Universidade Federal do Rio Grande
Instituto de Matemática, Estatística e Física
Programa de Pós-Graduação em Física

John W. B. de Araújo

Estudo, desenvolvimento e implementação de uma rede CAN de Sensores

Rio Grande
2021

John W. B. de Araújo

Estudo, desenvolvimento e implementação de uma rede CAN de Sensores

Dissertação apresentada ao Programa de Pós-Graduação em Física, do Instituto de Matemática Estatística e Física da Universidade Federal do Rio Grande como requisito parcial para a obtenção do título de Mestre em Física.

Orientador: Prof. Dr. Fabricio Ferrari
Coorientador: Prof. Dr. Edson M. Kakuno

Rio Grande
2021

Ficha Catalográfica

A663e Araújo, John W. B. de.
Estudo, desenvolvimento e implementação de uma rede CAN de Sensores / John W. B. de Araújo. – 2021.
299 f.

Dissertação (mestrado) – Universidade Federal do Rio Grande – FURG, Programa de Pós-Graduação em Física, Rio Grande/RS, 2021.
Orientador: Dr. Fabricio Ferrari.
Coorientador: Dr. Edson M. Kakuno.

1. Redes de Sensores 2. CAN 3. Arduino I. Ferrari, Fabricio II. Kakuno, Edson M. III. Título.

CDU 528.8

Catalogação na Fonte: Bibliotecário José Paulo dos Santos CRB 10/2344

Estudo, desenvolvimento e implementação de uma rede de sensores CAN

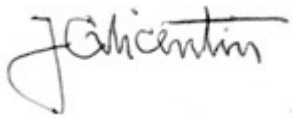
John Welvins Barros de Araújo

Orientador:
Prof. Dr. Fabricio Ferrari

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Física no Curso de Mestrado em Física, como parte dos requisitos necessários à obtenção do título de Mestre em Física.

Aprovada por:

Prof. Dr. Fabricio Ferrari



_____

Prof. Dr. Flavio Cesar Vicentin

_____

Prof. Dr. Marcelo Gonçalves Hönnicke

Rio Grande
Junho de 2021

Aos meus pais, Eusebio e Irani

Agradecimentos

Em primeiro lugar agradeço a minha família pelo suporte financeiro e emocional, que foi fundamental durante a pandemia. Agradeço aos meus orientadores, pela orientação e aconselhamentos. Agradeço aos colegas de pós graduação (e agregados) pelo ótimo tempo de convívio. Agradeço a Universidade Federal do Rio Grande, ao instituto IMEF, ao programa de pós graduação em Física e ao colegiado de professores do programam pelo suporte oferecido durante o mestrado. Agradeço a Universidade Federal do Pampa, em especial ao curso de licenciatura em Física de Bagé-RS, pelo espaço concedido durante pandemia. E agradeço a CAPES, pela bolsa de mestrado concedida.

"*Não convém a gente levantar escândalo de começo,*
só aos poucos é que o escuro é claro."

Guimarães Rosa

Resumo

Uma rede de sensores fornece um caminho de controle e coleta de dados centralizado, além de outras vantagens que podem ser desejadas em experimentações. Entretanto, o tempo de implementação desse tipo de solução pode não ser atrativo para pequenos experimentos. Nesse sentido explorou-se e elaborou-se uma rede acessível, tanto no custo quanto na facilidade de uso, denominada FURG CAN. A CAN - *Controller Area Network* é um padrão estabelecido na indústria, tem um protocolo enxuto e seu custo de implementação é relativamente baixo. A FURG CAN foi elaborada com dispositivos populares, como a plataforma Arduino, possui um desenho modular e de simples implementação. O meio de conexão dos dispositivos é um barramento composto de um par de fio trançado (linhas de dados) e um mais via (GND), opcionalmente pode ser usado uma linha para alimentação, o barramento pode ser feito, por exemplo, em um cabo de Cat5. O protocolo é executado pelo controlador CAN MCP2515, que é gerenciado com as funções da biblioteca desenvolvida pelos autores. A FURG CAN pode operar nas taxas de 500 k bit/s, 250 k bit/s e 125 k bit/s, com os respectivos comprimentos máximos, de barramento, 50 m, 125 m e 275 m (valores teóricos e não testados por falta de cabos). Uma limitação da FURG CAN é o tráfego de mensagens por segundo, que vai de 454,9 mensagens por segundo até 2830,2 mensagens por segundo. A FURG CAN foi implementada em uma aplicação, onde foram aferidos 19 termo-higrômetros digitais, sendo um deles um psicrômetro feito com dois Pt100 AA. O barramento da aplicação foi feito em um cabo Cat6 de aproximadamente 14 m, quatro emissores e receptor. A aplicação durou um pouco mais do que 437 horas, e o maior intervalo ininterrupto de coleta foi de 71 h 41 min 24 s. Foram processados em média 3,2 mensagens por segundo. A FURG CAN mostrou-se robusta e estável durante todo a aplicação, sendo observado uma perda menor do que $0,2\%$.

Abstract

A sensor network provides control and data collection centralized, in addition to other advantages that may be aimed in experiments. The time implementation of this type of solution may not be attractive for small experiments. In this sense, an accessible network (FURG CAN) was explored and developed, in terms of cost and use. The CAN - Controller Area Network is an established standard in industry, has a short protocol and cost effective implementation. The FURG CAN was developed with popular Arduino platform, which has a modular design with easy implementation. The devices are connected in a bus composed of a pair of twisted wire (data lines) and one more via (GND). Optionally a power line can be used. The bus can be made, for example, in a Cat5 cable. The protocol is executed by the CAN controller MCP2515, which is managed with library functions developed by the authors. The FURG CAN operates with three transfer rates, 500 k bit/s, 250 k bit/s e 125 k bit/s, with respect to the maximum bus lengths of 50 m, 125 m, and 275 m (theorectical specified values, since we do not have the cables for testing). A limitation of FURG CAN is the message traffic per second, which ranges from 454,9 messages per second to 2830.2 messages per second. The FURG CAN was implemented in an application, where 19 digital thermo hygrometers were measured, one of them being a psychrometer made with two Pt100 AA. The application bus was made using a Cat6 cable of, approximately, 14 meters, four transmitters, and a receiver. The tests were carried out by a little more than 437 hours. The longest ininterrupted collection interval was 71 h 41 min 24 s, with an average of 3,2 messages per second. The FURG CAN showed to be robust and stable throughout the tests, with a losses smaller than 0.2%.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

As redes de sensores permitem o controle quase simultâneo de dispositivos, tão rápido quanto a conexão permita, e fornecem um mecanismo de centralização dos dados. Ainda que traga vantagens, o tempo dedicado com a implementação pode afastar as redes de sensores de pequenas experimentações, pois não se deseja gastar mais tempo com uma ferramenta do que com o objeto de pesquisa. De modo a tornar este tipo de ferramenta mais atrativa à pequenos experimentos, objetivou-se explorar, apresentar e testar uma alternativa acessível de rede de sensores, que não demande muito tempo, nem muito recurso em sua implementação e uso.

A plataforma de redes de sensores apresentada, denominada de FURG CAN, foi elaborada sobre o protocolo CAN 2.0 (*Controller Area Network*). Ele é composta principalmente pelo o Arduino Nano (Arduino, 2020a), e por um módulo CAN com dois dispositivos específicos para o protocolo: o controlador CAN MCP2515 (Microchip Technology Inc, 2019) e o transceptor CAN TJA1050 (Philips, 2003). A conexão entre os elementos da FURG CAN, é feita com um par trançado e uma via. Por exemplo, de um cabo de rede Cat5, usado em comunicação de computadores do tipo TCP/IP - *Transmission Control Protocol/Internet Protocol*. Optou-se por essa configuração, de rede de sensores, devido à facilidade de obter os materiais e a simplicidade de implementação. Além de pesar a acessibilidade, a escolha do padrão de rede e dos dispositivos que compõem a proposta, também foi pautada na confiança, *i.e.*, na estabilidade ao longo do tempo. O padrão de rede escolhido, a CAN, é utilizada na indústria desde dos anos 1990, e seu funcionamento é discutido na seção 1.3.

A FURG CAN é uma transposição tecnológica, de baixo custo e de código aberto (licença: CC BY-NC 4.0). Em um repositório do grupo (Kaki), pode-se encontrar, uma série de códigos, e exemplos, para nodos da FURG CAN. Incluindo a biblioteca do CI

MCP2515, a MCP2515-*bib*, que foi, juntamente com os demais códigos, construída pelos autores. No repositório também se encontra uma documentação da biblioteca MCP2515-*bib*. A biblioteca, também, está disponível no Apêndice B. A FURG CAN é apresentada no capítulo 2, onde são detalhados os procedimentos de implementação. A caracterização da FURG CAN é apresentada no capítulo 3. Lá são descritos os limites operacionais da FURG CAN e, os testes contra falhas realizados. No capítulo 4 é relatado um exemplo de aplicação, durante o qual a FURG CAN mostrou-se robusta e confiável.

Na continuidade deste capítulo, 1, tem-se uma introdução sucinta sobre as Redes de Sensores 1.1. O Arduino Nano é a unidade de processamento dos elementos da FURG CAN. Por isso uma breve descrição, do Arduino Nano, é feita na seção 1.2. O protocolo CAN é descrito na seção 1.3.

1.1 Rede de Sensores

Uma rede de sensores é um sistema de medição organizado como uma rede de comunicação, e tem por objetivo mensurar e tornar acessível uma ou mais grandezas de interesse. As redes de sensores são constituídas, majoritariamente, de instrumento de medição interligados que, além de coletar os dados, devem ser capazes de processá-las, armazená-las e transmiti-las por qualquer tipo de conexão. No contexto de rede de sensores, o instrumento de medição é chamado de sensor, nodo sensor, nó sensor ou ainda sensor remoto. Neste trabalho será usado o termo nodos, para referir-se genericamente a aparatos conectados em um ponto da rede, e nodo sensor quando este aparato for um instrumento de medição.

Além dos nodos sensores uma rede deve possuir ao menos um nodo capaz de receber as informações transmitidas e dar-lhes um destino final, *e.g.*, salvar em um banco de dados para análise futura, ou transferir para um terminal de monitoramento, esse tipo de nodo é genericamente chamado de nodo *sink* (coletor), ele se comporta como um ralo que suga o fluxo de dados da rede. Se construirmos uma rede de sensores com apenas os dois tipos de nodos descritos, nodos sensores e *sink*, teremos uma rede de monitoramento, que é solução para diversos problemas, como o monitoramento de paramentos ambientais. Entretanto, redes de sensores são construídas de acordo com a aplicação, ou seja, existem diversos tipos de nodos, para as mais diversas funções. Também são comuns nodos atuadores, que realizam algum trabalho sobre o meio, nodos de controle, que gerenciam o funcionamento

dos demais nodos, e nodos de processamento, que realizam processamentos sofisticados sobre os dados coletados na rede.

É interessante utilizar uma rede de sensores quando o evento estudado exige o uso simultâneo (ou quase) de diversos instrumentos de medição e/ou controle, ou ainda no acompanhamento de fenômenos distantes. Uma rede de sensores pode possibilitar a automação do controle de processos. O número máximo de nodos conectados é limitado pela configuração da rede, *i.e.*, pela capacidade individual de seus nodos, pelo protocolo, topologia e meio de propagação utilizados na comunicação. O meio de publicação limita o formato da conexão entre os nodos, afinal suas conexões se dão através do meio de publicação. Os meios físicos utilizados como meio de transmissão (ou publicação) de dados são os fios condutores, a fibra ótica e as ondas de rádio.

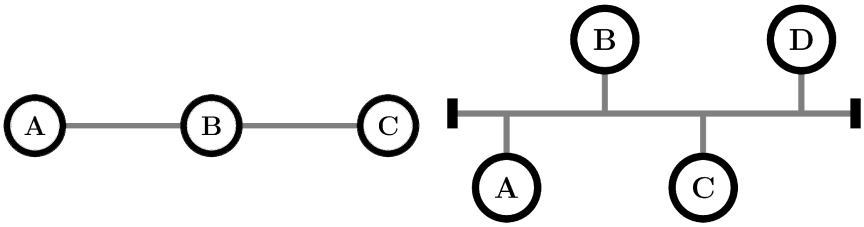
As redes sem fio são bem populares hoje em dia, impulsionadas pelo avanço dos dispositivos portáteis e, mais especificamente no âmbito das redes de sensores, com o advento do IoT - *Internet of things* (Internet das Coisas). É possível encontrar no mercado, dispositivos com interface amigável, que tornam a implementação de uma rede sem fio mais acessível, sobretudo se for usado a família de protocolos IEE 802.11, a mesma utilizada no protocolo sem fio mais popular o Wi-Fi. De qualquer forma, os protocolos usados em comunicações sem fio tendem a ser mais complexos do que os usados em uma rede com cabo, que costuma ser mais estável. Os materiais e dispositivos envolvidos em redes com fibra ótica ainda são pouco acessíveis, em contrapartida, existe um leque muito grande de dispositivos e acessórios para conexão via cabo.

O formato da conexão entre os nodos é conhecido como topologia, ou topologia física, pois diz respeito exclusivamente da conexão física. A forma como o *link* de comunicação entre os nodos é gerenciado (roteamento), é conhecido como topologia lógica: conjunto de regras (protocolos) utilizadas na comunicação entre os nodos. O roteamento de dados ao longo da rede, é limitado tanto pela topologia física quanto pela topologia lógica. Existem protocolos de comunicação que são executados após o roteamento das mensagens, e vão desde de verificações da informação, como encriptação e decriptação, até aplicações de interface direta com o usuário. Em meados da década de 1980 a ISO - *International Organization for Standardization* lançou o Modelo OSI - *Open Systems Inter-connection* (ISO, 1994), que divide os protocolos de comunicação de uma rede em sete camadas. O modelo OSI é bem difundido e serve com referência para criação de redes de comunicação.

1.1.1 Topologia Física

A organização de uma rede é conhecida por topologia, e pode ser dividida em topologia física, e em topologia lógica, que diz respeito a conexão lógica dos nodos, isto é, como o link de comunicação entre os nodos é feito. A topologia lógica consiste em regras de comunicação que sobrepõem a topologia física e ajudam a definir a dinâmica da comunicação entre os nodos. As figuras 1.1 e 1.2, apresentam cinco das topologias físicas mais convencionais, *Point to point* e *Bus* (Figura 1.1), *Ring*, *Star* e *Tree* (Figura 1.2). Há várias outras formas de topologia, inclusive topologias híbridas, que mesclam mais de um tipo de topologia em uma mesma rede.

Em uma rede com topologia ponto a ponto (*Point to Point*) os nodos são conetados apenas com seus primeiros vizinhos. O diagrama **(a) *Point to Point*** da Figura 1.1 apresenta uma rede com três nodos (**A**, **B** e **C**) conectados ponto a ponto. Trocas de mensagens entre dois nodos em uma rede do tipo ponto a ponto, só podem ser diretas quanto os nodos estão diretamente conectados, como os nodos **A** e **B** do diagrama (a) da Figura 1.1, e indireta, quando a mensagem precisa ser re-transmitida por um ou mais nodos da rede. Por exemplo, uma mensagem transmitida do nodo **A** para o nodo **C**, no diagrama (a) da Figura 1.1, deverá ser re-transmitida pelo nodo **B** para o nodo **C**, uma vez que não há conexão direta entre o nodo **A** e o nodo **C**. Esse tipo de topologia não é imune a falhas individuais de nodos, e caso isso ocorra, a rede será no mínimo dividida.

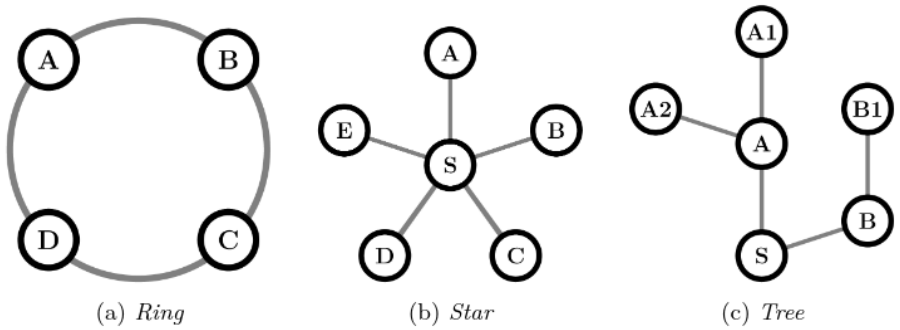


(a) *Point to Point* (b) *Bus*

Figura 1.1: Diagrama das topologias (a) *Point to Point* e (b) *Bus*. Fonte: Autores.

A Figura 1.1 também apresenta um diagrama da topologia barramento, **(b) *Bus***, no qual quatro nodos (**A**, **B**, **C** e **D**) são conectados via um barramento. O retângulos pretos nos extremos, esquerdo e direito, representam dois terminadores, que garantem o casamento de impedância nos extremos. Diferente de uma rede ponto a ponto, em um barramento qualquer nodo conectado pode conversar com qualquer outro, sem a necessidade

de um intermediário. A topologia *Bus* é imune a falhas individuais de nodos, porém podem ocorrer colisões de mensagens no barramento, o que resulta em erros de comunicações.



(a) *Ring* (b) *Star* (c) *Tree*

Figura 1.2: Diagrama das topologias (a) *Ring*, (b) *Star* e (c) *Tree*. Fonte: Autores.

A topologia anel, diagrama **(a)** ***Ring*** da Figura 1.2, tem um funcionamento semelhante à topologia ponto a ponto, a diferença é que os nodos dos extremos são conetados formando um anel. Assim como na topologia ponto a ponto, um nodo só pode estabelecer comunicação direta com seus primeiros vizinhos. Comunicações mais distantes dependem de nodos intermediários. Outra configuração comum é a estrela, diagrama **(b)** ***Star*** da Figura 1.2, onde os nodos são conectados a um nodo central. Na Figura 1.2 o nodo central é o **S**. A topologia *Star*, é muito usadas em redes de computadores, onde o nodo central costuma ser um *hub*. Em redes de sensores, o nodo central é, em geral do tipo *sink*, uma vez que é o único que possui comunicação direta com todos os nodos.

A última topologia citada é apresentada no diagrama **(c)** ***Tree*** da Figura 1.2. Nela os nodos são distribuídos em níveis de acesso diferentes, com relação ao nodo *sink*. Por exemplo, no diagrama (c) da Figura 1.2, o nodo **A** possui comunicação direta com o nodo **S** (*sink*), enquanto que as mensagens dos nodos **A1** e **A2** precisam ser retransmitidas pelo nodo **A** antes de chegar ao nodo *sink*.

1.1.2 Organização dos protocolos de comunicação

Os nodos utilizam diversas regras durante a comunicação, que regulam desde a propagação no meio físico, até a interação do usuário final com a informação. O modelo OSI (*Open Systems Interconnection*) serve com referência para arquitetura de redes de comunicação, nele os protocolos são organizados em sete camadas (ISO, 1994), descritas na Figura 1.3. A camada 7, de aplicação, é responsável pela interface máquina-usuário. A ca-

mada 6, de apresentação, formata os dados da camada de acima para serem transportados. A camada 5, de sessão, gerencia a conexão entre as aplicações nos nodos comunicantes. A camada 4, de transporte, garante a confiabilidade da entrega. A camada 3, de rede, realiza o roteamento. A camada 2, de enlace, gerencia os protocolos entre dispositivos conectados. E, a camada 1, física, que gerencia a transmissão no meio físico.

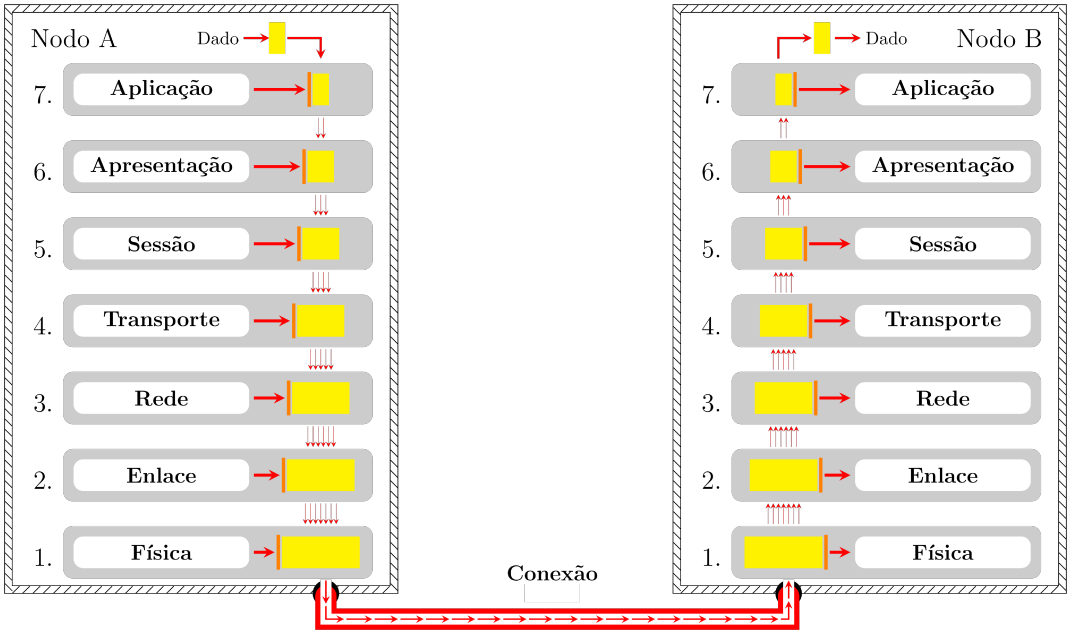


Figura 1.3: Comunicação entre dois nodos através do modelo OSI. Fonte: Autores.

A Figura 1.3 apresenta um diagrama da comunicação entre dois dispositivos através o modelo OSI, o **Nodo A** na esquerda e o **Nodo B** na direita. As setas vermelhas representam o fluxo de dados e as interações sobre os dados nas camadas do modelo OSI. No diagrama o dado se origina no **Nodo A**, passa pela camada 7 **Aplicação**, onde são incorporados parâmetros de controle, na sequência passa para a camada 6 **Apresentação** onde o dado é formatado e também são adicionados novos parâmetros de controle, e assim por diante até que chegue na camada 1 **Física** que gerencia o envio dos dados pelo meio físico. Ao chegar no **Nodo B** o processo ocorre de forma inversa e ao final o dado é disponibilizado para o usuário final ou processo de aplicação específica.

O modelo OSI serve como referência para a organização dos protocolos de comunicação, porém nem todos os dispositivos da rede precisam possuir todas as sete camadas descritas, *e.g.*, dispositivos de roteamento possuem apenas às três camadas inferiores (Física, Enlace

e Rede). Existem padrões de rede de comunicação que não possuem todas as setes camadas do modelo OSI, como o TCP/IP, que possui ao todo quatro camadas: a de Aplicação, a de Transporte, a de Internet e a de Acesso à Rede. Redes do tipo CAN 2.0 possuem apenas às duas primeiras camadas (Física e Enlace) do modelo OSI. A abstração de interação camada a camada é conveniente para o estudo e aplicação dos diferentes modelos de comunicação citados.

1.1.3 Arquitetura de um Nodo Sensor

O nodo sensor é o tipo mais comum em uma rede de sensores e os demais tipos podem ser entendidos como variações dele. Espera-se que um nodo sensor seja capaz de recolher informações do meio, processá-las e enviá-las para o nodo *sink* através da rede. Pode-se dividir a arquitetura do nodo sensor em três subsistemas: subsistema sensor; subsistema de processamento e; subsistema de comunicação. A Figura 1.4, apresenta um diagrama da arquitetura genérica de um nodo sensor, seguindo a divisão em subsistemas.

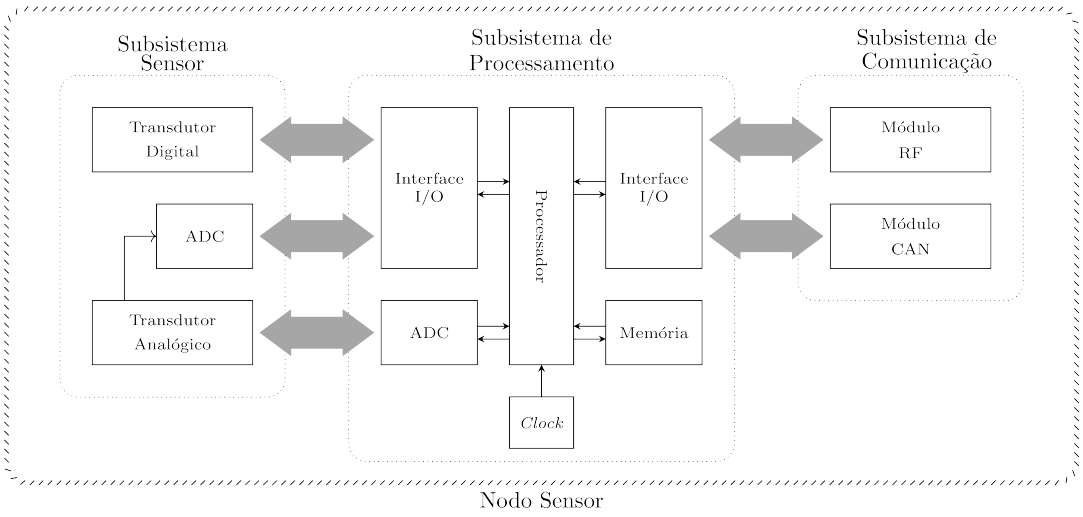


Figura 1.4: Diagrama genérico da arquitetura de um nodo sensor. Fonte: Autores.

Ao centro da Figura 1.4 tem-se o subsistema de processamento, também chamado de unidade de controle, que é responsável por interligar e gerenciar os demais subsistemas do nodo, processar as informações coletadas através do subsistema sensor, e encaminhá-las ao subsistema de comunicação para que possam ser enviadas pela rede. Um subsistema de processamento é composto por um processador central, interface de entrada e saída (I/O

- *Input/output*) e de memória. Ainda pode possuir outros componentes internos, *e.g.*, um ADC - *Analog-to-Digital Converter* como na Figura 1.4. Na Figura 1.4, à esquerda, tem-se o subsistema sensor, que contém os dispositivos sensíveis aos parâmetros de interesse, como transdutores e detectores, podendo conter também conversores, condicionadores de sinal entre outros dispositivos envolvidos no processo de medição.

Na direita da Figura 1.4 tem-se a representação do subsistema de comunicação que é responsável por estabelecer a conexão com os demais nodos da rede. Esse subsistema é normalmente composto por transmissores, receptores, antenas, condicionadores de sinal entre outros. Também pode-se optar por módulos de comunicação que possuem todos os dispositivos necessários para o tipo de comunicação escolhida em uma placa de testes. Os protocolos que utilizam ondas de rádio (sem fio) são bem populares hoje em dia, sendo possível encontrar módulos de comunicação que fornecem uma interface de uso amigável. Entretanto, os protocolos que usam fibra ótica ou cabos, costumam oferecer conexões mais estáveis, produzem menos ruído eletromagnético, além de serem mais eficiente em carregar o sinal através de barreiras físicas, com interfaces entre dois meios. No diagrama da Figura 1.4 são dispostos dois módulos: um RF (Radio Frequência) e outro CAN, que utiliza fios condutores como meio físico de conexão.

Além dos subsistemas apresentados um nodo pode ter outros dispositivos, *e.g.*, atuadores (motores, lâmpadas, etc.), sistemas de alimentação (bateria, placa solar, etc.) e memória externa (cartões SD). A primeira diferença entre nodos sensores e nodos do tipo *sink* é a ausência do subsistema sensor. Porém outros dispositivos são desejados em nodos do tipo *sink*. Por exemplo, pode-se adicionar um módulo de cartão SD e construir um nodo *data-logger*, que é do tipo *sink*, ou um módulo de comunicação que seja capaz de fornecer conexão com a internet, possibilitando que os dados sejam enviados a um servidor *on-line*.

1.1.3.1 Subsistema de Processamento

O subsistema de processamento, ou unidade de controle, é o núcleo do nodo, ele conecta e gerência o funcionamento de todos os outros subsistemas através do seu programa de instruções. Há vários dispositivos comerciais que podem ser usados como subsistema de processamento, por exemplo, os microcontroladores, os FPGA's - *Field Programable Gate Array*, os DISP's - *Digital Signal Processors* e os ASIC's - *Application-Specific Integrated Circuit*. Qualquer que seja o dispositivo escolhido ele deve possuir, pelo menos, uma

unidade de processamento, uma unidade de memória, uma base temporal interna e alguma interface de conexão digital, para conectar com os demais subsistemas e dispositivos do nodo.

Microcontroladores são computadores no formato de CI's, em geral, possuem uma unidade central de processamento, barramentos, memória, um sistema de supervisão, temporizador, um gerador de *clock*, que pode ser externo, e interfaces I/O para comunicação (DARGIE, 2010). Esses dispositivos são cada vez mais empregados, em escala industrial, em diversos aparelhos, *e.g.*, televisões, geladeiras, máquinas de lavar, e aplicações IoT - *Internet of Things*. Além disso, hoje dia, podem ser adquiridos em placas de prototipagem, eletronicamente configuradas para facilitar seu uso, e com carregadores de inicialização (*bootloaders*) cada vez mais sofisticados, que permitem sua programação em linguagens de alto nível.

Devido a sua flexibilidade e acessibilidade, os microcontroladores costumam ser a primeira escolha para compor o subsistema de processamento em redes de sensores. A funcionalidade dos nodos, e até mesmo da rede, pode mudar com o tempo, por exemplo, para melhorar o roteamento ou acompanhar um novo parâmetro. Nestes casos os programas de instruções das unidades de controle devem ser atualizados. Este é mais um ponto que pesa a favor dos microcontroladores, pois, em geral, é mais fácil reprogramar um microcontrolador do que qualquer um dos outros dispositivos listados. Em casos onde a acessibilidade e flexibilidade são irrelevantes, pode-se optar por processadores mais específicos, como os outros citados, que costumam ser energeticamente mais eficientes e computacionalmente mais performáticos. Todavia são mais caros e de implementação mais complexa.

Os DISP's, são dispositivos especificados para o processamento de operações matemáticas complexas, através da aplicação de filtros digitais, que podem fazer, por exemplo, operações de translação de bits e alterações espectrais em conjuntos de bits. Devido a essas características são processadores desejados em aplicações com processamento direto na rede (DARGIE, 2010).

Outra opção recorrente para o subsistema de processamento são os ASIC's (DARGIE, 2010), dispositivos de design flexível, compostos essencialmente por conjuntos de componentes eletrônicos ativos, tipicamente transistores MOS - *Metal-Oxide-Semiconductor*. A construção de um ASIC e tão flexível quanto se queira, e eles entregam ótimo desempenho no que diz respeito ao consumo e poder processamento. Porém uma vez prontos não po-

dem ser reconfigurados, e possuem alto custo de desenvolvimento. O uso de um ASIC em cooperação com outros processadores podem ser bem-vindo, nessa situação eles realização tarefas específicas em algum subsistema, como por exemplo, protocolos de comunicação em um módulo de comunicação. Isso diminui a carga de processamento no processador central, e possibilita uma construção modular do nodos o que facilita a manutenção. Por exemplo, pode-se ter um subsistema de comunicação com um ASIC para realizar as verificações relacionadas ao transporte de informações e outros processamentos, diminuindo o número de instruções executadas pela unidade de controle.

Processadores do tipo FPGA permitem processamento paralelo, possuem grande velocidade de processamento e são mais flexíveis em nível de controle e programação do que um ASIC. Entretanto têm alto custo e alta complexidade de implementação, um FPGA são bem mais complexos do que microcontroladores. Um FPGA é tipicamente composto de componentes lógicos programáveis: entradas LUT - *LooKUp Table*, flip-flop e blocos de saída, células logicas e matriz de interconexões programáveis, além de células I/O também programáveis localizadas ao entrono do núcleo (DARGIE, 2010). Esse tipo de dispositivo é programável de forma eletrônica, em linguagens de descrição de hardware com o suporte do diagrama de circuito, como VHDLe Verilog, em procedimentos que podem levar de milissegundos até alguns minutos, dependendo da tecnologia usada. A programação dos FPGA's são realizadas de forma mais complexa, em relação aos microcontroladores, por isso são mais usados em aplicações específicas e com pouca flexibilidade, como em processamento primário de sinal de vídeo.

1.1.3.2 Subsistema Sensor

O subsistema sensor realiza a interface com o meio, sendo responsável por mensurar as grandezas de interesse. Sua constituição varia de aplicação para aplicação, entretanto o universo de elementos usados costuma ser delimitado por detectores, transdutores analógicos, ADC, e por transdutores digitais (ou sensores digitais), que já possuem conversores internamente.

A depender da literatura o termo sensor pode ser usado para designar um transdutor, porém um transdutor é um dispositivo tecnológico elaborado para realizar medidas, e tem a capacidade de fornecer uma saída de relação especificada com grandeza de entrada (INMETRO, 2012). Um transdutor também pode ser entendido como um dispositivo

que transforma uma quantidade física em outra, nesse texto, vamos considerar somente dispositivos que transformam qualquer grandeza física em uma grandeza elétrica (tensão, corrente, resistência, capacitância ou indutância). Já um sensor é um corpo ou substância que sofre alteração devido a um fenômeno (grandeza de entrada) (INMETRO, 2012), ele não é um dispositivo tecnológico. Um sensor pode até fornecer indicativos empíricos a respeito do fenômeno, mas seu uso exclusivo não basta para inferir valores à grandeza de entrada, *i.e.*, não é possível mensurar uma grandeza usando apenas um sensor. Podemos separar os transdutores em digitais e analógicos. O tipo analógico fornece como saída um sinal contínuo, que é relacionado com a grandeza de entrada através de operações explicitadas para o transdutor, em geral, pelo seu fabricante. O tipo digital, que fornecem uma informação quantizada (*e.g*, em binário) que pode ser codificação a partir de operações especificas para o transdutor. Um transdutor deve possui as informações necessárias para a obtenção adequada do valor medido em relação ao seu sinal de saída, seja ela digital ou analógico.

Dentre os transdutores analógicos aqueles que fornecem uma saída elétrica são os preferidos, pois, o processo de digitalização de sinais elétricos pode ser facilitado pelo uso de um ADC. Simplificadamente, um ADC funciona comparando amostragens da entrada contra uma referência, de modo a encontrar o valor amostrado dentro de um dos intervalos (níveis) de representação digital. Os níveis de representação digitais são definidos em relação à faixa de medida, pré definida, e a quantidade de bits disponível para conversão (resolução). Por exemplo, um ADC com 10 *bits* de resolução, com entradas entre 0 V e 5 V, sendo a maior a tensão de referência, possui 1024 intervalos (do 0 à 1023), o nível 0 representa valores entre 0 V e 4,88 mV, o nível 1 entre 4,88 mV e 9,66 mV, e assim por diante, até o último nível 1023 que representa valores entre 4,995 V e 5 V. A menor diferença entre dois níveis digitais é conhecida como passo, que equivale à resolução do conversor, e no exemplo citado é de 4,88 mV.

Alguns ADC's podem realizar funções especiais como médias e medida diferencial. Os ADC's funcionam como um elo entre o mundo contínuo dos transdutores analógicos e o mundo discreto dos processadores. A comunicação entre um ADC e um processador é digital, e comumente restringe-se a forma serial, como os padrões I2C, SPI e UART, que são muito usados. Há opções de subsistema de processamento que possuem um ou mais ADC internos, possibilitando a leitura de um sinal analógico sem a necessidade de um

ADC externo. Entretanto, os ADC internos são relativamente limitados em resolução e velocidade de conversão. Em aplicações de alta resolução, e/ou velocidade, e/ou entrada diferencial, se faz necessário um ADC externo.

Hoje em dia há uma grande variedade de transdutores digitais no mercado, e que, na maioria dos casos, já realiza um pré processamento de sinal. Fatores que podem influenciar na escolha de um transdutor digital são: auto aquecimento devido à potência destinada ao pré-processamento e conversão, geralmente precisam de um fio extra (referência, alimentação e sinal) e dimensões físicas. Alguns transdutores digitais ainda realizam pré-processamento interno, como no caso da linha de termo-higrômetros SHT3x (Sensirion, 2019). O uso de transdutores digitais facilita a manutenção e atualização dos nodos. Também estão disponíveis no mercado, módulos sensores (placas de testes com transdutores, prontas para o uso), que facilitam a montagem do nodo, pois alguns transdutores requerem circuitos auxiliares como reguladores de tensão e condicionadores de sinal.

1.2 Arduino Nano

Em geral, as unidades de controle em redes de sensores são compostas por microcontroladores (DARGIE, 2010). O Arduino Nano, Figura 1.5, foi escolhido como unidade de controle padrão para nodos da rede proposta a FURG CAN, que é apresenta no capítulo 2. Optou-se pela versão Nano do Arduino pelo fato desta ser compacta e ainda possuir um conversor USB-Serial, facilitando a conexão com o computador. A versão (Arduino) Mini não possui o conversor USB-Serial. Em princípio todo o código desenvolvido para a versão Nano deve funcionar de forma transparente em outras versões do Arduino. Podem ser necessárias modificações no código, para que fique de acordo com as especificidades da versão do Arduino escolhida, como por exemplo alterações de pinos I/O.

O Arduino é um conjunto de *hardware* e *software* desenvolvido para ser fáceis de usar, tendo como público alvo *hobbyists* e iniciantes (Arduino, 2020b) da microeletrônica. Ao nível de *hardware* o Arduino é uma placa de prototipagem eletrônica de projeto aberto para microcontroladores (Arduino, 2020b), o que também é chamado de placa microcontroladora. Há diversos modelos de placas, com tamanhos, microcontroladores e acessórios diferentes, todas compatíveis na medida do possível, pois algumas funcionalidades são específicas. Trocar de placa requer pequenas adaptações, essa compatibilidade é uma das

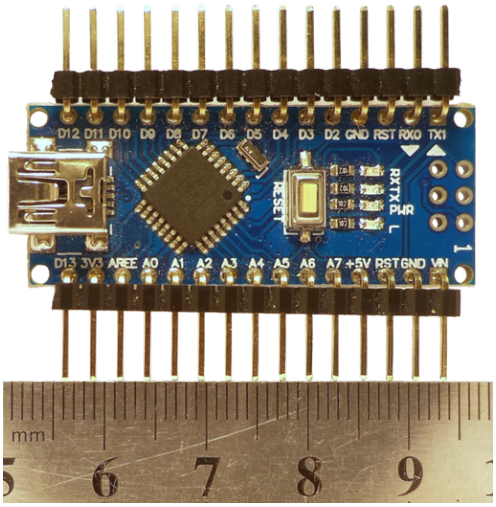


Figura 1.5: Placa microcontroladora Arduino Nano v3. Fonte: Autores.

principais vantagens das plataformas Arduino. Entre as placas Arduino, além da versão Nano, vale destacar a Arduino Uno, que é a mais difundida e a Arduino Portenta H7, que é uma alternativa mais potente.

O Arduino Uno é desenvolvido sobre o mesmo microcontrolador que o Nano, o ATmega328 da Microchip. A primeira diferença notável entre a Nano e Uno é o tamanho, porém, é a menor (a Nano) delas que disponibiliza o maior número de pinos digitais. A Figura 1.6 representa a pinagem, *pinout* em inglês, da placa Arduino Nano. Nela pode-se consultar as posições e as funções dos pinos. O Arduino Nano possui 22 pinos digitais. Destes 8 fornecem acesso ao único conversor ADC de 10 *bits* do ATmega328 (Microchip, 2020) por isso podem ser usados como entrada de sinal analógico. Estes pinos vão do A0 ao A7 e estão localizados ao lado esquerdo na Figura 1.6. Ressalta-se que um mesmo pino pode ser utilizado para mais de uma função e é configurado via *software*. O ATmega328 não possui conversor digital analógico (DAC) (Microchip, 2020). Porém pode emular uma saída analógica através da modulação de largura de pulso (PWM - *Pulse Wave Modulation*). Os pinos digitais que podem realizar essa função são o D3, D5, D6, D9, D10, D11 e são indicados nas placas pelo símbolo $\sim$ ao lado no número. Eles podem ser vistos no lado direito da Figura 1.6.

O ATmega328 não possui uma interface USB nativa, mas sim uma serial UART (Microchip, 2020). Porém a maioria dos computadores atuais não possuem uma interface serial, por isso as placas Arduino possuem um conversor serial USB. Para que o computador possa

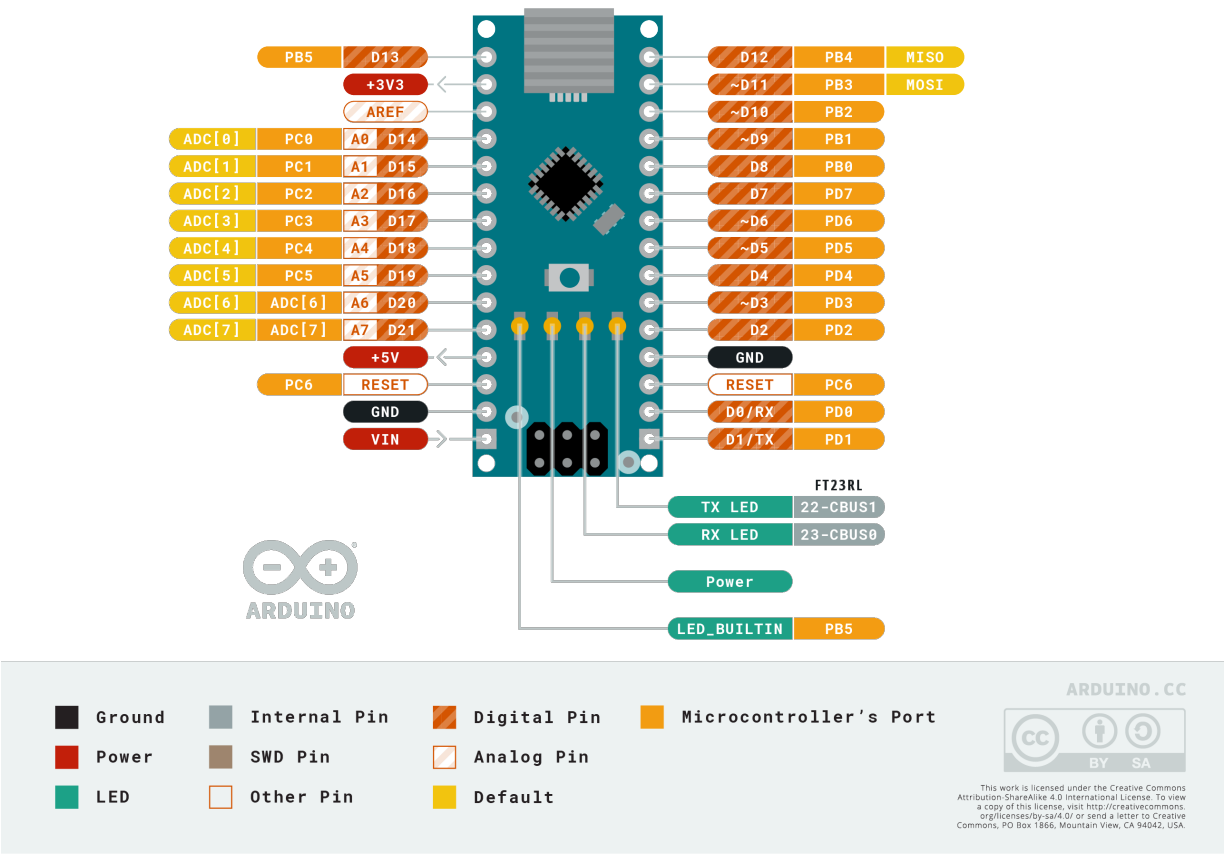


Figura 1.6: Pinout da placa microcontroladora Arduino Nano v3. Fonte: Adaptado de www.arduino.cc.

enxergar o Arduino, deve-se instalar, no computador, o drive do conversor específico, que pode variar em função do fabricante da placa. Além da comunicação serial o ATmega328 possui nativamente os protocolos SPI- *Serial Peripheral Interface* e I2C - *Inter-Integrated Circuit* (Microchip, 2020), que são muito utilizados por transdutores digitais. Em nodos da FURG CAN o protocolo SPI é usado na comunicação com controlador CAN MCP2515 (Microchip Technology Inc, 2019) e com cartão o micro SD. O SPI utiliza 4 pinos, o D13 como *clock*, o D12 como MISO (*Master In Slave Out*), o D11 como MOSI (*Master Out Slave In*), e um outro qualquer como selecionador de *chip* (CS). Nos exemplos disponibilizados, o pino digital 4 é usado com CS do CI MCP2515 e o 10 com CS do cartão SD. A comunicação UART usa os pinos digitais D0 e D1, por isso deve-se evitar a utilização destes juntamente com a comunicação serial, uma vez que isso pode acarretar falhas, por exemplo, durante o carregamento do código.

Ao nível de *software*, o Arduino é um ambiente de programação, compilação, carregamento e monitoramento para suas placas microcontroladoras, e outras compatíveis. Esse

ambiente é chamado de Arduino IDE - *Integrated Development Environment*. A Figura 1.7 apresenta uma foto da interface Arduino IDE, versão 1.8.13. Nela pode-se ver o exemplo *CANTXshort.ino* da biblioteca MCP2515, código no espaço de edição (área branca da janela). Os arquivos de códigos do Arduino IDE são chamados de *sketch*.

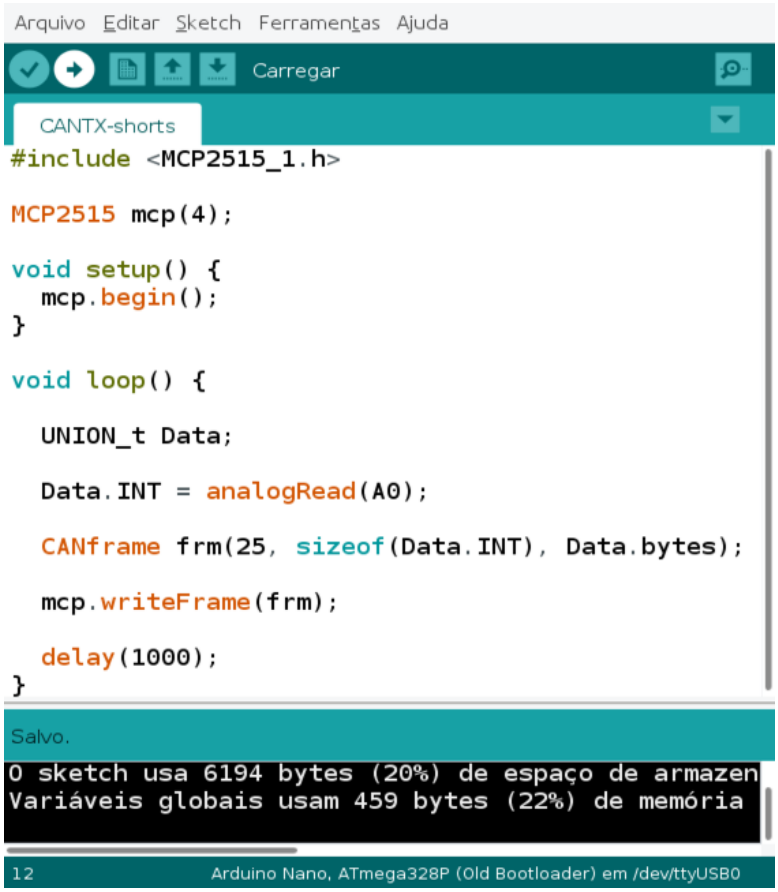


Figura 1.7: IDE, ambiente de programação do Arduino. Código representado, demonstra o uso da função writeFrame(...) da biblioteca MCP2515. Fonte: Adaptado de www.arduino.cc.

Ainda na Figura 1.7, o primeiro botão no topo à esquerda, abaixo do menu Arquivo, é usado para compilar o código, o segundo botão para carregar para o microcontrolador, o terceiro para abrir um novo *sketch*, o quarto para abrir um *sketch* já existente e o quinto para salvar o *sketch*. O botão no topo e na extrema-direita serve para chamar o serial monitor, uma janela que imprime os dados recebidos do Arduino. Além dos botões descritos, o ambiente de programação possui uma barra de menu (*Arquivo*; *Editar*; *Sketch*; *Ferramentas*; *Ajuda*) onde pode-se acessar as demais funções. Na aba *Ferramentas* pode-se selecionar o tipo de plataforma microcontroladora e a porta serial usada. Também destaca-se o caminho para inclusão de bibliotecas: *Sketch/Incluir Biblioteca/Adicionar biblioteca*

.ZIP.

Um relatório da compilação é sempre disponibilizado na parte preta da janela do Arduino IDE. Nele pode-se consultar possíveis erros de sintaxe, quantidade de memória de programa usada, quantidade de memória dinâmica utilizada entre outras informações. O Arduino IDE é atualizado frequentemente, e novas funcionalidades aparecem com frequência. Até o momento (versão 1.8.13) tem-se apenas duas opções de monitoramento uma pela janela *Monitor serial* ou outra pela janela de *Plotter serial*, disponíveis na aba *ferramentas*, porém nenhuma delas possibilita salvar o relatório da comunicação, o que pode ser desejável. Aplicações com essa necessidade, o usuário deve procurar um *software* para monitoramento da comunicação serial. Uma solução possível é o ExtraPutty disponível em: extraputty.com.

A plataforma Arduino é muito popular e a sua comunidade de usuários é muito ativa o que favorece a difusão do Arduino, é possível encontrar diversos sites divulgando informações sobre a plataforma. O livro Arduino Cookbook (Margolis, 2011) é boa referência para. Para consultas rápidas sugere-se a própria pagina da plataforma arduino.cc/reference. Muitos dispositivos possuem suporte prévio para a plataformas Arduino, desenvolvido pela comunidade de usuários. Em caso de dúvidas sobre o uso de um dispositivo específico, pode-se consultar os *forums* da internet, por exemplo, forum.arduino.cc ou realizar buscas no Git Hub, repositório comumente utilizado para depositar/desenvolver códigos e bibliotecas feitas por usuários.

A contribuição deste trabalhado para a comunidade de usuários é a biblioteca do CI MCP2515, a MCP2515-*bib*, dispositivo padrão da FURG CAN. Também é disponibilizado uma documentação em português e diversos exemplos de código, inclusive usando outros dispositivos. Esse conteúdo está disponível em repositório dos autores KakiArduino. Uma parte deste mesmo conteúdo está disponível no apêndice B.

1.3 CAN

Redes de sensores costumam ser uma ferramenta na experimentação física, e nesse sentido, se deseja um modelo de rede acessível, tanto no custo quanto na facilidade de uso, de modo que não se gaste muito tempo e recurso na sua implementação. Também deseja-se uma rede estável ao longo tempo e que tenha uma boa oferta de dispositivos no mercado.

A CAN - *Controller Area Network* é uma opção razoável de rede para experimentos de pequeno porte, em número de nodos. Ela possui um protocolo enxuto, apenas as duas primeiras camadas do modelo OSI (Física e Enlace). A CAN foi projetada para trabalhar em ambientes ruidosos (veículos), fornece uma conexão estável e é padrão na indústria desde meados da década de 1990, ou seja, há uma boa oferta de dispositivos no mercado.

A CAN foi apresentada em 1986 como uma proposta de conexão serial para dispositivos (ECU) de mesmo nível hierárquico em veículos (Kiencke et al., 1986). Os dois principais apelos eram a economia de fiação, e a possibilidade de implementar novas funcionalidades sem aumentar o número de conexões. A CAN foi implementada pela primeira vez em um veículo em 1992, no Mercedes S-Class (Lawrenz e *et al*, 2013). Pouco tempo depois, entre 1994 e 1995, tornou-se o protocolo mais usado em carros e ganhou padronizações da ISO.

Os protocolos CAN *low-speed* e CAN *high-speed* são hoje, os mais tradicionais da família CAN. A primeira diferença evidente entre estes é a taxa de operação, que é 125 kbit/s para o primeiro, e vai até 1 Mbit/s para o *high-speed*. O protocolo *low-speed* é preparado para operar mesmo com uma série de falhas, o que deve ser garantido pelos transceptores e terminações do barramento que são mais sofisticados do que os usados no modo *high-speed*. Aqui focaremos no protocolo de alta velocidade, pois este é mais amplamente utilizado, e foi escolhido para elaboração da rede proposta (FURG CAN).

A princípio o protocolo CAN é baseado em eventos e não no tempo, ou seja, as mensagens são enviadas de acordo com a necessidade de cada nodo da rede. Portanto, não há um ordenamento temporal entre os envios de nodos diferentes. Soluções que exijam maior compromisso temporal, podem migrar para a *Time Triggered* CAN (TTCAN), método de operação baseado no tempo. O princípio de funcionamento da TTCAN é simples: um nodo envia periodicamente uma mensagem para sincronizar os demais nodos, esta mensagem marca o início de um clico de transmissão, que é dividido em intervalos, sendo que cada nodo envia sua mensagem em um intervalo específico. Na TTCAN os nodos são ordenados ao longo dos intervalos de modo que o receptor sabe a ordem na qual os envios devem acontecer, por isso a falta de qualquer um dos nodos é facilmente percebida (Bosch, 2005) pelo receptor.

Os protocolos CAN, 1.0, 2.0 A, 2.0 B e TTCAN possuem uma quantidade máxima de dados fixa de 8 *bytes*, o que começou a ser um fator limitante para algumas aplicações veiculares, *e.g.* multimídia, e nesse sentido a BOSH criou a CAN FD (*Flexible Data-Rate*),

lançada em abril de 2012. As mensagens do protocolo CAN FD podem transportar de 8 à 64 *bytes* (Bosch, 2012) de uma única vez, além disso, como o nome sugere, a CAN FD permite o aumento da taxa de transmissão durante o campo de dados, intervalo de tempo no qual são propagados os *bytes* de dados, o que confere a CAN FD uma transmissão efetiva de dados muito superior aos demais protocolos da família CAN citados. Até o momento a BOSCH tem tomado cuidado para que os novos protocolos englobem os anteriores. Isso tem adiado a obsolescência de dispositivos feito para as primeiras versões do protocolo.

1.3.1 Arquitetura e Layout básico

Com intuito de possuir imunidade contra falhas individuais (Kiencke et al., 1986), a topologia proposta inicialmente para CAN foi do tipo barramento *broadcasting* multi-mestre. Que ao contrário da topologia estrela, não possui um dispositivo gerenciador. Nem precisa que os nodos retransmitam os pacotes como em uma topologia do tipo anel (*ring*), uma vez que todos podem escutar as mensagens transmitidas. Todavia hoje em dia, a CAN também é usada com as topologias estrela, duas estrelas e híbridas (Lawrenz e *et al*, 2013). *broadcasting* é método de transmissão, no qual as mensagens são simultaneamente enviadas para todos os nodos. No caso do protocolo CAN, qualquer nodo da rede transmite sua mensagem para todos os demais ao mesmo tempo, cabendo a cada nodo ouvinte selecionar as mensagens de seu interesse. As mensagens da CAN possuem identificadores, e os nodos CAN utilizam filtros de aceite para recolher as mensagens em função de seus identificadores. Por conta desse mecanismo as mensagens da CAN são denominadas de mensagens auto-orientadas (Kiencke et al., 1986).

As principais vantagens da topologia de barramento são a facilidade de expansão e conexão de novos nodos, além do baixo custo relativo, pois usa menos fiação que as outras configurações. Por outro lado, uma falha no barramento físico, *e.g.*, rompimento da linha, pode comprometer o funcionamento da dividindo a rede em duas. Um outro ponto crítico é a possibilidade de colisão de pacotes. Todavia os projetistas da CAN já contornaram esse último problema, através de um mecanismo de arbitragem não destrutiva, que apresentado na subseção 1.3.3.5.

1.3.1.1 O barramento CAN

O barramento CAN é tipicamente composto por um par de fios trançados usados no tráfego de dados, o CAN HIGH e o CAN LOW. Aplicações automotivas costumam usar cabos do tipo UTP - *Unshielded Twisted Pair* (par trançado sem blindagem), com passos de 20 mm à 50 mm (Lawrenz e *et al*, 2013).

A configuração com três linhas, garante que não haja diferença de potencial entre os GND dos nodos. No caso de implementações com apenas as duas vias para dados, deve-se ficar atendo para que essa diferença não ultrapasse o limite do transceptor utilizado, que em geral é de −12 V à 12 V (Lawrenz e *et al*, 2013). Essa limitação é fornecida pelo transceptor utilizado pelo nodo. No caso da FURG CAN é usado o TJA1050, sua diferença de potencial entre o GND dos nodos não pode ser maior que 40 V, nem menor que −27 V (Philips, 2003).

Os nodos conectados ao barramento, se comunicam através de um sinal de tensão nas linhas de dados, CAN HIGH (V_H) e CAN LOW (V_L). O estado do barramento, valor lógico usado para a codificação do sinal, é aferido pela diferença de potencial, entre as duas linhas de dados ($V_H - V_L$). Os limites de tensão para aferição dos níveis lógicos do barramento, são especificados pela ISO11898-2:2003 como sendo: No mínimo 0,9 V, no máximo 5,0 V e tipicamente 2,0 V para o estado dominante, nível de tensão alto e ”0”lógico; No mínimo −1,0 V, no máximo 0,5 V e tipicamente 0 V para estado recessivo, nível de tensão baixo e ”1”lógico (ISO, 2003b).

Devido a simetria de montagem, medida diferencial em um par trançado, o barramento CAN é pouco sensível a interferências causadas por ruídos eletromagnéticos. A CAN até pode operar com apenas uma via de dados (CAN HIGH ou CAN LOW), porém isso a resulta em um barramento mais sensível a interferências eletromagnéticas. Por isso, este modo de operação costuma ser guardado para casos de falhas, curtos ou rompimento em uma das duas linhas de dados. O limite de ruído para a detecção de um estado dominante (alto) é de 0,3 V, enquanto que para a detecção de um estado recessivo (baixo) é de 0,5 V (ISO, 2003b).

A distribuição física da CAN é do tipo barramento, como representado pela Figura 1.8, onde as distância destacadas representam os limites geográficos da rede que variam em função da taxa de transmissão de *bits* e da configuração de *Bit-Timing* usadas na rede.

Na Figura 1.8, L é a distância máxima entre dois nodos que pode ser confundida com o comprimento do barramento e d_e é comprimento máximo de uma extensão. A distância mínima entre dois nodos (d_n) também é definida, como a soma das duas extensões (d_e) mais d_c. Estas distâncias são especificadas pela ISO 11898-2, para a frequência de 1 Mbit/s da seguinte forma: $L < 40\text{m}$, $d_n > 0,1\text{m}$ (ISO, 2003b). Na seção seguinte 1.3.2, que trata da temporização, ficará mais claro como a taxa de transmissão e a a configuração de *Bit-Timing* interferem na distância máxima entre dois nodos, e vice-versa.

