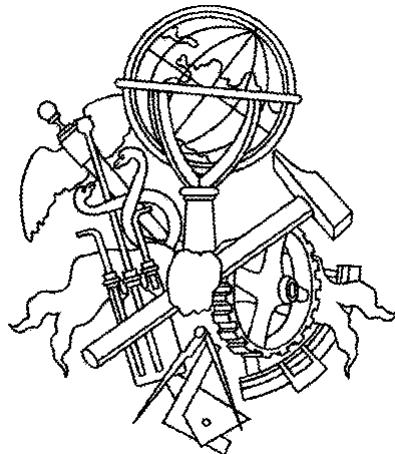


# REDE DE COMUNICAÇÃO CAN

Flávio Vasconcelos  
Tiago Ferreira



Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

2016

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Disciplina de Laboratórios de Mecatrónica do 2º ano, do Mestrado em Engenharia Electrotécnica e de Computadores

Aluno: Flávio Vasconcelos, Nº 1100419 , 1100419 @isep.ipp.pt  
Aluno: Tiago Ferreira, Nº 1100498 , 1100498 @isep.ipp.pt



Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

3 de Janeiro de 2016

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Enquadramento e Motivação . . . . .	2
1.2	Cenários de aplicação do CAN . . . . .	3
1.3	Objetivos . . . . .	4
1.4	Calendarização . . . . .	5
<b>2</b>	<b>Estado da Arte</b>	<b>7</b>
2.1	Protocolos de comunicação . . . . .	7
2.1.1	Tipos de Comunicação . . . . .	7
2.1.2	Meios de Comunicação . . . . .	11
2.1.3	Protocolos Mais Utilizados . . . . .	11
2.2	Redes de Campo . . . . .	13
2.2.1	Rede de Campo Sensor Bus – ASI . . . . .	16
2.2.2	Rede de Campo Device Bus – <i>DeviceNet</i> . . . . .	17
2.2.3	Rede de Campo Field Bus – <i>Profibus</i> . . . . .	18
<b>3</b>	<b>Protocolo CAN</b>	<b>21</b>
3.1	História . . . . .	21
3.2	Características . . . . .	23
3.3	Funcionamento . . . . .	25
3.3.1	Camada física . . . . .	26
3.3.2	Implementação em hardware . . . . .	31
3.3.3	Camada de ligação de dados . . . . .	32

3.3.4	Camada de aplicação . . . . .	40
3.4	Tecnologias CAN existentes . . . . .	40
3.4.1	Microcontrolador . . . . .	41
3.4.2	Controlador CAN . . . . .	42
3.4.3	<i>Transceiver</i> CAN . . . . .	42
3.4.4	A Escolha . . . . .	46
<b>4</b>	<b>Arquitectura Geral do Sistema</b>	<b>47</b>
4.1	Arquitetura do sistema . . . . .	47
4.2	Microcontrolador . . . . .	48
4.2.1	STM32f103 Sistema Mínimo . . . . .	49
4.2.2	Raisonance Primer 1 . . . . .	50
4.3	Configuração do periférico CAN no STM32 . . . . .	51
4.3.1	Modos de Operação . . . . .	53
4.3.2	Transmissão . . . . .	55
4.3.3	Recepção . . . . .	56
4.3.4	Filtragem . . . . .	56
4.3.5	<i>Bit Timing</i> . . . . .	57
4.3.6	Bibliotecas . . . . .	58
4.4	Sensores . . . . .	60
4.4.1	Sensor Ótico Infravermelho . . . . .	60
4.4.2	Sensor Ultrassónico - HC-SR04 . . . . .	62
4.4.3	Sensor de Luminosidade - LDR . . . . .	64
4.4.4	Acelerómetro LIS3LV02DL . . . . .	65
4.5	Outros Periféricos . . . . .	67
4.5.1	Matriz de LED's - MAX7219 . . . . .	68
4.5.2	Modulo <i>Bluetooth</i> - HC-05 . . . . .	69
4.5.3	LCD TFT - ILI9341 . . . . .	71
4.6	Teste dos Sensores . . . . .	71
4.6.1	Protocolo CAN . . . . .	71

4.6.2	Protocolo SPI . . . . .	73
<b>5</b>	<b>Arquitectura e Projecto do Sistema</b>	<b>75</b>
5.1	Descrição de <i>Hardware</i> . . . . .	76
5.1.1	Placa1 . . . . .	77
5.1.2	Placa 2 . . . . .	78
5.1.3	Placa 3 . . . . .	80
5.2	Descrição do <i>Firmware</i> . . . . .	80
5.2.1	Placa 1 . . . . .	84
5.2.2	Placa 2 . . . . .	93
5.2.3	Placa 3 . . . . .	97
5.2.4	Tama . . . . .	101
5.3	<i>Android</i> . . . . .	103
<b>6</b>	<b>Resultados</b>	<b>117</b>
<b>7</b>	<b>Conclusão e Trabalho Futuro</b>	<b>121</b>
<b>A</b>	<b>Codigos de teste</b>	<b>127</b>
A.1	Rede CAN . . . . .	127
A.2	SPI . . . . .	129
<b>B</b>	<b>Funções de Inicialização</b>	<b>133</b>
B.1	can_init . . . . .	133
B.2	sonar_init . . . . .	134
B.3	sonar . . . . .	135
B.4	adc_init . . . . .	136
B.5	max7219_init e max7219_config . . . . .	138

Esta página foi intencionalmente deixada em branco.

# Listas de Figuras

1.1	Cadeira de rodas que permite subir escadas . . . . .	3
1.2	Calendarização . . . . .	5
2.1	Exemplificativo da comunicação serie/paralelo . . . . .	8
2.2	Exemplificativo da diferença entre síncrona e assíncrona . . . . .	9
2.3	Exemplificativo da diferença entre <i>Simplex</i> , <i>Half-Duplex</i> , <i>Full-Duplex</i> . . . . .	10
2.4	Exemplo de uma trama USART [4] . . . . .	12
2.5	Exemplo de uma trama SPI [33] . . . . .	12
2.6	Exemplo de uma trama I2C [33] . . . . .	13
2.7	Diferença entre controlo centralizado e distribuído . . . . .	14
2.8	Tipo de dispositivos . . . . .	16
2.9	Comparativo de redes industriais . . . . .	16
2.10	Nível de Automação [3] . . . . .	17
2.11	Variantes do Profibus . . . . .	19
2.12	Comparativo de rede de campo ASI, <i>DeviceNet</i> , <i>Profibus</i> e CAN [12] [31] . . . . .	20
3.1	Modelo de camadas OSI e CAN. . . . .	26
3.2	Exemplo de stream de bits transmitidos, codificados em NRZ. . . . .	27
3.3	Exemplo de <i>bit stuffing</i> numa trama de bits . . . . .	28
3.4	Esquema do <i>timing</i> de um bit [5] . . . . .	28
3.5	Relação entre taxa de transferência e comprimento do barra-mento . . . . .	30

3.6	Topologia de uma rede CAN . . . . .	31
3.7	Blocos funcionais de um nó CAN . . . . .	31
3.8	Difusão de mensagens CAN [14] . . . . .	34
3.9	Sistema de anti-colisões CSMA/DCR . . . . .	36
3.10	Trama de dados [20] . . . . .	37
3.11	Trama de dados (identificador) [20] . . . . .	39
3.12	Conexão de nós CAN sem <i>transceivers</i> [30] . . . . .	43
3.13	Registo CIOCON [22] . . . . .	44
3.14	Círculo final sem <i>tansceivers</i> usando PIC18FXX8 . . . . .	44
3.15	Potencia consumida pelos CAN <i>transceivers</i> [7] . . . . .	45
3.16	Esquemas elétricos dos <i>transceivers</i> CAN . . . . .	46
4.1	Arquitetura geral do sistema . . . . .	48
4.2	STM32f103 sistema mínimo (esquerda) e placa de desenvolvimento Primer 1 (direita) . . . . .	49
4.3	Placa Sistema Mínimo . . . . .	50
4.4	Raisonance Primer 1 . . . . .	50
4.5	Esquema de ligações (Inercial, <i>buzzer</i> e LCD) da placa Raisonance Primer 1 [27] . . . . .	51
4.6	Diagrama de blocos do controlador CAN [6] . . . . .	52
4.7	Modos de operação do controlador CAN [1] . . . . .	53
4.8	Mecanismo de filtragem de mensagens CAN [1] . . . . .	58
4.9	<i>Bit Timing</i> [1] . . . . .	59
4.10	Constituição de um Sensor Ótico Infravermelho [13] . . . . .	60
4.11	Esquema de ligações do Sensor Ótico Infravermelho com saída digital [13] . . . . .	61
4.12	Modulo Sensor Ótico Infravermelho . . . . .	61
4.13	Funcionamento de um Sensor Ultrasónico . . . . .	62
4.14	Principio de funcionamento de um Sensor Ultrasónico . . . . .	63
4.15	Pinout do Sensor Ultrasónico . . . . .	64

4.16 Esquema de ligações do LDR . . . . .	65
4.17 Constituição do Sensor LIS3LV02DL [32] . . . . .	66
4.18 Funcionamento de uma trama SPI para o LIS3LV02DL . . . . .	66
4.19 Sinalizador traseiro . . . . .	68
4.20 Esquema de ligações MAX7219 . . . . .	69
4.21 Modulo <i>bluetooth</i> HC-05 . . . . .	69
4.22 <i>Plot</i> dos níveis lógicos capturados na rede CAN . . . . .	72
5.1 Vistas do prototipo funcional . . . . .	76
5.2 Esquema elétrico da Placa 1 . . . . .	77
5.3 Esquema elétrico da Placa 2 . . . . .	79
5.4 Esquema elétrico da Placa 3 . . . . .	80
5.5 Fluxograma da inicialização do CAN . . . . .	83
5.6 Fluxograma do ciclo principal da Placa 1 . . . . .	85
5.7 Ecrãs disponíveis no sistema . . . . .	86
5.8 Fluxograma da função <i>sonar_init</i> . . . . .	87
5.9 Fluxograma da função sonar . . . . .	89
5.10 <i>Bouncing</i> . . . . .	91
5.11 Circuito para <i>debouncing</i> por <i>hardware</i> . . . . .	91
5.12 Fluxograma do método de <i>debouncing</i> . . . . .	92
5.13 Fluxograma do ciclo principal da Placa 2 . . . . .	93
5.14 Fluxograma da inicialização e configuração do max7219 . . . . .	94
5.15 Fluxograma da rotina de interrupção de receção de mensagens via USART . . . . .	96
5.16 Fluxograma do ciclo principal da Placa 3 . . . . .	98
5.17 Fluxograma da função ”motor” . . . . .	100
5.18 Fluxograma da situação de emergência . . . . .	101
5.19 Fluxograma do funcionamento do Tama . . . . .	102
5.20 Ecrã do Tama . . . . .	103
5.21 HC-05 . . . . .	104

5.22	IDE Android Studio . . . . .	104
5.23	. . . . .	105
5.24	Ecrã de comando . . . . .	110
5.25	8 possíveis direções de controlo . . . . .	114
6.1	Prototipo funcional . . . . .	117

# **Lista de Tabelas**

3.1	Compatibilidades das versões CAN . . . . .	33
4.1	Lista de comandos AT . . . . .	70
5.1	Tabela de fluxo de mensagens CAN . . . . .	84
5.2	Tabela da verdade para controlo dos motores . . . . .	99

Esta página foi intencionalmente deixada em branco.

# Lista de Acrónimos

**ABS** *Anti-lock Braking System*

**ACK** *Acknowledge*

**ARM** *Acorn RISC Machine*

**CAN** *Controller Area Network*

**CiA** *CAN in Automation*

**CIP** *Common Application Layer*

**CPU** *Central Processing Unit*

**CRC** *Cyclic Redundancy Check*

**CS** *Chip Select*

**CSMA** *Carrier Sense Multiple Access*

**DCR** *Deterministic Collision Resolution*

**DLL** *Data Link Layer*

**ECU** *Engine Control Unit*

**EoF** *End of Frame*

**ESD** *Electrostatic Discharge*

**FIFO** *First In First Out*

**FPGA** *Field-Programmable Gate Array*

**GPS** *Global Positioning System*

**I2C** *Inter-Integrated Circuit*

**IDE** *Identifier Extension*

**ISO** *International Organization for Standardization*

**LAMEC** *Laboratórios de Mecatrónica*

**LCD** *Liquid Crystal Display*

**LDR** *Light Dependent Resistor*

**LLC** *Logical Link Control*

**MAC** *Medium Access Control*

**MDI** *Medium Dependent Interface*

**NiMH** *Nickel–Metal Hydride Battery*

**NRZ** *Non Return to Zero*

**OSI** *Open Systems Interconnection model*

**PCB** *Printed circuit board*

**PLC** *Programmable Logic Controller*

**PLS** *Physical signaling*

**PMA** *Physical Medium Attachment*

**RJW** *Resyncronization Jump Width*

**RTR** *Remote Transmission Request*

**RST** *Reset*

**SD** *Secure Digital*

**SoF** *Start of Frame*

**SPI** *Serial Peripheral Interface*

**SMD** *Surface Mount Device*

**SRAM** *Static Random Access Memory*

**SRR** *Substitute Remote Request*

**SWD** *Serial Wire Debug*

**TFT** *Thin-Film Transistor*

**USART** *Universal Synchronous Asynchronous Receiver Transmitter*

**USB** *Universal Serial Bus*

Esta página foi intencionalmente deixada em branco.

# Capítulo 1

## Introdução

No âmbito da unidade curricular de LAMEC (Laboratórios de Mecatrónica) presente no 2º ano do mestrado em Engenharia Eletrotécnica e de Computadores (MEEC) foram analisadas as diferentes propostas de trabalho. Após, feita a escolha, optou-se por realizar um projeto que fizesse uso de uma rede com protocolo CAN (*Controller Area Network*) a ser implementada por microcontroladores.

O principal objetivo do projeto é o estudo e a apresentação dos conceitos do protocolo CAN. Um dos requisitos para a implementação da rede CAN é ter no mínimo 3 nós, para que assim possa ser mais evidente as vantagens do uso do protocolo CAN.

Atualmente no sector automóvel este tipo de comunicação é a mais utilizada pois assim é possível ligar um numero bastante elevado de dispositivos à rede. Além desta característica é possível ter uma rede com 1Mbit/s de *Bit rate* com um comprimento até 25 metros. Se mesmo assim for necessário aumentar a comprimento da rede, o *Bit rate* desce significativamente. Com estas características que CAN dispõe, não só a área automóvel utiliza CAN mas sim outras áreas começaram por implementar CAN, atualmente desde a medicina até ao desporto são áreas onde a comunicação CAN já não é novidade.

## 1.1 Enquadramento e Motivação

Acreditamos que hoje em dia o objetivo dos sistemas atualmente desenvolvidos é o auxílio nas diversas tarefas diárias, e quando se fala em pessoas com algum tipo de incapacidade estes sistemas são ainda mais importantes. A cadeira de rodas é um objeto indispensável para pessoas que apresentam dificuldade de locomoção, existem vários tipos de cadeiras de rodas, desde as mais tradicionais até às mais modernas. Um grupo de estudantes Suíços desenvolveu agora uma cadeira de rodas elétrica que traz inovação na conquista dos obstáculos arquitetónicos. A cadeira de rodas desenvolvida pelos estudantes foi concebida para trazer mais mobilidade a quem tem uma deficiência física e necessite de um equipamento destes. A cadeira é especial pela tecnologia que incorpora pois dá uma "liberdade" de movimentos substancialmente superior a tudo o que existe no mercado. Usada de forma "normal" em planos horizontais, a cadeira está assente em duas rodas, tal como se fosse uma *Segway* e permite aos utilizadores movimentos rápidos de mudança direção e com a facilidade de o fazer em rotação. Duas lagartas de borracha colocadas por baixo da cadeira podem ser ativadas, quando o utilizador pressiona um botão para a cadeira de rodas subir as escadas. É possível ver na figura 1.1 a sua estrutura mecânica. Depois de 10 meses de construção deste protótipo, com intenção de o colocar no mercado, o grupo de estudantes acredita que a maior valia desta inovação é mesmo a tecnologia estar totalmente posicionada a ajudar de forma prática e segura quem está agarrado a uma cadeira de rodas, tornando o utilizador muito mais autónomo em qualquer rua, edifício ou zona de uma qualquer cidade. A cadeira foi testada em vários tipos de escadas, até mesmo nas escadas em espiral ou caracol e todas elas foram subidas com sucesso. [29]

A nossa missão passa por desenvolver um sistema que consiga ajudar estas pessoas a um mínimo custo possível, pois as cadeiras de rodas com algum tipo de tecnologia são extremamente dispendiosas. Para isso vamos



Figura 1.1: Cadeira de rodas que permite subir escadas

recorrer a material de baixo custo porem com alguma fiabilidade. Hoje em dia o uso de *smartphones* é cada vez mais, assim o nosso sistema contem uma aplicação para o sistema móvel *android* para que haja uma interação entre o utilizador e a cadeira de rodas.

## 1.2 Cenários de aplicação do CAN

Redes CAN são utilizadas em diversas aplicações. Foi inicialmente desenvolvido para aplicações em automóveis pois é uma comunicação serie, robusta, *multi-master*, de modo conseguir proporcionar automóveis mais confiáveis, seguros e eficientes e com diagnóstico de falhas. Por exemplo a comunicação entre o ABS (*Anti-lock Braking System*) e a ECU (*Engine Control Unit*). Esta é a mais comum aplicação pese embora CAN vai além da área automóvel, outras industrias sabem as vantagens da utilização da rede CAN e atualmente transportes ferroviários já usam este tipo de comunicação.

CAN é também utilizado na aviação desde o controlo dos *flaps*, sistemas de navegação, até ao *cockpit*. Na área da medicina também é utilizado este protocolo de comunicação. No caso dos hospitais, CAN é utilizado em os vários equipamentos, luzes, camas, sistemas de vigilância, controlo de acessos, maquinas de raio-X, todos estes equipamentos entre outros sistemas usufruem das vantagens do CAN. Elevadores e escadas rolantes são outras aplicações que usam CAN.[17]

A utilização de CAN está onde menos possamos imaginar, uma grande prova disso é no ciclismo. Tudo na vida está em constante evolução, as bicicletas não são exceção, atualmente já existe sistemas de troca de mudanças elétrico. Falando em particularmente no modelo Di2 da *Shimano*, a ultima geração já possibilita ter esta tecnologia sem fios, mas nas primeiras versões da Di2 este utilizava protocolo CAN para fazer a ligação dos *shifters* aos desviadores. Outra aplicação ainda no ramo do ciclismo é *BionX*, muito sucintamente é um sistema que permite adaptar uma tradicional bicicleta numa bicicleta elétrica também utilizando o protocolo CAN pois é fiável permite ligar n dispositivos um numero reduzido de fios condutores o que numa bicicleta é bastante vantajoso.

### 1.3 Objetivos

O principal objetivo do projeto é o estudo e a apresentação dos conceitos do protocolo CAN, no caso será um sistema automatize uma cadeira de rodas tornando a vida de um utilizador de cadeira de rodas mais segura.

Primeiramente irá se fazer um estudo do protocolo CAN bem como as suas aplicações, vantagens, desvantagens e alternativas. Posteriormente irá se desenvolver a rede propriamente dita. A rede CAN terá 4 nós. Cada nó terá agregado um microcontrolador de arquitetura ARM. Vão ser utilizados alguns sensores e atuadores ligados aos microcontroladores de modo a que os dados dos sensores possam ser levados para o barramento CAN e que possam também ser posteriormente processados por outro nó da rede com o objetivo de exemplificar o uso e a implementação do protocolo.

Temos como objetivo desenvolver uma aplicação para dispositivos *android* de modo a fazer a ponte entre homem e a maquina.

Não menos importante o custo, um dos objetivos é reutilizar materiais que ambos os elementos do grupo possuam, não descartando a compra de novo material mas tendo sempre em conta o custo/beneficio que irá acres-

centar ao projeto

## 1.4 Calendarização

Tendo em conta o calendário deste corrente ano letivo, é apresentado a calendarização na seguinte figura 1.2

Seguidamente são apresentadas um conjunto de tarefas que irão ser desenvolvidas. Apesar desta calendarização, existe a possibilidade das tarefas sofrerem uma reorganização mediante a sua conclusão antes ou depois do previsto.

Tarefa	Duração	Setembro			Outubro			Novembro			Dezembro			Janeiro			
		13/set	20/set	27/set	04/out	11/out	18/out	25/out	01/nov	08/nov	15/nov	22/nov	29/nov	06/dez	13/dez	20/dez	27/dez
Escolha do tema de trabalho	1 sem.																
Pesquisa e estudo sobre o protocolo CAN	3 sem.																
Escolha do micro controlador a utilizar	1 sem.																
Entrega do 1º relatório (Relatório formulação do problema e análise de requisitos)																	
Desenvolvimento do código para o micro processador	3 sem.																
Análise da arquitetura do sistema do projecto	2 sem.																
Entrega do 2º relatório (Relatório projecto preliminar)																	
Implementação da rede CAN em breadboard e teste	5 sem.																
Interligação de sensores e actuadores	3 sem.																
Entrega do 3º relatório (Relatório de protótipo funcional)																	
Implementação de uma aplicação usando rede CAN	4 sem.																
Teste do protótipo desenvolvido	2 sem.																
Entrega do 4º relatório (Relatório Final)																	

Figura 1.2: Calendarização

Esta página foi intencionalmente deixada em branco.

# **Capítulo 2**

## **Estado da Arte**

O estudo de uma rede CAN implica, pelo menos indiretamente, um olhar sobre os outros protocolos de comunicação. Então seguidamente é apresentado uma breve análise sobre os protocolos de comunicação e as redes de campo.

### **2.1 Protocolos de comunicação**

Uma rede de dados está diretamente ligada a uma comunicação de informações, e esta fundamentação teórica também é verdadeira para uma rede de dispositivos eletrônicos, daí existir a necessidade de se criar normas e protocolos devidamente estruturados para a comunicação de dados entre sistemas eletrônicos.[8]

#### **2.1.1 Tipos de Comunicação**

A comunicação de dados entre dispositivos eletrônicos pode ser classificada em três aspectos:

- Ao número de vias da transmissão;
- À cadência de transmissão;

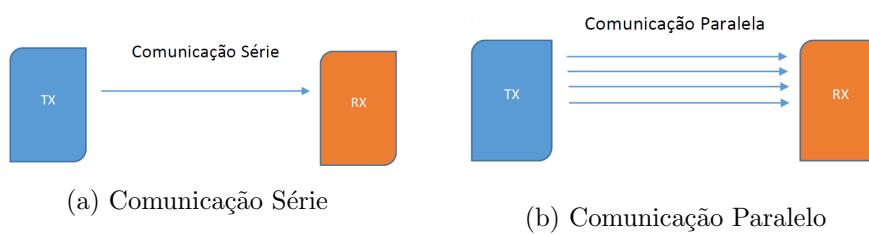
- Ao sentido da transmissão.

### Vias de transmissão

Face ao número de vias da transmissão esta pode ser de dois tipos: Série ou Paralela.

A comunicação Série caracteriza-se por uma linha de dados entre o transmissor e o receptor onde os bits são enviados um a um em sequência, e por essa razão é um tipo de comunicação com uma taxa de transferência de dados baixa em relação à Paralela. Existem diversos protocolos que fazem uso deste tipo de comunicação, e dentre eles, pode-se destacar o RS232, USB (*Universal Serial Bus*), CAN, SPI (*Serial Peripheral Interface*) e I2C (*Inter-Integrated Circuit*).

A Comunicação Paralela apresenta múltiplas linhas de dados, o que faz com que os bits sejam enviados simultaneamente entre o transmissor e o receptor, daí apresentar uma taxa de transferência de dados muito superior, comparativamente à Série. Este tipo de comunicação é utilizada no interior dos computadores, impressoras, FPGA (*Field-Programmable Gate Array*) entre outros. A figura 2.1 tem como intuito de demonstrar as diferenças entre a comunicação serie (2.1a) e a comunicação paralelo (2.1b).



(a) Comunicação Série

(b) Comunicação Paralelo

Figura 2.1: Exemplificativo da comunicação serie/paralelo

### Sincronismo na transmissão

Relativamente ao sincronismo na transmissão, os protocolos podem utilizar uma metodologia síncrona ou assíncrona.

Na transmissão de dados assíncrona não existe um sinal de relógio logo é necessário mecanismos, geralmente *flags*, que são entregues ao receptor onde informam a este quando começa e quando acaba o bloco de dados a ser transmitido, entre outras informações importantes. Com esta funcionalidade, este tipo de transmissão de dados faz com que a ordem seja qualquer uma e apenas cabe ao receptor fazer a interpretação das informações recebidas.

Já a transmissão de dados síncrona é comandada por um sinal de relógio e parte de um pressuposto diferente, ou seja, o emissor e o receptor devem estar num mesmo instante de amostragem antes da transmissão de dados começar e permanecer neste estado durante toda a transmissão. Neste tipo de comunicação cada bloco de dados é transmitido e recebido num instante de tempo definido, isto é, quando a informação é transferida, o receptor é bloqueado e só pode enviar outro bloco quando o primeiro for recebido pelo receptor. Os protocolos SPI e I2C fazem uso deste tipo de transmissão de dados.

Na figura 2.2 é possível verificar as diferenças entre a transmissão entre síncrona (2.2a) e assíncrona (2.2b).

