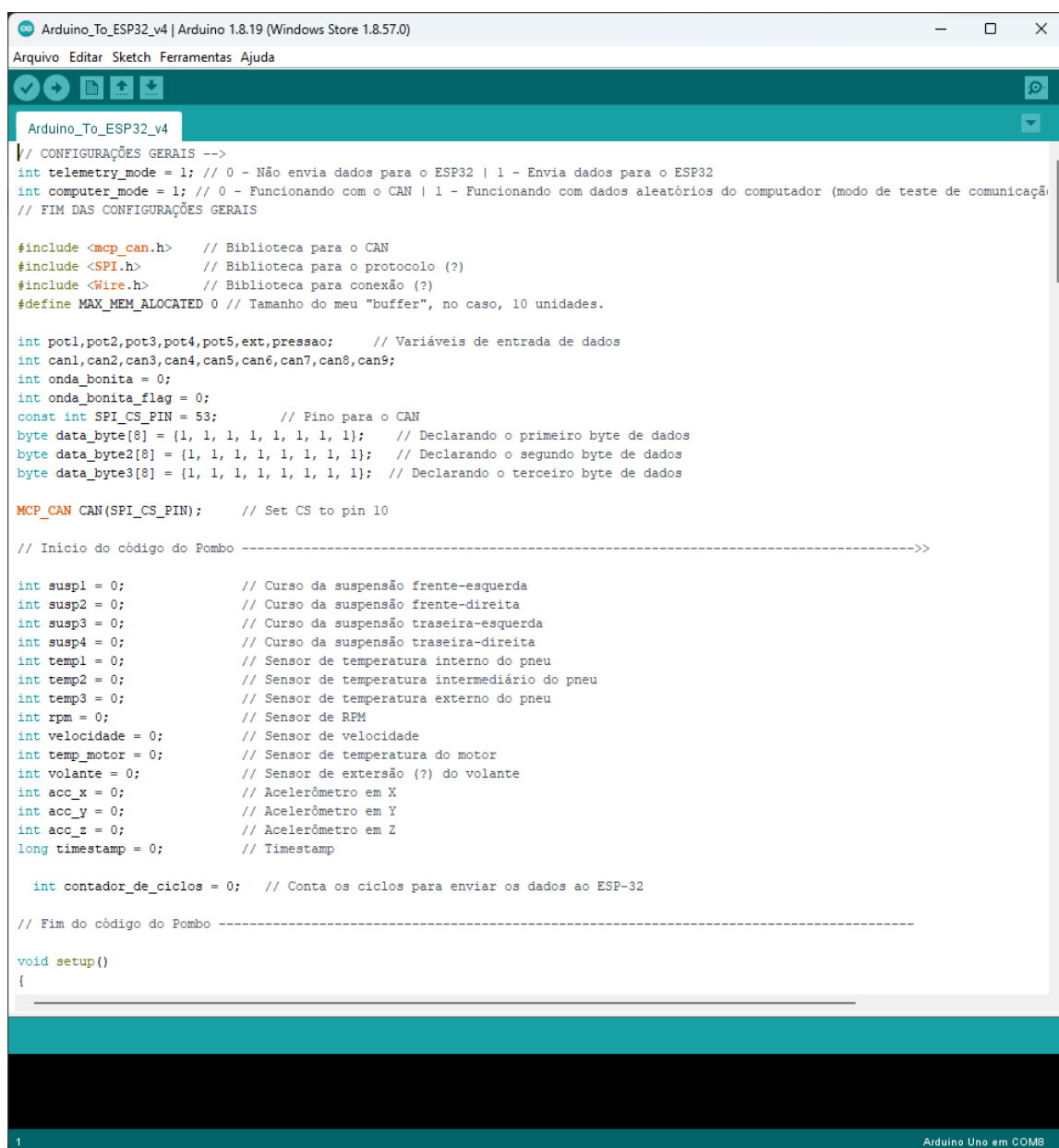
Após o primeiro campo, tem-se o campo “*Data Length*” que identifica para o receptor a quantidade de bytes referentes à informação transportada, que está localizada no próximo campo, definido como “*Data field*”. O último campo é o CRC que serve para identificação de possíveis erros no pacote de dados (SANTOS, 2022).

1.5. MICROCONTROLADOR ARDUINO

O Arduino é uma plataforma aberta de prototipação eletrônica, composta pelo *hardware* (placa controladora) e *software* (ambiente de desenvolvimento integrado).

O ambiente de desenvolvimento integrado foi inicialmente desenvolvido em 2005, na Itália, e tem compatibilidade com qualquer sistema operacional sob licença GPL/LGPL para *software*. A plataforma permite a rápida compilação e transferência de código em C/C++ pela interface USB para a placa controladora, de forma bem intuitiva e prática, é possível visualizar um exemplo da interface conforme Figura 6:

Figura 6. Ambiente de desenvolvimento integrado Arduino



```

Arduino_To_ESP32_v4 | Arduino 1.8.19 (Windows Store 1.8.57.0)
Arquivo Editar Sketch Ferramentas Ajuda

Arduino_To_ESP32_v4
// CONFIGURAÇÕES GERAIS -->
int telemetry_mode = 1; // 0 - Não envia dados para o ESP32 | 1 - Envia dados para o ESP32
int computer_mode = 1; // 0 - Funcionando com o CAN | 1 - Funcionando com dados aleatórios do computador (modo de teste de comunicação)
// FIM DAS CONFIGURAÇÕES GERAIS

#include <mcp_can.h> // Biblioteca para o CAN
#include <SPI.h> // Biblioteca para o protocolo (?)
#include <Wire.h> // Biblioteca para conexão (?)
#define MAX_MEM_ALLOCATED 0 // Tamanho do meu "buffer", no caso, 10 unidades.

int pot1,pot2,pot3,pot4,pot5,ext,pressao; // Variáveis de entrada de dados
int can1,can2,can3,can4,can5,can6,can7,can8,can9;
int onda_bonita = 0;
int onda_bonita_flag = 0;
const int SPI_CS_PIN = 53; // Pino para o CAN
byte data_byte[8] = {1, 1, 1, 1, 1, 1, 1, 1}; // Declarando o primeiro byte de dados
byte data_byte2[8] = {1, 1, 1, 1, 1, 1, 1, 1}; // Declarando o segundo byte de dados
byte data_byte3[8] = {1, 1, 1, 1, 1, 1, 1, 1}; // Declarando o terceiro byte de dados

MCP_CAN CAN(SPI_CS_PIN); // Set CS to pin 10

// Início do código do Pombo ----->>>

int susp1 = 0; // Curso da suspensão frente-esquerda
int susp2 = 0; // Curso da suspensão frente-direita
int susp3 = 0; // Curso da suspensão traseira-esquerda
int susp4 = 0; // Curso da suspensão traseira-direita
int temp1 = 0; // Sensor de temperatura interno do pneu
int temp2 = 0; // Sensor de temperatura intermediário do pneu
int temp3 = 0; // Sensor de temperatura externo do pneu
int rpm = 0; // Sensor de RPM
int velocidade = 0; // Sensor de velocidade
int temp_motor = 0; // Sensor de temperatura do motor
int volante = 0; // Sensor de extensão (?) do volante
int acc_x = 0; // Acelerômetro em X
int acc_y = 0; // Acelerômetro em Y
int acc_z = 0; // Acelerômetro em Z
long timestamp = 0; // Timestamp

int contador_de_ciclos = 0; // Conta os ciclos para enviar os dados ao ESP-32

// Fim do código do Pombo ----->>>

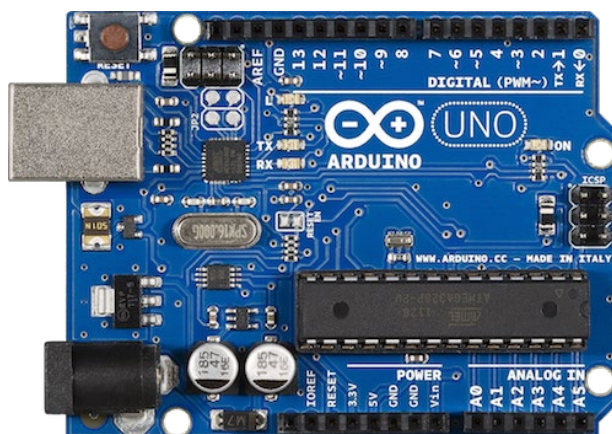
void setup()
{
  //

```

Fonte: Elaborado pelo autor

Em questão de *hardware*, o Arduino possui várias versões, cada uma com suas propriedades, como a quantidade de portas I/O analógicas ou digitais, saídas de alimentação, módulos de conexão, tipo de microprocessador utilizado, entre outras. Com o Arduino é possível a realização de leitura de dados ou processamento do mesmo. Os modelos Arduino UNO e Arduino MEGA têm uma resolução de 10-bit e uma taxa de 10000 leituras por segundo em uma porta analógica, ou seja, uma frequência máxima de 10kHz, muito acima do necessário para o desenvolvimento de um sistema de telemetria robusto conforme mostrado nos tópicos anteriores. É possível visualizar um microcontrolador Arduino UNO na Figura 7:

Figura 7. Arduino UNO



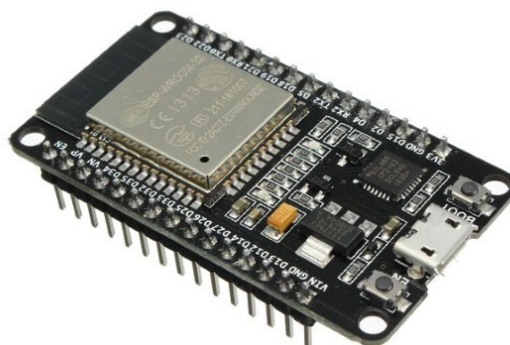
Fonte: Arduino UNO R3 Documentation, 2022

No canto superior esquerdo da Figura 7 pode-se visualizar um botão para controle e *reset* do processo, a entrada USB tipo B e a entrada de energia mais abaixo. Os quadrados pretos localizados nas extremidades superior e inferior da placa são as portas de conexão para controle da placa, podendo ser de energia, digital ou analógica.

1.6. MICROCONTROLADOR ESP-32

Com o avanço da tecnologia, microcontroladores ficaram cada vez mais baratos e mais simples de serem implementados, inclusive para o envio e recebimento de informações. O ESP-32 que é uma placa de desenvolvimento que possui módulo de conexão Wi-Fi e Bluetooth, além das interfaces I2C, CAN, SPI, UART, leitura de sinais analógicos e digitais, com uma confiabilidade razoável e custo de aquisição baixíssimo, que pode ser visualizado na Figura 8:

Figura 8. Placa de desenvolvimento ESP-32



Fonte: FilipeFlop, 2022

É possível realizar a transmissão de dados do ESP-32 para um servidor através de seu módulo Wi-Fi. O Módulo ESP-32 possui 18 canais analógicos/digitais, com uma frequência de amostra de 6kHz e uma resolução de 12-bit. Utiliza-se o ambiente de desenvolvimento integrado do Arduino para realizar a compilação e envio dos códigos programados em C/C++. Há além disso, *firmwares* próprios para programação em próprios para programação em Python, uma linguagem também muito utilizada e que possui suas vantagens e desvantagens se comparada a utilização do C/C++.

1.7. INJEÇÃO ELETRÔNICA MEGASQUIRT MS3/MSX

No desenvolvimento do veículo de categoria Fórmula SAE, mais precisamente na área da eletrônica, o fator mais importante para iniciar a montagem do sistema é a escolha de uma injeção eletrônica que permita a realização de todas as funcionalidades definidas previamente durante o período de projeto do veículo. Para o caso da equipe, a injeção eletrônica utilizada é a MEGASQUIRT MS3/MSX, ela permite conexão CAN com uma alta taxa de atualização, o que é essencial para o envio e recebimento de dados no sistema de telemetria. Na Figura 9 é possível visualizar o modelo instalado no FXX:

Figura 9. Injeção eletrônica MegaSquirt MS3/MSX instalada no veículo FX9



Fonte: Elaborado pelo autor

O acesso a Mega em um computador dá-se por um cabo USB ou por uma interface serial RS-232 acopladas na parte superior da carcaça do aparelho. Após a conexão, utiliza-se do *software* TunerStudio para realizar todas as configurações necessárias, tanto para o sistema integrado de aquisição de dados, quanto para o sistema de *Tuning* do motor. É possível visualizar a tela inicial do *software* TunerStudio na Figura 10:

Figura 10. Software TunerStudio



Fonte: Elaborado pelo autor

A interface inicial mostra os principais dados recebidos pela injeção eletrônica, o que é excelente para a resolução de possíveis problemas que possam vir a acontecer durante os primeiros testes de bancada com o veículo.