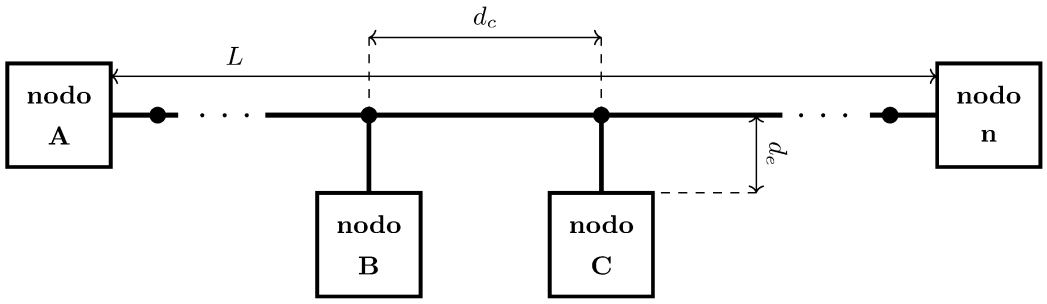


Figura 1.8: Diagrama em blocos de uma topologia física do tipo barramento. Fonte: Autores.

O número máximo de nodos da rede é limitado pelos transceptores utilizados, visto que eles são os dispositivos diretamente conectados ao barramento. A medida que se adiciona nodos, a carga do barramento cresce e, a medida que a carga cresce, o transceptor emissor deve aumentar sua corrente de saída, de modo a manter o nível correto no barramento durante a comunicação. Os transceptores possuem um limite de corrente de saída e acabam limitando o número de nodos da rede, *e.g.*, o transceptor TJA1050 da Philips pode operar em barramentos com até 110 nodos (Philips, 2003).

As linhas de dados do barramento CAN (CAN HIGH e CAN LOW) são conectadas, uma a outra, nos extremos, através de duas resistências, que garantem um nível específico de impedância. Essas resistências são chamada de terminações e são fundamentais para ao funcionamento da CAN, principalmente no modo *high-speed*. O barramento da CAN pode ser interpretado como um circuito RC, onde as linhas de dados formam um capacitor (cabos em paralelos) e as terminações e nodos, associados em paralelo, uma resistência equivalente. Na subseção 3.2.2 é apresentado um relato de testes feitos em um barramento CAN com uma e sem nenhum terminação.

O nível de impedância recomendado pela ISO é 120 Ω, no mínimo 95 Ω e, no máximo 140 Ω (ISO, 2003b). Além de especificar o nível de impedância a *ISO 11898-2* também

sugere algumas montagens que atendem as especificações. Aqui é descrito as duas mais comuns, pois as demais são variações. A forma mais simples de terminação da CAN, de acordo com a *ISO 11898-2*, é realizada por dois resistores de 120 Ω, que conectam a linha CAN HIGH com a linha CAN LOW em seus extremos (ISO, 2003b), como apresentada na Figura 1.9.

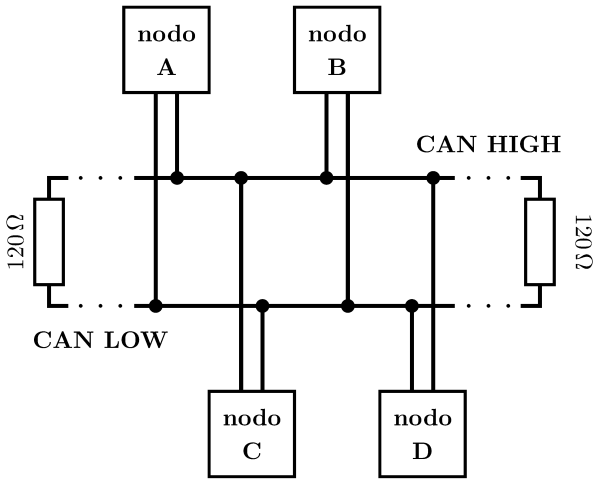


Figura 1.9: Representação da terminação de um barramento CAN. Fonte: Autores.

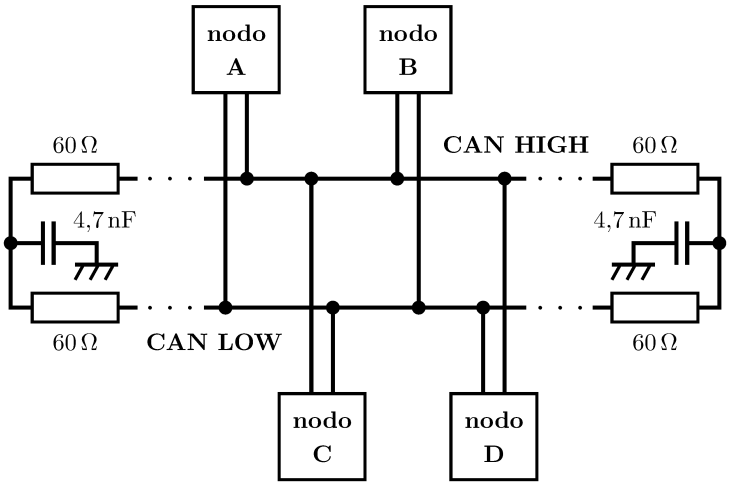


Figura 1.10: Representação da terminação com capacitor central de um barramento CAN. Fonte: Autores.

A diferença entre o primeiro modelo de terminação e o segundo é a adição de um *center tab* normalmente de 4,7 nF ou maior (Lawrenz e *et al*, 2013), a Figura 1.10 apresenta um diagrama deste modelo de terminação. Nessa configuração, um capacitor é ligado ao GND é adicionado entre dois resistores de 60 Ω. Duas dessas associações substituem os dois resistores de 120 Ω que conectam as linhas de dados em seus extremos. Essa configuração

minimiza interferências causadas por tempos de propagação diferentes entre as linhas de dados e por diferença de tempos de chaveamento dos transceptores (Lawrenz e *et al*, 2013).

1.3.1.2 Os nodos da CAN

Os nodos CAN são normalmente constituídos por três elementos, a Unidade de Controle (subsistema de processamento) comumente composta por microcontroladores, a Unidade de Controle CAN composta por dispositivos de dedicação exclusiva que podem ser também microcontroladores, e os transceptores, dispositivos responsáveis pelo acoplamento dos nodos ao barramento. Um diagrama simplificado de um nodo CAN típico é apresentado na Figura 1.11.

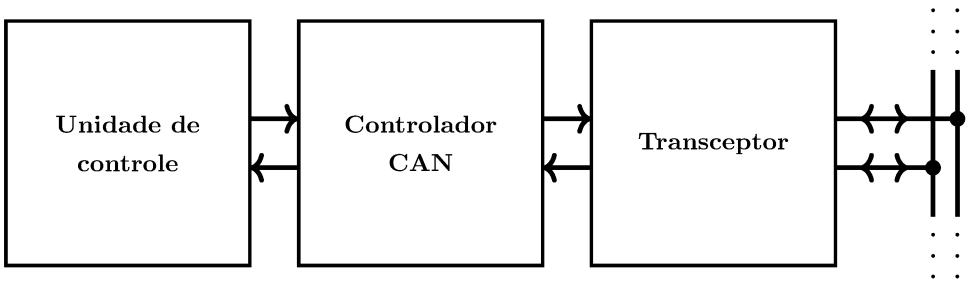


Figura 1.11: Diagrama simplificado de um nodo CAN típico. Fonte: Autores.

Além da configuração apresentada na Figura 1.11, um nodo CAN pode ter outros elementos, como unidades de armazenamento de dados, unidades de sensoriamento (transdutores), unidades de comunicação para outros protocolos, etc. Hoje em dia há dispositivos que podem desempenhar dupla função em nodos CAN, sendo ao mesmo tempo a Unidade de Controle e o controlador CAN, como por exemplo o ESP32, que possui uma *engine* CAN internamente (Systems, 2021), a fabricante chama essa interface de TWAI - *Two-Wire Automotive Interface*. Vale ressaltar que a filosofia do protocolo CAN exige que todos os nodos conectados ao seu barramento executem funções específicas de gerenciamento o que sobrecarrega as unidades de controle, nesse sentido é comumente incluído nos nodos um controlador CAN, como por exemplo o MCP2515 da Microchip (Microchip Technology Inc, 2019) e o TWAI da Philips (Philips, 1997).

Em uma abordagem simples pode-se dizer que os transceptores são adaptadores de nível, que separam os sinais de entrada (TX) e saída (RX), realizam a medida diferencial do barramento CAN, e convertem o resultado para níveis que possam ser logicamente

interpretados pelos dispositivos subsequentes, e vice-versa. Entretanto esses dispositivos são responsáveis pela realização do protocolo mais baixo da camada física do protocolo CAN. Eles devem atentar há uma série características relacionadas ao tempo, ao modo de operação, carga no barramento e a imunidade contra ruídos que um transceptor CAN deve atender para garantir o funcionamento ideal da rede. Além os tempos que o transceptor leva para transferir a informação do nível do barramento para sua saída, e para fazer o inverso, são fundamentais para a concepção de um projeto de rede CAN, visto que podem limitar o tamanho do barramento ou a taxa de *bits* da rede. Essa dependência será mais explicada na subseção 1.3.2. Como exemplo, a especificação do TJA1050 indica um atraso máximo do Tx para RX, soma dos dois tempos mencionados, de 250 ns (Philips, 2003).

Dentre as características elétricas de um transceptor, destaca-se a imunidade contra ruído de modo comum com altas amplitudes, *i.e.*, ruído intenso que aparece nas duas linhas de dados, que por exemplo pode ser causado por indução de campo e transientes nas linhas de dados do barramento. Imunidade de modo diferencial, ou seja, contra ruído que aparece em apenas uma das linhas, causado por indução não simétrica nas linhas de dados. Imunidade contra descarga eletrostática. Suportar uma diferença de potencial entre GND's relativamente alta, algo em torno de -12 V à 12 V (Lawrenz e *et al*, 2013). E ser capaz de gerar um sinal com alto grau de simetria entre as linha de dados do barramento, CAN HIGH e CAN LOW. Podem ser associados, opcionalmente, entre o transceptor e o barramento dispositivos para incrementar a imunidade de ruído da rede, por exemplo, são comumente utilizados filtro de bobina de modo comum (*common-mode choke*) e protetores contra descargas eletrostáticas.

Os controladores CAN são responsáveis pelo gerenciamento de quase todo o protocolo CAN, eles só não são responsáveis pelo controle do nível do barramento, detecção do nível do barramento e escrita no barramento, o que é feito pelos transceptores. Cabe aos controladores CAN aplicar todos os procedimentos da camada Enlace (*Data Link*), tanto os associados a subcamada LLC - *lógical Link Control* quanto a subcamada MAC - *Medium Access Control*. Além de lidar com a camada Enlace o controlador CAN deve realizar os três seguintes procedimentos ligados a camada Física da CAN: A codificação dos *bits*, o gerenciamento do tempo de duração dos *bits*, e o sincronismo dos nodos.

Um controlador CAN, costuma ter duas lógicas de comunicação, uma com protocolo CAN, e outra para a comunicação com a Unidade de Controle, que pode ser, por exemplo,

do tipo SPI como no caso do MCP2515 da Microchip (Microchip Technology Inc, 2019). A Figura 1.12 apresenta um diagrama em blocos simplificado de um controlador CAN, o bloco interno mais à esquerda representa a lógica de comunicação com a Unidade de Controle. O bloco interno mais à direita representa a lógica do protocolo CAN, nele deve conter quase todas as operações da camada de Enlace, como o encapsulamento e codificação dos dados, a camada de Enlace do protocolo CAN é descrita na seção 1.3.3.

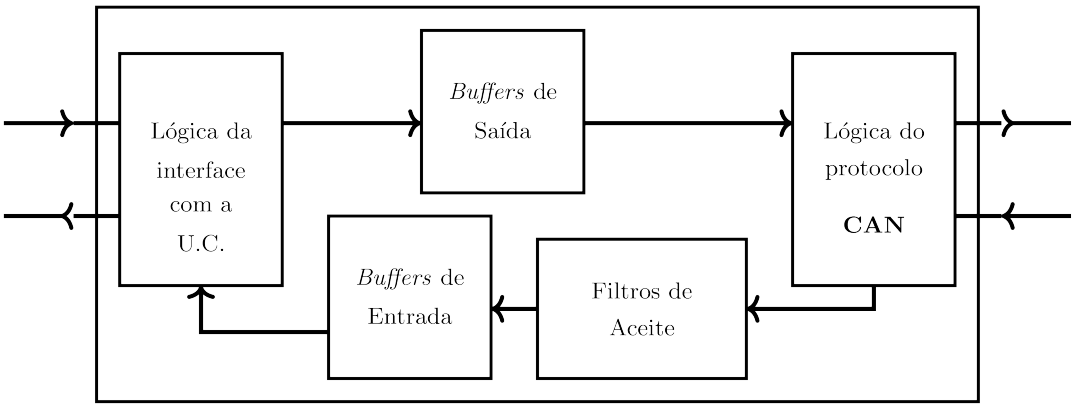


Figura 1.12: Diagrama em bloco simplificado de um controlador CAN. U.C. - Unidade de Controle. Fonte: Autores.

A única operação da camada de Enlace que não é representada pelo bloco da extrema-direita na Figura 1.12 é a filtragem de mensagens, que é indicada pelo bloco "Filtros de Aceite". Nele é feito a aplicação de filtros e máscaras digitais para selecionar as mensagens de interesse e descartar as demais, o que tem papel fundamental no funcionamento da CAN, pois evita a sobrecarga nas unidades de controle. Esse tipo de operação é necessária pois a comunicação CAN é tipo *broadcasting*, onde todos os nodos podem escutar todas as mensagens propagadas na rede.

Além dos blocos já mencionados há na Figura 1.12 outros dois blocos "*buffer*", um de entrada e outro de saída. O "*buffer* de entrada"serve como espaço de armazenamento temporário de mensagens recebidas e já decodificadas, e pode ser do tipo *First In First Out* - *First In First Out* como no caso do TWAI (Philips, 1997). Somente quando armazenadas no *buffer* de saída é que as mensagens podem ser acessadas pela Unidade de Controle. De forma análoga o *buffer* de saída também serve como espaço de armazenamento temporário, porém para mensagens a serem propagadas no barramento, elas saem desse *buffer* para a "Lógica do protocolo CAN", depois seguem para o transceptor, e assim por diante, até chegar nos dispositivos de destino.

1.3.2 Temporização

A princípio pode-se pensar que quanto maior for taxa de *bits* melhor, pois mais rápido será a troca de informação na rede. Entretanto não se pode aumentar indefinidamente a frequência de operação, há limites físicos de temporização que garantem a operacionalidade da rede. Esses limites possuem dependência com a velocidade de propagação do sinal no fio usado para as linhas de dados, com a distância máxima entre dois nodos, com os tempos de conversão de entrada e saída dos transceptores mais "lentos"da rede e com as frequências internas dos controladores CAN.

A CAN pode operar nas taxas de 1 kbit/s até 1 Mbit/s (Lawrenz e *et al*, 2013), sendo que as velocidades padrões são de 125 kbit/s para modo *fault tolerance*, e 250 kbit/s, 500 kbit/s e 1 Mbit/s para o modo *high-speed*. Entretanto a taxa efetiva de transmissão de dados não é explicitada nessas frequências, e claro é bem menor. A taxa de transmissão máxima de dados na CAN pode ser calculada pela equação 1.1, onde N é o número de *bits* do *frame* de dados utilizado sem contar os *bits* do campo dados, *i.e.*, 47 *bits* para *frames* com ID padrão ou 67 *bits* para *frames* com extensão de ID. O número de *bits* de dados enviados, é representado, em *bytes*, separadamente na equação 1.1 pela variável n, isso acontece pois o número de bytes de dados enviado pode variar de 0 até 8. Ainda na equação 1.1, f_{CAN} representa a taxa de *bits* usada durante a comunicação.

$$T_{max} = \frac{n8}{N + n8} f_{CAN} \left[\frac{\text{bit}}{\text{s}}\right] \tag{1.1}$$

A taxa efetiva efetiva calculada com a equação 1.1 será uma aproximação visto que o número total de bits, representado por $N + n8$, varia também em termos do número de *stuffing bits* (*bits* de enchimento) inseridos pelo emissor na codificação das mensagens enviadas. Pode-se calcular o melhor caso, sem *stuffing bits*, e o pior caso, com o número máximo de *stuffing bits*. No Apêndice A há uma tabela com os valores das taxas de transferências efetivas para as frequências padrões da CAN, os valores apresentados foram calculados para situações sem *stuffing bits* e com o número máximo de *stuffing bits*. O protocolo CAN não permite a existência de sequências com 6 *bits* de mesmo valor, por isso, a cada sequência de 5 *bits* com mesmo valor o emissor insere um *bit* de enchimento com valor oposto ao da sequência, esse mecanismo é mais bem explicado na seção 1.1.

1.3.2.1 *Bit-Timing* e *Time Quanta*

O ***time quanta*** (**tq**) é um conceito fundamental da temporização, ele é a unidade de expressão do tempo, *i.e.*, todos os parâmetros temporais são expressos em unidades de *time quanta*, com por exemplo, o *Bit-Timing*. O *time quanta* é calculado pela frequência interna do controlador CAN (f_{crt}) e pelo *Baud Rate Prescaler* (BRT) como na equação 1.2 (Lawrenz e *et al*, 2013). O BRT é o fator de divisão da frequência interna do controlador, é programável e seu valor costuma variar de 1 a 32 tq (Bosch, 1991), a f_{crt} pode ser igual a frequência fornecida pelo oscilador (f_{osc}) do controlador CAN ou sua metade.

$$tq = \frac{BRP}{f_{crt}} \text{ [s]} \tag{1.2}$$

O ***Bit-Timing***, ou tempo de *bit*, é o tempo de duração de um bit e é dado pelo inverso taxa de transmissão de *bits* da rede, por exemplo, em uma rede que opera com 500 kbit/s o *Bit-Timing* é de 2 µs. Além disso o *Bit-Timing*, e consequentemente a taxa de *bits*, é um parâmetro de implementação da rede CAN, ele é programado em registros específicos dos controladores CAN, e há uma série de regras para definir seu valor ideal, que estão sumarizadas no último paragrafo desta subseção, segunda a BOSCH seu valor deve variar de 8 à 25 tq (Bosch, 1991).

No protocolo CAN o *Bit-Timing* é dividido em quatro segmentos não sobrepostos (Bosch, 1991), como apresentado na Figura 1.13, onde **SS** é o segmento de sincronismo, **SP** é o segmento de propagação, **SF1** é o segmento de fase 1 e **SF2** é o segmento de fase 2. Ainda na Figura 1.13, **PA** é o ponto de amostragem, onde o valor do bit é aferido.

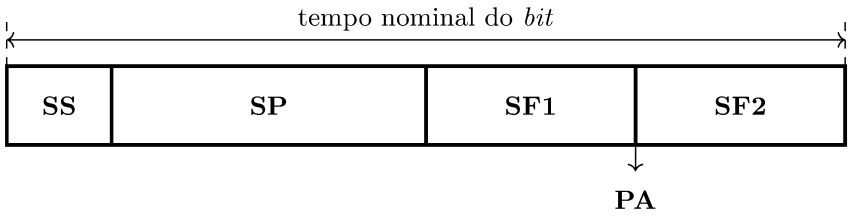


Figura 1.13: Partições do *Bit-Timing* do protocolo CAN. Fonte: Autores.

O **SS** (segmento de sincronismo) é um dos menores, se não o menor, segmento do *Bit-Timing*, durando apenas um tq, e ao contrário dos demais segmentos sua duração não é programável. É somente durante o SS que o controlador sincronizado pode observar uma transição de nível, uma borda de 0 para 1 ou vise-versa, quando a borda acontece fora

desse segmento o controlador CAN assume que possui um erro de fase (ϵ_f), e executará uma ressincronização assim que possível.

O **SP** (segmento de propagação) deve durar tempo suficiente para que um sinal escrito em um dos extremos do barramento, possa propagar até o outro extremo, ser aferido pelo transceptor e possivelmente sobrescrito pelo mesmo, propagar de volta até origem e por fim ser lido pelo transceptor que o emitiu. Esse tempo é descrito como o *delay* de propagação (**DP**) e fornecer o limite inferior do *Bit-Timing*, em geral é indicado um *Bit-Timing* maior do que duas vezes a duração do SP (Lawrenz e *et al*, 2013). A equação 1.3 pode ser usada para calcular o *delay* de propagação, nela a velocidade do sinal no fio foi arredondada para baixo, sendo considerada 2×10^8 m/s. Uma medida rápida, com o osciloscópio TDS 210, em laboratório, acusou $2,2 \times 10^8$. O princípio explorado nessa medição foi a reflexão na extremidade, causada por variação abrupta de impedância entre o fio e o ar.

O comprimento do barramento em metros é indicado por **cb**, e $\mathbf{a_{tr}}$ é a soma dos atrasos de leitura e escrita do transceptor mais lento utilizado. O valor de SP é expresso em tq, é programável e deve ser arrendondado para cima, segunda a BOSCH sua duração varia de 1 a 8 tq (Bosch, 1991).

$$DP = 2(5 \times 10^{-9} cb + a_{tr}) \text{ [s]} \tag{1.3}$$

O **SF1** (segmento de fase 1) é usado durante a ressincronização, que pode prolonga-lo, atrasando o ponto de amostragem (PA) do nível do barramento. A soma do SF1 com o SP deve fornecer um tempo grande o suficiente para o nível do barramento alcance a estabilidade, garantindo um valor confiável durante a amostragem. SF1 é um valor programável nos controladores CAN e segunda a BOSCH seu valor varia de 1 à 8 tq (Bosch, 1991), em alguns controladores SF1 e SP formam um único valor, chamado de TSEG1 - *Time Segment 1* (Lawrenz e *et al*, 2013).

O **SF2** (segmento de fase 2), que também pode ser chamado de TSEG2 - *Time Segment 2*, pode ser usado na ressincronização, durante o qual pode ser encurtado, antecipando o próximo SS, e conseguintemente o próximo PA. O **PA** (ponto de amostragem) é localizado entre o SF1 e o SF2, e por isso o SF2 deve durar um tempo suficientemente grande para que o valor da amostragem seja processado, o que varia de 1 à 2 tq, segundo a BOSCH o SF1 deve ser o máximo entre o tempo de processamento e o tempo de duração do SF1 (Bosch, 1991).

SF1 e SF2 são usados para corrigir erros de fase durante a ressincronização, e nesse sentido seus comprimentos são relacionados ao desvio máximo de frequência que pode haver entre os nodos da CAN, sem comprometer o funcionamento da rede, em geral quando maior for esses segmentos maior será o desvio de frequência aceito df. A equação 1.4 pode ser usado para verificar o desvio de frequência máximo aceito (Lawrenz e *et al*, 2013), nela mSF é o valor do menor segmento de fase, BT é o valor do *Bit-Timing* e df é expresso em porcentagem.

$$df = \frac{mSF}{2(13BT - SF2)} \tag{1.4}$$

O comprimento do *Bit-Timing* e de seus segmentos é de grande importância para o funcionamento da rede, para determinar o sua duração deve-se primeiro calcular o DP, por exemplo pela equação 1.3, ele fornecerá o limite inferior para o *Bit-Timing* que em geral deve ser maior que o dobro do *delay* de propagação. Em seguida deve-se escolher um BRT que possa representar o SP em 1 à 8 tq, no caso do SP está junto ao SF1 (TSEG1) poderá conter até 15 tq (Lawrenz e *et al*, 2013). Por fim pode-se dividir os *time quantas* restantes entre SF1 e o SF2, ou no caso do SP e do SF1 estarem juntos no TSEG1, os time quanta restantes são atribuídos ao TSEG2.

1.3.2.2 Sincronismo Hard e Ressincronização

Há dois momentos distintos para ocorrer o sincronismo da CAN, um ao início da transmissão, mais especificamente durante o SOF (*Start Of Frame*), e outro durante a transmissão da mensagem. O sincronismo que ocorre durante a transmissão da mensagem é chamado de ressincronização, e possui três mecanismos de funcionamento relacionados ao tamanho e sentido do erro de fase (e_f).

O **sincronismo *Hard*** é acionado sempre que uma borda do estado recessivo para o dominante é detectada durante o tempo ocioso do barramento, ou seja, sempre que ocorre o SOF. No instante anterior ao início de uma transmissão, o barramento deve estar ocioso, o que corresponde a um nível recessivo contínuo, no início de uma transmissão o nodo emissor gera um nível dominante, que sobrescreve o recessivo, essa ação é chama de SOF e marca o início do *frame*. Os demais nodos conectados a rede entende essa borda como o início de uma transmissão e executam o sincronismo *Hard*, esse mecanismo reinicia a

contagem interna do *Bit-Timing* dos controladores CAN, garantindo que todos os nodos da rede estejam sincronizados.

O **erro de fase** (e_f) é quem determina qual tipo de mecanismo será acionado pela ressincronização, e este é descrito pela distância temporal relativa ao SS (Segmento de Sincronismo). As bordas de sinal devem sempre ocorrer durante o SS, caso contrário o nodo assume que está dessincronizado, se uma borda ocorre antes do SS o e_f do nodo será negativo, se a borda ocorrer depois do SS o e_f do nodo será positivo.

A **ressincronização** pode ser acionado pelo controlador CAN de um nodo dessincronizado apenas uma vez por *Bit-Timing* (Bosch, 1991) e em geral para bordas que vão do nível recessivo para o nível dominante. Os controladores CAN possuem uma quantidade temporal, expressa em *tq*, usada para a ressincronização, a RJW - *Resyncronization Jump Width*. Se o e_f for menor RJW a ressincronização reinicia a contagem do *Bit-Timing* de forma semelhante ao sincronismo *Hard*. Se o módulo e_f for maior, em módulo, do que RJW, e for positivo a ressincronização atua aumentando SF1 em uma unidade de RJW. Se o e_f for maior, em módulo, do que RJW e for negativo, a ressincronização atua diminuindo em uma unidade de RJW o SF2. Há mais uma condição para que a ressincronização possa ser executado pelo controlador, o nível do sinal seguinte a borda que gerou o e_f, deve ser diferente do sinal antes desta mesma borda (Bosch, 1991), essa regra diminui a chance de um ruído induzir a ressincronização.

A quantidade **RJW** é programável e seu valor pode variar de 1 a 4 *tq*, ou no máximo o valor do Segmento de Fase 1 (SF1) (Bosch, 1991). Comumente o e_f não é eliminado em um única ressincronização, dessa forma quanto maior o número de bordas do nível recessivo para o dominante houver, maior será a chance do controlador CAN permanecer sincronizado. E esse é um dos motivos para a existência do mecanismo de *stuffing bit*, pois ele garante que não ocorram mais do que cinco *bits* consecutivos de mesmo valor durante a maior parte de uma mensagem, esse mecanismo só não atua nos últimos 11 *bits* de um *frame* (mensagem formatada) de dados.

1.3.3 Camada de Enlace de Dados

O protocolo CAN fornece as duas camadas inferiores do modelo OSI, a camada Física e a camada de Enlace de Dados. Há diversos protocolos que implementam a última camada OSI, Camada de Aplicação, para a CAN, como por exemplo a CANopen (CiA CAN, 2021),

as camadas intermediárias, da terceira a sexta, não são implementadas para rede CAN. A camada física corresponde as características dos transceptores, além da temporização e decodificação de *bits*, e do sincronismo (ISO, 2003a), assuntos tratados nas secções anteriores deste capítulo, 1.3.1 e 1.3.2. A camada de enlace de dados é composta por duas subcamadas, a LLC (*Logic Link Control*) e a MAC (*Media Access Control*), a Figura 1.14 apresenta a distribuição da camada de Enlace de Dados do protocolo CAN.

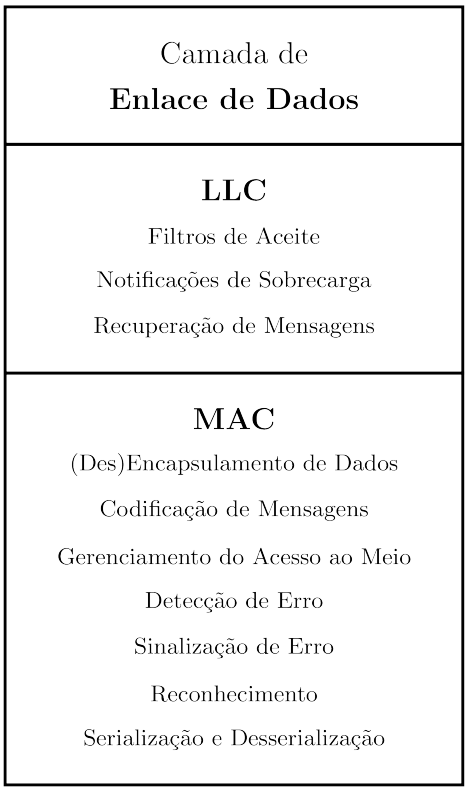


Figura 1.14: Diagrama das funções de Enlace do protocolo CAN distribuídas de acordo com o modelo OSI. Fonte Autores.

1.3.3.1 Subcamada LLC

Como pode ser visto na Figura 1.14, a subcamada LLC contém as operações de filtragem de mensagens, notificação de sobrecarga e recuperação (reenvio) de mensagens. As mensagens que chegam nessa subcamada, não estão codificadas, *i.e.*, sem *bits* de enchimento. O *frame* de dados da subcamada LLC é composto por três campos, o campo de ID (identificação), o campo de DLC (controle de dados) e o campo de dados, a Figura 1.15 representa esses três campos. Já o *frame* de pedido remoto é composto apenas pelos campos ID e DLC.

Figura 1.15: Partições do *frame* de dados da camada LLC do protocolo CAN. Fonte: Autores.

O campo de ID é composto pelo identificador padrão que contém 11 *bits*, pela sinalização de extensão (IDE) que contém 1 *bit* e pela extensão de identificação que contém 18 *bits* (ISO, 2003a). Se o *bit* de sinalização for 1 significa que o identificador é do tipo estendido, do contrário a extensão será ignorada. O campo DLC é formado por 4 *bits* que informam o número de bytes contidos no campo de dados. O protocolo CAN limita o tráfego de dados em até 8 *bytes* por mensagem. Um *frame* de pedido remoto não tem dados, mas possui o campo DLC, e neste caso é comum que a informação deste campo contenha a quantidade de bytes da informação solicitada, de qualquer forma, o conteúdo do campo DLC de um *frame* de pedido remoto é sempre ignorado pelas operações da camada LLC (ISO, 2003a).

Pensando por uma perspectiva física do barramento, podemos chegar a conclusão que a rede CAN é do tipo *broadcast*, entretanto o protocolo fornece um método de seleção de mensagens, que possibilita os roteamentos *Unicast* e *Multicast*. O mecanismo que proporciona essa flexibilidade pode ser entendido com comunicação orientada a objetos, na qual um link virtual entre dois nodos é estabelecido pela mensagem (Kiencke et al., 1986). A CAN implementa a orientação através da seleção, ou melhor dizendo pela utilização de **filtros de Aceite** que agem sobre a identificação das mensagens e as selecionam. O identificador não possui o endereço do destino da mensagem, e sim informações sobre a mensagem, neste sentido a seleção é feito em função do tipo da mensagem e não do seu destino, por isso o receptor deve saber qual tipo de informação quer, para então resgatá-la da difusão do barramento.

Os filtros propriamente ditos são gravados em registros dos controladores CAN, e são aplicados aos identificadores das mensagens, indicados no campo ID dos *frames*, já decodificadas, ou seja, sem *bits* de enchimento, mas antes de serem encaminhadas ao *buffer* de entrada do controlador. Ao programar os filtros de Aceite no controlador CAN deve-se saber de antemão quais mensagens serão utilizadas pelo nodo, nesse sentido uma boa dica para implementações de redes CAN é a construção de um dicionário de mensagens, no qual todos os tipos de mensagens são listadas e categorizadas pelos seus identificadores.

A **notificação de sobrecarga**, *overload notification*, serve para notificar aos demais nodos que o controlador CAN precisa de mais tempo antes para finalizar o processamento da última mensagem. O envio dessa notificação acaba por gerar um atraso na rede, visto que o *frame* de sobrecarga que será construído na subcamada seguinte (MAC) é composto por *bits* dominantes. O controlador pode gerar no máximo duas notificações de sobrecarga por mensagem, seja de dados ou pedido remoto, (ISO, 2003a).

O processo de **recuperação de mensagens** realizado na subcamada LLC é responsável pelo reenvio automático de mensagens que foram canceladas por erro ou por que perderam a arbitragem para uma mensagem mais prioritária (ISO, 2003a). Isso significa que o controlador CAN deve tentar reenviar uma mensagem interrompida, e só poderá confirmar a transmissão após o envio com sucesso da mensagem, essa confirmação é produzida pela subcamada MAC e encaminhada a LLC.

1.3.3.2 Subcamada MAC

A Figura 1.14 sumariza os procedimentos ou funções executadas pela subcamada MAC do protocolo CAN, os procedimentos relacionados a codificação de mensagens, gerenciamento do acesso ao meio e reconhecimentos de mensagens serão brevemente discutidos nesta subseção. O encapsulamento, ou formatação, das mensagens nos cinco tipos de *frames* utilizados pelo protocolo CAN, *frame* de dados, *frame* de pedido remoto, *frame* de sobrecarga, *frame* de erro ativo e *frame* de erro passivo, são apresentados na seção 1.3.4. A detecção e sinalização de erro, bem como o mecanismo de contagem dinâmica e de confinamento de nodos em termos dessa contagem são discutidos na seção 1.3.5.

Para transmitir uma mensagem o nodo deve certificar-se de que o barramento está ocioso, o que pode ser concluído através da observação uma sequência de três *bits* recessivos após a conclusão do último *frame*. Os procedimentos de gerenciamento de acesso ao meio da subcamada MAC foram concebidos com uma filosofia multi mestre, de modo que todos os nodos possuem os mesmos direitos de acesso ao barramento, e de certa forma isso acontece, pois quem sofre o processo de arbitragem é a mensagem e não o nodo que a enviou. Todavia, se um nodo contabiliza mais do que 127 erros de transmissão ou recepção ele é considerado passivo de erro, e por isso não pode enviar duas mensagens seguidas com o espaço mínimo de 3 *bits*, como feito pelos nodos saudáveis, eles devem esperar 8 *bits* a mais. As sinalizações de erro e de sobrecarga não seguem as temporizações descritas, elas

funcionam pelo princípio de sobreposição do estado dominante sobre o recessivo.

1.3.3.3 Transmissão e recepção na subcamada MAC

Pode-se pensar na subcamada MAC como dois mecanismos quase simétricos, um de transmissão e outro de recepção. Tanto o processo de recepção quanto o de transmissão acontecem na forma de um fluxo de *bits*, de modo análoga ao *streaming* de vídeos, onde os processos agem de maneira sequencial sobre o fluxo *bits*, entretanto alguns processos agem sobre um número maior de *bits* e por isso levam mais tempo do que outros.Os controladores possuem *buffers* de saída e de entrada, onde são armazenados os *frames* da camada LLC, no processo de transmissão o *buffer* de saída é o ponto de partida, ou seja, a subcamada LLC, enquanto o de chegada é o transceptor, ou seja, a camada Física. O processo de recepção é quase o caminho inverso, discordando apenas em alguns processo, *e.g.*, na verificação de erro, enquanto o transmissor verifica erros do tipo Erro de *Bit* e Erro de Reconhecimento, o receptor observa outros três: Erro de Enchimento; Erro de CRC; Erro de Forma.

Durante a transmissão de uma mensagem a subcamada MAC da CAN recebe o *frame* da subcamada LLC, com o identificador, o campo DLC e no caso de uma *frame* de dados com o campo de dados. O código de verificação redundante é calculado a partir da divisão do *frame* LLC recebido por um polinômio de 15° ordem. O *frame* da subcamada MAC é então construído adicionando o *bit* SOF, o *bit* SRR, os *bits* reservados RB1 e RB0, o campo CRC, que contém o código cálculo e um delimitador, um *bit* recessivo para o *slot* de reconhecimento (ACK) e outro para o seu delimitador, e por fim o campo *End of Frame* (ISO, 2003a).

A Figura 1.16 destaca os *bits* e campos adicionados pela subcamada MAC do protocolo CAN, o *bit* destacado em rosa no campo de arbitragem, é o *bit* SRR - *Standard Frame Remote Transmit Request* que é usado para indicar se a mensagem com ID padrão é um pedido remoto, porém para ID com extensão esse *bit* é fixado como recessivo e usado, independentemente do *frame* ser um pedido remoto ou não. Os dois *bits* destacados em azul no campo de controle, seguinte ao da arbitragem, são os *bits* reservados RB0 e RB1. Além deles são adicionados o Campo do CRC em verde, o campo ACK em laranja e Campo EOF em roxo.

Como o *frame* já montado, o controlador espera o barramento ficar disponível e então realiza a serialização do *frame* e a codificação por enchimento, encaminhando o fluxo de

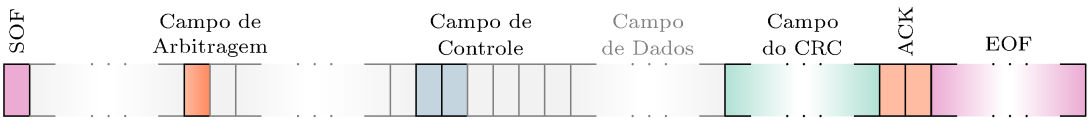


Figura 1.16: As regiões em destaque são adicionadas na subcamada MAC antes do envio do *frame* para a camada física. Fonte: Autores.

bits para camada física, que realizará a propagação. Durante esse processo, é feito um monitoramento *bit* a *bit* em busca de possíveis erros, e em especial durante a arbitragem, período no qual o controlador deve verifica se ainda detêm a prioridade do barramento, pois caso perca a arbitragem deve mudar para o modo de recepção a tempo de receber a mensagem mais prioritária. Além disso o controlador espera que seu *bit* recessivo do *slot* de reconhecimento (ACK) seja sobrescrito por um dominante, se isso não ocorrer o transmissor contabilizará um Erro de Reconhecimento.

Durante toda a transmissão o controlador vigia o barramento, e caso verifique que sua mensagem foi sobrescrita por uma sinalização de erro, ele interromperá o envio e contabilizará erro de transmissão, o que também acontece se o próprio emissor detectar um Erro de *Bit* ou um Erro de Reconhecimento. A mensagem interrompida não é apagada do *buffer* de saída da subcamada LLC, que voltará a solicitar o envio a subcamada MAC assim que possível. O envio pode ser interrompido também por uma sinalização de sobrecarga, neste caso o envio será forçadamente atrasado, pois o barramento ficará ocupado durante a sinalização de sobrecarga que detêm a prioridade do barramento.

O processo de recepção é quase o inverso do processo de transmissão, desta vez a sequência de *bits* é recebida da camada Física serialmente, enquanto o controlador realiza a desserialização e os demais processos, reconstruindo o *frame* em um *buffer* de entrada da camada LLC. Os processos executados na recepção são as checagem dos erros de enchimento, CRC e forma, a decodificação que é a retirada dos bits de enchimento e se tudo de certo o encaminhamento do bit de reconhecimento (ACK). Ao fim ainda é verificado se há uma sinalização de sobrecarga, se houver o controlador CAN respondera com outra sinalização de sobrecarga, produzindo o eco de sinalização. Se houver a detecção de qualquer falha o controlador contabilizara um erro de recepção, e camada MAC formatará um sinalização de erro, essa sinalização também pode ser estimulada pela propagação de uma sinalização de erro ativa por outro nodo, que na prática é entendida como um erro de enchimento.

Uma diferença curiosa entre o mecanismo de recepção e de transmissão acontece quando o último bit de um *frame* de dados ou de pedido remoto é sobrescrito por um dominante, neste caso a verificação de Erro de Bit pode perceber o erro, e considerar aquela mensagem como inválida, voltando a reenvia-la assim que possível. Do outro lado a verificação de Erro de Forma seria capaz de detectar a sobreposição do último *bit*, porém nesse caso específico, o erro é ignorado, resultando no recebimento duplicado da mensagem. Essa diferença acontece por que o momento de validação do *frame* na recepção acontece antes do que na transmissão, de qualquer modo, ao final de ambos processos a confirmação de sucesso deve ser repassado a subcamada LLC.

1.3.3.4 Codificação de enchimento

Os *frames* da CAN são codificados com a técnica de enchimento, de modo a não ocorrerem sequências de 6 *bits* com mesmo valor, o segmento codificado vai do SOF - *Start Of Frame* até o penúltimo bit do campo CRC (ISO, 2003a), a Figura 1.17 destaca esse segmento. Durante a transmissão o controlador CAN, insere um *bit* oposto a cada cinco *bits* consecutivos com o mesmo valor, nessa regra também são contabilizados *bits* inseridos pelo enchimento. A Tabela 1.1 possui um sequência de *bits* antes e depois da codificação de enchimento.

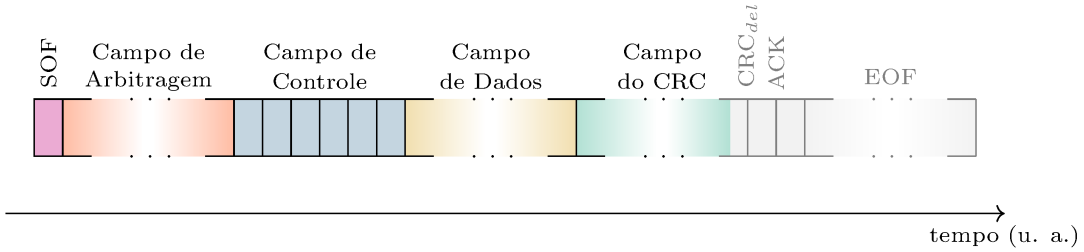


Figura 1.17: Representação da região codificada por enchimento, é apenas nesta que pode ocorrer erro de enchimento. Apesar do campo do CRC fazer parte da região seu delimitador (CRC_{del}) não faz. Essa parte do *frame* também é usada para calcular o código de CRC. Fonte: Autores.

Tabela 1.1 - Representação do processo de codificação por enchimento do protocolo CAN. Em negrito são destacados os *bits* de enchimento, dominante (**0**) e recessivos (**1**). Fonte: Autores.

sequência de *bits* sem enchimento	0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1
sequência de *bits* com enchimento	0 0 1 1 1 1 1 **0** 0 0 0 0 **1** 0 0 0 0 1 1 1 1 1 **0** 1

O *bit* delimitador do campo CRC, o *bit* delimitador do ACK e o campo EOF (*End of*

Frame), são utilizados nas verificações de Erro de Forma pelo receptor, enquanto o *slot* de ACK é verificado pelo transmissor para confirmação do sucesso do envio. Por isso estes não são codificados pelo enchimento.

1.3.3.5 Arbitragem

Na CAN os *bits* dominantes possuem naturalmente o poder de sobrescrever os recessivos, e isso é aproveitado durante a disputa de prioridade ao barramento, que deve ocorrer de forma não destrutiva (ISO, 2003a). Esse mecanismo de superposição também é utilizado para inicializar uma transmissão, o SOF é um *bit* dominante que sobrescreve o estado recessivo do barramento ocioso. Pode acontecer de dois nodos tomarem a iniciativa ao mesmo tempo, e neste caso acontecerá a arbitragem, além disso, um nodo pode participar da arbitragem, e até vencê-la, mesmo não tento tomado a iniciativa de transmissão, para isso ele sincroniza seu envio com o SOF do outro nodo.

O campo de arbitragem, período do *frame* no qual acontece a disputa, começa a partir do primeiro *bit* de ID, logo após o SOF, e termina no *bit* RTR (*Remote Transfer Request*). Para um *frame* com identificador padrão, o campo de arbitragem dura 12 *bits*, os 11 *bits* do ID e mais o *bit* RTR. Enquanto que para um *frame* com extensão a arbitragem pode durar até 32 bits, contendo os 11 *bits* do ID padrão, mais o *bit* SRR, mais o *bit* IDE que sinaliza a extensão, mais os 18 *bits* do extensor de identificação, e por fim, mais o *bit* de RTR. Em um disputa entre um *frame* dados com extensão de ID e outro sem, vencerá aquela que tiver o menor ID padrão, e se eles possuírem o mesmo valor de ID padrão, o *frame* que não possui a extensão vencerá.

A arbitragem atua sequencialmente sobre o fluxo de *bits*, e para que fique claro a distribuição de prioridade entre os ID, não se deve concatenar a extensão de ID com o ID padrão, e sim expressá-los separadamente. Vencerá sempre a mensagem que possuir o menor ID padrão, e entre mensagens com o mesmo ID padrão, mas com extensões diferentes, vencerá aquela que tiver a menor extensão. A arbitragem não diferencia os *frames* de pedido remoto dos *frames* de dados, entretanto no caso de uma competição entre um *frame* de dados e um de pedido remoto que possuam o mesmo ID, o *frame* de dados vencerá por conta do *bit* RTR, que é dominante para *frames* de dados, e recessivo para *frames* de pedido remoto.

A arbitragem do protocolo CAN é de contenção, ou seja, não destrutiva, e neste sentido

as mensagens interrompidas são reservadas para novas tentativas de envios. Durante toda a transmissão o controlador CAN realiza um monitoramento *bit* a *bit* do nível do barramento, e se ele detectar algo diferente do que escreveu contabilizara um Erro de Bit. Porém, durante o campo de arbitragem, ao detectar um nível dominante no lugar de um recessivo, o controlador CAN não deve contabilizar um erro e sim entender que perdeu a prioridade do barramento. Não pode haver dois nodos capazes de transmitir mensagens com campos de arbitragem idênticos, isso pode causar colisão destrutiva de *frames*.

Para exemplificar o mecanismo de arbitragem: Suponha que quatro nodos comecem a transmitir uma mensagem simultaneamente, um deles, o nodo A, pretende enviar um *frame* de dados com extensão de ID, outros dois nodos, o C e o D, tentam enviar um *frames* de dados sem extensão de ID, e o nodo B tenta enviar um *frame* de pedido remoto com o mesmo ID do C, essa situação é ilustrada na Figura 1.18. O ID padrão da mensagem do nodo A vale 1131 e sua extensão é $\geq 65\,536$, já o ID padrão do nodo B vale também 1131, ambos possuem o mesmo ID padrão, porém *frames* estendidos possuem os dois *bits* subsequentes recessivos, e por isso a prioridade da mensagem do nodo A é menor que do a do nodo B. O SRR do *frame* estendido corresponde, em posição, ao *bit* RTR do *frame* padrão, o nodo B envia um pedido remoto, logo esse *bit* também é recessivo para ele, entretanto o seu *bit* IDE é dominante indicando, que o *frame* não é estendido, pois, em *frames* estendidos esse *bit* é recessivo.

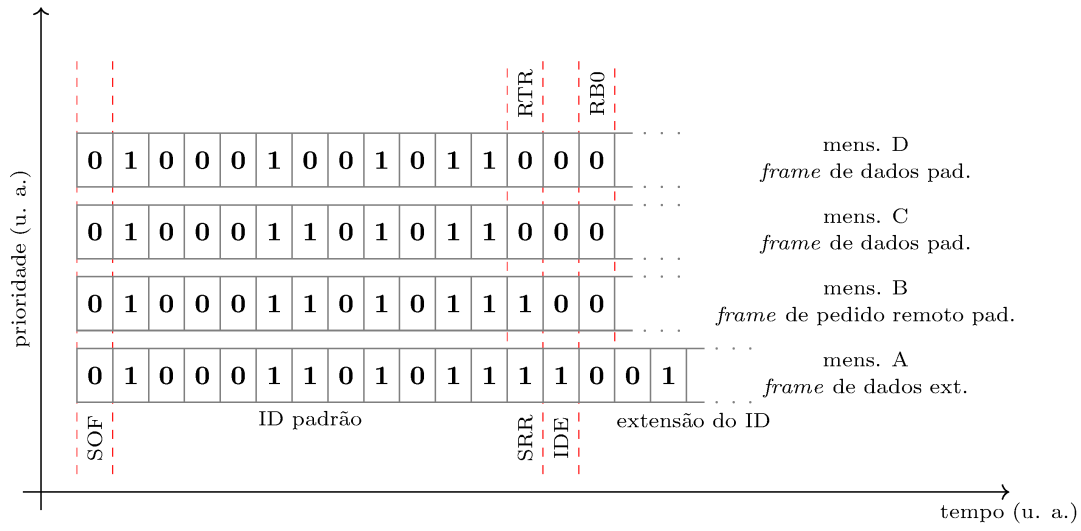


Figura 1.18: Representação do mecanismo de arbitragem do protocolo CAN. Os *frames* estão organizados, de baixo para cima, do menos para o mais prioritário, os *bits* estão ordenados, da esquerda para direita, por ordem de publicação no barramento, as unidades de tempo e prioridade são arbitrárias. Fonte: Autores.

Ainda na competição da Figura 1.18, o *frame* do nodo B perde em prioridade para o *frame* do nodo C, ambos possuem o mesmo ID padrão, porém o *RTR* da mensagem do nodo C é dominante, indicando que é um *frame* de dados. A arbitragem representada na Figura 1.18 é vencida pela mensagem do nodo D, uma vez que ele possui o menor ID padrão, que vale 1099. Na situação ilustrada os nodos A, B e C, perceberiam que perderam a arbitragem no sexto *bit* do ID padrão, e a partir desse ponto passaria a funcionar como receptores. Como a arbitragem da CAN é não destrutiva, os nodos A, B e C, voltaram a tentar enviar suas mensagens interrompidas.

1.3.4 Formato das Mensagens

O protocolo CAN possui cinco formatos de mensagens diferentes, o *Frame* de Dados, o *Frame* de Pedido Remoto, o *Frame* de Erro Ativo, o *Frame* de Erro Passivo e o *Frame* de sobrecarga. Os formatos podem ainda ser separados em duas categorias diferentes, *frames* convencionais, de dados e de pedido remoto, e *frames* de sinalizações, de erro e de sobrecarga. Um *frame* convencional deve ser separado do anterior, seja ele convencional ou de sinalização, por no mínimo um espaço de três *bits* recessivos, denominado de intervalo entre *frames* (ISO, 2003a). Por outro lado *frames* de sinalização não respeitam esse intervalo, podendo até mesmo sobrescrevem a mensagem atual do barramento com uma sequência de *bits* dominante.

O espaço total entre dois *frames*, é a soma do intervalo entre *frames* com um possível período de ociosidade do barramento, que tem duração indeterminada, e pode até mesmo não existir. A Figura 1.19 representa um espaço entre *frames* composto pelos 3 *bits* de intervalo e mais 6 *bits* de ociosidade, repare que o nível do barramento durante o tempo ocioso também é recessivo, assim como durante espaço entre *frames* e durante campo EOF - *End Of Frame* do *frame* anterior, e como já mencionado o início de uma transmissão é marcado por um *bit* dominante que sobrescreve o estado ocioso do barramento.

O protocolo CAN possui um mecanismo que privilegia os nodos "saudáveis", aqueles que operam no modo Erro Ativo, em detrimento dos "não tão saudáveis", aqueles que operam no modo Erro Passivo. Essa desigualdade e os modos de confinamento são apresentados na seção 1.3.5, por hora basta saber que nodos "não tão saudáveis", não concorrem ao barramento da mesma forma que os nodos "saudáveis". Um caso particular dessa desigualdade é quando um mesmo nodo confinado em Erro Passivo envia dois *fra-*

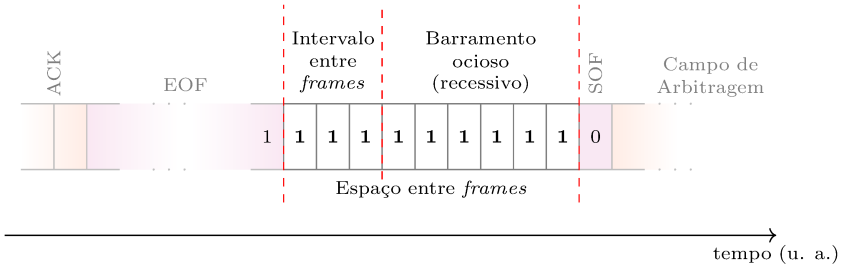


Figura 1.19: Espaço entre *frames* de pedido remoto e de dados. O estado ocioso do barramento é identificado como um nível recessivo interrupto, que começa logo após do EOF. Fonte: Autores.

mes seguidos, neste caso o nodo deve esperar um tempo mínimo de 11 *bits*, após o EOF do primeiro *frame*, antes de iniciar a segunda transmissão. Os 8 *bits* extras compõem o intervalo de transmissão suspensa, e acabam privilegiando a transmissão de outros nodos, a Figura 1.20, representa um espaço entre *frames* composto pelos 3 *bits* do intervalo entre *frames*, mais os 8 *bits* do intervalo de suspensão e mais 6 *bits* de ociosidade.

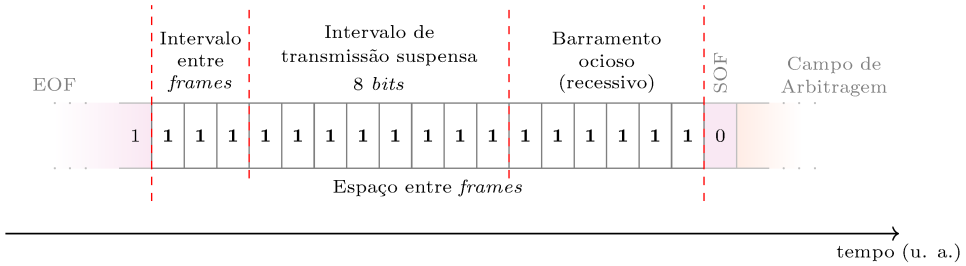


Figura 1.20: O espaço entre *frames* sequênciais enviados pelo mesmo nodo confinado em Erro Passivo, é prolongado em relação ao espaço tradicional, por um tempo extra de 8 *bits*, o tempo de suspensão. Fonte: Autores.

1.3.4.1 Frames de Dados e Frames de Pedido Remoto

De início as mensagens do protocolo CAN possuíam apenas ID padrão, composto de 11 *bits*, na versão 2.0 uma extensão de ID com 18 *bits* foi adicionado aos *frames*. O uso da extensão de ID é opcional, o que resulta em duas formatação diferentes para *frames* convencionais, uma chamada de padrão, e a outra de estendida, a diferença óbvia é no comprimento do campo de arbitragem, local das identificações, porém a outros detalhes que também diferem um formato do outro.

Pode-se fragmentar um *frame* de dados da CAN em sete campos não sobrepostos: O SOF - *Start Of Frame* que marca o início da transmissão com um *bit* dominante; O Campo de Arbitragem que contém a identificação das mensagens e pode durar no máximo 32 *bits*;

O Campo de Controle que possui a duração de 6 *bits*; O Campo de Dados que pode durar no máximo 64 *bits*; O Campo de CRC com 16 *bits* de duração; O ACK - *Acknowledge* (Reconhecimento) com 2 *bits*; E o EOF - *End Of Frame* composto por 7 *bits* recessivos. Essa divisão do *frame* de dados é representado na Figura 1.21.

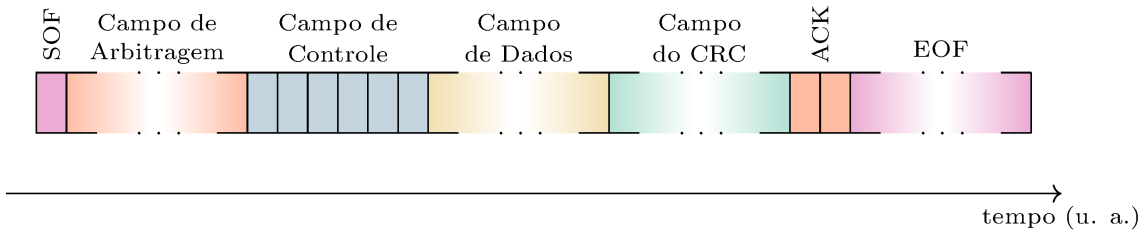


Figura 1.21: Representação dos campos de um *frame* de dados genérico. Fonte: Autores.

Como mencionado o **SOF** marca o início de uma transmissão de dados ou pedido remoto e é composto por apenas um *bit* dominante (Bosch, 1991), ele só pode ser escrito em um barramento ocioso. Ao detectar um *bit* dominante no lugar do último *bit* do intervalo entre *frames*, que é recessivo, o controlador deve aceitá-lo como SOF, e o mesmo deve acontecer quando nodos confinados em ERRO Passivo detectam de um *bit* dominante durante seu intervalo de suspensão. Todo nodo deve realizar o Sincronismo *Hard* assim que detecta um SOF, além disso se este possuir uma transmissão pendente, deve tentar enviá-la, escrevendo no barramento o primeiro *bit* de sua identificação, logo após o SOF detectado. Por outro lado, nodos que não possuem transmissão pendente devem, após o sincronismo, captar a mensagem propagada. Um nodo confinado em Erro Passivo que possui uma mensagem pendente é foi o último a enviar um *frame* convencional, só poderá disputar a transmissão se o SOF tiver sido detectado após o intervalo de suspensão.

Para um *frame* sem extensão de ID o **Campo de Arbitragem** é composto do identificador da mensagem (11 *bits*) e do *bit* RTR - *Request Remote Transfer* que indica se a mensagem é um pedido remoto (estado recessivo) ou uma transmissão de dados (estado dominante). Os 12 *bits* do Campo de Arbitragem de um *frame* sem extensão de ID são representados na Figura 1.22.

Na CAN 2.0 foi implementada a possibilidade de adicionar uma extensão de ID com 18 *bits*, o que prolongou o Campo de Arbitragem para 32 *bits*, a Figura 1.23 representa o Campo de Arbitragem estendido. Além da extensão foram adicionado outros dois *bits*, um desses é o IDE, que ocupa a mesma posição temporal em ambos *frames*, padrão e

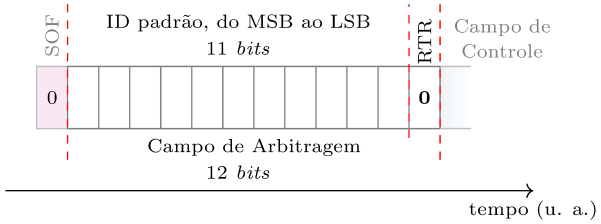


Figura 1.22: Representação genérica do Campo de Arbitragem de um *frame* de dados com ID padrão do protocolo CAN. Fonte: Autores.

estendido, porém em um *frame* padrão ele pertence ao Campo de Controle, Figura 1.24, o IDE deve ser recessivo quando acompanhado de um ID com extensão, do contrário deve ser dominante. O outro *bit* é o RTR que é deslocado para o fim do campo, logo apos o *bit* menos significante (LSB) da extensão. A posição ocupada pelo *bit* RTR na versão padrão do *frame* é substituído pelo SRR que deve ser recessivo, porém se não for os receptores não acusarão erro (ISO, 2003a).