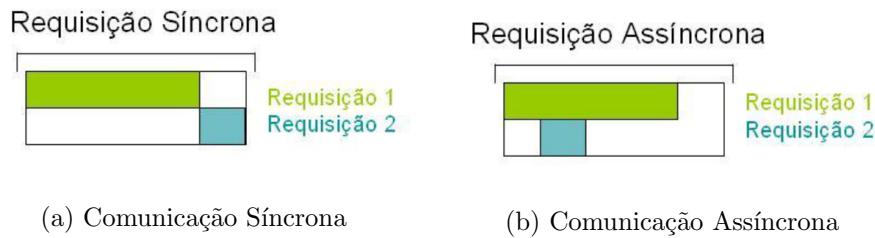


Figura 2.2: Exemplificativo da diferença entre síncrona e assíncrona

### Sentido da transmissão

O sentido da comunicação de dados também é um aspecto importante e este pode ser dividido em três tipos (figura 2.3):

- *Simplex* (figura 2.3a)
- *Half-Duplex* (figura 2.3b)
- *Full-Duplex* (figura 2.3c)

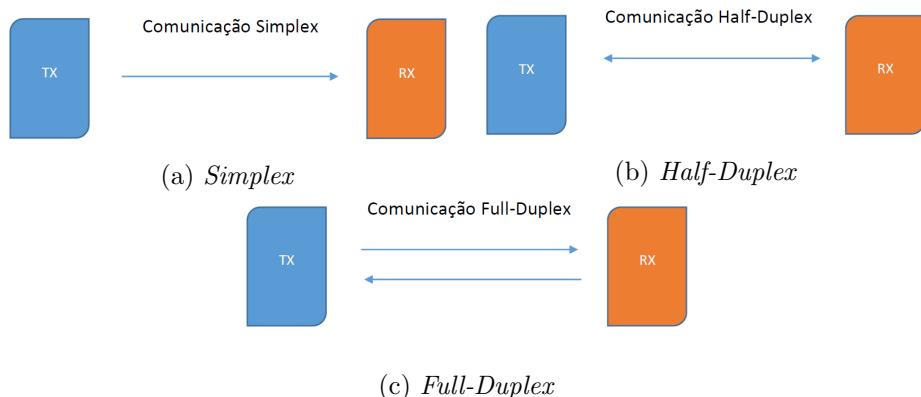


Figura 2.3: Exemplificativo da diferença entre *Simplex*, *Half-Duplex*, *Full-Duplex*

Uma comunicação *Simplex* caracteriza-se pelo sentido da transmissão de informação ser do transmissor para o recetor, não podendo em alguma situação algum inverter o seu papel, isto é, o fluxo de informação é unidirecional.

Já no caso da comunicação *Half-Duplex*, o fluxo de informação tem um sentido bidirecional, isto é, pode ser do transmissor para o recetor ou do recetor para o transmissor, embora a particularidade seja que a transmissão de dados nunca pode ser feita simultaneamente por ambos, isto é, quando um envia informação, o outro espera pela receção dessa informação e só depois pode enviar a informação dele.

Em sistemas de complexidade mais elevada existem comunicações *Full-Duplex* que se caracterizam por uma transmissão de dados bidirecional simultânea, desta forma, a comunicação entre o transmissor e o receptor pode ser feita em qualquer sentido e em qualquer instante. Este tipo de comunicação traz como grande vantagem a rapidez na transmissão de dados.

### 2.1.2 Meios de Comunicação

Em comunicações de rede, o meio de comunicação da informação entre os dispositivos eletrónicos também é um fator importante aquando a concessão do projeto ou implementação do produto final. E este pode ser efetuado de duas formas: através do uso de cabos ou fios em placas de circuito (*wired*) ou através de tecnologias sem fios (*wireless*).

O uso de cabos que fazem a conexão aos pinos dos diferentes módulos eletrónicos acaba por ser a abordagem mais tradicional, e neste ramo ainda é muito usada para a criação de redes com mais fiabilidade e segurança. Atualmente o uso de tecnologias *wireless* também tem tido bastante uso em projetos de sistemas eletrónicos, e de todas as existentes pode-se destacar quatro tecnologias: Infravermelho, Rádio Frequência, *Wifi* e *Bluetooth*.

### 2.1.3 Protocolos Mais Utilizados

De todos os protocolos de comunicação que existem em eletrónica, e mais especificamente, nos microcontroladores, os principais são:

- USART (*Universal Synchronous Asynchronous Receiver Transmitter*)-  
Este protocolo fornece a um computador a interface necessária para comunicação com modems e outros dispositivos série. É um protocolo para comunicação de dados série, em modo *full-duplex*, e utiliza dois pinos (TX e RX) para transmitir dados. Fornece uma cadência de informação que pode ser síncrona ou assíncrona dependendo do que seja configurado. As interfaces de comunicação série são mais baratas,

fáceis de utilizar e muito comuns no ramo. Na figura 2.4 é apresentado uma trama de bits provenientes da USART onde é possível ver o conteúdo da trama, tem se o *start bit* que indica o inicio da transmissão, seguido dos dados, posteriormente é enviado o bit de paridade e por fim o *stop bit*.

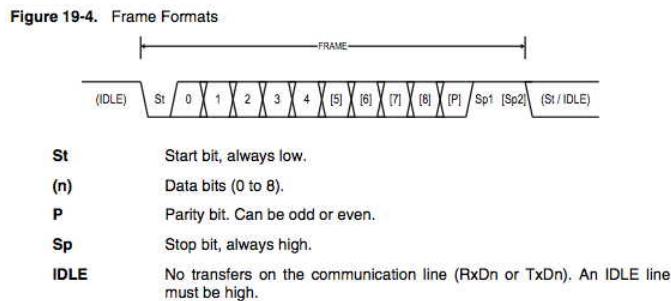


Figura 2.4: Exemplo de uma trama USART [4]

- SPI - O protocolo SPI é um protocolo de comunicação de dados série síncrono usado em comunicações de curta distância, maioritariamente em sistemas embebidos. Comunica em modo *full-duplex* usando um arquitetura *master-slave* com *single-master*. O dispositivo mestre origina a trama de leitura e escrita e o pino de *slave select* (SS) é o responsável pela escolha de cada dispositivo escravo. Na figura 2.5 é apresentado uma trama de bits provenientes de SPI.

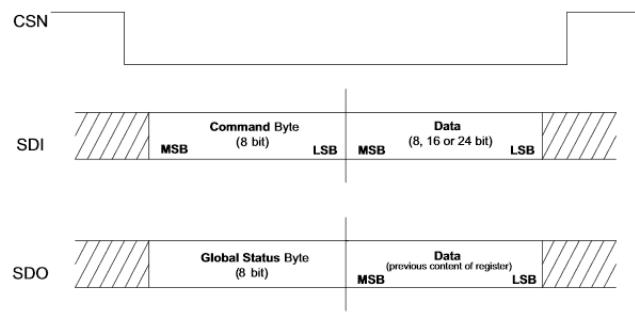


Figura 2.5: Exemplo de uma trama SPI [33]

- I2C - O I2C é um protocolo de dados série utilizado para comunicação de periféricos de baixa velocidade maioritariamente para sistemas embedidos. Utiliza apenas duas linhas bidireccionais de dreno aberto: Dados Série (*Serial Data* – SDA) e Sinal de Relógio Série (*Serial Clock* – SCL), especifica assim dois sinais de comunicação, um com sinal de relógio e outro de dados, bidirecional. Uma trama de dados enviada por este protocolo pode ser vista na figura 2.6:

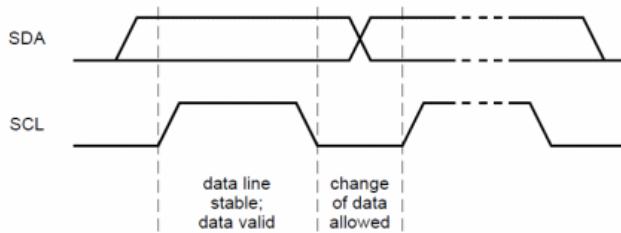


Figura 2.6: Exemplo de uma trama I2C [33]

- CAN - Devido ao facto do tema principal deste relatório ser o protocolo CAN, a explicação deste será feita posteriormente.

## 2.2 Redes de Campo

As redes de campo caracterizam-se pelo facto de serem redes utilizadas localmente para a interligação de dispositivos de controlo, tais como sensores e atuadores, e sistemas de controlo em instalações industriais. A ideia subjacente na conceção deste tipo de redes foi a substituição das tradicionais ligações ponto a ponto por um meio de comunicação partilhado, trazendo como vantagem o aumento da flexibilidade do sistema, assim como a menor complexidade na distribuição da inteligência nas cablagens do mesmo. Após a criação destas redes o sistema industrial passou de uma arquitetura centralizada para uma arquitetura distribuída, é possível ver na figura 2.7 um diagrama exemplificativo da diferença entre controlo centralizado e

distribuído.

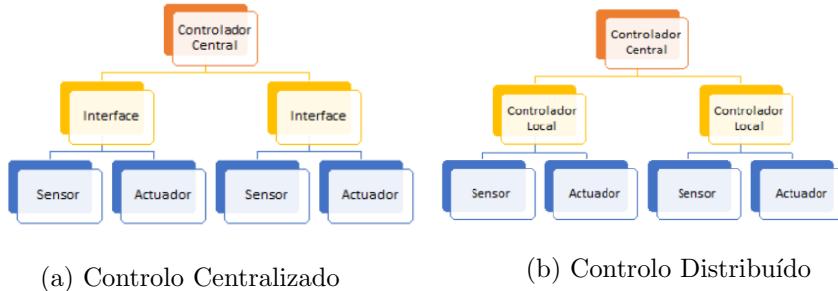


Figura 2.7: Diferença entre controlo centralizado e distribuído

No modelo de controlo centralizado existe um componente que é designado o controlador central e todos os aspectos relativos do sistema estão concentrados neste componente, ou seja, este é responsável pela gestão e envio de todos os comandos para os componentes de baixo-nível do sistema, e este tipo de gestão dos comandos pode ser feita de forma sequencial ou em paralelo[26].

Já o controlo descentralizado, ou distribuído, usa uma metodologia diferente: o tratamento de informações complexas resulta de operações dos componentes de baixo-nível do sistema, e não de instruções de nenhuma unidade central. Neste tipo de controlo todos os componentes do sistema tem uma responsabilidade igual na concretização do objetivo global[26].

Não existe uma abordagem ideal para a implementação de um sistema, pois tudo depende da aplicação que se pretende, daí existem redes industriais onde o controlo distribuído é mais eficiente, mas no caso de sistema de tempo real a abordagem centralizada é a mais indicada, na maioria dos casos.

As redes de campo preenchem uma série de características que fazem delas uma ótima escolha para implementar num meio industrial como o custo reduzido, a instalação e manutenção simples, tempos de resposta curtos, determinísticos e com reduzida variação, flexibilidade, fiabilidade e segurança nos dados. Atualmente existe um grande número de tipos de redes de

campo, com características distintas, e das quais se pode listar as seguintes:  
[26]

- CAN
- *Profibus* (Process Field Bus)
- *Fieldbus*
- ASI (Actuator Sensor Interface)
- *Modbus*
- *DeviceNet*
- *ControlNet*

A lista é muito superior á apresentada e a escolha de cada um destes tipos de rede de campo depende essencialmente de fatores como os tipos de sensores presentes, nível de complexidade do sistema, distância dos baramentos, entre outros. O diagrama na figura 2.8 ilustra como os tipos de dispositivos são organizados nas redes de campo.

É de salientar que a partir do diagrama representado na figura 2.8 se pode observar que á medida que o controlo passa para um nível hierárquico mais acima no sistema, os tipos de dispositivos a tratar e, consequentemente, o tipo de rede de campo que trata cada um dos dispositivos tornam-se cada vez mais complexos. Na figura 2.9 é possivel ver uma comparaçao entre as varias redes industriais, *Sensorbus* *Devicebus* e *Fieldbus*

Devido á panóplia de redes de campo existentes no mercado e á massificação das mesmas no meio industrial, a faixa de aplicações da maioria destas redes apresenta-se na figura 2.10.

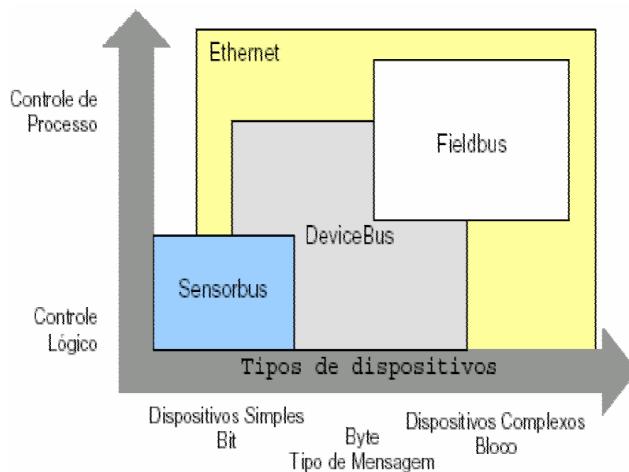


Figura 2.8: Tipo de dispositivos

	Sensorbus	Devicebus	Fieldbus
<b>Complexidade</b>	Pequena	Média	Grande
Sistema de Controlo Típico	PLC	PLC	DCS
Tamanho de dados	Menos de 1 byte	Até 32 bytes	Até 1000 bytes
Baseado em Microprocessadores	Não	Sim	Sim
Tempo de Resposta	5 ms ou menos	5 ms ou menos	100 ms
Distância dos barramentos	Pequena	Grande	Grande
Dispositivos Exemplo	Sensores	Sensores Inteligentes	Válvula inteligente com PID
Diagnóstico de Erros	Não	Simples	Sofisticado
Redes de campo usadas	CAN, ASI, LON, Seriplex	DeviceNet, Profibus DP, FIPIO, SDS	Fieldbus Foundation, Profibus PA, World FIP

Figura 2.9: Comparativo de redes industriais

### 2.2.1 Rede de Campo Sensor Bus – ASI

Atuador Sensor Interface, ou ASI, é uma rede de campo utilizada no nível mais baixo da hierarquia de redes industriais, e é utilizado em dispositivos como PLC's (*Programmable Logic Controller*), sensores e sistemas automatizados baseados em PC's. A ASI foi desenvolvida para fazer a conexão de dispositivo I/O simples em processos industriais simples.

Esta rede criada no início dos anos 90 por um grupo de empresas conhecidas pela sua oferta industrial em dispositivo sensoriais como sensores indutivos, capacitivos e ultrassónicos.

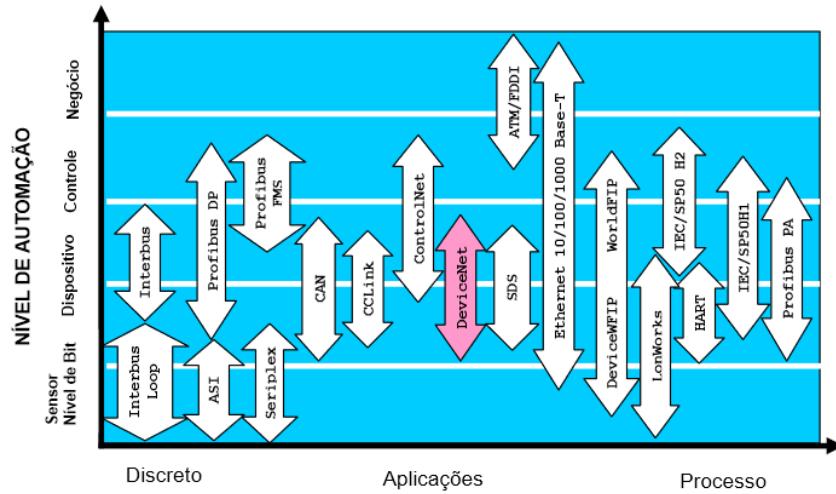


Figura 2.10: Nível de Automação [3]

As características desta rede de campo são as seguintes:

- Grande flexibilidade de topologias de rede, facilitando a instalação de cabos.
- Sistema *single-master*.
- Tempos de resposta de 5ms.
- Conexão de até 124 sensores e 124 atuadores binário para modo standard.
- Distância dos barramentos no máximo de 100 m.
- Deteção de erros na transmissão e supervisão do correto funcionamento dos escravos por parte do mestre da rede.

### 2.2.2 Rede de Campo Device Bus – *DeviceNet*

A rede *DeviceNet* classifica-se como ao nível *device bus*, sendo utilizada para a interligação de dispositivos como sensores e atuadores. Foi desenvolvida

pela *Allen Bradley* e uma das características particulares foi ter sido desenvolvido sobre o protocolo CAN, tem uma especificação aberta e gerida pela *DeviceNet Foundation*. Esta rede de campo utiliza um mecanismo de comunicação *peer-to-peer* com prioridade, permite a conexão de até 64 nós, comunicação até 500 kbit/s (máximo 100m), 250 kbit/s (máximo 250m) ou 125 kbit/s (máximo 500m) [3] [11].

Devido ao facto de ter sido criada sobre outro protocolo, a transferência de dados realiza-se bit a bit e herda funcionalidades do CAN. Utiliza uma camada de aplicação padronizada – *Common Application Layer* (CIP) que tem como características a abstração da camada física e da camada de ligação de dados, o que permite a conexão universal de componentes, desde o nível mais baixo até ao nível mais complexo [11].

### 2.2.3 Rede de Campo *Field Bus – Profibus*

O *Process Field Bus* é uma norma de comunicação ao nível do *field bus* usado em automação industrial usado pela primeira vez em 1989 pela Siemens. É um protocolo de comunicação aberto baseado em padrões e utiliza o modelo de referência OSI de 3 camadas.

No meio físico, utiliza-se um cabo de par entrançado blindado, por convenção o RS-485, ou fibra ótica, dependendo do tipo de aplicações que se pretende. No caso da fibra ótica, já que o custo é elevado na sua instalação, este só se justifica em aplicações que necessitem de grandes distâncias de comunicação e imunidade ao ruído.

O controlo de acesso ao meio é feito através de uma tecnologia denominada *token bus*, ou seja, por passagem de testemunho entre mestre e fazendo um ciclo de *pollings* aos escravos. O Profibus pode atingir velocidades de comunicação num intervalo de 9.6kbit/s até aos 12Mbit/s, com distâncias entre os 1200m e os 100m, respetivamente. A arquitetura desta rede de campo pode ser dividida em três variantes(figura 2.11 )

- Profibus DP
- Profibus FMS
- Profibus PA

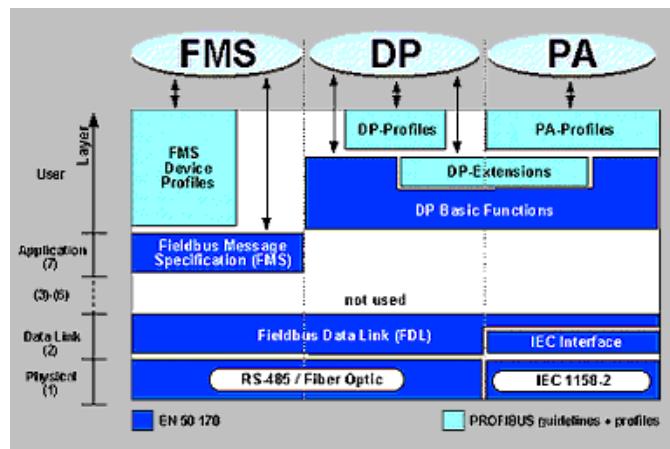


Figura 2.11: Variantes do Profibus

O Profibus DP é a variante que se destaca pela maior velocidade de comunicação das três, foi desenvolvido e otimizado para comunicações entre sistemas descentralizados e é utilizado, na atualidade, em 90% de aplicações que fazem uso do Profibus.

O Profibus FMS é uma variante que fornece as funções necessárias para a resolução de tarefas complexas de comunicação entre sistemas de automação e unidades inteligentes, sendo por isso, uma rede de campo utilizada maioritariamente em níveis de controlo.

Por último o Profibus PA é um tipo de Profibus que tem como funções a automação e controlo de processos, fazendo a comunicação de equipamentos como sensores e conversores.

Por fim, e de maneira a sistematizar o tema de Redes de campo é apresentada na figura 2.12 uma possível comparação entre as diferentes redes de campos descritas neste relatório.

	<b>ASI</b>	<b>DeviceNet</b>	<b>Profibus</b>	<b>CAN</b>
Número de dispositivos	1 master e até 62 slaves	64	32	30
Largura de Barramento	100-300 m	500 m	800 m	25-1000 m
Taxa de Comunicação de dados (máx)	167kbit/s	500kbit/s	12Mbit/s	1Mbit/s
Número de Bits Identificador	5	11	7	11(Standard) 29(Extendido)
Tecnologia Multi-Mestre	Não	Sim	Sim	Sim
Aplicações Usadas	Sensores binários Actuadores Analógicos	Aplicações de Robótica	Comunicação Inter-PLC	Sensores e Actuadores Inteligentes

Figura 2.12: Comparativo de rede de campo ASI, *DeviceNet*, *Profibus* e CAN [12] [31]

## Capítulo 3

# Protocolo CAN

O protocolo CAN, é um barramento standard desenvolvido para permitir a micro controladores e outros dispositivos comunicar entre eles em aplicações sem um módulo “mestre”, isto é, neste tipo de protocolo não existe a teoria de *master-slave* criada na maioria de comunicações, e devido a esta e outras características, o CAN é um tipo de comunicação segura e robusta.

Em sistemas que fazem uso deste protocolo as informações são transmitidas em tempo real, e daí surge a necessidade de um controlo mais rígido em relação a erros nas mensagens e na sua receção, isto é, a eficácia do protocolo deve ser tal que a troca de mensagens entre os dispositivos pertencentes á rede nunca deve falhar. As mensagens que percorrem os nós de uma rede deste tipo são geradas por *broadcast*, e fazem uso do conceito de comunicação *broadcasting multi-master*.

### 3.1 História

O Aparecimento deste protocolo de comunicação deve-se á empresa alemã BOSCH, quando em meados dos anos 80 apresentou o CAN pela primeira vez num congresso da indústria automóvel em Detroit, de maneira a dar solução a um problema que envolvia o facto dos barramento série existentes

nessa época não serem capazes de responder a todos as normas que estavam em uso nos automóveis de passageiros. O princípio do fundamento na criação do CAN foi desenvolver um sistema de comunicação entre diferentes unidades de controlo eletrónicas, sendo que nenhum dos protocolos de comunicação existentes na altura correspondia aos parâmetros e especificações de velocidade e fiabilidade, daí surgir necessidade de criar o próprio protocolo [17].

Embora o CAN tenha sido desenvolvido originalmente para ser usado em veículos de passageiros, as primeiras aplicações com este protocolo surgiram de outros segmentos do mercado, tal como na Europa, onde este protocolo foi bastante popular no sector têxtil e no controlo máquinas.

Já em 1992, Holger Zeltwanger conseguiu juntar empresas e utilizadores deste protocolo para criar a associação internacional CiA (*CAN in Automation*), sendo uma das primeiras tarefas desta associação a criação de uma especificação para a camada de aplicação do CAN [2].

Desde esta data todos os vendedores de materiais semicondutores que tinham implementado módulos CAN nos seus dispositivos estavam mais focadas na indústria automóvel, tal como a Siemens e a Motorola que enviaram uma grande quantidade de controladores CAN para as fábricas do sector automóvel da Europa. O ramo automóvel cresceu, e já em 1992, a Mercedes-Benz começou a usar CAN nos seus veículos de passageiros de gama-alta.

Em 1995, a associação CiA lançou o protocolo de comunicação CANopen, onde são definidos *frameworks* para sistemas programáveis assim como diferentes dispositivos, interfaces e perfis de aplicações. Este acaba por ser o aspeto mais importante para toda a indústria decidir utilizar o CANopen durante os anos 90 [2].

De maneira a criar uma melhor contextualização em relação ao tema, é em seguida apresentada uma lista organizada cronologicamente com datas

importantes relativas a este protocolo [2]:

- 1983: Desenvolvimento do primeiro projeto de um Barramento CAN na BOSCH;
- 1986: Apresentação e demonstração do protocolo CAN;
- 1987: Primeira comercialização de controladores com CAN;
- 1991: Publicação da norma e especificação CAN 2.0A;
- 1992: O CAN começa a ser utilizado nos automóveis da Mercedes-Benz;
- 1993: Padrão ISSO 11898 criado;
- 1995: Publicação do protocolo CANopen pela CiA;
- Atualidade: O sector automóvel adota por completo o protocolo CAN nos seus veículos;

Conclui-se que este protocolo embora tenha muito historial na sua evolução ainda necessite de certas melhorias, tendo neste campo tanto a BOSCH como a CiA intervindo proactivamente. Recentemente foi introduzido a extensão CAN-FD, uma versão com uma taxa mais flexível de dados, o que permite uma maior taxa de bits para serem usados. Estas extensões do protocolo fazem com que se consiga prolongar ainda mais o seu tempo de vida, e tendo em conta que a utilização do CAN no mercado ainda se encontra num estado prematuro, tudo leva a querer que haja um crescimento significativo do protocolo nos próximos anos.