A injeção eletrônica também conta com uma entrada para *SD Card*, que armazena os dados coletados pela ECU ao longo do tempo e permite a análise gráfica após testes e competições. Esses dados podem ser enviados através do protocolo CAN para o Arduino, a comunicação permite até 4000 pacotes por segundo, isso é o equivalente a 40 canais de informação em uma frequência de 100 Hz. A injeção eletrônica conta com muitas outras funcionalidades para análise e fundamentação dos dados coletados (MURRAY, 2018).

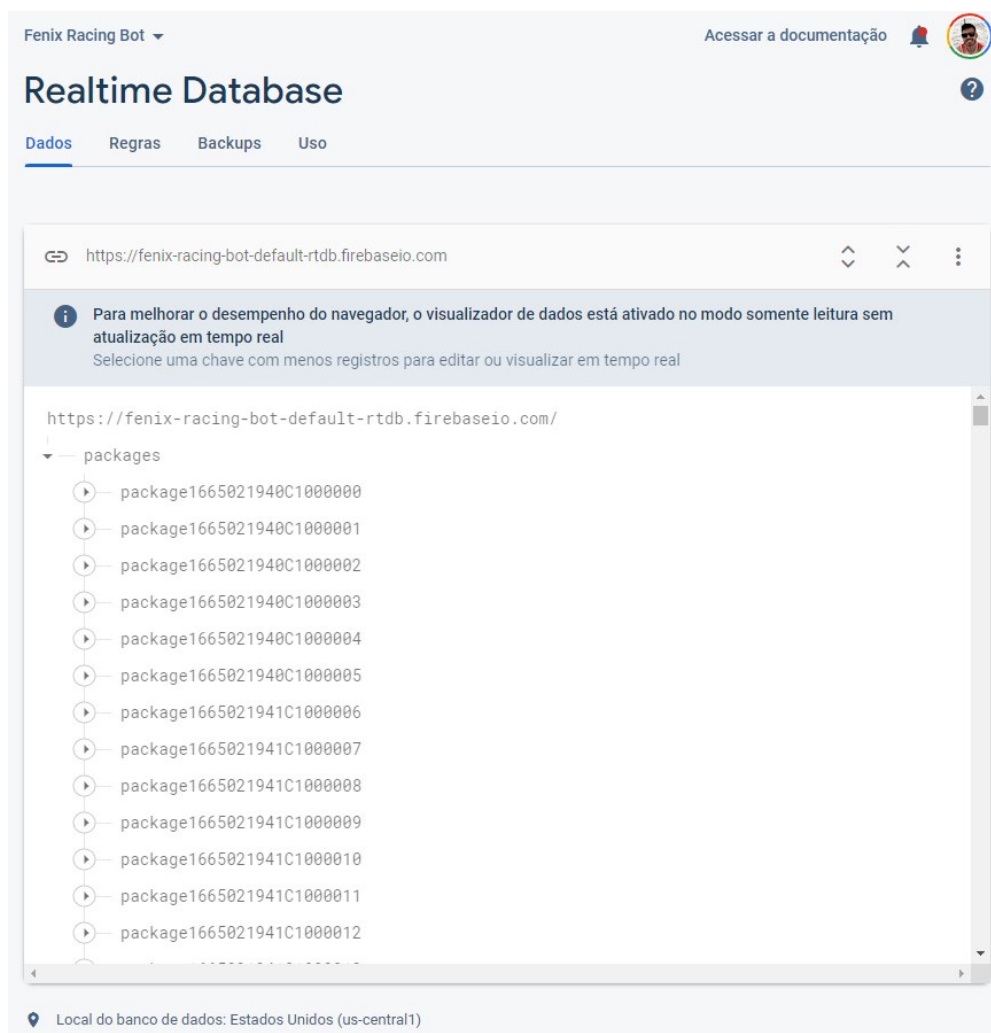
1.8. BASE DE DADOS EM TEMPO REAL

Para uma telemetria ser segura e robusta, é ideal que as informações além de serem apresentadas em tempo real também sejam armazenadas para a realização de consultas futuras. Para realizar o acesso ao histórico de dados na injeção eletrônica é necessário que o carro esteja parado para que o cartão SD seja removido e lido por um *software* em um computador localmente, o que atrasa as análises necessárias durante a realização de um teste. Uma forma de contornar este problema é

implementando um sistema de armazenamento em servidores que trabalhe junto com o sistema de telemetria.

Entre as diversas soluções existentes no mercado, há o serviço *Firebase Realtime Database* da Google que conta com um plano gratuito que permite até 100 conexões simultâneas, 1 GB de armazenamento de dados e 10 GB por mês de banda de transferência de dados, além de 100 mil transferências simultâneas por segundo, o que é mais que suficiente para cumprir com o objetivo da telemetria deste projeto. O serviço permite a sincronização de dados em formato JSON em tempo real na nuvem (FIREBASE, 2022). A Figura 11 mostra a interface da aplicação via *web*:

Figura 11. Interface *Realtime Database*



Fonte: Elaborado pelo autor

Através da plataforma *web* é possível visualizar de forma dinâmica todos os dados recebidos e armazenados pelo serviço, que também possui um sistema de segurança robusto, impedindo que pessoas não autorizadas possam utilizar da plataforma.

1.9. Front-end Interface

Como citado nos tópicos anteriores deste trabalho, a interface final com o usuário é um ponto chave no desenvolvimento de uma telemetria, para isso, pode-se utilizar tanto *softwares* já desenvolvidos e disponíveis no mercado como o MegaLogViewer ou desenvolver um próprio. Nos dias de hoje uma plataforma mundialmente utilizada para acesso de informações é o navegador de internet, é possível utilizá-lo através do computador, do *smartphone*, de televisões inteligentes e outros diversos aparelhos com conexão à internet, mostrando que *softwares* com interface desenvolvida em *Web* possuem uma compatibilidade muito maior e responsiva com usuários.

Uma maneira fácil de representar graficamente os dados enviados pela telemetria é a utilização de uma página *Web* desenvolvida em *javascript*, uma linguagem de programação interpretada estruturada em alto nível com tipagem dinâmica fraca e multiparadigma. O *javascript* possui excelentes bibliotecas pra uso gráfico, entre elas a biblioteca de código aberto ApexCharts.js que permite a criação de gráficos com *design* moderno, licenciada pelo MIT e gratuita inclusive para projetos comerciais. A biblioteca é de fácil desenvolvimento e adaptação para que seja possível a visualização dos gráficos gerados à partir de qualquer formato de visor, seja um celular ou um computador. A Figura 12 mostra exemplos da utilização da biblioteca ApexCharts.js em *javascript*.

Figura 12. Exemplos de gráficos através da biblioteca ApexCharts.js

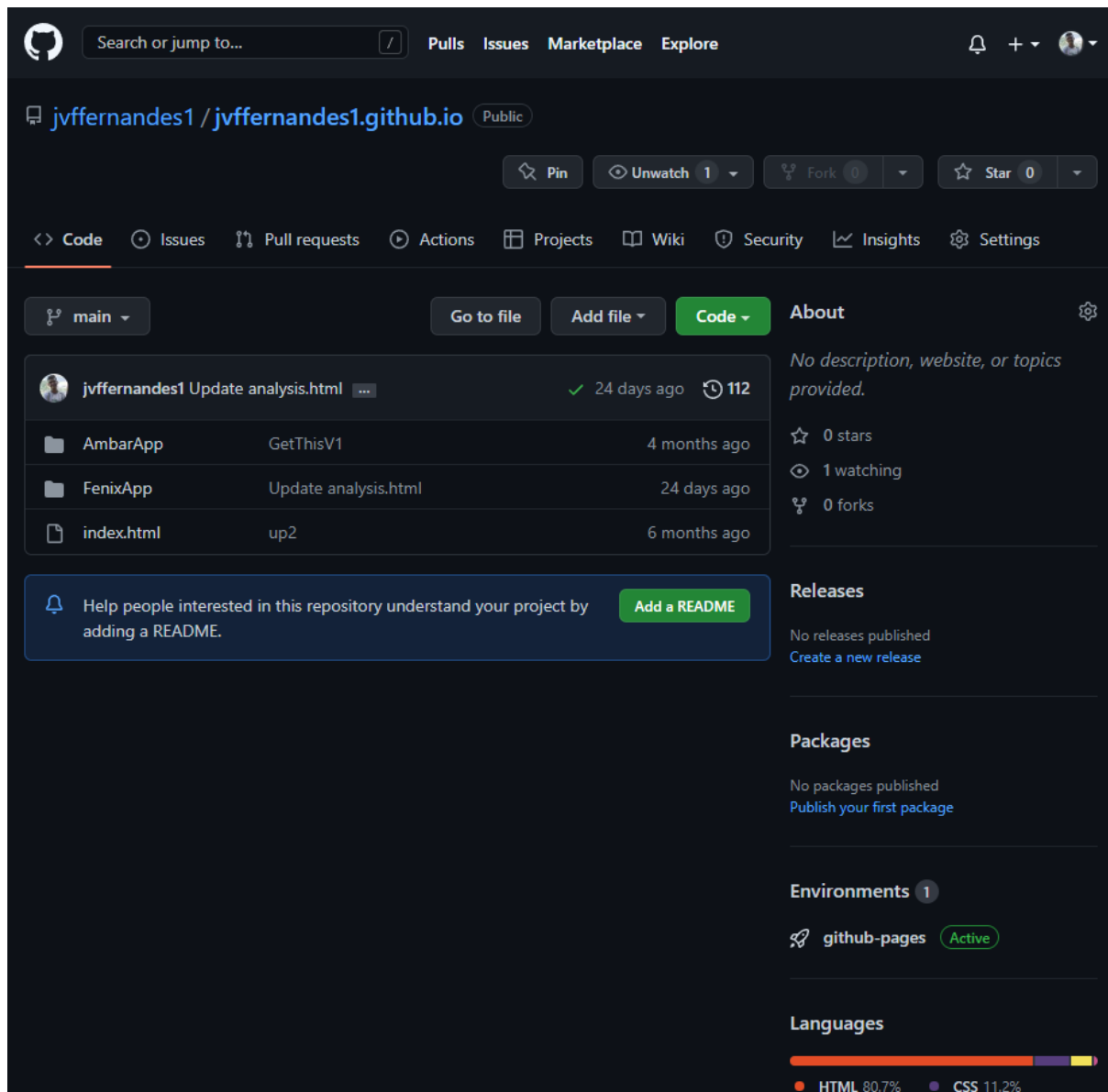


Fonte: Apexcharts.com (2022)

Para que uma página *Web* esteja disponível para os usuários, deve-se utilizar um serviço de hospedagem de sites, seja ele dinâmico ou estático. O GitHub é uma plataforma de hospedagem de código-fonte e arquivos com controle de versão através do Git, que, além disso, oferece um serviço de hospedagem através do *GitHub Pages*,

através da implementação de uma conta e o *upload* do código da página *Web* para o GitHub. A Figura 13 mostra a interface principal do Github para este projeto:

Figura 13 – Página do GitHub para upload do *app*



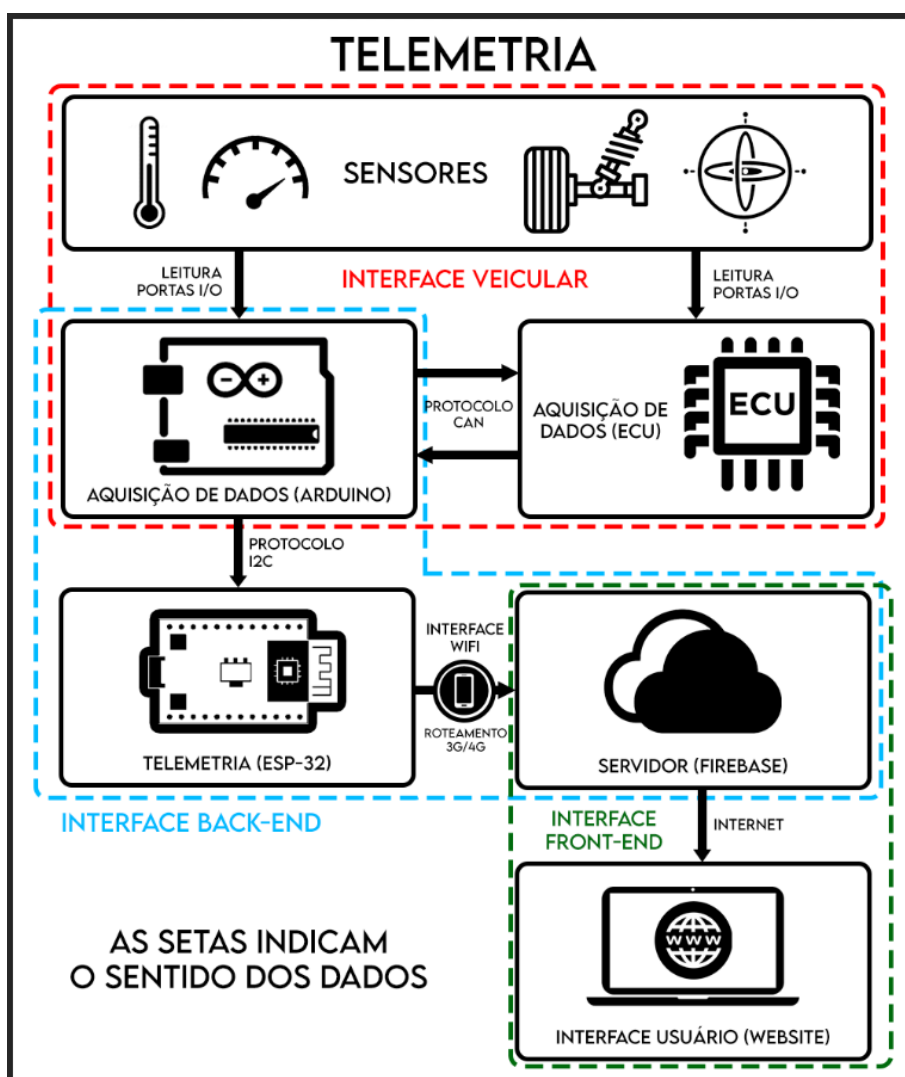
Fonte: Elaborado pelo autor

É possível visualizar na Figura 13 um resumo sobre as linguagens utilizadas para o desenvolvimento da aplicação, além de permitir gerenciar quem tem acesso ao código fonte desenvolvido.

2. IMPLEMENTAÇÃO DO SISTEMA DE TELEMETRIA

A metodologia proposta neste trabalho está dividida em três etapas. A primeira etapa corresponde ao desenvolvimento da interface entre os sensores presentes no veículo, a injeção eletrônica e o microcontrolador Arduino a qual será referida como interface veicular. Essa primeira etapa precisa adaptar-se ao sistema já utilizado pela equipe, assim possui limitações as quais serão identificadas no próximo capítulo. A segunda etapa consiste no desenvolvimento da interface entre o Arduino, o ESP-32 e o serviço de *Realtime Database*, a qual será referida como interface *back-end*. A terceira e última etapa abrange a interface relacionada ao usuário final e a visualização dos dados coletados por meio do *WebSite* em *javascript*, a qual será referida como interface *front-end*. A Figura 14 destaca as interfaces do sistema de telemetria:

Figura 14. Interfaces do projeto de telemetria



Fonte: Elaborado pelo autor

Na Figura 14 é possível visualizar as etapas citadas anteriormente neste texto, mostrando as divisões aplicadas para a etapa de desenvolvimento do projeto. A primeira parte é destacada em vermelha, mostra os sensores e o sentido dos dados que alcançam a ECU e o Arduino, além disso, é possível notar o protocolo CAN que é a interface entre o arduino e a ECU. Destacado em azul encontra-se a segunda interface, que mostra a comunicação entre o Arduino e o ESP-32, além da comunicação entre o ESP-32 e o servidor utilizando WIFI fornecido por um aparelho celular através da rede móvel 3G/4G. A última interface está destacada em verde, onde o *website* desenvolvido lê os dados contidos pelo servidor.

2.1. Interface veicular

O início do desenvolvimento experimental do projeto de telemetria deu-se na realização da configuração da interface veicular do sistema, para isso, realizou-se consultas com o time de eletrônica da equipe Fórmula SAE para entender como era realizada a aquisição dos dados dos sensores e quais seriam os pontos de leitura de informações para a próxima interface. Trabalhou-se em conjunto com a equipe para diferenciar os sensores que seriam lidos pela ECU dos sensores que seriam lidos pelo Arduino. Na ECU concentrou-se os sensores básicos para o funcionamento do veículo devido ao limitante do número de portas disponíveis, sendo eles:

- Sensor de RPM;
- Sensor de temperatura do motor;
- Sensor de posição da borboleta de admissão de ar;
- Sensor de velocidade para cada uma das quatro rodas;

O Arduino recebe os sensores não prioritários, sendo eles:

- Acelerômetro de 3 eixos;
- Sensor de temperatura dos pneus;
- Sensor de curso da suspensão para cada uma das quatro suspensões;
- Sensor de ângulo do volante;

A definição das portas I/O a serem utilizadas por cada sensor foi realizada pelos membros da equipe. Após a realização de uma reunião para priorização dos dados, selecionou-se como pacote para envio ao ESP-32 via protocolo I2C os instrumentos conforme mostra a Tabela 2.

Tabela 2. Lista de sinais a serem enviados para o ESP-32

Sensor	Porta I/O Arduino
Curso da suspensão frontal esquerda	A0
Curso da suspensão frontal direita	A1
Curso da suspensão traseira esquerda	A2
Curso da suspensão traseira direita	A3
Ângulo do volante	A4
RPM	Protocolo CAN
Temperatura do motor	Protocolo CAN

Fonte: Elaborado pelo autor

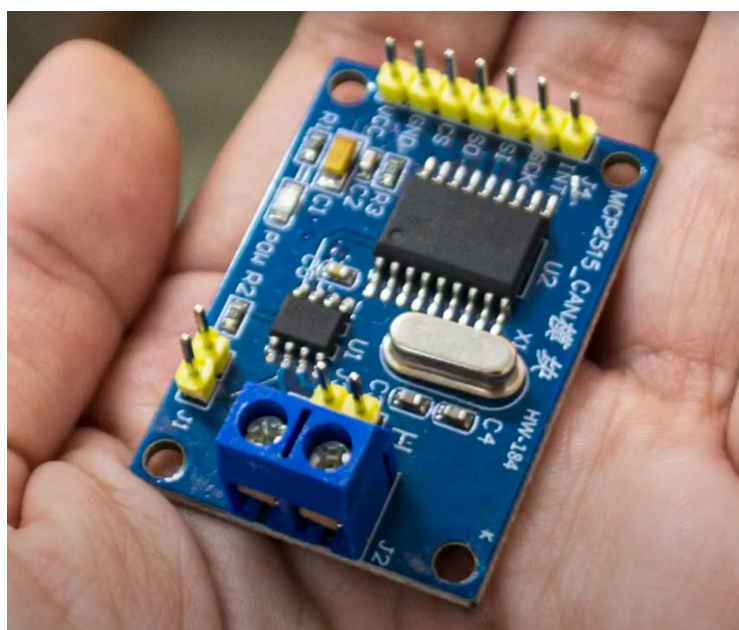
Mesmo que nem todos os sensores tenham sido selecionados para envio ao ESP32, preparou-se o código de forma a enviar o pacote completo de dados, substituindo os sensores faltantes pelo byte representando um valor decimal zero, para quando os próximos sensores forem adicionados, a atualização do código seja feita de forma mais rápida e prática.

Para os sensores de RPM e temperatura do motor, era necessário ainda o desenvolvimento de uma comunicação entre o Arduino e a ECU, para isso, utilizou-se do protocolo CAN que é capaz de garantir a velocidade e segurança necessárias para a realização dessa operação. O protocolo CAN foi configurado de tal forma que cada pacote tivesse 8 bytes de mensagem, e que, para alcançar a resolução desejada, cada informação utilizasse 2 bytes do pacote, assim, era possível transmitir 4 sinais diferentes por cada pacote. É de interesse da equipe que os dados enviados pelos sensores e coletados exclusivamente pelo Arduino estivessem também registrados no *DataLogger* presente na ECU, assim, o código elaborado foi construído de maneira que fosse possível enviar e receber os pacotes, para que o Arduino fosse apto a receber e processar as informações necessárias a serem enviadas ao ESP-32, e também para que a ECU possa armazenar no cartão SD de seu *datalogger* as informações presentes inicialmente apenas no Arduino.

Com o objetivo de realizar a comunicação CAN com o Arduino, utiliza-se uma placa de expansão, ou módulo CAN BUS, que é capaz de converter o que está sendo enviado pelo protocolo para o Arduino. O módulo utilizado é o modelo MCP2515 TJA1050 conforme a Figura 15. O módulo recebe o CAN_H e o CAN_L em uma de suas extremidades, e na outra realiza a conexão com o Arduino, através dos pinos:

- VCC = Alimentação 5 Volts para o módulo
- GND = Conexão *Ground*
- SCK = Serial Clock
- MISO = Master Input Slave Output
- MOSI = Master Output Slave Input
- CS = Chip Select

Figura 15 – Módulo MCP2515 TJA1050



Fonte: Elaborado pelo autor

Após a correta conexão do módulo MCP2515 com o Arduino e das conexões CAN_H e CAN_L devidamente montadas, deve-se realizar a correta configuração do protocolo no *software TunerStudio* para que a injeção eletrônica seja capaz de ler e enviar os dados de maneira apropriada. O último passo foi realizar a implementação do código em C/C++ no Arduino.

2.2. Interface *Back-end*

Com a finalização do desenvolvimento da interface veicular, pode-se concentrar os esforços no desenvolvimento da interface *back-end*. Era necessário estabelecer uma comunicação rápida e prática entre o microcontrolador Arduino e o microcontrolador ESP-32, para isso, utilizou-se do protocolo I2C que permite a

transmissão de 100kbit/s em seu modo normal, mais que suficiente para o que foi proposto neste trabalho. O ESP-32 funciona como mestre e realiza uma requisição ao Arduino (escravo) toda vez que estiver pronto para enviar dados ao servidor *firebase*, o Arduino por outro lado processa recebe as informações dos sensores conectados diretamente a ele e conectados pelo protocolo CAN, armazenando em um vetor os pacotes de dados recebidos. Os códigos em C/C++ para o Arduino e para o ESP-32 podem ser visualizados nos Anexo I e Anexo II deste trabalho.

Após a finalização do processo de conexão e criação do código para a comunicação I2C, realizou-se a expansão do código do ESP-32 para conectar com o servidor *firebase*, para isso utilizou-se de bibliotecas que permitem tal vínculo. O processo funciona de maneira que o pacote *json* é montado e enviado em quatro ciclos, um para cada pacote de dados enviado pelo Arduino, cada pacote é diferenciado pelo *timestamp* (carimbo de hora) em que o dado foi gerado, esse *timestamp* é realizado pelo próprio ESP-32 utilizando uma biblioteca e a conexão com a internet.

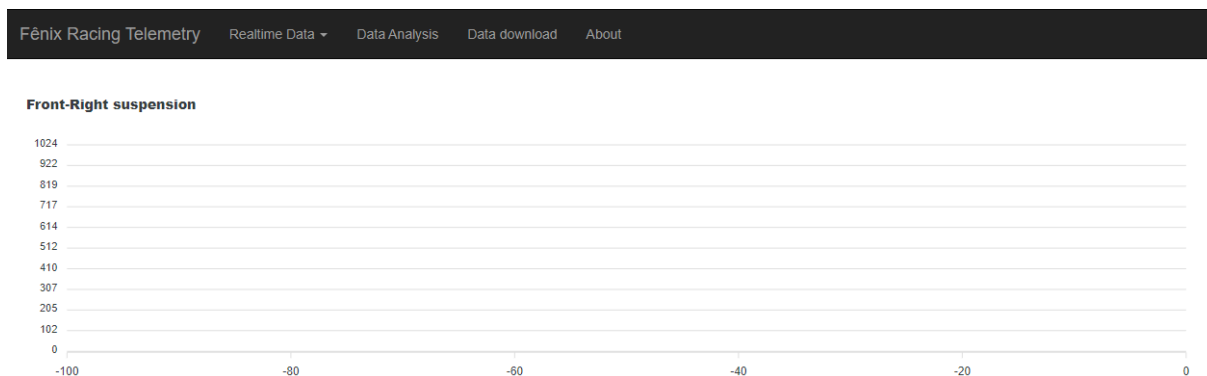
A conexão do ESP-32 com a internet é configurada através da conexão WiFi com um aparelho celular que está presente no bolso do piloto. O aparelho celular é responsável por transmitir rede móvel 3G/4G. Seguidamente a correta configuração do código presente no ESP-32 e, após a conexão do microcontrolador com o Arduino e a interface veicular, espera-se receber os pacotes no servidor *firebase*, organizados em formato *json* permitindo um fácil acesso pela interface *front-end*. Ao total foram escritas 559 linhas de código para o Arduino e para o esp-32, desconsiderando as bibliotecas importadas.