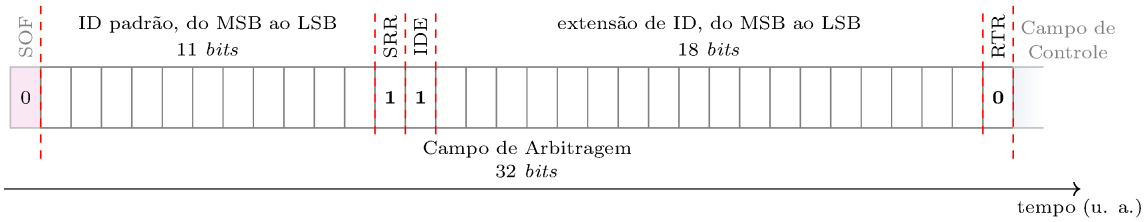


Figura 1.23: Representação genérica do Campo de Arbitragem de um *frame* de dados com extensão de ID do protocolo CAN. Fonte: Autores.

O **Campo de Controle** começa na 14° posição em *frames* padrão, enquanto em *frames* estendidos, ele é deslocado para a 34°, a duração dele é de 6 *bits* em ambos *frames*. Em *frames* estendidos o Campo de Controle é composto pelos *bits* reservados RB1 e RB0, e pelos quatro *bits* do código DLC (*Data Len Control*) que indica o tamanho do campo de dados, de 0 à 8 *bytes*. Em *frames* padrões o *bit* RB1 é substituído pelo IDE.

O **Campo de Dados** é o campo que leva a informação de interesse da mensagem, pode possuir até 8 *bytes* de dados, mensagens sem dados podem ser usadas para sincronizar o relógio dos nodos. Além de possui o *bit* RTR dominante, um *frame* de pedido remoto, Figura 1.25, não possui campo de dados, e seu DLC costuma informa o tamanho da informação solicitada.

O **Campo do CRC** tem a duração de 16 *bits*, sendo 15 deles para o carregar o código de verificação cíclica redundante (*Cyclic Redundancy Check*) e um como terminador do

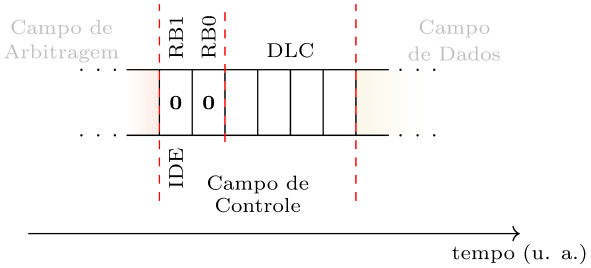


Figura 1.24: Representação do Campo de Controle de um *frame* CAN genérico. Em *frames* estendido o campo de controle é composto por RB1, RB0 e DLC, em *frames* padrões é composto por IDE, RB0 e DLC. Fonte: Autores.

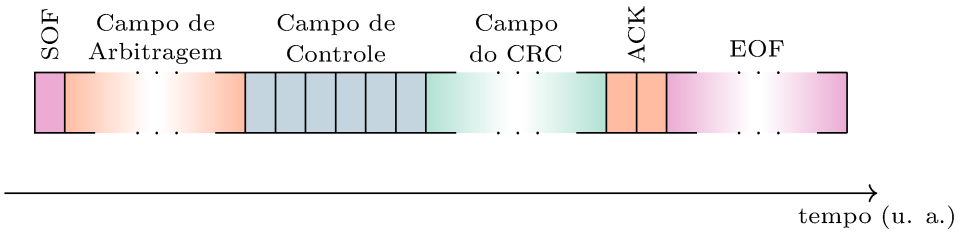


Figura 1.25: Representação dos campos de um *frame* de pedido remoto genérico. Fonte: Autores.

campo que deve ser recessivo. O transmissor calcula o código CRC sobre o SOF, o Campo de Arbitragem, o Campo Dados deslocado para esquerda por 15 *bits* recessivos. O receptor faz o mesmo cálculo sobre o SOF, o Campo de Arbitragem, o Campo Dados e os primeiros *15 bits* do campo de CRC, onde está o código calculado pelo transmissor. Ao finalizar a recepção do Campo CRC, o receptor deve obter pelo cálculo 0, se isso não acontece ele acusará Erro de CRC (ISO, 2003a).

O **ACK** dura dois *bits*, um é chamado de *slot* de reconhecimento, e outro é um delimitador que deve ser recessivo. Durante a transmissão o controlador escreve um *bit* recessivo no *slot* de reconhecimento, o receptor ao reconhecer a validade da mensagem, após a checagem do código de CRC, deve sobrescrever o nível do barramento com um *bit* dominante, informando ao transmissor que a mensagem foi recebida com sucesso.

O **EOF** - *End Of Frame* é composto por 7 *bits* recessivos, e marca o fim da transmissão (Bosch, 1991), este campo não é codificado pelo enchimento, e logo não pode gerar erro de enchimento.

1.3.4.2 Sinalizações, Frames de Erro e Frames de Sobrecarga

O protocolo CAN prevê três tipos de sinalização, duas para indicar erros, e uma para indicar sobrecarga de nodos, a grande diferença entre os *frames* convencionais e os de sinalização é o formato. As sinalizações são compostas apenas de uma sequência de *bits* dominantes, de Erro Ativa e de Sobrecarga, ou recessivos, de Erro Passivo, e do delimitador de sinalização, que é uma sequência de 8 *bits* recessivos. A intenção final do mecanismo de sinalização de erro é avisar para todos os nodos receptores da rede, que a mensagem atual do barramento está corrompida e deve ser descartada, e ao nodo transmissor que ele deve interromper e reiniciar o envio da mensagem. Para atender a esse objetivo as sinalizações de erro do protocolo CAN provocam propositalmente erros na mensagem corrompida, de modo que todo nodo possa classificá-la como inválida.

A **sinalização de erro ativa** só pode ser feita por nodos, transmissores e receptores, "saudáveis", *i.e.*, confinados no modo Erro Ativo, ela é composta por 6 *bits* dominantes, e por isso tem a capacidade de sobrescrever qualquer mensagem. Se uma sinalização de erro ativa é feita sobre a parte codificada com enchimento de um *frame* convencional, os receptores acusarão um Erro de Enchimento (ver 1.3.5). Se a sinalização ocorrer fora da região de enchimento, entre o *bit* delimitador do Campo do CRC, último do campo, e o fim da transmissão, o receptor acurará Erro de Forma. O transmissor que tem sua mensagem interrompida contabilizará um Erro de Bit, se a sinalização ocorrer após o campo de arbitragem, se ocorrer dentro da arbitragem ele deve acusar Erro de Enchimento.

A **sinalização de erro passiva** é composta por 6 *bits* recessivos, logo ela não tem a propriedade de sobrescrever mensagens, por isso uma sinalização de erro produzida por um receptor será imperceptível. A sinalização de erro passiva produzida pelo próprio transmissor será percebida pelos receptores com um Erro de Enchimento, mas para isso ela deve ocorrer na fração codificada do *frame* e após o Campo de Arbitragem, que vai do primeiro *bit* do Campo de Controle até o delimitador do Campo do CRC, se ela ocorrer dentro da arbitragem o nodo perderá a disputa, e os demais não perceberam o erro. Em disputas entres *frames* padrões e *frames* estendidos a arbitragem durará no máximo até o fim do Campo de Arbitragem estendido. Os *frames* de erros são compostos pela sobreposição das sinalizações de erro e pelo delimitador de erro, que é composto por 8 *bits* recessivos.

A consequência implícita da sinalização de erro é um efeito chamado de eco de erro, que

pode prolongar a sinalização de erro em até 6 *bits* dominantes extras (ISO, 2003a). Como o mecanismo de sinalização acaba por força os demais nodos a detectar um erro, estes também irão produzir sinalizações de erro, que irão sobrepor uma a outra, prolongando a sinalização, visto que a mesma é composta apenas de *bit* dominantes. Depois de enviar os *bits* da sinalização de erro o nodo deve chavear sua saída produzindo um nível recessivo, este nível será sobrescrito pelo eco de erro, porém após todos os nodos enviarem suas notificações, o barramento alçará o nível recessivo. Assim que o controlador detectar o primeiro *bit* recessivo, indicando o fim das sinalizações de Erro Ativa, ele deverá enviar mais 7 *bits* recessivos, de modo a produzir o delimitador de erro, que é composto por 8 *bits* recessivos.

A Figura 1.26 representa uma situação hipotética envolvendo apenas dois nodos, onde um transmissor detecta um Erro de *Bit*, e em seguida envia uma sinalização de erro ativa. Os últimos dois *bits* observados pelo receptor antes de o transmissor enviar a sinalização de erro, eram dominantes, e por isso o receptor detecta Erro de Enchimento no quarto *bit* da sinalização do transmissor. Após detectar o Erro de Enchimento, o receptor envia sua sinalização de erro ativa seguida de um *bit* recessivo na tentativa de delimitar a sinalização de erro, e como só há dois nodos no barramento, o delimitador do erro é estabelecido.

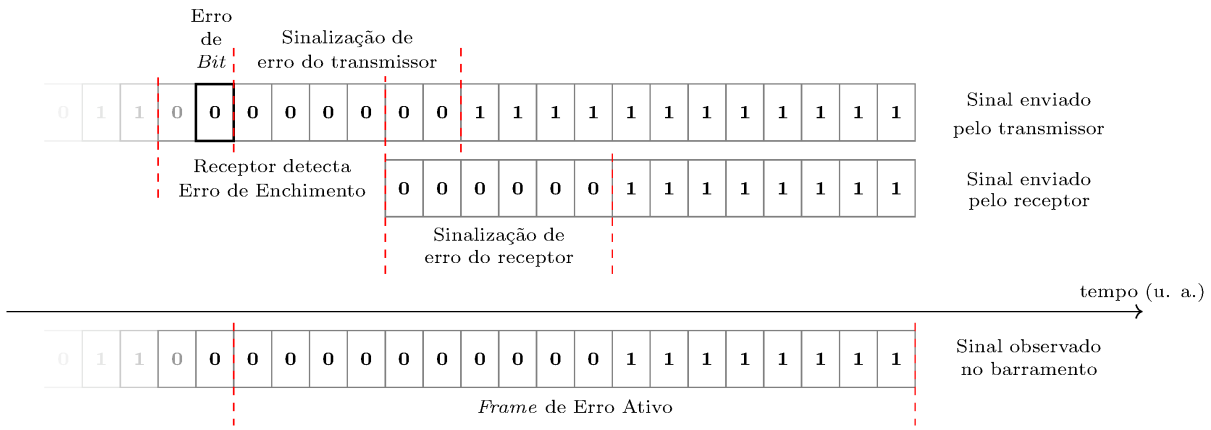


Figura 1.26: Representação de um *frame* de erro produzido pela superposição de sinalizações de Erro Ativa, de um transmissor e de um receptor. Fonte: Autores.

A **sinalização de sobrecarga** tem o mesmo formato da sinalização de erro ativa, 6 *bits* dominantes, seu delimitador também possui o mesmo formato, 8 *bits* recessivos, e além disso também provoca eco de sinalização. A diferença entre a sinalização de erro ativa e a de sobrecarga, é o momento em que elas podem acontecer, em geral sinalizações de erro

devem ser feitas durante a transmissão de um *frame* convencional de modo a interrompe-lo, enquanto a sinalização de sobrecarga só pode ser iniciada durante o primeiro *bit* do intervalo entre *frames* (Bosch, 1991). A sinalização de sobrecarga é usada para retardar o envio do próximo *frame*, prolongando o espaço entre *frames*, de modo que o receptor tenha um tempo extra para processar a última mensagem.

Ao detectar um *bit* dominante no lugar do primeiro *bit* recessivo do intervalo entre *frames*, o controlador deve produzir uma sinalização de sobrecarga. Esse mecanismo é responsável pelo eco de sobrecarga, que prolongar a sinalização com até 6 *bits* extras, da mesma forma que o eco de erro. Após realizar a sinalização o nodo deve setar um nível recessivo, e assim que detectar o primeiro *bit* recessivo, o que indica que todos finalizaram as notificações, devem enviar mais outros 7 *bits* recessivos, formando o delimitar da sinalização de sobrecarga. Se a sinalização de sobrecarga ocorrer depois do primeiro *bit* do intervalo entre *frames*, os nodos receptores entenderão o primeiro *bit* da sinalização como um SOF, e devido ao formato da sinalização, irão detectar um Erro de Enchimento (Bosch, 1991).

1.3.5 Erros de Comunicação

A CAN fornece dois tipos de *frames* de erro, a sinalização ativa composta de *bits* dominantes e a passiva composta de *bits* recessivos (Bosch, 1991). Inicialmente todos nodos podem realizar sinalizações ativas, entretanto a medida que a frequência de erros aumenta, mais tempo do barramento é ocupado por notificações, que possuem a capacidade de sobrescrever qualquer outra mensagem, diminuindo a eficiência da rede. Para fugir dessa situação o protocolo CAN fornece um mecanismo de confinamento em função do número atual de erros do nodo, quando um nodo é confinado do modo normal (Erro Ativo) para o primeiro modo mais restritivo (Erro Passivo), ele deixa de usar a sinalização ativa e passa a usar a passiva, que não tem a propriedade de sobrescrever outras mensagens. Ainda há mais uma possibilidade de confinamento, que vai do modo Erro Passivo para o modo *Bus Off*, no qual o nodo é virtualmente desligado do barramento, esse confinamento ocorre quando o nodo contabiliza mais do que 255 erros de transmissão.

Para saber em qual modo de operação deve funcionar, o controlador CAN executa uma contagem dinâmica, com possibilidade de incremento e decremento, dos erros de transmissão (TEC) e recepção (REC) separadamente (ISO, 2003a), e em função desses valores o controlador CAN muda seu modo de operação. Os contadores de erro, TEC e

REC, são incrementados e decrementados de acordo com as regras de incremento, subseção 1.3.5.2.

O mecanismo de detecção de erro pode ser sumarizado da seguinte forma: Um erro é detectado por um ou mais o nodos, que enviam uma sinalização, ativa ou passiva. A mensagem causadora do erro é descartada por todos os nodos receptores, e os contadores, em cada um dos nodos, são incrementados de acordo com as regras e pode haver ou não mudança no modo de confinamento. Na sequência o transmissor tentará reenviar a mensagem interrompida (Lawrenz e *et al*, 2013).

1.3.5.1 Tipos de erros e suas detecções

A CAN distingue os possíveis erros em cinco tipos não excludentes (Bosch, 1991), os Erros de *Bit*, os Erros de Enchimento (*stuffing*), os Erros de CRC - *Cyclic Redundancy Check*, os Erros de Forma (formato do *frame*) e os Erros de Reconhecimento (ACK - *Acknowledgment*).

O **Erro de *Bit*** (*Bit Error*) é detectado pelo transmissor quando o mesmo observa um valor diferente do qual ele tentou escrever (Bosch, 1991). O transmissor sempre vigia o barramento durante a escrita, e se ele observar um valor diferente do que escreveu, irá acrescentar uma unidade no seu contador de erros de transmissão, enviará uma sinalização erro ativa e após tentará reenviar o *frame* interrompido. Há três exceções para o Erro de *Bit*, todas relacionadas a detecção de um bit dominante no lugar de um recessivo, a primeira acontece durante a arbitragem, a segunda durante o *slot* de reconhecimento e última durante a sinalização de erro passivo. Na Figura 1.27 os campos e *bits* em destaque são checados pelo emissor incondicionalmente durante a transmissão de um *frame* de dados, por outro lado, os *bits* do campo de arbitragem só produziram um erro de *bit* se o controlador escrever um bit dominante e observar um recessivo (Bosch, 1991).

Figura 1.27: As regiões em destaque são monitoradas *bit* a *bit* pelo transmissor. O *bit* de ACK não é verificado pelo emissor, pois seu valor deve ser sobrescrito por um receptor. Fonte: Autores.

O **Erro de Enchimento** (*Stuffing Error*) é acusado pelo receptor sempre que ele

detectar seis *bits* consecutivos de mesmo valor (Bosch, 1991), o que provavelmente indica falha durante a codificação de enchimento da mensagem. O controlador que detecta esse erro avisa aos demais que aquela mensagem está comprometida emitindo uma sinalização de erro, assim o emissor entende que deve reenviar a mensagem. O erro de enchimento só pode ocorrer na fração do *frame* que é codificada com enchimentos, ou seja, do SOF até o penúltimo *bit* do campo CRC, o delimitar do CRC (CRC_{del}) não é codificado, na Figura 1.17 foram destacadas as parte codificada de um *frame* de dados.

O **Erro de CRC** acontece quando um receptor calcula um código CRC diferente do informado pelo emissor, no campo CRC do *frame* propagado (Bosch, 1991). O transmissor calcula o código de verificação cíclica redundante (CRC) a partir da divisão de um segmento do *frame* sem enchimentos, por um polinômio de 15° ordem (ISO, 2003a). O segmento do *frame* utilizado pelo emissor no cálculo do código CRC começa no SOF e vai até o fim do campo CRC sem contar seu terminador, porém as últimas 15 posições, onde deveria está o código CRC, são preenchidas apenas por zeros (ISO, 2003a). O receptor calcula o código CRC pelo mesmo polinômio, e para o mesmo segmento do *frame*, porém as últimas 15 posições que eram preenchidas por zeros alocam agora o código CRC do transmissor, e dessa forma a divisão deve retorna 0, no caso contrário um erro de CRC sera contabilizado, o receptor imitirá uma sinalização de erro e em seguida o transmissor tentará reenviar a mensagem interrompida.

O **Erro de Forma** (*Form Error*) ocorre quando o formato do *frame* é violado (Bosch, 1991). Há três regiões nas quais só podem conter bits recessivos, o delimitador do campo CRC, o delimitador do campo ACK e o campo EOF, como na Figura 1.28. Quando o nodo receptor detecta em uma ou mais regiões dessas um *bit* dominante ele contabiliza um erro de forma, emite uma sinalização de erro, e em seguida o emissor tentará reenviar a mensagem interrompida. Há uma exceção para o Erro de Forma, que ocorre quando o controlador detectar um *bit* dominante no local do último *bit* recessivo do EOF.

O **Erro de Reconhecimento** (*ACK Error*) é quando o emissor verifica que o *bit* recessivo que ele escreveu no *slot* de ACK, localizado após do delimitador do CRC, não foi sobrescrito por um dominante. Isso significa que nenhum nodo consegui receber corretamente a mensagem, e neste caso o emissor contabiliza um erro de reconhecimento.

Figura 1.28: Representação dos *Bits* de formatação do *frame* CAN, todos eles devem ser recessivos e são os únicos usados na verificação de erro de forma. O *bit* de ACK não é usada na verificação de forma, mas seu delimitador sim. Fonte: Autores.

1.3.5.2 O confinamento por erro

Se o protocolo CAN possuísse apenas a sinalização de erro ativa, aquela composta por *bits* dominantes, o mal funcionamento de alguns nodos poderia inviabilizar a rede. Por outro lado, se houvesse apenas *frames* de erro passivo, compostos por *bits* recessivos, um nodo que propaga a sinalização poderia nunca ser ouvido, visto que *bits* recessivos não sobrescrevem os dominantes.

Na CAN somente os nodos "saudáveis"podem realizar a sinalização ativa, nodos que contabilizaram mais do que 127 erros de transmissão (TEC) ou recepção (REC) são confinados no modo Erro Passivo, primeiro modo mais restritivo. Para regressar ao modo Erro Ativo, modo normal de operação, o nodo deve decrementar seus contadores de erro até que ambos possuam menos do que 128 contagens (ISO, 2003a). Se o nodo contabilizar mais do que 255 erros de transmissão ele é virtualmente desligado do barramento, sendo confinado no modo *Bus Off*, há dois requisitos para o nodo regressar deste modo ao modo Erro Ativo. A figura 1.29 representa a dinâmica de contagem e resume as regras de confinamento do protocolo CAN.

O modo **Erro Ativo** é o modo de operação normal, os nodos CAN confinados nele podem, receber e enviar mensagens pelo barramento, e sua sinalização de erro é ativa, composta por *bits* dominantes. Para permanecer neste modo nenhum dos seus contadores (TEC e REC) não devem ultrapassar 127 contagens, como visto na Figura 1.29.

No modo **Erro Passivo**, os nodos CAN ainda podem enviar e receber mensagens, porém como mencionado, só podem emitir sinalização passiva de erro, o que evita que o nodo interrompa a transmissão de mensagens propagadas por outros nodos. Além disso o nodo confinado no modo Erro Passivo que pretende enviar mais de uma mensagem deve escrever, e esperar, uma sequência de 8 *bits* recessivos apos os 3 *bits* que compõem o intervalo entre *frames*, antes de iniciar a transmissão de uma nova mensagem (ISO, 2003a).

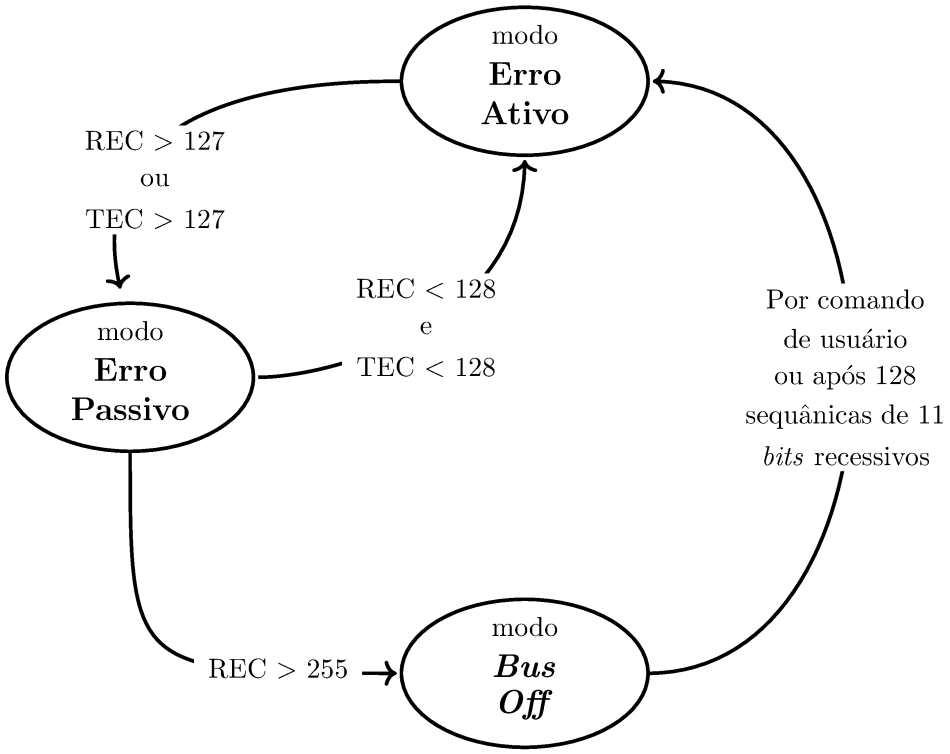


Figura 1.29: Modos de confinamento do protocolo CAN, Erro Ativo é o modo normal de operação. A medida que o erro aumenta o nodo migra para modos mais restritivos, e a medida que o erro diminui o nodo retorna. Fonte: Autores.

Isso resulta em um intervalo mínimo de 11 *bits*, entre duas mensagens consecutivas de um nodo confinado no modo Erro Passivo.

Os nodos confinados no modo ***Bus off*** não participam do gerenciamento do barramento e não podem transmitir nem receber mensagens, eles só podem regressar desse estado, por um comando de usuário e após observar 128 sequências de 11 *bits* recessivos (ISO, 2003a). Uma sequência de 11 *bits* pode ser observada em duas situações diferentes, durante um tempo equivalente de ociosidade do barramento ou apos a propagação com sucesso de uma mensagem seguida de um intervalo entre *frames* completo. Do delimitador ACK até o último bit do *frame* são contabilizados de 8 *bits* recessivos, como o espaço entre *frames* é composto de pelo menos 3 *bits*, será possível observar uma sequência de 11 *bits*, a cada mensagem propagada na rede. O problema é que nem sempre esse tempo de latência mínimo, intervalo entre *frames* é respeitado até o fim.

1.3.5.3 *Contagem dinâmica de erros*

O protocolo CAN fornece um mecanismo dinâmico de contagem de erros, os de recepção são contabilizados com REC (*Receive Error Count*), enquanto os de transmissão são contabilizados com TEC (*Transmission Error Count*). O incremento ou decremento dessas contadores segue um conjunto de regras, que serão apresentadas nos parágrafos seguintes. Vale ressaltar que o TEC só é acrescido de oito em oito unidades, enquanto o REC pode ser acrescido por uma unidade ou por oito unidades. Por outro lado os decréscimos são unitários.

O contador REC deve ser acrescido em uma unidade toda vez que o receptor detectar um erro. As exceções são quando o receptor detecta um Erro de *Bit* durante as sinalizações de sobrecarga ou de erro ativa nestes casos o REC deve ser acrescido de oito unidades (Bosch, 1991). O contador REC também deve ser acrescido em oito unidades se o receptor observar um *bit* dominante logo apos sua sinalização de erro ativo (Bosch, 1991), o que acontece durante o eco de erro. Além disso os nodos da CAN devem toleram uma sequência de até 7 *bits* dominantes após sinalizações de sobrecarga, de erro ativa ou passiva, o eco de sinalização pode estendê-la no máximo por 6 *bits*. Se isso for violado os receptores devem incrementar oito unidades no REC, e os transmissores devem incrementar oito unidades no TEC, (Bosch, 1991).

Ao detectar um erro de transmissão, o nodo deve enviar uma sinalização de erro e incrementar oito unidades no seu contador TEC (Bosch, 1991). Há duas exceções a essa regra, a primeira é quando um transmissor que está no modo Erro Passivo não detecta um *bit* dominante após o envio de uma sinalização de erro passiva. A segunda exceção é quando um o transmissor detecta um Erro de Enchimento durante a arbitragem, causado pela sobreposição de um bit recessivo por um dominante, o que pode indicar que ele perdeu a arbitragem. Em ambas exceções o TEC não deve ser acrescido (Bosch, 1991). O TEC deve ser acrescido de oito unidades se o transmissor detectar um Erro de *Bit* durante uma sinalização de sobrecarga ou de erro ativo (Bosch, 1991).

Se o REC possui um valor menor que 127, ele será decrescido por uma unidade toda vez que a recepção de um *frame* for confirmada com sucesso pelo mecanismo de reconhecimento (ACK). E se o REC possui um valor maior do que 127, lhe será atribuído um valor entre 119 e 127 (Bosch, 1991), e claro se ele valer zero permanecerá valendo zero. Já o TEC é

descrido toda vez que o transmissor conseguir completar uma transmissão com sucesso, e claro se ele já valer zero, não será diminuído (Bosch, 1991).

Capítulo 2

FURG CAN: Uma proposta de rede de sensores CAN

Com o intuito de explorar os materiais e métodos, elaborou-se a FURG CAN, uma rede de sensores, concebida para ser acessível, modular e confiável. Optou-se por elaborar uma rede de monitoramento, composta por nodos sensores e nodos *sink*, o que traz limitações de controle, pois não há um mecanismo prévio de comunicação bilateral entre dois nodos. Por outro lado, essa limitação possibilita a escolha de dispositivos populares e a adoção de soluções simples, resultando em um rede de fácil aquisição e manipulação. Além disso, essa limitação prévia não deve oferecer resistência a alterações, ou seja, com um pouco de trabalho é possível criar uma alternativa sobre a FURG CAN. Procurou-se escolher dispositivos populares e adeptos da filosofia *open-source*, pois uma grande comunidade de usuários pode favorecer a disseminação da proposta, enquanto que a licença *open-source* favorece reprodução e adaptação da plataforma.

A confiança da rede passa pela confiança e regularidade dos dispositivos usados, dessa forma optou-se por dispositivos bem conhecidos e testados. Por exemplo, para o subsistema de processamento escolheu-se a plataforma Arduino Nano (Arduino, 2020a) que possui uma grande comunidade de usuários e já foi bem explorada. O controlador CAN escolhido para compor o subsistema de comunicação, foi o MCP2515 (Microchip Technology Inc, 2019). Ele já foi experimentado pela mesma comunidade que o Arduino. Porém, em menor escala e, por isso, demandou grande parte das horas de programação. Foi necessário construir uma biblioteca para controlador CAN da Microchip, a *MCP2515 - bib* (Araújo et al., 2020).

Uma vantagem do MCP2515 é sua acessibilidade. A fabricante disponibiliza um bom material descritivo do controlador CAN (*datasheet*). Ele é facilmente encontrado no mercado, inclusive em módulos com o transceptor TJA1050 (Philips, 2003). O subsistema de comunicação da FURG CAN é composto por este Módulo CAN (MCP2515 + TJA1050).

A característica modular da FURG CAN vem do uso dos módulos: Placa de Desenvolvimento Arduino Nano, Módulo CAN (MCP2515 + TJA1050) e Módulo SD Catalex versão 2 (Aiea, 2020). O uso de módulos facilita a conexão entre os dispositivos, o reparo dos nodos e da rede.

Não se estabeleceu nenhum mecanismo extra de controle, o protocolo CAN já possui mecanismo de detecção de erro e isolamento por falhas. Além disso, o protocolo CAN possui a arbitragem que favorece os menores valores de ID (padrão e estendido) em possíveis disputas e, assim, evita colisões no barramento. Todos os mecanismos de controle do protocolo CAN são automaticamente executados pelos controladores CAN (MCP2515) conectados ao barramento. A rede resultante pode ser alterada durante seu funcionamento sem comprometer a comunicação, *i.e.*, nodos podem ser retirados, adicionados ou substituídos. Deve-se no entanto, respeitar os limites de operação da FURG CAN. Como por exemplo, o comprimento máximo barramento, o número de máximo de nodos e a taxa limite de ocupação do barramento. Estas e outras limitações são caracterizadas no capítulo 3.

2.1 *Layout* e componentes básicos

A topologia da FURG CAN é do tipo barramento com roteamento *broadcasting*, todos escutam todos, muito semelhante ao barramento clássico da CAN, apresentado na secção 1.3.1. Os nodos da FURG CAN distinguem-se em três tipos. O **CAN Sensor** (nodo sensor), responsável por mensurar e pré-processar as informações do meio. O **CAN Mon** (monitor), nodo do tipo *sink* responsável por encaminhar os dados para o computador. E o **CAN Save**, nodo do tipo *sink* responsável por armazenar os dados (nodo *data logger*) em um cartão micro SD.

Pode-se distinguir os tipos de nodos da FURG CAN pelo modo de operação. Nodos do tipo CAN Sensor só publicam. Seus controladores CAN (MCP2515) até observam a publicação de outros nodos e participam da validação de suas mensagens, porém, elas não são recolhidas para a memória interna do microcontrolador (Arduino Nano). Por outro lado, os nodos do tipo CAN Mon e CAN Save não publicam na rede. Eles recolhem as mensagens propagados pelos nodos do tipo CAN Sensor, e as transferem para um cartão micro SD no caso do CAN Save, e para o computador no caso do CAN Mon.

A Figura 2.1 apresenta um diagrama genérico da FURG CAN. Nele constam três nodos do tipo CAN Sensor (**S1**, **S2** e **S3**), um do tipo CAN Save (***Data Logger***), e um do tipo CAN Mon (**Monitor**) que é conectado a um computador (**PC**) via porta USB com comunicação Serial (USB-Serial). Os modos de operação dos nodos definem o fluxo de dados da FURG CAN, as setas indicam o sentido da propagação de dados. Repare que não há setas entrando nos nodos sensores, ou saindo do CAN Mon e do CAN Save.

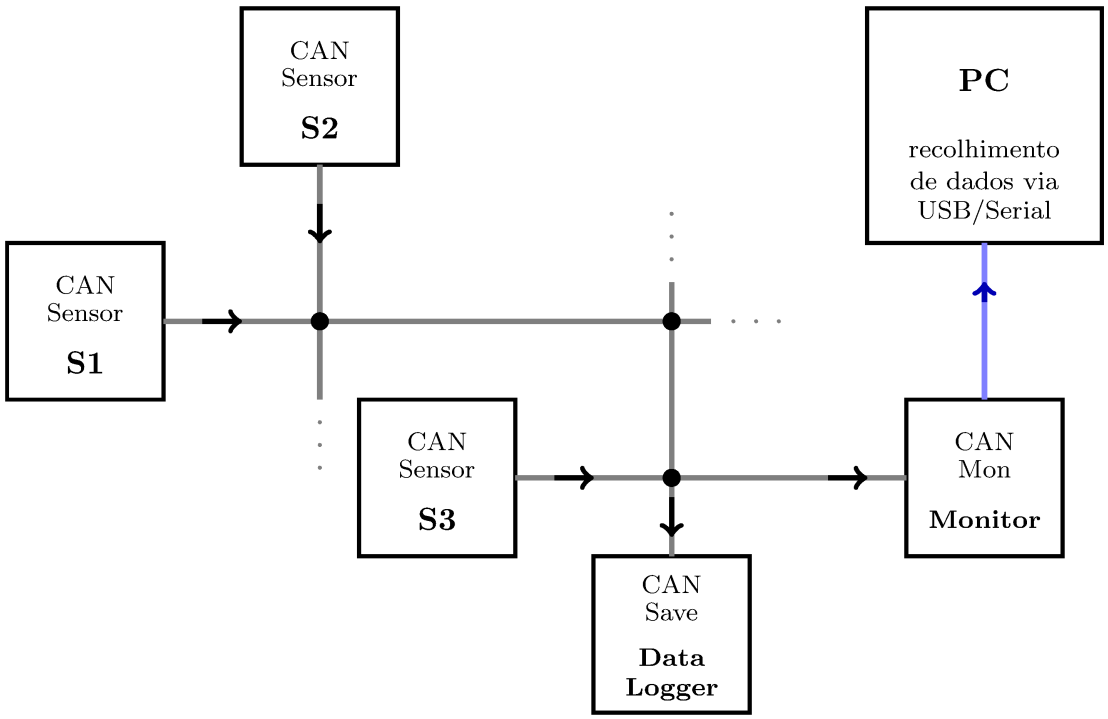


Figura 2.1: Diagrama simplificado da FURG CAN com três nodos CAN Sensor (**S1**, **S2** e **S3**), um nodo CAN Mon, um nodo CAN Save e um PC. As linhas cinzas simbolizam o barramento CAN, as setas simbolizam o fluxo de dados. O CAN Mon é conectado ao computador por um cabo USB, entretanto a comunicação é do tipo Serial. Fonte: Autores.

A Figura 2.2 apresenta uma foto de uma montagem da FURG CAN tirada durante uso, essa montagem foi utilizada na aplicação descrita no capítulo 4. Estão identificados na foto da Figura 2.2, um nodo CAN Mon (retângulo **A**) conectado a um Raspberry Pi (retângulo **3**). O Raspberry Pi fez a vez de um computador na rede apresentada pela foto. Um nodo CAN Save (retângulo **B**), dois nodos CAN Sensor (retângulos **C** e **D**), três conectores, dois RJ11 (círculos **1**), e um RJ45 (circulo **2**). O barramento usado na montagem da Figura 2.2 foi feito com quatro vias de um cabo Cat5 azul, visível no canto inferior da figura. As quatro vias do barramento foram distribuídas em dois pares trançados do cabo Cat5, um

par para linhas de dados (CAN HIGH e CAN LOW) e outro para as linhas de alimentação (VCC = 9 V e GND = 0 V). Os periféricos (mouse, teclado e monitor) aparentes na Figura 2.2 estão conectados ao Raspberry Pi, que no caso recebia as informações do nodo monitor e as armazenava. O retângulo identificado pela letra **A**, enquadra o nodo CAN Mon, e apresenta, na parte superior um Arduino Nano conectado a cabo USB e no canto inferior um Módulo CAN (MCP2515 + TJA1050) com *LED* acesso.

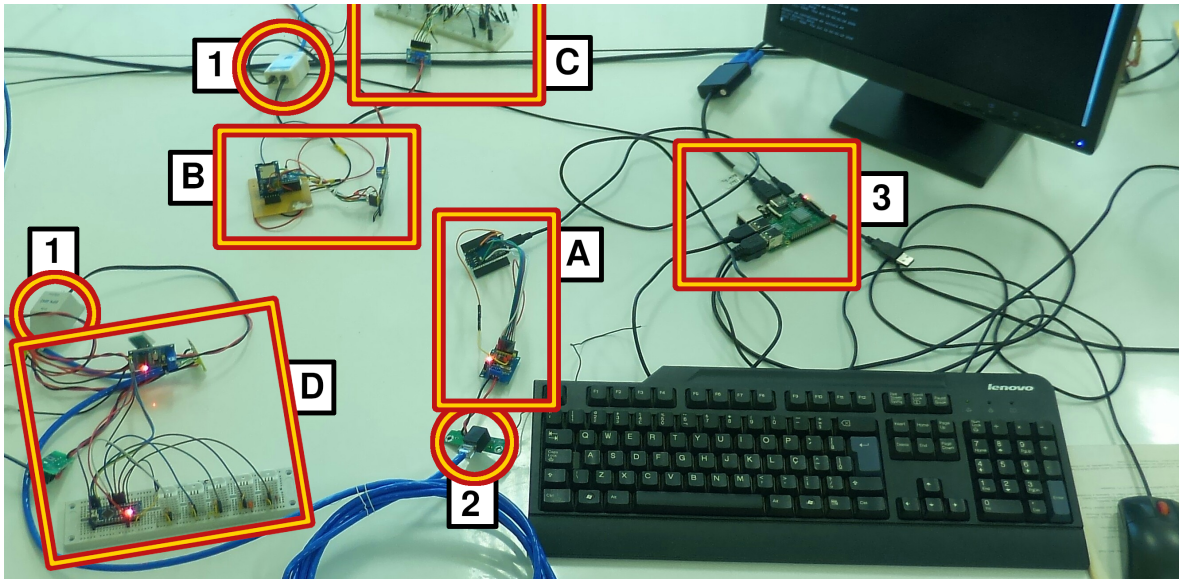


Figura 2.2: Foto da FURG CAN em uso. Nela estão identificados um nodo CAN Mon (A), um CAN Save (B), dois CAN Sensor (C e D), dois conector RJ11 (1), um conector RJ45 (2) e um Raspberry Pi Pi 3B (3). Além dos componentes identificados, pode ser visto no canto inferior um cabo Cat5 azul usado como barramento, e os periféricos (mouse, teclado e monitor) conectados ao Raspberry Pi. Fonte: Autores.

O nodo CAN Save da Figura 2.2 é identificado pela letra **B**, além do Arduino e do Módulo CAN ele possui um Módulo micro SD da Catalex versão 2 (Aiea, 2020), doravante Módulo SD. Há dois nodos CAN Sensor identificados na Figura 2.2. O Arduino do nodo **D** esta conectado sobre uma *protoboard*, placa branca na parte inferior do retângulo **D**, que também contém alguns transdutores. No canto superior do retângulo **D** está o Módulo CAN. O CAN Sensor D, recolhia os valores de temperatura e umidade dos transdutores e enviava para o CAN Mon e para o CAN Save pelo barramento CAN (cabo Cat5). Ainda na Figura 2.2 pode-se ver parcialmente, no canto superior, o nodo CAN Sensor **C**.

2.1.1 *Subsistema de processamento*

A unidade de controle ou subsistema de processamento é responsável por executar quase todos os processos dos nodos. Por exemplo, gerenciar as coletas e pré-processar as informações do meio em nodos do tipo CAN Sensor. Gerenciar o arquivamento de dados em nodos do tipo CAN Save. Além de gerenciar o subsistema de comunicação em todos os nodos. Comumente, as unidades de controle em redes de sensores são compostas por microcontroladores (Waltenegus e Poellabauer, 2010). Como já mencionado, o Arduino Nano foi escolhido como unidade de controle padrão para nodos da FURG CAN. É possível encontrar muitas soluções relacionadas ao sensoriamento usando a plataforma Arduino, desde de controle de conversores e transdutores, até tratamento estatísticos leves, como filtros digitais e médias.

Apesar da FURG CAN ser proposta inicialmente com o Arduino Nano, pode-se, com um pouco de trabalho, adaptar as rotinas para outras plataformas, com por exemplo, para o ESP32 , para o Arduino Uno e a Arduino Mega. Os nodos de uma rede não precisam ser compostos pelos mesmos microcontroladores, *i. e.*, pode-se ter numa mesma rede um nodo com o ESP32 , outro com o Arduino Mega. Ao escolher a plataforma de processamento, deve-se ter em mente as especificidades do nodo e, assim, escolher a plataforma mais adequada dentre as possíveis. Em geral, o código fonte de nodos do tipo CAN Mon e CAN Save não deve ser muito alterado, apenas em função da plataforma de processamento. Por outro lado, os nodos do tipo CAN Sensor, só compartilharam o mesmo código fonte no caso de serem compostos pelo mesmo microcontrolador e pelos mesmos dispositivos sensores.

2.1.2 *Subsistema de comunicação*

O subsistema de comunicação é composto, em nível de *hardware*, pelo controlador CAN MCP2515 (Microchip Technology Inc, 2019) e o pelo transceptor CAN TJA1050 (Philips, 2003). Eles formam o Módulo CAN, visível na Figura 2.3. Existem bibliotecas para o controlador CAN MCP2515 disponíveis na *web*, entretanto as bibliotecas testadas apresentaram limitações e/ou falhas. Por isso, foi desenvolvido pelos autores uma biblioteca para o CI da Microchip, a MCP2515-*bib* (Araújo et al., 2020). Apesar de não serem úteis para proposta (FURG CAN), os códigos das bibliotecas encontradas na *web* auxiliaram no desenvolvimento da MCP2515-*bib*, que está disponibilizada no Git Hub: KakiArdui-

no/MCP2515.

O módulo CAN, Figura 2.3, onde **U1**, o menor CI, é o TJA1050 e **U2** é o CI MCP2515. O controlador CAN precisa de um cristal para gerar o seu *clock* interno que, posteriormente, é usado no cálculo do *bit timing*. O cristal presente no módulo utilizado é de 8MHz. O módulo ainda possui um resistor para terminação e um *jumper* para conectá-la ou desconectá-la do barramento, conector amarelo no canto direito inferior.

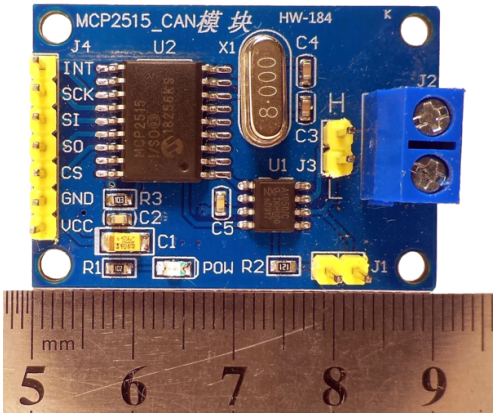


Figura 2.3: Módulo CAN (MCP2515 + TJA1050). Fonte: Autores.

O TJA1050 é um transceptor especificado para redes CAN *High Speed*, compatível com padronização *ISO 11898*, ele realiza a conversão diferencial do nível barramento (CAN HIGH - CAN LOW), para o TTL - *Transistor–Transistor Logic* e, separa os sinais de escrita e leitura. O TJA1050 pode trabalhar em um barramento de até 1 Mbit/s, com até 110 nodos e seu tempo de atraso de propagação máximo é de 250ns (Philips, 2003).

O CI MCP2515 é um controlador CAN 2.0 B autônomo, *i. e.*, ele realiza internamente e, de maneira independente, todas as operações diretamente relacionadas ao protocolo CAN 2.0 B, como por exemplo, a contagem dos erros e a validação de frames. O CI MCP2515 pode trabalhar em um barramento de CAN de até 1 Mbit/s, ele possui dois *buffer*s para recebimento (RXB0 e RXB1) e três para transmissão (TXB0, TXB1 e TXB2). Ele, também, possui máscaras e filtros que ajudam a selecionar os *frames* de interesse antes mesmo de serem alocados nos *buffer*s de entrada. Além disso, possui pinos de interrupção configuráveis, que auxiliam nos processos de leitura e escrita no barramento. A interface de comunicação do CI MCP2515 é SPI, e pode trabalhar em até 10 MHz.

A conexão física entre o Arduino e o Módulo CAN segue o padrão SPI, as linhas **SCK** - *Serial Clock*, **SI** - *Slave In* e **SO** - *Slave Out* são conectadas, respectivamente

nas entradas digitais 13, 11 e 12 do Arduino Nano. A linha **CS** - *Chip Select* pode ser conectada em qualquer pino digital, sendo por padrão em nodos da FURG CAN o pino digital 4. O pino **INT** - *Interrupt*, não faz parte da comunicação SPI, mas pode ajudar durante a comunicação gerando interrupções de acordo com o envio e ou recebimento de mensagens. O Arduino Nano possui apenas dois pinos que podem ser usados como entrada de interrupção o digital 2 e o digital 3 (Arduino, 2020b). Em nodos da FURG CAN o pino **INT** é, por padrão, conectado no pino digital 2, e em caso de mudança o código deverá ser alterado também. Ao utilizar outras plataformas Arduino deve-se revisar os pinos de conexão, pois variam de microcontrolador para microcontrolador.

A biblioteca MCP2515-*bib* foi elaborada em C++ e funciona em compiladores para plataforma Arduino e compatíveis. Nela é usado a abstração de objetos, que facilita o uso final. Ela auxilia na construção do código de configuração e operação do CI MCP2515, fornecendo funções de alto nível que substituem as operações de baixo nível exigidas pelo CI. Em destaque, tem-se as funções de leitura e escrita no barramento CAN, respectivamente *readDATA()* e *writeDATA()* que, provavelmente, serão as mais usadas pelo usuário final. Juntas elas podem operar até 72 registros internos do CI MCP2515. Uma descrição das funções da biblioteca MCP2515-*bib* é disponibilizada no Apêndice B.

2.1.3 Barramento da FURG CAN

O barramento da FURG CAN é composto por três vias, um par trançado para dados, CAN HIGH e CAN LOW, e uma via para o GND (terra), pode-se ainda adicionar um quarta via para o VCC (tensão de alimentação positiva). É possível utilizar um barramento CAN com apenas as duas linhas de dados, entretanto o funcionamento dos transceptores pode ser prejudicado, devido a diferenças de potencial entre os GND's dos nodos. Para evitar isso o barramento da FURG CAN usa uma via para o compartilhamento do GND.

Os cabos padrões para redes TCP/IP (TCP - *Transmission Control Protocol*, IP - *Internet Protocol*) são opções razoáveis e acessíveis para a implementação de barramentos do tipo CAN Eles possuem quatro pares trançados e diferem entre si principalmente pela blindagem, e robustez. O Cat5e e o Cat6 são exemplos populares de cabos para redes TCP/IP. Outra vantagem de usar cabos TCP/IP padrões é que seus conectores padrões, por exemplo, o RJ45, também são facilmente encontrados no mercado. A Tabela 2.1 apresenta algumas características elétricas dos cabos Cat5e e Cat6, fornecidas pela fabricante

Furukawa, e os respectivos valores normatizados pela ISO 11898-2:2003 (ISO, 2003b).

Tabela 2.1 - Especificações nominais de pares trançados para barramentos CAN fornecido pelo ISO 11898-2:2003 e para cabos Cat5e e Cat6 da fabricante Furukawa. Fonte: ISO 11898-2:2003(E) e www.furukawalatam.com.

Parâmetro	ISO 11898-2			Cat5e e Cat6
	min.	nom.	max.	fab. Furukawa
Impedância Ω	95	120	140	100 ± 15
Resistência Ω/m		$0,07$		$0,0938$
Atraso ns/m		5		$5,45$

A principal característica que um par trançado deve atender para ser usado em linhas de dados CAN é a impedância. Em geral, cabos Cat5e e Cat6 possuem uma impedância dentro da normatização. Em implementações da FURG CAN com cabos para TCP/IP, um par trançado deve ser usado para as linhas de dados, e outra via qualquer para compartilhar o GND. Restando ainda no cabo, uma via e dois outros pares trançados. Os fios restantes podem ser usados para outros fins, *e.g.*, a via trançada junto ao GND pode ser usada para compartilhar a tensão positiva (VCC) de uma fonte de alimentação. E as demais vias, podem ser usados para outros dois pares de dados (CAN HIGH e CAN LOW), podendo ser estabelecido até três barramentos CAN em paralelo.

Até o momento, a configuração utilizada e testada é composta de dois pares trançados, um para dados e outro para alimentação (VCC e GND), em cabos Cat5e e Cat6. Caso escolha-se fornecer um linha de alimentação junto ao barramento, aconselha-se usar uma tensão superior a exigida pelo Arduino (5 V), por exemplo, 12 V ou 24 V e, caso necessário, adicionar conversores de tensão locais para cada nodo, isso diminui a corrente propagada na linha. A placa de desenvolvimento do Arduino Nano possui um conversor de tensão interno, podendo ser alimentada pelo pino *Vin* com uma tensão de 7 à 12 V (Arduino, 2020a).

Como discutido na sub-secção 1.3.1.1 do capítulo 1.3, a impedância do barramento, entre as linhas de dados, deve possui um valor fixo dentro do especificado pela norma ISO 11898-2. Isso é garantido pela duas terminações de 120 Ω (ISO, 2003b). Os módulos CAN, utilizados pela FURG CAN, já possuem os resistores de terminação. As terminações são associadas ao barramento é feita pela conexão de um *jump* do módulo CAN. Apenas os nodos extremos do barramento devem possuir a terminação, somente seus módulos devem

possuir esse *jump* conectado. Nos demais módulos CAN, dos nodos intermediários, o *jump* deve estar desconectado. Os resistores da terminação devem possuir valor nominal de $120 \pm 10\Omega$ e suportar no mínimo 220 mW de dissipação (ISO, 2003b).

A Figura 2.4 apresenta o diagrama, mais detalhado, de um exemplo de configuração da FURG CAN. Nele pode se ver os resistores de terminação ($\mathbf{R_T}$) nos nodos extremos do barramento, no **CAN Mon** ao topo, e no **CAN Sensor n**. Os resistores de terminação ($\mathbf{R_T}$) estão localizados no canto esquerdo da Figura 2.4 (no Módulo CAN) ao lado esquerdo do TJA1050, próximo da conexão dos nodos com o barramento. O barramento representado no diagrama da Figura 2.4, localizado na extrema esquerda, é composto por dois pares trançado, o da extrema esquerda é usado para alimentação 5 V, e o outro, ao seu lado direito é usado para dados. De modo a simplificar o diagrama, da Figura 2.4, a mesma tensão positiva (VCC) alimenta paralelamente os módulos CAN e os Arduino Nano e por isso vale 5 V.

O **CAN Mon** aparente no topo da Figura 2.4 é conectado a um computador (**PC**) via USB com comunicação Serial. Os componentes do nodo CAN Mon estão delimitados por um retângulo pontilhado, que contém um Arduino Nano no canto direito, e um Módulo CAN na esquerda. O Módulo CAN possui internamente um controlador CAN **MCP2515** e um transceptor **TJA1050**, como pode ser visto no diagrama. Das conexões internas do Módulo CAN só estão representadas, na Figura 2.4, aquelas diretamente relacionadas com a comunicação e a alimentação.

A comunicação entre o Arduino Nano e o Módulo CAN é do tipo SPI e usa os pinos **13** para **SCK** (*Serial Clock*), **11** para **MOSI** (*Master Out Slave In*), **12** para **MISO** (*Master In Slave Out*) e **4** para **CS** (*Chip Select*). Ainda é, utilizado um pino extra (**2**) para interrupções (**INT**) geradas pelo MCP2515. Essas conexões estão explicitadas no diagrama da Figura 2.4. Os nodos são conectados ao barramento da esquerda através do Módulo CAN.

Também é representado na Figura 2.4 um nodo **CAN Save**, segundo de cima para baixo. O nodo CAN Save possui, na direita, um **Módulo SD** que também é conectado ao Arduino pelo protocolo SPI. Porém, via *software* e por isso usa pinos diferentes do Módulo CAN. Os pinos do Arduino usados na SPI, via *software*, com o Módulo SD são: **MISO** pino **5**, **MOSI** pino **6**, **SCK** pino **7** e **CS** pino **8**. Ainda, há dois nodos sensores na Figura 2.4, o **CAN Sensor 1**, terceiro de cima para baixo. E o **CAN Sensor n**, último de cima

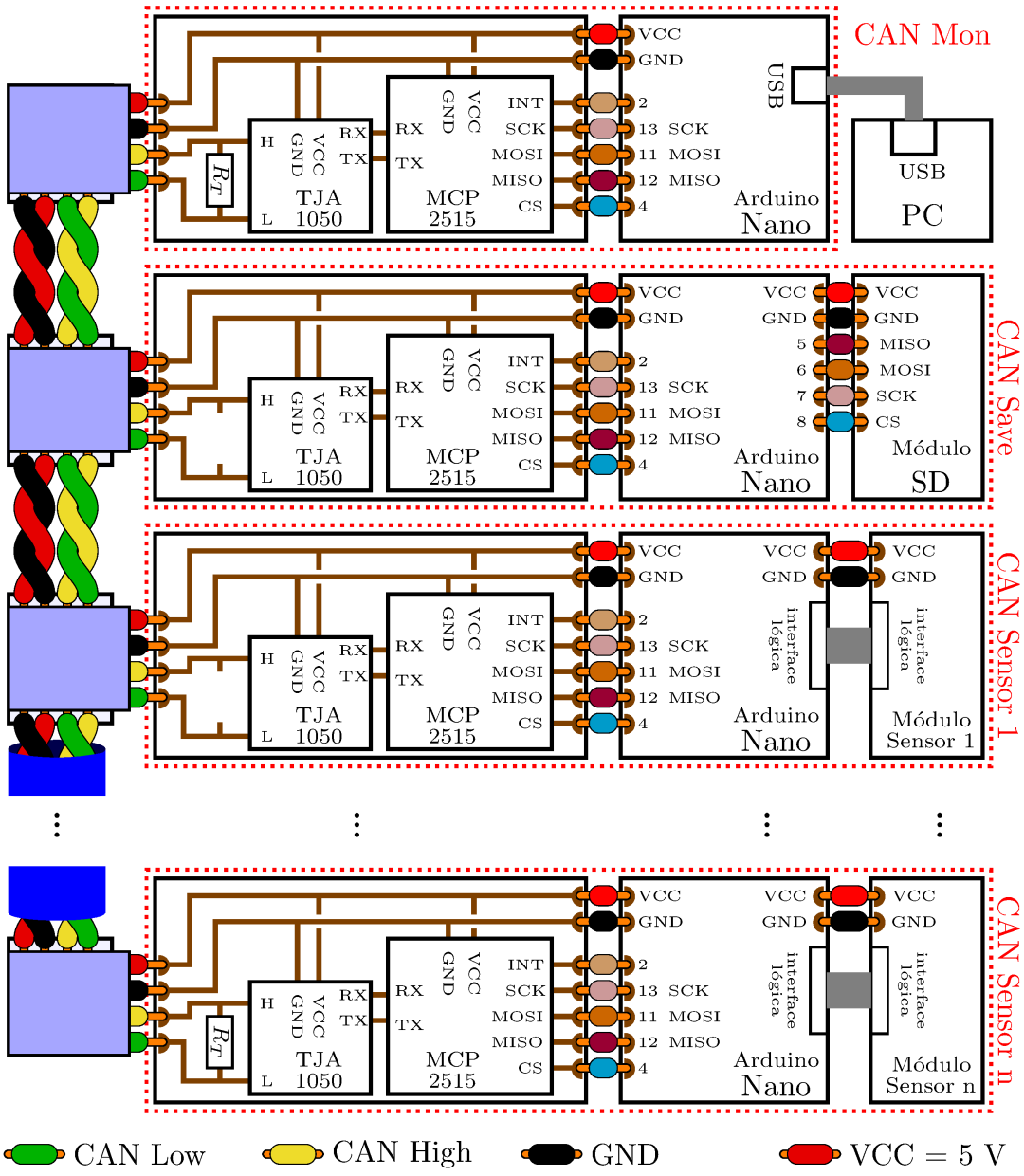


Figura 2.4: Representação do barramento da FURG CAN. Fonte: Autores.

para baixo. Os módulos sensores, conectados ao Arduino Nano pela direita, são apenas representações genéricas.

Como mencionado no capítulo 1.3, o comprimento máximo do barramento está relacionado com a temporização do transceptor, com a frequência de operação da rede e com a configuração do *timing bit*. A FURG CAN pode operar em três frequências padrões com as seguintes distâncias máximas teoricamente estimadas: 125 kHz/s até 275 m; 250 kHz/s até 125 m e 500 kHz/s até 50 m. O número máximo de nodos que podem ser conectados

depende principalmente do *drive* do transceptor e da carga no barramento. No caso da FURG CAN esse limite é dado pelo TJA150 como sendo 110 nodos (Philips, 2003). Vale ressaltar que, apesar da FURG CAN ser uma rede de sensores para monitoramento, ela pode ser usada para tráfego de outros tipos de informações. Claro que deve-se ter em mente as limitações da FURG CAN antes de qualquer implementação, seja para monitoramento ou não.

2.1.4 *Os frames de dados da FURG CAN*

As mensagens da FURG CAN possuem uma formatação semelhante as mensagens na subcamada LLC de controladores CAN, apresentada na Figura 1.15. As mensagens da LLC possuem três campos, o campo de ID, com o identificador, o campo DLC (*Data Len Code*) que, armazena o número de dados (de 0 a 8). E por fim o campo de dados com os *bytes* de dados. A diferença é no campo ID das mensagens da FURG CAN, que é desmembrado em dois, um para o ID padrão e outro para a extensão de ID. Isso facilita a comparação de prioridade entre as mensagens, em caso de duvida ver a subseção 1.3.3.5, onde o mecanismo de arbitragem foi explicado. Resultando em quatro campos, o campo **ID padrão**, **Extensão de ID**, **N° de bytes**, **Campo de dados**, , a Figura 2.5 apresenta um diagrama do formato das mensagens da FURG CAN.

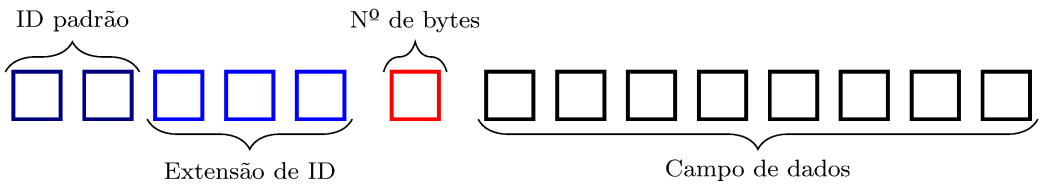


Figura 2.5: Organização dos *bytes* em *frames* de dados da FURG CAN. Fonte: Autores.