## **3.2 Características**

A utilização de um protocolo de comunicação como o CAN traz ao utilizador uma série de vantagens em relação a outros tipos de comunicação, e de todas as que existem, pode-se destacar as seguintes:

- Capacidade multi-mestre - O CAN é um protocolo de comunicação série que permite controlo distribuído em tempo real e o barramento disponível para as comunicações possui a capacidade de atender vários nós com pedidos de acesso ao meio de transmissão, daí a sua capacidade de multi-mestre.
- Processo de arbitragem não destrutiva - O endereçamento dos destinatários não é utilizado no CAN como no sentido convencional, pois este protocolo utiliza identificadores nas mensagens transmitidas. Isto resulta que uma mensagem que esteja a circular na rede vai ser recebida por um ou mais dispositivos que decidem, com base no identificador recebido, se devem ou não processar esta mesma mensagem. O identificador da mensagem é também responsável por determinar a prioridade da mensagem que compete com todas as outras pelo acesso ao barramento.
- Capacidade *broadcast* - O conceito de *broadcast* no CAN pressupõe que a transmissão de uma mensagem possa ser efetuada a um conjunto de receptores em simultâneo.
- Elevadas taxas de transferências - A taxa máxima de transmissão especificada é de 1 Mbit/s, e está associada a sistemas com barramentos de comprimento até 40 m, e ao aumentar a distância dos barramentos, esta mesma taxa diminui.
- Redução do número de fios - O barramento da rede CAN apenas é constituída por dois fios entrelaçados que correspondem ao CAN\_H e CAN\_L, do inglês CAN\_HIGH e CAN\_LOW, e resistências nos extremos do barramento o que torna este protocolo muito simples de implementar numa rede, em relação ao hardware.
- Máximo de 8 bytes de informação útil - O facto da informação trans-

mitida numa rede CAN ser de tamanho curto faz com que este sistema seja ideal para a ligação com subsistemas inteligentes, tais como sensores e atuadores. Uma mensagem CAN pode conter no máximo 8 bytes de informação útil, embora seja possível, através de segmentação, transmitir blocos de dados superiores.

- Deteção e Sinalização de Erros - O controlador CAN presente em cada nó utiliza mecanismos capazes de registar os erros ocorridos, avaliando-os estatisticamente, de forma a desencadear ações com eles relacionados.
- Flexibilidade e Confiabilidade - A flexibilidade neste protocolo está presente no facto de ser possível adicionar novos nós a uma rede CAN sem requerer alterações tanto da parte do software como de hardware dos restantes nós, no caso de o novo nó não ser emissor ou não necessitar de transmissão de dados adicionais.
- Baixo custo - Devido ao facto de a maioria destes sistemas que possuem uma rede CAN sejam constituídos por pouco hardware (microcontrolador, *transceiver*, sensores e atuadores) faz com que a implementação de sistemas deste tipo seja de mais baixo custo que sistemas de outros tipos.

### 3.3 Funcionamento

Neste subcapítulo o objetivo é explicar o funcionamento do protocolo CAN, Para isso irá se falar sobre a arquitetura do sistema bem como as suas camadas envolventes.

Face ao modelo ISO (International Organization for Standardization) - OSI (Open Systems Interconnection model), a especificação da rede CAN descreve apenas a camada física e a camada de ligação de dados apresentado na figura 3.1, ambas implementadas em hardware. Para facilitar a

programação, e devido aos diferentes cenários de utilização, foram desenvolvidas diversas camadas de software que se posicionam na camada de aplicação.

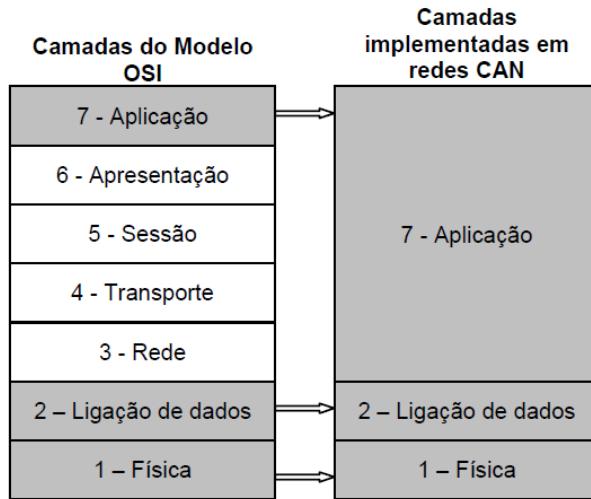


Figura 3.1: Modelo de camadas OSI e CAN.

### 3.3.1 Camada física

A camada física de um sistema de comunicação cobre os aspectos da transmissão física dos dados (bits) entre os nós da rede. Nesta camada destacam-se três sub-camadas [16]:

- *Physical signaling* (PLS), implementada no controlador CAN

Codificação/descodificação de bits;

Timing dos bits;

Sincronização entre nós;

- *Physical Medium Attachment* (PMA)

Características do transceiver;

- *Medium Dependent Interface* (MDI)

Características do meio de transmissão (cabos, fichas, etc);

**Codificação:** Uma trama de dados (figura 3.2) num barramento CAN é codificada em *non return to zero* (NRZ). Durante o tempo em que um nó transmite um bit, o barramento pode estar num estado recessivo ou dominante, de acordo com o bit a ser transmitido. O barramento CAN tem três níveis lógicos: o estado dominante, o estado recessivo e o estado *idle*, onde nenhum nó está a transmitir [16].

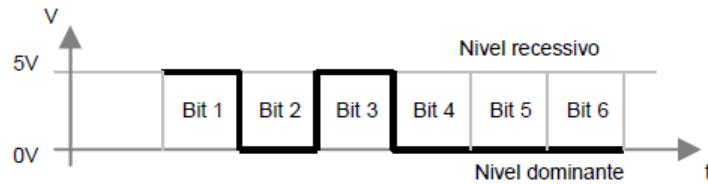
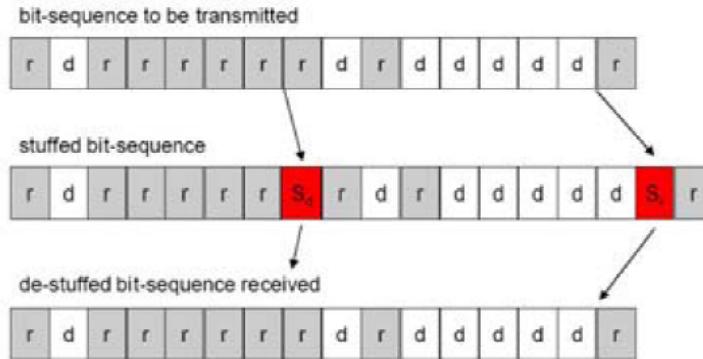


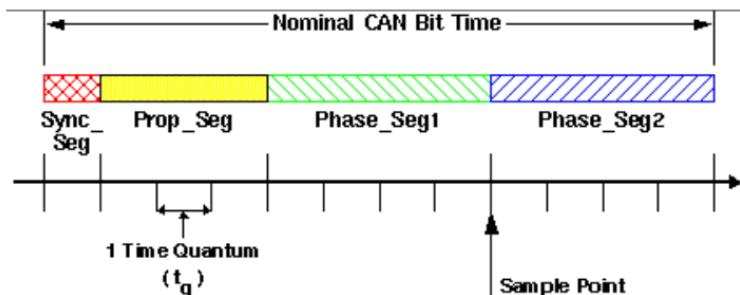
Figura 3.2: Exemplo de stream de bits transmitidos, codificados em NRZ.

A não implicação de uma mudança de estado no barramento para cada bit transmitido dificulta a sincronização entre os nós, devido à codificação utilizada. No caso de uma sequência de bits iguais (recessivos ou dominantes), os nós podem ter dificuldades na sincronização entre si. Estes podem não reconhecer os limites dos bits. Para ultrapassar este problema, recorre-se ao *bit stuffing*. Durante a utilização do barramento, são permitidos no máximo cinco bits consecutivos da mesma polaridade. Caso o nó pretenda transmitir mais do que o permitido, são introduzidos *stuff-bits* de polaridade inversa (pela camada de ligação de dados), de forma a garantir que de facto não existam mais do que cinco bits com a mesma polaridade no barramento (figura 3.3) [18].

**Timing:** Cada nó CAN opera em sincronia com um oscilador que gera um sinal de frequência pré programada. A cada conjunto de n períodos dessa frequência dá-se o nome de *time-quantum* (Tq), sendo o tempo de

Figura 3.3: Exemplo de *bit stuffing* numa trama de bits

transmissão de cada bit (bit-time) definido como múltiplo de  $T_q$ . De facto, o bit-time subdivide-se em quatro segmentos distintos, cada um deles com duração múltipla de  $T_q$ . Existe um segmento de sincronização com duração de 1  $T_q$ , um segmento de propagação com 1 a 8  $T_q$  e dois segmentos de fase com 1 a 8  $T_q$  cada (figura 3.4) [16] [5].

Figura 3.4: Esquema do *timing* de um bit [5]

**Sincronização de nós:** Um nó sincronizado com o barramento está dessincronizado com o emissor devido aos tempos de propagação. Numa rede CAN, um transmissor que envia um bit recessivo tem que ter a possibilidade de receber um bit dominante enviado por um nó sincronizado com o barramento (ACK bit). Para tal, é necessário estender o *bittime* ao longo destes quatro segmentos para garantir a coerência entre todos os nós da

rede.

O segmento de sincronização é utilizado pelos vários nós para uma sincronização inicial. Se existirem variações de estado do barramento, estas ocorrem durante este segmento. O segmento de propagação prevê a existência de um atraso de propagação e de processamento entre os vários nós da rede, e é usado para compensar esses atrasos. O instante entre os dois segmentos de fase (segmentos finais) define o instante de amostragem do barramento [16].

Este elaborado esquema de *timing* serve para cada um dos nós esperar pelos atrasos dos sinais e pela estabilização do bit antes de amostrar o barramento e determinar que tipo de *bit* está no canal.

No caso de um nó detetar uma variação do nível do barramento fora do segmento de sincronização, as durações dos segmentos de fase são reprogramadas na altura, de forma a variar o instante de amostragem e garantir uma amostragem mais fiável. Em caso de avanço do *bit*, a duração do primeiro segmento de fase é menor num valor programado RJW (*Resynchronization Jump Width*), e no caso de um atraso da transição, a duração do segundo segmento de sincronização é alargada em RJW. A este mecanismo chama-se *soft-synchronization* e só ocorre uma vez em cada *bit* [16].

Em qualquer caso, a soma de todos os atrasos (processamento e propagação) do nó mais distante tem que ser menor do que o segmento de propagação. Este detalhe de sincronização vem limitar a relação taxa de transmissão e comprimento do barramento, essa relação pode ser vista na figura 3.5

Quanto maior for o barramento, mais significativos são os atrasos de propagação e maior tem que ser o *bit-time*, reduzindo a taxa de transmissão máxima. É ainda possível a utilização de *bridges* e repetidores para aumentar o comprimento máximo do barramento.

O *standard ISO 11898-2* define o barramento como uma linha de dados

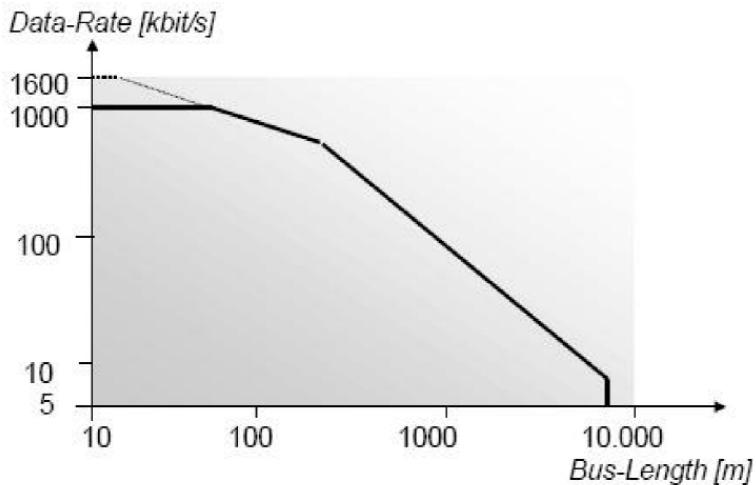


Figura 3.5: Relação entre taxa de transferência e comprimento do barramento

de 2 fios entrelaçados para minimizar as reflexões, nas extremidades é terminada com resistências (figura 3.6), resistências de terminação. A linha tem três estados de atividade: *idle*, bit recessivo e bit dominante. Estes estados são definidos pelas tensões diferenciais introduzidas na linha (CAN\_H e CAN\_L) pelos nós. O facto de ser tensões diferenciais tem como objetivo minimizar interferências eletromagnéticas, dado que os dois fios do barramento estarão fisicamente perto um do outro e, na presença de interferência, os efeitos sentidos em ambos serão semelhantes. Como a tensão é medida diferencialmente, o valor dessa interferência desaparece [16].

Dentro de um nó CAN, o *transceiver* é responsável pela deteção e sinalização ao microcontrolador de vários tipos de falhas no barramento. Existem *transceivers* tolerantes a falhas que permitem um modo de operação *single-wire* a uma relação sinal-ruído reduzida, de forma a tolerar várias falhas do barramento, como curto-circuitos entre as linhas CAN\_H e CAN\_L, entre as tensões de alimentação e as linhas ou ainda circuitos abertos em alguma linha.

Finalmente, e para terminar a descrição da camada física, existe a pos-

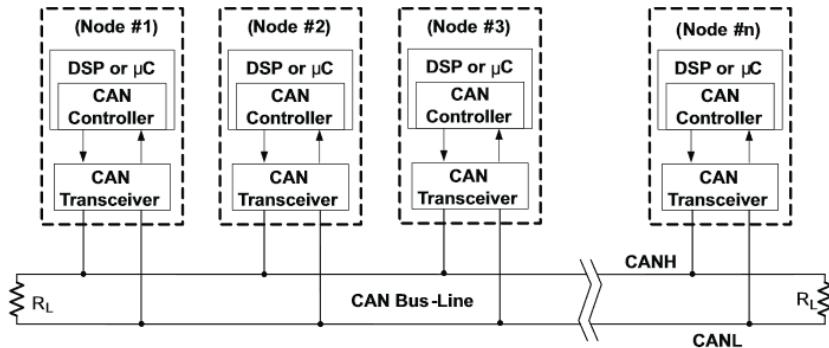


Figura 3.6: Topologia de uma rede CAN

sibilidade de que o barramento seja ligado a transmissores RF ou infravermelhos como meio de estabelecimento de ligações de/ou para outras redes CAN.

### 3.3.2 Implementação em hardware

A implementação de um nó CAN em *hardware* cobre a camada física, como já foi referido, e cobre também a camada de ligação de dados. O hardware presente em cada nó pode ser dividido em três blocos funcionais como pode ser observado na figura 3.7: o *transceiver*, o controlador CAN e o microprocessador.

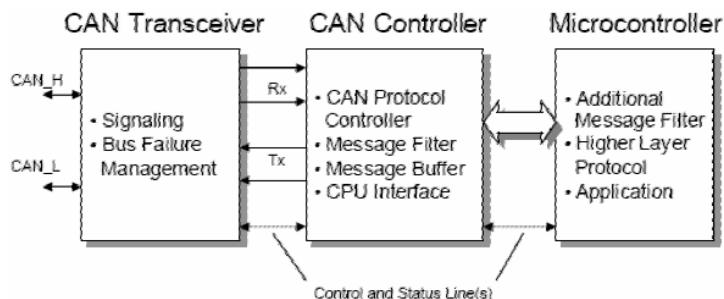


Figura 3.7: Blocos funcionais de um nó CAN

Em particular, a camada física é implementada no *transceiver* e no ex-

terior do nó (tomada, ficha, cabo, etc) e a camada de ligação de dados está no controlador CAN. O papel do micro-controlador no protocolo de comunicações é a implementação das camadas superiores pretendidas pelo utilizador através da programação do software adequado. Este bloco (microcontrolador) pode ser dedicado ao interface de comunicação do nó ou partilhado por outros periféricos, enquanto que todos os outros blocos se dedicam exclusivamente às suas funções no protocolo de comunicação [15].

Um nó CAN pode ter os três blocos funcionais implementados separadamente ou integrados em diversas formas. Cada solução de integração tem vantagens e desvantagens face às outras, mas a que parece ser mais popular é aquela em que o controlador CAN está integrado no micro-controlador, com o *transceiver* independente CAN [15].

### 3.3.3 Camada de ligação de dados

A camada de ligação de dados, DLL (*Data Link Layer*) impõe o formato das mensagens que viajam no barramento e fornece mecanismos de deteção e prevenção de falhas. Existe uma subdivisão desta camada de forma a distinguir níveis de serviço fornecidos [14]:

- *Logical Link Control* (LLC)

Filtro de aceitação de mensagens;

Notificação de *overload*;

Gestão de recuperação de mensagens ou de situações de erro;

- *Medium Access Control*(MAC)

Encapsulamento/desencapsulamento dos dados nas mensagens;

Codificação das mensagens (stuffing/destuffing);

Gestão de acesso ao meio;

Deteção e sinalização de erros;

Confirmação da integridade das mensagens (*acknowledge*);

**Versões:** Atualmente existem duas versões de mensagens especificadas: mensagens *standard* e *extended*. Por este motivo, o hardware pode eventualmente ser implementado tendo em conta as duas versões. Para isso foram diferenciadas três especificações do protocolo da ligação de dados: protocolo 2.0A, 2.0B passivo e 2.0B ativo. Destes, apenas o protocolo 2.0B ativo consegue receber e transmitir de forma transparente as mensagens das duas versões. O protocolo 2.0B passivo consegue receber as mensagens das duas versões mas apenas transmite mensagens *standard*, enquanto que o protocolo 2.0A descarta todas as mensagens *extended* recebidas porque as reconhece como erros [7]. Na tabela 3.1 é apresentado a comparação.

Tabela 3.1: Compatibilidades das versões CAN

Versão	2.0A	2.0B Passivo	2.0B
Standard	OK	OK	OK
Extended	ERRO	Tolerante	OK

**Filtro de aceitação de mensagens:** Na subcamada LLC da DLL existe um mecanismo que filtra as mensagens que recebe, ou seja, existe uma ou várias máscaras de bits de comprimento variável que são sobrepostas (AND) com os n primeiros bits do identificador da mensagem, de forma a decidir se a mensagem é entregue à camada acima ou descartada. Este mecanismo é programável pelas camadas superiores ou, em última análise, pelo utilizador [14].

**Funcionalidades adicionais:** Na implementação do controlador, existe a possibilidade de incluir funções não previstas na especificação, tais como: contadores (adicionais) de erros, *timestamps* nas mensagens, limites de aviso programáveis, geração de pedidos de interrupção, estado de consumo reduzido e contadores de mensagens sem erros.

**Stuffing/destuffing:** Sempre que o transmissor detetar a presença de 5 bits consecutivos da mesma polaridade numa trama de bits (após a seria-

lização), introduz um bit de polaridade inversa que vai ser automaticamente removido pelos receptores (figura 3.3). Esta regra de *stuffing* pode ser utilizada para uma verificação de integridade.

Na construção de uma mensagem, o nó transmissor preenche o cabeçalho e a terminação, executa o processo de *stuffing* em determinadas zonas da mensagem e calcula o CRC (*Cyclic redundancy check*). Nesta altura, a mensagem está pronta para ser enviada.

Nos nós receptores o processo é inverso. Após a receção da mensagem sem erros detetados no barramento, esta é *destuffed*, os campos de correção de erros são verificados, passa pelo filtro de aceitação e, caso seja essa a decisão, os bits podem ser novamente agrupados em bytes e enviados à camada acima [14].

**Difusão:** O conceito de difusão da rede CAN significa que todos os nós veem as mensagens transmitidas por qualquer um deles, como pode ser visualizado na figura 3.8. Após a receção de cada mensagem, cabe aos nós a decisão de descarte ou entrega da mensagem. [14]

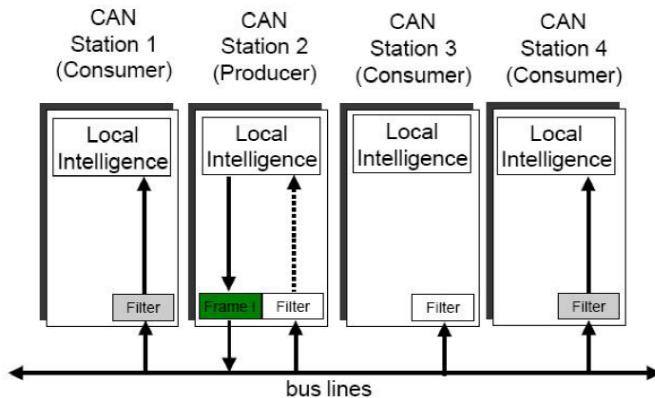


Figura 3.8: Difusão de mensagens CAN [14]

**Remote frame handling :** A maioria das implementações com armazenamento de objetos transmitidos integra um mecanismo de retransmissão de mensagens, RTR (*Remote Transmission Request*), por pedido de outros

nós. Dependendo da arquitetura do controlador, esta retransmissão pode ser feita por autorização do CPU (*Central processing unit*, sem conhecimento do CPU ou com conhecimento/autorização opcional do CPU [14]. O método utilizado para a retransmissão é semelhante a um processo de pergunta/resposta: o nó interessado envia uma trama do tipo *remote frame request* (pergunta) onde especifica a mensagem em que está interessado. Esta trama será eventualmente recebida por algum nó que ainda tenha a mensagem. Este nó responde então com a mensagem pretendida. A mensagem reenviada pode também ser recebida por outros nós interessados. Os *remote frame requests* são claramente distinguíveis das mensagens de dados por uma *flag* explicitamente assinalada e pelo facto de não conterem dados [14].

**Acesso múltiplo:** Um dos requisitos de uma rede CAN é a possibilidade de acesso múltiplo ao barramento, ou seja, vários nós podem ler do barramento ao mesmo tempo mas, para evitar colisões (dois ou mais nós a enviar ao mesmo tempo uma mensagem diferente, o que resultaria numa mensagem inválida), é preciso um mecanismo de prioridades. O mecanismo utilizado é o *Carrier Sense Multiple Access/Deterministic Collision Resolution* (CSMA/DCR) [14].

Cada mensagem leva consigo um identificador, uma identidade que é atribuída pelo nó. Quando o barramento está em estado *idle*, vários nós podem iniciar as suas transmissões de dados ao mesmo tempo. Cada nó lê do barramento a cada bit que envia e compara os valores. Caso o bit que o nó tenha tentado escrever seja recessivo, será lido do barramento um bit recessivo se e só se todos os nós que estão a transmitir na altura estiverem a enviar um bit recessivo. Por definição, o *transceiver* garante que um bit dominante permanece no barramento em detrimento de um bit recessivo. Para melhor exemplificação deste sistema é apresentado um exemplo na figura 3.9.

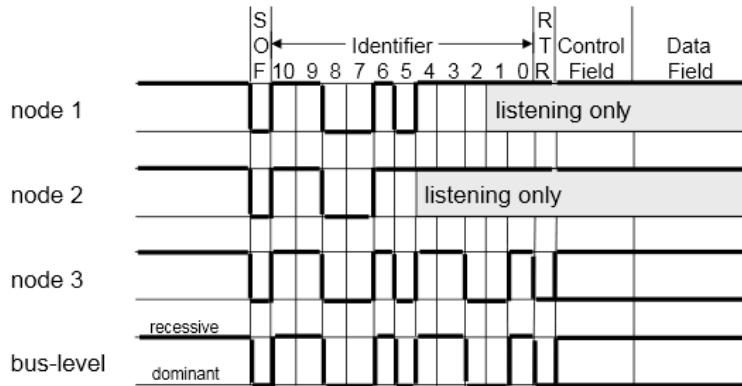


Figura 3.9: Sistema de anti-colisões CSMA/DCR

Durante a transmissão do identificador da mensagem, o algoritmo CSMA/DCR decide de forma não destrutiva e sem atrasos ou retransmissões quem transmite ou não a mensagem. No funcionamento do algoritmo CSMA/DCR em que  $n$  nós tentam escrever no barramento, e assumindo que todos tentam enviar mensagens diferentes,  $n-1$  nós vão desistir de transmitir a mensagem para que o restante nó possa transmitir a dele. Depois da transmissão da mensagem do nó que “ganhou” o barramento, os outros nós tentam novamente transmitir as suas mensagens. Com a utilização do algoritmo CSMA/DCR, é possível estabelecer ordens de prioridade nas mensagens, porque estas têm identificadores programáveis [14].

A figura 3.9 mostra como o algoritmo CSMA/DCR funciona. Como se pode verificar temos 3 nós (1,2,3) e o barramento ( $n$ ). Cada um dos nós está a tentar enviar uma mensagem simultaneamente. No bit 5 do identificador, o nó 2 percebe que enviou um bit recessivo mas leu do barramento um bit dominante e desiste de enviar a sua mensagem, este fica em modo de escuta apenas. No bit 2 do identificador, o nó 1 também entende que enviou um bit recessivo mas leu um bit dominante. Também este nó desiste de transmitir a mensagem. Desta forma, o nó 3 prevalece e transmite a sua mensagem.

**Serviços de comunicação fornecidos pelo protocolo CAN:** O pro-

Protocolo CAN especifica dois serviços de comunicação: o serviço *write object* e o serviço *read object*. O serviço *write object* é o serviço que escreve uma mensagem de um nó (emissor) para o barramento e, consequentemente, para todos os nós a ele ligados (serviço básico do protocolo). Este serviço não implica que algum nó esteja interessado na mensagem [14]. O serviço *read object* é o complementar do anterior e é iniciado pelo facto de um receptor interessado numa trama transmitir uma mensagem de RTR com a identificação dessa trama. Se algum nó tiver os dados pretendidos, envia de volta a trama correspondente [14].

**Trama de dados:** Até agora referenciadas genericamente como mensagens, as tramas de dados são as unidades de comunicação entre as camadas de ligação de dados dos vários nós. Como pode ser visualizado na figura 3.10, é apresentado a trama de dados, no caso *standard* de 11 bits de identificador.