2.3. Interface Front-end

A última interface a ser desenvolvida neste trabalho é a interface *front-end* que é responsável pela entrega das informações coletadas para o usuário final de forma clara e útil, utilizando de recursos em *javascript* para desenvolvimento da página *web*.

Inicialmente realizou-se uma pesquisa para escolha da melhor biblioteca para visualização gráfica das informações. A biblioteca escolhida em questão foi a *apex charts* que permite uma visualização limpa e clara em forma gráfica. Desenvolveu-se então a primeira página que permite a visualização dos dados em tempo real com um histórico de até 100 informações, como pode ser vista na Figura 16.

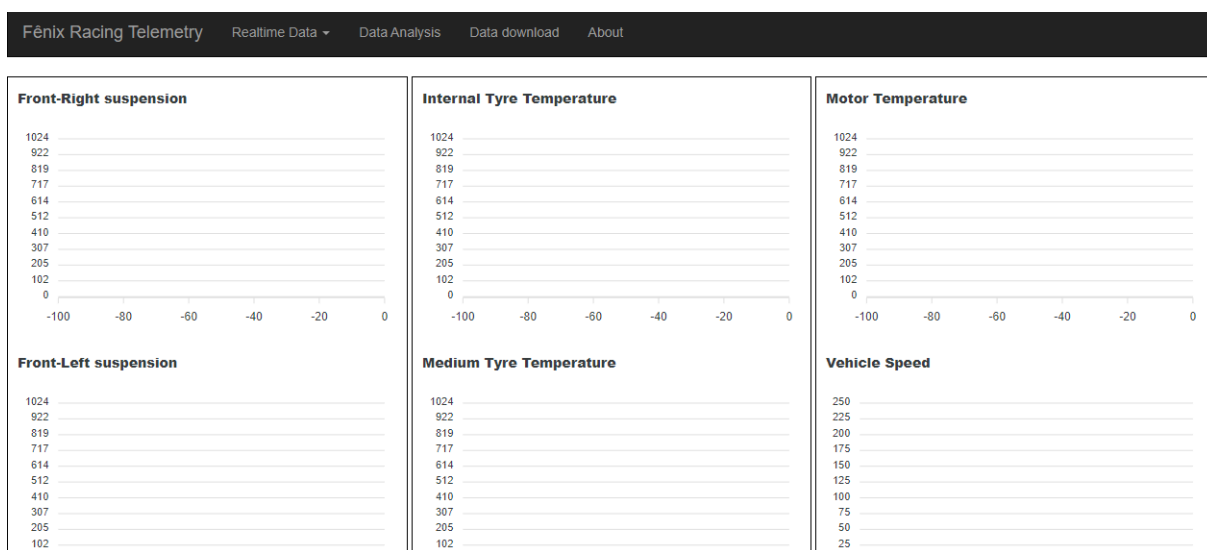
Figura 16 – Página de visualização em tempo real dos dados coletados.



Fonte: Elaborado pelo autor

O menu superior permite a seleção da página a ser visualizada. Realizou-se uma reunião com os gerentes de cada área da equipe de Fórmula SAE para decidir a melhor forma de destacar as informações coletadas, o que levou ao desenvolvimento de páginas auxiliares com informações mais compactadas, como mostra a Figura 17.

Figura 17 – Página de visualização em tempo real, compactada.



Fonte: Elaborado pelo autor

A ferramenta ainda permite diferentes tipos de recursos visuais para otimizar a visualização das informações, conforme pode ser visto na Figura 18, porém que não foram implementadas para este trabalho:

Figura 18 – Implementação de recursos visuais para a criação dos gráficos

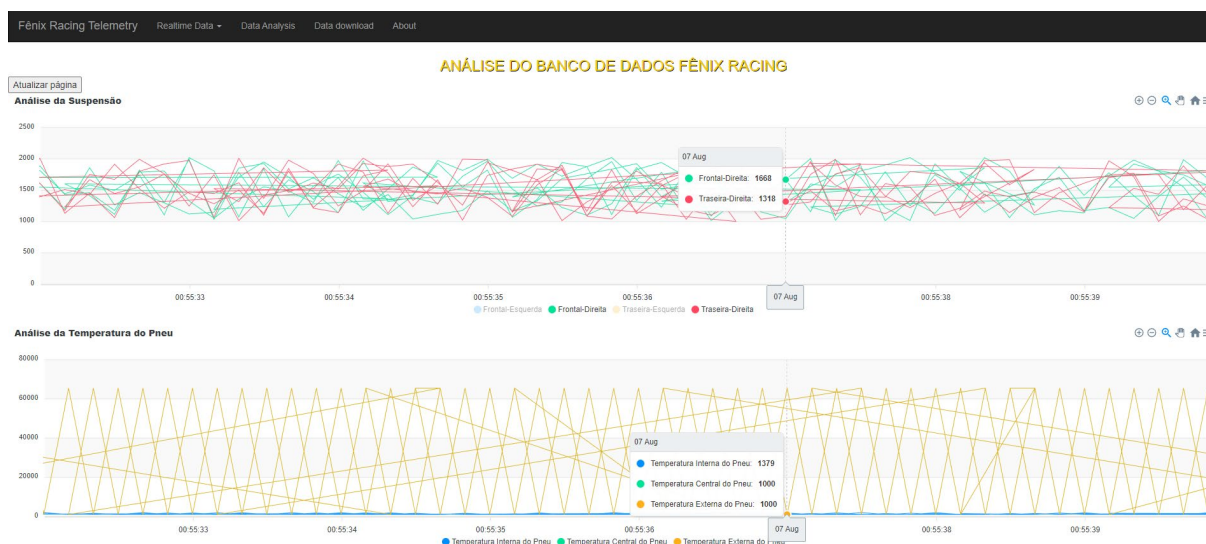


Fonte: Elaborado pelo autor

O desenvolvimento da página web também contou com *HTML* para implementação da biblioteca em *javascript* e *CSS* para o *design*.

Além das páginas para visualização em tempo real, também desenvolveu-se uma página para visualização de todos os dados que estão no banco de dados, não limitando-se as últimas 100 informações, além disso, essa página de análise permite modificações na curva demonstrada pelos gráficos, permitindo *zoom*, seleção de dados, linhas de auxílio para comparação e até mesmo uma opção de *download* em formato de imagem, conforme Figura 19.

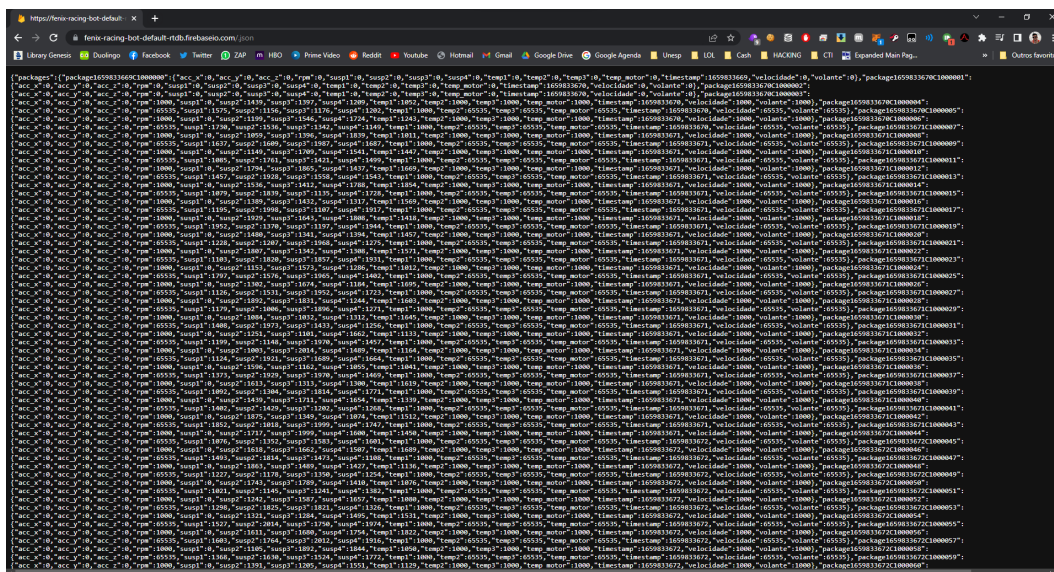
Figura 19 – Página de análise de informações



Fonte: Elaborado pelo autor

Por último, é oferecido uma opção para baixar o banco de dados caso seja necessária a análise *offline* de dados, conforme mostra a Figura 20:

Figura 20 – Página para seleção e *download* das informações no banco de dados.



Fonte: Elaborado pelo autor

Para que a interface faça conexão com o banco de dados *firebase* em tempo real utiliza-se a biblioteca fornecida pela própria *Google* (desenvolvedora do *firebase*) em *javascript*, e, com algumas adaptações, criou-se a conexão da mesma com os gráficos desenvolvidos. Para a interface de análise, utiliza-se a própria página de *download* da ferramenta para acessar os dados e então tratá-los conforme necessário.

Ao total foram escritas 2482 linhas para o desenvolvimento das páginas *web*, desconsiderando as bibliotecas importadas. O código pode ser encontrado inteiramente no github no repositório disponível para atualização do mesmo: <https://github.com/jvffernandes1/jvffernandes1.github.io/tree/main/FenixApp>.

A interface pode ser acessada pelo link:
<https://jvffernandes1.github.io/FenixApp>.

2.4. Testes

Após a finalização das três interfaces, realizou-se adaptações nos códigos para criação de um modo de teste que permite *benchmarks* de velocidade com maior facilidade, assim, duas variáveis no código do Arduino permitem a realização de testes com dados randômicos ou até mesmo com o ESP-32 desligado (funcionando apenas para envio e recebimento de dados pelo protocolo CAN, e não para o envio a internet). A funcionalidade de testes também permite a análise do código de forma mais prática,

não necessitando do funcionamento do veículo Fórmula SAE (que depende de outras áreas para teste em pista).

Assim finalizou-se o desenvolvimento do sistema de telemetria e tornou-se possível coletar os resultados dos testes realizados em bancada e em pista. É possível encontrar um esquemático do projeto no Anexo III.

3. SIMULAÇÃO DO SISTEMA

Para medir a eficácia do projeto, dividiu-se em algumas partes os testes elaborados para a obtenção de velocidades de transmissão de cada interface. O primeiro passo a ser realizado era obter resultados em relação a transferência de dados entre os dois microcontroladores (Arduino e ESP-32), para que isso fosse possível, desenvolveu-se um modo de testes onde era possível obter valores sequenciais de 0 à 1023 no Arduino, e transferí-los para o ESP-32, e após alcançar o valor de 1023, o contador se tornaria decrescente e enviaria a sequência de 1023 à 0, repetindo esse processo enquanto o sistema estivesse conectado. Através dessa abordagem é possível visualizar a frequência de envio de dados através do protocolo I2C de maneira que o carregamento de dados pelo protocolo CAN e pela entrada direta de dados nas portas analógicas não sejam um limitante na análise, ou seja, essa abordagem isola a transferência de dados entre o Arduino e o ESP-32 de qualquer interferência externa para garantir os resultados encontrados. Para a obtenção dos dados em questão, utiliza-se a própria ferramenta de visualização *serial* fornecida pela *IDE* do Arduino, assim é possível visualizar o *timestamp* e os dados que foram enviados pelo Arduino, e recebidos pelo ESP-32, coletou-se uma amostra de aproximadamente 10 segundos, e, através da análise da amostra obteve-se os resultados encontrados na Tabela 3.