2.2 *CAN Mon o nodo monitor*

O CAN Mon é o nodo mais simples, em nível de *hardware*, da FURG CAN, sendo composto apenas pelo módulo CAN e pelo Arduino Nano, como pode ser visto nos diagramas da Figura 2.6 e da Figura 2.7. Apesar de simples, o CAN Mon possui uma grande responsabilidade, adquirir todos os *frames* propagados na rede, e enviá-los ao computador

para que possam ser processados ou, ao menos, armazenados.

O controlador CAN MCP2515 do CAN Mon é programado com *rollover* do *buffer* RXB0 para o RXB1, ou seja, os *frames* recém recebidos pelo controlador CAN são sempre armazenados no *buffer* de entrada RXB0. Quando o RXB0 estiver ocupado, o *frame* antigo é transferido para o *buffer* RXB1, esvaziando o RXB0, que receberá o novo *frame*. A transferência de informação de um *buffer* para outro é chamado de *rollover*. Todos os filtros e máscaras de aceite do controlador CAN MCP2515 são desativados, e qualquer *frame* é aceito pelo CAN Mon.

O pino INT do CI MCP2515 é configurado para gerar uma interrupção, alterando seu estado de 1 para 0, sempre que um novo *frame* for recebido. Essa variação informa ao Arduino Nano que uma nova mensagem está pronta para ser lida no *buffer* de entrada RXB0 do controlador CAN. Por isso, além das conexões do barramento SPI, o Arduino Nano e o Módulo CAN, são conectados através do pino INT. Para que a sinalização do MCP2515 funcione uma interrupção física é programado no Arduino Nano, que vigia o pino digital 2. Assim que o Arduino Nano perceber a variação, borda de descida, executa a função de leitura readCAN(), mesmo que, para isso, tenha que interromper a execução de alguma outra sub-rotina.

Os *frames* lidos pelo Arduino Nano são armazenados temporariamente em um *buffer First In First Out* (*First In First Out*) interno que pode armazenar até n *frames*. Onde n é um valor programável, para o Arduino Nano, recomenda-se $n = 50$. Pode-se alterar n de acordo com a disponibilidade de memória dinâmica. O compilador do Arduino IDE recomenda não ocupar mais de 75 % da memória dinâmica, pois isso pode resultar em instabilidade. Sempre que possível, o Arduino Nano tentará enviar os *frames* armazenados pela porta serial. A prioridade de envio é por ordem de chegada. Os primeiros a serem recebidos, serão os primeiros a serem enviados, *first in first out*. O diagrama da Figura 2.6 representa o fluxo interno de dados em nodos do tipo CAN Mon.

Na extrema-esquerda do diagrama da Figura 2.6 tem-se a representação do módulo CAN, que é conectado pelo barramento SPI e pelo pino INT ao Arduino Nano. No extremo direito tem-se o computador, que é conectado ao Arduino Nano pelo barramento USB-Serial. Ao centro da Figura 2.6 tem-se o bloco que representa o Arduino Nano. E dentro dele, estão representadas as instruções executados sobre os dados. Primeiro, o CI MCP2515 sinaliza o recebimento de uma mensagem pelo pino **INT**. Assim que percebe a interrupção

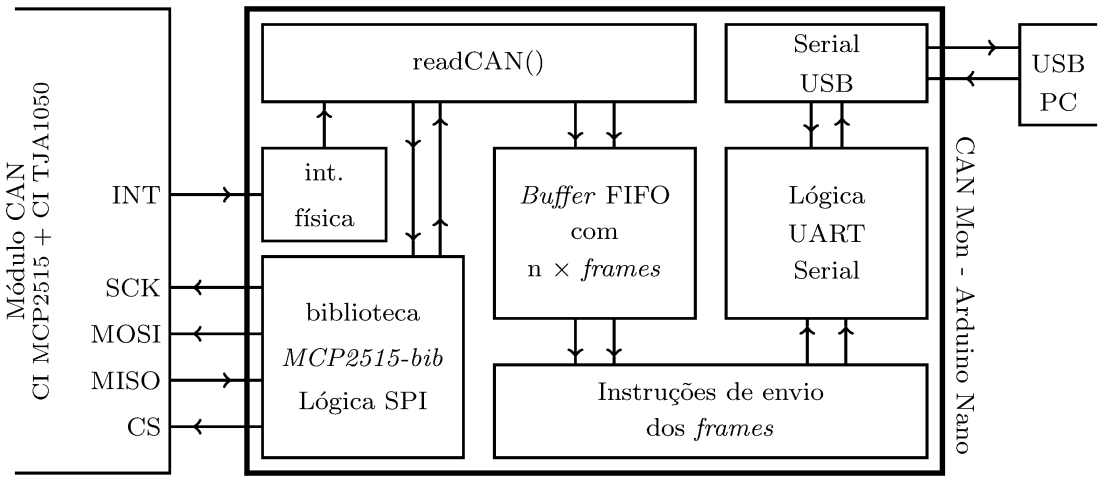


Figura 2.6: Diagrama do fluxo de dados de nodos do tipo CAN Mon. Fonte: Autores

(**int. física**), o Arduino Nano chama a função **readCAN()**. A função **readCAN()**, utiliza comandos da **biblioteca MCP2515-*bib*** para ler os *frames* armazenados no CI MCP2515, que em seguida, são alocados no ***buffer First In First Out*** interno do Arduino.

As **Instruções de envio dos *frames*** são executadas de forma cíclica, no *void loop()*, elas transfere as informações do ***buffer* FIFO** para o *buffer* de saída da serial do Arduino Nano que possui 64 *bytes*. O bloco que representa as **Instruções de envio dos *frames*** no diagrama da Figura 2.6 é localizado no canto inferior direito do Arduino Nano. Do *buffer* de saída da serial, os *frames* são enviados ao **PC** pela porta USB, passando antes pelo conversor **Serial USB** da plataforma Arduino.

Como mencionado, o CAN Mon é composto pelo Módulo CAN e pelo Arduino Nano. Deve ter um computador para recebimento dos dados recolhidos pelo CAN Mon. A Figura 2.7 apresenta um diagrama das conexões de nodos do tipo CAN Mon. O Módulo CAN é associado ao barramento da extrema-direita pela conexão do pino **H** com a linha **CAN HIGH**, do pino **L** com a linha **CAN LOW** e, do pino **GND** a linha **GND**.

Do outro lado, o módulo é conectado ao Arduino, pela conexão do pino **INT** ao **D2**. Do pino **SCK** ao **D13**. Do pino **SI** ao **D11**. Do pino **SO** ao **D12**. Do pino **CS** ao **D4**. Além das conexões do **VCC** e do **GND**. Dos pinos usados na conexão do Arduino com Módulo CAN somente o **D2** e o **D4** podem ser alterados, desde de que o *software* também seja alterado. Os demais são, específicos da comunicação SPI. No canto superior

esquerdo, o Arduino é conectado via barramento USB ao computador. O nodo CAN Mon não é conectado com a linha de tensão positiva (9 V) do barramento, é alimentado pelo computador.

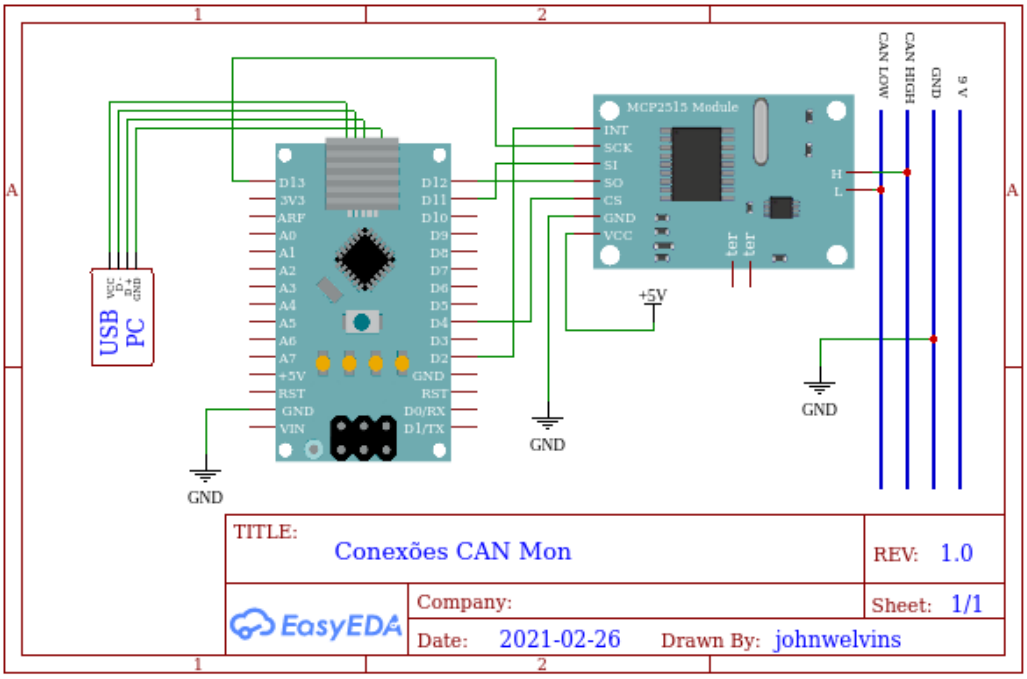


Figura 2.7: Diagrama de conexões do CAN Mon.

Há dois mecanismos importantes no funcionamento do CAN Mon. O primeiro é, o uso de uma interrupção física para indicar o recebimento de um *frame* válido pelo CI MCP2515. O segundo é, a criação de um *buffer* interno, no CAN Mon, para alocar, temporariamente, *n frames*, até que estes possam ser enviados pela porta serial para um computador. A interrupção física, torna o processo de captura de *frames* independente da rotina cíclica do Arduino (*void loop()*) e, ainda, garante prioridade frente a qualquer outro processo. O resultado final, do uso da interrupção física, é a diminuição do tempo de espera médio do *frame* no *buffer* de entrada do CI MCP2515, o que minimiza a sobreposição de *frames* (*overflow*) no *buffer* RXB0.

Com o *buffer* interno, segundo mecanismo, o CAN Mon pode realizar até n leituras sequenciais sem a necessidade de transferir *frames* para o computador, diminuindo temporariamente o tempo médio de processamento, visto que a maior parte desse tempo o CAN Mon gasta na transferência de *frames* para o computador. Pode-se entender que o *buffer* interno possibilita ao CAN Mon operar em um barramento com uma taxa de ocupação

superior a sua capacidade, durante um período equivalente ao tempo de propagação de no máximo *n frames*. Entretanto, deve-se garantir que a taxa de ocupação média do barramento não ultrapasse a capacidade de processamento do CAN Mon. Se isso acontecer o CAN Mon, com um *buffer* interno, perderá *frames*.

A princípio, as publicações no barramento não são ordenadas ao longo de um tempo global, referência para todos nodos, até mesmo porque não há sincronismo entre os nodos da FURG CAN. Isso permite uma aglomeração temporal de frames, o que resulta, durante um intervalo de tempo, uma taxa de ocupação do barramento superior a projetada. O *buffer* interno foi projetado para evitar perdas durante um período de aglomeração de *frames*. A capacidade de processamento do CAN Mon é explorada na seção 3.13.

O código fonte do CAN Mon é um exemplo da biblioteca MCP2515-*bib* e está disponibilizado no Apêndice B.4.5. A princípio o código do CAN Mon não será alterado, mas caso seja necessário realizar adaptações pode-se consultar a documentação da biblioteca MCP2515-*bib*, disponível no Apêndice B.

2.3 *CAN Save o nodo data logger*

O funcionamento interno do CAN Save é muito semelhante ao do CAN Mon. A diferença é que nodos do tipo CAN Mon enviam os *frames* recebidos pela USB-Serial, enquanto que nodos CAN Save salvam os dados em um cartão micro SD. O CI MCP2515 de nodos CAN Save é configurado da mesma forma que no CAN Mon, sem filtros nem máscaras de aceite, com o *rollover* do RXB0 para o RXB1 e para gerar interrupções sempre que recebe uma nova mensagem. A Figura 2.8 apresenta um diagrama onde são representados os processos internos do CAN Save. A diferença do diagrama do CAN Mon, apresentado na Figura 2.6, está no lado direito. Onde tinha-se a interface serial UART, tem-se agora a interface SPI (via *software*) controlada pela biblioteca *SDFat.h*. Onde se tinha o computador, tem-se agora o Módulo SD.

No diagrama da Figura 2.8, as instruções de envio de *frames* pela serial do CAN Mon, são substituídas pela função **SDprint(...)**. Sempre que possível, a função **SDprint(...)**, recolhe os *frames* salvos no **Buffer FIFO** interno, do Arduino, e os grava no cartão micro SD, com auxilio da **biblioteca *SDFate.h*** (Greiman, 2009). Os *frames* recolhidos são salvos no formato apresentado na Figura 2.5 em um arquivo *.csv*, que terá, no máximo, 10

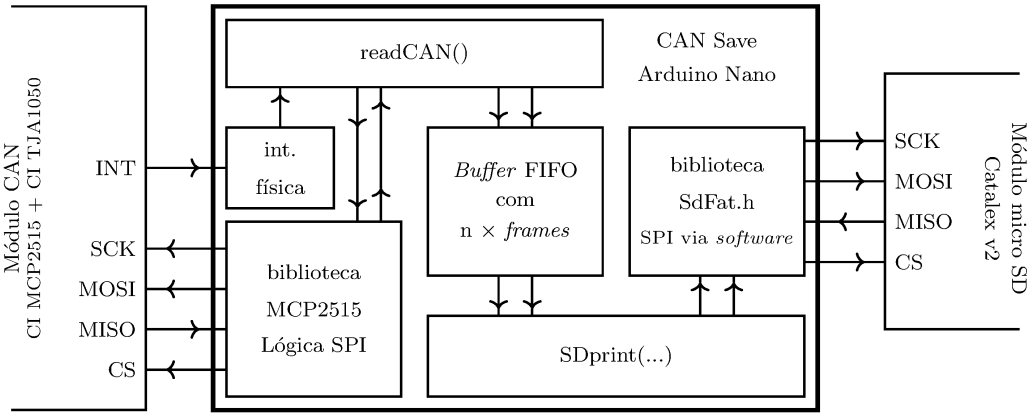


Figura 2.8: Diagrama do fluxo de dados de nodos do tipo CAN Save. Fonte: Autores

M *bytes*. Se o arquivo ultrapassar esse tamanho, um novo arquivo será criado. O código do CAN Save, é um exemplo da biblioteca MCP2515-*bib* e está disponibilizado no Apêndice B.4.6.

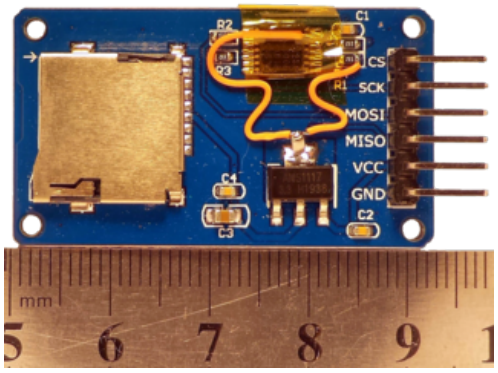


Figura 2.9: Catalex SD módulo 1, com modificação para operar da mesma forma que a ver. 2.

Cartões SD podem se comunicar por dois protocolos, o *SDIO* e o SPI. O Arduino Nano só possui a SPI. O nível lógico da comunicação de cartões SD é 3,3 V. Para que ele seja conectado ao Arduino Nano é necessário um conversor de nível, no caso o módulo da Catalex já possui esse conversor, o que facilita a conexão. Entretanto, há um erro de projeto nos módulos Catalex v1 que impossibilita o compartilhamento do barramento SPI. Esse erro é explorado por Aiea em sua *home page* (Aiea, 2020), onde ainda indica uma modificação que corrige esse problema. A princípio esse problema não deve ser crítico, uma vez que o CAN Save utiliza o SPI via *software*. De qualquer modo e, em razões de testes anteriores, todos módulos Catalex v1 utilizados já estavam modificados. Vale ressaltar que

os módulos micro SD, são em geral, conversores de sinal (3,3 V para 5 V), sendo possível construir um módulo micro SD a partir de um adaptador SD micro SD. A pinagem do SD e do micro SD podem ser consultadas na página sobre o SD da Wikipédia. Também há outras opções de módulos SD no mercado. O módulo apresentado na Figura 2.9 está modificado.

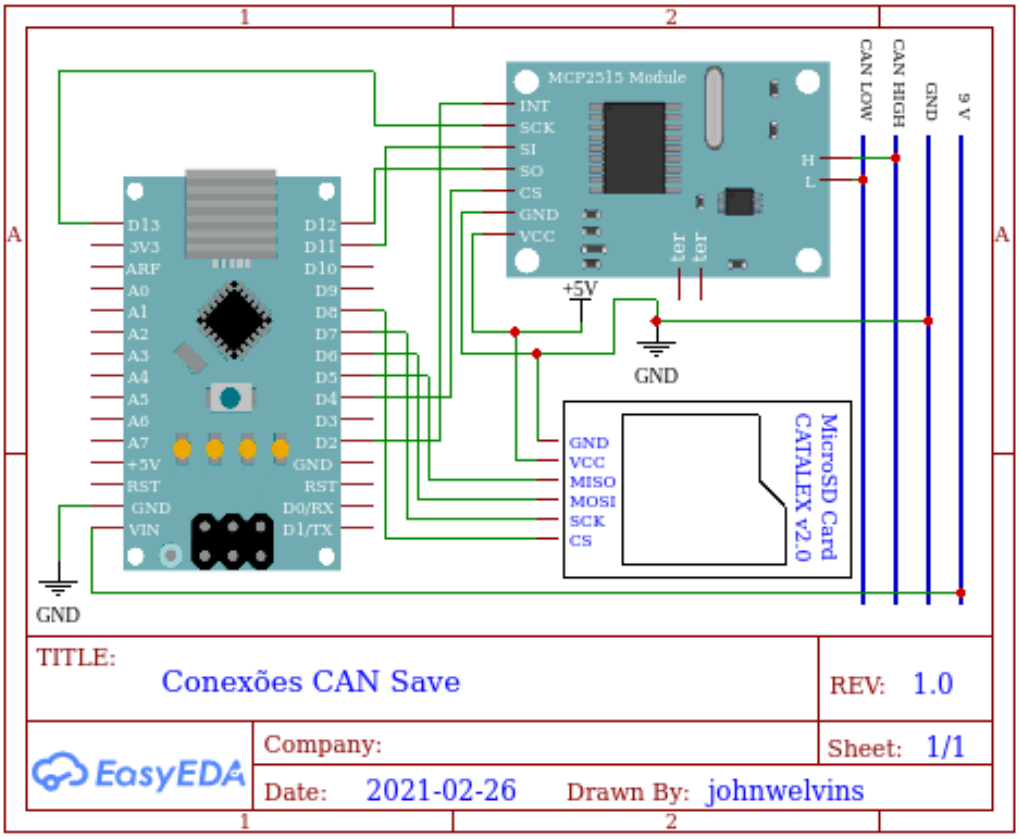


Figura 2.10: Diagrama de conexões do CAN Save.

As conexões dos módulos SD e CAN com o Arduino Nano são apresentadas na Figura 2.10, como mencionado, são utilizados dois barramentos SPI um para o Módulo CAN e outro para o Módulo SD. Da parte do Módulo CAN a conexão é a mesma já apresentada para o CAN Mon. A conexão do Módulo SD com o Arduino utiliza o pino digital 8 para **CS**, o 7 para **SCK**, o 6 para **MOSI** e o 5 para **MISO**. O diagrama da Figura 2.10 fornece um exemplo de barramento com 4 linhas, as duas de dados, a do GND e uma extra de alimentação (9 V). A tensão fornecida pelo barramento da FURG CAN, no diagrama, é conectado ao Arduino pelo **Vin** e convertida, internamento, para 5 V. Em barramentos que não fornecem alimentação o nodo CAN Save deve ser alimento separadamente. Neste

caso a conexão com barramento da FURG CAN fica igual ao apresentado para o CAN Mon na Figura 2.7.

2.4 CAN Sensor

Os nodos do tipo CAN Sensor são os responsáveis por coletar as informações desejadas, e encaminhá-las para o CAN Mon e/ou para o CAN Save através do barramento da FURG CAN, eles ainda podem pre-processar as informações antes do envio. A configuração destes nodos varia muito de aplicação para aplicação, entretanto as instruções relacionadas ao gerenciamento do envio e a formatação dos *frames* não precisa mudar, nem a conexão com módulo CAN.

O código fonte de um CAN Sensor pode ser construído a partir de alguns dos exemplos da biblioteca MCP2515-*bib*, o exemplo *CANSensor.ino* disponível em B.4.7 fornece um exemplo de código fonte genérico mais completo para nodos sensores da FURG CAN, e seu funcionamento será um explicado mais abaixo. Pode usar ainda o *CANTX.ino* que é o programa de transmissão mais simples e direto da MCP2515-*bib*, que também é um de seus exemplos. Além desdes estão disponíveis na MCP2515-*bib* outros 9 exemplos, dentre os quais destaca-se o *CANSensor_ADS1248_Pt100.ino* que controla o ADS1248 (Texas Instruments, 2016a) lendo dois Pt100 a quatro fios e o *CANSensor_31856.ino* que controla o conversor de termopar MAX31856 (Maxim Integrated, 2015).

A Figura 2.11 apresenta o diagrama genérico dos processos executados pelo Arduino Nano de um nodo sensor baseado no exemplo *CANSensor.ino*. O fluxo de dados dentro do CAN Sensor possui sentido oposto em relação ao de nodos do tipo CAN Mon e CAN Save, desta vez a informação é gerada por um ou mais **dispositivos sensores**, representados na extrema-direita da Figura 2.11, lida pelo Arduino Nano através de uma interface lógica e possivelmente **pré-processada**. Os processos internos do CAN Sensor descritos até agora, variam de aplicação para aplicação, porém os processos seguintes não precisam mudar pois são específicos da FURG CAN.

Uma vez que a informação já foi lida e pré-processada, o Arduino Nano do CAN Sensor chama a função **putinBuff(...)**, representada na parte inferior ao centro da Figura 2.11. A função **putinBuff(...)** constrói o *frame* de dados da FURG CAN, o armazena no *Buffer FIFO* interno do Arduino, e caso haja algum *buffer* de saída do CI MCP2515 vazio, a

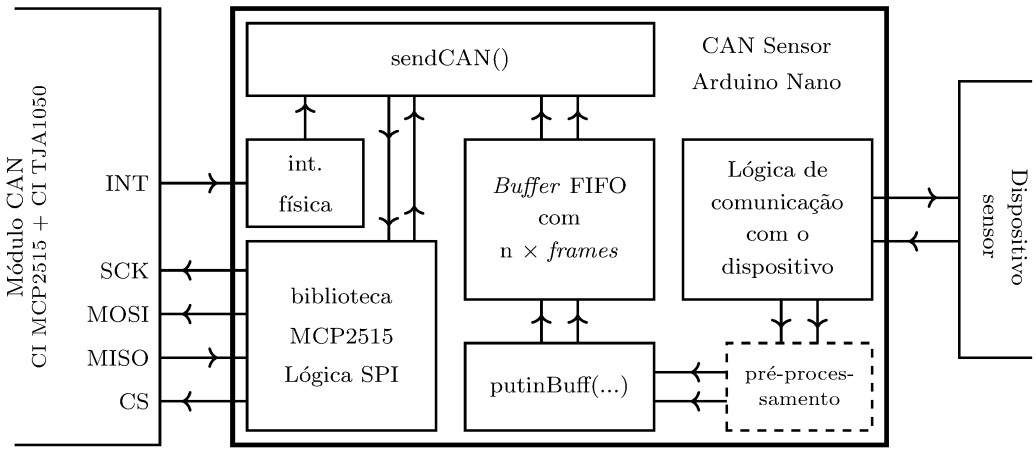


Figura 2.11: Diagrama genérico do fluxo de dados de nodos do tipo CAN Sensor. Fonte: Autores

função **sendCAN()** é chamada, sendo ela é responsável por alocar o *frame* no *buffer* de saída do CI MCP2515 e solicitar seu envio pelo barramento CAN. No caso dos *buffer*s de saída do CI MCP2515 estarem todos ocupados, a **putinBuff(...)** apenas adiciona o *frame* no *Buffer FIFO* do Arduino.

A configuração do CI MCP2515 de nodos CAN Sensor, difere da configuração dos nodos CAN Mon e CAN Save apenas pelo pino **INT**, que desta vez é configurado para gerar uma interrupção toda vez o *buffer* de saída TXB0 do controlador CAN for desocupado. A interrupção física (**int. física**) do Arduino é programada para chamar a função **sendCAN()**, processo representado no canto superior esquerdo do bloco central da Figura 2.11. Quando a função **sendCAN()** é chamada pela interrupção, ela irá transferir até dois *frames* armazenados no *Buffer FIFO* do Arduino para os *buffer*s de saída do CI MCP2515 e solicitará o envio do mesmo pelo barramento CAN. Os *frames* serão alocados no CI MCP2515 de modo que o mais antigo seja enviado primeiro, se não houver *frames* armazenados no *Buffer FIFO* do Arduino a função **sendCAN()** apenas zera uma variável de controle que habilita e desabilita a interrupção física do Arduino, de acordo com a ocupação do *buffer* interno e TXB0 saída do controlador CAN (MCP2515).

Há um universo de dispositivos periféricos, com implementações junto à plataforma Arduino ou compatíveis, documentada e divulgada pela comunidade de usuários. Muitos destes dispositivos já são comercializados em módulos que facilitam a sua utilização, com módulos RTC - *Real Time Clock* e próprio Módulo CAN. A princípio qualquer dispositivo, ou quase todos, compatíveis com a plataforma Arduino, podem ser associados a um CAN

Sensor. Neste sentido nodos CAN Sensor possuem boa flexibilidade de projeto, também há um conversor analógico digital de 10bit interno ao Arduino Nano, o que fornece uma resolução de 1024 níveis digitais, a referência pode ser interna, de 5V ou 1, 1V, ou ainda pode ser fornecida externamente, a MCP2515-*bib* possui um exemplo com leitura analógica *CANSensor_LM35.ino*. Para medidas com mais resolução pode-se adicionar dispositivos específicos, como o ADS1248 (Texas Instruments, 2016a) que já possui biblioteca (Ferrando et al., 2017).

As conexões de *hardware* variam de acordo com os periféricos adicionados, a conexão entre o Arduino Nano e o Módulo CAN segue a mesma já apresentada para os nodos CAN Mon e CAN Save. Se o barramento da FURG CAN tiver uma linha de alimentação a conexão do Nodo Sensor com o barramento segue a mesma apresentada para o CAN Save na Figura 2.10. Entretanto se ele for alimento externamente a conexão com o barramento da FURG CAN deve ser como a apresentada para o CAN Mon na Figura 2.7.

Capítulo 3

Caracterização

Os testes da caracterização foram divididos em três grupos, cada um com uma montagem diferente. A Figura 3.1 apresenta uma foto da segunda montagem, nela também pode-se ver alguns dos elementos usados nas demais montagens.

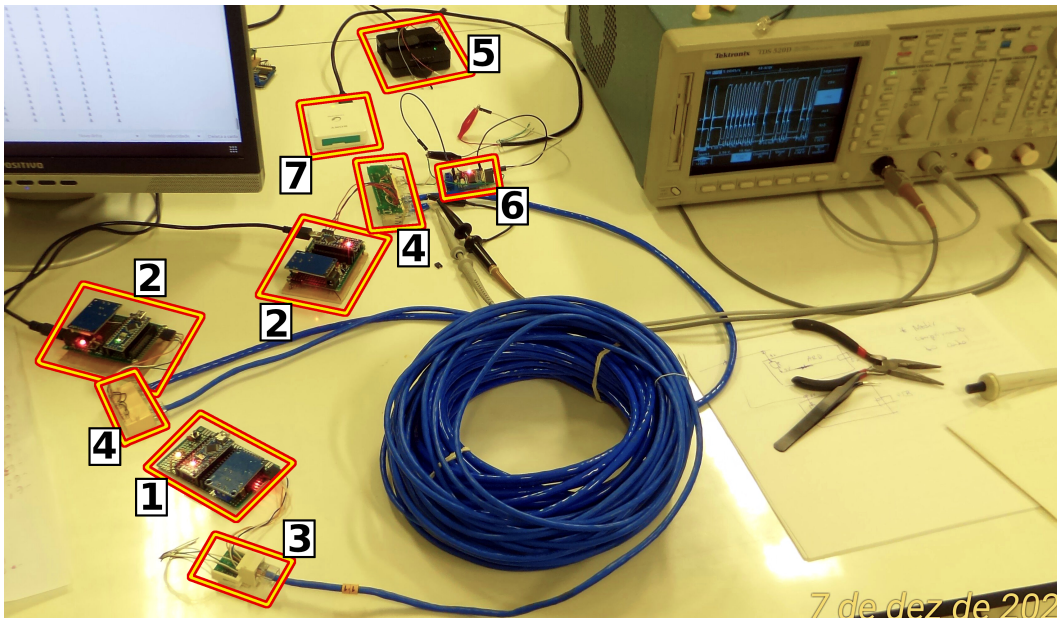


Figura 3.1: Protótipo da rede FURG CAN utilizado na caracterização. Os componentes identificados são: (1) Nodo monitor (MCP2515-Arduino Nano); (2) Nodos emissores (MCP2515-Arduino Nano); (3) Conector RJ45 comercial; (4) Conector RJ45 reciclado; (5) Analisador Lógico DSLogic; (6) Módulo CAN modificado para conectar o analisador lógico a rede CAN; (7) Bateria (fonte isolada) de 5 V. O osciloscópio TDS 520D da Tektronix, no canto superior, foi usado nos teste contra falhas. O cabo Cat6 azul contém as 4 vias do barramento CAN. Fonte: Autores.

Todos os testes foram realizados em um barramento composto por dois pares trançados, um para as linhas de dados (CAN HIGH e CAN LOW) e outro para alimentação (GND e 5V), em um cabo Cat6 de 14,35 m, o cabo azul apresentado no centro da foto da Figura

3.1. Dos três nodos visíveis na foto, apenas o nodo monitor, número 1, é alimentado pela rede, os outros dois, número 2, são alimentados pela entrada USB do Arduino Nano e, um deles fornece a alimentação ao barramento.

Na primeira montagem, foi investigado a formatação dos *frames* gerados pelos nodos seguem o protocolo CAN. Para isso, o sinal do barramento foi observado por um analisador lógico. Nesse teste foi utilizado um nodo monitor (1 na Figura 3.1) sem computador, ele serve apenas para validar dos frames (gerar o ACK *bit*). Além do monitor, foram usados dois nodos emissores identificados com pelo número 2 na Figura 3.1. Foi utilizado um analisador lógico DSLogic conectado a um módulo CAN modificado para acessar o sinal de saída do TJA1050 e, uma bateria de 5 V para alimentar esse módulo, respectivamente identificados pelos números 5, 6 e 7 na Figura 3.1. Também, foi utilizado um computador pessoal para coletar os dados do DSLogic. Ainda estão identificados na foto da Figura 3.1, os conectores utilizados nos testes, número 3, um conetor RJ45 comercial e, número 4, dois conectores retirados de sucatas de roteadores.

Na segunda montagem foi verificado se a FURG CAN suporta falhas de curto, *teste a* e *teste b*, como previsto na ISO 11898-2 (ISO, 2003b). Também foi verificado o comportamento do sinal em função da existência ou não das terminações de 120 Ω, *teste c* e *teste d*. Dos dispositivos já mencionados da Figura 3.1, não foram utilizados na segunda montagem o analisador lógico, o módulo CAN modificado e a bateria. O sinal do barramento foi monitorado com um osciloscópio, TDS 520D da Tektronix, visível no canto superior direito da Figura 3.1.

Na terceira montagem, foi realizado para determinar os tempos e leitura de nodos CAN Mon e, o de escrita de nodos CAN Sensor, de modo a obter parâmetros limitadores da FURG CAN. Para isso foi utilizado um nodo monitor, um nodo emissor, um contador de frequência HP 5316B, no modo medição de período de pulso e um osciloscópio TDS 210 da Tektronix para monitoramento do sinal.

3.1 Formatação dos frames

Como apresentado na seção 1.3.4, as mensagens do protocolo CAN possuem formatos específicos, ou seja, alguns *bits* dos *frames* devem possuir sempre o mesmo valor, *e.g.*, o *bit* ACK (*Acknowledged*) deve ser dominante (0), enquanto seus vizinhos, o CRC *del*

(delimitador do campo CRC - *Cyclic Redundancy Check*) e o ACK *del*, devem ser recessivos (1), ver seção 1.3.5.1. Sendo assim, objetiva-se com o primeiro teste, capturar o sinal do barramento, reconstruir um *frame* de forma independente aos dispositivos já usados na FURG CAN e, verificar a sua formatação. Para isso, o analisador DSLogic foi conectado ao pino RXD de um TJA1050 acoplado as linhas de dados (CAN HIGH e CAN LOW) do barramento. Os dados coletados pelo analisador foram salvos em um arquivo '.csv', que possui a informação temporal das transições de estado do barramento. Os nodos sensores usados neste teste, foram programados para gerar *frames* de dados com ID padrão aleatório, extensão de ID conhecida (218 453) e, 8 *bytes* de dados, todos com o mesmo valor (10). Durante esse teste foram capturados 124,23 M *bytes*, nas três frequências compatíveis da FURG CAN, 125 k *bit*/s, 250 k *bit*/s e 500 k *bit*/s. Aqui será descrito um dos *frames* capturados quando o barramento operava em 500k *bit*/s.

A estratégia usada para decodificar os *frames* foi a seguinte: Construir um *clock* virtual para a referência, com um período de 2× *Bit timing* do sinal capturado. E a cada variação de nível do *clock* virtual (2× por período) procurar, no ponto temporal mais próximo do sinal, o valor do *bit*. A rotina de decodificação foi elaborada em *Python 3*.

O primeiro passo foi obter o *Bit timing* e verificar seu valor, para gerar o *clock* virtual. Em uma primeira aproximação, pode-se assumir o *Bit timing* como o menor intervalo entre duas transições de estado. No caso a ser apresentado foi de 1,90 µs, próximo do esperado (2 µs). Porém pode haver divergência entre o *clock* dos controladores CAN. Nesse sentido, filtrou-se todos valores menores que do que 3,80 µs, $2 \times 1,90$µs (menor valor), ou seja, descartou-se intervalos com 2 *bits* de duração ou mais, selecionando todos intervalos com um *bit* de duração.

Ao todo 4 310 547 intervalos foram selecionados, com um valor médio de $2,004 \pm 0,096$µs e, mediana de $2,000$µs. A Figura 3.2 apresenta um histograma dos valores selecionados pelo filtro, pode-se perceber que o valor mais provável é ≈ 2µs, como esperado.

O pico próximo aos 1,9 µs e o próximo aos 2,1 µs na Figura 3.2, podem ser resultado da combinação das flutuações dos *clocks* dos nodos emissores e, do analisador lógico. Outra possibilidade seria a superposição de estados, de um dominante por outro, que prolongaria o *bit* e, de recessivo por um dominante, que encurtaria o *bit*. Neste caso, a alteração na duração do *bit* será proporcional a distância entre os nodos. As sobreposições de estados é prevista no protocolo CAN e não compromete a comunicação, além disso, só podem

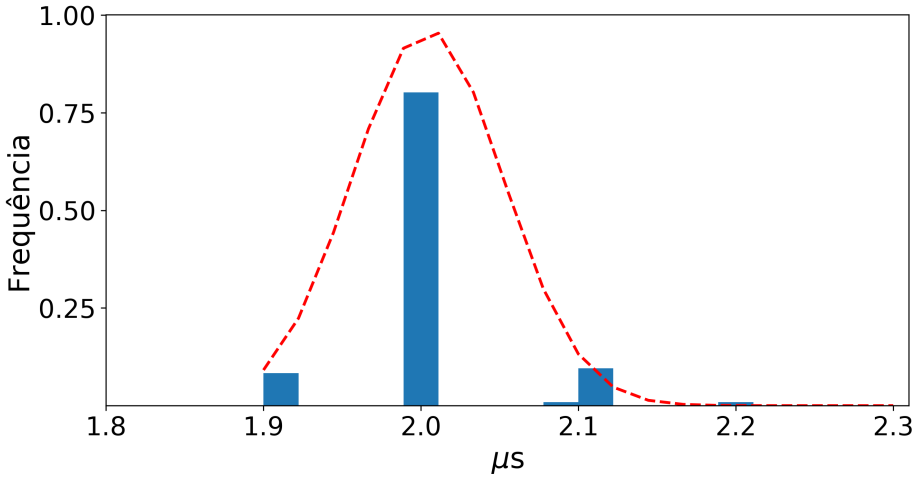


Figura 3.2: Histograma de valores encontrados para o *Bit timing* na lista de frames analizada. Fonte: Autores.

acontecer durante a arbitragem ou durante o ACK *bit*. Conhecendo o *Bit timing* pode-se gerar o *clock* virtual e obter os valores dos *bits*.

Para exemplificar a decodificação, será apresentado um dos *frames* capturado com o analisador lógico. Esse *frame* está segmentado em quatro figuras, a 3.3, 3.4, 3.5 e 3.3. Todas as quatro figuras possuem a mesma estrutura Na parte superior está o *clock* virtual de referência em vermelho e, logo abaixo, o nível do barramento em função do tempo, curva azul. Além disso, todos os *bits* estão nomeados no *clock* e seus valores são explicitados entre as duas curvas.

Na Figura 3.3 pode-se ver o *bit* **SOF** - *Start Of Frame*, terceira posição (0 µs) em ralação ao *clock* virtual e, como esperado o SOF é dominante, 0 digital. Depois do **SOF** tem-se os 11 *bits* do ID padrão (do **ID10** ao **ID0**) que, no caso apresentado é 397 (0x18D, em hexadecimal). Na sequência do ID padrão aparecem os bits **SRR** e **IDE** (*flag* de ID estendido). O primeiro é um *bit* de formato fixo em frames estendidos e deve valer 1, valor do estado recessivo digital, como do *frame* apresentado, o que pode ser verificado pelo estado recessivo do *bit* **IDE**.

Ainda na Figura 3.3, depois da *flag* de ID estendido, vem os *bits* da extensão de ID (do **IDe 17** ao **IDe 0**) que vale 218 453 (0x35555) como esperado. Repare que há um *stuffing bit* dominante (em vermelho) entre o **IDe 16** e o **IDe 17**. Isso ocorreu devido a sequência de 5 *bits* recessivos, do **IDe 0** 0 até o **IDe 16**. O enchimento ocorre mesmo que o próximo *bit*, após a sequência de 5 iguais, possua um valor oposto. Isso se deve a forma serial que

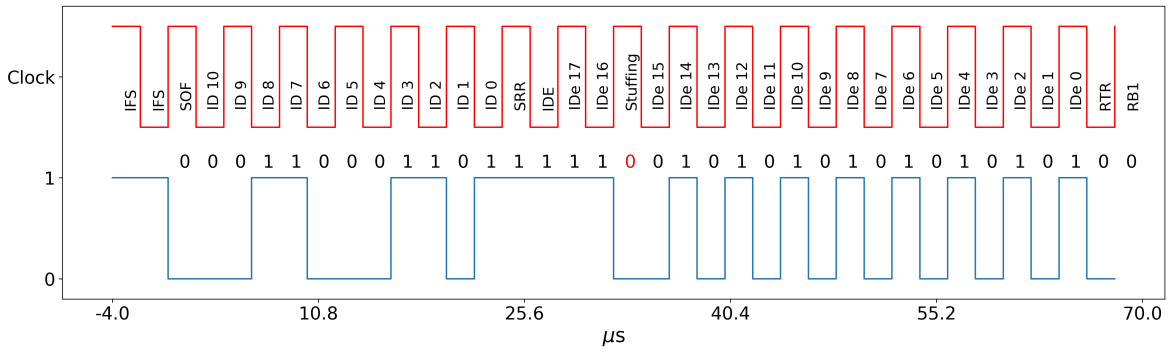


Figura 3.3: Primeiro segmento (1/4), de um *frame* genérico da lista estudada. Fonte: Autores.

o *frame* é codificado (enchido), *i.e.*, no momento de adicionar o enchimento o controlador ainda não sabe o valor do próximo *bit*. Depois do **IDe 0**, tem-se o *bit* **RTR**, que indica se o *frame* é um pedido remoto (1), ou não (0), o *frame* apresentado é de dados, logo o **RTR** deve valer 0. Os *bits* **RB1** (último da Figura 3.3) e o **RB0** (primeiro da Figura 3.3) possuem formato fixo e devem ser dominantes.

Na Figura 3.4, segundo segmento do *frame* e, após o **RB0** tem-se os quatro *bits* do código de comprimento (do **DLC 3** ao **DLC 0**), que valem 8 (b1000) e, na sequência começam os *bits* do *Data Byte* 7, do **DB7 7** ao **DB7 0**. Todos os *bytes* de dados devem possui o valor 10. Isso pode ser verificado nos últimos 4 *bits* (**DBX 3**, **DBX 2**, **DBX 1** e **DBX 0**) de qualquer um dos *Data Bytes* pela sequência b1010.

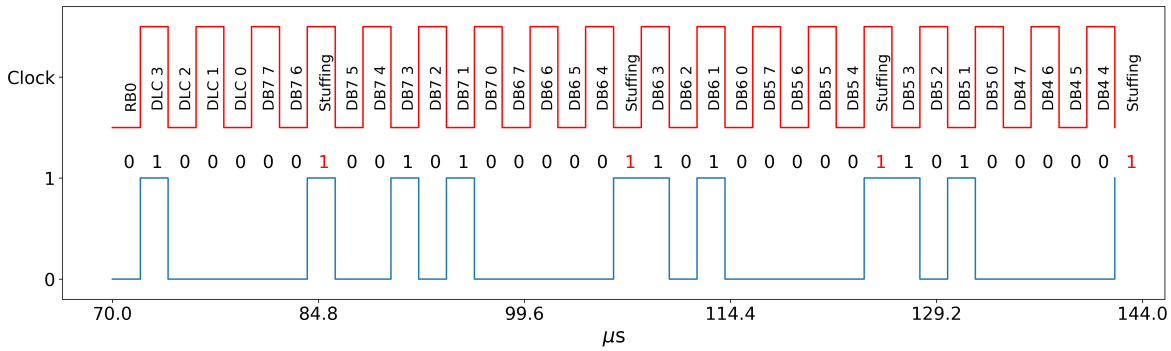


Figura 3.4: Segundo segmento (1/4), de um *frame* genérico da lista estudada. Fonte: Autores.

Durante o campo de dados não há *bits* de formato fixo e, todos os quatro *bytes* de dados, possuem o mesmo valor. Logo a única coisa que resta verificar é a presença dos *bits* de enchimentos, que aparecem 8 vezes ao longo do campo de dados. Nas figuras 3.4 e 3.5,

sempre depois de uma sequência de 5 *bits* dominantes.

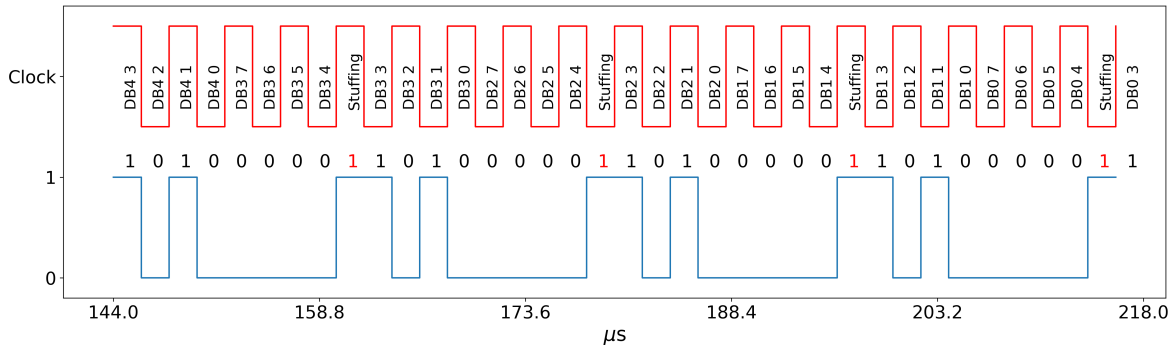


Figura 3.5: Terceiro segmento (1/4), de um *frame* genérico da lista estudada. Fonte: Autores.

Tem-se, no início do último segmento do *frame* Figura 3.6. Os três últimos *bits* do campo de dados, **DB0 2**, **DB0 1** e **DB03**. Após o campo de dados, tem-se os 15 *bits* do CRC (do **CRC 14** ao **CRC 0**), seguido pelo *bit* delimitador do campo, **CRC *del***, que deve ter um estado recessivo. Depois do **CRC *del*** vem o **ACK** *bit* que deve ser um estado dominante definido pelos outros nodos (ouvintes) da rede. Este *bit* serve como uma confirmação de reconhecimento do *frame* (subseção 1.3.4). Logo, o valor 0 no **ACK** *bit* significa que este *frame* específico foi reconhecido como válido por, pelo menos, um dos outros dois nodos presentes na rede. Os últimos *bits* do *frame* devem ser recessivo, a começar do delimitador do ACK (**ACK *del***), seguido pelo campo EOF (*End Of Frame*) com 7 *bits* recessivos, do **EOF 6** ao **EOF 0**. Por fim, tem-se o IFS (*Inter Frame Space*) que, deve possuir, ao menos, três menos *bits* recessivos. E deve durar até o início do próximo *frame*, logo, seu comprimento é indeterminado.

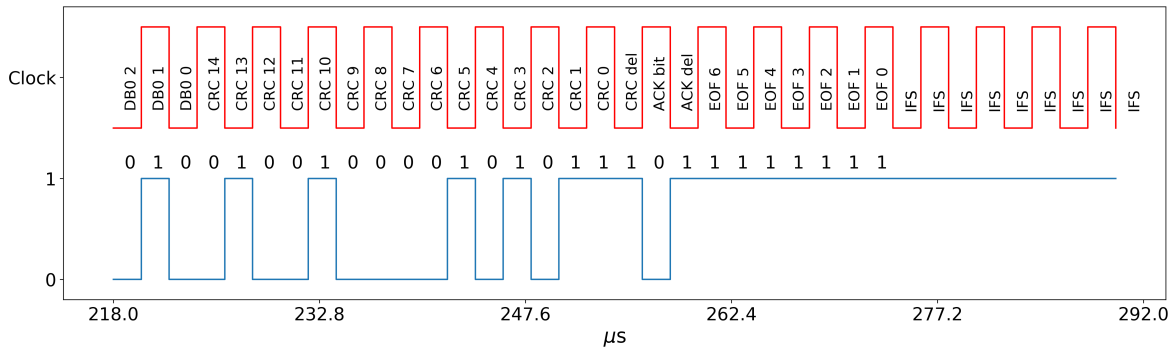


Figura 3.6: Quarto segmento (1/4), de um *frame* genérico da lista estudada. Fonte: Autores.

Os *frames* observados com o analisador lógico, seguem o formato previsto pelo pro-

tocolo e, as informações recuperadas, extensão de ID e *bytes* de dados, condizem com as programadas.

3.2 Teste contra falhas

O barramento CAN deve ser imune a certas falhas, redes do tipo *Fault-Tolerant Low-Speed* devem ser tolerantes a um número maior de falhas do que o tipo de rede estudada, *High-Speed*. Barramentos *High-Speed* devem ser imunes a dois tipos de curto, da linha CAN LOW para o GND e, da CAN HIGH para o VCC (ISO, 2003b). A Phillips afirma, no *data sheet* do TJA1050 (Philips, 2003), que seu transceptor é imune as duas falhas citadas.

Para a análise de tolerância a falhas da rede FURG CAN serão considerados dois casos, (*teste a*) linha CAN LOW em curto-circuito com a linha GND (0 V) e (*teste b*) linha CAN HIGH em curto-circuito com a linha VCC (5 V). Em ambos os casos (*a* e *b*) o transceptor TJA1050 é alimentado com $VCC = 5\,\text{V}$ e $GND = 0\,\text{V}$. O barramento é o mesmo usado na montagem anterior, dois pares trançados de um cabo Cat6, com 14,35 m de comprimento.

Essa montagem também é usada para verificar a dependência do barramento em relação as duas terminações de 120 Ω. Nesse sentido, são realizados dois testes. (c) a ausência de uma das terminações e, (d) a ausência das duas terminações. Barramentos sem terminação ou com a terminação errada podem possuir uma comunicação menos estável e até mesmo inviável.

3.2.1 Testes de curto

A entrada do TJA1050, lado conectado ao barramento, é composto por um estágio analógico. No datasheet o limite de tensão das linhas de dados (CANH e CANL), em relação ao GND, é de $-27\,\text{V}$ a 40 V (Philips, 2003). De acordo com a ISO 11898-2 (ISO, 2003b), em condições normais, as linhas de dados devem possuir 2,5 V em relação ao GND durante estado recessivo. Em estado dominantes a linha de dados CAN HIGH deve possuir 3,5 V e a linha CAN LOW 1,5 V.

O mais relevante no sinal da rede CAN é a diferença CAN HIGH - CAN LOW. Que deve ser de normalmente 0 V em um estado recessivo, mas é tolerável dentro da faixa de $-120\,\text{mV}$ à 12 mV (ISO, 2003b). Em um estado dominante a diferença CAN HIGH - CAN

LOW deve ser de 2 V, sendo tolerável a faixa de 1,2 V à 3 V (ISO, 2003b). De acordo com o *data sheet* o limite inferior para o estado dominante do TJA1050 é de 1,5 V (Philips, 2003), maior do que o especificado na ISO 11898-2. A saída do TJA1050 corresponde ao pino RxD em que o estado *Low* vale 0 V (GND, nível lógico "0") e *High* = 5V, nível lógico "1".

Na Figura 3.7 pode-se ver uma representação hipotética do sinal de um barramento CAN operando a 500k *bits*/s. Os estados recessivos são indicados, no terceiro gráfico de cima para baixo da figura, pela letra 'R' e os dominantes pela letra 'D'. Ao todo, o sinal possui 5× a sequência de estados: 'R D R'. A primeira curva, de cima para baixo (vermelha), representa o sinal na linha CAN HIGH. A segunda (azul), o sinal da linha CAN LOW. A última (preta), a diferença CAN LOW - CAN HIGH. Durante os testes de curto a operação diferencial, realizada com o osciloscópio, foi CAN LOW - CAN HIGH e, não a tradicional (CAN HIGH - CAN LOW). Para facilitar a correlação com as fotos tiradas da tela do osciloscópio (figuras 3.8 e 3.9) a Figura 3.7 implementa a diferença usada nos testes.

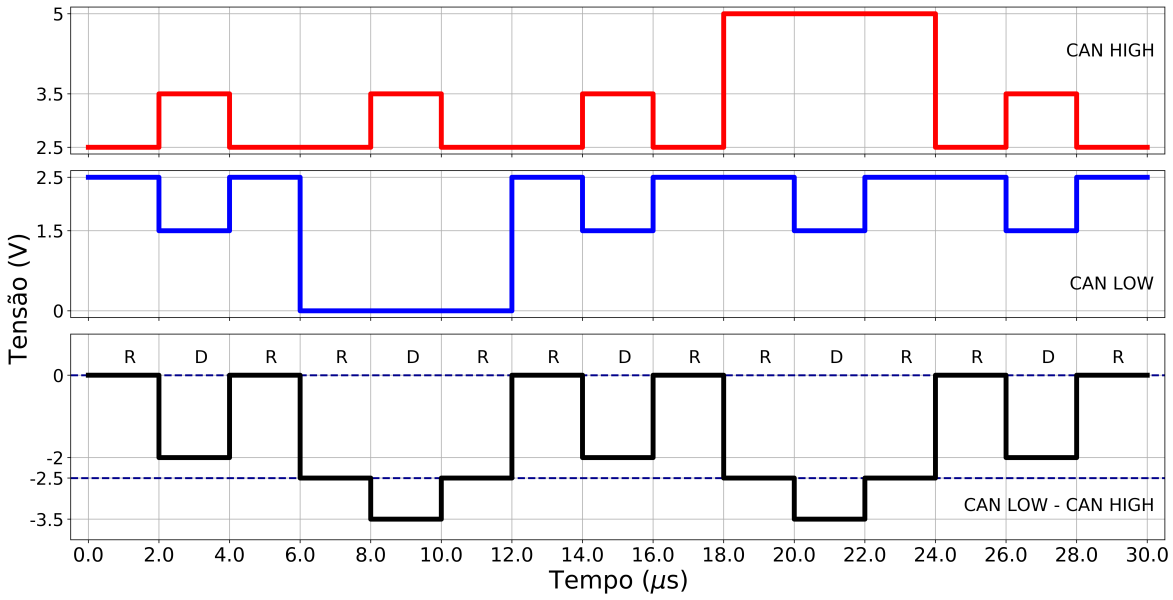


Figura 3.7: Sinal de um barramento CAN hipotético operando à 500k *bits*/s em condições normais e com falhas de curto. Dos 6 aos 12 µs com curto entre o CAN LOW e o GND, dos 18 aos 24 µs com curto entre o CAN HIGH e o VCC de 5 V. Nos demais intervalos não há curto. Os estados recessivos são indicados por 'R' e os dominantes por 'D', ao longo dos 30 µs tem-se 5× a sequência de estados 'R D R'. Fonte: Autores.

O sinal apresentado na Figura 3.7 pode ser dividido em 5 segmentos de 6 µs, todos com

a sequência de estados 'R D R'. Cada estado dura 2 µs. Nos segmentos impares, (1) dos 0 aos 6, (3) dos 12 aos 18 e (5) dos 24 aos 30 µs, o barramento opera sem curtos. Nestes segmentos a linha CAN HIGH apresenta um nível de 2,5 V para os estados recessivos e 3,5 V para dominantes, a linha CAN LOW apresenta 2,5 V para estados recessivos e 1,5 V para dominantes. A diferença, CAN LOW - CAN HIGH, apresenta estados recessivos de 0 V (2,5 V - 2,5 V) e dominantes de −2,0 V (3,5 V - 1,5 V).

No segundo segmento da Figura 3.7, dos 6 aos 12 µs, a linha CAN LOW está em curto-circuito com a linha GND (0 V), caso explorado no *teste a*. O comportamento da linha CAN HIGH segue o mesmo dos segmentos ímpares, porém a linha CAN LOW possui 0 V durante todo o intervalo (6 µs). A diferença entre as linhas, sofre um deslocamento apresentando −2,5 V nos estados recessivos, dos 6 aos 8 µs e dos 10 aos 12 µs, onde normalmente apresenta 0 V. O estado dominante, dos 8 aos 10 µs, apresenta −3,5 V. Para compensar esse deslocamento o transceptor (TJA1050) reescala a referência para −2,5 V. Análise semelhante pode ser feito para o caso do *teste a*, Figura 3.8.

No quarto segmento da Figura 3.7, dos 18 aos 24 µs a linha CAN HIGH está curto-circuito com a linha VCC (5 V), caso do *teste b* e, apresenta um nível de 5 V durante todo o segmento. O sinal da linha CAN LOW não sofre alteração, seguindo o padrão dos segmentos ímpares, enquanto que a diferença sofre a mesma deformação apresentada no segundo segmento. Dessa forma, o transceptor deve fazer o mesmo ajuste no ponto de referência, análise semelhante pode ser feito no caso do *teste b*, Figura 3.9.

A Figura 3.8 apresenta uma foto tirada da tela do osciloscópio TDS 520D da Tektronix durante o *teste a*, quando o barramento operava à 125k *bits*/s. O sinal da linha CAN LOW é a pela primeira curva de baixo para cima, canal 1 do osciloscópio (*Ch1*) e, apresentou um nível constante de ≈ 0 V, somado por ruído do chaveamento da linha CAN HIGH. Para evitar uma sobreposição das curvas, o sinal do canal 1 foi deslocado e, seu nível zero está indicado do lado esquerdo na Figura 3.8 pelo número '1'.

Na captura da Figura 3.8, o *trigger* do osciloscópio foi armado para 3,1 V no canal 2 (*Ch2*), onde estava conectado a linha de dados CAN HIGH, primeira linha de cima para baixo. O sinal da linha CAN HIGH sofreu um deslocamento nos estados, apresentado 0 V para o recessivo e 3,2 V para o dominante, quando de deveria ser 2,5 V e 3,5 V, respectivamente. Essa deformação é resultado de uma carga maior na linha, visto que a CAN HIGH é conectada a CAN LOW pela terminação de 120 Ω e, que a CAN LOW estava em curto

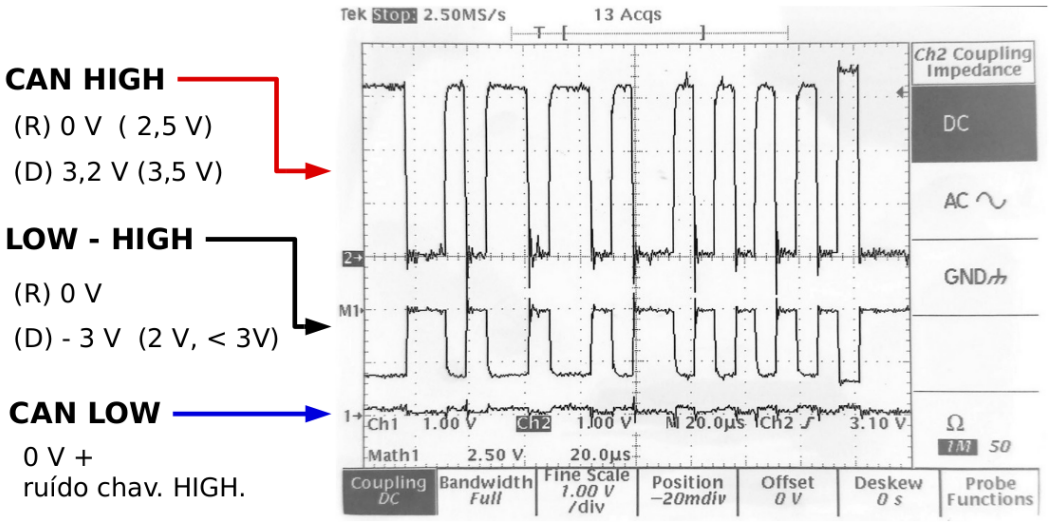


Figura 3.8: Sinal em barramento CAN de 125k *bit*/s com curto da linha CAN LOW para a GND (0 V). Fonte: Autores.

com o GND (0 V).

O deslocamento em estados recessivos foi maior do que em estados dominantes. Pode-se assumir que os estados recessivos sejam mais suscetíveis a alterações, visto que podem ser sobre-escritos por estados dominantes, por isso, tenham sofrido um deslocamento maior. Os estados dominantes foram deslocados em apenas −0,3 V. Esse efeito não foi previsto na situação hipotética apresentada na Figura 3.7.