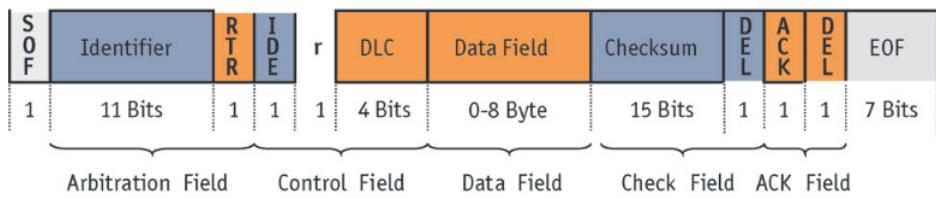


Figura 3.10: Trama de dados [20]

Sempre que um nó pretende enviar dados para o barramento, é feito através do encapsulamento da informação numa ou mais tramas de dados, em que cada uma pode transportar 1 a 8 bytes de informação. O trama de dados começa com um bit *Start of Frame* (SoF) para a sincronização em todos os nós. Depois do SoF está o campo de identificação (*arbitration field*) que identifica a prioridade e o conteúdo da mensagem. O campo de controlo (*control field*) especifica o comprimento em bytes da mensagem. O campo de dados (*data field*) contém a informação propriamente dita e o campo de CRC (*cyclic redundancy check*) guarda informação dedicada para deteção de erros em determinadas zonas da trama. O campo de confirmação (Acknowledge

field - ACK) é utilizado para o transmissor entender se pelo menos um nó recebeu a mensagem sem erros de barramento. A mensagem é terminada pelo *End of Frame* (EoF). Atualmente, existem dois tipos de tramas: tramas de dados *standard* e as *extended*, como referido anteriormente. Os dois tipos diferem no significado de algumas *flags* e no comprimento do campo de identificação [7].

O bit SoF era antigamente utilizado para uma sincronização do barramento, mas as especificações mais recentes tornam essa sincronização desnecessária. O campo de identificação é utilizado pelo algoritmo CSMA/DCR para determinar o nó a que envia a mensagem. O bit final do campo de identificação, o RTR *Remote Transmission Request*, é utilizado no estado recessivo para marcar as mensagens como remotas, caso o bit seja dominante a trama fica marcada como uma trama de dados. O bit seguinte, IDE (*Identifier Extension*), serve para distinguir mensagens *standard* das *extended*. A mensagem *standard* tem o bit IDE dominante para que, em caso de transmissão simultânea, a mensagem standard prevaleça devido ao algoritmo CSMA/DCR explicado anteriormente.

Até à ocorrência do bit IDE, ainda não se sabe de que tipo de trama se trata. Se o bit IDE for dominante, o nó entende-o como o bit r1 do campo de controlo, *control field* na figura 3.11, de uma mensagem *standard*. Se o bit for recessivo, então a trama é *extended* e o bit é o bit IDE do *arbitration field* após o SRR (*Substitute Remote Request*), que é transmitido recessivo. Para terminar o campo de identificação das mensagens *extended*, existe ainda um campo de identificação extra de 18 bits e 1 bit RTR (verdadeiro), totalizando 32 bits de comprimento para este campo. O campo de controlo é semelhante nos dois tipos de mensagem. Os dois primeiros bits são dominantes e os últimos quatro, DLC (*Data Length Code*), especificam o comprimento do campo de dados (em bytes)[14].

O campo de dados marcado a encarnado na figura 3.11 como *Data Field*

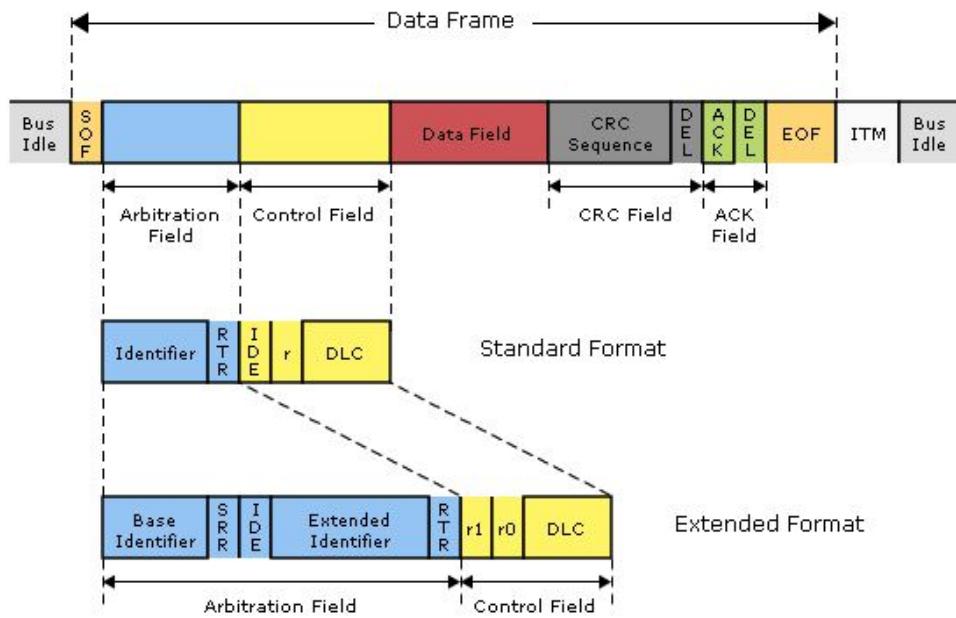


Figura 3.11: Trama de dados (identificador) [20]

leva a informação na mensagem num comprimento entre 0 e 8 bytes (64 bits). O campo de CRC é o seguinte, este composto por uma sequência CRC de 15 bits e um bit limitador recessivo (CRL). A sequência CRC usada provém de um código de *hamming* de tamanho mínimo de 6, o que significa que possibilita a deteção de 15 erros no máximo. O campo de ACK, a verde na figura 3.11, é utilizado para que o emissor saiba se pelo menos um nó na rede recebeu a mensagem livre de erros até ao momento. Este é composto por dois bits: ACK e ACK delimiter (DEL). Finalmente, o campo EoF é composto por sete bits recessivos e foi introduzido para garantir que, se algum nó enviar uma error flag, essa flag fica contida dentro da mensagem [28].

**Controlo de Erros:** Na ocorrência de um erro, é enviada uma trama de erro (*error frame*). A trama é composta por uma *flag* de 6 bits dominante e é seguida de sobreposições dos outros nós, pois é possível que nem todos detetem o erro exatamente no mesmo bit. No fim existe o delimitador do

erro que é composto por 8 bits recessivos e permite aos nós recomeçar a transmissão após a ocorrência de um erro.

Os erros que podem surgir e causar a geração de uma trama de erro são [34]

- Erros de *stuffing*: Quando o receptor deteta mais do que 5 consecutivos no mesmo estado, sejam recessivos ou dominantes. Esta regra é apenas aplicada entre o SoF e o CRC *field* inclusive.
- Erros de bit: Sempre que um nó escreve no barramento ele lê, por assim dizer o que escreveu. Quando um nó escreve no barramento mas a leitura é diferente da escrita então é detetado um erro de bit.
- Erro de formato: Este erro ocorre quando o formato da trama não é o esperado, por exemplo, como já foi dito anteriormente no EoF, este campo é composto por sete bits recessivos, se por ventura o EoF conter um bit dominante significa que o formato da trama não é o esperado.

### 3.3.4 Camada de aplicação

A camada de aplicação não está explicitamente definida para o CAN. Está em aberto para que os utilizadores definam os seus algoritmos de acordo com as características do sistema quando usando CAN.

## 3.4 Tecnologias CAN existentes

Para construir uma rede CAN é necessário fazer uma pesquisa exaustiva de modo a ver as tecnologias existentes. Para se poder constituir uma rede CAN completa, não basta ter apenas os controladores CAN autónomos ou os microcontroladores com controladores CAN integrados. Na figura 3.6 é possível observar é indispensável ter um microcontrolador , um controlador CAN e um CAN *Transceiver*. Só assim se consegue implementar um nó na rede.

### 3.4.1 Microcontrolador

Dos grandes fabricantes de microcontrolador pode se destacar a Microchip Technology, Atmel Corporation e a ST Electronics. Estes fabricantes são os mais conhecidos e são também os que atendem mais facilmente às necessidades dos utilizadores.

**Microchip Technology:** Este fabricante não produz apenas microcontrolador mas também é produtor de semicondutores analógicos. Focando só na parte dos microcontrolador , a Microchip Technology dispõe de uma vasta gama de microcontroladores . A gama vai desde os microcontroladores de 8bit passa pelos 16 e chega mesmo aos 32bit de arquitetura. Os micro controladores mais utilizados deste fabricante são PIC16F84, PIC16F628, PIC18f458. A família PIC18FXX8 toda ela tem incorporado um controlador CAN [22], deixando assim de ser necessário o uso de um controlador CAN exterior.

**Atmel Corporation:** Talvez este fabricante seja a grande concorrência da Microchip Technology. Apesar deste produtor não fabricar apenas microcontroladores e à semelhança da Microchip Technology irá se falar apenas dos microcontrolador . A gama passa pela arquitetura 8051, 8/32bit e por fim microcontroladores de arquitetura ARM. Os microcontroladores mais procura talvez seja o atmega 328 devido a um projeto *opensouce* denominado Arduino. A Atmel Corporation tem uma linha dentro da gama dos 8/16bits dedicada a área automotiva alguns deles possuem também o controlador CAN incorporado.

**ST Electronics:** Este fabricante possuí uma vasta gama de microcontroladores , à semelhança dos anteriores fabricantes, este tem microcontroladores de 8bit até aos de arquitetura ARM. Mas sem dúvida a ST Electronics é particularmente conhecida pelos seus microcontroladores ARM. Os ARM mas utilizados atualmente são a linha stm32, nomeadamente stm32f4. Na família stm32f todos os microcontroladores já tem um controlador CAN in-

corporado com a exceção do stm32f100 que não tem nenhum controlador CAN.

### 3.4.2 Controlador CAN

Em alguns microcontroladores já têm controlador CAN incorporado embora seja possível utilizar um microcontrolador, como unidade de processamento, e um controlador CAN externo. Em algumas circunstâncias isso é vantajoso, assim é possível escolher o melhor microcontrolador para a aplicação. Mas nos dias de hoje e pela oferta de microcontroladores que possuem controladores CAN incorporados a solução microcontrolador + controlador CAN externo caiu em desuso.

Quer seja externo, quer seja interno num microcontrolador, o controlador CAN, é o responsável pela realização das tarefas relacionadas com a camada 1 (Físico) e a camada 2 (Ligaçāo de dados) do protocolo CAN [25].

A Microchip Technology detém um controlador com *transceiver* integrado [21] (MCP25625). Uma característica relevante deste circuito integrado é a tensão de trabalho, vai desde os 2.7V até aos 5.5V o que é interessante pois pode ser utilizado pelos microcontroladores que operem a 3.3V ou 5V dispensando a utilização de *shifters*. Este contém proteção ESD, pode operar até a 1Mb/s.

A Philips, atualmente designada por NXP também possui um controlador externo, o SJA1000 [24]. Este é mais básico e possui apenas as funções mais básicas como funções de deteção de erros, possível operação apenas modo de escuta, modo de teste.

### 3.4.3 Transceiver CAN

Será mesmo necessário a utilização de *transceivers* CAN na implementação da rede? É uma boa questão, a regra geral diz que é necessário um *Transceiver* CAN para cada nó da rede. Contudo, em certas circunstâncias pode-se

estabelecer comunicação sem *Transceiver* CAN, segundo a Siemens [30]. As circunstâncias são:

- Barramento pequeno, menos de um metro;
- Preferencialmente todos os microcontroladores na mesma PCB (*Printed circuit board*);
- taxa de transferência relativamente baixa;
- Ambiente com pouco ruído eletromagnetismo.

Para conseguir isso, é necessário saber um pouco como um *transceiver* CAN funciona. A maior parte dos *transceivers*, tem uma saída alta e outra baixa (*high* e *low*) que representam 1 e 0 respectivamente, sendo o zero o bit dominante e o bit recessivo o um. Podemos então recriar a mesma situação simplesmente usando diodos como mostra a figura 3.12.

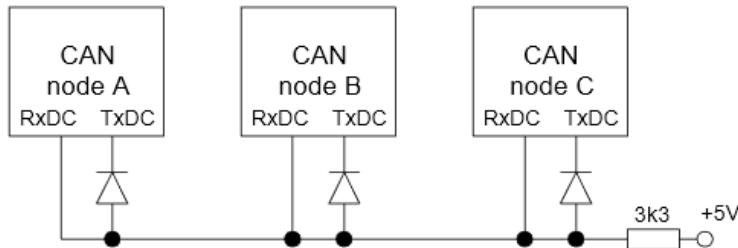


Figura 3.12: Conexão de nós CAN sem *transceivers* [30]

Uma coisa interessante é que os microcontroladores PIC atualmente possuem suporte para a não utilização de transceivers (*transceiverless*). É possível ver (figura 3.13) no *data sheet* do PIC18FXX8 o registo CIOCON em que podemos configurar o pino TX do CAN como *tri-state*.

### 23.2.5 CAN MODULE I/O CONTROL REGISTER

This register controls the operation of the CAN module's I/O pins in relation to the rest of the microcontroller.

#### REGISTER 23-55: CIOCON: CAN I/O CONTROL REGISTER

U-0	U-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0
—	—	ENDRHI <sup>(1)</sup>	CANCAP	—	—	—	—
bit 7							bit 0

##### Legend:

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared

- bit 7-6      **Unimplemented:** Read as '0'  
 bit 5      **ENDRHI:** Enable Drive High bit<sup>(1)</sup>  
 1 = CANTX pin will drive V<sub>DD</sub> when recessive  
 0 = CANTX pin will be tri-state when recessive  
 bit 4      **CANCAP:** CAN Message Receive Capture Enable bit  
 1 = Enable CAN capture, CAN message receive signal replaces input on RC2/CCP1  
 0 = Disable CAN capture, RC2/CCP1 input to CCP1 module  
 bit 3-0      **Unimplemented:** Read as '0'

**Note 1:** Always set this bit when using differential bus to avoid signal crosstalk in CANTX from other nearby pins.

Figura 3.13: Registro CIOCON [22]

Quando se configura o pino deste modo fica a funcionar exatamente da mesma maneira que um *transceiver*. Isto significa que podemos retirar os diódodos, assim só precisaríamos de uma resistência (figura 3.14).

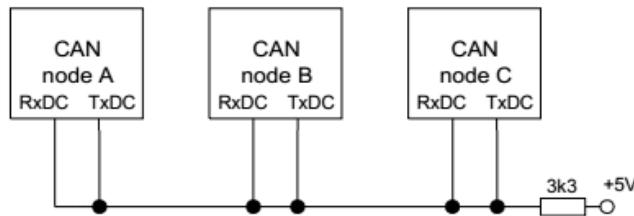


Figura 3.14: Circuito final sem transceivers usando PIC18FXX8

Deixando este caso em especial de parte pois, isto retira um grande vantagem da utilização do protocolo CAN, a sua flexibilidade e a do tamanho da rede.

A maior parte dos *transceivers* requerem 5V como tensão de alimentação como era requerido na norma *standard ISO 11898*. Contudo começou a sur-

gir uma preocupação com a eficiência dos circuitos criados e já existe *transceivers* a operar com 3.3V. Esta alteração reduz 50% da potencia consumida ou mais como se pode observar na figura 3.15 [7].

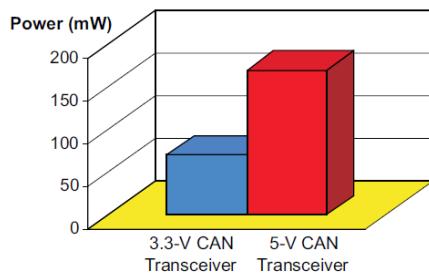
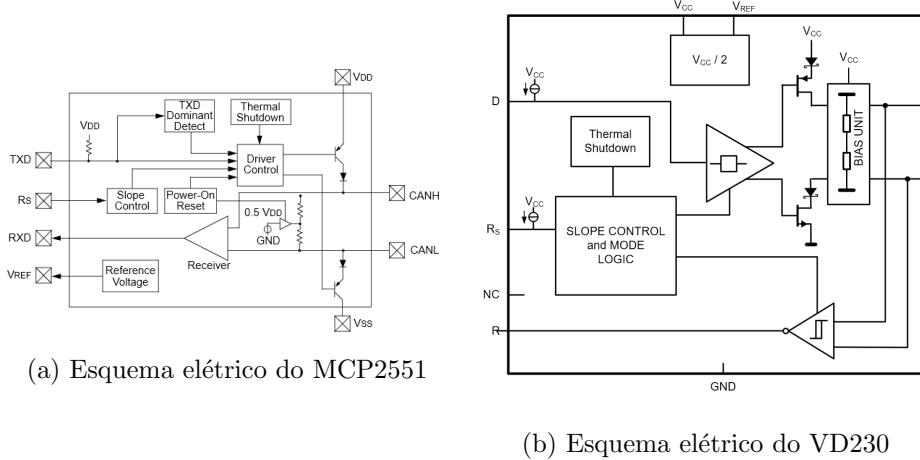


Figura 3.15: Potencia consumida pelos CAN *transceivers* [7]

Da Microchip Technology existe o MCP2551, este é no *package* DIP o que facilita a montagem de *breadboards* por exemplo, tem mecanismo de desligar automático quando a temperatura aumenta, é anunciado como imune a curto circuitos, é possivel ligar até 112 nós com este *transceiver*.

Por parte da Texas Instruments são os VD230 que são *transceivers* SMD que operam exclusivamente com microcontroladores que usam 3.3V como tensão de funcionamento, consegue funcionar numa rede de 120 nós. Se for necessário um *transceiver* mais robusto, a Texas Instruments oferece o ISO1050 que é semelhante ao VD230 mas este é *galvanically isolated* o que torna ainda mais imune ao ruído, na folha do fabricante é referido como um *transceiver* de alta duração. Na figura 3.16 é possível ver os esquemas elétricos dos *transceivers* CAN referidos acima.

Estes 3 *transceivers* são os mais conhecidos e utilizados, existem outros *transceivers* para aplicações mais específicas, um exemplo disso é o ADM3053 da Analog Devices, é um *transceiver isolated* com um conversor DC-DC integrado. Utiliza a tecnologia *iCoupler* da própria Analog Devices para o isolamento da parte *transceiver* CAN e a *isoPower* para o isolamento do DC-DC [9].

Figura 3.16: Esquemas elétricos dos *transceivers* CAN

De uma certa maneira podemos ter 3 dispositivos físicos microcontrolador + controlador + *transceiver*, mas se quisermos ter apenas um com tudo integrado já é possível a NXP oferece um microcontrolador com isso tudo integrado, o LPC11C00, o primeiro microcontrolador com *transceiver* CAN incorporado [23]. O LPC11C00 é um ARM Cortex-M0 que trabalha a 50MHz com o *transceiver* CAN, nomeadamente o TJF1051 [23]. Esta solução é de baixo custo e de alta *performance*. É prometido um redução de espaço até 50% e em termos de custo até 20%.

#### 3.4.4 A Escolha

Um dos objetivos é realizar este projeto sem grandes custos, posto isto então teremos que reutilizar microcontroladores que já temos adquiridos.

Escolhendo um microcontrolador , para mais simples implementação iremos apenas nos focar nos microcontrolador com controlador CAN já incorporado dos que estamos familiarizados são os microcontrolador ARM da ST Electronics. Sobrando apenas a escolha do *transceiver* CAN, uma escolha acertada devido à nossa aplicação será os ISO1050 da Texas Instruments. Estes componentes também já possuímos.

## Capítulo 4

# Arquitectura Geral do Sistema

Neste capítulo, irão se abordar vários aspectos referentes à arquitetura do sistema a implementar. Primeiramente irá ser expor o diagrama de blocos do sistema a ser implementado. Posteriormente irá ser apresentado o microcontrolador a utilizar, algumas das suas características mais importantes para desenvolvimento do projeto. Não menos importante os *transceivers* CAN também serão explicados bem como a sua importância. Sensores que iremos utilizar também serão devidamente apresentados e analisado o seu funcionamento.

### 4.1 Arquitetura do sistema

O sistema irá ser composto por 4 microcontroladores de arquitetura ARM (*Acorn RISC Machine*) nomeadamente da família STM32 produzida pela *STMicroelectronics*. Cada um destes microcontroladores terá um *transceiver* de modo a que seja possível implementar a rede CAN. Estes serão assim ligados entre si através de uma rede CAN. Cada um dos nós terá acoplado uma série de periféricos dos quais se destacam, sensor de proximidade, ecrã

TFT (*Thin-Film Transistor*), Motor DC, matriz de LED's, sensor inercial entre outros. Na figura 4.1 é apresentado um diagrama da arquitetura do sistema.

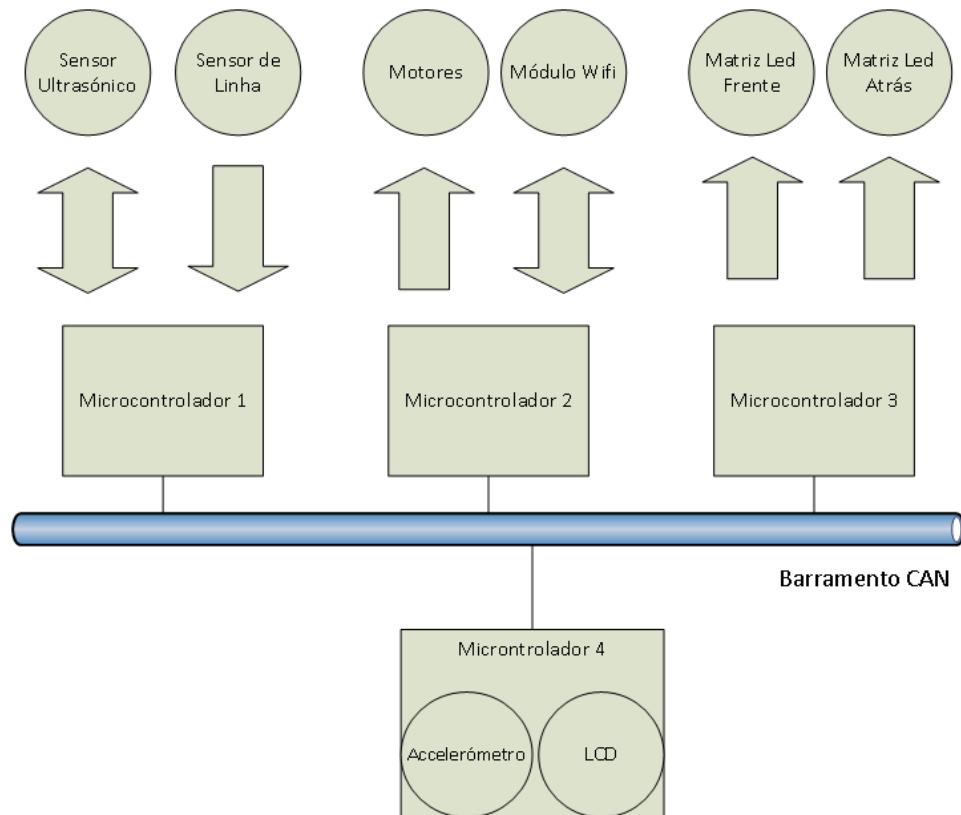


Figura 4.1: Arquitetura geral do sistema

## 4.2 Microcontrolador

Neste projeto iremos utilizar 4 microcontroladores da família STM32, 3 placas iguais, denominadas como Sistema Mínimo e uma placa de desenvolvimento da Raisonance. Na figura 4.2 é possível visualizar as placas a ser utilizadas.