Tabela 3 – Velocidade (*step*) e frequência de transmissão de um único dado entre os microcontroladores

	ARDUINO	ESP-32	(UNIDADE)
STEP MÁXIMO	1	11	UN.
STEP MÉDIO	1	6,53	UN.
STEP MÍNIMO	1	0,0	UN.
FREQUÊNCIA MÉDIA	2402	366	Hz
APROXIMADA			
INTERVALO APROXIMADO DE	13	15	s
TEMPO DA AMOSTRA			

Fonte: Elaborado pelo Autor

O *step* mostra a diferença entre dois valores lidos em relação ao tempo, como a geração de dados é contínua e linear, sabe-se que os dados devem seguir a ordem gerada pelo Arduino (1,2,3,...,1022,1023,1022,1021,...,3,2,1,2,3,...), assim, pode-se

calcular a perda de pacotes no ESP-32 medindo a diferença entre cada *step*. Nota-se que o *step* máximo no ESP-32 foi de 11, ou seja, a cada 11 dados gerados pelo Arduino, 1 é lido pelo ESP-32 através do protocolo. Nota-se também que em algum momento obteve-se um *step* de 0, isso nos diz que o ESP-32 gerou um valor repetido, o que pode vir a acontecer quando o ciclo de processamento do ESP-32 é mais rápido do que um novo envio de dados por parte do Arduino. A média de frequência de geração de dados do Arduino na condição descrita para o teste foi de 2402 Hz, enquanto a do ESP-32 foi de 366 Hz, isso deve-se ao tempo de espera de requisitar e receber o dado através do protocolo I2C.

Realizou-se então uma segunda análise (Tabela 4), agora com um pacote de 11 dados ao invés de apenas 1, que é uma condição mais aproximada do que será utilizado na realidade.

Tabela 4 – Velocidade (*step*) e frequência de transmissão de pacotes entre os microcontroladores

	ARDUINO	ESP-32	(UNIDADE)
STEP MÁXIMO	1	2	UN.
STEP MÉDIO	1	1,03	UN.
STEP MÍNIMO	1	0,0	UN.
FREQUÊNCIA MÉDIA	933	905	Hz
APROXIMADA DE DADOS			
FREQUÊNCIA MÉDIA	72	70	Hz
APROXIMADA POR PACOTE			
INTERVALO APROXIMADO DE	9	15	s
TEMPO DA AMOSTRA			

Fonte: Elaborado pelo Autor

Percebe-se agora que a frequência média de dados gerados pelo Arduino foi reduzida de 2402 para 933, porém, a maior quantidade de dados enviados reduziu a quantidade de dados perdidas pelo ESP-32, funcionando com quase 100% de aproveitamento visto que o *step* médio é de aproximadamente 1, e, com uma frequência próxima de 70 Hz por pacote, tem-se um resultado de transferência de dados entre os dois microcontroladores suficiente para uma boa telemetria.

Agora que realizou-se os testes de velocidade de transferência de dados entre os dois microcontroladores, o próximo passo é realizar testes a partir do protocolo CAN e com a leitura dos instrumentos conectados diretamente ao Arduino para que

seja possível visualizar a velocidade de transferência se conectarmos toda a interface veicular à interface *back-end*. O objetivo dessa parte do teste é que seja ao menos possível emparelhar a frequência do Arduino com a frequência do registrador de dados da injeção eletrônica, ou seja, atingir ao menos 50 Hz. A princípio enviou-se dados da suspensão coletados pelo Arduino para a injeção eletrônica para que fosse possível visualizar o correto armazenamento do mesmo, conforme a Figura 21.

Figura 21 – Variação dos dados gravados pela injeção eletrônica

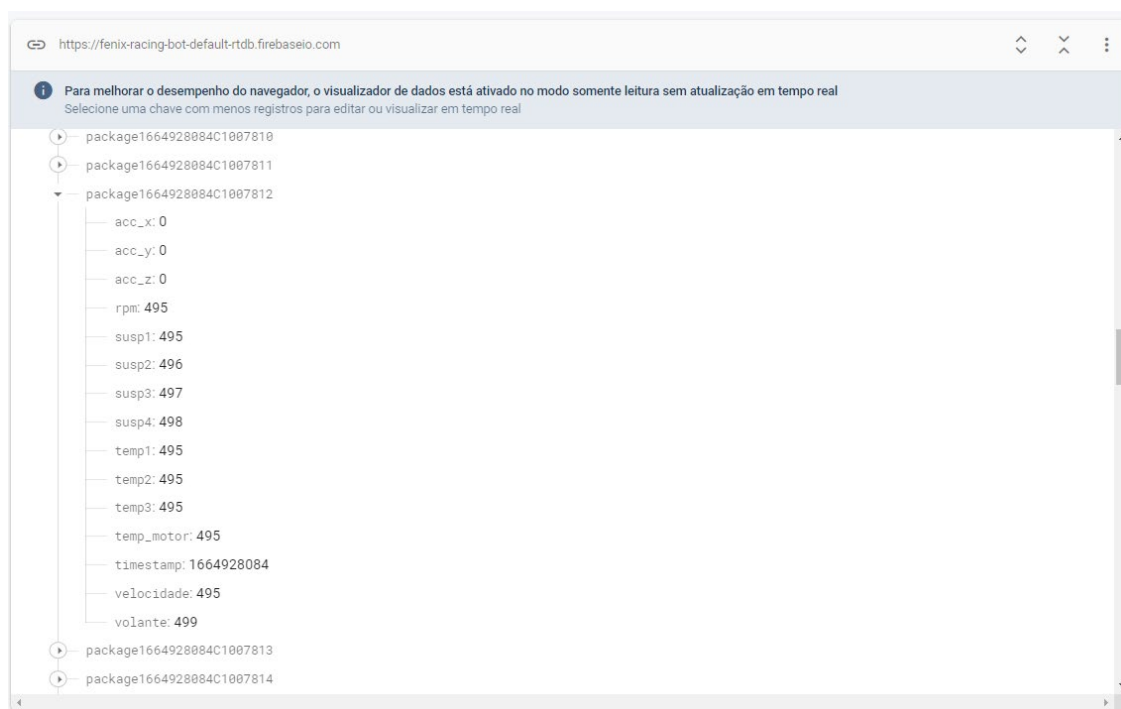
	A	B	C	S	T	U	V	W
1	MS3 Format 0566.05 : MS3 1.5.1 release 20171006 16:30BST (c) JSM/KC *****							
2	Capture Date: Thu Oct 07 16:36:24 BRT 2021							
3	Time	SecL	RPM	Sensor 01	Sensor 02	Sensor 03	Sensor 04	Sensor 05
4	s	s	RPM					
2169	43.446	171	3455	926.1	448.8	335.9	319.6	281.0
2170	43.466	171	3514	926.1	447.6	335.9	332.9	280.5
2171	43.486	171	3478	926.1	447.1	335.9	338.6	280.3
2172	43.506	171	3534	926.1	446.2	335.9	348.3	277.8
2173	43.526	171	3549	926.1	445.0	339.5	356.3	275.9
2174	43.547	171	3431	926.1	443.5	345.6	363.0	273.8
2175	43.567	171	3406	926.1	442.4	350.5	368.5	271.7
2176	43.587	171	3453	926.2	440.9	354.5	373.1	270.0
2177	43.607	171	3381	926.4	439.1	357.7	376.9	268.8
2178	43.627	171	3364	926.6	437.6	360.3	380.1	268.2
2179	43.648	171	3561	927.0	432.8	362.6	384.9	267.7
2180	43.667	171	3538	927.3	429.0	364.5	388.9	266.0
2181	43.687	171	3549	927.4	427.3	365.3	390.6	264.6
2182	43.708	171	3638	927.7	424.8	366.6	393.5	262.3
2183	43.728	171	3634	928.1	423.1	367.5	396.0	260.4
2184	43.748	171	3708	928.3	421.6	368.3	398.0	257.7
2185	43.768	171	3708	928.5	420.4	369.0	400.2	255.6
2186	43.788	171	3739	928.7	419.5	369.7	402.3	253.9
2187	43.808	171	3735	928.9	418.7	370.3	404.0	252.7
2188	43.828	171	3704	929.1	418.1	370.6	406.9	251.7
2189	43.848	171	3719	929.1	417.6	370.8	410.5	250.9
2190	43.868	171	3763	929.1	417.4	370.9	412.0	250.4
2191	43.888	171	3819	929.1	417.0	371.1	414.6	249.5
2192	43.908	171	3878	929.1	419.0	370.9	418.0	248.7
2193	43.928	171	3896	929.1	420.7	370.9	420.8	248.1
2194	43.948	172	3899	929.1	422.0	370.9	423.1	247.6
2195	43.968	172	3948	929.1	423.3	370.3	427.3	247.2
2196	43.989	172	4033	929.1	424.3	369.7	430.7	246.7
2197	44.008	172	4086	929.1	425.1	369.3	433.5	246.3
2198	44.029	172	4127	929.1	425.5	368.9	437.5	245.9

Fonte: Elaborado pelo autor

As colunas referenciadas como Sensor 02, Sensor 03, Sensor 04 e Sensor 05 representam o curso da suspensão frontal esquerda, frontal direita, traseira esquerda e traseira direita respectivamente, junto a alguns outros sinais que não são o foco do projeto. É possível visualizar que os dados são armazenados corretamente e que a frequência mínima de 50 Hz foi atingida já que há variação de uma informação a outra a cada *step* conforme pode ser visualizado nas colunas destacadas da Figura. Para a leitura dos dados que chegaram ao Arduino através do protocolo CAN, utilizou-se como teste o envio do sinal de RPM do motor, mostrando uma leitura bem sucedida através do monitor serial do Arduino e do ESP-32.

Com esse passo concluído, faltava agora realizar um teste com o sistema completo através do envio de dados pelo Wifi para o servidor na nuvem (*firebase*), assim, com o sistema ligado e conectado, enviou-se alguns dados com o intuito de comprovar o funcionamento do código inserido no ESP-32, a Figura 22 mostra os dados alocados dentro do *firebase*:

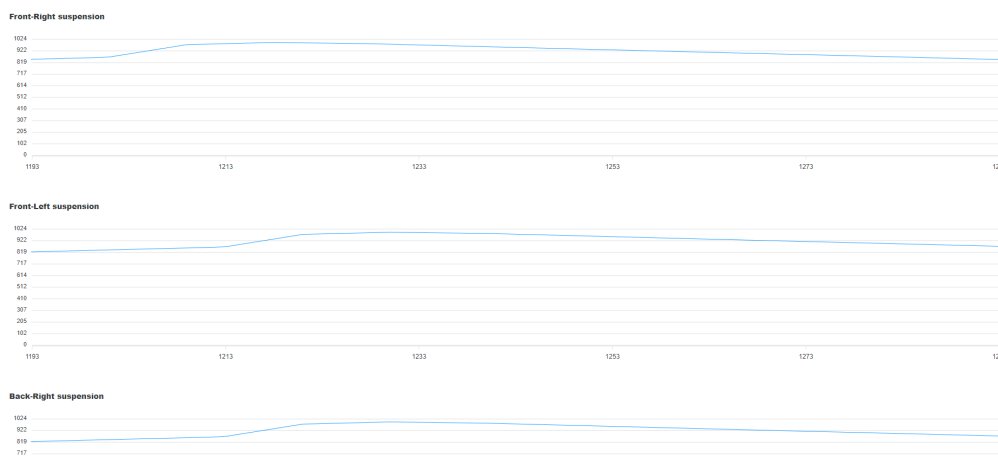
Figura 22 – Dados armazenados no servidor *firebase*



Fonte: Elaborado pelo autor

Com o sistema de envio de dados funcionando, abriu-se a interface *web* com o objetivo de visualizar de forma gráfica os dados que foram enviados para o servidor em tempo real, conforme Figura 23:

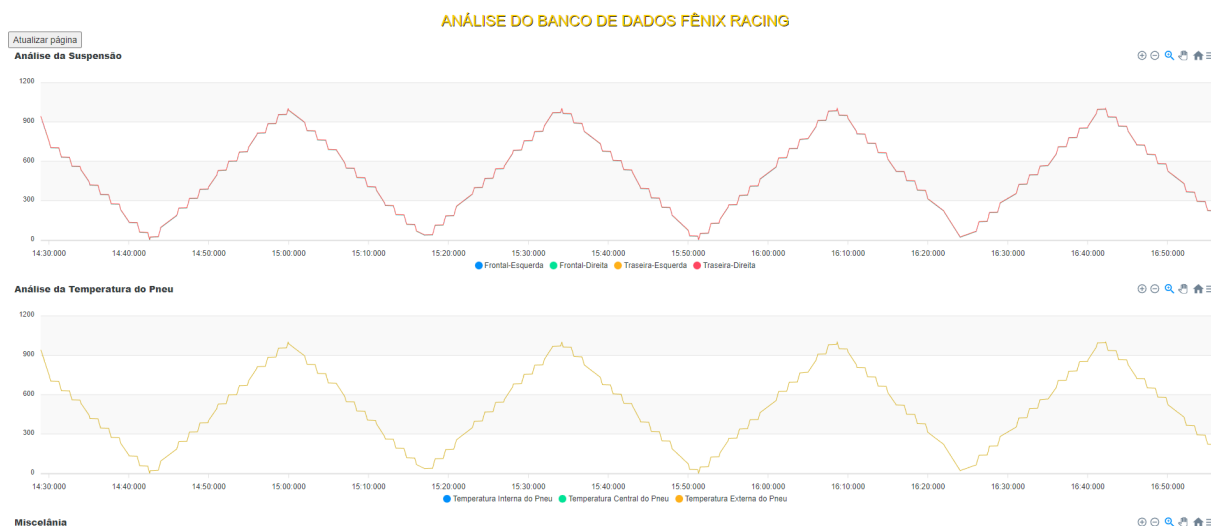
Figura 23 – Visualização em tempo real dos dados no servidor:



Fonte: Elaborado pelo autor

Ao analisar os dados, notou-se pequenas elevações nas curvas que deveriam ser geradas e enviadas pelo sistema, para que a análise fosse feita com uma maior intervalo de tempo, utilizou-se a página de análise do banco de dados, conforme a Figura 24:

Figura 24 – Visualização completa dos dados no servidor



Fonte: Elaborado pelo autor

Através dessa interface nota-se que a curva gráfica possui “dentes” acentuados, isso deve-se a baixa frequência na aquisição de dados por parte do servidor, assim, confirma-se a existência de um “gargalo” no envio de dados do ESP-32 para o servidor *firebase*. Essa perda de informações deve-se a maneira que os dados são enviados, visto que para cada pacote de dados, a biblioteca responsável pelo gerenciamento das informações e autenticação do banco de dados abre uma nova sessão com o *firebase*, o que pode levar alguns milissegundos a mais que são importantes para os dados recebidos do Arduino, assim, para minimizar os efeitos causados pelo *request* do servidor, utilizou-se um *buffer* no próprio ESP-32, capaz de armazenar até 10 pacotes de dados para enviá-los de uma vez só, diminuindo bastante o *delay* de envio e perda de pacotes no processo. O efeito final foi um aumento na frequência de dados de 4 pacotes por segundo (em média) para 34 pacotes por segundo (em média), para esse cálculo, analisou-se um intervalo de aproximadamente 6 minutos de dados e realizou-se a média de pacotes de acordo com o *timestamp*, por segundo.

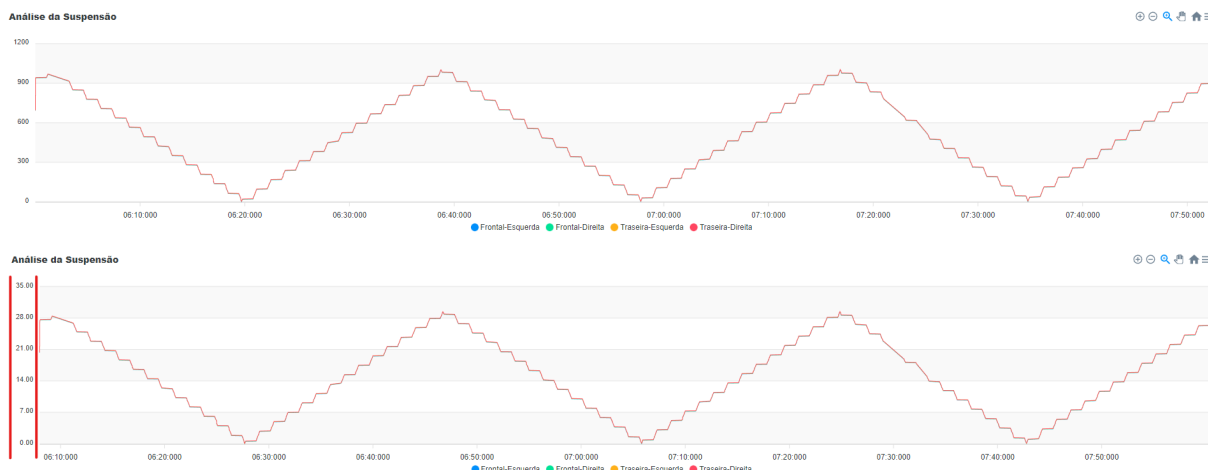
Nota-se que, mesmo com a utilização do sistema de *buffer*, ainda não alcançou-se a frequência ideal do sistema (50 Hz para igualar com a injeção eletrônica), mesmo

com o aumento do *buffer* para 20 pacotes. Analisou-se novamente o código, agora com o sistema completo, e constatou-se que, devido ao *loop* completo de envio de dados, a frequência de pacotes processados pelo ESP-32, era de 34 pacotes por segundo (em média), assim, fica evidente que o “gargalo” não é mais o envio dos dados ao servidor, e sim a coleta e processamento dos pacotes por parte do ESP-32. O microcontrolador ESP-32 com o código em sua última versão (*buffer*) possui uma relação de aproximadamente 1 ciclo para 2 do Arduino, ou seja, a cada 2 ciclos do Arduino (coleta, processamento e envio de dados), o ESP-32 realiza 1, o que diminui sua frequência média aproximada por pacote de 70 (conforme mostrado na Tabela 4), para 35, valor muito próximo do alcançado no banco de dados *firebase*. Devido a estratégia de utilização de um *buffer*, um problema que ocorre na visualização final dos dados é o *timestamp* gerado que, por causa da junção dos pacotes em um único envio, acabam recebendo o mesmo “tempo de processamento”, isso faz com que um pequeno ajuste de tempo seja feito apenas na visualização gráfica dos dados na interface *web*, assim, dá-se a impressão que a frequência continua baixa, porém o que ocorre é uma variação de microssegundos entre um pacote e outro (impedindo que exista dois valores diferentes para uma mesma marcação de tempo).

Há ainda a possibilidade de otimização do código para que o ciclo do ESP-32 dure menos tempo, porém esse processo requer um conhecimento mais aprofundado das linguagens de programação C/C++, que não são o foco deste projeto a princípio.

Após a execução dos testes e da análise dos resultados obtidos, focou-se na melhoria da interface utilizada pelo usuário, incluindo melhorias no redirecionamento da página de *download* para que seja possível baixar todos os dados presentes no banco de dados de forma a ser analisada localmente ou armazenada em *backup* fora dos servidores, além disso, com o auxílio dos times, converteu-se as escalas dos eixos com a unidade que o dado representa, para uma leitura mais prática e rápida por parte dos usuários, conforme pode ser visualizado na Figura 25.

Figura 25 – Comparativo entre os eixos da primeira e da segunda versão



Fonte: Elaborado pelo autor

Nota-se dois gráficos quase idênticos, com diferença na representação do Eixo-Y, onde na primeira versão tem-se os dados de forma bruta, sem conversão de escala, assim lê-se de 0 à 1023 que é o *range* entregue pelo Arduino ao capturar o sinal do sensor. Já na segunda versão do gráfico há uma conversão de valores, para que seja possível representar, em milímetros, a diferença do curso da suspensão, para isso bastou equacionar a distância máxima que o curso da suspensão permite (30 milímetros) e relacioná-lo com o *range* do sensor:

$$D = X \cdot \left(\frac{30}{1023} \right) \quad (1)$$

Onde D representa a distância em milímetros e X o valor digital processado pelo Arduino ao capturar a medição do sensor.

Por fim, realizou-se a documentação do código e do sistema para que o mesmo possa ser futuramente desenvolvido e melhorado pelo time que gerencia a parte eletrônica do veículo Fórmula SAE.

4. CONCLUSÕES

Durante o desenvolvimento deste trabalho, notou-se pontos fortes e fracos do desenvolvimento de um sistema de telemetria utilizando o protocolo CAN e Wi-Fi. Através da realização de testes individuais para cada uma das três interfaces citadas no projeto (Veicular, *back-end* e *front-end*) pode-se avaliar com mais profundidade os limites de cada ferramenta utilizada.

O Arduino utilizado para o projeto mostrou-se rápido o suficiente para superar todos os obstáculos apresentados durante o projeto, funcionando como um sistema isolado de coleta de informações o mesmo seria capaz de entregar uma frequência de aquisição de dados muito maior do que a necessária para qualquer instrumento conectado ao veículo, desde sensores de temperatura do motor que necessitam de uma taxa de atualização bem mais modesta, até os sensores de curso da suspensão que é recomendado a utilização de até 200 Hz por instrumento. A comunicação CAN entre o Arduino e a injeção eletrônica também mostra-se eficiente, visto que a velocidade necessária para a comunicação entre o *data logger* disponível na injeção e o microcontrolador superaram os 50 Hz exigidos para que não houvesse perda de dados na comunicação entre os dois.

A comunicação entre o Arduino e o ESP-32 também apresenta bons resultados, com uma média de aproximadamente 70 Hz, acima da frequência de aquisição da injeção eletrônica, por outro lado, nota-se que a utilização de bibliotecas por parte do microcontrolador ESP-32 e o envio desses dados para um servidor em nuvem possui um nível de complexidade maior, o que pode gerar variações na entrega de dados entre a interface *back-end* e a interface *front-end*, entre os fatores que podem dificultar tal comunicação estão a velocidade de conexão da internet 3G/4G roteada pelo celular (Wi-Fi) e a quantidade de informações mantidas em *buffer* dentro da memória do microcontrolador. Além disso, foi possível compreender os limites físicos de memória do ESP-32, visto que se aumentarmos o *buffer* de informações acima de 20 pacotes, corre-se o risco de um *overflow* na memória ram do microcontrolador, o que faz o mesmo passar por um processo de *reset* automático e impedir o envio de informações para o servidor em um curto período de tempo.

A frequência de aquisição de dados final do sistema é aceitável em algumas situações, e não recomendada para outras, como por exemplo, sensores de temperatura, velocidade das rodas, GPS e rotações por minuto do motor se encaixam

dentro dos padrões de atualização alcançados, porém, sensores que requerem uma frequência de atualização mais alta, como o curso e movimento da suspensão, provavelmente não entregarão informações suficientes para uma análise em tempo real, ainda assim, com os resultados alcançados, o aumento na segurança do piloto através do monitoramento em tempo real de pontos chave do veículo FSAE e a otimização dos modelos utilizados como teoria para a execução do projeto podem ser atingidos.

Por último, vale citar pontos que podem ser explorados e otimizados pelo time de desenvolvimento do veículo FSAE, com o objetivo de obter melhores resultados na coleta e apresentação dos dados, como a otimização do código do ESP-32 para uma maior frequência de atualização de dados, a mudança dos microcontroladores utilizados para potencialização do sistema, como a utilização de um *raspberry PI* que possui conexões semelhantes às utilizadas neste projeto, e o desenvolvimento de uma interface *front-end* com mais ferramentas de análise gráfica, contando com curvas calculadas pelos próprios participantes do projeto que demonstrem a informação necessária para aprimorar o projeto do veículo de Fórmula SAE nos anos posteriores.

5. REFERÊNCIAS BIBLIOGRÁFICAS

APEX CHARTS | Synchronized Charts guide. Disponível em: <<https://apexcharts.com/docs/chart-types/synchronized-charts/>>. Acesso em: 10 out. 2022.

ARDUINO | Arduino UNO R3 Documentation . Disponível em: <<https://docs.Arduino.cc/hardware/uno-rev3>>. Acesso em: 10 out. 2022

BOSCH, R. CAN Specification 2.0, Stuttgart, 1991

EFI ANALYTICS | MegaLogViewer Documentation. Disponível em: <<http://www.tunerstudio.com/>> Acesso em: 29 out. 2022

FILIFELOP. Confira o Módulo WiFi ESP32 Bluetooth. Disponível em: <<https://www.filieflop.com/produto/modulo-wifi-esp32-bluetooth/>>.

FIREBASE | Get Started with Firebase Realtime Database for C++. Disponível em: <<https://firebase.google.com/docs/database/cpp/start>>. Acesso em: 10 out. 2022.

GITHUB. About GitHub Pages. Disponível em: <<https://docs.github.com/en/pages/getting-started-with-github-pages/about-github-pages>>. Acesso em: 10 out. 2022.