A diferença (CAN LOW - CAN HIGH) é representada por *M1* (*Math 1*), segunda curva de baixo para cima na Figura 3.8 e, seu zero está deslocado em uma divisão (2,5 V) para baixo. Foi observado, na diferença, um nível de 0 V para os estados recessivos. E para estado dominante um pouco menor do que −3 V. A deformação do sinal diferencial foi ≈ 1 V (em módulo) aumentando a amplitude do estado dominante.

A diferença entre o caso hipotético (Figura 3.7) e o observado no *teste a*, bem provavelmente se deu por não considerar, no caso hipotético, que níveis de estados recessivos podem ser alterados com mais facilidade do que níveis de estados dominantes. Apesar da deformação no sinal, a rede operou normalmente durante o teste. Ou seja, a FURG CAN demonstrou ser imune a curtos entre a linha de dados CAN LOW e a de alimentação GND (0 V). A largura dos *bits* não foi alterada, cada divisão vertical na Figura 3.8 vale 20 µs, o menor intervalo entre duas transições de estado corresponde a 8 µs, duas subdivisões.

A Figura 3.9 apresenta uma foto da tela do osciloscópio TDS 520D, durante o *teste b*,

curto-circuito entre as linhas CAN HIGH e VCC (5 V). O barramento utilizado na captura da Figura 3.9 é o mesmo da Figura 3.8 e, operava com a mesma taxa, 125k *bits*/s. O sinal da linha CAN HIGH, não pode ser visto na Figura 3.9, ele está acima do limite da tela do osciloscópio, que é de 4 V em relação ao zero do canal 2 (meio da tela). De forma análoga ao caso (a), a linha CAN LOW teve sua carga aumentada e, seu sinal (primeiro de cima para baixo) sofreu deformação na amplitude. A linha CAN LOW apresentou, 5 V para estados recessivos, ao invés de 2,5 V e, 2 V para estados dominantes, ao invés de 1,5 V. Assim como no *teste a*, os estados recessivos sofreram um deslocamento maior do que os dominantes.

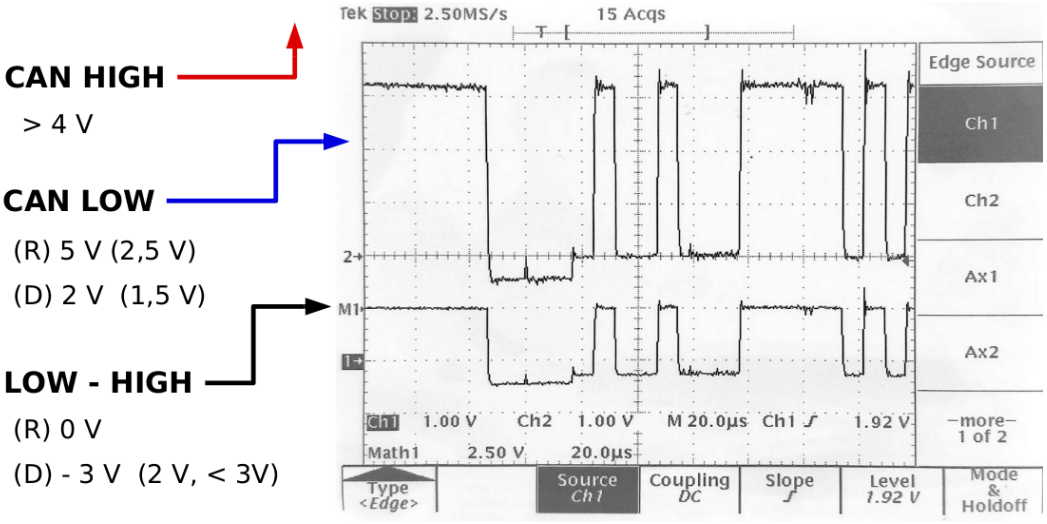


Figura 3.9: Sinal em barramento CAN de 125k *bit*/s com curto da linha CAN HIGH para a VCC (5 V). Fonte: Autores.

A diferença (*M1*), continuou sendo CAN LOW - CAN HIGH e, assim como no *teste a* os níveis observados para os estados recessivos e dominantes foram, respectivamente, 0 V e um pouco menor que −3 V, essa simetria também pode ser vista na Figura 3.7. Assim como no *teste a* a deformação da amplitude não comprometeu o funcionamento da rede e, a falha não alterou a duração dos *bits*. Nesse sentido pode-se concluir que a FURG CAN também é imune a curtos entre as linhas CAN HIGH e VCC (5 V). Os mesmos testes (*a* e *b*) foram realizados para as outras duas taxas de opreação (250k *bits*/s e 500k *bits*/s), sendo observado o mesmo comportamento apresentado nas figuras 3.8 e 3.9. Também foi verificado a imunidade a curtos da linha CAN LOW para a linha VCC e da linha CAN HIGH para a linha GND e, ambos casos a FURG CAN não funcionou.

3.2.2 *Necessidade das terminações de* 120 *ohm*

Idealmente o barramento CAN precisa ter duas terminações de 120 Ω para funcionar, uma em cada extremidade, conforme seção 1.3.1.1, isso evita reflexões do sinal nas extremidades. Quanto maior a frequência do barramento mais crítico deve ser a ausência da terminação, pois isso pode alterar a duração dos *bits* e provocar falhas na rede. Nesse sentido, dos testes relacionados a ausência da terminação escolheu-se apresentar o feito na maior taxa da FURG CAN, 500k *bits* / s.

Esquecer, ou perder uma das terminações pode não comprometer o funcionamento do barramento. Em testes realizados no barramento descrito, 14,35 m de comprimento com três nodos, a ausência de uma terminação não comprometeu o funcionamento da rede. A Figura 3.10 apresenta uma captura do sinal do barramento com apenas uma terminação (*teste c*), o sinal *1* é da linha de dados CAN HIGH, segundo de cima para baixo e, o sinal *2* primeiro de cima para baixo é da linha de dados CAN LOW, *M1* é a diferença do CAN LOW menos o CAN HIGH. Pode-se verificar no sinal *M1* que o formato do *bit* não foi afetado pela ausência de um das terminações, o *Bit timing* contínua valendo 2 µs (5 µs por divisão) e a amplitude diferencial ficou em 2,5 V (uma divisão), lembrando que o limite inferior é de 1,2 V e o superior é de 3 V (ISO, 2003b), para o estado dominante.

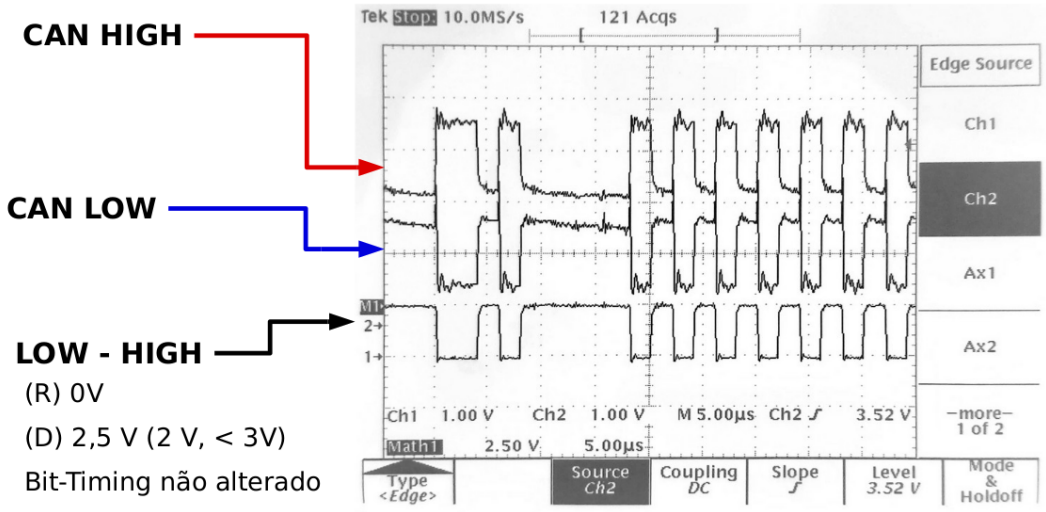


Figura 3.10: Sinal do barramento CAN 500 K *bit*/s com apenas uma terminação. Fonte: Autores.

Ainda que o barramento tenha continuado operacional durante este teste, não é recomendável a elaboração de um barramento com apenas uma terminação, pois isso pode

comprometer a estabilidade da comunicação.

Ao retirar as duas terminações o barramento ficou inoperante, a Figura 3.11 apresenta essa situação, repare que a escala temporal da captura está 10 vezes maior do que apresentado anteriormente (Figura 3.10). A primeira curva de cima para baixo na Figura 3.11 representa o sinal da linha CAN HIGH (*Ch2*) que está deslocado em 2 V, 1 V por divisão. A curva central representa o sinal da linha CAN LOW (*Ch1*) e está deslocado em 1 V (uma divisão). A última curva de cima para baixo representa a diferença CAN LOW - CAN HIGH (*M1*) e também está deslocada em 1 V.

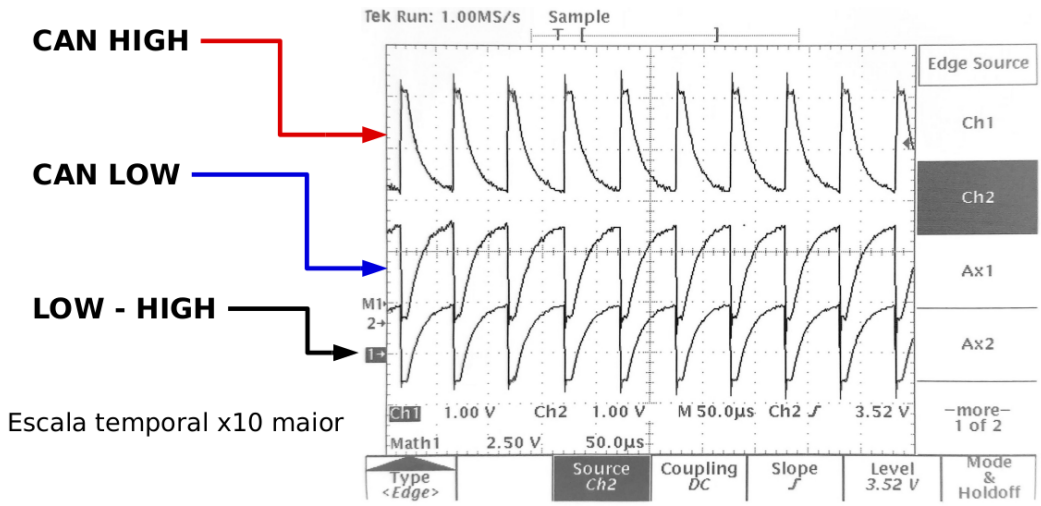


Figura 3.11: Sinal do barramento CAN 500 K *bit*/s sem terminação. Fonte: Autores.

O barramento pode ser analisado como um circuito RC, no qual o par trançado possui uma capacitância C e a terminação um resistência R. Ao retirar as terminações a resistência R aumenta, de 120 Ω para a impedância equivalente a associação das entrada dos nodos. Isso prolonga as transições dos estados dominantes para os recessivos, inviabilizando as codificações dos *bits*. As transições dos estados recessivos para dominantes são menos afetadas, o que resulta em um formato semelhante ao de uma onda dente de serra (*sawtooth*), alterada por uma descarga de capacitor no caso da linha CAN HIGH e uma carga de capacitor no caso da linha CAN LOW.

3.3 Limites Operacionais

3.3.1 Comprimento e frequência

A taxa de *bits* máxima suportada pela FURG CAN é de 500 k *bit*/s, isso está limitado pelo cristal do módulo (8 MHz), que fornece a base de tempo para os controladores CAN MCP2515. A frequência do cristal é usada pelo MCP2515 no calculo do *time quanta*, que por sua vez é usado para o calculo do *Bit timing*, ver seção 1.3.2.1 e, o *Bit timing* limita o comprimento do barramento CAN. As configurações de *Bit timing* utilizadas pela FURG CAN, limitam o comprimento do barramento em 275 m para 125 k *bit*/s, em 125 m para 250 k *bit*/s e em 50 m para 500 k *bit*/s. Estes limites são teóricos, uma vez que não havia disponíveis cabos com os comprimentos citados. Para calcular os limites, foram utilizadas as regras descritas na subseção 3.2.

3.3.2 Tempo de escrita de *frames*

O tempo de escrita dos *frames* fornece o limite de envios por segundo e contribui para o tempo médio (período) de execução de um ciclo do programa do nodo, o que pode ajudar a decidir se a FURG CAN é solução para uma aplicação ou não. A depender do caso, pode-se diminuir o número de envios por ciclos e minimizar a contribuição do tempo de envio no período de execução. O que pode ser feito acumulando um número de informações na memória e, enviando em único momento e, a depender do tamanho da informação pode-se até mesmo enviar mais do que uma por *frame*.

O contador de frequência HP 5316B foi usado para medir o tempo que o Arduino Nano leva para escrever um *frame* de dados estendido com 8 *bytes*. o maior *frame* possível, no *buffer* de saída do controlador CAN MCP2515. Nesse sentido o Arduino Nano foi programado para gerar um pulso antes de chamar a rotina de escrita e outro quando ela acabar, o contador da HP obtém o intervalo de tempo entre estes dois pulsos. A confiança do HP 5316B é de ±1ns.

O CAN Sensor usa um pré *buffer* para alocar os frames antes de seu envio, isso acontece para evitar que o nodo fica parado esperando a possibilidade de enviar. Primeiro é executado a função *putinBuff()*, que leva 24 µs para formatar a informação no *frame* e aloca-lo no pré *buffer* interno do Arduino Nano. Se o um dos dois *buffers* de saída do MCP2515 usados (TXB0 ou TXB1) estiver livre, o Aduino executa a função *sendCAN()*.

Ela gerencia os *buffers* de saída do MCP2515 e o pré *buffer* interno do Arduino Nano. E leva 640 µs para escreve um *frame* no *buffer* de saída TXB1, que o caminha de escrita mais rápido e mais comum. O tempo total (mínimo) observado de escrita foi de 664 µs, a Figura 3.12 apresenta um diagrama linear do tempo de escrita de um *frame*, explicitado as contribuições de cada função.

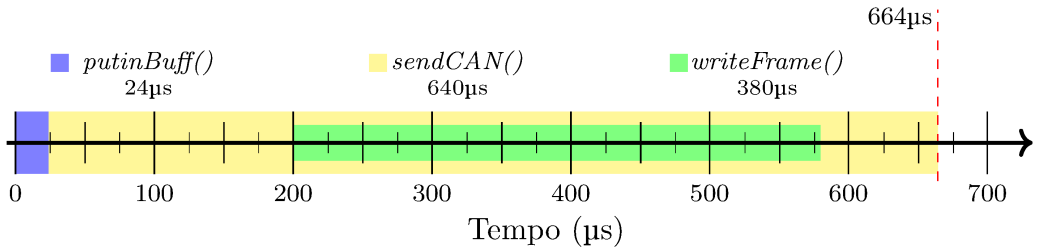


Figura 3.12: Tempo mínimo de escrita de um *frame* de dados estendido com 8 *bytes* de dado. Ao todo são executadas 3 rotinas, *readFrame()* Fonte: Autores.

O primeiro processo indicado na Figura 3.12 é a função *putinBuff()*, que leva 24 µs. A segunda função é a *sendCAN()*, que leva 640 µs, internamente ela chama a função *writeFrame()* da biblioteca *MCP2515.h*, que leva 380 µs para ser executada. A função *writeFrame()* está apresentada no diagrama da Figura 3.12 dos 200 aos 580 µs, porém o momento no qual ela é executado não foi verificado, ela é representada nessa posição para facilitar a leitura de seu comprimento. Após *writeFrame()* a função *sendCAN()* atualiza uma variável de controle e o pré *buffer* interno do Arduino Nano. Se o usuário quiser mais agilidade pode abrir mão das funções *putinBuff()* e *sendCAN()*, usando direto a *writeFrame()* porém, deverá tomar cuidado para que *frames* não sejam sobre gravados nos *buffers* de saída do MCP2515 e conseguintemente perdidos.

3.3.3 Tempo de processamento do CAN Mon

A capacidade de leitura do nodo monitor (CAN Mon) pode limitar a taxa de ocupação do barramento e, conseguintemente a taxa máxima de informação útil trafegada. Neste sentido saber a capacidade de processamento do CAN Mon, número de *frames* lidos do barramento e enviados ao computador por segundo, pode ajudar a decidir se FURG CAN é solução para uma aplicação ou não.

Como descrito no seção 2.2, o Arduino Nano do CAN Mon captura o *frame* nos *buffers* de entrada do MCP2515, os aloca em um *buffer* interno e, assim que possível os envia,

pela porta USB-Serial, ao computador. A função que realiza as operações de leitura e armazenamento no *buffer* interno é a *readCAN()*, ela pode operar com ou sem a função *errCount()*. A função *errCount()* atualiza os contadores de erro e *overload* nos *buffer* de entrada do MCP2515 (Apêndice B). Se a função *errCount()* estiver ativa o tempo de execução da *readCAN()* é de 452 µs, sem a *errCount()* o tempo cai para 332 µs. A *readCAN()* executa necessariamente a função *readFrame()* da biblioteca MCP2515-*bib*, que leva 232 µs, os 100 µs restante é gasto na formatação e manipulação do *frame*. A Figura 3.13 apresenta um diagrama linear do tempo de processamento de um *frame* pelo CAN Mon, nele pode-se ver o tempo de execução da função *readCAN()* e também os tempos de suas duas sob rotinas, *readFrame()* e *errCount()*.

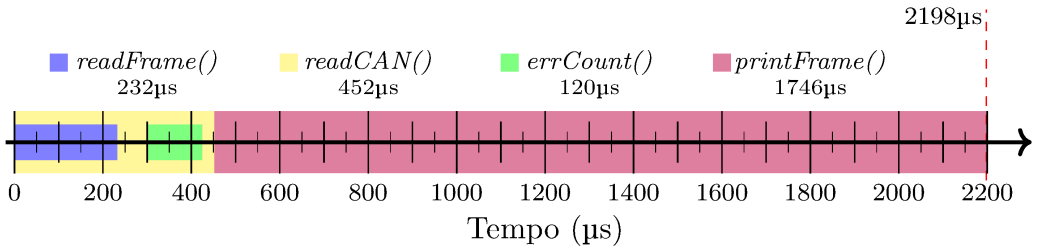


Figura 3.13: Tempo médio de processamento de um *frame* de dados estendido com 8 *bytes* de dado. Ao todo são executadas 4 rotinas, *readFrame()* Fonte: Autores.

O caminho preferencial de envio de *frames* pela porta Serial-USB é a função *printFrame()* que utiliza o padrão ASCII, o que facilita a análise futura dos dados. Cada *frame* enviado pela *printFrame()* leva aproximadamente 1746 µs em uma frequência de 9600 baud. O ciclo de envio se encera com a função *printFrame()* e por isso ela é a última função representa no diagrama da Figura 3.13. Outro caminho é utilizar a rotina *wrtFrm()* que envias os *frames* sem formatação (*byte* a *byte*) pela porta USB, essa rotina leva apenas 200 µs, porém torna a análise futura mais complexa, uma vez que ID padrão, extensão de ID, número de bytes de dados e, os bytes de dados, deverão ser separados e decodificados.

Somando os tempos da *readCAN*, com a *errCount()* ativa e, da *printFrame()*, obtém-se 2198 µs para processar cada *frame*. Neste caso o CAN Mon pode processar até 454,9 *frames* por segundo. Utilizando a rotina *wrtFrm()* e sem a *errCount()*, caminho mais rápido, o CAN Mon pode processar até 1879,7 *frames* por segundo.

3.3.4 Tempo de processamento do CAN Save

Pode-se optar por operar sem um nodo CAN Mon e com um CAN Save, neste caso a taxa de ocupação do barramento será limitada pela capacidade de processamento do nodo CAN Save. A rotina de leitura de nodos CAN Save é exatamente a mesma que de um nodo CAN Mon. A única diferença é a parte de transferira de dados, que no nodo monitor é para o computador e, no CAN Save é para o cartão micro SD. Ela dura em média 608 us por *frame*. Somando a esse tempo o tempo de leitura de 452 µs chega-se um tempo de processamento médio 1060 µs por *frame*. A Figura 3.14 apresenta um diagrama linear do tempo de processamento de um *frame* do CAN Save, nele estão explicitadas as rotinas e sub-rotinas executas durante o processo.

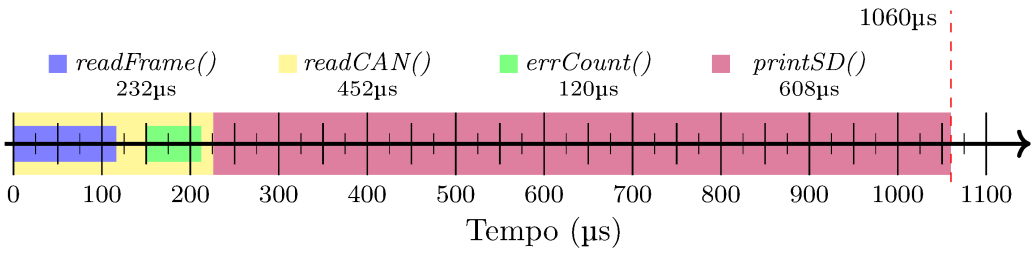


Figura 3.14: Tempo médio de processamento de um *frame* de dados estendido com 8 *bytes* de dado. Ao todo são executadas 4 rotinas, *readFrame()* Fonte: Autores.

O tempo observado para escrever um *frame* no cartão micro SD, pelo método SPI, variou de 380 á 420 µs. Entretanto o tempo de escrita do 35º*frame* é muito maior, algo em torno de 7 ms. Essa demora deve ocorrer quando o *buffer* da comunicação entre o Arduino Nano e o cartão SD, que tem 512 *bytes*, fica cheio. Para calcular o tempo médio, considerou-se um grupo de 35 *frames*, onde os 34 primeiros são gravados em 420 µs e, o último em 7 ms, o que resultou no valor médio de 608 µs. Ressalta-se que os tempos de gravação variam de acordo com o tipo de cartão micro SD, em geral cartões maiores são mais rápidos. O CAN Save leva em média 1060 µs para processar um *frame*, o que resulta 943,4 *frames* por segundo.

3.3.5 Taxa de Ocupação do barramento

Pre-supondo uma rede FURG CAN, composta de vários nodos emissores (CAN Sensor), um nodo monitor (CAN Mon) e, que não haja nenhum outro fator limitante para a taxa

de ocupação do barramento, como por exemplo, erros de comunicação. Nestas condições a taxa de ocupação limite do barramento pode ser encontrada pela razão do número máximo de *frames* processados pelo CAN Mon por segundo, divido pelo número de frames trafegados por segundo em um barramento lotado.

O CAN Mon, utilizando a rotina *printFrame()*, modo mais simples e mais lento, pode processar até 454,9 *frames* estendidos com 8 *bytes* de dados por segundo. Por outro lado, em um barramento CAN operando com 500k*bits*/s podem trafegar até 3816,8 *frames* do mesmo tipo por segundo, resultando em uma taxa de ocupação limite de $11,92\%$. Para esse calculo foi considerando um *Inter Frame Espace* de 3 *bits*, tamanho preferencial, o que resulta em um tempo de 262 µs por *frame* (131 *bits* * 2µs/*bit*).

Repetindo o procedimento acima para um CAN Mon usando a função *wrtFrame()* ao invés da *printFrame()*, modo mais rápido, no mesmo barramento e com o mesmo tipo de *frame* o limite da taxa de ocupação sobe para $49,25\%$, porém utilizar essa rotina torna o procedimento de análise muito mais complexo, uma vez que os dados armazenados no computador terão que ser decodificados antes da análise. Lembrando que um dos objetivos deste trabalho e obter uma rede fácil de usar, essa possibilidade não foi explorada e, os dados apresentados a seguir se restringem a um nodo CAN Mon com a rotina *printFrame()*. A Tabela 3.1 apresenta um resumo das taxas de ocupação limites, do número de *frames* por segundo e do número de *bytes* de dados por segundo para as três frequências compatíveis da FURG CAN (500k*bits*/s, 250k*bits*/s e 125k*bits*/s) e para 6 formatos de *frames* diferentes, padrões (pad.) e estendidos (ext.) com 8, 4 e 2 *bytes* de dados. Esses valores são para um CAN Mon elaborado em um Arduino Nano usando a rotina *printFrame()* para transferir *frames* para o computador.

Os valores apresentados na Tabela 3.1 são resultados de uma extrapolação teórica em relação ao tempo de processamento de um *frame* estendido com 8 *bytes* de dados, onde foi considerado os diferentes tamanhos dos *frames*, além disso, devem variar um pouco de Arduino para Arduino. Ainda sim servem para uma avaliação previa do potencial da FURG CAN como solução para aplicações.

As transferências de *frames* do CAN Save para o cartão micro SD são mais rápidas do que a do CAN Mon para o computador. Por tanto, espera-se que o CAN Save possa operar em barramentos mais ocupados do que o nodo monitor. Considerando que a rede descrita acima fosse alterado, tendo seu nodo CAN Mon substituído por um CAN Save, com o

Tabela 3.1 - Taxas de ocupação limites da FURG CAN em relação ao CAN Mon para *frames* com ID padrão e estendidos e, com 8, 4 e 2 *bytes* de dados. A coluna '500 (%)' lista as taxas limites para a 500 k *bit*/s e, de forma análoga seguem as colunas '250 (%)' e '125 (%)'. A coluna 'n.*frames*/s' lista o número de *frames* processado por no limite e, a coluna 'n.*bytes*/s' lista o número de *bytes* de dados por segundo. Fonte: Autores.

Formato do *frame*	500 (%)	250 (%)	125 (%)	n. *frames*/s	n. *bytes*/s
ID ext. e DLC = 8	11, 92	23, 84	47, 67	454, 9	3639, 2
ID ext. e DLC = 4	12, 41	24, 81	49, 63	626, 6	2506, 3
ID ext. e DLC = 2	12, 81	25, 62	51, 23	771, 6	1543, 2
ID pad. e DLC = 8	11, 59	23, 17	46, 34	521, 9	4175, 4
ID pad. e DLC = 4	12, 04	24, 09	48, 17	762, 2	3048, 8
ID pad. e DLC = 2	12, 40	24, 8	49, 61	984, 3	1968, 5

apresentado em 3.3.4. Repetindo o procedimento descrito para o CAN Mon, obteve-se os limites de ocupação do barramento em termos do nodo CAN Save. A tabela 3.2 apresenta as taxas de ocupação limites em como relação a capacidade de processamento do CAN Save citado, os valores foram obtidos de forma análoga a feita para o CAN Mon e, além de variarem um pouco de um Arduino Nano para outro, podem variar, também, em função do tipo de cartão micro SD escolhido.

Tabela 3.2 - Taxas de ocupação limites da FURG CAN em relação ao CAN Save para *frames* com ID padrão e estendidos e, com 8, 4 e 2 *bytes* de dados. A coluna '500 (%)' lista as taxas limites para a 500 k *bit*/s e, de forma análoga seguem as colunas '250 (%)' e '125 (%)'. A coluna 'n.*frames*/s' lista o número de *frames* processado por no limite e, a coluna 'n.*bytes*/s' lista o número de *bytes* de dados por segundo. (*) Apesar do calculo indicar um limite 100%, ou mais, deve fornecer periodicamente um tempo para que o *buffer* da comunicação com o cartão micro SD seja esvaziado. Fonte: Autores.

Formato do *frame*	500 (%)	250 (%)	125 (%)	n. *frames*/s	n. *bytes*/s
ID ext. e DLC = 8	24, 72	49, 44	98, 88	943, 4	7547, 2
ID ext. e DLC = 4	25, 47	50, 94	100*	1286, 45	5145, 8
ID ext. e DLC = 2	26, 10	52, 20	100*	1572, 3	3144, 6
ID pad. e DLC = 8	28, 05	56, 1	100*	1286, 45	10291, 6
ID pad. e DLC = 4	31, 13	62, 26	100*	2021, 6	8086, 4
ID pad. e DLC = 2	34, 53	69, 06	100*	2830, 2	5660, 4

3.4 Avaliação geral

De uma forma geral, a FURG CAN obedece o protocolo CAN 2.0 *High Speed*, sendo imune a curtos da linha CAN HIGH com o VCC e da linha CAN LOW com o GND.

Porém os comprimentos máximos do barramento são menores do que os encontrados na literatura (ISO, 2003b), isso se deve ao formato do *Bit timing* utilizado na FURG CAN, o comprimento máximo (teórico) do barramento é de 275 m quando operando a 125 k *bit*/s.

O fator mais limitante da FURG CAN é a capacidade de processamento do nodo monitor (CAN Mon), principalmente no tempo de transferência de frames para o computador. Redes compostas por nodos sensores e um CAN Save, sem um monitor, podem operar como taxas de ocupação maiores, um pouco mais do que 2×, em comparação a redes como o CAN Mon. Em um caso razoável, com informações de 4 *bytes* e apenas ID padrões, o nodo CAN Mon pode processar uma mensagem a cada 1,3 ms, enquanto um CAN Save leva aproximadamente 0,92 ms.

Pode-se aumentar a taxa de ocupação limite do barramento adicionando um outro CAN Mon e programando os filtros e máscaras de ID dos controladores CAN (MCP2515), de modo que cada monitor recolha uma parte das mensagens, por exemplo, um recolhe os *frames* de ID pares, enquanto o outro recolhe os de ID impares. Para esse mecanismo funcionar os ID dos nodos sensores devem ser escolhidos de acordo. Para realizar as configurações de máscaras e filtros pode-se usar a função de configuração *confFM()* da biblioteca MCP2515-*bib*, a descrição desta função com exemplo pode ser vista no Apêndice B. O mesmo mecanismo pode ser feito com o CAN Save, com uma vantagem, para cada nodo CAN Mon adicionado será necessário uma porta USB-serial independe para armazenamento de dados.

Uma forma mais efetiva para resolver as limitações dos nodos CAN Mon e CAN Save é substituir o Arduino Nano por microcontroladores que possuam modos de transferência mais eficiente. Por exemplo o ESP8266 e ESP32 possuem nativamente uma interface *wi-fi* podendo enviar os *frames* a um computador através de um roteador sem fio ou até mesmo pela internet. O ESP32 ainda possui ainda um *SDMMC Host* o que permite uma comunicação mais rápida com cartões SD. Também possui uma *engine*, interna, do protocolo CAN. Por isso ele pode substitui o controlador CAN MCP2515 e o Arduino nano simultaneamente, diminuindo um intermediário da comunicação e provavelmente economizando tempo de processamento.

Outra possibilidade é construir um nodo monitor com um Raspberry Pi, ou equivalente, que possui protocolo SPI e por isso pode se comunicar com MCP2515, a vantagem deste dispositivo é sua capacidade de processamento e armazenamento. Ele pode fazer as funções

do Arduino Nano do CAN Mon, e do computador ao mesmo tempo, o que na pratica elimina um intermediário e deve economizar tempo. Ressalta-se que qualquer modificação de *hardware* nos nodos CAN Mon e CAN Save deverá ser acompanhada de respectiva mudação no *software*. E isso, pode trazer desvantagens, sobretudo do ponte de vista da acessibilidade.

Capítulo 4

Exemplo de Aplicações

Nesta aplicação foram testados 19 sensores de umidade relativa comerciais, de seis modelos diferentes, três DHT11 (Aosong (Guangzhou) Electronics, b) e sete do DHT22 (Aosong (Guangzhou) Electronics, a) ambos da fabricante Aosong, duas unidades do HDC1080 da Texas Instruments (Texas Instruments, 2016b), quatro unidades do SHT31 , três do SHT35 (Sensirion, 2019) e um SHT75 (Sensirion, 2011) todos os três últimos são da fabricante Sensirion. Os parâmetros de qualidade dos transdutores podem ser consultados na Tabela 4.1, um sumário com suas especificações técnicas. As colunas "Faixa", indicam as faixas de operações. As colunas "Inc.", ao lado, indicam os valores típicos de incerteza. As "Res." indica a resolução e, as "Rep." a repetibilidade dos dispositivos. Os parâmetros de qualidade podem não ser constantes ao longo de toda faixa de operação, para maiores detalhes consultar os *datasheet* dos fabricantes.

Tabela 4.1 - Resumo das especificações dos 19 transdutores de umidade relativa e temperatura analisados durante o teste da FURG CAN. Fonte: Autores

Sensor	Umidade Relativa (RH)/%				Temperatura/°C			
	Faixa	Inc.	Res.	Rep.	Faixa	Inc.	Res.	Rep.
DHT11	20 à 90	±4	1	±1	0 à 50	±2	1	1
DHT22	0 à 100	±2	0,1	±1	−40 à 80	±0,5	0,1	±0,2
HDC1080	20 à 100	±4	14*bits*	±0,1	5 à 60	±0,2	14*bits*	±0,1
SHT31	0 à 100	±2	0,01	±0,15	−40 à 125	±0,2	0,01	±0,08
SHT35	0 à 100	±1,5	0,01	±0,15	−40 à 125	±0,1	0,0	±0,08
SHT75	0 à 100	±1,8	0,05	±0,1	−40 à 125	±0,3	0,01	±0,1

Além dos sensores comerciais, a umidade relativa foi mensurado também através de um psicrômetro formado por dois Pt100 AA da OMEGA (OMEGA Engineering inc., c) lidos com o auxilio do ADC ADS1248 (Texas Instruments, 2016a). O caminho mais fácil

para controlar tantos sensores é utilizar mais do que um Arduino. Além disso os módulos HDC1080 possuem o mesmo endereço, fixado pela placa de teste utilizada, o que dificulta o compartilhamento do barramento I2C. Entretanto ao utilizar mais do que um Arduino um novo problema surge. Como sincronizar as coletas de diversos pontos?

A solução escolhida foi a implementação da FURG CAN, que facilitou a junção e organização dos dados, possibilitando adquirir todas as informações de diferentes fontes por apenas uma porta USB. Dessa forma os nodos compartilharam a mesma base de tempo, fornecida por um Raspberry Pi 3B que foi utilizado no acúmulo de dados. O sucesso dessa implementação depende da taxa de ocupação do barramento, pois uma vez que a base de tempo está no receptor, o congestionamento de dados pode afetar a qualidade do sincronismo. Opcionalmente, a base de tempo poderia estar associado aos nodos sensores. Porém, isso aumentaria a dificuldade da implementação e, de fato, a taxa de ocupação do barramento era baixa o suficiente para a solução utilizada. Neste caso, especifico a taxa de ocupação do barramento foi de $0,058\%$.

O cálculo da taxa de ocupação é bem simples, primeiro deve-se calcular o tamanho dos *frames*, em número de *bits*. Nesta aplicação foram utilizados apenas *frames* padrão com 4 *bytes* de dados, o que resulta em *frames* de 76 *bits*. Os demais 44 *bits* são da estrutura do *frame*. Em seguida, deve-se calcular o número máximo possível de *frames* por segundo, o que resultaria em uma taxa de ocupação de 100%, esse valor é dado pela razão da frequência de *bits* utilizada pelo número de *bits* do *frames*. A frequência utilizada foi 500 k $bits/s$ o que resultaria em 6578,9 frames por segundo. Em seguida deve-se calcular o número total de *frames* a serem enviados pelo barramento, no caso específico eram enviados 32 *frames* a cada 10 segundo. A taxa de ocupação é dada pela razão no número máximo de *frames*, pelo número de *frames* trafegados.

Ao todo foram realizadas treze coletas, que foram interrompidas para análise e checagem de erros. A montagem mudou uma vez ao longo das coletas, na maioria do tempo os sensores foram distribuídos em quatro nodos, todos controlados por um Arduino Nano (cada), na primeira montagem um dos nodos sensor era controlado por um Arduino Mega. A foto da Figura 4.1 mostra uma vista da primeira montagem, todos os dispositivos presentes na foto estão identificados, sendo: (1) 7x DHT22 (sensores brancos) e 3x DHT11 (sensores azuis); (2) 2x HDC1080 ; (3) 1x SHT31 ; (4) 1x SHT35 ; (5) 1x SHT75; (6) 2x Pt100AA; (7) placa de PCB do AD1248; (8) 2x MCP2515; (A), (B) e (C) são os

controladores (placas Arduino) dos nodos CAN Sensor A, B e C respectivamente.

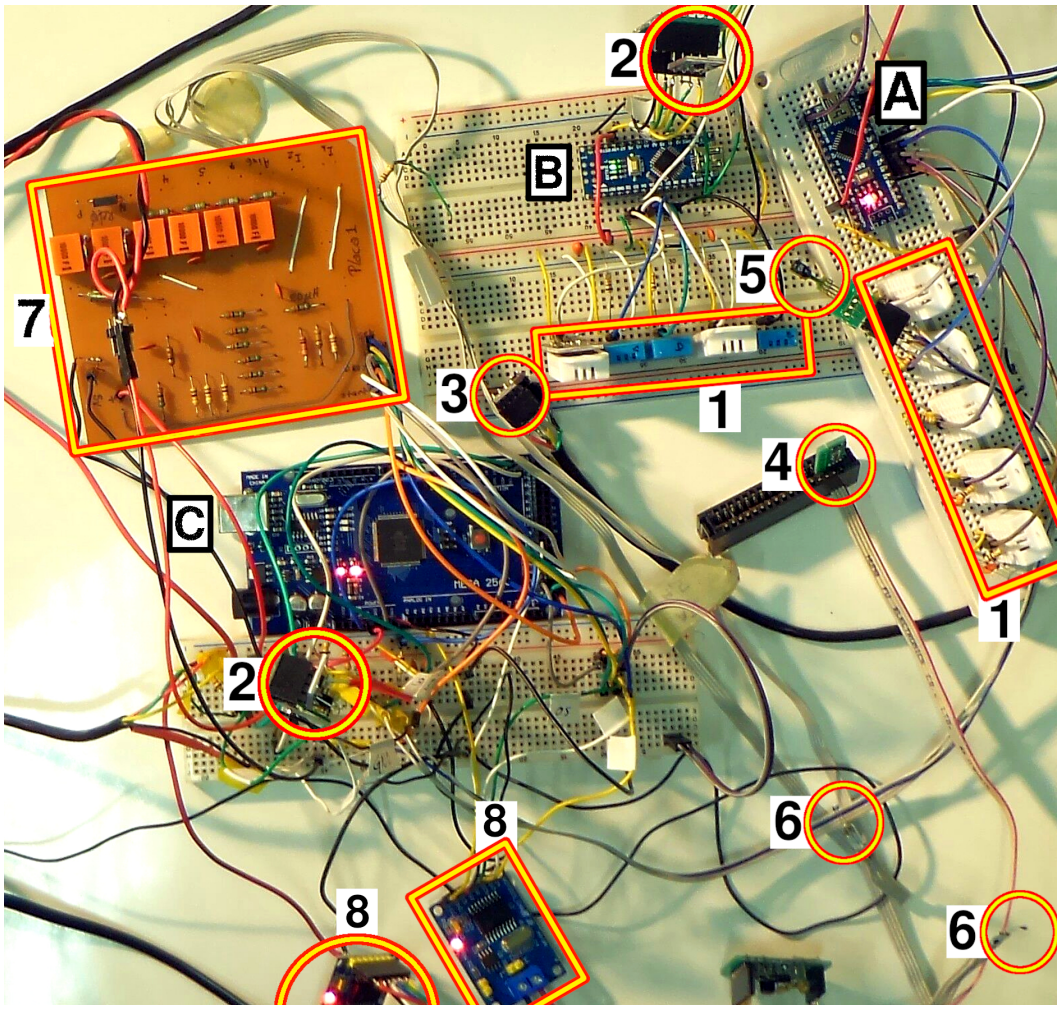


Figura 4.1: Foto da montagem A, em destaque os sensores, (1) DHT11 em azul e o DHT22 em branco, (2) DHC1080, (3) SHT31 , (4) SHT35 , (5) SHT75, (6) Pt100AA, (7) Placa com o AD1248, (8) MCP2515. Além dos sensores estão destacados controladores dos nodos A, Arduino Nano a direita e acima, B, Arduino Nano ao centro e acima e C, Arduino Mega ao centro e esquerda. Fonte: Autores.

A coleta mais longa, sem interrupções, durou de 71 h 41 min 24 s, a menor foi de 9 h 42 min, ao todo foram coletadas 437 h 19 min 48 s total, somando todos os períodos de coletas. A taxa de ocupação média máxima (segunda montagem) do barramento foi de $3,2$ *frames*$/s$, enquanto a taxa de perda observada durante essa aplicação variou de $0,05\%$ à $0,18\%$, dependendo do nodo e da coleta, mas não apresentou erros sistemáticos.

4.1 Montagem

Durante a aplicação fora utilizado um barramento composto por dois pares trançados, um utilizado para comunicação, CAN *HIGH* e CAN *LOW*, e outro usado para alimentação de 9 V, os conectores usados entre o barramento CAN e os módulos CAN (MCP2515 + TJA1050) foram do tipo RJ11, os mesmos usados em linhas de telefonia fixa. Vale ressaltar que os nodos extremos (monitor e CAN Sensor D) possuem terminação padrão de 120 Ω. Ao longo das coletas foram utilizados duas montagens, a primeira com 17 sensores e a segunda com 19, em ambas as montagens foram utilizados 4 nodos do tipo CAN Sensor, 1 nodo do tipo CAN Mon e um Raspberry Pi 3B que acúmulou os dados coletados e forneceu base de tempo para a coleta.

Na primeira montagem foram monitorados 17 sensores, 15 termo-higrômetros e 2 Pt100 AA. Os Pt100 foram lidos com o auxílio do ADC ADS1248, usando a biblioteca ADS124X (Ferrando et al., 2017) desenvolvida pelo grupo. Os sensores térmicos constituem um psicrômetro, e suas temperaturas foram processadas posteriormente de modo a obter a umidade relativa. A Figura 4.2 apresenta um diagrama simplificado da montagem A, nela pode-se ver quatro nodos CAN Sensor, o A, o B e o D controlados por um Arduino Nano cada, e o C controlado por um Arduino Mega. Além dos nodos sensores há um nodo do tipo CAN Mon (Monitor) controlado por um Arduino Nano, ele recolheu os dados e enviou a um Raspberry Pi 3B, representado no diagrama da Figura 4.2 pelo bloco PC.

Na segunda montagem o Arduino Mega do nodo C foi substituído por um Arduino Nano, por motivos de simplificação o psicrômetro (2x Pt100) não foi utilizado. E um dos DHT22 foi descartado pois apresentava leituras muito fora do esperado. Além dos sensores remanescentes da montagem A, foi adicionado na montagem B dois SHT35 e três SHT31 , resultando em 19 termo-higrômetros. O restante da configuração permaneceu igual ao da montagem A, a Figura 4.3 apresenta um diagrama simplificado da montagem B.

As conexões dos sensores (individualmente) aos controladores estão disponíveis no Apêndice C. Esse apêndice é um conjunto de exemplos para referência, por isso só é apresentado as conexões de um módulo sensor, dos já mencionados, do módulo CAN (com barramento) e do Arduino Nano.

A Figura 4.4 apresenta diagrama de conexões entre um Arduino Nano, um módulo HDC1080 e um MCP2515 ligado ao barramento CAN. Na extrema direita da Figura 4.4

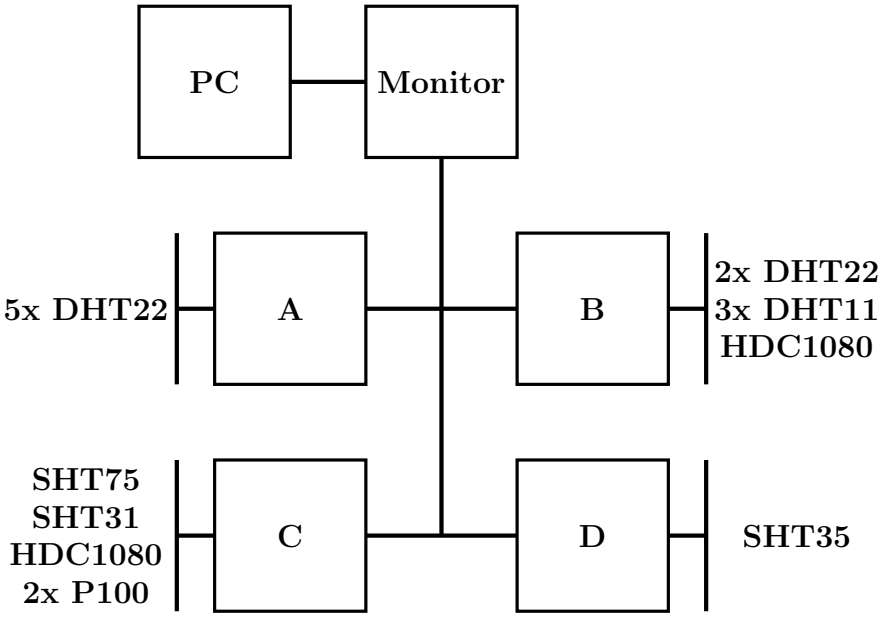


Figura 4.2: Diagrama de montagem A da aplicação, o Nodo C é controlado por um Arduino Mega, os demais nodos são controlados por um Arduino Nano cada, inclusive o Monitor, e o PC é um Raspberry Pi 3B. O barramento foi feito em dois pares de um cabo Cat6, sendo um dos pares para comunicação e outro para alimentação, os conectores usados foram do tipo RJ11. Fonte: Autores.

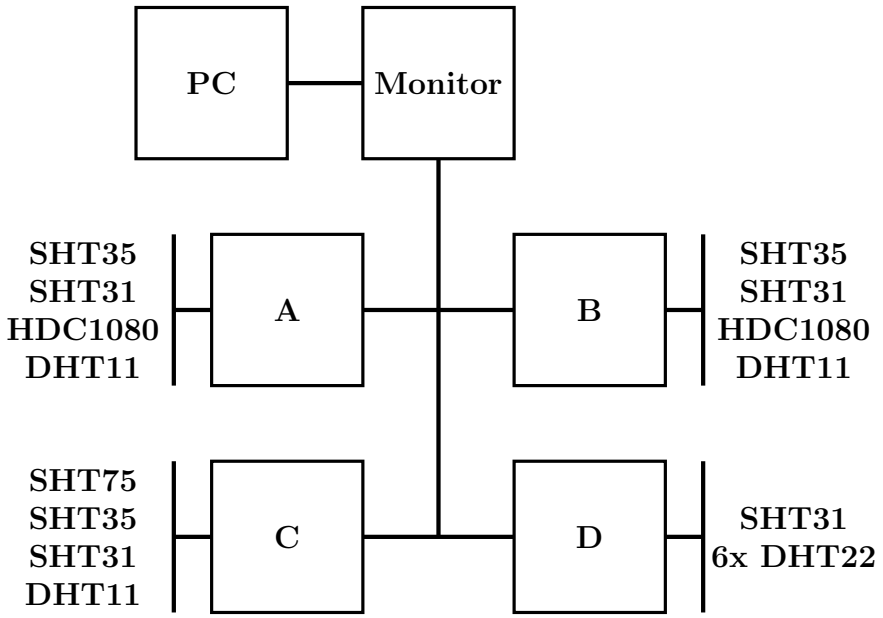


Figura 4.3: Diagrama da montagem B da aplicação, todos os nodos (A, B, C, D e Monitor) são controlados com um Arduino Nano cada, o PC é um Raspberry Pi 3B. O barramento foi feito em dois pares de um cabo Cat6, sendo um dos pares para comunicação e outro para alimentação, os conectores usados foram do tipo RJ11. Fonte: Autores.

é representado as quatro vias do barramento CAN utilizado, da esquerda para direita, as linhas de dados CAN LOWe CAN HIGH, seguidas das linhas de alimentação a terra (GND) e a positiva (9 V). O módulo CAN, no canto direito superior e à esquerda do

barramento no diagrama da Figura 4.4, está conectado as linhas de dados e ao terra do barramento CAN. A comunicação entre o módulo CAN e o Arduino segue o protocolo SPI, sendo usado os pinos padrões do Arduino (**SO: D12**, **SI: D11** e **SCK: D13**), para *Chip Select* (**CS**) é utilizado o pino **D4**. Além disso o pino **INT** do módulo é conectado o **D2** do Arduino.

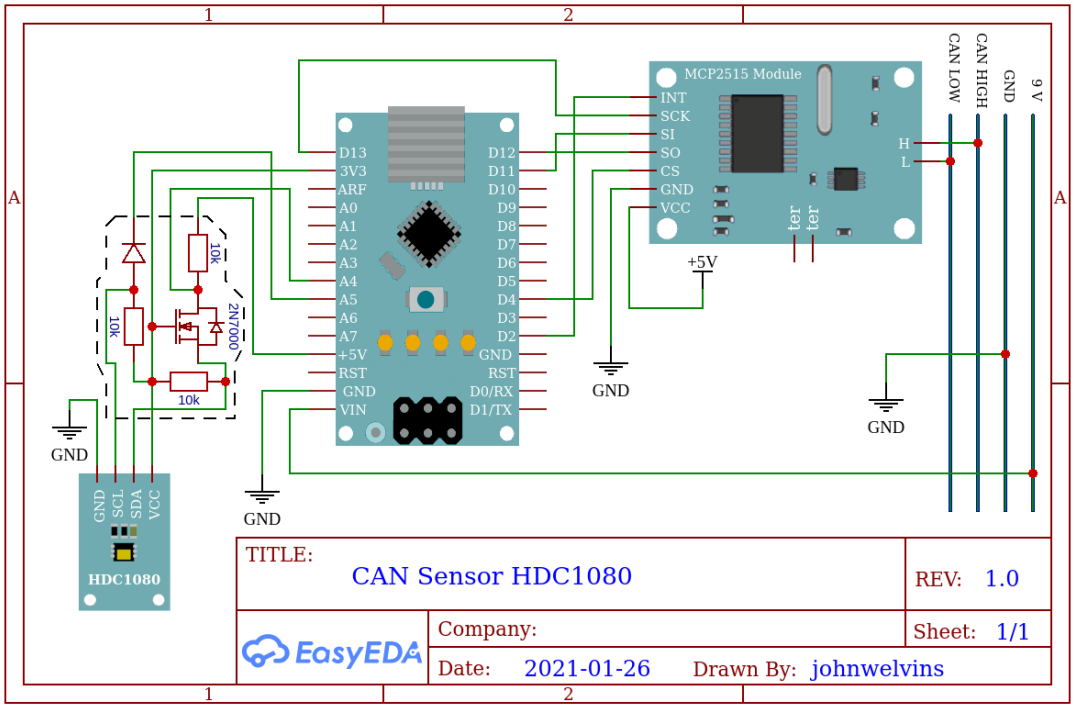


Figura 4.4: Diagrama das conexões de um nodo do tipo CAN Sensor com o HDC1080 , transdutor de umidade relativa e temperatura da Texas Instruments. Fonte: Autores.

Ainda no diagrama da Figura 4.4, tem-se no canto esquerdo inferior o módulo HDC1080 , a comunicação dele com Arduino Nano segue o protocolo I2C e utiliza os pinos A5 para *clock* e o A4 *data*. O circuito acima do HDC1080 , na área demarcada pelo linha tracejada, realiza a conversão de nível da comunicação, dos 5 V do Arduino para os 3,3 V do transdutor. O fabricante do HDC1080 informa no *datasheet* que o dispositivo pode comunicar a 5 V Texas Instruments (2016b), neste sentido a utilização desse conversor é opcional.

O circuito de conversão utiliza um MOSFET (*Metal Oxide Semiconductor Field Effect Transistor*) 2N7000, o pino SDA (*Serial data*) do módulo é conectado ao *source* do MOSFET, do lado do Arduino o SDA (pino A4) é conectado ao dreno do MOSFET. O pino SCL (*Serial Clock*) do módulo do HDC1080 é conectado ao pino A5 do Arduino através

de um diodo. O pino de alimentação (**VCC**) do módulo HDC1080 é conectado ao pino **3V3** do Arduino, o gate do MOSFET também é conectado o **3V3**. Os pino **SDA** e **SCL** do módulo são conectados a alimentação **3V3** do Arduino através de um resistor de 10 kΩ cada. Por fim o dreno do MOSFET é conectado a alimentação **+5V** do Arduino Nano através de um resistor de 10 kΩ.

Os códigos de controle e aquisição de todos os nodos foram construídos com a ajuda das bibliotecas próprias do ADC ADS1248 (Ferrando et al., 2017) e do MCP2515-*bib* (Araújo et al., 2020), e de bibliotecas de terceiros encontradas na rede Web: SHT31 e SHT35 (Adafruit, 2020); SHT75 (SPEASE e PER1234, 2014); HDC1080 [34] (Texas Instruments, 2017a); DHT11 e DHT22 (Adafruit, 2021). O programa do nodo variava em função dos sensores usados, porém e a ideia do programa era a mesma para todos além disso a parte do enviou de dados era a mesma. Os nodos sensores foram programados de modo a enviar um conjunto de dados a cada 10 s, um conjunto de dados individual, de apenas um nodo, é formado por um par de *frames* de dados de cada sensor gerenciado pelo nodo.

Um Raspberry Pi 3B é usado para acúmulo de dados e com base de tempo. Por isso, o nodo conectado a ele foi programado com o exemplo CANMon.ino, utilizando a função *writeFrame_serial()*. Consequentemente as conexões do nodo são as mesmas apresentadas para nodos do tipo CAN Mon 2.7. Foi elaborado um programa em python (*CANMon.py*) para converter e salvar (formato.cvs) os dados recebidos do CAN Mon, executado pelo Raspberry Pi. Este programa está desenvolvido apenas para Linux base Debian, mas pode ser adaptado para Windows.

4.2 Procedimento de análise

O objetivo da aplicação era comparar os valores médios indicados pelos sensores. Porém cada sensor deve possuir um comportamento diferente em relação a variação da umidade, como por exemplo, um pode ter uma inercia térmica maior ou menor que outro. Essa diferença é agravada quando os sensores são de diferentes tipos, e nesta aplicação foram usados três tipos de transdutores: resistivo, capacitivo e o psicrômetro. O caminho mais simples para evitar problemas gerados pelos diferentes comportamentos foi selecionar regiões estáveis dentro da coleta. O procedimento de análise em si, foi dividido em dois momentos, um realizado durante as interrupções da coleta, isto é, a cada coleta. E um

outro ao fim de toda experimentação.

No primeiro momento, pré-análise, os dados eram capturados do arquivo salvo pelo CANMon.py, os erros de transmissão eram contabilizados, os dados já filtrados eram organizados e plotados para verificação qualitativa da coleta. Isto é, verificar se foi observado uma região estável o suficiente para comparação das curvas obtidas na coleta. Se não tiver sido observado nenhum dado estável a coleta seria descartada.

Uma forma simples de obter uma região estável é realizar coletas de alguns dias em um cômodo fechado. Já foi observado anteriormente um patamar de algumas horas com dispersões $(2 \times \sigma)$ menores do que de $\pm 0,03$, em uma temperatura ambiente média de 20,21 °C, indicado pela média de dois Pt100 AA, modelo F2222-100-1/3B da OMEGA (OMEGA Engineering inc., c), lidos pelo ADC ADS1248 (Texas Instruments, 2016a).

No mesmo patamar dois termopares K apresentaram dispersões menores do $\pm 0,3$ em uma temperatura de 20,21 °C e indicada pela média dos dois sensores, estes termopares possuem baixa massa e consequentemente baixa capacidade térmica. Eles são formados com um segmento do fio de Cromel TFCY-003 da OMEGA (OMEGA Engineering inc., b), e um segmento do fio Alumel TFAL-003 da OMEGA (OMEGA Engineering inc., a), ambos possuem um diâmetro de 0,08 mm (0,003 polegadas). Os termopares são lidos com a ajuda do conversor da MAX31856 da Maxim (Maxim Integrated, 2015).

Entre um conjunto de oito unidades do LM35 (Texas Instruments, 2017b) pode-se observar dispersões $(2 \times \sigma)$ menores do que $\pm 0,035$ no patamar já citado a temperatura média do grupo de sensores foi de 19,77 °C. A umidade relativa, possui dependência com a temperatura, logo uma região estável em temperatura é potencialmente estável em umidade relativa.

A Figura 4.5 apresenta um gráfico típico obtido ao fim da pré-análise, porém por simplicidade este contém apenas o psicrômetro e um representante de cada sensor comercial testado: SHT31 , HDC1080 , DHT11 , DHT22 e o SHT75. O SHT75 foi utilizado como referência e a curva representada por ele já foi corrigida anteriormente com soluções saturadas de $NaCl$, KCl e $MgCl_2$ e valores padrões foram retirados da designação E104-20 da ASTM (ASTM international, 2020). Algumas soluções saturadas, normalmente água saturada de algum sal, são capazes de produz uma estabilidade de umidade relativa, em pequenos volumes. Dessa forma o procedimento de aferição do sensor SHT75, foi feita, comparando os valores padrões, com as leituras feitas pelo SHT75. Vale ressaltar que o

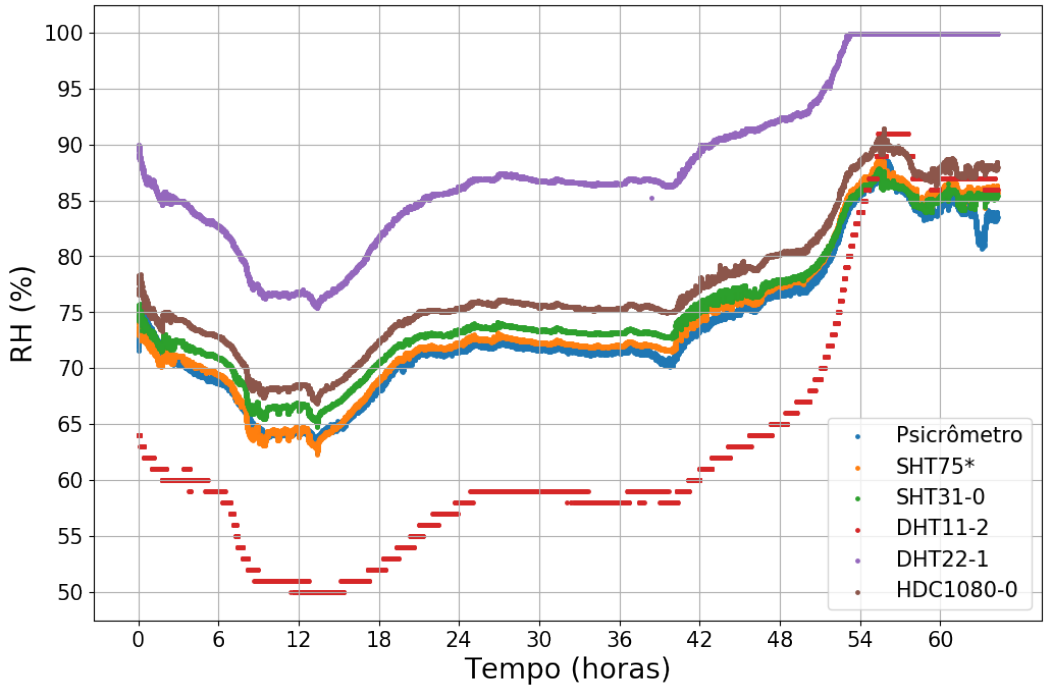


Figura 4.5: Fonte: Autores.

nível de estabilidade da umidade relativa depende da temperatura e, algumas soluções são mais ou menos sensíveis.