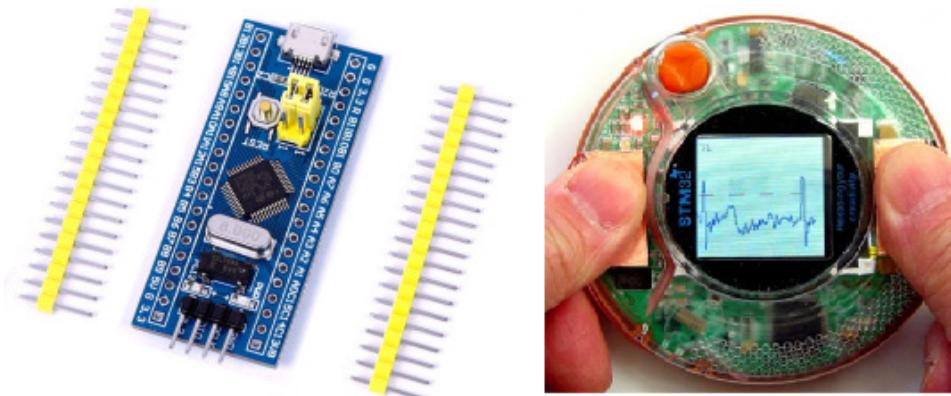


Figura 4.2: STM32f103 sistema mínimo (esquerda) e placa de desenvolvimento Primer 1 (direita)

#### 4.2.1 STM32f103 Sistema Mínimo

O microcontrolador que equipa a placa de desenvolvimento é STM32F103 em termos de *performance* pode operar até a uma frequência de 72Mhz. Para programar é necessário utilizar o conector SWD (*Serial Wire Debug*) que permite fazer a ligação entre um programador ST-LINK e o microcontrolador a programar. A placa Sistema Mínimo, não tem periférico nenhum acoplado, daí o seu nome Sistema Mínimo. Segundo o fabricante, a vcc-gnd, a placa (figura 4.3) apenas dispõem de um LED no pino PC13 e um RTC (32.768KHz) tudo o resto que a placa incorpora (resistencias, condensadores, etc) é o necessário para o perfeito funcionamento do microcontrolador . Embora esta placa seja útil para desenvolvimento pois tem o mesmo *socket* que um microcontrolador PDIP de 40 pinos a documentação sobre a placa em uso é escassa. No entanto sabendo que esta placa tem um STM32F103 podemos contar com 64 *Kbytes* de *Flash* e 20 *Kbytes* de SRAM (*Static Random Access Memory*). Tem também 4 *timers* 2 conversores A/D de 10 canais, 2 periféricos SPI e I2C, 3 USART e um periférico CAN e outro USB. Quanto aos GPIOs tem disponível 37 para uso do utilizador. [1]

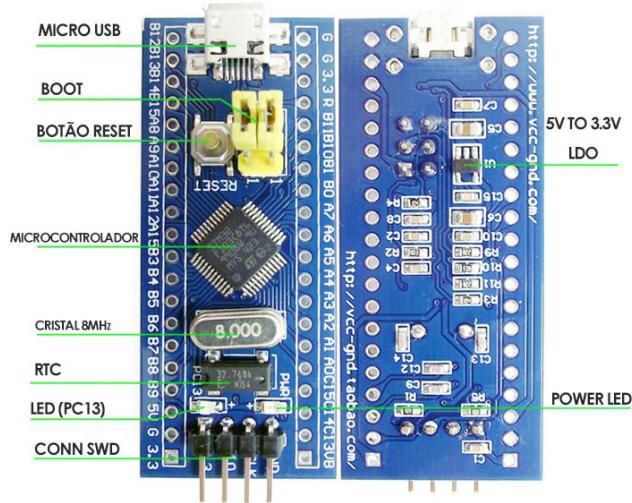


Figura 4.3: Placa Sistema Mínimo

#### 4.2.2 Raisonance Primer 1

A placa Primer 1 da Raisonance (figura 4.4) já tem diversos periféricos incorporados, dos quais se destaca, o LCD de 128x128 pixels, um sensor inercial, um *buzzer* e baterias do tipo NiMH (*Nickel–Metal Hydride Battery*). Mas quem controla estes periféricos todos de forma eficiente é o mesmo microcontrolador da placa Sistema Mínimo, o stm32f103.

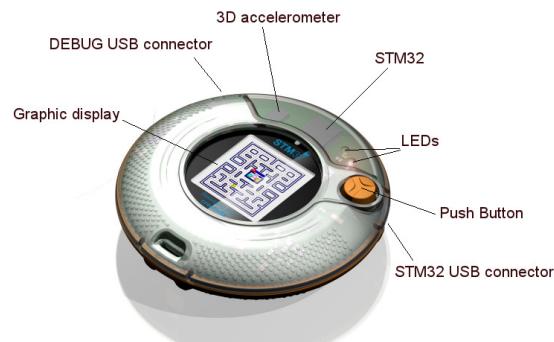


Figura 4.4: Raisonance Primer 1

Como já dispõe vários periféricos acoplados e soldados à *board* é apresentado o esquema de ligações na figura 4.5 é possível ver que o acelerómetro está ligado ao microcontrolador por SPI, o LCD ligado de modo paralelo e o *buzzer* a um pino de PWM.

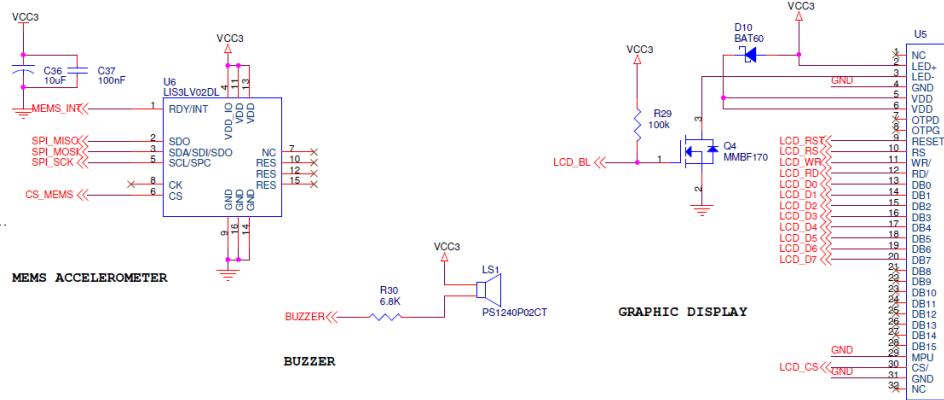


Figura 4.5: Esquema de ligações (Inercial, *buzzer* e LCD) da placa Raisonnece Primer 1 [27]

### 4.3 Configuração do periférico CAN no STM32

Nesta secção irá se falar da configuração do periférico CAN no STM32F103, pois é este o microcontrolador que irá ser utilizado no nosso sistema.

Segundo o *datasheet* [1] podemos apresentar algumas características significantes:

- Suporte para versão 2.0A e B Ativo
- *Bit rate* até 1Mbit/s
- 3 *mailboxes* para envio
- 2 *buffer FIFO* (*First In, First Out*) com 3 camadas para a receção das mensagens

Na figura 4.6 é possível ver o diagrama de blocos do controlador CAN no STM32.

Temos pinos de entrada e saída, no caso da figura 4.6 são usados os PB8 e PB9, estes vão ser ligados diretamente ao *transceiver* CAN. Temos também 3 *mailboxes* para envio onde aqui é escrito o ID, data e DLC da mensagem a ser enviada, esta passa para um conversor paralelo para série e posteriormente para o pino TX do CAN.

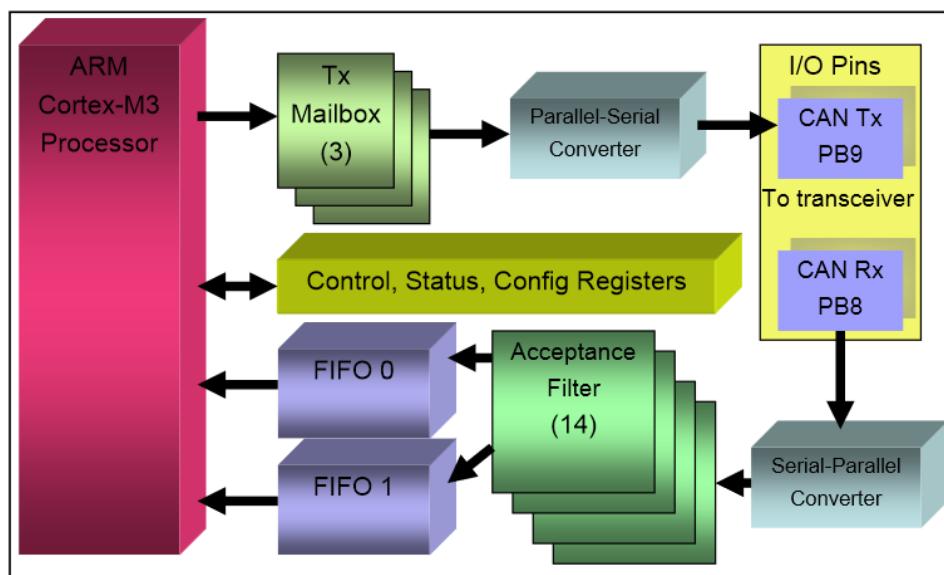


Figura 4.6: Diagrama de blocos do controlador CAN [6]

Para a receção de mensagens temos um conversor série para paralelo, seguidamente passa por um bloco que filtra as mensagens, por defeito o filtro deixa passar todas a mensagens se não for configurado, posteriormente irá para um dos 2 *buffer* FIFO e só depois é que vai seguir para o processador.

Tem ainda uma série de registos onde é possível configurar o modo de trabalho do controlador CAN, e estas são inicializados no início do programa do microcontrolador e neles é possível indicar a velocidade de transmissão, gerir a receção das mensagens, ativar interrupções entre outras. Existe uma série de flags que nos indicam o estado do controlador CAN estas são uti-

lizadas para gerir as *mailboxes* de forma a só enviar mensagens quando as *mailboxes* estiverem livres, por exemplo [6].

Todos os controladores CAN da STM tem esta arquitetura. O que pode diferir é o numero de *Mailboxes*, *buffers FIFO*, números de filtros e os pinos de entrada e saída. O principio de funcionamento é o mesmo.

#### 4.3.1 Modos de Operação

O controlador CAN presente no microcontrolador tem 3 modos de operação diferentes (figura 4.7 ):

- Inicialização
- Normal
- *Sleep*

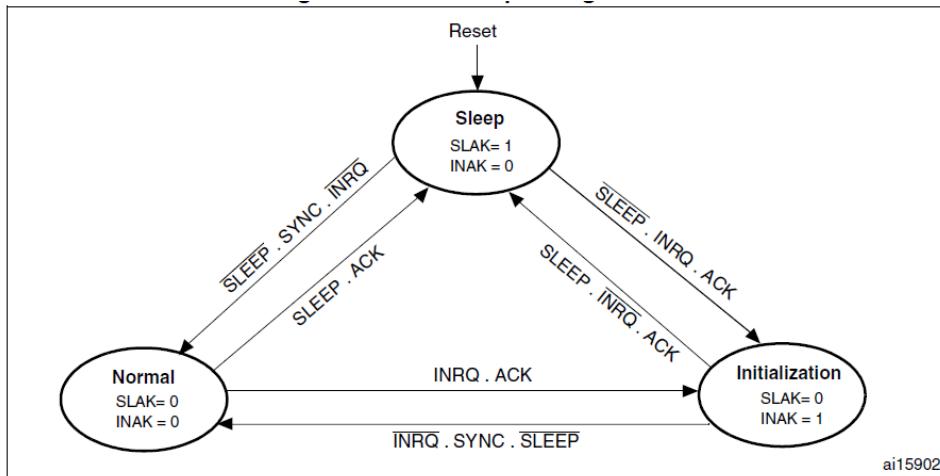


Figura 4.7: Modos de operação do controlador CAN [1]

Depois do microcontrolador stm32f103 receber um *reset* por *hardware*, o controlador CAN entra em modo *sleep* para reduzir o consumo de energia do microcontrolador. Por *software* é possível fazer com que o controlador fique em modo inicialização ou em modo *sleep*, para isso basta ativar o bit INRQ

ou SLEEP, respetivamente, no registo CAN\_MCR. Assim que o controlador muda de modo é recebido um *acknowledge*. Enquanto os bits INAK e SLAK não forem ativos o controlador CAN está em modo normal[1].

**Modo de inicialização:** Para entrar neste modo, por intermédio do *software*, é ativado o bit INRQ do registo CAN\_MCR. Depois espera que receba o *acknowledge*. Para sair deste modo é necessário limpar o bit INQR. Enquanto o controlador está neste modo todas as transferências de mensagens são paradas e o pino CANTX é colocado no estado recessivo (estado alto). O facto de entrar neste modo por si só não faz nenhuma mudança nos registos de configuração. Para iniciar o controlador é essencial configurar o *Bit Timing* e as diversas configurações que o CAN dispõe. Os registos associados aos filtros não são necessários para a configuração inicial [1].

**Modo normal:** Uma vez feita a inicialização o software deve pedir ao controlador que entre neste modo. Logo de seguida o controlador CAN sincroniza-se com o barramento e começa a transmissão e receção de dados. O pedido para entrar neste modo é feito limpando o INRQ e o SLEEP. Após entrar e estar pronto para a transmissão. A especificação dos filtros para as mensagens é independente do modo inicialização mas deve ser feito antes de entrar no modo normal [1].

**Modo sleep:** Para reduzir o consumo de energia, o controlador CAN tem o modo *low-power*. Para entrar neste modo o utilizador por intermedio do software faz o pedido ao controlador CAN colocando o bit SLEEP a 1 no registo CAN\_MCR. Neste modo o controlador fica parado, contudo ainda é possível ter acesso às *mailboxes* do controlador. O controlador pode sair deste modo de duas formas: ou limpando o bit SLEEP ou pela deteção de atividade no barramento. Nesta deteção, o *hardware* automaticamente limpa o bit SLEEP [1].

### 4.3.2 Transmissão

Para transmitir, o *software* desenvolvido deve selecionar uma *mailbox* vazia, preenchendo-a com o identificador, o DLC e a data fazendo imediatamente o pedido de transmissão ativando o bit TXRQ no registo CAN\_TIxR. Uma vez ativado o bit TXRQ a *mailbox* entra no modo pendente e espera que tenha mais prioridades que as restantes para poder transmitir. Quando isso acontecer é iniciado a tarefa de transmissão, o estado da *mailbox* passa para *transmit*. Uma vez feita a transmissão corretamente a *mailbox* volta ao seu estado inicial, vazia. Por *hardware* é indicado se a transmissão foi feita com sucesso, sendo ativados os bits RQCP e TXOK do registo CAN\_TSR. Caso a transmissão falhe os bits ALST e TERR são ativados dependendo do motivo da falha [1].

Como foi dito em cima a *mailbox* quando está no estado pendente fica à espera que tenha mais prioridade para poder enviar o seu conteúdo. Existem duas maneira para determinar quem tem maior prioridade. Por identificador:

- A mensagem com o ID mais baixo: Tem prioridade mensagens com ID mais baixo como já foi visto no capítulo anterior. Se dois identificadores forem iguais, a *mailbox* com número menor faz a transmissão primeiro [1]
- Por ordem de chegada: A *mailbox* pode ser configurada para transmitir por ordem de chegada, primeiro a chegar, primeiro a sair (FIFO). Para isso deverá ser ativo o bit TXFP no registo CAN\_MCR [1].

O pedido de transmissão pode ser cancelado pelo utilizador ativando o bit ABRQ no registo CAN\_TSR. Se a *mailbox* ainda não estiver a transmitir o processo é terminado imediatamente [1].

### 4.3.3 Re却是ão

Para a rece却是ão de mensagens CAN temos *buffers* usando a lógica FIFO. De forma a poupar tempo de processamento e simplificar o *software* e garantir a consistência da informa却是ão, FIFO é totalmente gerido por *hardware* [1].

Uma mensagem só é considerada v谩ida quando for recebida corretamente, de acordo como o protocolo CAN e passagem pelos filtros for feita com sucesso [1].

A gest茫o do FIFO como foi dito é feito por *hardware*. Num estado inicial todos os *buffers* est茫o vazios. As mensagens v茫o chegando e v茫o sendo armazenadas, com a "etiqueta" *pending\_x* at茫 serem lidas. Dependendo do microcontrolador STM32, este pode ter mais ou menos *buffers* FIFO de rece却是ão mas admitindo que tem 3 *buffers* pois este é o numero que o microcontrolador STM32 que se ir谩 utilizar dispõe. Quando estiverem todos cheios e chega uma mensagem nova corre um processo de *overrun* e uma das mensagens ir谩 ser perdida dependendo da configura却是ão do FIFO. A mensagem a ser perdida pode ser a mensagem mais antiga e ocorre uma reescrita no FIFO ou ent茫o, todas as mensagens novas s茫o descartadas. Para configurar qual o tipo de *overrun* é necess谩rio ativar ou desativar, respetivamente o bit RFLM no registo CAN\_MCR [1].

Sempre que o FIFO acolhe uma nova mensagem, fica cheio ou ocorre o processo de *overrun* é possivel gerar uma interrup茫o [1].

### 4.3.4 Filtragem

No protocolo CAN o identificador da mensagem n茫o est茫 associado a um endere茫o de um n茫o mas sim ao conteúdo da mensagem. Consequentemente todos os dispositivos conectados 脿 rede recebem todas as mensagens. Na rece却是ão da mensagem o n茫o decide, dependendo do identificador, se deve processar a mensagem ou n茫o. Para total resposta o controlador CAN tem 28 filtros configuráveis, 14 filtros para cada controlador CAN num total de 2.

O processo de configuração é feito por *software* mas a filtragem propriamente dia é feita por *hardware* poupando assim tempo de processamento [1].

Para otimizar e adaptar os filtros para as necessidades da aplicação, cada base de filtros pode ser definida independentemente. Além disso, os filtros podem ser configurados no modo *mask* ou *identifier list*. No modo *mask* é utilizado uma máscara para de modo a fazer filtrar as mensagens aceitando mensagens com uma gama de identificadores. Já no modo *identifier list* é definido um identificador específico de modo a só aceitar determinadas mensagens [1].

Na figura 4.8 é mostrado como funciona o mecanismo de filtragem depois de configurado devidamente. Quando ocorre a receção de alguma mensagem o identificador é comparado primeiro com os filtros do modo *identifier list*, se houver alguma ocorrência esta é armazenada nos *buffers FIFO*. Caso não haja nenhuma ocorrência para o modo *identifier list* este passa para o modo *mask* e novamente, se houver alguma ocorrência esta é armazenada nos *buffers FIFO*. Se a mensagem passar por estes 2 modos e não corresponder a nenhum dos filtros a mensagem é descartada por *hardware* sem nenhuma alteração ou ativação de nenhuma *flag* no *software* [1].

#### 4.3.5 Bit Timing

*Bit Timing* serve para monitorizar o barramento e garantir uma amostragem e um ajuste do *sample point* de modo a obter uma sincronização no primeiro bit por forma de todos os elementos da rede receberem todos ao mesmo tempo. Existem 3 segmentos com diferentes tamanhos onde temos que os configurar de modo a estabelecer sincronia com todos os nós. Na figura 4.9 é possível ver o processo de cálculo para cada um dos segmentos. Para facilitar este processo existem ferramentas online para ajuda desse cálculo (<http://www.bittiming.can-wiki.info/>). Muito rapidamente, se o *clock* do barramento responsável pelo controlador CAN estiver a operar a 48MHz

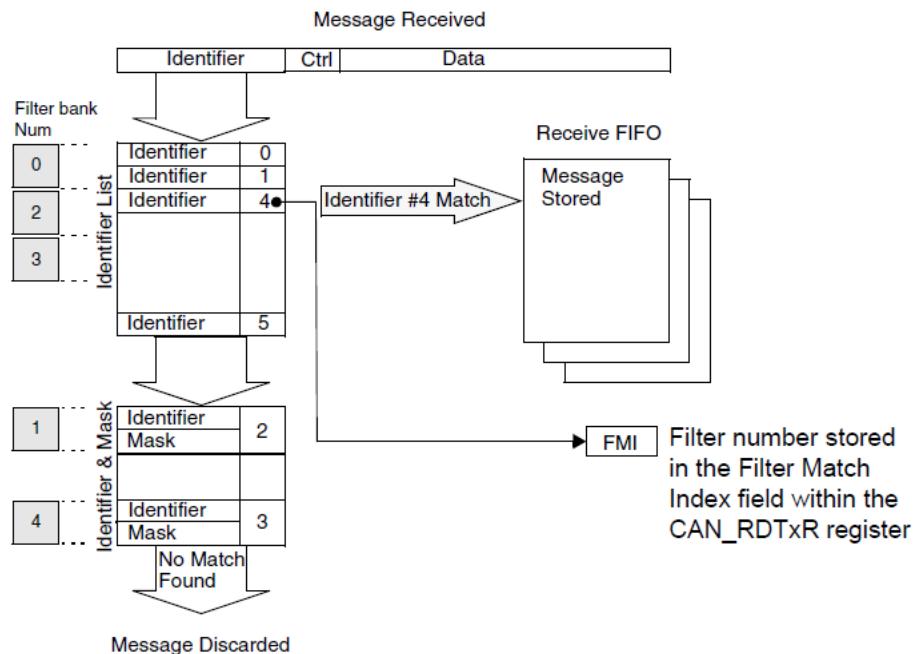


Figura 4.8: Mecanismo de filtragem de mensagens CAN [1]

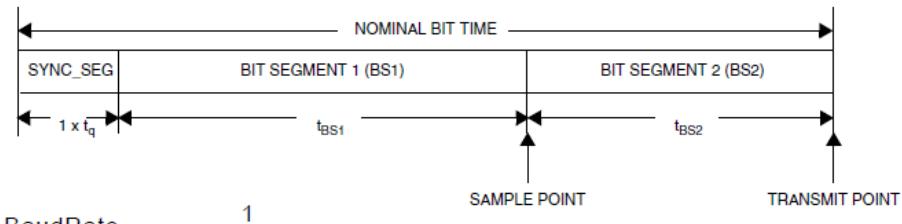
e se for necessário uma taxa de transferência de 500kbit/s, pode-se assim concluir que o *prescaler* terá de ser 6, tq de 16, o segmento 1 com tamanho 13 e o segmento 2 com 2, isto para o *sample point* aos 87.5%

#### 4.3.6 Bibliotecas

Existe uma grande variedade de bibliotecas para o microcontrolador STM32, desde as de mais baixo nível, até as mais *user friendly*. As que se irá utilizar (*Standard Peripherals Library Drivers*) são consideradas meio termo, nem muito baixo nível nem completa abstração do *hardware*.

Para iniciar a configuração deve se seguir uma série de processos:

- Ativar o periférico
- Reiniciar o controlador com as definições padrão
- Preencher a estrutura de dados referente ao controlador



$$\text{BaudRate} = \frac{1}{\text{NominalBitTime}}$$

$$\text{NominalBitTime} = 1 \times t_q + t_{BS1} + t_{BS2}$$

with:

$$t_{BS1} = t_q \times (\text{TS1}[3:0] + 1),$$

$$t_{BS2} = t_q \times (\text{TS2}[2:0] + 1),$$

$$t_q = (\text{BRP}[9:0] + 1) \times t_{PCLK}$$

where  $t_q$  refers to the Time quantum

$t_{PCLK}$  = time period of the APB clock,

$\text{BRP}[9:0]$ ,  $\text{TS1}[3:0]$  and  $\text{TS2}[2:0]$  are defined in the CAN\_BTR Register.

Figura 4.9: Bit Timing [1]

- Preencher a estrutura de dados referente ao filtro de mensagens
- Configurar interrupções (se necessário)
- Dar ordem de arranque ao controlador CAN

Como se vai utilizar 2 placas de desenvolvimento diferentes apesar do microcontrolador ser o mesmo as bibliotecas disponíveis diferem um pouco. São as duas com o mesmo nome, apenas difere a versão, enquanto a versão da placa Primer 1 é a V2.0.3 lançada a 22/09/2008 a versão da placa Sistema Mínimo é mais recente V3.3.0 lançada 16/04/2010.

## 4.4 Sensores

Nesta secção irá se falar dos vários sensores a utilizar de modo a deixar a cadeira de rodas o mais autónoma possível. Desde sensores inerciais até aos mais simples como sensores de luminosidade.

### 4.4.1 Sensor Ótico Infravermelho

Um sensor ótico é um dispositivo de curto alcance baseado num emissor de luz e um recetor, que apontam na mesma direção e se baseiam na capacidade de reflexão da luz no objeto e sua posterior deteção do raio refletido pelo recetor [10].

Estes tipos de sensores são maioritariamente usados em aplicações que envolvem AGV's e, através do mecanismo da reflexão da luz no objeto, tem como funcionalidade distinguir entre duas cores:

- Cor Preta: Onde a reflexão da luz é mínima;
- Cor Branca: Onde a reflexão da luz é máxima;

Este tipo de sensor é constituído por um diodo infravermelho a fazer o papel de emissor e um fototransístor como recetor, como é apresentada na figura 4.10.

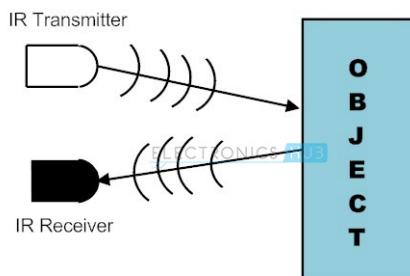


Figura 4.10: Constituição de um Sensor Ótico Infravermelho [13]

Como já dispomos de um modulo com sensor ótico então será esse que vamos utilizar. O esquema eletrico é apresentado na figura 4.11 e através do circuito pode-se concluir que a saída é uma saída digital e não analógica.

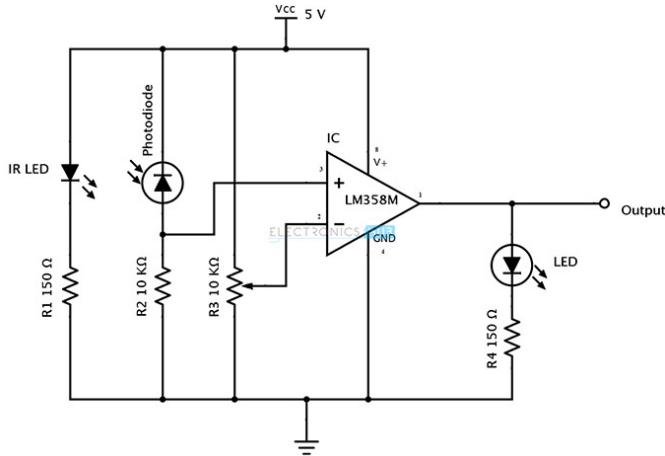


Figura 4.11: Esquema de ligações do Sensor Ótico Infravermelho com saída digital [13]

Este tipo de aplicação permite uma criação muito mais simples por parte do software, mas obriga a um *hardware* muito mais complexo [10].

Na figura 4.12 é apresentado o modulo do sensor ótico infravermelho utilizado no projeto. Este sensor tem de estar localizado numa posição estratégica de modo que a deteção do chão seja eficaz.



Figura 4.12: Modulo Sensor Ótico Infravermelho

Este modulo tem um potenciómetro como se pode ver na figura 4.11

apresentado como R3 e permite fazer um ajuste do valor de referencia para o comparador o que permite fazer assim uma calibração ao sensor.