I2C PROTOCOL. UM10204 I2C-bus specification and user manual - Rev. Abril 2014. Disponível em: <<https://www.nxp.com/docs/en/user-guide/UM10204.pdf>>. Acesso em: 10 out. 2022

JAVASCRIPT. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>>. Acesso em: 10 out. 2022

MURRAY, J. MEGASQUIRT MS3X/V3.57 Hardware Manual Megasquirt-3 Product Range MS3 1.5.x Última atualização em: 01/08/2018. Disponível em <http://www.msextra.com/doc/pdf/MS3XV357_Hardware-1.5.pdf>. Acesso em: 10 out. 2022

SANDEEP, M. (2022) sandeepmistry/Arduino-CAN - An Arduino library for sending and receiving data using CAN bus. Disponível em <<https://github.com/sandeepmistry/Arduino-CAN>> Acesso em: 10 out. 2022

SAE. SAE International's Collegiate Design Series (CDS) Competitions. 2022. Disponível em: <<https://www.sae.org/attend/student-events>>. Acesso em: 10 out. 2022.

SAE. Fórmula SAE BRASIL | SAE BRASIL. Disponível em: <<https://saebrasil.org.br/programas-estudantis/Fórmula-sae-brasil/>>. Acesso em: 10 out. 2022.

SANTOS, ALLISON A D Comunicação CAN transmitida por PLC aplicada em veículo de fórmula SAE. Universidade Estadual Paulista (Unesp), 2021. Disponível em: <<http://hdl.handle.net/11449/215611>>. Acesso em: 10 out. 2022.

SANTOS, RUI ESP32: Getting Started with Firebase (Realtime Database) | Random Nerd Tutorials. Disponível em: <<https://randomnerdtutorials.com/esp32-firebase-realtime-database/>>. Acesso em: 10 out. 2022.

SANTOS, SARA ESP32 I2C Communication: Set Pins, Multiple Bus Interfaces and Peripherals | Random Nerd Tutorials. Disponível em: <<https://randomnerdtutorials.com/esp32-i2c-communication-Arduino-ide/>>. Acesso em: 10 out. 2022

SEGERS, J. Analysis Techniques for Racecar Data Acquisition. 2. ed. Warrendale: SAE International, 2014.

TEXAS INSTRUMENTS | Introduction to the Controller Area Network, 2016. Disponível em: <<https://www.ti.com/lit/an/sloa101b/sloa101b.pdf>> Acesso em: 29 out. 2022

WIKICOMMONS, CAN-Frame from MIT, 2020. Disponível em: <<https://commons.wikimedia.org/w/index.php?curid=31571812>> Acesso em: 29 out. 2022

6. ANEXO I – Código Arduino

CÓDIGO DO ARDUINO

```
// CONFIGURAÇÕES GERAIS -->
int telemetry_mode = 1; // 0 - Não envia dados para o ESP32 | 1 - Envia dados para o ESP32
int computer_mode = 1; // 0 - Funcionando com o CAN | 1 - Funcionando com dados aleatórios do computador (modo de teste de comunicação)
// FIM DAS CONFIGURAÇÕES GERAIS

#include <mcp_can.h> // Biblioteca para o CAN
#include <SPI.h> // Biblioteca para o protocolo (?)
#include <Wire.h> // Biblioteca para conexão (?)
#define MAX_MEM_ALLOCATED 0 // Tamanho do meu "buffer", no caso, 10 unidades.

int pot1,pot2,pot3,pot4,pot5,ext,pressao; // Variáveis de entrada de dados
int can1,can2,can3,can4,can5,can6,can7,can8,can9;
int onda_bonita = 0;
int onda_bonita_flag = 0;
const int SPI_CS_PIN = 53; // Pino para o CAN
byte data_byte[8] = {1, 1, 1, 1, 1, 1, 1, 1}; // Declarando o primeiro byte de dados
byte data_byte2[8] = {1, 1, 1, 1, 1, 1, 1, 1}; // Declarando o segundo byte de dados
byte data_byte3[8] = {1, 1, 1, 1, 1, 1, 1, 1}; // Declarando o terceiro byte de dados

MCP_CAN CAN(SPI_CS_PIN); // Set CS to pin 10

// Início do código do Pombo ----->>

int susp1 = 0; // Curso da suspensão frente-esquerda
int susp2 = 0; // Curso da suspensão frente-direita
int susp3 = 0; // Curso da suspensão traseira-esquerda
int susp4 = 0; // Curso da suspensão traseira-direita
int temp1 = 0; // Sensor de temperatura interno do pneu
int temp2 = 0; // Sensor de temperatura intermediário do pneu
int temp3 = 0; // Sensor de temperatura externo do pneu
int rpm = 0; // Sensor de RPM
int velocidade = 0; // Sensor de velocidade
int temp_motor = 0; // Sensor de temperatura do motor
int volante = 0; // Sensor de extensão (?) do volante
int acc_x = 0; // Acelerômetro em X
int acc_y = 0; // Acelerômetro em Y
int acc_z = 0; // Acelerômetro em Z
long timestamp = 0; // Timestamp

int contador_de_ciclos = 0; // Conta os ciclos para enviar os dados ao ESP-32

// Fim do código do Pombo ----->>

void setup()
{
  Serial.begin(115200); // Iniciando a leitura/escrita serial em velocidade de 115200 bits por segundo.

  if(computer_mode == 0) // Só funciona se o modo computador estiver desativado!
  {
    // Iniciando o Leitor CAN (MCP2515) em 8 MHz com uma taxa de transferência de dados de 500kb/s. Filtros desabilitados.
    if(CAN.begin(MCP_ANY, CAN_500KBPS, MCP_8MHZ) == CAN_OK)
    {
      Serial.println("MCP2515 Initialized Successfully!"); // Mensagem dizendo que a conexão foi realizada com sucesso!
    }
    else
    {
      Serial.println("Error Initializing MCP2515..."); // Erro ao identificar o MCP2515 (CAN)!
    }
    CAN.setMode(MCP_NORMAL); // Colocando o CAN em modo "normal" para que as mensagens possam ser transmitidas
  }

  // Início do código do Pombo ----->>
  if(telemetry_mode == 1)
  {
    Wire.begin(1); // PROTOCOLO I2C SLAVE NUMERO 1
    Wire.onRequest(requestEvent); // EVENTO PARA ENVIO QUANDO REQUISITADO PELO ESP32
  }

  //Zerando variaveis e evitando lixo.
```

```

for(int idx = 0; idx < MAX_MEM_ALLOCATED; idx++)
{
    susp1 = 1000;
    susp2 = 1000;
    susp3 = 1000;
    susp4 = 1000;
    temp1 = 1000;
    temp2 = 1000;
    temp3 = 1000;
    rpm = 1000;
    velocidade = 1000;
    temp_motor = 1000;
    volante = 1000;
    acc_x = 1000;
    acc_y = 1000;
    acc_z = 1000;
    timestamp = 1000;
}

// Fim do código do Pombo -----
}

void loop()
{
    if(computer_mode == 0)
    {
        // Leitura de valores na entrada do Arduino MEGA:
        pot1 = analogRead(10); // Lê o pino analógico 10: Suspensão frontal esquerda
        pot2 = analogRead(11); // Lê o pino analógico 11: Suspensão frontal direita
        pot3 = analogRead(12); // Lê o pino analógico 12: Suspensão traseira esquerda
        pot4 = analogRead(13); // Lê o pino analógico 13: Suspensão traseira direita
        pot5 = analogRead(14); // Lê o pino analógico 14: Volante
        pressao = analogRead(9); // Lê o pino analógico 09: Sensor de pressão (?)

        // Mostra na tela do computador os dados lidos:
        Serial.println(pot1); // Mostra a leitura do pino analógico 10
        Serial.println(pot2); // Mostra a leitura do pino analógico 11
        Serial.println(pot3); // Mostra a leitura do pino analógico 12
        Serial.println(pot4); // Mostra a leitura do pino analógico 13
        Serial.println(pot5); // Mostra a leitura do pino analógico 14
        Serial.println(pressao); // Mostra a leitura do pino analógico 09

        // Primeiro pacote de dados:
        data_byte[0] = highByte(pot1); //Conversão hexadecimal pino analógico 10
        data_byte[1] = lowByte(pot1); //Conversão hexadecimal pino analógico 10
        data_byte[2] = highByte(pot2); //Conversão hexadecimal pino analógico 11
        data_byte[3] = lowByte(pot2); //Conversão hexadecimal pino analógico 11
        data_byte[4] = highByte(pot3); //Conversão hexadecimal pino analógico 12
        data_byte[5] = lowByte(pot3); //Conversão hexadecimal pino analógico 12
        data_byte[6] = highByte(pot4); //Conversão hexadecimal pino analógico 13
        data_byte[7] = lowByte(pot4); //Conversão hexadecimal pino analógico 13

        // Segundo pacote de dados:
        data_byte2[0] = highByte(pot5); //Conversão hexadecimal pino analógico 14
        data_byte2[1] = lowByte(pot5); //Conversão hexadecimal pino analógico 14
        data_byte2[2] = highByte(pressao); //Conversão hexadecimal pino analógico 09
        data_byte2[3] = lowByte(pressao); //Conversão hexadecimal pino analógico 09

        // Envio dos dados pelo protocolo CAN "Standard", tamanho de 8 bytes com o array "data_byte". ID = 0x100 (?)
        byte sndStat = CAN.sendMsgBuf(0x01, 0, 8, data_byte);

        // Verifica se o CAN realmente conseguiu enviar o pacote 1
        if(sndStat == CAN_OK){
            Serial.println("Pacote 1 enviado com sucesso!");
        } else {
            Serial.println("Erro enviando pacote 1");
        }
    }

    // EDITAR CÓDIGO DO PROTOCOLO CAN AQUI ----->>>

    can1 = 0; // Lê através do CAN o sensor de temperatura interno do pneu
    can2 = 0; // Lê através do CAN o sensor de temperatura médio do pneu
    can3 = 0; // Lê através do CAN o sensor de temperatura externo do pneu
    can4 = 0; // Lê através do CAN o sensor de RPM
    can5 = 0; // Lê através do CAN o sensor de velocidade
    can6 = 0; // Lê através do CAN o sensor de Temperatura do motor

```



```

can7 = 0; // Lê através do CAN o acelerômetro X
can8 = 0; // Lê através do CAN o acelerômetro Y
can9 = 0; // Lê através do CAN o acelerômetro Z

// Envio dos dados pelo protocolo CAN "Standard", tamanho de 8 bytes com o array "data_byte". ID = 0x200 (?)
byte sndStat1 = CAN.sendMsgBuf(0x002, 0, 8, data_byte2);

// Verifica se o CAN realmente conseguiu enviar o pacote 2
if(sndStat == CAN_OK){
  Serial.println("Pacote 2 enviado com sucesso!");
} else {
  Serial.println("Erro enviando pacote 2");
}
}

// Início do código do Pombo ----->>

if(computer_mode == 1)
{
  if(onda_bonita < 1)
  {
    onda_bonita_flag = 0;
  }
  if(onda_bonita > 1000)
  {
    onda_bonita_flag = 1;
  }

  //Serial.println("teste");
  //Serial.println(onda_bonita);
  pot1 = onda_bonita; // Curso da suspensão frente-esquerda
  pot2 = onda_bonita+1; // Curso da suspensão frente-direita
  pot3 = onda_bonita+2; // Curso da suspensão traseira-esquerda
  pot4 = onda_bonita+3; // Curso da suspensão traseira-direita
  pot5 = onda_bonita+4; // Volante
  can1 = onda_bonita; // Sensor de temperatura interno do pneu
  can2 = onda_bonita; // Sensor de temperatura intermediário do pneu
  can3 = onda_bonita; // Sensor de temperatura externo do pneu
  can4 = onda_bonita; // Sensor de RPM
  can5 = onda_bonita; // Sensor de velocidade
  can6 = onda_bonita; // Sensor de temperatura do motor
  can7 = onda_bonita; // Acelerômetro em X
  can8 = onda_bonita; // Acelerômetro em Y
  can9 = onda_bonita; // Acelerômetro em Z
  timestamp = onda_bonita; // Timestamp (deixaremos para depois ok?)

  if(onda_bonita_flag == 0)
  {
    onda_bonita++;
  }
  if(onda_bonita_flag == 1)
  {
    onda_bonita--;
  }
}

if(telemetry_mode == 1)
{
  // ATENÇÃO: ESTOU SOMANDO 1000 PARA DEPOIS SUBTRAIR PARA QUE TODAS AS VARIÁVEIS TENHAM 4 BYTES
  OK?
  susp1 = pot1 + 1000; // Curso da suspensão frente-esquerda
  susp2 = pot2 + 1000; // Curso da suspensão frente-direita
  susp3 = pot3 + 1000; // Curso da suspensão traseira-esquerda
  susp4 = pot4 + 1000; // Curso da suspensão traseira-direita
  temp1 = can1 + 1000; // Sensor de temperatura interno do pneu <<$
  temp2 = can2 + 1000; // Sensor de temperatura intermediário do pneu <<$
  temp3 = can3 + 1000; // Sensor de temperatura externo do pneu <<$
  rpm = can4 + 1000; // Sensor de RPM <<$
  velocidade = can5 + 1000; // Sensor de velocidade <<$
  temp_motor = can6 + 1000; // Sensor de temperatura do motor <<$
  volante = pot5 + 1000; // Sensor de extersão (?) do volante
  acc_x = can7 + 1000; // Acelerômetro em X <<$
  acc_y = can8 + 1000; // Acelerômetro em Y <<$
  acc_z = can9 + 1000; // Acelerômetro em Z <<$