Ao todo são seis curvas, que de baixo para cima na posição 30 horas correspondem, ao DHT11-2, ao psicrômetro, ao SHT75 corrigido, ao SHT31-0, ao HDC1080-0 e ao DHT22-1. As curvas do SHT75 (corrigido) e do psicrômetro estão quase sobrepostas, reforçando a confiança do SHT75 após a correção. A curva do SHT31-0 (verde) é relativamente mais próxima à referência, e começa a sobrepô-la para RH $> 80\%$.

Do conjunto de dados apresentado na Figura 4.5, destaca-se a região com a maior estabilidade, que vai da 34° a 36° hora, por volta das 5 h à 7 h do terceiro dia, destacada na Figura 4.6. Nesta região o psicrômetro indicou, média e desvio de $2 \times \sigma$, $71,42 \pm 0,19\%$ de RH, o SHT75 (corrigido) indicou $71,96 \pm 0,11\%$, o SHT31-0 indicou $73,12 \pm 0,10\%$ de RH, o HDC1080-0 indicou $75,30 \pm 0,10\%$ de RH, o DHT11-2 indicou $58,00 \pm 0,00\%$ de RH e o DHT22-1 $86,43 \pm 0,10\%$ de RH. O patamar descrito contém 785 pontos de umidade. A dispersão nula apresentada pelo DHT11-2 é consequência da sua baixa resolução (1%) (Aosong (Guangzhou) Electronics, b).

O conjunto de curvas da 4.5 representa parte de uma das várias coletas de dados realizadas, e o processo descrito nos últimos dois parágrafos foi repetido em todo conjunto

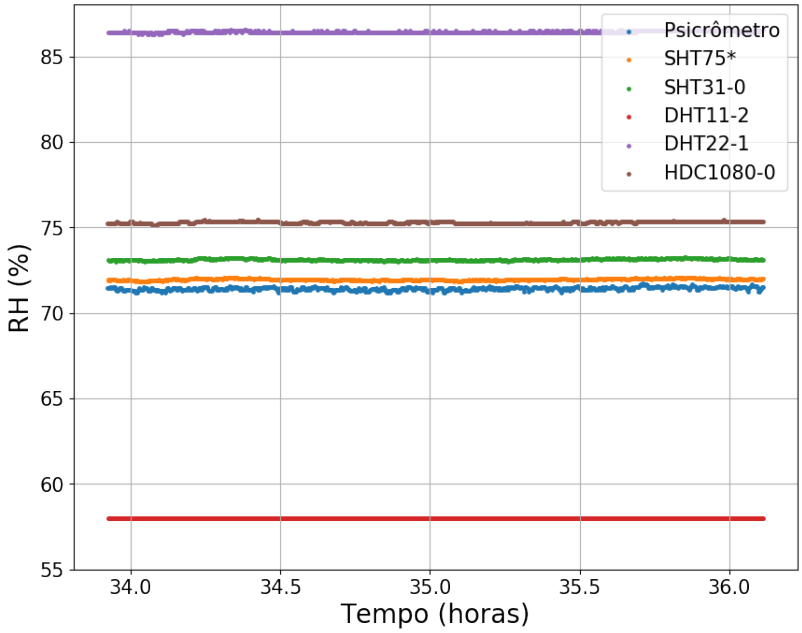


Figura 4.6: Fonte: Autores.

de dados, resultando em 28 patamares de estabilidade em RH. Cada patamar foi composto de pelo menos 197 pontos de cada sensor, e como referência não absoluta escolheu-se o SHT75 já corrigido. Os resultados obtidos a partir dos 28 patamares estão resumidos nas tabelas 4.2 e 4.3 e foram agrupados em seis faixas de RH: faixa de 44,77 % (2 patamares, RH = 44, 29 % e 45,24 %); faixa de 57,12 % (5 patamares, RH de 55,43 % a 59,18 %); faixa de 63,92 % (4 patamares, RH de 69,84 % a 66,33 %); faixa de 69,03 % (9 patamares, RH de 67,15 % a 71,62 %); faixa de 75,50 % (5 patamares, RH de 74,52 % a 76,47 %) e faixa de 96,29 % (3 patamares, RH = 94,03 % a 97,76 %).

Nas colunas "SHT75 (corrigido)" estão representados os valores de RH de referência e entre parênteses os valores de dois desvios padrão correspondente a flutuações do sensor SHT75. Nas colunas seguintes são representados os sensores por modelo, os valores correspondem a média entre sensores, do modelo indicado, subtraído da referência e entre parênteses os valores de dois desvios padrão correspondente a flutuação entre os sensores de mesmo modelo, indicando a diferença média esperada devido o câmbio de um sensor por outro de mesmo modelo, por exemplo, de um DHT-11 por outro DHT-11. Esse último valor pode ser usado para qualificar a cambiabilidade dos modelos de sensores.

A Tabela 4.2 sumariza os sensores SHT31 , SHT35 e HDC1080 . O SHT31 foi representado por 4 sensores e foi o que se mostrou mais regular, em acurácia e cambiabilidade,

Tabela 4.2 - Sumário dos desvios de valores de RH (%) em relação ao SHT75 corrigido, referência não absoluta. * Para valores de RH maiores que 94%, os sensores HDC- 1080 passam a registrar 99,9% de RH, i.e. passam ao estado de saturação. Fonte: Autores

SHT75 (corrigido) $RH \pm 2 \times \sigma\%$	SHT31 $\Delta RH \pm 2 \times \sigma\%$	SHT35 $\Delta RH \pm 2 \times \sigma\%$	HDC1080 $\Delta RH \pm 2 \times \sigma\%$
$44,77 \pm 0,07$	$1,07 \pm 0,67$	$1,36 \pm 1,86)$	$5,81 \pm 0,24)$
$57,12 \pm 0,12$	$-1,18 \pm 0,55$	$-0,34 \pm 1,70)$	$4,55 \pm 0,43)$
$63,92 \pm 0,20$	$-1,39 \pm 0,39$	$-0,42 \pm 1,18)$	$3,96 \pm 0,69)$
$69,03 \pm 0,12$	$-1,08 \pm 0,27$	$0,65 \pm 1,54)$	$3,67 \pm 0,75)$
$75,50 \pm 0,20$	$-1,20 \pm 0,56$	$1,48 \pm 2,63)$	$4,72 \pm 0,97)$
$96,29 \pm 0,52$	$-4,83 \pm 0,77$	$-1,68 \pm 3,49)$	$3,57 \pm 0,14)*$

ao longo de quase toda da faixa de umidade medida, divergindo para RH >90%. O SHT35 é superior ao SHT31 conforme o data sheet, contudo os resultados das 3 unidades testadas mostrou-se menos cambiável do que o SHT31 , os autores atribuem este resultado ao fato de que este foi o único sensor adquirido sem estar montado em uma placa de testes (foi montado localmente), os demais já foram adquiridos montados. O HDC1080 foi representado por duas unidades com boa concordância entre elas, contudo apresenta pouca acurácia se comparado ao SHT31 e ao SHT35 , além disso as unidades saturam (leitura de 99,99% de RH) a partir de 94% de RH ambiente. O desvio padrão para a faixa de 90% do HDC1080 , indicado por um "*" na tabela não possui significado, pois depois que o sensor satura o desvio colapsa a zero.

Tabela 4.3 - Sumário dos desvios de valores de RH (%) em relação ao SHT75 corrido, referência não absoluta. * média de 5 sensores, foram descartados 2 unidades consideradas inapropriadas para uso. ** média de 7 sensores. Fonte: Autores

SHT75 (corrigido) $RH \pm 2 \times \sigma\%$	DHT11 $\Delta RH \pm 2 \times \sigma\%$	SHT22* $\Delta RH \pm 2 \times \sigma\%$	DHT22** $\Delta RH \pm 2 \times \sigma\%$
$44,77 \pm 0,07$	$-5,93 \pm 1,84$	$-1,74 \pm 1,80$	$-0,10 \pm 4,03$
$57,12 \pm 0,12$	$-11,09 \pm 4,86$	$-3,63 \pm 1,97$	$1,63 \pm 10,03$
$63,92 \pm 0,20$	$-14,91 \pm 7,79$	$-5,00 \pm 2,13$	$1,34 \pm 11,60$
$69,03 \pm 0,12$	$-16,41 \pm 10,16$	$-5,00 \pm 2,44$	$0,63 \pm 10,43$
$75,50 \pm 0,20$	$-15,71 \pm 14,02$	$-3,83 \pm 3,25$	$-1,11 \pm 6,95$
$96,29 \pm 0,52$	$-14,56 \pm 15,78$	$-1,06 \pm 2,56$	$-1,59 \pm 2,62$

A Tabela 4.3 sumariza os resultados para os sensores DHT11 e DHT22 . O DHT11 foi representado por 3 sensores e foi o que apresentou o maior desvio padrão entre eles, i.e.

menor cambiabilidade, e maior divergência contra a referência. O DHT22 foi representado por 7 unidades, contudo duas destas unidades apresentaram valores divergentes em mais de 10 unidades de RH em relação a média das demais unidades, portanto o resultado do DHT22 é apresentado em dois grupos. Com um "*" o resultado para 5 sensores e o com "**" para 7 sensores e a discrepância pode ser evidenciada pelos valores dos desvios padrão.

Capítulo 5

Conclusão

Neste trabalho foi apresentado uma rede de sensores (FURG CAN) baseada no protocolo CAN e de implementação acessível. A conexão entre os elementos da FURG CAN é feita por um barramento com um par trançado (linhas de dados) e mais uma via (GND), que pode ser feito, por exemplo, com cabos Cat5 ou Cat6, que são usados em redes de computadores. Os nodos (elementos da rede) são construídos com dispositivos populares, como o Ardunio Nano (Arduino, 2020a). Para ajudar a gerenciar o protocolo de comunicação, é utilizado um módulo CAN. Uma placa de teste montada com o controlador CAN MCP2515 (Microchip Technology Inc, 2019) e o transceptor TJA1050 (Philips, 2003). Para que se tenha um parâmetro do custo de implementação, o módulo CAN custa, em geral e no momento, no mercado nacional, a metade do valor do Arduino Nano. No mercado internacional eles possuem o mesmo valor. Os demais componentes da FURG CAN, são cabos e conectores. Além, dos dispositivos específicos de cada aplicação, que possuem uma grande variedade de valores. No estágio deste trabalho, a FURG CAN é uma rede de publicação e monitoramento, *i.e.*, os elementos sensores (nodos sensores) enviam dados a um receptor (nodo *sink*).

O protocolo escolhido, CAN 2.0 (*Controller Area Network*), é relativamente pequeno, e bem difundido na Indústria, sendo estabelecido como padrão na década de 1990. Já foi amplamente testado e há uma boa oferta de dispositivos compatíveis no mercado. A FURG CAN não suporta a taxa máxima de transmissão do protocolo usado (1 M *bits*/s). O tamanho do barramento, distância máxima entre dois nodos comunicantes, também é menor que a especificada. Essas divergências são impostos pelo *clock* (de 8 MHz) fornecido ao controlador CAN MCP2515. O módulo CAN possui um cristal que, fornece a base de tempo do controlador CAN.

O comprimento máximo do barramento da FURG CAN varia de acordo com a taxa de operação, sendo de até 275 m para 125 k *bits*/s, 125 m para 250 k *bits*/s e de até 50 m para a maior taxa suportada, 500 k *bits*/s. Esses valores foram obtidos teoricamente, e não foram testados por falta de cabos com essas dimensões. O transceptor usado, o TJA1050, limita o número de nodos no barramento em até 110 nodos (Philips, 2003), esse limite não foi testado.

Todos os processos relacionados ao protocolo CAN são executadas pelo controlador CAN MCP2515, através de funções da biblioteca criada, pelos autores, a *MCP2515-bib* (Apêndice B). Também são fornecidos exemplos de códigos que podem ser usados como *template* para elaboração de programas específicos. A programação dos nodos tende a ser o processo mais demorado da implementação da FURG CAN. As rotinas específicas da FURG CAN possuem sintaxe curta. O que mais deve demandar tempo, são os códigos dos dispositivos específicos. Dependendo do dispositivo, o trabalho de programação pode ser encurtado pela utilização de exemplos produzidos previamente pela comunidade de usuários, contudo, levará no mínimo o tempo da pesquisa e adaptação de soluções prontas. Ter uma rede fixa pode ser algo vantajoso. Pois pode-se adicionar, excluir ou modificar nodos, de modo a, atualizar a rede para um novo experimento. O que, em geral, consumirá menos tempo do que uma nova implementação.

A taxa de ocupação do barramento é limitada pela capacidade de processamento dos nodos *sinks* da rede, CAN Mon e CAN Save. No caso menos favorável, podem ser enviadas até 454,9 mensagens por segundo, em uma rede com um CAN Mon, operando somente com *frames* estendidos de 8 *bytes* de dados. Se a mesma rede utilizar, exclusivamente, *frames* padrões com 4 *bytes* de dados, pode se ter um tráfego, com até 762,2 mensagens por segundo. Na configuração mais rápida da FURG CAN, com um CAN Save no lugar do CAN Mon e, utilizando somente *frames* padrões com 2 *bytes* de dados, podem trafegar até 2830,2 mensagens por segundo. Entretanto, a plataforma de rede de sensores apresentada é flexível, sendo possível, com um pouco de esforço, altera-lá para atender especificidades ou ainda otimizá-la.

A FURG CAN foi utilizada em um aplicação, onde foram monitorados 19 transdutores digitais, termo-higrômetros e um psicrômetro com dois Pt100 AA. Durante esta aplicação foram coletadas um pouco mais do que 437 horas de dados, valores de temperatura e umidade relativa. O tráfego médio de publicações foi de 3,2 *frame*/s e, o maior intervalo

ininterrupto, de coleta durou 71 h 41 min 24 s. A FURG CAN apresentou um funcionamento estável durante toda a aplicação, operando satisfatoriamente e de forma ininterrupta durante dias. A maior taxa de perda observada durante a aplicação foi de $0,18\%$, e não comprometeu as análises subsequentes.

5.1 Trabalhos Futuros

Destaca-se dentre as possibilidade de trabalhos futuros, a implementação de nodos *sinks* com unidades de controle mais potentes, de modo a aumentar os limites de ocupação do barramento da FURG CAN, por exemplo, como um Raspberry Pi e ou um ESP32.

A inclusão de um dicionario de identificadores é um possível ponto de melhora futura. Ele contém informações sobre a formatação dos dados enviados, por todos os nodos sensores conectados ao barramento. Com essa informação, os nodos monitores, CAN Mon ou CAN Save, poderão realizar conversões, operações matemáticas, entre outros trabalhos específicos, e diferenciados para cada tipo de dado recebido. Isso tudo antes mesmo de enviar ao computador. O que pode minimizar um incômodo: Os dados salvos no computador ou no cartão micro SD, são formatados em bytes. Isso não acontece para as demais informações, ID padrão e extensão. A conversão não é complicada, trata-se apenas de uma concatenação de bytes para o formato original dos dados.

A proposta inicial foi elaborada apenas para monitoramento, *i.e.*, não foi criado nenhum mecanismo prévio de suporte a operações de controle. Entretanto, entende-se que esse tipo função é interessante para uma rede e, é outro potencial trabalho futuro. Ressalta, que novos mecanismos podem adicionar mais funções ao nodo mais sensível da FURG CAN, o CAN Mon, e por isso a substituição da plataforma de processamento, Arduino Nano, é aconselhável antes de implementar um novo mecanismo. Abaixo segue uma lista dos potenciais trabalhos futuros.

- Nodo *sink* com o ESP32: Com processador dual-core de 32 *bits* e com um drive interno equivalente ao protocolo CAN 2.0, o ESP32 é um potencial nodo *sink*. Além disso ele fornece uma interface de conexão *Wi-Fi* possibilitando explorar outros caminho para o armazenamento definitivo dos dados.

- Nodo *sink* com o Raspberry Pi: O Raspberry Pi pode realizar a leitura direta do controlador CAN MCP2515. Uma adaptação da biblioteca, nesse sentido, possibili-

taria a conexão 'direta' de um modesto computador (o Raspberry) ao barramento. Um nodo *sink* com um Raspberry pode ser muito mais capaz em processamento e armazenamento do que um feito com microcontroladores convencionais.

- Comunicação bilateral: Implementar um mecanismo de comunicação bilateral que auxilie processos de controle. Aumenta o leque de aplicações possíveis da FURG CAN.

- Dicionario de ID: A inserção de dicionários de ID, contendo informações sobre a mensagem, em nodos do tipo *sink*. Possibilita um pré-processamento mais sofisticado, que por exemplo, converter os dados para um formato mais intuitivo antes do envio para o computador.

Referências Bibliográficas

Adafruit, 2020 SHT31-D Temperature and Humidity Sensor Breakout https://github.com/adafruit/Adafruit_SHT31

Adafruit, 2021 DHT sensor library https://github.com/adafruit/DHT-sensor-library

Aiea V. M., , 2020 Interfacing Catalex Micro SD Card Module with Arduino, www.vishnumaiea.in

Aosong (Guangzhou) Electronics DHT22 (also AM2302). Digital-output relative humidity and temperature sensor/module, www.aosong.com

Aosong (Guangzhou) Electronics Temperature and humidity module. DHT11 Product Manual

Araújo J. W. B., Ferrari F., Kakuno E., , 2020 Biblioteca MCP2515 para o Arduino https://github.com/KakiArduino/CAN-Bus-MCP2515

Arduino SPI Library, www.arduino.cc

Arduino, 2020a Arduino Nano, www.arduino.cc

Arduino, 2020b Documentação de Referência da Linguagem Arduino, www.arduino.cc

ASTM international, 2020 Designation: E104-20 Standard Practice for Maintaining Constant Relative Humidity by Means of Aqueous Solutions, www.astm.org

Bosch R., , 1991 CAN Specification https://edisciplinas.usp.br/mod/resource/view.php?id=1744822

Bosch R., Manual de tecnologia automotiva. São Paulo: Edgard Blücher, 2005, 1072

Bosch R., , 2012 CAN with Flexible Data-Rate http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can_fd_spec.pdf

CiA CAN, 2021 CANopen – The standardized embedded network https://www.can-cia.org/canopen/

DARGIE Waltenegus; POELLABAUER C., Fundamentals of Wireless Sensor Networks: Theory and Practice 1 edn. Wireless Communications and Mobile Computing, John Wiley & Sons, 2010

Ferrando D. F. C., Araujo J. W. B. d., M. K. E., , 2017 ADS124X https://github.com/KakiArduino/ADS124X

Greiman W., , 2009 SDFat.h https://github.com/adafruit/SD/blob/master/utility/SdFat.h

INMETRO Vocabulário Internacional de Metrologia. http://www.inmetro.gov.br/inovacao/publicacoes/vim_2012.pdf, 2012

ISO, 1994 Standard Information technology - Open Systems - Basic Reference Model: Interconnection The Basic Model. International Organization for Standardization Geneva, CH

ISO, 2003a Standard Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. International Organization for Standardization Geneva, CH

ISO, 2003b Standard Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit. International Organization for Standardization Geneva, CH

Kiencke U., Dais S., Litschel M., Automotive Serial Controller Area Network, SAE Technical Paper 860391, 1986

Lawrenz W., *et al* CAN System Engineering: From Theory to Practical Applications. Second Edition. Lodon: Springer, 2013

Margolis M., Arduino Cookbook. 2. Sebastopol: O'Reilly, 2011

Maxim Integrated, 2015 Precision Thermocouple to Digital Converter with Linearization https://datasheets.maximintegrated.com/en/ds/MAX31856.pdf

Microchip T. I., , 2020 ATmega48A/PA/88A/PA/168A/PA/328/P http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf

Microchip Technology Inc, 2019 MCP2515: Stand-Alone CAN Controller with SPI Interface http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf

OMEGA Engineering inc. Fio de termopar isolado com PFA de trançado único de bitola fina: TFAL-003 https://br.omega.com/pptst/TFIR_CH_CI_CC_CY_AL.html

OMEGA Engineering inc. Fio de termopar isolado com PFA de trançado único de bitola fina: TFCY-003 https://br.omega.com/pptst/TFIR_CH_CI_CC_CY_AL.html

OMEGA Engineering inc. OMEGAFILM RTD Elements: Flat Profile Thin Film Platinum https://br.omega.com/pptst/F_SERIES.html

Philips S., , 1997 SJA1000: Stand-Alone CAN Controller https://www.nxp.com/docs/en/application-note/AN97076.pdf

Philips S., , 2003 TJA1050 High speed CAN transceiver https://www.nxp.com/docs/en/data-sheet/TJA1050.pdf

Sensirion, 2011 Datasheet SHT7x. Humidity and Temperature Sensor IC SHT7x Datasheet

Sensirion, 2019 Datasheet SHT3x-DIS. Humidity and Temperature Sensor SHT3x Datasheet

SPEASE PER1234, 2014 Sensirion: An Arduino Library for the Sensirion SHT1x and SHT7x family of temperature and humidity sensors https://github.com/spease/Sensirion

Systems E., , 2021 ESP32-C3 Family Datasheet https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf

Texas Instruments, 2016a ADS124x24-Bit, 2-kSPS, Analog-To-Digital Converters With Programmable Gain Amplifier (PGA) For Sensor Measurement https://www.nxp.com/docs/en/data-sheet/TJA1050.pdf

Texas Instruments, 2016b HDC1080 Low Power, High Accuracy Digital Humidity Sensor with Temperature Sensor https://www.ti.com/lit/ds/symlink/hdc1080.pdf?ts=1615960244031

Texas Instruments, 2017a How to interface with HDC1080 humidity and temperature sensor with Arduino using I2C https://training.ti.com/how-interface-hdc1080-humidity-and-temperature-sensor-arduino-using-i2c

Texas Instruments, 2017b LM35 Precision Centigrade Temperature Sensors http://www.ti.com/lit/ds/symlink/lm35.pdf

Waltenegus D., Poellabauer C., FUNDAMENTALS OF WIRELESS SENSOR NETWORKS. John Wiley and Sons, 2010

Glossário

ADC Termo usado para se referir a qualquer conversor de sinal analógico pra sinal digital. O termo é uma abreviatura do inglês, Analog-to-Digital Converter.. 32, 34–36

ADC ADS1248 Conversor analógico digital da Texas Instruments. Para mais informações visitar o site oficial ADS1248.. 119, 122, 125, 126

Arduino IDE Integrated Development Environment. Ambiente de programação, compilação, carregamento e monitoramento para placas compatíveis. Mais informações no site oficial Arduino IDE.. 39, 40

Arduino Mega Plataforma microcontrolada de desenvolvimento aberta. Mais informações no site oficial: Arduino Mega.. 81, 120–123

Arduino Nano Plataforma microcontrolada de desenvolvimento aberta (Arduino, 2020a), elaborada com o microcontrolador ATmega328.. 25, 26, 77, 78, 80, 81, 83–89, 92–94, 96–98, 110, 111, 113–117, 122

Arduino Portenta H7 Plataforma microcontrolada de desenvolvimento aberta. Mais informações no site oficial: Arduino Portenta.. 37

Arduino Uno Plataforma microcontrolada de desenvolvimento aberta. Mais informações no site oficial: Arduino Uno.. 37, 81

ATmega328 É um microcontrolador criado pela Atmel e, hoje em dia, produzido pela Microship ATmega328. Ele é usado nas plataformas Arduino Nano e Arduino Uno.. 37, 38, 139

Bit-Timing É a duração temporal de um bit.. 13, 43, 44, 50–53

bootloaders Plural de bootloader, programa usado para inicialização, programa de boot. No caso de microcontroladores, o bootloader, permite uma programação mais rápida.. 33

broadcasting Método de transmissão, no qual as mensagens são simultaneamente enviadas para, todos os receptores.. 42, 48

buffer Espaço de memória temporário.. 48, 55, 57, 58, 82, 88–91, 94, 95

CAN Controller Area Network. É um protocolo de comunicação serial, apresentado pela BOSCH em 1986 (Kiencke et al., 1986).. 13, 14, 17, 25, 41–50, 52–57, 59–63, 65–67, 69, 70, 72–74, 97–99, 103, 110, 116

CAN 2.0 Atualização do protocolo CAN, na qual foi adicionando a extensão de ID, incluindo 17 bits extras de identificação. Uma descrição completa pode ser consultada na publicação da BOSH de 1991. (Bosch, 1991).. 25, 31, 64, 115

CAN FD CAN Flexible Data Rate é o protocolo mais atual da família CAN. Ele possibilita o envio de até 64 bytes de dados por frame, e pode acelerar a taxa de transmissão de bits durante a publicação (Bosch, 2012).. 41, 42

CAN HIGH Uma das duas linha de dados do barramento CAN. Apresenta 2,5 V, em estado recessivos, 1 digital. E durante estados dominantes, 0 digital, apresenta 3,5 V.. 14, 43–45, 47, 80, 82–84, 89, 97, 99, 103–109, 115, 123

CAN high-speed É o protocolo de alta velocidade da CAN, até 1 M bit/s. A FURG CAN é construída sobre essa versão da CAN.. 41

CAN LOW Uma das duas linha de dados do barramento CAN. Apresenta 2,5 V, em estado recessivos, 1 digital. E durante estados dominantes, 0 digital, apresenta 1,5 V.. 14, 43–45, 47, 80, 82–84, 89, 97, 99, 103–109, 115, 123

CAN low-speed É o protocolo de baixa velocidade da CAN, mas precisamente operar a 125 k bit/s. Sua imunidade a falhas é mais ampla do que para a versão high-speed.. 41

CAN Mon Um dos nodos dois nodos sink da FURG CAN. Ele recolhe as mensagens do barramento CAN, e as envia para um computador via USB-Serial.. 14, 17, 78–81, 85, 87–91, 93–96, 98, 111–117, 122, 125

CAN Save Um dos nodos dois nodos sink da FURG CAN. Ele recolhe as mensagens do barramento CAN, e as armazena em um cartão SD via SPI.. 14, 17, 78–81, 85, 91–96, 113–117

CAN Sensor Nodo sensor genérico da FURG CAN.. 14, 15, 78–81, 85, 94–96, 98, 110, 113, 121, 122, 124

CI Abreviatura de Circuito Integrado. Dispositivos que agregam um circuito inteiro em um único chip.. 33

data-logger Dispositivo utilizado para armazenamento local de dados.. 32

DHT11 Termo-higrômetro digital da Aosong. Para mais informações visitar o site oficial DHT11.. 17, 119–121, 125, 126, 129

DHT22 Termo-higrômetro digital da Aosong, também chamado de AM2302. Para mais informações visitar o site oficial AM2302.. 17, 119–122, 125, 126, 129, 130

DSLogic Analisador logico da Dream Source Lab. Para mais informações visitar o site oficial DSLogic.. 97–99

ESP32 Microcontrolador da Espressif, mais informações no site oficial ESP32.. 46, 81, 116

ESP8266 Microcontrolador da Espressif. Para mais informações visitar o site oficial ESP8266.. 116

First In First Out Memoria FIFO, é um mecanismo de armazenamento que garante que o primeiro dado armazenado seja o primeiro a sair da de pilha armazenamento.. 48, 88, 89

frame Frame é uma mensagem codificada. Um frame contém a informação de interesse e uma séries de outras informações relacionadas ao controle da comunicação.. 14, 99–102, 110–114

frames Plural de frame.. 98–100, 102, 110–116

FURG CAN Transposição tocológica do protocolo CAN para redes de sensores de pequeno porte em plataforma aberta. No estado atual de desenvolvimento a FURG CAN

suporta apenas aplicações de monitoramento. O repositório oficial, Kaki, contém os os códigos necessário para implementação, exemplos e outras informações.. 14, 17, 25, 26, 40, 41, 43, 77–81, 83–87, 91, 93, 94, 96–99, 103, 106–108, 110, 111, 113–116

GND abreviatura de *ground*, terra em inglês. Simboliza, em geral e neste trabalho, a tensão usada como referência comum em um circuito elétrico, sendo que as diferenças de potenciais são normalmente medidas em ralação ao GND. Também é conhecido como dreno comum, pois costuma apresentar uma tensão de 0 V. O GND também pode indicar um ponto de aterramento (conexão com o solo).. 14, 43, 45, 47, 83, 97, 103–107, 115, 123

HDC1080 Termo-higrômetro digital da Texas Instruments. Para mais informações visitar o site oficial HDC1080.. 15, 17, 119, 120, 122, 124–126, 128, 129

HP 5316B Contador universal da HP. Mais informações nessa versão digital do datasheet HP 2316B.. 98, 110

I2C Padrão de comunicação serial, desenvolvido pela Philips. Ele um dos protocolos mais usados em sistemas embarcados.. 35, 38, 120

IEE 802.11 Conjunto de protocolos, que fazem parte da família IEE 802. Ele fornece os protocolos de controle de acesso ao meio e da camada física, para redes de computadores sem fio.. 27, 145

instrumento de medição Aparato utilizado na obtenção de valores referentes a medição (INMETRO, 2012).. 26

IoT Internet of Things. Internet das coisas, é um movimento tecnológico, que promove a interconexão de dispositivos corriqueiros. Por exemplo, redes de dispositivos residenciais.. 27, 33

ISO International Organization for Standardization. Organização Internacional de Normalização, foi responsável pelas normatizações do protocolo CAN. Para mais informações consultar o site oficial: iso.org.. 27, 41

MCP2515 Controlador CAN da Microchip (Microchip Technology Inc, 2019).. 25, 26, 38, 40, 46, 48, 77, 78, 80–83, 85, 88–91, 94, 95, 97, 110–112, 116, 120–122

MCP2515-bib Biblioteca do CI MCP2515 para Arduino IDE, desenvolvida pelos autores (Araújo et al., 2020).. 26, 40, 81, 83, 89, 91, 92, 94, 96, 112, 116, 125

MOSFET Metal Oxide Semiconductor Field Effect Transistor. É mais popular transistor de efeito de campo.. 124, 125

nodo sensor Dispositivo comunicante de uma rede de sensores, cujo objetivo é coletar, pré-processar e, compartilhar dispositivos informações do meio. Eles também pode ser chamado de sensor, sensor remoto.. 26, 31

nodo sink Dispositivo comunicante de uma rede de sensores, cujo objetivo é recolher e pré-processar, as informações publicados na rede pelos demais dispositivos conectados.. 26, 29, 31

OSI Open Systems Inter-connection. Modelo de organização de protocolos de comunicação para sistemas interligados (redes) (ISO, 1994).. 27, 29–31

psicrômetro Instrumento de medição de umidade relativa. Elaborado com dois termômetros, um para temperatura ambiente (bulbo seco) e outro para a temperatura de um tecido úmido (bulbo úmido). A taxa de evaporação da água do tecido, depende da temperatura ambiente e da umidade relativa do ambiente, além de outros fatores. Quanto maior a taxa de evaporação maior será a diferença de temperatura entre o bulbo seco e o o úmido (depressão psicrométrica). Existem diversas equações semi-empíricas para calcular a umidade relativa em função das temperaturas do bulbo seco e úmido.. 119, 122, 125–127

Pt100 AA Pt100 AA modelo F2222-100-1/3B da OMEGA (OMEGA Engineering inc., c). Pt100 é um transdutor termico, seu elemento sensivel, ou sensor, é uma resistencia de platina, que apresenta 100 Ω a temperatura de 0 °C. A platina confere uma variação linear da resistencia em relação a termperatura.. 119, 122, 126

PWM Pulse Wave Modulation, é um mecanismo digital, para produção de saída analógica. O princípio de funcionamento é pulsar uma saída digital, de modo que, ela passe uma parte do período em estado alto, ou outra em estado baixo. A tensão de saída é o porcentual de tempo ligado, em relação a tensão de estado alto.. 37

Raspberry Pi É um computador de placa única. Para mais informações visitar o site oficial Raspberry Pi.. 79, 80, 116, 122

rollover Técnica de transferir uma informação entre dois espaços de memoria. Normalmente de um espaço mais prioritário para um menos prioritário.. 88, 91

SDIO Padrão de comunicação de cartões de memória SD. E mais eficiente para transferência de arquivos, do que o padrão SPI, outra opção de comunicação para cartões SD. Para mais informações visitar o site oficial SD Association.. 92

SHT31 Termo-higrômetro digital da Sensirion. Para mais informações visitar o site oficial SHT31.. 17, 119–122, 125, 126, 128, 129

SHT35 Termo-higrômetro digital da Sensirion. Para mais informações visitar o site oficial SHT35.. 17, 119–122, 125, 128, 129

SHT75 Termo-higrômetro digital da Sensirion. Para mais informações visitar o site oficial SHT75.. 17, 119–121, 125–129

sistema de medição Conjunto de um ou mais instrumento de medição e dispositivos acessórios, elaborado para obtenção de valores referentes a medição (INMETRO, 2012).. 26

SPI Padrão de comunicação serial. Ele um dos protocolos mais usados em sistemas embarcados.. 35, 38, 82, 83, 85, 88, 89, 91–93, 113, 116, 124

TCP/IP Transmission Control Protocol/Internet Protocol. Conjunto de protocolos muito usados na comunicação entre computadores.. 25, 31, 84

TDS 210 Osciloscópio da Tektronix Uma descrição do equipamento pode ser consultada Aqui.. 51, 98

TDS 520D Osciloscopio da Tektronix Completa eu, vai. Uma descrição do equipamento pode ser consultada no manual de usuários TDS 520 & 540.. 97, 98, 105, 106

time quanta Menor unidade temporal, utilizada pelo controlador CAN, na construção de um bit. Ou seja a duração de um bit em medida em time quanta pelo controlador CAN.. 50

TJA1050 Transceptor CAN da Philips (Philips, 2003).. 25, 43, 77, 78, 80–82, 85, 98, 99, 103–105, 122

transdutor Dispositivo tecnológico, que produz um saída previsível com relação a entrada (INMETRO, 2012). No mínimo um transdutor é composto por elemento sensível e uma relação regras (*e.g.*, matemáticas) de conversão. Com essas regras deve ser possível obter o valor da entrada a partir da saída.. 32, 34–36

TTCAN Time Triggered CAN. É um mecanismo de distribuição temporal de mensagens para redes CAN. Ou seja, é um protocolo de comunicação para CAN, lançado em 2001 na SAE. link para o artigo TTCAN.. 41

TWAI *Two-Wire Automotive Interface*. Interface de comunicação para protocolo automotivo que utilize dois fios, como a CAN. Nome usado pela Espressif, fabricante do ESP32, para referir a engine interna da CAN de seu microcontrolador.. 46, 48

UART Abreviatura de Universal asynchronous receiver/transmitter. Dispositivo que gerencia comunicações paralelas, recepção e transmissão simultâneas. São, normalmente, usados em padrões de comunicação serial, com o RS-232C.. 35, 37, 38

USB Universal Serial Bus. Protocolo de comunicação serial. A sua primeira versão foi lançada em 1996 e hoje em dia é amplamente utilizada.. 37, 80, 90, 98

UTP Abreviatura de *Unshielded Twisted Pair*. Refere-se genericamente a qualquer par de fios, individualmente isolados, trançados e sem blindagem.. 43

Wi-Fi É uma marca da Wi-Fi Alliance. Serve como certificação para dispositivos de rede local sem fio (WLAN: Wireless Local Area Network). A WLAN, por sua vez, é definida pelo padrão de protocolos IEE 802.11. Popularmente o termo Wi-Fi é usado para se referir a WLAN.. 27

Apêndice

Apêndice A

Taxas de transmissões efetivas da CAN

Tabela A.1 - Taxas de transmissão de dados máximas da CAN, para frames padrão (std) e estendidos (ext), sem (esquerda) e com o máximo (direita) de *stuffing bits*, para as frequências padrões. As quatro colunas da extrema esquerda, possuem os números de *bits* para os possíveis frames.

Frame *nbits* em *bits*				para 125 kbit/s em kbit/s				250 em kbit/s				500 em kbit/s				1000 em kbit/s			
std		ext		std		ext		std		ext		std		ext		std		ext	
55	63	75	87	18,18	15,87	13,33	11,49	36,36	31,75	26,67	22,99	72,73	63,49	53,33	45,98	145,45	126,98	106,67	91,95
63	73	83	97	31,75	27,4	24,1	20,62	63,49	54,79	48,19	41,24	126,98	109,59	96,39	82,47	253,97	219,18	192,77	164,95
71	82	91	106	42,25	36,59	32,97	28,3	84,51	73,17	65,93	56,6	169,01	146,34	131,87	113,21	338,03	292,68	263,74	226,42
79	92	99	116	50,63	43,48	40,4	34,48	101,27	86,96	80,81	68,97	202,53	173,91	161,62	137,93	405,06	347,83	323,23	275,86
87	101	107	125	57,47	49,5	46,73	40	114,94	99,01	93,46	80	229,89	198,02	186,92	160	459,77	396,04	373,83	320
95	111	115	135	63,16	54,05	52,17	44,44	126,32	108,11	104,35	88,89	252,63	216,22	208,7	177,78	505,26	432,43	417,39	355,56
103	121	123	145	67,96	57,85	56,91	48,28	135,92	115,7	113,82	96,55	271,84	231,4	227,64	193,1	543,69	462,81	455,28	386,21
111	130	131	154	72,07	61,54	61,07	51,95	144,14	123,08	122,14	103,9	288,29	246,15	244,27	207,79	576,58	492,31	488,55	415,58

Apêndice B

Biblioteca MCP2515

A biblioteca MCP2515 foi feita para auxiliar/facilitar o controle do CI da Microchip, de mesmo nome, por plataformas Arduino, ou compatíveis. As funções de configuração possuem valores padrão, de modo a fornecer uma configuração simples e rápida. O CI MCP2515 configurado no modo padrão dessa biblioteca estará preparado para operar em uma rede de 125 k/bits, sem a implementação de filtros de aceite nos *buffers* de entrada, com *rollover* do *buffer* de saída RXB0 para o RXB1, e além disso são habilitadas interrupções de recebimento e sucesso de envio. Para setar outra configuração o usuário deverá atualizar os valores das variáveis de configuração B.2.2, e em seguida executar a função correspondente aos registros modificados, como a **confMode()** B.3.8, **confRX()** B.3.9, **confTX()** B.3.10, **confINT()** B.3.11, **confFM()** B.3.12 e **confCAN()** B.3.13. Ou ainda as usando as funções de escrita básica, **write (REG, VAL, CHECK = 1)** B.3.6 e **bitModify (REG, MASK, VAL, check = 0)** B.3.7.

Os frames podem ser escritos no barramento com a função **writeFrame (frameToSend, txb_ = 0, timeOut = 10, check = 0)** B.3.19 e podem ser lidos pela função **readFrame ()** B.3.22 que atualiza os valores dos frames internos frameRXB0 e frameRXB1. As informações da mensagem são armazenadas na estrutura apresentada na Figura B.1, e podem ser acessados pelas variáveis descritas na seção B.1. Também é possível obter os dados acessando diretamente os registros correspondentes aos buffers de entrada do MCP2515 com a função **read (REG, data, n = 1)** B.3.4. De forma análoga pode-se escrever nos buffers de saída do MCP2515 e solicitar o envio usando as funções de escrita básicas.

A principal referência desta biblioteca é o *datasheet* MCP2515 Stand-Alone CAN Controller with SPI Interface. Por questões de comodidade muitos dos códigos numéricos

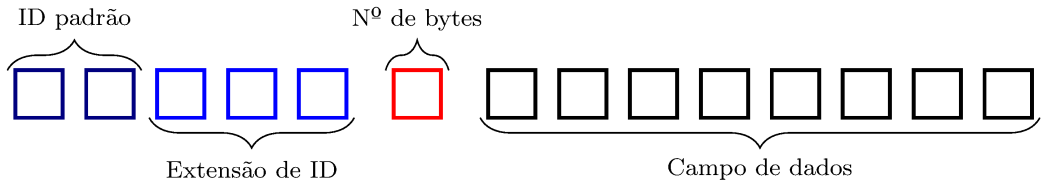


Figura B.1: Organização dos *bytes* em *frames* de dados. Fonte: Autores.

relacionados a comunicação do MCP2515 com a plataforma Arduino foram expressos em hexadecimal indicado por 0x, e quando não estarão em decimal.

Um exemplo de conexão com um Arduino Nano e um módulo CAN (MCP2515 + TJA1050 (Philips, 2003)) pode ser visto na Figura B.2.

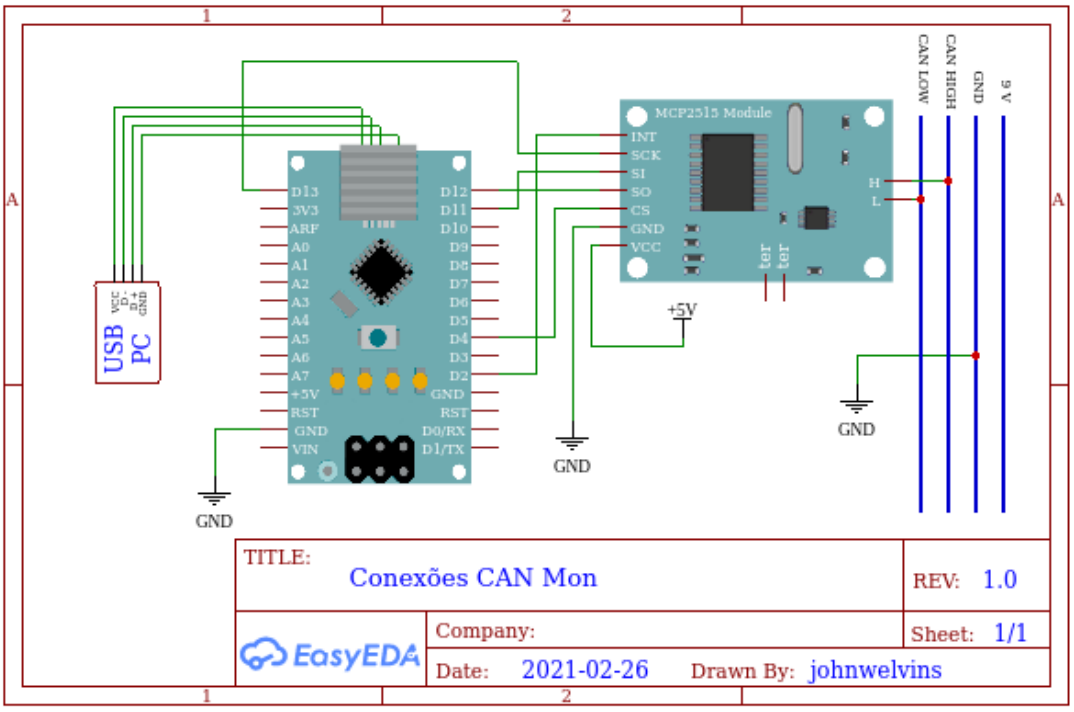


Figura B.2: Diagrama de conexão do MCP2515 no Arduino Nano. Fonte: Autores.

O barramento é composto por pelo menos dois dispositivos, na Figura B.3 pode-se ver um diagrama simplificado de um barramento CAN.

São fornecidos 5 exemplos de rotinas, uma para transmissão **CANTX.ino**, um para recepção **CANRX.ino**, dois exemplos de nodos *sink* (receptores), o **CANMon.ino** envia para as mensagens para um computador pela porta USB-Serial, e o **CANSave.ino** que salvas as mensagens recebidas em um cartão SD usando SPI via software. Também é

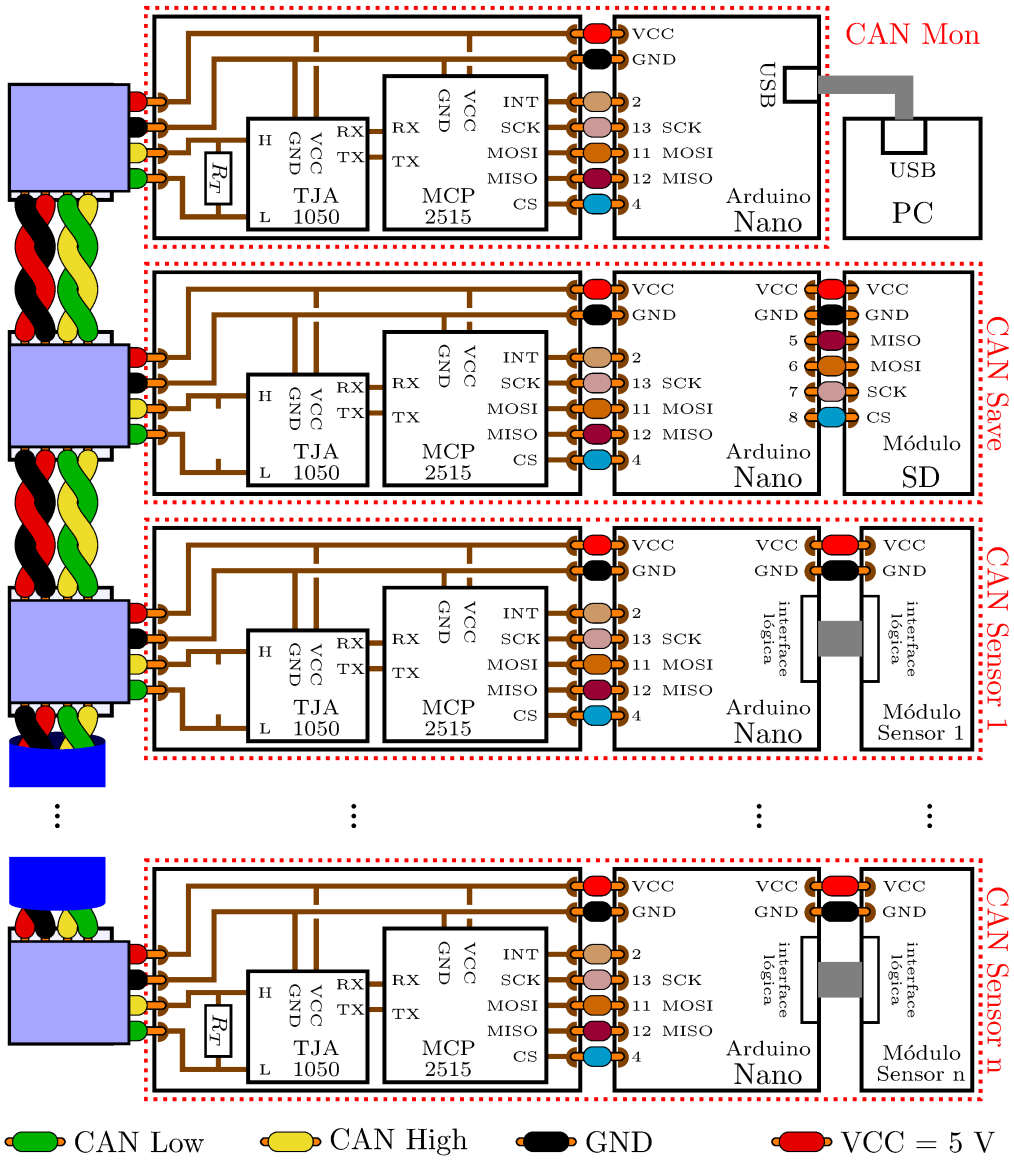


Figura B.3: Representação de um barramento. Fonte: Autores.

fornecido um template para nodos sensores, o exemplo **CANSensor.ino**.

B.1 Frames

B.1.1 Variáveis de um frame

- `uint16_t id_std;`

  Variável de 2 *bytes* que armazena o valor do ID padrão do *frame*.

- `uint32_t id_ext;`

Variável de 4 *bytes* que armazena o valor da extensão do ID do *frame*.

- `uint8_t dlc;`

  Variável de 1 *byte* que armazena o código de comprimento, número de *bytes* de dados, do *frame*.

- `uint8_t data[8];`

  Lista com os *bytes* de dados do *frame*.

- `uint8_t bts[14];`

  Lista com todos os *bytes* do *frame*.

- `String type = "Unknown";`

  String com o tipo do *frame*, padrão ("Std. Data"), estendido ("Ext. Data") ou "No frame"que é usado para indicar que há novos frames nos *buffers* de entrada do MCP2515.

B.1.2 ***CANframe*** *()*

```
CANframe();
```

Função para declaração de uma estrutura de variáveis do tipo *CANframe* sem argumentos.

- Exemplo de uso:

  1. Declaração de um *frame* CAN.

     ```
     CANframe frm();
     frm.id_std = 0x7FF;
     ```

     Declaração de um *frame* sem fornecer parâmetros de entrada, seguida, na linha de baixo, pela atribuição de 0x7FF para o ID padrão, o maior valor possível, do *frame* criado acima.

B.1.3 ***CANframe*** *(frameBytes, extFlag)*

```
CANframe(uint8_t *frameBytes, uint8_t extFlag = 0);
```

Função para criação de *frame*, a partir de uma lista com todos os *bytes* do *frame*.

- Parâmetros de entrada:
  - **frameBytes** lista com todos os *bytes* do a serem atribuídos ao *frame*.
  - **extFlag** sinalizador de extensão de ID, se 0 o *frame* é padrão, se 1 é estendido.
- Exemplo de uso:
  1. Declaração de um *frame* CAN.

```
frameBytes [10] = {0x07, 0xFF,
                   0x03, 0xFF, 0xFF,
                   0x04,
                   0x3, 0xFF, 0xF7, 0x21};


CANframe frm(frameBytes, 1);


Serial.println(frm.id_std);
```

Primeiro é declarado a lista *frameBytes*, os dois primeiros *bytes* (0 e 1) compõem o ID padrão, concatenando tem-se 0x7FF o maior valor possível. Os três *bytes* seguintes da lista *frameBytes*, posições 2, 3 e 4, correspondem a extensão de ID, que neste caso vale o valor máximo 0x3FFFF. O *byte* seguinte, posição 5, é o código dlc, que no caso indica quatro *bytes* de dados.

Após a declaração da lista com os *bytes*, é feito a declaração do *frame* estendido com a lista *frameBytes*. Por fim o ID padrão é impresso através da variável *id_std* na última linha, isto é possível, pois ao declarar o *frame*, todos suas variáveis são atribuídas.

B.1.4 ***CANframe*** *(idstd, idext, dlc_, data)*

```
CANframe(uint16_t idstd, uint32_t idext, uint8_t dlc_, uint8_t *data);
```

Função para criação de *frame* estendido com declaração do ID padrão, extensão de ID, DLC - *Data Len Code* e uma lista de bytes de dados, data.

- Parâmetros de entrada:
  - **idstd** variável de 2 *bytes* onde deve ser informado o valor do ID padrão do *frame*, no máximo 0x7FF.
  - **idext** variável de 4 *bytes* onde deve ser informado o valor da extensão de ID do *frame*, no máximo 0x3FFFF.
  - **dlc_** é o número de *bytes* de dados.
  - **data** é a lista contendo os *bytes* de dados dos *frames*.
- Exemplo de uso:
  1. Declaração de um *frame* CAN estendido.

```
uint8_t data[2] = {0, 10};
CANframe frm(1, 6, 2, data);
```

     Na primeira linha é declarada uma lista com os *bytes* de dados. Seguido, na linha abaixo, da declaração de um *frame* estendido com ID padrão igual a 1, extensão de ID igual a 6, e com dois *bytes* de dados (dlc_ = 2), sendo 0 e 10.

B.1.5 ***CANframe*** *(idstd, dlc_, data)*

```
CANframe(uint16_t idstd, uint8_t dlc_, uint8_t *data);
```

Função para criação de *frames* padrões, devem ser informados o ID padrão, o DLC, e de uma lista com os *bytes* dos dados.

- Parâmetros de entrada:
  - **idstd** variável de 2 *bytes* onde deve ser informado o valor do ID padrão do *frame*, no máximo 0x7FF.
  - **dlc_** número de *bytes* de dados.
  - **data** lista contendo os *bytes* de dados dos *frames*.
- Exemplo de uso:

1. Declaração de um *frame* CAN estendido.

```
uint8_t data[2] = {0, 10};
CANframe frm(1, 2, data);
```

Na primeira linha é declarada uma lista com os *bytes* de dados. Seguido, na linha abaixo, da declaração de um *frame* padrão com ID padrão igual a 1, e com dois *bytes* de dados (dlc_ = 2), sendo 0 e 10.

B.1.6 ***reload*** *(idstd, idext, dlc_, data)*

```
reload(uint16_t idstd, uint32_t idext, uint8_t dlc_, uint8_t *data);
```

Função para recarregar um *frame* qualquer com estendido, atribuindo os valores fornecidos com entrada nas variáveis correspondentes. Os parâmetros de entrada são os mesmos, e em mesma ordem, da função *CANframe(idstd, idext, dlc_, data)*, descrita um pouco acima.

- Exemplo de uso:

1. Declaração de um *frame* CAN estendido.

```
uint8_t data[2] = {0, 10};
CANframe frm(1, 2, data);
frm.reload(1, 10, 2, data);
```

As duas primeiras linhas deste exemplo já foram apresentadas como exemplo para a função *CANframe(idstd, dlc_, data)*, descrito anteriormente, nele é criado um *frame* padrão. Na última linha o *frame frm* é recarregado como estendido com a função *reload(..)*, a única diferença é a inserção do valor da extensão de ID (10).

B.1.7 ***reload*** *(idstd, dlc_, data)*

```
reload(uint16_t idstd, uint8_t dlc_, uint8_t *data);
```

Função para recarregar um *frame* qualquer com padrão, atribuindo os valores fornecidos com entrada nas variáveis correspondentes. Os parâmetros de entrada são os mesmos, e em mesma ordem, da função *CANframe(idstd, dlc_, data)*, descrita um pouco acima.

- Exemplo de uso:

    1. Declaração de um *frame* CAN estendido.

```
uint8_t data[2] = {0, 10};
CANframe frm(1, 10, 2, data);
frm.reload(1, 2, data);
```

    Na primeira linha é declarada uma lista com dois *bytes* de dados. Na segunda linha um *frame* estendido é declarado, tendo o ID padrão 1, a extensão de ID 10, o DLC 2, e os dados atribuídos a lista *data* na linha de cima. Na última linha o *frame frm* é recarregado com padrão, com a função *reload(..)*, a diferença é a ausência da extensão de ID.

B.1.8 ***reload*** *(dlc_, data_)*

```
reload(uint8_t dlc_, uint8_t *data_);
```

Função para recarregar o campo de dados de *frame* qualquer, essa função não altera os valores de ID, também pouco o tipo de *frame*, também é alterado o DLC. Essa função pode ser usada em um *loop*, onde os dados do *frame* são atualizados periodicamente, mas seus valores de ID não.

- Exemplo de uso:

    1. Declaração de um *frame* CAN estendido.

```
uint8_t data[4] = {0, 10, 10, 0};
frm.reload(4, data);
```

    Na primeira linha é declarada uma lista com quatro *bytes* de dados. Na segunda linha um *frame* anteriormente criado (*frm*) é recarregado com a lista de *bytes* de dados *data*.

B.2 Variáveis Públicas

B.2.1 SPI

- `unsigned long int SPI_speed = 10000000;`

Máxima frequência do *clock* do SPI na comunicação entre o Arduino e o MCP2515, que é limitada pelo último em 10 MHz. O valor praticado pelo Arduino é outro, definido em função do seu *clock* interno e do limite fornecido em *SPI_speed*.

- `uint8_t SPI_wMode = 0;`

  Modo de operação da comunicação SPI entre o Arduino e o MCP2515, o controlador CAN suporta dois modos o [0,0] (0) e o [1,1] (3).

- `uint8_t SPI_cs;`

  Número do pino do Arduino usado como *chip select* na comunicação SPI entre o Arduino e o MCP2515.

B.2.2 Configuração gerais do MCP2515

- `uint8_t crystalCLK = 8;`

  Frequência do oscilador que fornece a base de *clock* para o MCP2515, em MHz.

- `uint16_t bitF = 125;`

  Taxa de *bits* do barramento, em k *bits*/s.

- `uint8_t wMode = 0x00;`

  Está variável é usada para manipular o registro 0x0F do controlador CAN, 0x00 é modo de operação padrão da biblioteca, modo de operação *Normal*, encerra a solicitação para cancelar o envio de todas as transmissões, modo *One-Shot* desabilitado, pino *CLKOUT* do MCP25 desabilitado e setando o $F_{CLKOUT} = SystemClock/1$. Para outras opções consultar o *datasheet* do controlador CAN (Microchip Technology Inc, 2019).

- `uint8_t RXB0CTRL = 0x66;`

  Está variável é usada para manipular o registro 0x60 do MCP2515, ele configura o *buffer* de entrada RXB0. O valor padrão 0x66, desabilita as máscaras e filtros e ativa o *rollover* do *buffer* de entrada RXB0 para o RXB1.

- `uint8_t RXB1CTRL = 0x60;`

  Está variável é usada para manipular o registro de configuração do *buffer* de saída RXB1 do MCP2515 (0x70). O valor padrão 0x60, desabilita as máscaras e filtros.

- `uint8_t TXB0CTRL = 0x00;`

  Está variável manipula o registro 0x30, que corresponde a configuração do *buffer* de saída TXB0 do MCP2515. O valor padrão 0x00, atribui o nível mais baixo de prioridade (0) e aborta possível *frame* alocado buffer TXB0. Obs.: Quando todos os *buffers* de saída possuem a mesma prioridade o, o TXB0 se torna o menos prioritário.

- `uint8_t TXB1CTRL = 0x00;`

  Está variável manipula o registro 0x40, que corresponde a configuração do *buffer* de saída TXB1 do MCP2515. O valor padrão 0x00, atribui o nível mais baixo de prioridade (0) e aborta possível *frame* alocado buffer TXB0. Obs.: Quando todos os *buffers* de saída possuem a mesma prioridade o, o TXB1 se torna mais prioritário do que o TXB0 e menos prioritário do que o TXB2.

- `uint8_t TXB2CTRL = 0x00;`

  Está variável manipula o registro 0x50, que corresponde a configuração do *buffer* de saída TXB2 do MCP2515. O valor padrão 0x00, atribui o nível mais baixo de prioridade (0) e aborta possível *frame* alocado buffer TXB0. Obs.: Quando todos os *buffers* de saída possuem a mesma prioridade o, o TXB2 se torna o mais prioritário.

- `uint8_t CANINTE = 0xBF;`

  Está variável manipula o registro 0x2B do MCP2515. O valor padrão *0xBF* no registro 0x2B habilita interrupções, quando uma mensagem é recebida no RXB0 ou no RXB1, quando qualquer um dos *buffers* de saída (TXB0, TXB1 e TXB2) ficar vazio, quando ocorrer erros de múltiplas fontes indicadas pelo registro EFLG, e quando uma transmissão de mensagem for interrompida.