#### 4.4.2 Sensor Ultrassónico - HC-SR04

Este tipo de sensores apresenta uma mecânica de funcionamento bastante idêntica à maioria dos instrumentos de localização e determinação de distâncias por som, como o sonar, isto é, emite uma onda sonora e espera até esta detetar um obstáculo, após a deteção, a onda volta até ao emissor e a distância entre o dispositivo e o obstáculo é obtida através do tempo que demorou entre a emissão da onda sonora e a sua receção.

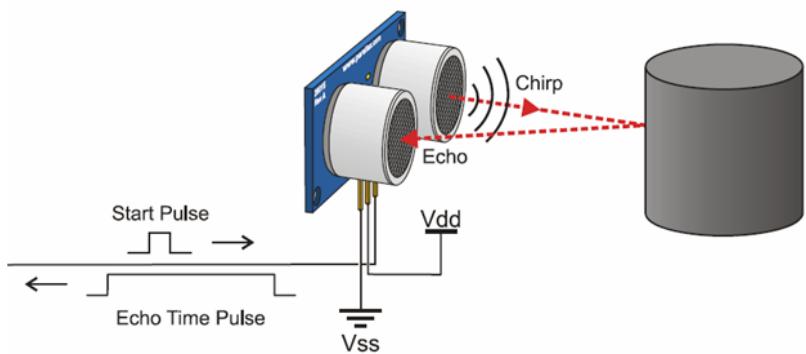


Figura 4.13: Funcionamento de um Sensor Ultrasónico

O HC-SR04 apresenta um alcance entre os 2cm-400cm e o cálculo da distância pode ser feito de acordo com o tempo em que o pino *ECHO* permaneceu em nível alto após o pino *trigger* ter sido colocado em nível alto.

Para iniciar a medição é necessário colocar um sinal em nível alto no *trigger* durante 10us. O sensor emite automaticamente uma onda sonora que ao encontrar um obstáculo rebaterá de volta em direção ao módulo. O tempo de emissão e receção do sinal será recebido no pino *ECHO* que ficará em nível alto. A distância pode ser calculada de acordo com o tempo em que o pino *ECHO* permaneceu em nível alto após o pino de *trigger* ter sido

colocado em nível alto (figura 4.14). A distância entre o sensor e o obstáculo pode ser obtida através da seguinte fórmula:

$$Distancia = \frac{(TempodoECHOemnívelalto \times Velocidadedosom(340m/s))}{2} \quad (4.1)$$

O resultado é obtido em metros, considerando o tempo em segundos, e a divisão por 2 na fórmula deve-se ao facto da onda ser emitida e recebida, logo percorre 2 vezes a distância pretendida [10].

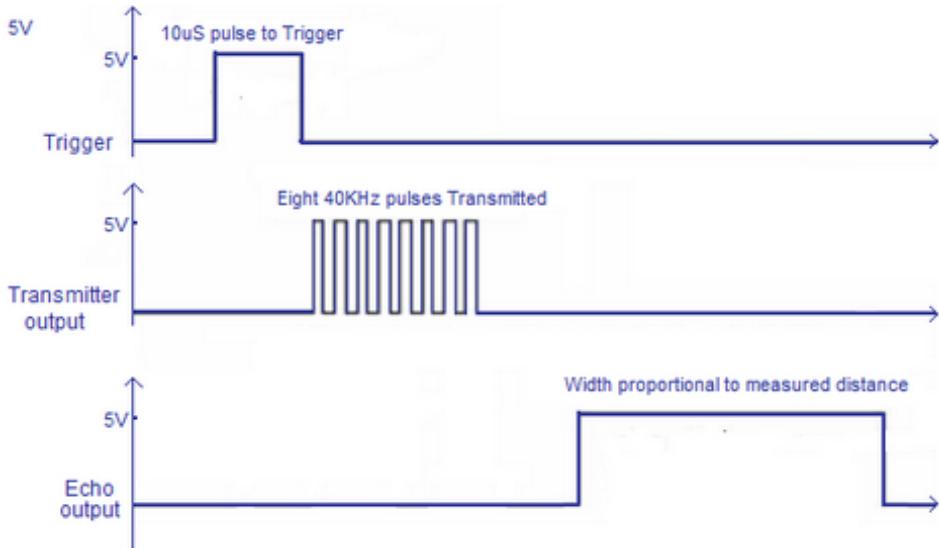


Figura 4.14: Princípio de funcionamento de um Sensor Ultrasónico

O *pinout* do HC-SR04 apresenta-se de uma maneira bastante simplificada na figura 4.15 [10].

O sinal de saída é digital e obtido quando o pino de *ECHO* é colocado a nível alto e os métodos matemáticos para obter a distância percorrida pelos impulsos enviados pelo sensor devem ser feitos por software.

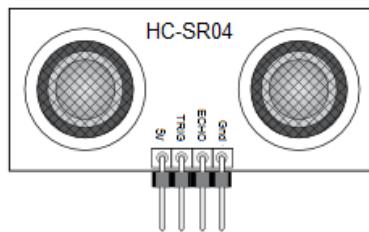


Figura 4.15: Pinout do Sensor Ultrasónico

#### 4.4.3 Sensor de Luminosidade - LDR

Um sensor de luminosidade do tipo LDR apresenta um modo de funcionamento bastante simples na aquisição de dados, pois a intensidade da luz que o sensor recebe é registada segundo uma variação no valor da sua resistência [10].

Devido ao facto dos resultados das leituras do sensor serem obtidos a partir do valor da resistência, este apresenta uma saída analógica e em resistência. A ligação do sensor ao microcontrolador deverá ser feita através de um conversor Analógico/Digital, sendo que a abordagem a utilizar é ler a saída de um divisor de tensão [10].

Um circuito equivalente da montagem para obter valores do sensor apresenta-se na figura 4.16.

Tal como o sensor ótico, este tipo de sensores apresenta como vantagem a simplicidade de funcionamento quer a nível de hardware como de software a implementar.

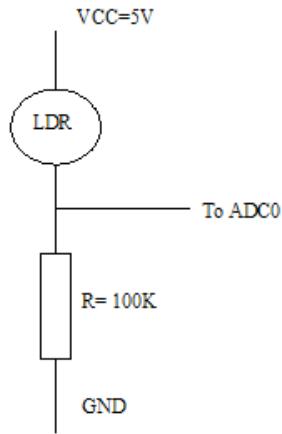


Figura 4.16: Esquema de ligações do LDR

#### 4.4.4 Acelerómetro LIS3LV02DL

Vulgarmente denominado Acelerómetro, o LIS3LV02DL é um sensor inercial de 3 eixos de saída digital que inclui uma interface capaz de obter a informação de um elemento sensorial e fornecer os sinais das acelerações medidas para a unidade de controlo através de SPI ou I2C (figura 4.17). O sensor apresenta diversas funcionalidades tais como [32]:

- Representação de dados programáveis a 12 ou 16 bits;
- Operação entre 2.6V a 3.6V;
- Interrupções ativadas por movimento;
- Auto-Testes Incorporados;
- Grande resistência a choques;
- Possui uma escala selecionável pelo utilizador de +2g, +6g;
- Capaz de medir acelerações numa banda de 640Hz para todos os eixos;
- Garante uma operação correta entre os -40°C até ao +85°C

A calibração do sensor vem por omissão calibrado para a maioria das operações que é normal atuar, podendo o utilizador recalibrar o sensor se assim for necessário. Sempre que o sensor é ligado, existem parâmetros que vão ser descarregados para os registos que são usados numa operação de modo normal, fazendo que o utilizador possa usar o dispositivo sem nenhuma anterior calibração.

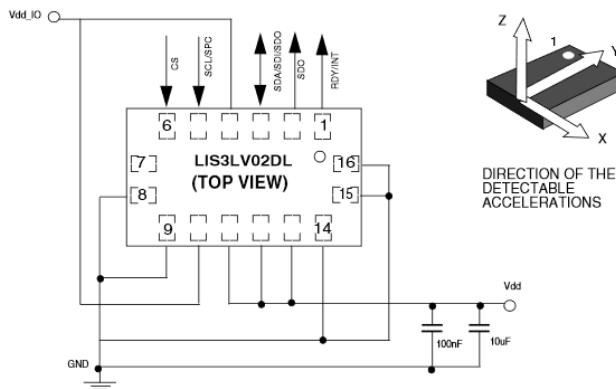


Figura 4.17: Constituição do Sensor LIS3LV02DL [32]

O Protocolo de Comunicação a utilizar será SPI e, segundo este protocolo, o sensor funciona como um barramento escravo, permitindo ler e escrever registos para o dispositivo. Utilizando SPI a comunicação é realizada através de 4 pinos: CS, SPC, SDI e SDO (figura 4.18).

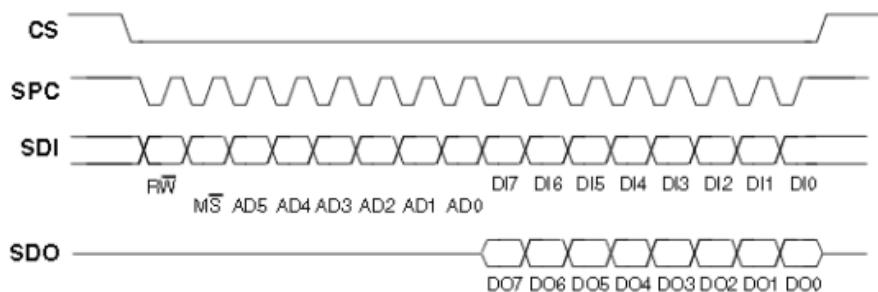


Figura 4.18: Funcionamento de uma trama SPI para o LIS3LV02DL

CS (*Chip Select*) - responsável por fazer a seleção do *slave* e é controlado pelo módulo mestre do SPI [32].

SPC (*Serial Port Clock*) – faz o controlo do sinal de relógio e é comandado pelo módulo mestre do SPI, sendo que este bit fica parado sempre que não há transmissão de dados (o bit CS encontra-se em nível alto) [32].

SDI (*Serial Data Input*) e SDO (*Serial Data Output*) – São os responsáveis pelo envio e receção dos dados, respetivamente, sendo que a transmissão de dados começa num flanco descendente no SPC e uma receção num flanco ascendente [32].

Tanto os comandos dos registos de Leitura como de Escrita são completos em 16 pulsos do sinal de relógio ou em múltiplos de 8 em caso de múltiplas leituras/escritas [32].

Estes tipos de sensores podem ser utilizados numa grande variedade de aplicações, tais como:

- Deteção de Queda-livre
- Navegação Inercial
- Dispositivos de Realidade Virtual
- Monitorização de Vibrações

## 4.5 Outros Periféricos

Apesar dos sensores referidos anteriormente, é necessário utilizar outros periféricos que mesmo sendo dispensáveis têm que ser incluído no projeto de modo a representar uma simulação mais real do protótipo de uma cadeira de rodas. No caso de uma luz traseira controlada por SPI, uma luz frontal, um módulo *bluetooth* e um LCD para mostrar alguma informação relevante.

### 4.5.1 Matriz de LED's - MAX7219

Para sinalizador traseiro da cadeira de rodas terá uma matriz de LED's vermelhos (figura 4.19). Para controlar a matriz de forma rápida é utilizado o MAX7219 que é um controlador de 8 *displays* de segmentos o que permite na verdade controlar 8x8 LED's independente da sua forma física. Na figura 4.20 é possível ver o esquema elétrico do MAX7219 conectado a uma matriz de LED's genérica.

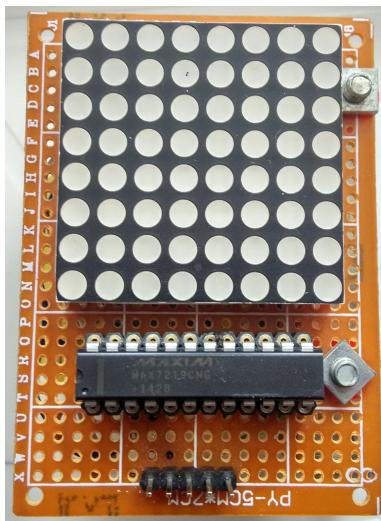


Figura 4.19: Sinalizador traseiro

MAX7219 é um circuito integrado que usa o protocolo SPI. Velocidade até 10MHz, 16bit de dados SPI, controlo de cada LED de forma individual, são algumas das características. Antes de operar com o MAX7219 devemos activar o *shutdown mode*, configurar o *display* como luminosidade pretendida, modo de *decode* (no caso, não é necessário *decode*) e por fim devemos desativar o *shutdown mode* de modo a entrar no funcionamento normal.

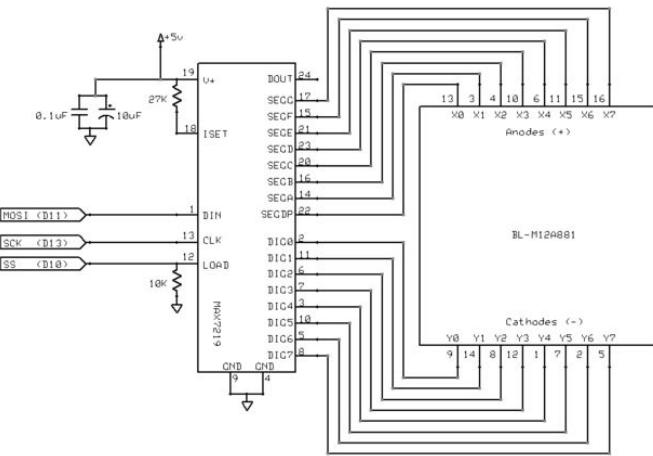
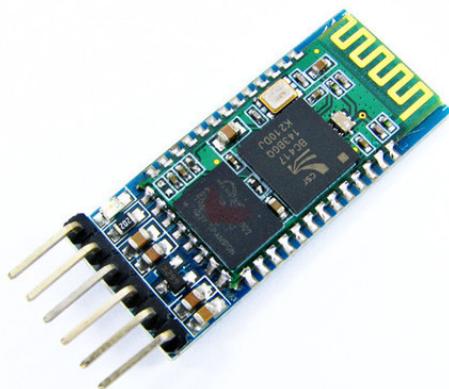


Figura 4.20: Esquema de ligações MAX7219

#### 4.5.2 Modulo *Bluetooth* - HC-05

O módulo *Bluetooth* HC-05 (figura 4.21) é utilizado para comunicação wireless entre o microcontrolador e algum outro dispositivo com *bluetooth*, como por exemplo um *smartphone*, um computador ou *tablet*. As informações recebidas pelo módulo são enviadas para o microcontrolador via serial (USART).

Figura 4.21: Modulo *bluetooth* HC-05

O HC-05 tem 6 pinos.

- VCC (alimentação de 3,6 a 6v)
- GND
- TX (nível logico 3.3v)
- RX (nível logico 3.3v)
- STATE
- EN

Para configurar este modulo é necessário enviar comandos AT via serial com um *baudrate* de 115200 bps Na tabela 4.1 é apresentado os comandos que o modulo interpreta como configurações.

Tabela 4.1: Lista de comandos AT

Comando	Descrição
AT	Teste
AT+RESET	<i>Reset</i>
AT+VERSION	Mostra a versão do firmware
AT+ORGL	Restaura as configurações padrão
AT+ADDR?	Mostra o endereço do modulo
AT+NAME	Mostra/Altera o nome do modulo
AT+ROLE	Seleciona o modulo <i>Master/Slave</i>
AT+PSWD	Muda a senha do modulo
AT+UART	Muda o <i>baudrate</i>
AT+RMAAD	Limpa a lista de dispositivos emparelhados
AT+INQ	Inicia a procura de dispositivos <i>Bluetooth</i>
AT+PAIR	Emparelha com o dispositivo <i>Bluetooth</i> remoto
AT+LINK	Faz a conexão com o dispositivo <i>Bluetooth</i> remoto

Para este projeto foi usado o modo *slave* um *baudrate* de 9600 bps e o nome e senha padrão. O nome é HC-05 e a senha é 1234 no ato de emparelhar o dispositivo.

### 4.5.3 LCD TFT - ILI9341

O LCD (*Liquid Crystal Display*) usa SPI como protocolo de comunicação. O ecrã é de 2.4 polegadas e tem uma resolução de 240x320. Funciona com uma tensão de 3.3V mas é tolerante aos 5V. A placa onde está montado o LCD tem também um leitor de cartões SD (*Secure Digital*) mas não vai ser usado neste projeto. Em questão de software para controlo deste periférico foi utilizada uma biblioteca

## 4.6 Teste dos Sensores

Durante esta fase do projeto foi feito alguns testes aos periféricos que vamos utilizar.

### 4.6.1 Protocolo CAN

Numa fase inicial, para testar o funcionamento do protocolo CAN utilizou-se 2 nós, sem filtros, apenas comunicação unidirecional. A função de configuração para o protocolo CAN está apresentado no Anexo A.1. Para verificar o perfeito funcionamento foi ligado um analisador lógico e foi obtido uma resposta satisfatória, na figura 4.22 é possível ver o *plot* de uma trama CAN. O *software* do analisador lógico já identifica o conteúdo da trama CAN, é possível ver claramente o valor do identificador, no caso com um valor de 236 e o campo data com 165, estes dados são em decimal, embora seja possível configurar para serem apresentados em binário ou em hexadecimal.

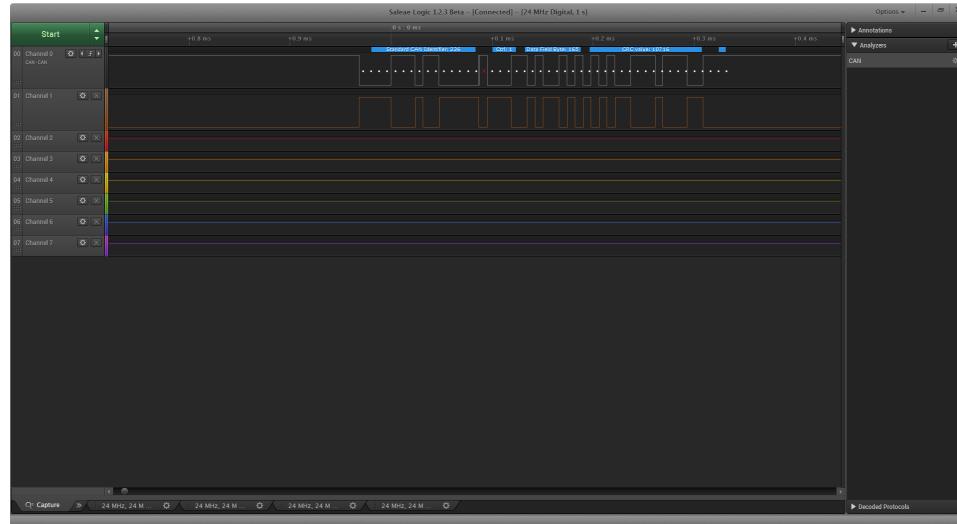


Figura 4.22: Plot dos níveis lógicos capturados na rede CAN

Para enviar e receber tramas CAN no STM32F103 a biblioteca em uso permite que apenas chamando a função seguinte permita enviar mensagens CAN.

```
1 CAN_Receive(CAN_FIFO0, &rx_message);
```

Basicamente recebe a trama e guarda os seus dados na estrutura rx\_message podendo depois ser acedida e retirar o identificador bem como a data da mensagem. CAN\_FIFO0 é onde o controlador CAN vai buscar a trama no caso ao *buffer* FIFO0.

Para enviar, existe uma função similar que basta indicar a estrutura.

```
1 CAN_Transmit(&canMessage);
```

Para este primeiro teste como já foi dito não se utilizou nenhum filtro de mensagens, no teste foi usado 2 placas e não foi configurada nenhuma rotina de interrupção.

### 4.6.2 Protocolo SPI

À semelhança do teste da rede CAN, foi feito um teste para o protocolo SPI visto que vai ser utilizado por outros periféricos como por exemplo o acelerómetro, LCD ou a matriz de LED's. No anexo A.2 é apresentado a configuração do periférico SPI, neste caso para o acelerómetro, podendo ser utilizada para outros periféricos mas será necessário a alteração de alguns parâmetros.

Para enviar dados é utilizado o seguinte código:

```

1 GPIO_WriteBit(GPIOB, GPIO_Pin_2,0); //ativar CS
2 //esperar que o buffer de saida esteja livre
3 while(!SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE));
4 SPI_I2S_SendData(SPI1, adress); //envio do 8 primeiros bits
5 //esperar que receba a mensagem
6 while(!SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE));
7 SPI_I2S_ReceiveData(SPI1); //Clear RXNE bit
8 while(!SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE));
9 SPI_I2S_SendData(SPI1, data);
10 while(!SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE));
11 rev=SPI_I2S_ReceiveData(SPI1);
12 GPIO_WriteBit(GPIOB, GPIO_Pin_2,1); //desativar cs

```

Primeiramente é ativado o pino CS (negado), logo depois é feita uma espera até que o buffer de saída esteja pronto (vazio), após isso é feito o envio dos 8 primeiros bits (endereço) e obrigatoriamente são recebidos 8 bits de "lixo". Depois é repetido o processo para indicar o que escrever no endereço do registo indicado anteriormente. A variável "rev" guarda o retorno do pedido, por exemplo o valor do eixo X.

Existe dispositivos que não recebemos informação de retorno, por exemplo o LCD ou a matriz de LED's. A comunicação é apenas feita num sentido.

Esta página foi intencionalmente deixada em branco.

## Capítulo 5

# Arquitectura e Projecto do Sistema

Neste capítulo será feita a descrição do projeto. Irá ser abordado vários pontos relevantes no seu desenvolvimento. Como o sistema tem quatro placas irá se atribuir o nome seguintes. Na figura 5.1 a vista frontal figura 5.1a, lateral figura 5.1c e traseira figura 5.1b do prototipo funcional.

**Placa 1** - Placa que dispõe de um LCD, um sonar e um LDR (*Light Dependent Resistor*), esta placa está posicionada estratégicamente na cadeira de rodas pois contem um sonar e um LCD, logo terá que estar acessivel ao utilizador e numa posição frontal.

**Placa 2** - Placa que está encarregue do controlo do *array* de LED's e de uma matriz de LED's vermelhos. Tem também um modulo de conectividade *Bluetooth* que fará a interação entre o utilizador e a cadeira de rodas.

**Placa 3** - Placa que faz uso de um sensor ótico para evitar que a cadeira caia perante escadas por exemplo. Ligado à placa existe também uma ponte H dupla para permitir o controlo dos 2 motores existentes na cadeira de rodas.

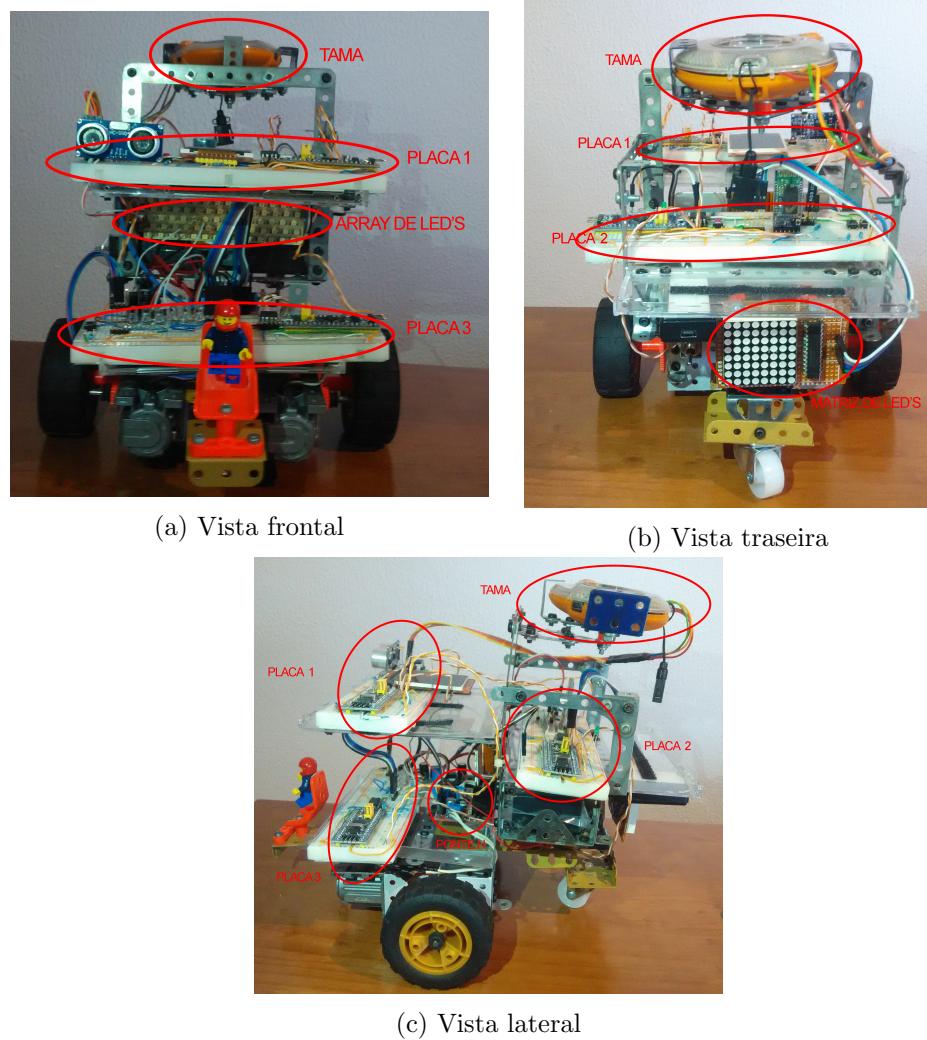


Figura 5.1: Vistas do prototipo funcional

**Tama** - Este é o nome dado à placa STM32 Primer1 da *Raisonance*

## 5.1 Descrição de *Hardware*

Neste secção iremos abordar alguns aspectos referente ao *hardware* do sistema. O sistema é constituído por 3 módulos STM32F103 e uma Placa de Desenvolvimento STM32 Primer 1, sendo que cada módulo do sistema é responsável pelo tratamento de informações provenientes de sensores e

atuadores específicos.