```

```

//timestamp[contador_de_ciclos-1000] = 0 + 1000; // Timestamp (o ESP32 dá conta sozinho, não precisa enviar, mas como já
montei o pacote assim, deixa aqui!)
Serial.print("Susp1: ");
Serial.println(susp1-1000);
Serial.print("Susp2: ");
Serial.println(susp2-1000);
Serial.print("Susp3: ");
Serial.println(susp3-1000);
Serial.print("Susp4: ");
Serial.println(susp4-1000);
Serial.print("Temp1: ");
Serial.println(temp1-1000);
Serial.print("Temp2: ");
Serial.println(temp2-1000);
Serial.print("Temp3: ");
Serial.println(temp3-1000);
Serial.print("rpm: ");
Serial.println(temp1-1000);
Serial.print("velocidade: ");
Serial.println(temp2-1000);
Serial.print("temp_motor: ");
Serial.println(temp3-1000);
Serial.print("acc_x: ");
Serial.println(temp1-1000);
Serial.print("acc_y: ");
Serial.println(temp2-1000);
Serial.print("acc_z: ");
Serial.println(temp3-1000);
}
}

void requestEvent() {
Serial.println("***Request***");
// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(susp1)); // 1º byte
Wire.write(lowByte(susp1)); // 2º byte

// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(susp2)); // 1º byte
Wire.write(lowByte(susp2)); // 2º byte

// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(susp3)); // 1º byte
Wire.write(lowByte(susp3)); // 2º byte

// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(susp4)); // 1º byte
Wire.write(lowByte(susp4)); // 2º byte

// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(temp1)); // 1º byte
Wire.write(lowByte(temp1)); // 2º byte
// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(temp2)); // 1º byte
Wire.write(lowByte(temp2)); // 2º byte
// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(temp3)); // 1º byte
Wire.write(lowByte(temp3)); // 2º byte
// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(rpm)); // 1º byte
Wire.write(lowByte(rpm)); // 2º byte
// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(velocidade)); // 1º byte
Wire.write(lowByte(velocidade)); // 2º byte
// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(temp_motor)); // 1º byte
Wire.write(lowByte(temp_motor)); // 2º byte
// Pacote de 2 bytes repetidos 10x o que são 20 bytes.
Wire.write(highByte(volante)); // 1º byte
Wire.write(lowByte(volante)); // 2º byte

//Serial.println("Ciclo Enviado!!!");
contador_de_ciclos = 0;
}

```

7. ANEXO II

CÓDIGO DO ESP-32

```
//This example shows how to send data fast and continuously.
#include <WiFi.h>
#include <Wire.h>
#include "time.h"
#include <Firebase_ESP_Client.h>
#include <addons/TokenHelper.h>
#include <addons/RTDBHelper.h>

int telemetry_mode = 1; // 0 - Não envia dados para o firebase | 1 - Envia dados para o firebase

// Insert your network credentials
#define WIFI_SSID "" // <----- NOME DA REDE INTERNET ROTEADA NO CELULAR
#define WIFI_PASSWORD "" // <----- SENHA DA REDE DE INTERNET ROTEADA NO CELULAR
#define API_KEY ""
#define DATABASE_URL "https://fenix-racing-bot-default-rtdb.firebaseio.com/"
#define USER_EMAIL "fenixracing@unesp.br"
#define USER_PASSWORD ""

#define MAX_MEM_ALLOCATED 4 // Tamanho do meu "buffer", no caso, 10 unidades.

const char* ntpServer = "pool.ntp.org";
//Define Firebase Data object
FirebaseData fbdo;

FirebaseAuth auth;
FirebaseConfig config;

unsigned long sendDataPrevMillis = 0;

#define qtd_sensores 16 // Quantidade de sensores (MAIS UM) que o ESP32 vai receber!
#define max_length 5 // Quantidade de caracteres máximos que cada sensor irá enviar!

int conta_string = 0; // Contador utilizado pra transformar a string total em cada dado separado!
int conta_letra = 0;
int i,a,b,c;
String total_value;

long int count = 1000000; // O contador começa em 1 milhão para organizar os números de forma crescente (0 a esquerda não conta)
int susp1 = 0; // Curso da suspensão frente-esquerda
int susp2 = 0; // Curso da suspensão frente-direita
int susp3 = 0; // Curso da suspensão traseira-esquerda
int susp4 = 0; // Curso da suspensão traseira-direita
int temp1 = 0; // Sensor de temperatura interno do pneu
int temp2 = 0; // Sensor de temperatura intermediário do pneu
int temp3 = 0; // Sensor de temperatura externo do pneu
int rpm = 0; // Sensor de RPM
int velocidade = 0; // Sensor de velocidade
int temp_motor = 0; // Sensor de temperatura do motor
int volante = 0; // Sensor de extensão (?) do volante
int acc_x = 0; // Acelerômetro em X
int acc_y = 0; // Acelerômetro em Y
int acc_z = 0; // Acelerômetro em Z
//long timestamp = 0; // TimestampTimestamp
int contador_de_ciclos = 0;
unsigned long timestamp ; // Timestamp (deixaremos para depois ok?)
int watchdog_flag = 0;
unsigned long waiter = 0;
String pacote;
int idx = 0;
int conta_ciclos = 0;

byte one, two;

FirebaseJson json;

unsigned long getTime() {
    time_t now;
    struct tm timeinfo;
    if (!getLocalTime(&timeinfo)) {
```

```

    //Serial.println("Failed to obtain time");
    return(0);
}
time(&now);
return now;
}

void setup()
{

    Serial.begin(115200);
    Wire.begin();

    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
    Serial.print("Connecting to Wi-Fi");
    while (WiFi.status() != WL_CONNECTED)
    {
        Serial.print(".");
        delay(300);
    }
    Serial.println();
    Serial.print("Connected with IP: ");
    Serial.println(WiFi.localIP());
    Serial.println();

    Serial.printf("Firebase Client v%s\n\n", FIREBASE_CLIENT_VERSION);

    config.api_key = API_KEY;
    auth.user.email = USER_EMAIL;
    auth.user.password = USER_PASSWORD;
    config.database_url = DATABASE_URL;
    config.token_status_callback = tokenStatusCallback; //see addons/TokenHelper.h

    Firebase.begin(&config, &auth);

    configTime(0, 0, ntpServer);

    Firebase.reconnectWiFi(true);
    #if defined(ESP8266)
    fbdo.setBSSLBufferSize(512, 2048);
    #endif
}

void loop()
{
    Wire.begin();
    contador_de_ciclos = 0;
    Wire.requestFrom(1, 22, false); // request 220 bytes from peripheral device #1
    while (Wire.available()) { // peripheral may send less than requested
        one = Wire.read(); // 1º Byte
        two = Wire.read(); // 2º Byte
        susp1 = (int)one << 8 | (int)two;

        one = Wire.read(); // 1º Byte
        two = Wire.read(); // 2º Byte
        susp2 = (int)one << 8 | (int)two;

        one = Wire.read(); // 1º Byte
        two = Wire.read(); // 2º Byte
        susp3 = (int)one << 8 | (int)two;

        one = Wire.read(); // 1º Byte
        two = Wire.read(); // 2º Byte
        susp4 = (int)one << 8 | (int)two;

        one = Wire.read(); // 1º Byte
        two = Wire.read(); // 2º Byte
        temp1 = (int)one << 8 | (int)two;

        one = Wire.read(); // 1º Byte
        two = Wire.read(); // 2º Byte
        temp2 = (int)one << 8 | (int)two;

        one = Wire.read(); // 1º Byte
        two = Wire.read(); // 2º Byte

```

```

temp3 = (int)one << 8 | (int)two;

one = Wire.read(); // 1º Byte
two = Wire.read(); // 2º Byte
rpm = (int)one << 8 | (int)two;

one = Wire.read(); // 1º Byte
two = Wire.read(); // 2º Byte
velocidade = (int)one << 8 | (int)two;

one = Wire.read(); // 1º Byte
two = Wire.read(); // 2º Byte
temp_motor = (int)one << 8 | (int)two;

one = Wire.read(); // 1º Byte
two = Wire.read(); // 2º Byte
volante = (int)one << 8 | (int)two;
//Serial.println("Ciclo finalizado!");
Serial.print("Susp1: ");
Serial.println(susp1-1000);
Serial.print("Susp2: ");
Serial.println(susp2-1000);
Serial.print("Susp3: ");
Serial.println(susp3-1000);
Serial.print("Susp4: ");
Serial.println(susp4-1000);
Serial.print("Temp1: ");
Serial.println(temp1-1000);
Serial.print("Temp2: ");
Serial.println(temp2-1000);
Serial.print("Temp3: ");
Serial.println(temp3-1000);
Serial.print("rpm: ");
Serial.println(temp1-1000);
Serial.print("velocidade: ");
Serial.println(temp2-1000);
Serial.print("temp_motor: ");
Serial.println(temp3-1000);
Serial.print("acc_x: ");
Serial.println(temp1-1000);
Serial.print("acc_y: ");
Serial.println(temp2-1000);
Serial.print("acc_z: ");
Serial.println(temp3-1000);

contador_de_ciclos = 0;
if (telemetry_mode == 1)
{
    if (Firebase.ready())
    {
        sendDataPrevMillis = millis();
        if (susp1 == 65535)
        {
            Serial.println("-----");
            Serial.print(contador_de_ciclos);
            Serial.print(" - ");
            Serial.println(susp1);
            Serial.println("Data Loss maybe!");
            Serial.println("-----");
        }
        else
        {
            idx = contador_de_ciclos;
            timestamp = getTime();
            json.set("/package"+String(timestamp)+"C"+String(count)+"/susp1", susp1-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/susp2", susp2-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/susp3", susp3-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/susp4", susp4-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/temp1", temp1-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/temp2", temp2-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/temp3", temp3-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/rpm", rpm-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/velocidade", velocidade-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/temp_motor", temp_motor-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/volante", volante-1000);
            json.set("/package"+String(timestamp)+"C"+String(count)+"/acc_x", 0);
        }
    }
}

```

```

        json.set("/package"+String(timestamp)+"C"+String(count)+"/acc_y", 0);
        json.set("/package"+String(timestamp)+"C"+String(count)+"/acc_z", 0);
        json.set("/package"+String(timestamp)+"C"+String(count)+"/timestamp", timestamp);
        count = count + 1;
        Serial.println("-----");

        if(conta_ciclos >= 20)
        {
            if (timestamp == 0)
            {
                //Serial.printf("Set          json...          %s\n",          Firebase.RTDB.pushJSONAsync(&fbdo,
                "/packages/package"+String(timestamp)+"E"+String(count), &json) ? "ok" : fbdo.errorReason().c_str());
                Serial.println("Possibly Internet Loss!");
            }
            else
            {
                Serial.printf("Set json... %s\n", Firebase.RTDB.updateNodeAsync(&fbdo, "/packages", &json) ? "ok" :
                fbdo.errorReason().c_str());
                //Serial.println("OK");
            }

            json.clear();
            //Firebase.RTDB.setJSON(&fbdo, "/packages/package"+String(count), json);
            conta_ciclos = 0;
        }
        else
        {
            conta_ciclos = conta_ciclos + 1;
        }
    }
}
else
{
    Serial.println("Firebase was not ready yet!");
}
}
}
Wire.end();
}

```