- `uint8_t BFPCTRL = 0x0F;`

Está variável manipula o registro 0x0C do MCP2515. o valor padrã *0x0F* no registro 0x0C configura os pinos do MCP2515, o RX0BF é habilitado e configurado como interrupção quando uma mensagem valida é carregado no RXB0, e de forma análoga o RX1BF é para o *buffer* RXB1.

- `uint8_t TXRTSCTRL = 0x00;`

  Está variável manipula o registro 0x0D do MCP2515. O valor padrão *0x00* não habilita nenhum interrupção com relação os pinos TX0RTS, TX1RTS e TX2RTS.

- `uint8_t CNF1 = 0x00;`

  Está variável salva o valor grava no registro 0x2A do MCP2515, ele faz parte da configuração do *Bit-Timing*. Há algumas opções pré definidas, estas podem ser configuradas atribuindo valores as variáveis *crystalCLK* e *bitF* 125 k *bit*/s, e dessa forma as variáveis *CNF1*, *CNF2* e *CNF3* são atualizadas durante a inicialização feita pela função *begin()* ou pela função de configuração específica, *confCAN()*. Os casos pré-definidos são: *crystalCLK* = 4 com duas possíveis taxas 125 k *bit*/s e 250 k *bit*/s. *crystalCLK* = 8 com três possíveis taxas 125 k *bit*/s, 250 k *bit*/s e 500 k *bit*/s. *crystalCLK* = 20 com quatro possíveis taxas 125 k *bit*/s, 250 k *bit*/s, 500 k *bit*/s e 1000 k *bit*/s. Para outros casos deve-se alterar *CNF1* diretamente, mais detalhes consultar o *datasheet* do MCP2515 (Microchip Technology Inc, 2019).

- `uint8_t CNF2 = 0x42;`

  Está variável salva o valor gravado no registro 0x29 do MCP2515. Sua configuração segue de forma análoga a *CNF1*.

- `uint8_t CNF3 = 0x02;`

  Está variável salva o valor grava no registro 0x28 do MCP2515. Sua configuração segue de forma análoga a *CNF1*.

B.2.3 Filtros e máscaras do MCP2515

- `uint16_t MASstd[2] = {0x00, 0x00};`

Lista com os valores dos ID padrões das máscaras 0 (MASstd[0]) e 1 (MASstd[1]) do MCP2515.

- `uint32_t MASext[2] = {0x00, 0x00};`

  Lista com os valores das estações de ID das máscaras 0 (MASext[0]) e 1 (MASext[1]) do MCP2515.

- `uint16_t FILstd[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};`

  Lista com os valores de ID padrão dos filtros do MCP2515, FILstd[0] corresponde ao filtro 0, e assim por diate, até FILstd[6] para o filtro 6.

- `uint32_t FILext[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};`

  Lista com os valores de extensão ID dos filtros do MCP2515, FILext[0] corresponde ao filtro 0, e assim por diate, até FILext[6] para o filtro 6.

B.2.4 Erros

- `String errLog = "no error";`

  Armazena o último erro genérico ocorrido. Pode-se atualizar o valor de *errLog* chamando a função *errCont()*.

- `String errMode = "Error Active";`

  Armazena o modo de confinamento de erro, "Error Active", "Error Passive"e "Bus-Off". Pode-se atualizar o valor de *errMode* chamando a função *errCont()*.

- `uint16_t RX0OVR = 0;`

  Armazena o número de *overload* no *buffer* RXB0. Pode-se atualizar o valor de *RX0OVR* chamando a função *errCont()*.

- `uint16_t RX1OVR = 0;`

  Armazena o número de *overload* no *buffer* RXB1. Pode-se atualizar o valor de *RX1OVR* chamando a função *errCont()*.

- `uint8_t multInt = 0;`

  Armazena o número de erros de multiplicas fontes detectados, indicados pelo registro *ERROR FLAG REGISTER* (0x2D). Pode-se atualizar o valor de *multInt* chamando a função *errCont()*.

- `uint8_t  MERRF = 0;`

  Armazena o número de erros de mensagens detectados. Pode-se atualizar o valor de *MERRF* chamando a função *errCont()*.

- `uint8_t  WAKIE;`

  Armazena o valor da *Wake-up flag* de MCP2515. Pode-se atualizar o valor de *WAKIE* chamando a função *errCont()*.

- `uint8_t TEC = 0;`

  Armazena o valor do contador de erros de transmissão TEC. Pode-se atualizar o valor de *TEC* chamando a função *errCont()*.

- `uint8_t REC = 0;`

  Armazena o valor do contador de erros de transmissão REC. Pode-se atualizar o valor de *REC* chamando a função *errCont()*.

B.2.5 Frames

- `CANframe frameRXB0;`

  Variável do tipo *frame*, criada para receber os *frames* recebidos no *buffer* RXB0. Pode-se atualizar o valor de *frameRXB0* chamando a função *readFrame()*.

- `CANframe frameRXB1;`

  Variável do tipo *frame*, criada para receber os *frames* recebidos no *buffer* RXB1. Pode-se atualizar o valor de *frameRXB1* chamando a função *readFrame()*.

B.3 Funções Públicas

B.3.1 Função: ***MCP2515*** *(spi_cs, spi_speed, spi_wMode)*

```
mcp.MCP2515(uint8_t spi_cs,
            unsigned long int spi_speed = 10000000,
            uint8_t spi_wMode = 0);
```

A função *MCP2515(...)* criar um objeto MCP2515, que herda as funções da biblioteca MCP2515. Todos os parâmetros de entrada da função *MCP2515(...)* são relacionados a comunicação SPI, e são listados abaixo, juntamente de dois exemplos de uso.

- Parâmetros de entrada:
  - **spi_cs** é o número do pino do Arduino usado como *chip select* do MCP2515.
  - **spi_speed** é a máxima frequência do *clock* da comunicação SPI, seu valor padrão é 10000000. Obs.: Esse é o valor máximo suportado pelo MCP2515 e serve apenas como limite superior, a frequência do *clock* é setada automaticamente pelo Arduino, dentro do limite informado.
  - **spi_wMode** indica o modo de operação do SPI, o MCP2515 suporta o modo [0,0] e o modo [1,1] (Microchip Technology Inc, 2019), que equivalem respectivamente, o 0 e 3 do Arduino (Arduino, Arduino). O valor padrão é 0.
- Exemplos de uso:
  1. Declaração de um objeto *MCP2515*.

     ```
     #include <MCP2515_1.h>
     MCP2515 mon(4);
     ```

     Neste exemplo foi, na primeira linha, incluído a versão 1 da biblioteca MCP2515, através do arquivo *MCP2515_1*. Na segunda linha foi declarado um objeto *MCP2515* chamado *mon*, que tem como *chip select* o pino digital 4 do Arduino.
  2. Outro exemplo de declaração de objeto *MCP2515*.

     ```
     MCP2515 mcp(7, 10000000, 3);
     ```

Desta vez foi declarado um objeto chamado *mcp*, que tem como *chip select* o pino digital 7 do Arduino, e usa o modo 3 do SPI do Arduino, que equivale [1, 1].

B.3.2 Função: ***begin*** *()*

```
mcp.begin();
```

A função *begin()* inicializa a comunicação SPI entre o Arduino e o MCP2515, e configura o controlador CAN para operar de acordo com os valores setados nas variáveis de configuração. As variáveis de configuração, bem como seus valores padrões, foram descritas na secção anterior B.2.

- Exemplos de uso:

  1. Parte inicial da função *void setup()* do CAN Mon.

     ```
     void setup() {
         Serial.begin(9600);
         mon.bitF = 125; // k bits/s
         mon.begin();
     ```

     Estas são as quatro primeiras linhas da função *void setup()* do nodo CAN Mon (*CANMon.ino*). Na penúltima linha é setado a taxa de *bits* para 125 k *bits*/s, **bitF** é a variável de controle que armazena o valor da taxa *bits*. Na última o MCP2515 é inicializado e configurado com a função *mon.begin()*.

  2. Configuração padrão.

     ```
     sensor.begin();
     ```

     Neste exemplo o MCP2515 é inicializado e configurado no modo padrão, *sensor* é o nome do objeto.

B.3.3 Função: ***reset*** *()*

```
mcp.reset();
```

A função **reset()** não possui argumentos, ela reinicia o CI MCP2515 enviando a instrução **0xC0** pela comunicação SPI. Atenção ao voltar do *reset* o CI se encontra em modo

de configuração e com valores padrão nos registros, e deve-se esperar algo entorno de 2 µs antes de usa-lo (Microchip Technology Inc, 2019), isso pode ser feito através da função **delayMicroseconds()** do Arduino (Arduino, 2020b). É aconselhável o uso dessa função logo após a inicialização do CI, e antes de configurá-lo, pois assim tem-se certeza dos valores salvos em seus registros.

B.3.4 Função: ***read*** *(REG, data, n = 1)*

```
mcp.read(uint8_t REG, uint8_t *data, uint8_t n = 1);
```

A função **read(...)** realiza **n** (de 1 à 128) leituras sequencias de registros do MCP2515, a partir do registro **REG** informado, e aloca os **n** *bytes* em uma lista previamente criada e indicada pelo endereço ***data**, a lista data deve ter o tamanho de **n** *bytes*. Abaixo segue a descrição dos parâmetros de entradas da função **read(...)**, e na sequencia dois exemplos, um lendo um registro e o outro lendo 13 registros.

- Parâmetros de entrada:
  - **REG** é o endereço do primeiro registro do MCP2515 a ser lido, deve ter um tamanho de 1 *byte* e seu valor vai 0x0 (0) até 0x80 (128).
  - ***data** é o endereço da lista criada para armazenamento dos valores lidos, 1 B para cada registro lido. A lista **data** deve ter o tamanho de **n** *bytes*.
  - **n** é o número de registros a serem lidos, contando a partir do **REG**, por padrão **n**= 1, logo se não alterado a função **read(..)** lerá apenas um registro.
- Exemplos de uso:
  1. Leitura do registro TEC do MCP2515.

     ```
     uint8_t data[1];
     mcp.read(0x1C, data);
     ```

     Neste exemplo foi primeiro declaro uma lista (**data[1]**) com 1 *byte* e na sequencia é realizado a leitura do registro **0x1C**, que armazena a contagem de erro de transmissão (TEC) do MCP2515.
  2. Leitura do *buffer* de entrada RXB1 do MCP2515.

```
uint8_t data[13];
mcp.read(0x61, data, 2);
```

Neste exemplo foi primeiro declaro uma lista com 13 *bytes*, e na sequencia é realizado a leitura dos 13 registros do *buffer* de entrada RB1 do MCP2515.

B.3.5 Função: ***regCheck*** *(REG, VAL, extraMask = 0xFF)*

```
mcp.regCheck(uint8_t REG, uint8_t VAL, uint8_t extraMask = 0xFF);
```

A função **regCheck(...)** confere se o *byte* salvo no registro **REG** é igual ao *byte* **VAL**, se for a função retorna 0, se não ela retorna 1. Se não for possível escrever no registro **REG** a função retorna 2. Pode-se usar essa função também para verificar o valor de um ou mais *bits* do *byte* armazenado em **REG**, para isso deve-se informar a mascara (**extraMask**) capaz de selecionar os *bits* desejados pela operação lógica & (*and*). Essa função pode ser chamada em conjunto com a função **write(...)**, de modo que seja feita uma verificação do sucesso da escrita no registro. A seguir a descrição dos parâmetros de entrada da função **regCheck(...)** e dois exemplos de uso, sem e com mascara de seleção de *bits*.

- Parâmetros de entrada:
  - **REG** é o endereço do registro do MCP2515 a ser verificado. Deve ter um tamanho de 1 *byte* e seu valor vai 0x0 (0) até 0x80 (128).
  - **VAL** é o *byte* que se deseja verificar em **REG**. Deve ter o tamanho de 1 *byte*.
  - **extraMask** é uma mascara de 1 *byte*, para selecionar os bits desejados pela operação logica & (and). Por padrão **extraMask** = 0xFF, logo por padrão a verificação é feita sobre todos os *bits*.
- Exemplos de uso:
  1. Verificação do registro TEC do MCP2515.

```
uint8_t check = 0;
check = mcp.regCheck(0x1C, 0);
if(check =! 0){
Serial.println("TEC > 0");
}
```

Neste exemplo primeiro é declaro a variável check para armazenar o retorno da função **regCheck(...)**. Depois é chamada a função **regCheck(0x1C, 0)**, que verificará se o valor salvo no registro **0x1C** é 0. O registro **0x1C** armazena a contagem dos erros de transmissão (TEC) do MCP2515. O valor retornado pela função **regCheck(...)** e salvo na variável check é comparado com 0, se ele for diferente de zero, significa que TEC possui um valor maior que zero, e ao menos um erro de transmissão foi detectado.

2. Verificando se o MCP2515 está em modo de configuração.

```
uint8_t check = 0;
check = mcp.regCheck(0x0F, 0x80, 0xE0);
if(check == 0){
    //realizar as configuração desejadas aqui.
}
```

Dessa vez a função **regCheck(...)** é usado com mascara, para verificar se o MCP2515 está no modo de configuração. O código do modo de operação é armazenado nos três *bits* mais significantes do registro 0x0F, por isso o uso da mascara 0xE0. Se o MCP2515 estiver no modo de configuração o valor do registro 0x0F, após aplicação da mascara 0xE0, deve ser 0x80.

*B.3.6 Função: **write** (REG, VAL, CHECK = 1)*

```
mcp.write(uint8_t REG, uint8_t VAL, uint8_t CHECK = 1);
```

A função **write(...)** escreve o *byte* **VAL** no registro **REG** e verifica se o valor foi realmente escrito. O sucesso pode ser verificado na variável **errLog**, no caso de falha ela valerá "Erro na escrita", se seu valor for "Reg invalido", o registro **REG** não pode ser escrito. Obs.: Há alguns registros que só podem ser escritos se o MCP2515 estiver no modo de configuração. A checagem da escrita é opcional, e por padrão é feita, para que ela não seja feita basta fornecer 0 para **check**. Desabilitar a checagem fará com que a rotina **write(...)** execute em menos tempo, visto que não será feita a checagem. A seguir a descrição dos parâmetros de entrada da função **write(...)**, e dois exemplos de utilização, com e sem verificação de escrita.

- Parâmetros de entrada:

- **REG** é o endereço do registro do MCP2515 para escrita. Deve ter um tamanho de 1 *byte* e seu valor vai 0x0 (0) até 0x80 (128).
- **VAL** é o *byte* que se deseja escrever em **REG**. Deve ter o tamanho de 1 *byte*.
- **check** é a sinalização de checagem, se fornecido 1 para **check** a função **write(...)** realizará a checagem, se fornecido 0 a função **write(...)** não realizará a checagem. Por padrão a checagem é feita.

- Exemplos de uso:

1. Escrevendo no registro 0x36 com verificação automática.

```
mcp.write(0x36, 0xFE);
if(errLog=="Erro na escrita"){
    Serial.println(mcp.errLog)
}
}
```

Neste exemplo o *byte* 0xFE é escrito no registro 0x36. Por padrão a função **write(...)** verifica o sucesso da escrita, o que é feito comparando o valor da variável **errLog**. O registro 0x36 pertence ao *buffer* de saída TXB1 do MCP2515.

2. Escrevendo no registro 0x36 sem verificação.

```
 mcp.write(0x36, 0xFE, 0);
}
```

Desta vez a checagem é desabilitada informando o valor 0 para o parâmetro **check**.

B.3.7 Função: ***bitModify*** *(REG, MASK, VAL, check = 0)*

```
mcp.bitModify(uint8_t REG, uint8_t MASK, uint8_t VAL, uint8_t check = 0);
```

A função **bitModify(...)** é usada para alterar valores de bits específicos no registro **REG** do MCP2515. Para tal deve-se fornecer um mascara (MASK, 1 *byte*) capaz de selecionar os *bits* que se deseja modificar pela operação lógica & (and).

- Parâmetros de entrada:

- **REG** é o endereço do registro do MCP2515, onde se quer alterar valores de *bits*. Deve ter um tamanho de 1 *byte* e seu valor vai 0x0 (0) até 0x80 (128).
- **MASK** é a mascara para selecionar a posições de *bits* que deseja-se modificar. O *bits* são selecionados pela operação lógica é a & (and). Deve ter 1 *byte* de tamanho.
- **VAL** é o *byte* que contém o valor dos *bits* a serem escritos no registro **REG**. Deve ter 1 *byte* de tamanho.
- **check** é a sinalização de checagem, para está função o padrão é não checar e por isso **check**= 0. Caso queira verificar o sucesso da alteração dos *bits* fornece 1 para **check**, a habilitação da checagem prolonga o tempo de execução.

- Exemplos de uso:

1. Setar o modo de operação *Listen-Only*

```
mcp.bitModify(0x0F, 0xE0, 0x60);
```

   Neste exemplo o MCP2515 é setado no modo *Listen-Only*, alterando o valor dos três bits mais significantes do registro 0x0F, com a mascara 0xE0, para 0 1 1.

2. Setar o modo de operação *Sleep*

```
mcp.bitModify(0x0F, 0xE0, 0x20, 1);
```

   Desta vez o MCP2515 é setado para o modo *Sleep*, alterando os três *bits* mais significantes do registro 0x0F para 0 1 0. Além disso é feita a verificação fornecendo 1 para o parâmetro **check**.

B.3.8 Função: ***confMode*** *()*

```
mcp.confMode();
```

A função **confMode()** atua sobre o registro 0x0F do MCP2515, nele pode configurar o modo de operação do controlador CAN e outras funções, para mais detalhes consulte o registro CANCTRL no *datasheet* (Microchip Technology Inc, 2019). A variável *wMode* guarda o valor a ser gravado no registro CANCRL, se este valor já estiver configurado, e a função for chamada, o MCP2515 é setado para o modo de configuração, no qual se pode

alterar registros 'protegidos'. Se o MCP2515 estiver no modo de configuração, a função retorna ao modo *wMode*.

- Exemplo de uso:

  1. Configurando o MCP2515 para *Listen-Only*.

```
mcp.wMode = 0x60;
mcp.confMode();
```

     Na primeira linha é atribuído o valor 0x60 na variável de controle *wMode*, este valor é escrito no registro 0x0F do MCP2515, setando o controlador para modo *Listen-Only*. O valor padrão de *wMode* é 0x00, modo *Normal* do MCP2515.

B.3.9 Função: ***confRX*** *()*

```
mcp.confRX();
```

Esta função manipula os registros 0x60 e 0x70 do MCP2515, estes valores são definidos pelas variáveis de controle RXB0CTRL e RXB1CTRL, por padrão e respectivamente, 0x66 e 0x60.

- Exemplo de uso:

  1. Habilitação dos filtros de aceite do *buffer* RXB1.

```
mcp.RXB1CTRL = 0x00;
mcp.confRX();
```

     Neste exemplo é setado o valor 0x00 no registro 0x70 (RXB1CTRL) do MCP2515, este valor habilita os filtros e máscaras no *buffer* de saída RXB1 do MCP2515.

B.3.10 Função: ***confTX*** *()*

```
mcp.confTX();
```

A função **confTX()** manipula os registros 0x30, 0x40 e 0x50, neles são configurados os *buffers* de saída do MCP2515, respectivamente, TXB0, TXB1 e TXB2.

- Exemplo de uso:

1. Configurando a prioridade interna dos *buffers* de saída do MCP2515.

```
mcp.TXB0CTRL = 0x03
mcp.TXB1CTRL = 0x00
mcp.TXB2CTRL = 0x01
mcp.confTX();
```

Neste exemplo o TXB0 é configurado com a mais alta prioridade, enquanto o TXB1 e configurado no nível mais baixo, e por fim o TXB2 é setado no nível 1 de prioridade. Existem quatro níveis de prioridade.

B.3.11 Função: ***confINT*** *()*

```
mcp.confINT();
```

A função **confINT()** configura os registros 0x2B, 0x0C e 0x0D do MCP2515, através das variáveis de controle e respectivamente, *CANINTE*, *BFPCTRL* e *TXRTSCTRL*. Outros valores podem ser consultados no *datasheet* do MCP2515 (Microchip Technology Inc, 2019).

- Exemplo de uso:

1. Configurações padrões.

```
mcp.CANINTE = 0xBF;
mcp.BFPCTRL = 0x0F;
mcp.TXRTSCTRL = 0x00;
mcp.confINT();
```

Neste exemplo a função *confINT()* é chamada na ultima linha, após a atribuição das três variáveis de controle correspondentes aos registros manipulados pela a função. Mas detalhes sobre as variáveis na subseção B.2.2.

B.3.12 Função: ***confFM*** *()*

```
mcp.confFM();
```

A função **confFM()** configura os valores das máscaras e filtros de ID, padrão e extensão, nos *buffers* de entrada do MCP2515.

- Exemplo de uso:

1. Configurações de filtros e máscaras.

```
mcp.MASstd[0] = {0x400};
mcp.MASstd[1] = {0x400};
mcp.FILstd[0] = {0x7FF};
mcp.FILstd[1] = {0x7FF};
mcp.FILstd[2] = {0x7FF};
mcp.FILstd[3] = {0x7FF};
mcp.FILstd[4] = {0x7FF};
mcp.FILstd[5] = {0x7FF};
mcp.FILstd[6] = {0x7FF};
mcp.confFM();
```

Neste exemplo as máscaras e filtros são setados através da atribuição das variáveis de configuração relacionadas, no caso especifico o MCP2515 só aceitara *frames* padrão, cujo valor do ID padrão seja maior que 0x400. Mais detalhes do funcionamento dos filtros e máscaras do controlador CAN podem ser vistas no *datasheet* (Microchip Technology Inc, 2019).

B.3.13 Função: ***confCAN*** *()*

```
mcp.confCAN();
```

A função **confCAN()** configura a taxa de operação do CI MCP2515, para isso ela manipula os registros relacionados ao *Bit-Timing* do controlador CAN.

- Exemplo de uso:

1. Configurações padrões.

```
mcp.crystalCLK = 8; // M Hz
mcp.bitF = 125; // k bit/s
mcp.confCAN();
```

Na primeira linha é atribuído o valor de 8 para a variável de configuração *crystalCLK*, deve ser informado nesta variável o valor do *clock* fornecido ao controlador CAN MCP2515. Também é atribuído, na segunda linha, o valor da

variável de configuração *bitF*, que contém a taxa de transferência de *bits* utilizada no barramento CAN. Na ultima linha é chamada a função *confCAN()*, que realiza a configuração dos registros 0x2A, 0x29 e 0x28, os valores gravados podem ser consultados respectivamente, após a função *confCAN()*, nas variáveis de configuração CNF1, CNF2 e CNF3.

B.3.14 Função: ***status*** *(status)*

```
mcp.status(uint8_t *status);
```

A função **status(...)** realiza a leitura do *byte* de *status* do MCP2515 e salva no endereço indicado por ***status**, que deve ter o tamanho de um *byte*. O valor retornado pela função **status(...)** indica o status de funções de envio e recebimento.

- Parâmetros de entrada:
  - ***status** é o ponteiro que indica a posição a ser salvo o *byte* lido pela função **status(...)**. Aconselha-se usar uma lista de uma posição, como no exemplo.
- Exemplo de uso:
  1. Leitura e impressão serial do *status* do MCP2515.

```
uint8_t status[1];
mcp.status(status);
Serial.println(status[0]);
```

B.3.15 Função: ***errCont*** *()*

```
mcp.errCont();
```

A função **errCont()** realiza a leitura dos registros relacionados a contagem de erro, TEC (0x1C) e REC (0x1D), e verifica a ocorrência de *overflow* nos *buffers* de entrada, lendo o registro de sinalização de erros (0x2D). As leituras são atualizadas nas variáveis, *TEC*, *REC*, *RX0OVR*, *RX1OVR*, *MERRF*, *WAKIE*, *multInt* e *errMode*. Uma descrição destas variáveis está disponível na subseção B.2.

- Exemplo de uso:

1. Contagem de erros.

```
mcp.errCont();
Serial.println(mcp.TEC);
Serial.println(mcp.errMode);
```

Neste exemplo é consultado o valor do contador de erros de transmissão do MCP2515 (TEC), e o modo do confinamento por erro. A execução da função *errCont()*, atualiza as variáveis dos contadores de erros.

B.3.16 Função: ***writeID*** *(sid, id_ef = 0, eid = 0, txb = 0, timeOut = 10, check = 0)*

```
writeID(uint16_t sid, uint8_t id_ef = 0, uint32_t eid = 0,
        uint8_t txb = 0, uint8_t timeOut = 10, uint8_t check = 0);
```

A função **writeID(...)** realiza a escrita do identificador da mensagem **ID** no *buffer* de saída indicado por **txb**. No caso de haver mensagem pendente de transmissão, a função não irá sobrepor o ID da mensagem pendente, ela tentará repetidamente até conseguir escrever o ID ou até esgotar o tempo **timeOut** fornecido em ms.

- Parâmetros de entrada:
  - **sid** é uma variável de dois *bytes*, nela é informado o valor do ID padrão. O maior valor possível é 0x3FF.
  - **id_ef** é a *flag* de extensão de ID, e deve ser informado 0 para ID padrão e 1 para ID com extensão. Por padrão essa variável vale 0.
  - **eid** é uma *word* de 32 *bits* que contém a extensão identificação (ID) da mensagem a ser enviada. Por padrão essa variável vale 0. O maior valor possível é 0x3FFFF.
  - **txb** indica o *buffer* no qual pretende-se escrever o ID, se fornecido 0 o ID é escrito no TXB0, se 1 é escrito no TXB1, se 2 é escrito no TXB2, se for fornecido 3 o mesmo ID é escrito nos três *buffers* de saída. Por padrão essa variável vale 0.
  - **timeOut** é o tempo (em ms) limite para que a função consiga escrever no ID. Se o *buffer* escolhido permanecer ocupado durante todo o **timeOut** a função retornará sem escrever o ID. É esperado por padrão 10 ms

- **check** é a sinalização de checagem de escrita, quando fornecido 1 a checagem é feita, se fornecido 0 ela não é feita. Por padrão a checagem é feita.

- Exemplos de uso:

1. Setando um ID padrão e uma extensão de ID no *buffer* de saída TXB0.

```
uint16_t IDstd = 0x700;
uint32_t IDext = 0x30000;
mcp.writeID(IDstd, 1, IDext);
```

   Neste exemplo o ID padrão 0x700 e a extensão de ID 0x30000 são escritos no *buffer* de saída TXB0, com os valores padrões de *time out* e checagem.

2. Escrevendo o ID padrão 10 em um *buffer* de saída. *time out* desabilitado.

```
mcp.writeID(10, 0, 0, 1, 0);
```

   Desta vez, o ID padrão escrito no *buffer* de saída TXB1 o valor 10, sem extensão de ID e sem *time out*.

B.3.17 Função: ***loadTX*** *(data, n = 8, abc = 1)*

```
mcp.loadTX(uint8_t *data, uint8_t n = 8,  uint8_t abc = 1);
```

A função **loadTX(...)** realiza a escrita sequencial dos **n** *bytes* armazenados na lista indicada por ***data**, nos **n** registros sequenciais dos *buffers* de saída do MCP2515, começando pelo registro indicado pelo código **abc** (de 0 à 5). Se fornecido 0 para **abc** a escrita começa do registro 0x31, se fornecido 1 começa do 0x36, se se fornecido 2 começa do 0x41, se se fornecido 3 começa do 0x46, se **abc** = 4 a começa do 0x51 e se **abc** = 5 a começa do 0x56.

- Parâmetros de entrada:

  - ***data** é o endereço da lista que armazena os **n** *bytes* a serem escritos nos *buffers*.
  - **n** é o número de *bytes* a serem escritos.
  - **abc** é o código que marca o registro de partida da escrita sequencial.

- Exemplo de uso:

1. Escrita sequencial com a função **loadTX(...)**.

```
uint8_t data = {0, 2, 1, 3, 4, 5, 6, 7};
mcp.loadTX(data, 8, 1);
```

Neste exemplo a sequencia de 8 *bytes* armazenada na lista **data** é sequencialmente escrita nos registros que armazenam os dados da mensagem a ser enviada pelo *buffer* TXB0 (do 0x36 até o 0xD).

B.3.18 Função: ***send*** *(txBuff = 0x01)*

```
mcp.send(uint8_t txBuff = 0x01);
```

A função **send(...)** realiza o pedido de transmissão da mensagem armazenada no *buffer* ou *buffers* indicados por **txBuff**. Se **txBuff** = 1, será pedido o envio da mensagem salva no *buffer* TXB0, se **txBuff** = 2 o pedido será para o TXB1, se **txBuff** = 3 o pedido será para os *buffers* TXB0 e TXB1, se **txBuff** = 4 o pedido será para o TXB2, se **txBuff** = 5 o pedido será para os *buffers* TXB0 e TXB2, se **txBuff** = 6 o pedido será para os *buffers* TXB1 e TXB2, e se **txBuff** = 7 o pedido será para os todos os *buffers* de saída. Quando for solicitado o envio de mais de uma mensagem, será primeiro enviado a armazenada no *buffer* mais prioritário.

- Parâmetros de entrada:
  - **txBuff** é o código de pedido de transmissão, seu valor vai de 1 a 7.
- Exemplo de uso:
  1. Solicitação de envio da mensagem salva no *buffer* de transmissão TXB2.

```
mcp.send(0x04);
```

B.3.19 Função: ***writeFrame*** *(frameToSend, txb_ = 0, timeOut = 10, check = 0)*

```
writeFrame(CANframe frameToSend, uint8_t txb_ = 0,
           uint8_t timeOut = 10, uint8_t check = 0)
```

A função **writeFrame(...)** realiza a escrita de um *frame*, indicado por *frameToSend*, no *buffer* informado por **txb_** (0, 1 ou 2). A função **writeFrame(...)** tentará escrever repetidamente até conseguir, ou até que acaba o tempo limite de execução indicado em **timeOut**.

- Parâmetros de entrada:
  - **frameToSend** é a mensagem a ser enviada. Essa variável contém todas informações do *frame*, ver subseção B.1 para mais detalhes.
  - **txb_** indica o *buffer* de saída no qual pretende escrever os dados. Para TXB0 deve ser fornecido 0, para TXB1 1 e para TXB2 2.
  - **timeOut** é o tempo máximo de execução da função, a **writeFrame(...)** tentará escrever no *buffer* escolhido até que este esteja desocupada e possa receber a nova mensagem, ou até que acabe o tempo de execução máximo (*time out*) fornecido. **timeOut** é expresso em ms é vale, por padrão, 10 ms.
  - **check** é a sinalização de checagem de escrita, se for fornecido 1 ela será feita, se for fornecido 0 não. A checagem custa tempo de execução, que não é incrementado no **timeOut**, e por padrão ela é feita.
- Exemplos de uso:
  1. Envio de um *frame* com os *bytes* de uma leitura analógica.

```
uint16_t ID_std = random(0x7FF);
uint32_t ID_ext = random(0x3FFFF);

UNION_t Data;
Data.INT = analogRead(A0);

CANframe frm(ID_std, sizeof(Data), Data.bytes);

mcp.writeFrame(frm);
```

  2. Envio de um *frame* com os *bytes* de um variável do tipo *float* pelo *buffer* de saída TXB2.

```
UNION_t Data;
Data.FLT = analogRead(A0)*(5.0 / 1023);

frm.reload(sizeof(Data.FLT), Data.bytes);
```

```
mcp.writeFrame(frm, 2);
```

Neste exemplo o *frame* (*frm*) foi apenas atualizado com um novo dado, dessa forma os valores de ID, não são alterados.

B.3.20 Função: ***abort*** *(abortCode = 7)*

```
mcp.abort(uint8_t abortCode = 7);
```

A função **abort(...)** cancela o envio de *frames*.

- Parâmetros de entrada:
  - **abortCode** é o código de cancelamento de envio, seu valor pode ser 1, 2, 4 ou 7, e por padrão é 7 que aborta todos os envios pendentes nos *buffer* de saída.
- Exemplos de uso:
  1. Cancelamento do envio da mensagem salva no *buffer* de saída TXB2.

     ```
     mcp.abort(4);
     ```

  2. Cancelamento do envio de todas mensagem salvas nos *buffers* de saída.

     ```
     mcp.abort();
     ```

B.3.21 Função: ***readID*** *(id, rxb = 0)*

```
mcp.readID(uint8_t *id, uint8_t rxb = 0);
```

A função **readID(...)** faz a leitura do ID padrão, e caso haja da extensão, de uma mensagem armazenada no *buffer* de entrada **rxb**. Os 5 *bytes* do ID lidos, dois para o ID padrão e três para extensão, são salvos *byte* a *byte* na lista indicada pelo endereço ***id**. A função **readID(...)** é capaz de ler tanto o ID padrão quanto o estendido, e a princípio não é sabido o formato do ID, por isso deve-se sempre reservar uma lista de 5 *bytes*.

- Parâmetros de entrada:
  - ***id** é o endereço da lista onde serão armazenados os *bytes* do ID. Deve ser um lista de 5 *bytes*.

– **rxb** indica o *buffer* a ser lido, se fornecido 0 é lido o RXB0, se fornecido 1 é lido RXB1.

- Exemplo de uso:

1. Leitura, reconstrução e impressão do ID padrão e da extensão salvas no RXB1.

```
uint8_t ID[5];
mcp.readID(ID, 1);

IDunion_t IDstd, IDext;

IDstd.bytes[0] = ID[0];
IDstd.bytes[1] = ID[1];

IDext.bytes[0] = ID[2];
IDext.bytes[1] = ID[3];
IDext.bytes[2] = ID[4];
IDext.bytes[3] = 0;

Serial.print(IDstd);
Serial.print(' ');
Serial.println(IDext);
```

B.3.22 Função: ***readFrame*** *()*

```
mcp.readFrame()
```

A função **readFrame(...)** realiza a leitura dos *buffers* de entrada RXB0 e RXB1, e caso haja mensagem valida a salva nos *frames* auto declarados *frameRXB0* e *frameRXB1*, respectivamente. Os frames são declarados automaticamente na criação de um objeto MCP2515. Antes da leitura a função verifica se ha mensagem valida em um dos *buffers* de entrada, e caso haja realiza a leitura da mensagem, se não houver a função **readFrame(...)** atualiza o valor *type* do *frame* respectivo para "No frame". Podem haver mensagens validas nos dois *buffers* e neste caso a função **readFrame(...)** atualizará ambos *frames*.

- Exemplo de uso:

1. Leitura e impressão de mensagens no *buffer* de entrada RXB0 do MCP2515.

```
mcp.readFrame();

if(frameRXB0.type != "No frame"){
 Serial.println("Frame recebido pelo RXB0");
 Serial.print(frameRXB0.id_std);
 Serial.print(' ');
 Serial.print(frameRXB0.id_ext);
 Serial.print(' ');
 Serial.print(frameRXB0.dlc);

 for(int i=0; i < frameRXB0.dlc; i++){
     Serial.print(' ');
     Serial.print(frameRXB0.data[i]);
 }
 Serial.println();
}
```

O processo feito para impressão do RXB0, a partir do *if(...)*, pode ser feito para o RXB1, trocando o *frameRXB0* por *frameRXB1*.

B.3.23 Função: ***digaOi*** *(oi)*

```
mcp.digaOi(char *oi);
```

A função **digaOi(...)** pode imprimir diversos parâmetros internos da biblioteca, inclusive aqueles relacionados a configurações do MCP2515. Se o parâmetro de entrada não for informado a função *digaOi(...)*, imprimira todas as 40 variáveis descritas na seção B.2.

- Parâmetros de entrada:
  - **oi** é o *char* que informa quais variáveis serão impressas, se informado "status", será impresso somente as variáveis relacionadas ao estado do MCP2515 e seus *buffers* de saída e entrada. Se informado "eros"ou "errors", serão impressos valores de erros. Se informado "SPI", serão impressos os parâmetros do SPI. Se informado "CAN", serão impressos os parâmetros da configuração da CAN. Se informado "controle", serão impressos os parâmetros de controle do MCP2515. Se informado "filtros", serão impressos os filtros setados no MCP2515. Se informado "filtros", serão impressos as máscaras setados no MCP2515.
- Exemplo de uso da função **digaOi(...)** sem informar parâmetro de entrada.

  ```
  Serial.begin(9600);
  mcp.digaOi();
  ```

  Neste exemplo a função **digaOI(...)** imprimira todas as variáveis.
- Exemplo de uso da função **digaOi(...)** imprimindo somente os parâmetros relacionados a erros.

  ```
  Serial.begin(9600);
  mcp.digaOi("error");
  ```

B.4 Códigos e exemplos

B.4.1 MCP2515.h

```
/* Biblioteca para o CI MCP2515, datasheet disponivel em:
 * http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf
 *
*/
#ifndef MCP2515_1_h
#define MCP2515_1_h
#include <Arduino.h>
#include <SPI.h>

typedef union {
 uint16_t u16;
 uint32_t u32;
 uint8_t bytes[4];
} IDunion_t;

const String werr = "Reg writing error";

struct CANframe{
```

```
  uint16_t id_std = 0;
  uint32_t id_ext = 0;
  uint8_t dlc = 0;
  uint8_t data[8] = {0,0,0,0,0,0,0,0};
  uint8_t bts[14] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0};
  String type = "Unknown";

 CANframe();

 CANframe(uint8_t *frameBytes, uint8_t extFlag = 0);
 void reload(uint8_t *frameBytes, uint8_t extFlag = 0);

 CANframe(uint16_t idstd, uint32_t idext, uint8_t dlc_, uint8_t *data);
 void reload(uint16_t idstd, uint32_t idext, uint8_t dlc_, uint8_t *data);

 CANframe(uint16_t idstd, uint8_t dlc_, uint8_t *data);
 void reload(uint16_t idstd, uint8_t dlc_, uint8_t *data);

 void reload(uint8_t dlc_, uint8_t *data_);

};
```

```
class MCP2515{

private:
  void start();
  void end();

public:

   //SPI valores conf.
   unsigned long int SPI_speed = 10000000;
   uint8_t SPI_wMode = 0;
   uint8_t SPI_cs;

   //Valores padrões de configuração do MCP2515
   uint8_t crystalCLK = 8;
   uint16_t bitF = 125; // k Hertz
   uint8_t wMode = 0x00;

   uint8_t RXB0CTRL = 0x66;//whit RollOver to RXB1
   uint8_t RXB1CTRL = 0x60;
```

```
uint8_t TXB0CTRL = 0x00;
uint8_t TXB1CTRL = 0x00;
uint8_t TXB2CTRL = 0x00;
uint8_t CANINTE = 0xBF;
uint8_t BFPCTRL = 0x0F;
uint8_t TXRTSCTRL = 0x00;

uint8_t CNF1 = 0x00;
uint8_t CNF2 = 0x42;
uint8_t CNF3 = 0x02;

uint16_t MASstd[2] = {0x00, 0x00};//{0x400, 0x400};
uint32_t MASext[2] = {0x00, 0x00};

uint16_t FILstd[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};//0x7FF
uint32_t FILext[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};//0x3FFFF

//Error Loggers
String errLog = "no error";
String errMode = "Error Active";
uint16_t RX0OVR = 0;
```

```
uint16_t RX1OVR = 0;
uint8_t multInt = 0;
uint8_t  MERRF = 0;
uint8_t  WAKIE = 0;
uint8_t TEC = 0;
uint8_t REC = 0;

// frames
CANframe frameRXB0;
CANframe frameRXB1;

//funções publicas
MCP2515(uint8_t spi_cs, unsigned long int spi_speed = 10000000, uint8_t spi_wMode = 0); //
void begin();
void reset();
void read(uint8_t reg, uint8_t *data, uint8_t n = 1);
uint8_t regCheck(uint8_t reg, uint8_t val, uint8_t extraMask = 0xFF);
void write(uint8_t reg, uint8_t val, uint8_t check = 0);
void bitModify(uint8_t reg, uint8_t mask, uint8_t val, uint8_t check = 0);
void confMode();
void confRX();
```

```
    void confTX();
    void confINT();
    void confCAN();
    void confFM();
    void status(uint8_t *status);
    void RXstatus(uint8_t *status);
    void errCount();
    void writeID(uint16_t sid, uint8_t id_ef = 0, uint32_t eid = 0, uint8_t txb = 0,
                 uint8_t timeOut = 10, uint8_t check = 0);
    void loadTX(uint8_t *data, uint8_t n = 8,  uint8_t abc = 1);
    void send(uint8_t txBuff = 0x01);
    void writeFrame(CANframe frameToSend, uint8_t txb = 0, uint8_t timeOut = 10, uint8_t check = 0);
    void abort(uint8_t abortCode = 7);
    void readID(uint8_t *id, uint8_t rxb = 0);
    void readFrame();
    void quickread();
    void digaOi(); //char S = "Ola"
    void digaOi(char *oi);
};
#endif
```