A rede CAN criada faz uso de um arquitetura *multi-master*, isto é, todos os intervenientes da rede podem tomar decisões sobre a informação que nesta circula, fazendo com que o sistema criado seja muito mais robusto e com um tempo de resposta muito mais curto. De maneira a ser mais percepível compreender todo o sistema criado, mostra-se em seguida, de forma individual o esquema elétrico de cada placa.

### 5.1.1 Placa1

Na figura 5.2 está ilustrado o esquema elétrico de todo o hardware utilizado na placa 1, controlado pelo ARM da família STM32F1.

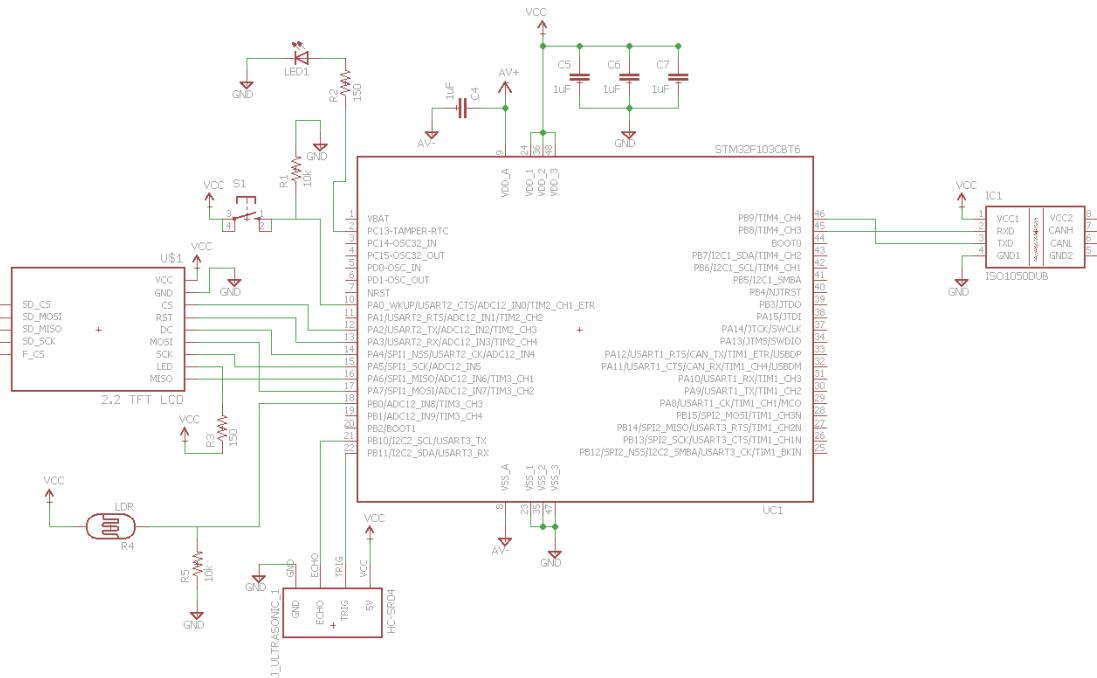


Figura 5.2: Esquema elétrico da Placa 1

Como se pode perceber pelo esquema, existem diferentes dispositivos acoplados ao microcontrolador STM32F103CBT6. De todos pode-se destacar o *Transceiver CAN ISO105DUB* que tem a principal função de receber os

dados provenientes do microcontrolador e transformá-los em dados legíveis e que sejam compreendidos dentro de uma rede CAN. Este dispositivo prima pela sua simplicidade na integração do hardware, pois somente necessita da ligação de dois pinos do lado do microcontrolador (RXD e TXD) e que resultam nos pinos CANH e CANL do lado da rede.

Os sensores também são dispositivos importantes na implementação de qualquer rede deste tipo, e daí estarem ligados dois sensores a este módulo: um sensor ultrassónico HC-SR04 e um sensor de luminosidade LDR.

O sensor ultrassónico está ligado aos pinos PB10 e PB11 do microcontrolador e tem a função de efetuar o varrimento do meio envolvente, detetando se existe algum obstáculo e qual a sua distância do sistema em questão. Já o sensor de luminosidade apenas necessita de um pino, de entrada analógica, o microcontrolador depois efetua os cálculos provenientes do conversor A/D, e consoante os resultados obtidos despoleta mudanças na luz do sistema, isto é, através de mensagens que circulam na rede CAN, este nó envia para a rede os valores obtidos no LDR que, por sua vez, são recebidos noutro nó que contém matriz e *arrays* LED's que acendem ou apagam. Ainda dentro deste nó encontra-se um LCD TFT de 2.2" que é usado para visualizar todos os valores provenientes da rede e, desta maneira, confirmar o seu correcto funcionamento e fornecer uma interface de output para o utilizador do sistema.

### 5.1.2 Placa 2

Na figura 5.3 é ilustrado o esquema elétrico do segundo (placa 2) nó que constitui o sistema, também controlado por um microcontrolador da família STM32F1.

Este nó do sistema também recebe o controlo através de um microcontrolador ARM STM32F103CBT6 e tem acoplado a ele dispositivos distintos do nó anterior, sendo que um dispositivo comum a todos os nós da rede é o

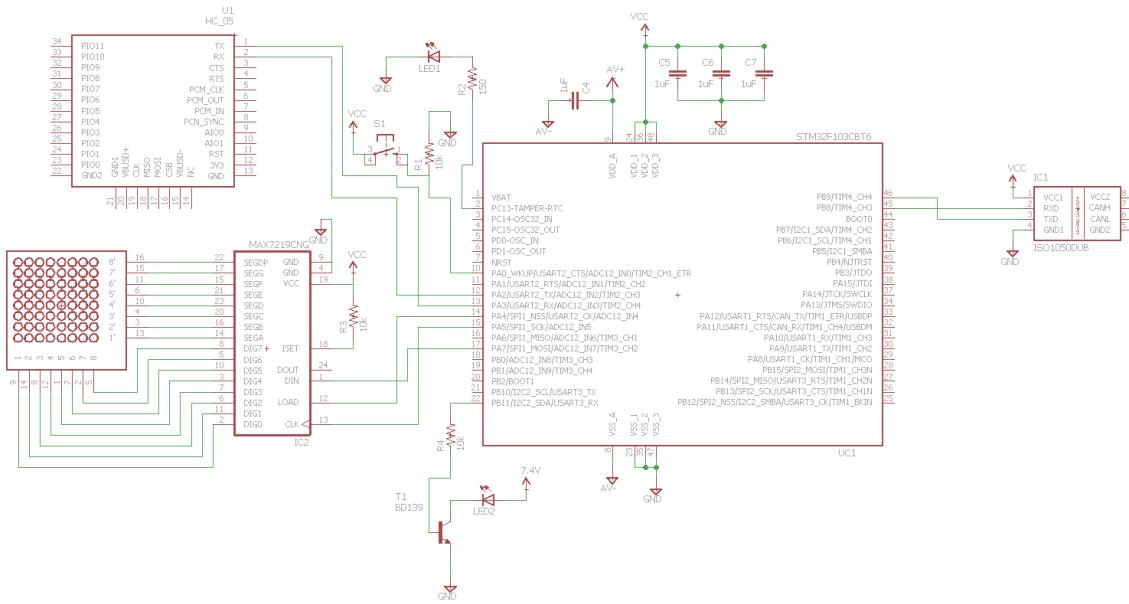


Figura 5.3: Esquema elétrico da Placa 2

transceiver CAN, de maneira a criar um padrão de conformidade na rede.

O Hardware ligado a este nó efetua ações importantes na rede, de entre as quais se destaca:

Matriz LED 8x8 - Atuador da rede CAN responsável por ilustrar padrões representativos de ações do sistema, como uma paragem de emergência ou arranque dos motores.

Módulo *Bluetooth* HC-05 - Responsável por receber comandos via *Bluetooth* provenientes de um *smartphone* com software Android exterior à rede CAN que são atuados nos motores do sistema, existentes no terceiro nó da rede.

*Array de LED's - No esquema elétrico apresentado como LED2, tem como função de ser ativo quando o ambiente rodeante à cadeira é fraco em luminosidade.*

### 5.1.3 Placa 3

Em seguida apresenta-se um esquema elétrico representativo do terceiro nó da rede, como mostra a figura 5.4.

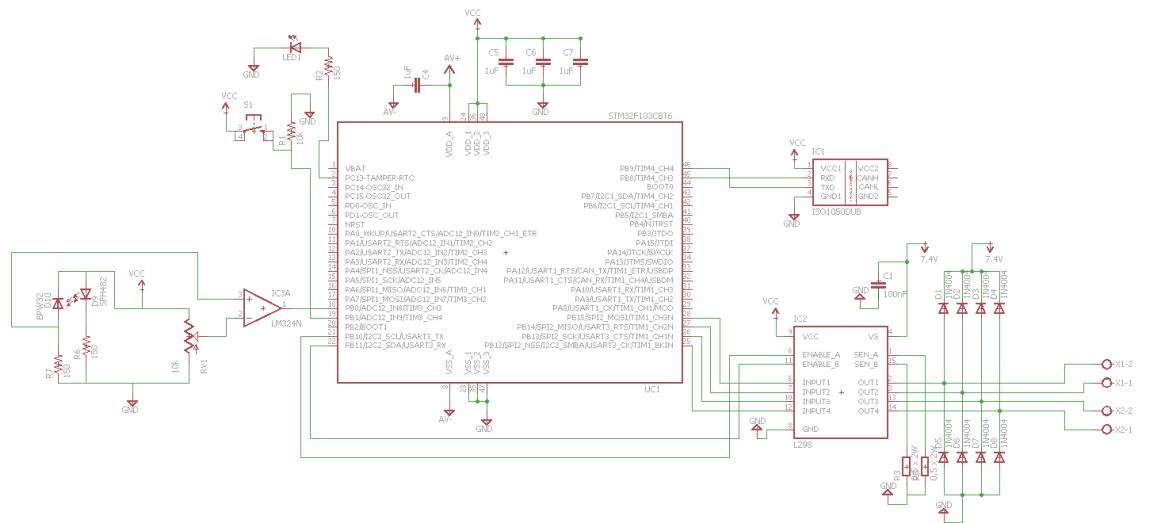


Figura 5.4: Esquema elétrico da Placa 3

Este nó, tal como os restantes, é controlado pelo microcontrolador STM32-F103CBT6 e é o responsável pelo envio dos comandos para os motores, envio esse feito a partir de quatro pinos específicos (PB12/13/14/15) e que coordenam as direções que a cadeira deve tomar (Frente, Trás, Esquerda, Direita). Além dos motores também existe acoplado ao ARM um sensor óptico ligado ao pino PB0 que indica se o carrinho detetou chão ou não à media que se movimenta. De maneira a existir comunicação de dados via rede CAN existe ligado aos pinos PB8 e PB9 o *transceiver* CAN.

## 5.2 Descrição do *Firmware*

Neste secção iremos abordar alguns aspetos referentes ao *firmware* do sistema dividindo por placas. Antes disso é necessário apresentar como foi

distribuído os endereços das mensagens CAN, de salientar que sempre que se refere aos endereços das mensagens estes são valores em hexadecimal.

**99** - É um endereço que indica de algum modo a paragem dos motores, este endereço é mais baixo que os restantes propositadamente, pois no caso de tráfego excessivo na rede mensagens com ID's mais baixos têm prioridade perante as com ID's mais altos.

**100** - Este é o endereço dedicado para indicar a velocidade dos motores. Tem um tamanho de 4 variáveis de 8 bits em que cada um deles contem valores de 0 até 255, valores estes que correspondem à força que se quer que a cadeira ande para a frente e para a direita por exemplo.

**101** - Endereço exclusivo para a indicar a que distância a cadeira de rodas se encontra de um obstáculo. A mensagem vai para o barramento já em centímetros pronta a ser interpretada por outro membro da rede.

**102** - Mensagens com este endereço contêm o valor do A/D onde está ligado o LDR. O ADC do microcontrolador é de 12 bits então a mensagem é enviada em duas variáveis de 8 bits cada. O membro da rede que necessitará desta mensagem terá que fazer o devido tratamento dos valores do A/D.

**103** - Mensagens circulantes no barramento com este endereço são referentes ao sensor ótico existente na parte dianteira da cadeira de rodas que informa a presença de chão.

**104** - Do mesmo formato das mensagens com o endereço 100 mas estas mensagens de velocidade, são provenientes do módulo de conectividade *bluetooth*.

**105** - Endereço reservado a mensagens para alterar a informação apresentada na matriz de LED's vermelhos situada na parte traseira da cadeira de rodas.

**106** - Para controlar de forma manual o *array* de LED's deverá ser enviada uma mensagem para a rede CAN com este endereço.

Quando ao funcionamento do sistema como um todo, temos 4 placas pode se dizer que a placa 1 tem o LCD onde é apresentado alguma informação relevante, um sonar que caso detete algum obstáculo a menos de 50 cm envia uma mensagem de emergência para a rede (99), um LDR que envia informação sobre a luminosidade que a cadeira de encontra no momento para a rede.

A placa 2 tem uma matriz de LED's vermelha, situada na parte traseira da cadeira de rodas que é acesa quando está um ambiente com menos luminosidade, ou quando a cadeira está parada para simulação das luzes de STOP, o *array* que é proporcional à condição de luminosidade, informação esta proveniente do barramento CAN com o ID 102. Tem ainda um modulo de *bluetooth* que tem como objetivo permitir o controlo da cadeira através de um *smartphone*, as mensagens recebidas via *bluetooth* seguem para o barramento CAN com o ID 104.

A placa 3 é utilizada para controlo dos motores, dispõe de duas pontes H e de um sensor ótico para detetar a presença de chão. Para o controlo dos motores pode ser feito de duas maneiras com mensagens com ID 100 ou com 104. Com a informação do sensor ótico é colocado na rede uma mensagem com o ID 0x0103.

Em todas as placas foi inicializado o periférico CAN de forma igual (com a exceção dos filtros), pois todos tem de funcionar ao mesmo *bit rate* e dentro dos mesmos modos. Para a configuração começa-se por ativar os periféricos, CAN e o porto onde os pinos estão ligados. Seguidamente são configurados os pinos como *alternate function* pois deixarão de ser pinos de carácter normal mas sim para uso exclusivo do CAN. Posteriormente é configurado o periférico CAN preenchendo uma estrutura onde é indicado o modo de funcionamento do CAN, *Prescaler*, prioridades das FIFOs, modo *automatic*

*wakeup*, entre outros. Depois é configurado os filtros, onde é dito quais são os endereços que a placa não vai ignorar, pode ser feita dizendo uma gama de valores (*mask*) ou por ID (*identifier list*). Como nesta aplicação são poucas mensagens a circular foi optado pelo método de ID. É possível visualizar na figura 5.5 o fluxograma correspondente à inicialização do CAN.

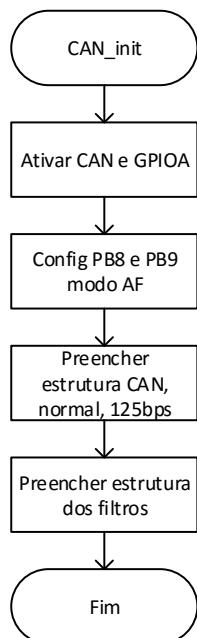


Figura 5.5: Fluxograma da inicialização do CAN

Para mais fácil compreensão do fluxo de mensagens é apresentada a tabela 5.1 onde é indicado o que cada placa recebe e envia do barramento CAN.

No anexo B.1 é possível ver na íntegra o código utilizado para fazer a inicialização completa. Com esta informação iremos analisar as placas de forma individual.

Tabela 5.1: Tabela de fluxo de mensagens CAN

Placa 1		Placa 2		Placa 3		Tama	
Envia	Recebe	Envia	Recebe	Envia	Recebe	Envia	Recebe
101	103	104	102	103	100	100	
102			105	105	104	99	
99					99		
105							

### 5.2.1 Placa 1

Falando apenas na placa 1 e apenas do ciclo principal desta placa é relativamente simples, de forma periódica é executada algumas funções, nomeadamente função para determinar a distancias até ao obstáculo, função para determinar a luminosidade. Seguidamente é enviada assim as mensagens CAN referente aos dados retornados pelas funções. Imediatamente a seguir irá verificar se recebeu alguma mensagem CAN se sim, então deverá processar essa mensagem. Na figura 5.6 é apresentado o fluxograma correspondente ao ciclo principal.

Seguidamente é apresentado o que contem acoplada e como é configurado os periféricos.

### LCD

Esta placa tem um LCD, este utiliza comunicação SPI. Para o seu funcionamento no stm32f103 teve de ser feito uma pequena adaptação à biblioteca pois a original estava implementada num stm32f407. As alterações foram ao nível das funções de enviar dados por SPI.

Para fazer a inicialização do LCD é executado o seguinte comando.

```
1 TM_ILI9341_Init();
```

Onde basicamente é feita a configuração dos pinos SPI no caso (PA5, PA6 e PA7), CS (*Chip Select*) (PA2), RST (*Reset*) (PA3) e DC (*Data/Command*) (PA4) logo de seguida é enviado uma serie de comandos via SPI para o LCD.

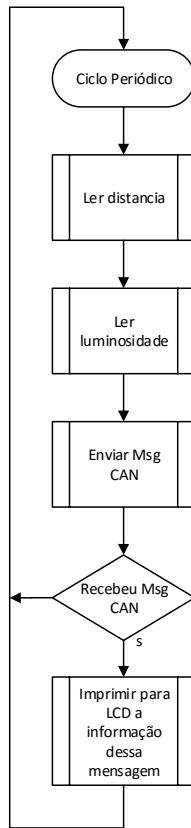


Figura 5.6: Fluxograma do ciclo principal da Placa 1

Depois de feita a inicialização devemos definir a orientação e limpar o ecrã para estar pronto para escrita então são executados as seguintes instruções de código.

```

1 // Orientacao vertical
2 TM_ILI9341_Rotate(TM_ILI9341_Orientation_Portrait_1);
3 // Pinta o ecrã de branco
4 TM_ILI9341_Fill( ILI9341_COLOR_WHITE);
```

Posto isto o LCD está pronto para ser utilizado, existe funções para es-

crever *strings* diretas no ecrã desenhar formas geométricas, e ainda carregar uma imagem para o ecrã.

O sistema tem uma serie de ecrãs e sempre que carregamos no botão (ligado ao PA0) é mudado de ecrã. Como se pode ver na figura 5.7. Dependendo do ecrã selecionado podemos ter informações por exemplo da distancia medida do sonar figura 5.7c ou a frequência de relógio que o sistema está a operar figura 5.7b.

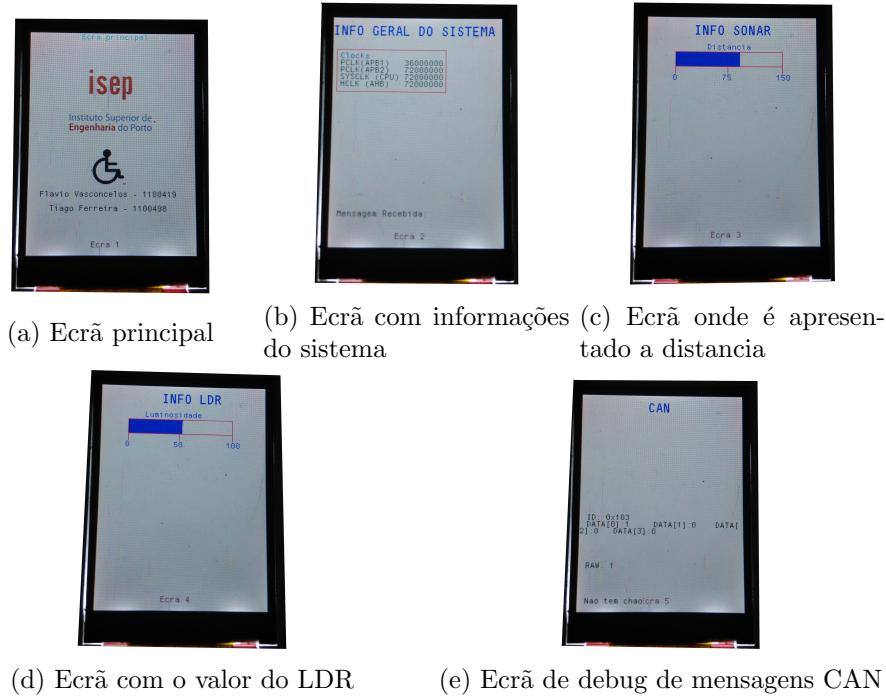


Figura 5.7: Ecrãs disponíveis no sistema

## Sonar

Passando agora para a configuração do sonar, tem-se a seguinte função apresentada no fluxograma da figura 5.8 que permite inicializar tudo o que é necessário para a sua utilização. É possível ver o código por completo no anexo B.2

Sabe-se que o *Trigger* está ligado ao pino PB11 e que está configurado

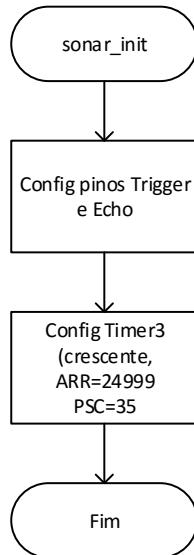


Figura 5.8: Fluxograma da função `sonar_init`

como uma saída *Push-Pull*. O *Echo* é configurado como entrada pois vamos estar à escuta do pino. Logo após a configuração dos pinos é colocado o pino do *Trigger* a nível baixo pois ainda não queremos iniciar uma medida.

Para executar uma medida temos que ter algo que nos meça o tempo desde que o *Trigger* é disparado até que recebemos algum impulso no pino *Echo*, então é configurado um *timer* no caso o TIM3 para auxiliar na contagem do tempo. Os valores 24999 e 35 são provenientes da seguinte formula [1].

$$T = \frac{PSC + 1}{CK\_INT} * (ARR + 1) \quad (5.1)$$

Fixando o valor do *prescaler* e do período, que no caso queremos de 25 milissegundos é fácil encontrar o valor do *auto-reload*.

$$0.025 = \frac{35 + 1}{36 * 10^6} * (ARR + 1) \quad (5.2)$$

$$ARR = 24999 \quad (5.3)$$

Vamos ter um *timer* com um período de 25 milissegundos e sendo o valor do *auto-reload* é proporcional ao tempo é mais fácil para a conversão entre o tempo decorrido e a distância, por exemplo se o valor do *timer counter* for 10000 isto quer dizer que já passou 10000 micro segundos ou 10 milissegundos desde que o *timer* foi ativo ou que o *timer counter* chegou ao topo, no caso 24999.

Para obter a distância do obstáculo, foi implementada a seguinte função descrita no fluxograma descrito na figura 5.9.

Fazendo uma breve descrição da função, esta começa por colocar o valor do *timer counter* a zero, seguidamente é enviado um impulso durante 10us no pino do *Trigger* e depois de certificar que o *Echo* está a nível lógico baixo no inicio da medição é feito um *reset* ao *timer counter*. Ficamos assim à espera que o *Echo* passe de nível lógico baixo para alto ou que chegue aos 17400, valor de *timeout* que corresponde a 150 cm, logo depois é desligado o *timer* de modo a parar o incremento do *timer counter*. Seguidamente é feita conversão de tempo para a distância, como o valor do *timer counter* é proporcional a micro segundos basta dividir esse valor por 116. Esse valor da distância é enviado para o barramento CAN com o endereço 101. Logo depois se o ecrã que estiver selecionado for o da distância é desenhado uma barra em função da distância do obstáculo. É possível ver o código completo no anexo B.3

Sempre que a distância é inferior a 50 cm esta placa envia uma mensagem 99 com um '1' ou um '0' dependendo da distância medida.