B.4.2 MCP2515.cpp

```
/* Biblioteca para o CI MCP2515, datasheet:
 * http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-
   CAN-Controller-with-SPI-20001801J.pdf
 * Doc.
*/

#include "MCP2515_1.h"

//Data Frame functions

CANframe::CANframe(uint8_t *frameBytes, uint8_t extFlag = 0){
  IDunion_t std_id;
  IDunion_t ext_id;

  std_id.bytes[0] = frameBytes[1];
  std_id.bytes[1] = frameBytes[0];
  id_std = std_id.u16;

  if(extFlag == 1){
    ext_id.bytes[0] = frameBytes[4];
```

```
    ext_id.bytes[1] = frameBytes[3];
    ext_id.bytes[2] = frameBytes[2];
    ext_id.bytes[3] = 0;
    id_ext = ext_id.u32;
    type = F("Ext. Data");
  }else{
    ext_id.bytes[0] = 0;
    ext_id.bytes[1] = 0;
    ext_id.bytes[2] = 0;
    ext_id.bytes[3] = 0;
    id_ext = ext_id.u32;
    type = F("Std. Data");
  }

  dlc = frameBytes[5];

  for (uint8_t i = 0; i < dlc + 6; i++){
    bts[i] = frameBytes[i];
    if(i > 5){
      data[i - 6] = frameBytes[i];
    }
```

```
  }

  return;
}
void CANframe::reload(uint8_t *frameBytes, uint8_t extFlag = 0){

  dlc = frameBytes[5];

  for (uint8_t i = 0; i < dlc + 6; i++){

    bts[i] = frameBytes[i];
    if(i > 5){
      data[i - 6] = frameBytes[i];
    }
  }

  IDunion_t std_id;
  IDunion_t ext_id;

  std_id.bytes[0] = frameBytes[1];
  std_id.bytes[1] = frameBytes[0];
```

```
  id_std = std_id.u16;

  if(extFlag == 1){
    ext_id.bytes[0] = frameBytes[4];
    ext_id.bytes[1] = frameBytes[3];
    ext_id.bytes[2] = frameBytes[2];
    ext_id.bytes[3] = 0;
    id_ext = ext_id.u32;
    type = F("Ext. Data");
  }else{
    ext_id.bytes[0] = 0;
    ext_id.bytes[1] = 0;
    ext_id.bytes[2] = 0;
    ext_id.bytes[3] = 0;
    id_ext = ext_id.u32;
    type = F("Std. Data");
  }

  return;
}
```

```
CANframe::CANframe(uint16_t idstd, uint32_t idext, uint8_t dlc_, uint8_t *data_){

  if(idstd > 0x7FF){
    Serial.println("Erro IDsdt > 0x7FF");
    return;
  }
  if(idext > 0x3FFFF){
    Serial.println("Erro IDsdt > 0x7FF");
    return;
  }
  if(dlc_ > 8){
    Serial.println("Erro DLC > 8");
    return;
  }
  type = F("Ext. Data");
  IDunion_t idstd_;

  id_std = idstd;
  idstd_.u16 = idstd;
  bts[1] = idstd_.bytes[0];
  bts[0] = idstd_.bytes[1];
```

```
  IDunion_t idext_;

  id_ext = idext;
  idext_.u32 = idext;
  bts[4] = idext_.bytes[1];
  bts[3] = idext_.bytes[2];
  bts[2] = idext_.bytes[3];

  dlc = dlc_;
  bts[5] = dlc_;

  for (uint8_t i = 0; i < dlc_; i++){
    bts[i + 6] = data_[i];
    data[i] = data_[i];
  }
}
void CANframe::reload(uint16_t idstd, uint32_t idext, uint8_t dlc_, uint8_t *data_){

  if(idstd > 0x7FF){
    Serial.println("Erro IDsdt > 0x7FF");
```

```
  return;
}
if(idext > 0x3FFFF){
  Serial.println("Erro IDsdt > 0x7FF");
  return;
}
if(dlc_ > 8){
  Serial.println("Erro DLC > 8");
  return;
}

type = F("Ext. Data");

IDunion_t idstd_;

id_std = idstd;
idstd_.u16 = idstd;
bts[1] = idstd_.bytes[0];
bts[0] = idstd_.bytes[1];

IDunion_t idext_;
```

```
  id_ext = idext;
  idext_.u32 = idext;
  bts[4] = idext_.bytes[1];
  bts[3] = idext_.bytes[2];
  bts[2] = idext_.bytes[3];

  dlc = dlc_;
  bts[5] = dlc_;

  for (uint8_t i = 0; i < dlc_; i++){
    bts[i + 6] = data_[i];
    data[i] = data_[i];
  }
}

CANframe::CANframe(uint16_t idstd, uint8_t dlc_, uint8_t *data_){
  if(idstd > 0x7FF){
    Serial.println("Erro IDsdt > 0x7FF");
    return;
  }
```

```
if(dlc_ > 8){
  Serial.println("Erro DLC > 8");
  return;
}
id_ext = 0;
type = F("Std. Data");

IDunion_t idstd_;

id_std = idstd;
idstd_.u16 = idstd;
bts[1] = idstd_.bytes[0];
bts[0] = idstd_.bytes[1];

dlc = dlc_;
bts[5] = dlc_;

  for (uint8_t i = 0; i < dlc_; i++){
    bts[i + 6] = data_[i];
    data[i] = data_[i];
  }
```

```
}
void CANframe::reload(uint16_t idstd, uint8_t dlc_, uint8_t *data_){
  if(idstd > 0x7FF){
    Serial.println("Erro IDsdt > 0x7FF");
    return;
  }
  if(dlc_ > 8){
    Serial.println("Erro DLC > 8");
    return;
  }
  id_ext = 0;
  type = F("Std. Data");
  IDunion_t idstd_;

  id_std = idstd;
  idstd_.u16 = idstd;
  bts[1] = idstd_.bytes[0];
  bts[0] = idstd_.bytes[1];

  dlc = dlc_;
  bts[5] = dlc_;
```

```
    for (uint8_t i = 0; i < dlc_; i++){
      bts[i + 6] = data_[i];
      data[i] = data_[i];
    }
}

CANframe::CANframe(){

}

void CANframe::reload(uint8_t dlc_, uint8_t *data_){

  if(dlc_ > 8){
    Serial.println("Erro DLC > 8");
    return;
  }

  dlc = dlc_;
  bts[5] = dlc_;
```

```
    for (uint8_t i = 0; i < dlc_; i++){
      bts[i + 6] = data_[i];
      data[i] = data_[i];
    }
}

//MCP2515 object functions

//SPI Construtor
MCP2515::MCP2515(uint8_t spi_cs, unsigned long int spi_speed = 10000000, uint8_t spi_wMode = 0){
  SPI_cs = spi_cs;
  SPI_speed = spi_speed;
  SPI_wMode = spi_wMode;
}

//MCP2515 initialize
void MCP2515::begin(){
  if(SPI_speed > 10000000){
    errLog = F("SPI speed must be <= 10 Mhz");
    return;
  }
```

```
if(SPI_wMode != 0 and SPI_wMode != 3){
  errLog = F("SPI mode must be 0 | 3");
  return;
}
if(crystalCLK != 4 and crystalCLK != 8 and crystalCLK != 20){
  errLog = F("Cirstal clk supported (MHz): 4 | 8 | 20");
  return;
}

pinMode(SPI_cs, OUTPUT);
digitalWrite(SPI_cs, HIGH);
SPI.begin();



reset();
confMode();
confINT();
confRX();
confTX();
confFM();
confCAN();
```

```
}

//SPI initialize
void MCP2515::start(){

  SPI.beginTransaction(SPISettings(SPI_speed, MSBFIRST, SPI_wMode));
  digitalWrite(SPI_cs, LOW);
}

//SPI off
void MCP2515::end() {
    digitalWrite(SPI_cs, HIGH);
    SPI.endTransaction();
}

//Reset MCP215
void MCP2515::reset(){
    start();
    SPI.transfer(0xC0);
    end();
    delayMicroseconds(10);
```

```
}

//Read bytes from registers
void MCP2515::read(uint8_t reg, uint8_t *data, uint8_t n = 1){
    for(uint8_t i = 0; i < n; i++){
      data[i] = 0;
    }
    start();
    SPI.transfer(0x03);
    SPI.transfer(reg);
    for(uint8_t i = 0; i < n; i++){
      data[i] = SPI.transfer(0x00);
    }
    end();
    return;
}

//Check register value
uint8_t MCP2515::regCheck(uint8_t reg, uint8_t val, uint8_t extraMask = 0xFF){
        uint8_t erroRegCheck = 0;
```

```
uint8_t data[1];
read(reg, data);
data[0] = data[0] & extraMask;

//pode tentar por switch
//Condicoes para mascaras 0XFF, i. e., sem mascaras
if ((0x36<=reg && reg <=0x3D) | (0x46<=reg && reg <=0x4D) |
    (0x56<=reg && reg<=0x5D) | ((reg & 0x0F)==0x0F)){

        if(data[0] != val){
                erroRegCheck = 1;
    }
        return erroRegCheck;
}

//Condicoes sem mascaras continuacao
//REG de filtros/mascaras nao le/escreve fora do modo conf
uint8_t FFvet[] = {0x31, 0x33, 0x34, 0x41, 0x43, 0x44, 0x51,
                                    0x53, 0x54, 0x18, 0x14, 0x10, 0x08, 0x04,
                                    0x00, 0x1B, 0x17, 0x13, 0x0B, 0x07, 0x03,
                                    0x1A, 0x16, 0x12, 0x0A, 0x06, 0x02, 0x20,
```

```
                                    0x24, 0x27, 0x23, 0x26, 0x22, 0x2C, 0x2B,
                                    0x2A, 0x29};
for (uint8_t i = 0; i< sizeof(FFvet); i++){
  if(reg == FFvet[i]){
    if(data[0] != val){
            erroRegCheck = 1;
    }
    return erroRegCheck;
  }
}

//Condicoes para os demais valores de mascaras
//mapa de mascaras para check de escritra
uint8_t checkMaskMap [25][2];
//Buff de saida
checkMaskMap[0][0] = 0x30;  checkMaskMap[0][1] = 0x0B;
checkMaskMap[1][0] = 0x32;  checkMaskMap[1][1] = 0xEB;
checkMaskMap[2][0] = 0x35;  checkMaskMap[2][1] = 0x4F;
checkMaskMap[3][0] = 0x40;  checkMaskMap[3][1] = 0x0B;
checkMaskMap[4][0] = 0x42;  checkMaskMap[4][1] = 0xEB;
checkMaskMap[5][0] = 0x45;  checkMaskMap[5][1] = 0x4F;
```

```
checkMaskMap[6][0] = 0x50;  checkMaskMap[6][1] = 0x0B;
checkMaskMap[7][0] = 0x52;  checkMaskMap[7][1] = 0xEB;
checkMaskMap[8][0] = 0x55;  checkMaskMap[8][1] = 0x4F;
checkMaskMap[9][0] = 0x0D;  checkMaskMap[9][1] = 0x07;

//Buff de entrada
checkMaskMap[10][0] = 0x60;  checkMaskMap[10][1] = 0x64;
checkMaskMap[11][0] = 0x70;  checkMaskMap[11][1] = 0x60;
checkMaskMap[12][0] = 0x0C;  checkMaskMap[12][1] = 0x3F;

//filtros
checkMaskMap[13][0] = 0x01;  checkMaskMap[13][1] = 0xEB;
checkMaskMap[14][0] = 0x05;  checkMaskMap[14][1] = 0xEB;
checkMaskMap[15][0] = 0x09;  checkMaskMap[15][1] = 0xEB;
checkMaskMap[16][0] = 0x11;  checkMaskMap[16][1] = 0xEB;
checkMaskMap[17][0] = 0x15;  checkMaskMap[17][1] = 0xEB;
checkMaskMap[18][0] = 0x19;  checkMaskMap[18][1] = 0xEB;

//mascaras
checkMaskMap[19][0] = 0x21;  checkMaskMap[19][1] = 0xE3;
checkMaskMap[20][0] = 0x25;  checkMaskMap[20][1] = 0xE3;
```

```
//logger erro
checkMaskMap[21][0] = 0x1C;  checkMaskMap[21][1] = 0x00;
checkMaskMap[22][0] = 0x1D;  checkMaskMap[22][1] = 0x00;
checkMaskMap[23][0] = 0x2D;  checkMaskMap[23][1] = 0xC0;

//configuracao
checkMaskMap[24][0] = 0x28;  checkMaskMap[24][1] = 0xC7;

for(uint8_t i = 0; i< sizeof(checkMaskMap)/sizeof(checkMaskMap[0]); i++){
  if(checkMaskMap[i][0] == reg){
    if((data[0] & checkMaskMap[i][1]) != (val & checkMaskMap[i][1])){
      erroRegCheck = 1;
    }
    return erroRegCheck;
  }
}

//Condicoes de registro invalido para escrita
if (((reg & 0x0F)==0x0E) | (0x61<=reg<=0x6D) |
     (0x71<=reg<=0x7D) | (0x1C<=reg<=0x1D)){
```

```
            erroRegCheck = 2;
            return erroRegCheck;
          }
          return erroRegCheck;
}

//Write a byte in a register
void MCP2515::write(uint8_t reg, uint8_t val, uint8_t check = 0){
    start();
    SPI.transfer(0x02);
    SPI.transfer(reg);
    SPI.transfer(val);
    end();

    if(check == 1){
      uint8_t erroWrite =0;
      erroWrite = regCheck(reg, val);

      if(erroWrite == 1){
        errLog = werr;
              return;
```

```
      }

      if(erroWrite == 2){
        errLog = F("Invalid Reg");
                    return;
      }

          return;
    }
        return;
}

//Write a bits in a register
void MCP2515::bitModify(uint8_t reg, uint8_t mask, uint8_t val, uint8_t check = 0){
        start();
        SPI.transfer(0x05);
        SPI.transfer(reg);
        SPI.transfer(mask);
  SPI.transfer(val);
  end();
```

```
  if(check == 1){
    uint8_t erroBitModify =0;
    val = mask & val;

    erroBitModify = regCheck(reg, val);

    if( erroBitModify == 1){
      errLog = werr;
            return;
    }

    if( erroBitModify == 2){
      errLog = F("Invalid Reg");
                  return;
    }
          return;
  }
        return;
}

//configuracoes
```

```
void MCP2515::confMode(){
  if(regCheck(0x0F, 0x80, 0xE0) != 0){
    uint8_t crt = 0;
    while((regCheck(0x0F, 0x80, 0xE0) != 0) && (crt < 100) ){
      bitModify(0x0F, 0xE0, 0x80);
      crt = crt + 1;
      delayMicroseconds(10);
      if((crt == 100) && (regCheck(0x0F, 0x80, 0xE0) != 0)){
        errLog = F("Conf e1");
        return;
      }
    }
    return;
  }

  if(regCheck(0x0F, 0x80, 0xE0) == 0){
    write(0x0F, wMode, 1);
    if(errLog == werr){
      return;
    }
    return;
```

```
  }
}

void MCP2515::confRX(){

  confMode();

  write(0x60, RXB0CTRL, 1);
  if(errLog == werr){
    return;
  }

  write(0x70, RXB1CTRL, 1);
  if(errLog == werr){
    return;
  }

  confMode();
}

void MCP2515::confTX(){
```

```
  confMode();
  write(0x30, TXB0CTRL, 1);
  if(errLog == werr){
    return;
  }
  write(0x40, TXB1CTRL, 1);
  if(errLog == werr){
    return;
  }
  write(0x50, TXB2CTRL, 1);
  if(errLog == werr){
    return;
  }
  confMode();
}

void MCP2515::confINT(){
  confMode();
    write(0x2B, CANINTE, 1);
    if(errLog == werr){
      return;
```

```
    }
    write(0x0C, BFPCTRL, 1);
    if(errLog == werr){
      return;
    }
    write(0x0D, TXRTSCTRL, 1);
    if(errLog == werr){
      return;
    }
  confMode();
}

void MCP2515::confFM(){

  uint8_t FILreg[6] = {0x0, 0x4, 0x8, 0x10, 0x14, 0x18};
  uint8_t MASreg[2] = {0x20, 0x24};
  IDunion_t Eid, Sid;

  confMode();

  for(uint8_t i = 0; i < 6; i++){
```

```
Eid.u32 = FILext[i];
Sid.u16 = FILstd[i];

if(Sid.u16 > 0x7FF){
  errLog = F("confFM: sID must be <= 0x7FF");
  return;
}
if(Eid.u32 > 0x3FFFF){
  errLog = F("confFM: eID must be <= 0x3FFFF");
  return;
}

//if(Eid.u32 > 0){
  bitWrite(Eid.bytes[2], 3, 1);
//}else{
  //bitWrite(Eid.bytes[2], 3, 0);
//}
write(FILreg[i]+3, Eid.bytes[0], 1);
write(FILreg[i]+2, Eid.bytes[1], 1);

Eid.bytes[2] = Eid.bytes[2] | Sid.bytes[0] << 5;
```

```
    write(FILreg[i]+1, Eid.bytes[2], 1);
    Sid.bytes[1] = Sid.bytes[1] << 5 | Sid.bytes[0] >> 3;
    write(FILreg[i], Sid.bytes[1], 1);
    if(errLog == werr){
      return;
    }
  }

  for(uint8_t i = 0; i < 2; i++){
    Eid.u32 = MASext[i];
    Sid.u16 = MASstd[i];
    if(Sid.u16 > 0x7FF){
      errLog = F("confFM: sID must be <= 0x7FF");
      return;
    }
    if(Eid.u32 > 0x3FFFF){
      errLog = F("confFM: eID must be <= 0x3FFFF");
      return;
    }

    write(MASreg[i]+3, Eid.bytes[0], 1);
```

```
    write(MASreg[i]+2, Eid.bytes[1], 1);
    Eid.bytes[2] = Eid.bytes[2] | Sid.bytes[0] << 5;
    write(MASreg[i]+1, Eid.bytes[2], 1);
    Sid.bytes[1] = Sid.bytes[1] << 5 | Sid.bytes[0] >> 3;
    write(MASreg[i], Sid.bytes[1], 1);
    if(errLog == werr){
      return;
    }
  }
  confMode();
  return;
}

void MCP2515::confCAN(){
  //Verifica e coloca em modo de configuracao
  switch (crystalCLK) {
    case 8:
      switch (bitF) {
        case 500:
          break;
        case 250:
```

```
      CNF2 = 0x57;
      CNF3 = 0x03;
      break;
    case 125:
      CNF1 = 0x01;
      CNF2 = 0x57;
      CNF3 = 0x03;
      break;
    default:
      errLog = F("Bit frequency must be 500 | 250 | 125 for 8 MHz cristal");
      break;
  };
  break;
case 4:
  switch (bitF) {
      case 250:
        break;
      case 125:
        CNF2 = 0x57;
        CNF3 = 0x03;
        break;
```

```
      default:
        errLog = F("Bit frequency must be 250 | 125 for 4 MHz cristal");
        break;
  };
break;

case 20:
  switch (bitF) {
    case 1000:
      CNF2 = 0x44;
      break;
    case 500:
      CNF2 = 0x80;
      CNF2 = 0x6F;
      CNF3 = 0x04;
      break;
    case 250:
      CNF2 = 0x81;
      CNF2 = 0x6F;
      CNF3 = 0x04;
      break;
```

```
        case 125:
          CNF2 = 0x83;
          CNF2 = 0x6F;
          CNF3 = 0x04;
          break;
        default:
          errLog = F("Bit frequency must be 1000 | 500 | 250 | 125");
          break;
      };
    break;

    default:
      errLog = F("Cristal clock must be 4 | 8 | 20");
      break;

};

 confMode();

 write(0x2A, CNF1, 1);
 if(errLog == werr){
```

```
    return;
  }

  write(0x29, CNF2, 1);
  if(errLog == werr){
    return;
  }
  write(0x28, CNF3, 1);
  if(errLog == werr){
    return;
  }

  //setando o modo de operacao OPMODE  bitF, crystalCLK
  confMode();
  return;
}

//Contador de erros e status functions
void MCP2515::status(uint8_t *status){
  start();
  SPI.transfer(0xA0);
```

```
  status[0] = SPI.transfer(0x00);
  status[0] = SPI.transfer(0x00);
  end();
}

void MCP2515::RXstatus(uint8_t *status){
  start();
  SPI.transfer(0xB0);
  status[0] = SPI.transfer(0x00);
  SPI.transfer(0x00);
  end();
}

void MCP2515::errCount(){
  uint8_t readE[1] = {0};

  read(0x1C, readE);
  TEC = readE[0];

  read(0x1D, readE);
  REC = readE[0];
```

```
read(0x2D, readE);
if(readE[0] != 0){
  if(bitRead(readE[0], 5) == 1){
    errMode = F("Bus-Off");
  }else{
    if(bitRead(readE[0], 4) == 1){
      errMode = F("Error Passive");
    }else{
      if(bitRead(readE[0], 2) == 1){
        errMode = F("Error Active");
      }
    }
  }
  if(bitRead(readE[0], 3) == 1){
      errMode = F("Error Passive");
  }else{
    if(bitRead(readE[0], 1) == 1){
      errMode = F("Error Active");
    }
  }
```

```
}

if(bitRead(readE[0], 7) == 1){
  //bitModify(0x2D, 0x80, 0);
  RX1OVR = RX1OVR + 1;
}
if(bitRead(readE[0], 6) == 1){
  //bitModify(0x2D, 0x40, 0);
  RX0OVR = RX0OVR + 1;
}
write(0x2D, 0);

read(0x2C, readE);
if(bitRead(readE[0], 7) == 1){
  //bitModify(0x2C, 0x80, 0);
  MERRF = MERRF + 1;
}

if(bitRead(readE[0], 6) == 1){
  //bitModify(0x2C, 0x40, 0);
  WAKIE = 1;
```

```
  }else{
    WAKIE = 0;
  }

  if(bitRead(readE[0], 5) == 1){
    //bitModify(0x2C, 0x20, 0);
    multInt = multInt + 1;
  }
  //interrupt flags reset
  bitModify(0x2C, 0xE0, 0x00);
  return;
}

//Writing functions
void MCP2515::writeID(uint16_t sid, uint8_t id_ef = 0, uint32_t eid = 0,
                      uint8_t txb = 0, uint8_t timeOut = 10, uint8_t check = 0){
    uint32_t t = micros();
    uint8_t txcrt[1]={0};

    if(sid > 0x7FF){
      errLog = F("sID must be <= 0x7FF");
```

```
    return;
  }
  if(eid > 0x3FFFF){
    errLog = F("eID must be <= 0x3FFFF");
    return;
  }

  switch(txb){
    case 0:
      txb = 0x30;
      break;

    case 1:
      txb = 0x40;
      break;

    case 2:
      txb = 0x50;
      break;

    default:
```

```
    errLog = F("TXB must be 0 | 1 | 2");
    return;
};

IDunion_t id_ext, id_std;

id_std.u16 = sid;

if(id_ef > 0){
  id_ext.u32 = eid;
  bitWrite(id_ext.bytes[2], 3, 1);
}else{
  id_ext.u32 = 0;
  bitWrite(id_ext.bytes[2], 3, 0);
}
do {
  read(txb, txcrt);
  if(bitRead(txcrt[0], 3) == 0 ){
    write(txb+4, id_ext.bytes[0], check);
    write(txb+3, id_ext.bytes[1], check);
```

```
        id_ext.bytes[2] = id_ext.bytes[2] | id_std.bytes[0] << 5;

        write(txb+2, id_ext.bytes[2], check);

        id_std.bytes[1] = id_std.bytes[1] << 5 | id_std.bytes[0] >> 3;
        write(txb+1, id_std.bytes[1], check);
        return;
      }
      delayMicroseconds(10);
    } while(micros() - t < timeOut*1000);

}

void MCP2515::loadTX(uint8_t *data, uint8_t n = 8, uint8_t abc = 1){
        /* Descricao da funcao...
        *
        *         1) Verifica se abc e valido, ele tem que ser que esta no intervalo de
        * 0 a 5 de acordo com a pagina 69 do datasheet.
        *
        *         2) Realiza a transferencia dos bytes contidos em data
        *
```

```
      *        3) futuro ->  possibilidade de abilitar um check.
      *
      * Argumentos,         data : posicao da memoria onde inicia o vetor de dados
      *                             abc  : numero inteiro de 0 a 5, de acordo com a instrucao pg. 69
      *                             n    : numero de registros a serem escritos
      *        futuro ->        arg4 : possibilidade de abilitar um check.
      *
      * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

if(abc > 5){
  errLog = F("Invalid abc");
  return;
}

start();
SPI.transfer( 0x40 + abc );

for (uint8_t i = 0; i < n; i ++){
  SPI.transfer(data[i]);
}
end();
```

```
  return;
}

void MCP2515::send(uint8_t txBuff = 0x01){

  start();
  SPI.transfer(0x80 + txBuff);
  end();
}

void MCP2515::writeFrame(CANframe frameToSend, uint8_t txb_ = 0, uint8_t timeOut = 10, uint8_t check = 0){

  uint8_t txcrt[1] = {0xFF}, sendCode, loadCode, txb;
  uint32_t  t = micros();

  errLog = F("no error");

  if(frameToSend.dlc > 8){
    errLog == F("The data field size must be <= 8 bytes");
    return;
```

```
}

      switch (txb_){
               case 0:
    txb = 0x30;
    sendCode = 0x01;
    loadCode = 0x01;
                 break;

               case 1:
    txb = 0x40;
    sendCode = 0x02;
    loadCode = 0x03;
                 break;

               case 2:
    txb = 0x50;
                 sendCode = 0x04;
    loadCode = 0x05;
                 break;
```

```
            default:
                    errLog = F("TXB must be 0 | 1 | 2");
                    return;
      };

do {
  read(txb, txcrt);
  if(bitRead(txcrt[0], 3) == 0 ){
    write(txb+5, frameToSend.dlc, check); // alterar para permitir frame de pedido remoto
    if(errLog == werr){
      return;
    }
    if(frameToSend.type == F("Ext. Data")){
      writeID(frameToSend.id_std, 1, frameToSend.id_ext, txb_);
    }
    if(frameToSend.type == F("Std. Data")){
      writeID(frameToSend.id_std, 0, 0, txb_);
    }else{
      errLog = F("Invalid Frame type");
    }
    loadTX(frameToSend.data, frameToSend.dlc, loadCode);
```

```
      send(sendCode);
      return;
    }
    delayMicroseconds(10);
  } while((micros() - t) < timeOut*1000);

        errLog == F("Buffs full");
        return;
}

void MCP2515::abort(uint8_t abortCode = 7){

                            uint8_t rreadd[1];

  if(abortCode == 7){
                            read(0x0F, rreadd);

    bitModify(0x0F,  0x10, 0x10);
                            read(0x0F, rreadd);

    return;
```

```
}

confMode();
if((abortCode & 0x1) == 0x1){
                            read(0x30, rreadd);

              bitWrite(TXB0CTRL, 3, 0);
              write(0x30, TXB0CTRL);
              bitWrite(TXB0CTRL, 3, 1);
  bitModify(0x30,  0x08, 0x00);
                            read(0x30, rreadd);

}
if((abortCode & 0x2) == 0x2){

                            read(0x40, rreadd);

                bitWrite(TXB1CTRL, 3, 0);
                write(0x40, TXB1CTRL);
                bitWrite(TXB1CTRL, 3, 1);
  bitModify(0x40,  0x08, 0x00);
```

```
                                read(0x40, rreadd);


  }
  if((abortCode & 0x4) == 0x4){
                                read(0x50, rreadd);

                  bitWrite(TXB2CTRL, 3, 0);
                  write(0x50, TXB2CTRL);
                  bitWrite(TXB2CTRL, 3, 1);
    bitModify(0x50,  0x08, 0x00);
                                read(0x50, rreadd);


  }
  confMode();
  return;

}

//Reading functions
```

```
void MCP2515::readID(uint8_t *id, uint8_t rxb = 0){

  id[0] = 0;
  id[1] = 0;
  id[2] = 0;
  id[3] = 0;
  id[4] = 0;

  switch(rxb){
    case 0: //RXB0
      rxb = 0x61;
      //read(0x61, id, 4); //SID[0] = SIDH e SID[1] = SIDL
      break;

    case 1:
      rxb = 0x71;
      //read(0x71, id, 4);
      break;

    default:
      errLog  = F("RXB must be 0 | 1");
```

```
        return;
    };

    read(rxb, id, 4);
    id[4] = id[3];
    id[3] = id[2];
    id[2] = id[1]  & 0x3;
    id[1] = id[1] >> 5 | id[0] << 3;
    id[0] = id[0] >> 5;

    return;
}

void MCP2515::readFrame(){

    frameRXB0.type = F("No frame");
    frameRXB1.type = F("No frame");

    uint8_t rx_status[1];
    RXstatus(rx_status);
    if((rx_status[0] >> 6) == 0){
```

```
  return;
}

switch((rx_status[0] >> 6)){
  case 1: //Only RXB0 have a new data
    read(0x65, frameRXB0.bts);

    frameRXB0.dlc = frameRXB0.bts[0] & 0xF;

    read(0x66, frameRXB0.bts, frameRXB0.dlc);

    frameRXB0.bts[13] = frameRXB0.bts[7];
    frameRXB0.bts[12] = frameRXB0.bts[6];
    frameRXB0.bts[11] = frameRXB0.bts[5];
    frameRXB0.bts[10] = frameRXB0.bts[4];
    frameRXB0.bts[9] = frameRXB0.bts[3];
    frameRXB0.bts[8] = frameRXB0.bts[2];
    frameRXB0.bts[7] = frameRXB0.bts[1];
    frameRXB0.bts[6] = frameRXB0.bts[0];
    frameRXB0.bts[5] = frameRXB0.dlc;
```

```
readID(frameRXB0.bts, 0);
if((rx_status[0] & 0x18) == 0x10){
  frameRXB0.type = F("Ext. Data");
  frameRXB0.reload(frameRXB0.bts, 1);
  //interrupt flags reset
  bitModify(0x2C, 0x01, 0);
  return;
}
if((rx_status[0] & 0x18) == 0){
  frameRXB0.type = F("Std. Data");
  frameRXB0.id_ext = 0;
  frameRXB0.reload(frameRXB0.bts);
  //interrupt flags reset
  bitModify(0x2C, 0x01, 0);
  return;
}
if((rx_status[0] & 0x18) == 0x08){
  frameRXB0.type = "Std. Remote";
  frameRXB0.id_ext = 0;
  frameRXB0.reload(frameRXB0.bts);
  //interrupt flags reset
```

```
    bitModify(0x2C, 0x01, 0);
    return;
  }
  if((rx_status[0] & 0x18) == 0x18){
    frameRXB0.type = F("Ext. Remote");
    frameRXB0.reload(frameRXB0.bts, 1);
    //interrupt flags reset
    bitModify(0x2C, 0x01, 0);
    return;
  }

case 2:  //Only RXB1 have a new data

  read(0x75, frameRXB1.bts);

  frameRXB1.bts[8] = frameRXB1.bts[0] & 0xF;

  read(0x76, frameRXB1.bts, frameRXB1.bts[8]);

  frameRXB1.bts[13] = frameRXB1.bts[7];
  frameRXB1.bts[12] = frameRXB1.bts[6];
```

```
frameRXB1.bts[11] = frameRXB1.bts[5];
frameRXB1.bts[10] = frameRXB1.bts[4];
frameRXB1.bts[9] = frameRXB1.bts[3];
frameRXB1.bts[8] = frameRXB1.bts[2];
frameRXB1.bts[7] = frameRXB1.bts[1];
frameRXB1.bts[6] = frameRXB1.bts[0];
frameRXB1.bts[5] = frameRXB1.dlc;

readID(frameRXB1.bts, 1);
if((rx_status[0] & 0x18) == 0x10){
  frameRXB1.type = F("Ext. Data");
  frameRXB0.reload(frameRXB0.bts, 1);
  //interrupt flags reset
  bitModify(0x2C, 0x02, 0);
  return;
}
if((rx_status[0] & 0x18) == 0){
  frameRXB1.type = F("Std. Data");
  frameRXB1.id_ext = 0;
  frameRXB0.reload(frameRXB0.bts);
  //interrupt flags reset
```

```
    bitModify(0x2C, 0x02, 0);
    return;
  }
  if((rx_status[0] & 0x18) == 0x08){
    frameRXB1.type = F("Std. Remote");
    frameRXB1.id_ext = 0;
    frameRXB0.reload(frameRXB0.bts);
    //interrupt flags reset
    bitModify(0x2C, 0x02, 0);
    return;
  }
  if((rx_status[0] & 0x18) == 0x18){
    frameRXB1.type = F("Ext. Remote");
    frameRXB0.reload(frameRXB0.bts, 1);
    //interrupt flags reset
    bitModify(0x2C, 0x02, 0);
    return;
  }

case 3: // Both receivers buffers have a new data
```

```
read(0x65, frameRXB0.bts);
read(0x75, frameRXB1.bts);

frameRXB0.dlc = frameRXB0.bts[0] & 0xF;
frameRXB1.dlc = frameRXB1.bts[0] & 0xF;


read(0x66, frameRXB0.bts, frameRXB0.dlc);
read(0x76, frameRXB1.bts, frameRXB1.dlc);

frameRXB0.bts[13] = frameRXB0.bts[7];
frameRXB0.bts[12] = frameRXB0.bts[6];
frameRXB0.bts[11] = frameRXB0.bts[5];
frameRXB0.bts[10] = frameRXB0.bts[4];
frameRXB0.bts[9] = frameRXB0.bts[3];
frameRXB0.bts[8] = frameRXB0.bts[2];
frameRXB0.bts[7] = frameRXB0.bts[1];
frameRXB0.bts[6] = frameRXB0.bts[0];
frameRXB0.bts[5] = frameRXB0.dlc;

frameRXB1.bts[13] = frameRXB1.bts[7];
```

```
frameRXB1.bts[12] = frameRXB1.bts[6];
frameRXB1.bts[11] = frameRXB1.bts[5];
frameRXB1.bts[10] = frameRXB1.bts[4];
frameRXB1.bts[9] = frameRXB1.bts[3];
frameRXB1.bts[8] = frameRXB1.bts[2];
frameRXB1.bts[7] = frameRXB1.bts[1];
frameRXB1.bts[6] = frameRXB1.bts[0];
frameRXB1.bts[5] = frameRXB1.dlc;

readID(frameRXB0.bts, 0);

read(0x62, rx_status);
if((rx_status[0] & 0x08) == 0){
  frameRXB0.type = F("Std. Data");
  frameRXB0.id_ext = 0;
  frameRXB0.reload(frameRXB0.bts);
}
if((rx_status[0] & 0x08) == 0x08){
  frameRXB0.type = F("Ext. Data");
  read(0x65, rx_status);
  if((rx_status[0] & 0x40) == 0x40){
```

```
    frameRXB0.type = F("Ext. Remote");
  }
  frameRXB0.reload(frameRXB0.bts, 1);
}

readID(frameRXB1.bts, 1);

read(0x72, rx_status);
if((rx_status[0] & 0x08) == 0){
  frameRXB1.type = F("Std. Data");
  frameRXB1.id_ext = 0;
  frameRXB1.reload(frameRXB0.bts);
}
if((rx_status[0] & 0x08) == 0x08){
  frameRXB1.type = F("Ext. Data");
  read(0x75, rx_status);
  if((rx_status[0] & 0x40) == 0x40){
    frameRXB1.type = F("Ext. Remote");
  }
  frameRXB1.reload(frameRXB0.bts, 1);
}
```

```
      //interrupt flags reset
      bitModify(0x2C, 0x03, 0x00);
      return;

    default:

      return;
  };

}


//General informations
void MCP2515::digaOi(){
  uint8_t readE[1] = {0};
////////////////////////////////////////////////////////////////////////
  Serial.println(F("Status reg"));
  read(0x0E, readE);
  switch (readE[0] & 0xE0){
    case 0:
      Serial.println(F("Normal mode"));
```

```
    break;
  case 1:
    Serial.println(F("Sleep mode"));
    break;
  case 2:
    Serial.println(F("Loopback mode"));
    break;
  case 3:
    Serial.println(F("Listen-Only mode"));
    break;
  case 4:
    Serial.println(F("Configuration mode"));
    break;
};
switch (readE[0] & 0x0E){
  case 0:
    Serial.println(F("No interrupt flags"));
    break;
  case 1:
    Serial.println(F("Error interrupt"));
    break;
```

```
    case 2:
      Serial.println(F("Wake-up interrupt"));
      break;
    case 3:
      Serial.println(F("TXB0 interrupt"));
      break;
    case 4:
      Serial.println(F("TXB1 interrupt"));
      break;
    case 5:
      Serial.println(F("TXB2 interrupt"));
      break;
    case 6:
      Serial.println(F("RXB0 interrupt"));
      break;
    case 7:
      Serial.println(F("RXB1 interrupt"));
      break;
  };
  Serial.println();
////////////////////////////////////////////////////////////////////////////////
```

```
  Serial.println(F("Errors"));
  Serial.print(F("Logg_error: "));
  Serial.println(errLog);

  errCount();
  Serial.print(F("TEC: "));
  Serial.println(TEC);
  Serial.print(F("REC: "));
  Serial.println(REC);
  Serial.print(F("Confinament Mode: "));
  Serial.println(errMode);
  Serial.print(F("RB1_Over_Flow_cont: "));
  Serial.println(RX1OVR);
  Serial.print(F("RB0_Over_Flow_cont: "));
  Serial.println(RX0OVR);
  Serial.println();
////////////////////////////////////////////////////////////////////////////////
  Serial.println(F("SPI setted  parameters"));
  Serial.print(F("SPI_max_speed: "));
  Serial.println(SPI_speed);
  Serial.print(F("SPI_work_mode: "));
```

```
  Serial.println(SPI_wMode);
  Serial.print(F("SPI_CS: "));
  Serial.println(SPI_cs);
  Serial.println();
//////////////////////////////////////////////////////////////////////////////
  Serial.println(F("CAN_2515 setted parameters"));
  Serial.print(F("CLK_2515: "));
  Serial.println(crystalCLK);
  Serial.print(F("Bits_frequency: "));
  Serial.println(bitF);
  Serial.print(F("CAN_work_mode: "));
  switch (wMode) {
    case 0:
      Serial.println(F("Normal mode"));
      break;
    case 1:
      Serial.println(F("Sleep mode"));
      break;
    case 2:
      Serial.println(F("Loopback mode"));
      break;
```

```
    case 3:
      Serial.println(F("Listen-Only mode"));
      break;
    case 4:
      Serial.println(F("Configuration mode"));
      break;
  };
  Serial.println();
//////////////////////////////////////////////////////////////////////////////
  Serial.println(F("RX, TX and INT setted parameters in HEX"));
  Serial.print(F("RXB0: "));
  Serial.println(RXB0CTRL, HEX);
  Serial.print(F("RXB1: "));
  Serial.println(RXB1CTRL, HEX);
  Serial.print(F("TXB0: "));
  Serial.println(TXB0CTRL, HEX);
  Serial.print(F("TXB1: "));
  Serial.println(TXB1CTRL, HEX);
  Serial.print(F("TXB2: "));
  Serial.println(TXB2CTRL, HEX);
  Serial.print(F("CANINTE: "));
```

```
  Serial.println(CANINTE, HEX);
  Serial.print(F("BFPCTRL: "));
  Serial.println(BFPCTRL, HEX);
  Serial.print(F("TXRTSCTRL: "));
  Serial.println(TXRTSCTRL, HEX);
  Serial.println();
//////////////////////////////////////////////////////////
Serial.println(F("Setted filters in HEX"));
Serial.print(F("Filter_0: "));
Serial.print(FILstd[0], HEX);
Serial.print(F(", "));
Serial.println(FILext[0], HEX);

Serial.print(F("Filter_1: "));
Serial.print(FILstd[1], HEX);
Serial.print(F(", "));
Serial.println(FILext[1], HEX);

Serial.print(F("Filter_2: "));
Serial.print(FILstd[2], HEX);
Serial.print(F(", "));
```

```
Serial.println(FILext[2], HEX);

Serial.print(F("Filter_3: "));
Serial.print(FILstd[3], HEX);
Serial.print(F(", "));
Serial.println(FILext[3], HEX);

Serial.print(F("Filter_4: "));
Serial.print(FILstd[4], HEX);
Serial.print(F(", "));
Serial.println(FILext[4], HEX);

Serial.print(F("Filter_5: "));
Serial.print(FILstd[5], HEX);
Serial.print(F(", "));
Serial.println(FILext[5], HEX);
Serial.println();
//////////////////////////////////////////////////////////////////
Serial.println(F("Setted Maks in HEX"));
Serial.print(F("Mask_0: "));
Serial.print(MASstd[0], HEX);
```

```
Serial.print(F(", "));
Serial.println(MASext[0], HEX);
Serial.print(F("Mask_1: "));
Serial.print(MASstd[1], HEX);
Serial.print(F(", "));
Serial.println(MASext[1], HEX);

Serial.println();
  return;
}

void MCP2515::digaOi(char *oi){
  uint8_t readE[1] = {0};

////////////////////////////////////////////////////////////////////
if(oi[1] == 't'){
  Serial.println(F("Status reg"));
  read(0x0E, readE);
  switch (readE[0] & 0xE0){
    case 0:
      Serial.println(F("Normal mode"));
```

```
    break;
  case 1:
    Serial.println(F("Sleep mode"));
    break;
  case 2:
    Serial.println(F("Loopback mode"));
    break;
  case 3:
    Serial.println(F("Listen-Only mode"));
    break;
  case 4:
    Serial.println(F("Configuration mode"));
    break;
};
switch (readE[0] & 0x0E){
  case 0:
    Serial.println(F("No interrupt flags"));
    break;
  case 1:
    Serial.println(F("Error interrupt"));
    break;
```

```
    case 2:
      Serial.println(F("Wake-up interrupt"));
      break;
    case 3:
      Serial.println(F("TXB0 interrupt"));
      break;
    case 4:
      Serial.println(F("TXB1 interrupt"));
      break;
    case 5:
      Serial.println(F("TXB2 interrupt"));
      break;
    case 6:
      Serial.println(F("RXB0 interrupt"));
      break;
    case 7:
      Serial.println(F("RXB1 interrupt"));
      break;
  };
  Serial.println();
  return;
```

```
}
////////////////////////////////////////////////////////////////////////////////
if(oi[1] == 'r'){
  Serial.println(F("Errors"));
  Serial.print(F("Logg_error: "));
  Serial.println(errLog);

  errCount();
  Serial.print(F("TEC: "));
  Serial.println();
  Serial.print(F("REC: "));
  Serial.println();
  Serial.print(F("Confinament Mode: "));
  Serial.println(errMode);
  Serial.print(F("oflw_RB1_cont: "));
  Serial.println(RX1OVR);
  Serial.print(F("oflw_RB0_cont: "));
  Serial.println(RX0OVR);
  Serial.println();
  return;
}
```

```
////////////////////////////////////////////////////////////////////////////
if(oi[1] == 'P'){
  Serial.println(F("SPI setted  parameters"));
  Serial.print(F("SPI_max_speed: "));
  Serial.println(SPI_speed);
  Serial.print(F("SPI_work_mode: "));
  Serial.println(SPI_wMode);
  Serial.print(F("SPI_CS: "));
  Serial.println(SPI_cs);
  Serial.println();
  return;
}
////////////////////////////////////////////////////////////////////////////
if(oi[1] == 'A'){
  Serial.println(F("CAN_2515 setted parameters"));
  Serial.print(F("CLK_2515: "));
  Serial.println(crystalCLK);
  Serial.print(F("Bits_freq: "));
  Serial.println(bitF);
  Serial.print(F("CAN_work_mode: "));
  switch (wMode) {
```

```
      case 0:
        Serial.println(F("Normal mode"));
        break;
      case 1:
        Serial.println(F("Sleep mode"));
        break;
      case 2:
        Serial.println(F("Loopback mode"));
        break;
      case 3:
        Serial.println(F("Listen-Only mode"));
        break;
      case 4:
        Serial.println(F("Configuration mode"));
        break;
    };
    Serial.println();
    return;
}
////////////////////////////////////////////////////////////////////////////////
if(oi[1] == 'o'){
```

```
  Serial.println(F("RX, TX and INT setted parameters in HEX"));
  Serial.print(F("RXB0: "));
  Serial.println(RXB0CTRL, HEX);
  Serial.print(F("RXB1: "));
  Serial.println(RXB1CTRL, HEX);
  Serial.print(F("TXB0: "));
  Serial.println(TXB0CTRL, HEX);
  Serial.print(F("TXB1: "));
  Serial.println(TXB1CTRL, HEX);
  Serial.print(F("TXB2: "));
  Serial.println(TXB2CTRL, HEX);
  Serial.print(F("CANINTE: "));
  Serial.println(CANINTE, HEX);
  Serial.print(F("BFPCTRL: "));
  Serial.println(BFPCTRL, HEX);
  Serial.print(F("TXRTSCTRL: "));
  Serial.println(TXRTSCTRL, HEX);
  Serial.println();
  return;
}
//////////////////////////////////////////////////////////
```

```
if(oi[1] == 'i'){
  Serial.println(F("Setted filters in HEX"));
  Serial.print(F("Filter_0: "));
  Serial.print(FILstd[0], HEX);
  Serial.print(F(", "));
  Serial.println(FILext[0], HEX);
  Serial.print(F("Filter_1: "));
  Serial.print(FILstd[1], HEX);
  Serial.print(F(", "));
  Serial.println(FILext[1], HEX);
  Serial.print(F("Filter_2: "));
  Serial.print(FILstd[2], HEX);
  Serial.print(F(", "));
  Serial.println(FILext[2], HEX);
  Serial.print(F("Filter_3: "));
  Serial.print(FILstd[3], HEX);
  Serial.print(F(", "));
  Serial.println(FILext[3], HEX);
  Serial.print(F("Filter_4: "));
  Serial.print(FILstd[4], HEX);
  Serial.print(F(", "));
```

```
  Serial.println(FILext[4], HEX);
  Serial.print(F("Filter_5: "));
  Serial.print(FILstd[5], HEX);
  Serial.print(F(", "));
  Serial.println(FILext[5], HEX);
  Serial.println();
  return;
}
////////////////////////////////////////////////////////////////
if(oi[1] == 'a'){
  Serial.println(F("Setted Maks in HEX"));
  Serial.print(F("Mask_0: "));
  Serial.print(MASstd[0], HEX);
  Serial.print(F(", "));
  Serial.println(MASext[0], HEX);
  Serial.print(F("Mask_1: "));
  Serial.print(MASstd[1], HEX);
  Serial.print(F(", "));
  Serial.println(MASext[1], HEX);
  Serial.println();
  return;
```

```
}

Serial.println(F("Invalid Parameter for digaOi() \n
                  try: nothing | Status? | Errors? | SPI? | CAN? | ConfRegs? | Filters? | Masks?"));
return;
}
```

B.4.3 CANRX.ino

```
#include <MCP2515_1.h>

MCP2515 mcp(4);
/*  objectName(SPI_CS, SPI_MAX_SPEED, SPI_MODE)
 *  Default values:
 *  objectName.SPI_MAX_SPEED = 10000000
 *  objectName.SPI_MODE = 0
*/

const char separator = ' ';

void setup() {
  Serial.begin(9600);

// MCP2515 initialization and configure.
  mcp.begin();


}
```

```
void loop() {
  mcp.readFrame();

  if (mcp.frameRXB0.type != F("No frame")) {
    Serial.print(mcp.frameRXB0.id_std);
    Serial.print(separator);
    Serial.print(mcp.frameRXB0.id_ext);
    Serial.print(separator);
    Serial.print(mcp.frameRXB0.dlc);
    Serial.print(separator);

    for(uint8_t i = 0; i < mcp.frameRXB0.dlc; i++){
      Serial.print(mcp.frameRXB0.data[i]);
      Serial.print(separator);
    }
    Serial.print('\n');
  }

    if (mcp.frameRXB1.type != F("No frame")) {
      Serial.print(mcp.frameRXB1.id_std);
      Serial.print(separator);
```

```
      Serial.print(mcp.frameRXB1.id_ext);
      Serial.print(separator);
      Serial.print(mcp.frameRXB1.dlc);
      Serial.print(separator);

      for(uint8_t i = 0; i < mcp.frameRXB1.dlc; i++){
        Serial.print(mcp.frameRXB1.data[i]);
        Serial.print(separator);
      }
      Serial.print('\n');
    }

}
```

B.4.4 CANTX.ino

```
#include <MCP2515_1.h>

MCP2515 mcp(4);
/*  objectName(SPI_CS, SPI_MAX_SPEED, SPI_MODE)
 *  Default values:
 *  objectName.SPI_MAX_SPEED = 10000000
 *  objectName.SPI_MODE = 0
*/

void setup() {
  //Serial.begin(1000000);

// MCP2515 initialization and configure.
  mcp.begin();



}

void loop() {
  uint16_t ID_std; // Maximum value = 0x7FF, use uint16
```

```
uint32_t ID_ext; // Maximum value = 0x3FFFF, use uint32
uint8_t data_len; // From 0x0 to 0x8, use unit8
uint8_t data_field[8]; // use unit8 - list

ID_std = random(0x7FF);// Generate a randon Standard Id
ID_ext = random(0x3FFFF);

// Generate a randon lenght of data (data_field) on the frame.
data_len = random(0x8);

for(uint8_t i = 0; i < data_len; i++){
  // Generate a randon data for the data_field
  data_field[i] = random(0xFF);
}

// Create a std. data Frame
CANframe frm(ID_std, data_len, data_field);

// Send a frame by TXB0
mcp.writeFrame(frm); // defalt (frm, 0)
```

```
// Reload 'frm' as ext. data Frame
frm.reload(ID_std, ID_ext, data_len, data_field);

// Send a frame by TXB1
mcp.writeFrame(frm, 1);

// Create a ext. data Frame
CANframe frm1(ID_std, ID_ext, data_len, data_field);

// Send a frame by TXB2
mcp.writeFrame(frm1, 2);

//Atencion after
delay(1000);

data_len = random(0x8);

for(uint8_t i = 0; i < data_len; i++){
  // Generate a randon data for the data_field
  data_field[i] = random(0xFF);
}
```

```
  // Is also can reload only whith data_len and data_field
  frm1.reload(data_len, data_field); //but it don't change the IDs
  // Send a frame by TXB0
  mcp.writeFrame(frm1);
  delay(1000);

}
```

B.4.5 CAN Mon

B.4.5.1 CANMon.h

```
MCP2515 mon(4);

CANframe rec;

const uint8_t n = 70; //n = numero maximo de frames armazenados
uint8_t buff[n][14];

const char separator = ' ';

void readCAN () {

  noInterrupts();
  static uint8_t bxi = 0;

  if(bxi == n){
    bxi = 0;
  }
```

```
  mon.readFrame();

  if (mon.frameRXB0.type != F("No frame")) {
        for (uint8_t j = 0; j < mon.frameRXB1.dlc + 6; j++) {
          buff[bxi][j] = mon.frameRXB0.bts[j];
        }
        bxi = bxi + 1;
    }

  if (mon.frameRXB1.type != F("No frame")) {
        for (uint8_t j = 0; j < mon.frameRXB1.dlc + 6; j++) {
          buff[bxi][j] = mon.frameRXB1.bts[j];
        }
        bxi = bxi + 1;
  }

  interrupts();
  return;
}
```

```
void printFrame(const char separator = ','){
  static uint8_t ii = 0;

  if ((buff[ii][5]) < 255 && Serial.availableForWrite() > (buff[ii][5] + 6)) {
    rec.reload(buff[ii]);

    Serial.print(rec.id_std);
    Serial.print(separator);
    Serial.print(rec.id_ext);
    Serial.print(separator);
    Serial.print(rec.dlc);
    Serial.print(separator);

    for(uint8_t i = 6; i < rec.dlc+6; i++){
      Serial.print(rec.bts[i]);
      Serial.print(separator);
    }

    Serial.println();
    buff[ii][5] = 255;
  }
```

```
  ii = ii + 1;
  if (ii == n) {
    ii = 0;
  }
}

void writeFrame_serial(){
  static uint8_t ii = 0;

  if ((buff[ii][5]) < 255 && Serial.availableForWrite() > (buff[ii][5] + 6)) {
    Serial.write(buff[ii], (buff[ii][5] + 6));
    buff[ii][5] = 255;
  }

  ii = ii + 1;
  if (ii == n) {
    ii = 0;
  }
}
```

B.4.5.2 CANMon.ino

```
#include <MCP2515_1.h>
#include "CANMon.h"

void setup() {
  Serial.begin(9600);

  mon.bitF = 125; // k Hertz
  mon.begin();

  if(mon.errLog != F("no error")){
    Serial.println(mon.errLog);
    while(1){
      delay(1000);
    }
  }

  // Zerar o buff, e setar 255 na posicao dlc para controle
  for (uint8_t i = 0; i < 2; i++) {
    for (uint8_t j = 0; j < 13; j++) {
      buff[i][j] = 0;
```

```
      if(j == 5){
        buff[i][j] = 255;
      }
    }
  }

  attachInterrupt(digitalPinToInterrupt(2), readCAN, FALLING);
}

void loop() {
/* Choose between wireFrame_serial () or print_Frame (separator char),
 * commenting and uncommenting the lines below.
 */

 //writeFrame_serial(); // More fast, but out puts bytes format,

 printFrame(' '); //For friendly use and little busy bus, chosen it.
 // ' ' is columns separator char. ',' is default.

}
```

B.4.6 CAN Save

B.4.6.1 CANSave.h

```
CANframe rec;


void readCAN () {


  noInterrupts();
  static uint8_t bxi = 0;


  if(bxi == n){
    bxi = 0;
  }


  mcp.readFrame();


  if (mcp.frameRXB0.type != F("No frame")) {
        for (uint8_t j = 0; j < mcp.frameRXB1.dlc + 6; j++) {
          buff[bxi][j] = mcp.frameRXB0.bts[j];
        }
        bxi = bxi + 1;
```

```
    }

  if (mcp.frameRXB1.type != F("No frame")) {
        for (uint8_t j = 0; j < mcp.frameRXB1.dlc + 6; j++) {
          buff[bxi][j] = mcp.frameRXB1.bts[j];
        }
        bxi = bxi + 1;
  }

  interrupts();
  return;
}


void filesC(){
  while(sd.exists(fileName) and file.fileSize() > 10){
    fileName[16] = fileName[16] + 1;
    if(fileName[16] > 0x39){
      fileName[16] = 0x30;
      fileName[15] = fileName[15] + 1;
      if(fileName[15] > 0x39){
```

```
        fileName[15] = 0x30;
        fileName[14] = fileName[14] + 1;
        if(fileName[14] > 0x39){
          fileName[14] = 0x30;
        }
      }
    }
  }

  file.open(fileName, O_APPEND | O_RDWR | O_CREAT);
    file.print(F("Arquivo log Mini CAN\nIDstd, IDext, dlc, d1, d2, d3, d4, d5, d6, d7, d8\n"));
    return;
}
```

B.4.6.2 CANSave.ino

```
#include <SPI.h>
#include "SdFat.h"
#if ENABLE_SOFTWARE_SPI_CLASS  // Must be set in SdFat/SdFatConfig.h
//
// Pin numbers in templates must be constants.
const uint8_t SOFT_SCK_PIN  = 7;
const uint8_t SOFT_MOSI_PIN = 6;
const uint8_t SOFT_MISO_PIN = 5;
// Chip select may be constant or RAM variable.
const uint8_t SD_CHIP_SELECT_PIN = 8;
// SdFat software SPI template
SdFatSoftSpi<SOFT_MISO_PIN, SOFT_MOSI_PIN, SOFT_SCK_PIN> sd;
// Test file.
SdFile file;

/////////////////////////////////////// MCP2515
#include <MCP2515_1.h>
MCP2515 mcp(4);
const uint8_t n = 20; // n =numero maximo de frames armazenados
uint8_t buff[n][14];
```

```
char fileName[21] = "LOGG/logg_001.csv";
uint8_t fileNumber[3] = {0x30, 0x30, 0x31};
#include "CANSave.h"

void setup() {

  if(!sd.begin(SD_CHIP_SELECT_PIN, SD_SCK_MHZ(50))){
    //Serial.println("CardErr");
    while(1){
      delay(1000);
    }
  }

  delay(100);
  if(!sd.exists("LOGG")){
    sd.mkdir("LOGG");
  }
  filesC();

  delay(100);
```

```
  mcp.begin();
  if(mcp.errLog != F("no error")){
    //Serial.println(mcp.errLog);
    while(1){
      delay(1000);
    }
  }

  for (uint8_t i = 0; i < n; i++) {
      buff[i][5] = 0;
  }

  attachInterrupt(digitalPinToInterrupt(2), readCAN, FALLING);
}

void loop() {
  interrupts();
  static uint8_t ii = 0, buts = 0;
  static unsigned long int nlines = 0;

  //SDprint function
```

```
if (buff[ii][5] > 0 and buts == 0){

  rec.reload(buff[ii]);

  file.printField(rec.id_std, ',');
  file.printField(rec.id_ext, ',');
  file.printField(rec.dlc, ',');

  for(uint8_t kk = 6; kk < rec.dlc + 6; kk++){
    if(kk < rec.dlc + 6 - 1){
      file.printField(rec.bts[kk], ',');
    }
    else{
      file.printField(rec.bts[kk], '\n');
    }
  }


  buff[ii][5] = 0;
  nlines = nlines+1;
}
```

```
  if(nlines >= 3600){//maxl){
    file.close();
    filesC();
    nlines = 0;
  }

  ii = ii + 1;
  if (ii == n) {
    ii = 0;
  }
}
#else  // ENABLE_SOFTWARE_SPI_CLASS
#error ENABLE_SOFTWARE_SPI_CLASS must be set non-zero in SdFat/SdFatConfig.h
#endif  //ENABLE_SOFTWARE_SPI_CLASS
```

B.4.7 CAN Sensor

B.4.7.1 CANSensor.h

```
const uint8_t n = 6;
uint8_t buff[n][14], nF = 0, nF25 = 0;
MCP2515 mcp(4);

typedef union
{
 //data formats
 long int INT;
 unsigned long int uINT;
 float FLT;

 int16_t INT16;
 uint16_t uINT16;

 int32_t INT32;
 uint32_t uINT32;

 int64_t  INT64;
```

```
 uint64_t  uINT64;

 uint8_t bytes[8];

} UNION_t;

void sendCAN(){
  static uint8_t i = 0;//, INT = 0;
  uint8_t ctr[1] = {0}, tx = 0, TX[8];
  IDunion_t sID, eID;

  if(nF25 == 2){
    mcp.read(0x30, ctr);
    if(bitRead(ctr[0], 3) == 0){
      noInterrupts();
      nF25 = 0;
      mcp.bitModify(0x2c, 0x04, 0x00);
    }
  }

  while(1){
```

```
if(nF == 0){
  return;
}

mcp.read(0x30, ctr);
if(bitRead(ctr[0], 3) == 1){
  nF25 = 2;
  interrupts();
  return;
}
mcp.bitModify(0x2c, 0x04, 0x00);

mcp.read(0x40, ctr);
if(bitRead(ctr[0], 3) == 1){
  tx = 0;
}else{
  tx = 1;
  mcp.bitModify(0x2c, 0x08, 0x00);
}
```

```
    CANframe FTS(buff[i]);
    mcp.writeFrame(FTS, tx);

    nF = nF - 1;
    i = i + 1;
    if(i == n){
      i = 0;
    }
  }
}

void putinBuff(uint16_t IDstd, uint32_t IDext, uint16_t nB, uint8_t *data){
  static uint8_t i = 0;
  uint8_t nb = 0;
  IDunion_t sID, eID;

  if(nB > 8*n){
    //nB muito grande,
    return;
  }
```

```
if(IDstd > 0x7FF or IDext > 0x3FFFF){
  //erro IDstd ou IDext invalido;
  return;
}

sID.u16 = IDstd;
eID.u32 = IDext;

uint8_t k = 0;
while(nB >= k){

  buff[i][6+nb] = data[k];
  k = k + 1;
  nb = nb +1;
  if(nb == 8 | k == nB){
    buff[i][0] = sID.bytes[1];
    buff[i][1] = sID.bytes[0];
    buff[i][2] = eID.bytes[2];
    buff[i][3] = eID.bytes[1];
    buff[i][4] = eID.bytes[0];
    buff[i][5] = nb;
```

```
      nb = 0;
      i = i + 1;
      if(i == n){
        i = 0;
      }

      nF = nF + 1;
      if(nF > n){
        nF = n;
      }
    }
  }

  if(nF25 < 2){
    sendCAN();
  }
  return;
}
```

```
void putinBuff(uint16_t IDstd, uint16_t nB, uint8_t *data){
  static uint8_t i = 0;
  uint8_t nb = 0;
  IDunion_t sID;

  if(nB > 8*n){
    //nB muito grande,
    return;
  }

  if(IDstd > 0x7FF){
    //erro IDstd invalido;
    return;
  }

  sID.u16 = IDstd;

  uint8_t k = 0;
  while(nB >= k){

    buff[i][6+nb] = data[k];
```

```
k = k + 1;
nb = nb +1;
if(nb == 8 | k == nB){
  buff[i][0] = sID.bytes[1];
  buff[i][1] = sID.bytes[0];
  buff[i][2] = 0;
  buff[i][3] = 0;
  buff[i][4] = 0;
  buff[i][5] = nb;

  nb = 0;
  i = i + 1;
  if(i == n){
    i = 0;
  }

  nF = nF + 1;
  if(nF > n){
    nF = n;
  }
}
```

```
}
        if(nF25 < 2){
            sendCAN();
        }
        return;
    }
```

B.4.7.2 CANSensor.ino

```
/// CANSensor
#include <MCP2515_1.h>
#include "CANSensor.h"

void setup() {

  mcp.CANINTE = 0x04;
  mcp.begin();

  attachInterrupt(digitalPinToInterrupt(2), sendCAN, FALLING);
}

void loop() {

  // ID declaration
  uint16_t Sid = 0x7F0; //standart
  uint32_t Eid = 0; // extend
  Sid = random(0x7FF); //Set std. ID, maximun is 0x7FF
  Eid = random(0x3FFFF); //Set ext. ID, maximun is 0x3FFFF
```

```
  // Example of data acquisition and send -----------------------------
  // Use union type to convert variable formats
  UNION_t Data;

  // Int data example
  Data.INT = analogRead(A0);

  //Send a extended frame
  putinBuff(Sid, Eid, sizeof(Data.INT), Data.bytes);

  // Float data example
  Data.FLT = analogRead(A0)*(5.0/1023.0); // convert in Volts

  // Send a standard frame
  putinBuff(Sid, sizeof(Data.FLT), Data.bytes);

  delay(1000);
}
```

Apêndice C

Exemplos de Conexão

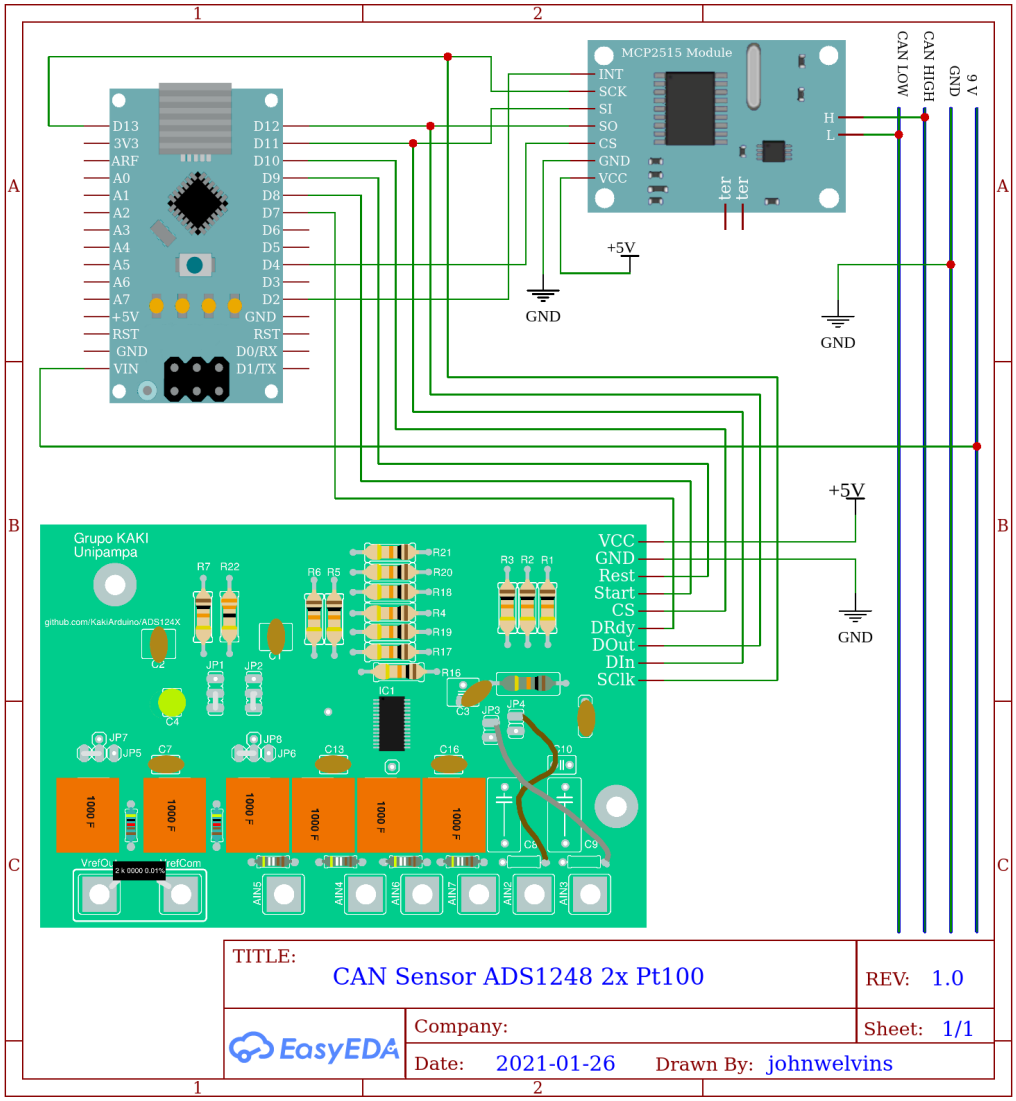


Figura C.1: Diagrama das conexões de um nodo do tipo CAN Sensor com o ADS124x, ADC da Texas Instruments. Fonte: Autores.

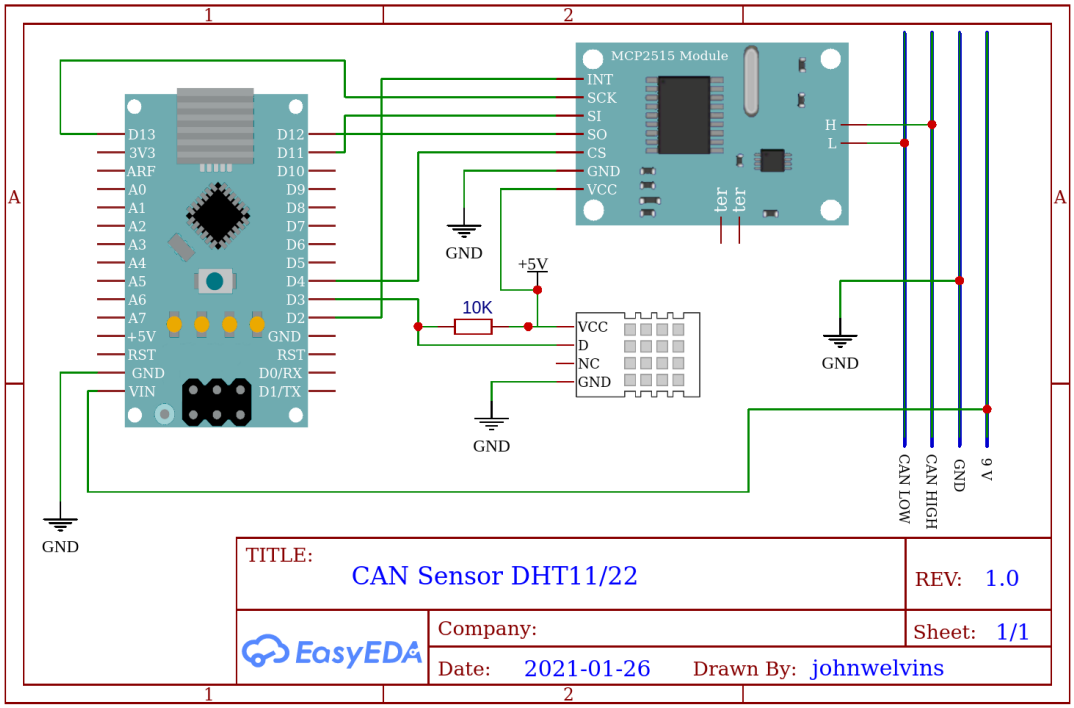


Figura C.2: Diagrama das conexões de um nodo do tipo CAN Sensor com o DHT11 ou DHT22, transdutores de umidade relativa e temperatura da Aosong. Fonte: Autores.

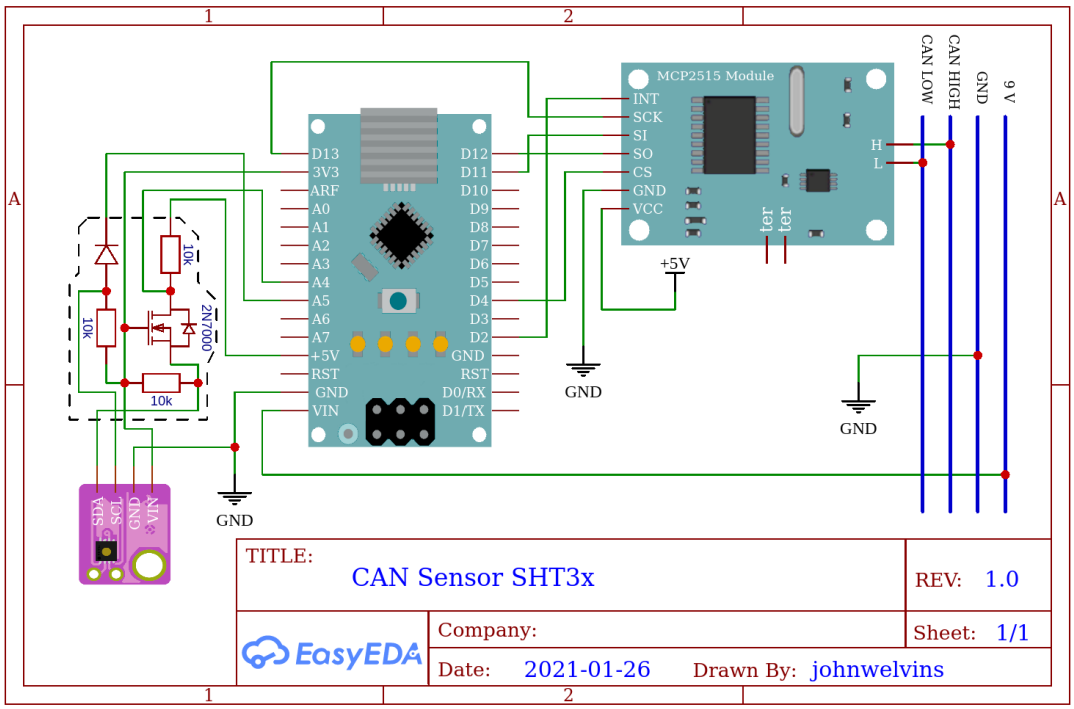


Figura C.3: Diagrama das conexões de um nodo do tipo CAN Sensor com o SHT3x, linha transdutores de umidade relativa e temperatura da Sensirion. Fonte: Autores.

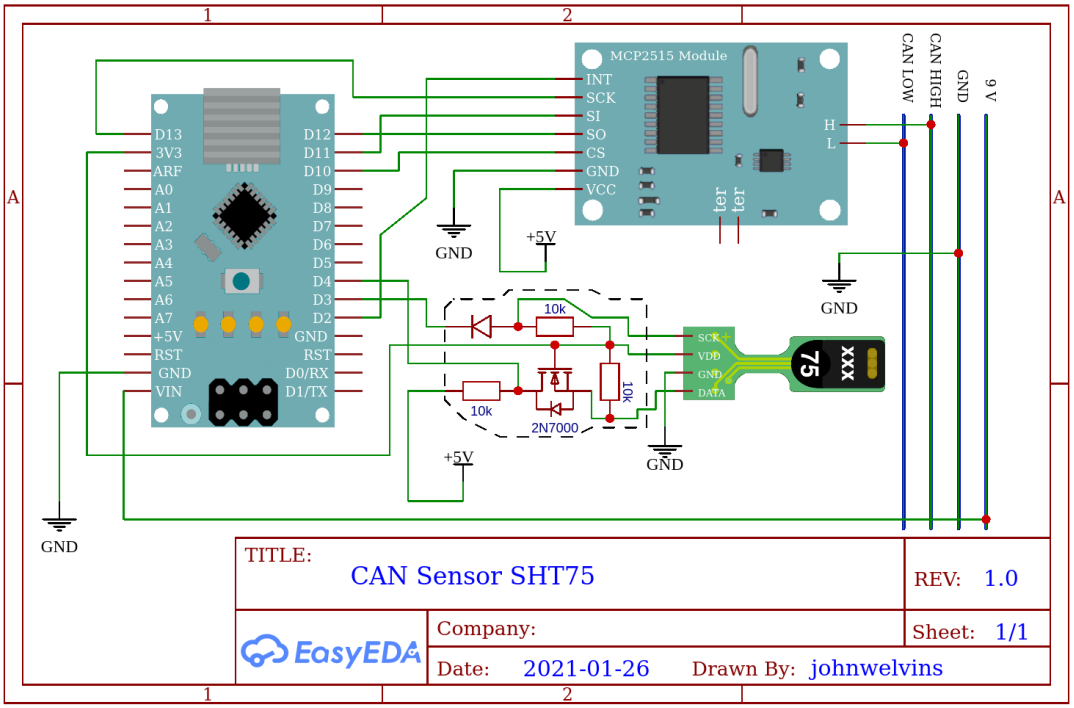


Figura C.4: Diagrama das conexões de um nodo do tipo CAN Sensor com o SHT75, transdutor de umidade relativa e temperatura da Sensirion. Fonte: Autores.

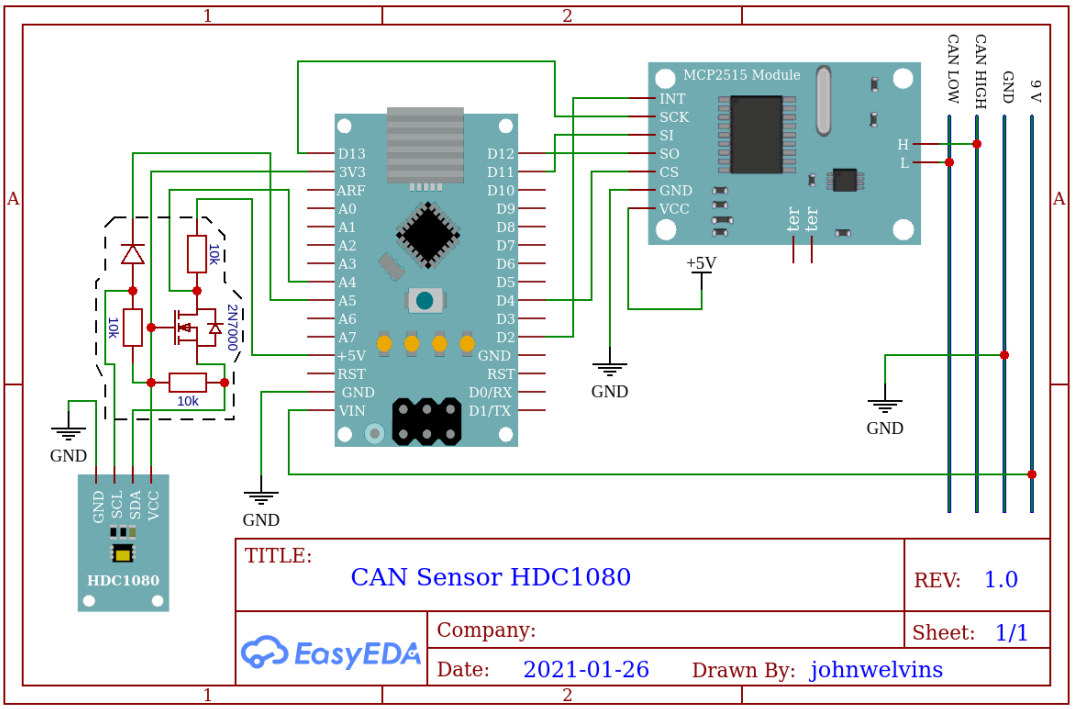


Figura C.5: Diagrama das conexões de um nodo do tipo CAN Sensor com o HDC1080, transdutor de umidade relativa e temperatura da Texas Instruments. Fonte: Autores.