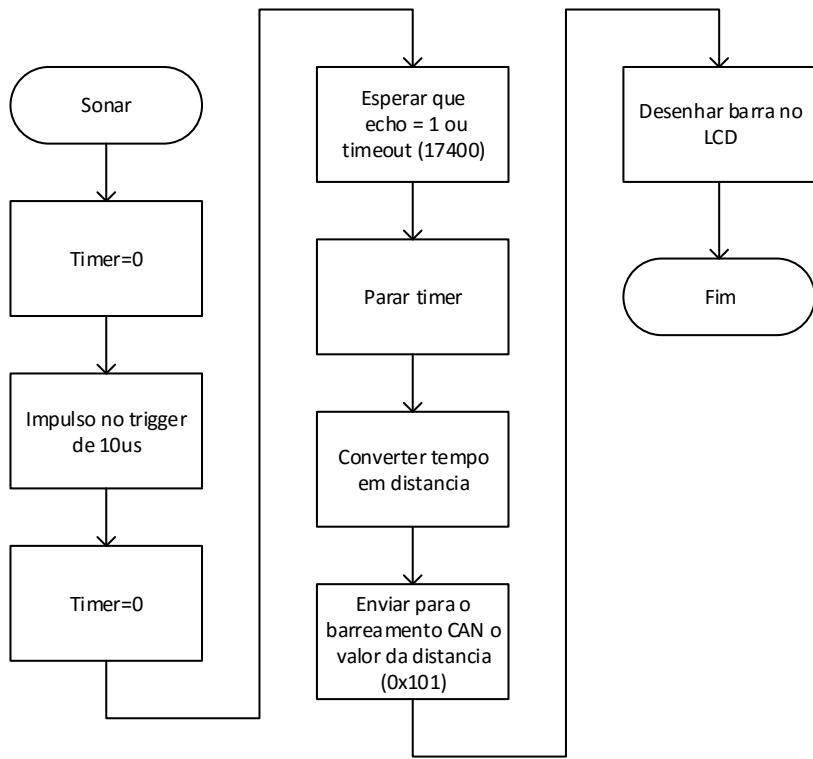


Figura 5.9: Fluxograma da função sonar

## LDR

Abordando agora o LDR, este está ligado ao PB0 ( ADC channel 8) através de um divisor de tensão. Na função de configuração do ADC, a primeira coisa a ser feita é a ativação do periférico ADC, seguidamente configurar o modo de funcionamento preenchendo uma estrutura genérica, onde é indicado a utilização de 12 bits, uma única conversão, entre outros. O ADC pode-se auto calibrar de modo a reduzir erros nas leituras devido a variações dos condensadores internos. O manual de referência [1] sugere que o ADC seja auto calibrado uma vez após a ativação do periférico ADC. Antes de iniciar a auto-calibração, o ADC deve estar estado de desligamento daí a função

*ADC\_ResetCalibration(ADC1);* Não esquecendo de da configuração do pino PB0 como entrada no inicio da função. Função de inicialização disponível no anexo B.4.

Sempre que seja necessário fazer uma conversão tem-se a seguinte função, bastante simples:

```

1  u16 readADC1(u8 channel) {
2      ADC-RegularChannelConfig(ADC1, channel, 1,
3          ADC_SampleTime_1Cycles5);
4      // inicia a conversao
5      ADC_SoftwareStartConvCmd(ADC1, ENABLE);
6      // espera que esteja completa
7      while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);
8      // retorna o valor obtido
9      return ADC_GetConversionValue(ADC1);
10 }
```

Esta função tem como parâmetro o *channel* do ADC que queremos fazer a leitura no caso do LDR, este está ligado ao *channel 8*. No fim a função retorna o valor obtido. Depois de se ter esse valor é manipulado (separado em 2 variáveis de 8 bits) depois é enviado para o barramento CAN com o ID de 102 como se pode observar no seguinte excerto de código.

```

1  adc = readADC1(ADC_Channel_8);
2  adc1 = adc \& 0xff; //mascara para dividir 12 em 2 de 8
3  adc2 = (adc >> 4);
4  canMessage.StdId = 102;
5  canMessage.ExtId = 0;
6  canMessage.RTR = CAN_RTR_DATA;
7  canMessage.IDE = CAN_ID_STD;
8  canMessage.DLC = 2;
9  canMessage.Data[0] = adc1;
10 canMessage.Data[1] = adc2;
11 CAN_Transmit(CAN1, \&canMessage);
```

### ***Debouncing***

Quando se tem botões no sistema deve se fazer *debouncing* de modo a que o nosso *click* no botão seja interpretado apenas como um impulso e não como uma serie deles. Pois como o botão é um objeto mecânico existe um tempo em o contacto oscila e cria uma variação (ruído) à entrada do microcontrolador como se pode ver na figura 5.10.

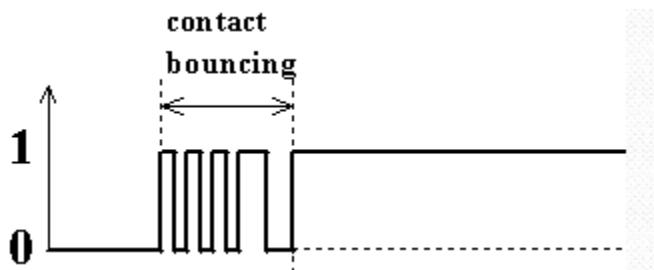


Figura 5.10: *Bouncing*

Este ruído pode ser retirado de duas forma ou por *hardware* ou por *software*. Para aplicações extremamente sensíveis e imunes ao ruído, é aconselhável utilizar *debouncing* por *hardware* (figura 5.11). Mas para pequenas aplicações o *debouncing* pode ser implementado por *software*. Existem diversos métodos e algoritmos para isso.

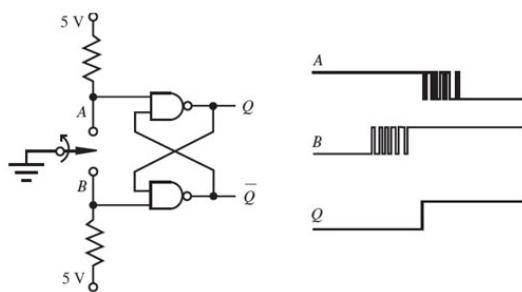


Figura 5.11: Circuito para *debouncing* por *hardware*

O *debouncing* está implementado da seguinte forma. Caso o botão esteja ligado a um pino de interrupção externa é utilizado um método. Caso não haja essa possibilidade foi implementada uma função que faz o *debouncing* por *polling*. No caso da interrupção externa tempos o fluxograma da figura 5.12 que serve de guia para a explicação.

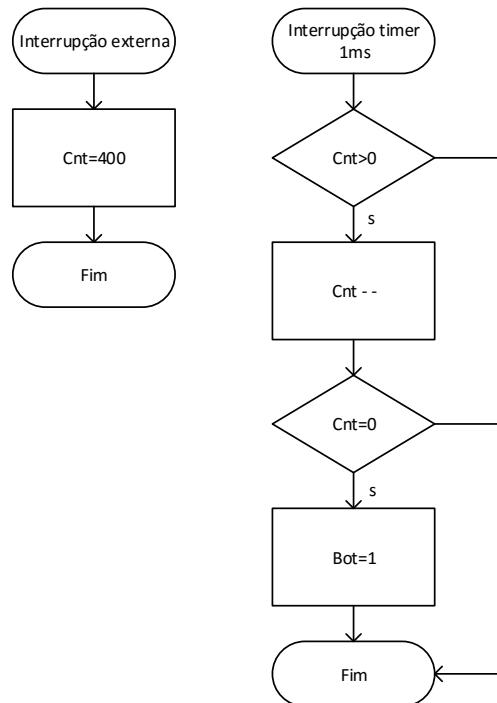


Figura 5.12: Fluxograma do método de *debouncing*

Temos 2 funções, uma que é executada periodicamente e outra que só é executada quando carregamos no botão. Sabemos que no instante o botão é pressionado irá ocorrer o *bouncing* e então a variável *cnt* estará a ser sempre recarregada, só quando o nível lógico do botão estiver estável é que vai ocorrer "tempo" suficiente para que a *cnt* chegue a 0, e quando isso acontecer é ativado uma *flag* que indica que o botão está definitivamente pressionado.

### 5.2.2 Placa 2

Descrevendo um pouco o funcionamento do ciclo principal este também é executado de forma periódica e relativamente simples. Sempre que receber uma mensagem via USART (vinda do modulo *bluetooth*) essa informação é tratada e posteriormente colocada da rede CAN. Depois caso tenha alguma mensagem CAN no barramento do seu interesse deve fazer o processamento conforme a mensagem que capturar, quer seja para a Matriz de LED's ou para o Array de LED's. Na figura 5.13 é apresentado o fluxograma para o ciclo principal desta placa.

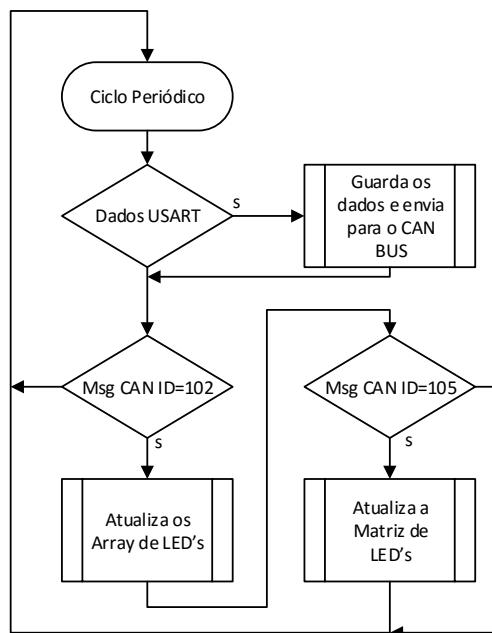


Figura 5.13: Fluxograma do ciclo principal da Placa 2

### Matriz LED's vermelhos

A matriz de LED's vermelhos situados na parte traseira da cadeira é controlada por SPI. É necessário fazer uma configuração do periférico e uma definição dos caracteres a inserir na matriz. Para melhor explicação é apresentado o seguinte fluxograma, da figura 5.14.

Primeiramente é ativado o periférico SPI1 e os pinos (PB5 PB6 e PB7) são configurados para ser utilizados para pelo SPI. Logo de seguida é configurado e ativado o SPI de acordo com as indicações fornecidas no *datasheet* [19] do MAX7219.

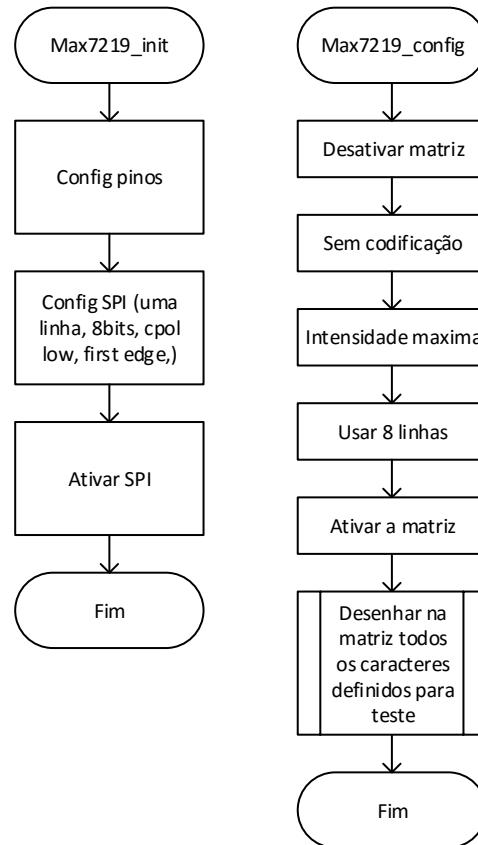


Figura 5.14: Fluxograma da inicialização e configuração do max7219

Para o arranque a configuração da matriz tem-se que fazer a configuração do MAX7219, o *datasheet* informa que primeiro se deve configurar o modo de descodificação, o brilho dos LED's, quantas linhas da matriz se vai utilizar e só depois desta configuração é que se a matriz está pronta para ser utilizada.

### **Array LED's**

Na cadeira de rodas existe um *array* de LED's brancos de forma a aumentar a luminosidade em ambientes mais escuros. O controlo desse *array* é feito através de um PWM. Para isso foi necessário configurar um *timer* no caso o TIM2 para gerar um PWM à saída do pino PB11. Este é configurado para receber valores de 0 a 4096 correspondente a 0 e 100% de brilho. Quem impõe o brilho do *array* é o LDR que está situado noutra placa, nomeadamente a placa 1. Os valores que são recebidos via CAN são valores de 0 a 4096 (12bits) mas sempre que a cadeira de rodas é ligada é feito uma calibração para definir que valores de ADC corresponde ao máximo e ao mínimo, então para isso é utilizado a seguinte formula apresentada em 5.4. É apenas uma maneira de mapear o valor.

$$pwm = (ADC - minimo) * (4096 - 0) / (maximo - minimo) + 0; \quad (5.4)$$

Após fazer o calculo basta executar a seguinte linha de código para o TIM2 atualizar a saída (PB11).

```
1 TIM_SetCompare4(TIM2, pwm);
```

Esta linha de código é sempre executado quando a placa 2 recebe uma mensagem com o endereço 102 que contem a informação do LDR.

### **Modulo Bluetooth**

O *bluetooth* que o sistema tem está ligado nesta placa e este utiliza como protocolo de comunicação a USART, as configurações usadas são as padrão

(8 bits, um *stop* bit, sem paridade, *baudrate*9600). É utilizado a interrupção de receção de mensagens, ou seja, sempre que recebe alguma mensagem via USART é executada uma rotina. O fluxograma dessa rotina é apresentada na figura 5.15.

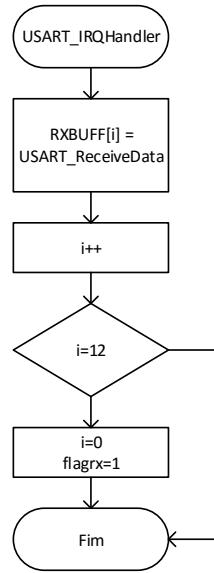


Figura 5.15: Fluxograma da rotina de interrupção de receção de mensagens via USART

A mensagem recebida segue um padrão de 4 grupos de 3 dígitos (FFFTT-TEEEEDDD) em que os 3 primeiros são valores de 000 a 255 e corresponde à para a frente, os seguintes correspondem à força para trás, esquerda e direita respectivamente. Essa mensagem recebida é enviada para o Barramento CAN com o endereço 100, pois a informação é proporcional à velocidade. A mensagem a ser enviada por CAN é um vetor de tamanho 4 em que na primeira posição vai o valor de 0 a 255 para a frente, na segunda, terceira e quarta posição segue o valor de 0 a 255 para trás, esquerda e direita respectivamente como se pode ver no seguinte excerto de código.

```

1 canMessage.StdId = 100;
2 canMessage.ExtId = 0;
3 canMessage.RTR = CAN_RTR_DATA;
4 canMessage.IDE = CAN_ID_STD;
5 canMessage.DLC = 4;
6 canMessage.Data[0] = frente;
7 canMessage.Data[1] = tras;
8 canMessage.Data[2] = esquerda;
9 canMessage.Data[3] = direita;
10 CAN_Transmit(CAN1, \&canMessage);

```

### ***Debouncing***

Esta placa usa 3 botões, dois deles é para definir o máximo e mínimo para o brilho do LCD. E outro é apenas para *debug*. E como já foi dito em 5.2.1 é boa pratica usar *debouncing* quando se usa botões. Então esta placa dispõe de 3 botões e todos eles tem implementado um sistema de *debouncing* por *software*.

#### **5.2.3 Placa 3**

A placa 3 que está responsável pelos controlos dos motores e pelo sensor ótico. O ciclo principal desta placa à semelhança das outras é com um ciclo periódico que primeiramente verifica a existência de chão e se a resposta for negativa este deverá parar os motores, faz também um varrimento à rede CAN e se esta conter uma mensagem com o ID 100 ou 104 ou 99 deverá processar as mensagens e atualizar o estado dos motores de acordo com a informação recebida das mensagens. No fluxograma da figura 5.16 é possível visualizar a sequencia de operações que o ciclo principal toma.

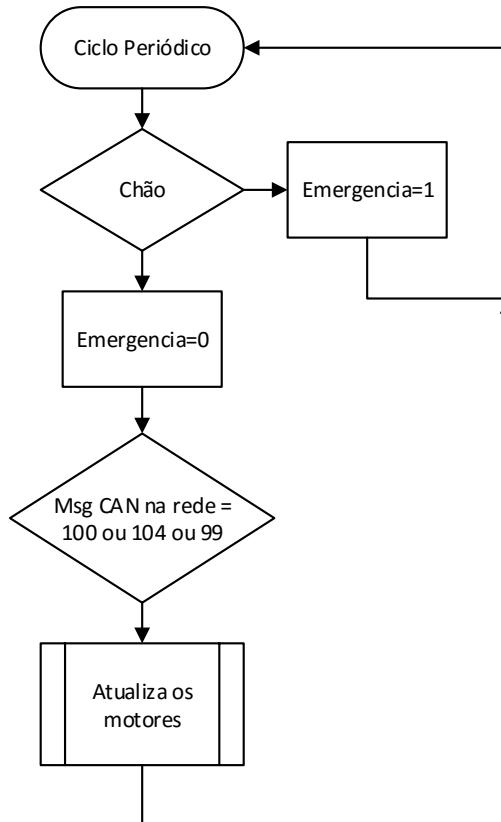


Figura 5.16: Fluxograma do ciclo principal da Placa 3

### Ponte H

Para fazer o controlo dos motores é utilizado 2 ponte H para isso é utilizado o driver L298n. Para controlar um motor temos 2 bits que definem o sentido e outro de *enable* que podemos aplicar um PWM para controlar a velocidade, na tabela 5.2 é possível ver os casos possíveis para o sentido do motor.

Para controlar o bit EN (*enable*) usando pwm é necessário configurar um *timer* à semelhança do que foi feito no *array* de LED's em 5.2.2, só que neste caso trabalhamos com 2 canais do TIM2. A gama de valores que

Tabela 5.2: Tabela da verdade para controlo dos motores

<b>EN</b>	<b>A</b>	<b>B</b>	<b>Sentido</b>
1	0	1	Frente
1	1	0	Trás
1	1	1	Parado
1	0	0	Parado
0	0	1	Parado
0	1	0	Parado
0	1	1	Parado
0	0	0	Parado

podemos aplicar no canais de pwm do *timer* é de 0 a 255, em que 0 o motor está parado e 255 a rotação do motor é máxima.

Sempre que é recebido uma mensagem com o identificador 100 significa que a função para atualizar os motores deve ser executada. Esta recebe por parâmetro os valores da força para frente,trás, esquerda, direita como é possível ver na seguinte declaração da função.

```
1 void motor( uint8_t frente , uint8_t tras, uint8_t esquerda,
      uint8_t direita);
```

Dentro desta função corre um algoritmo que faz dependendo do valor das variáveis que vão por parâmetro vai alterar diretamente o PWM ligados aos *enable* de cada motor, é apresentado o fluxograma (figura 5.17) da implementação da função "motor".

Depois de entrar na função ela é testado qual é o sentido, se para a frente ou para trás dependendo depois do valor da direção, se para a esquerda se para a direita, é alterado o valor do PWM de cada motor.

Por exemplo, existe a seguinte trama a circular na rede CAN

**ID** 100

**data(0)** 150

**data(1)** 0

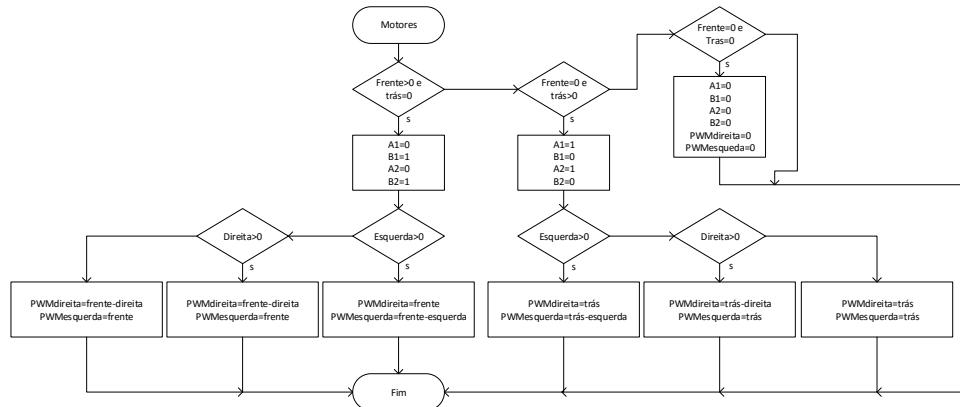


Figura 5.17: Fluxograma da função ”motor”

**data(2) 25**

**data(3) 0**

O ID 100 significa que é uma mensagem com informação para a velocidade. A mensagem é interpretada pela placa 3 e é chamada a função ”motor” de modo a atualizar o estado dos motores.

```
1 void motor(data[0], data[1], data[2], data[3]);
```

Com os valores da mensagem CAN sabemos que o PWM para a frente é 150 e queremos virar para a esquerda com uma diferença de 25, então sabe-se que para a cadeira curvar para a esquerda a roda da esquerda terá que andar menos que a direita. Então, depois de a função ser executada o PWM da roda esquerda assumirá o valor de  $150 - 25 = 125$  e o PWM da roda da direita terá o valor de 150, o que fará com que a cadeira de rodas curve ligeiramente.

### Sensor ótico

Depois de Testar o CNY70 e devido à estrutura do prototipo não se conseguiu obter resultados satisfatórios, alterou-se para um modulo com um

sensor ótico com mais alcance. Este modulo tem como saída, uma saída digital que o estado alto significa que tem chão, o oposto, nível lógico baixo significa que não tem chão. O valor do modulo está a ser verificado com um período de 100ms, caso a resposta do modulo seja nível baixo é ativada uma flag de emergência que vai impedir que o utilizador tenha controlo sob os motores e obriga os motores a parar. A flag só é limpa quando a cadeira voltar a ter chão. Para melhor entendimento é apresentado o fluxograma da figura 5.18.

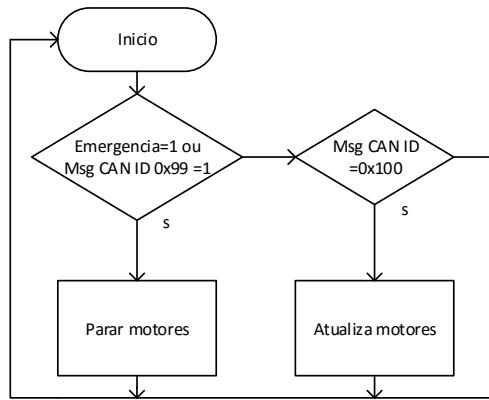


Figura 5.18: Fluxograma da situação de emergência

#### 5.2.4 Tama

Para o Tama é utilizado o acelerómetro para duas funcionalidades, para determinar se a cadeira teve alguma colisão ou simplesmente para controlar a cadeira de rodas. O acelerômetro tem a interface SPI e a configuração usada é uma configuração em si idêntica à utilizada na matriz de LED's apresentada em 5.2.2. Os valores provenientes do acelerómetro são de 12 bits, ou seja de 0 a 4096. Para qualquer um dos modos é necessário os valores do eixo X e do eixo Y, estes têm de ser mapeados para uma gama

de valores entre 0 255 de modo que o Tama quando estiver paralelamente ao chão o valor Seja 0 e que perpendicular seja 255.

Para uma explicação mais simples é apresentado o fluxograma (figura 5.19) caso o modo de comando esteja ativado é criada uma mensagem CAN com o endereço 100, em que os seus 4 campos são respectivamente: aceleração positiva no eixo dos Y, aceleração negativa no eixo dos Y, aceleração negativa no eixo dos X e aceleração positiva no eixo dos x. Ou de um modo mais claro, inclinação para a frente, trás, esquerda e direita.

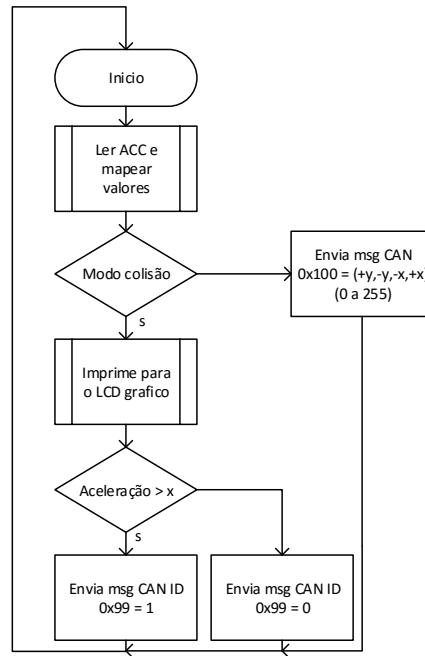


Figura 5.19: Fluxograma do funcionamento do Tama

O outro modo de funcionamento, é o modo de colisão. O Tama está preparado de modo a que quando ocorra acelerações acima de um determinado valor assume que ouve uma colisão, enviando imediatamente uma mensagem com o ID 99 obrigando os motores a parar imediatamente, neste modo

é apresentado no LCD um gráfico com duas linhas para apenas apresentar a inclinação da cadeira é possível visualizar o ecrã na figura 5.20. Para mudar de modo basta carregar no botão que a própria placa já dispõe.

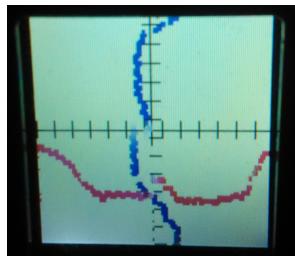


Figura 5.20: Ecrã do Tama

### 5.3 Android

De maneira a criar mais comodidade no controlo do sistema criado, e de fazer uma interface mais amigável para o utilizador, optou-se pela elaboração de uma aplicação móvel que corre em dispositivos Android e que, de todas as funcionalidades que tem, a mais importante passa mesmo pelo controlo na locomoção da cadeira de rodas.

É utilizada a tecnologia Bluetooth para o envio e receção de dados entre o *smartphone* e o sistema, e o hardware necessário para conseguir receber os dados provenientes do *smartphone* e trata-los para a rede denomina-se HC-05, um módulo que emula uma porta série através da tecnologia Bluetooth. Uma possível imagem representativa deste módulo, bem como o seu esquema elétrico apresenta-se na figura 5.21.

Este módulo tem sido amplamente utilizado e ganho popularidade pela comunidade eletrônica e em projetos maioritariamente envolvidos em temas como a ”internet das coisas”, IoT. O seu uso justifica-se pela facilidade da integração da tecnologia Bluetooth em qualquer sistema e pelo tratamento de dados simples que provêm do mesmo.

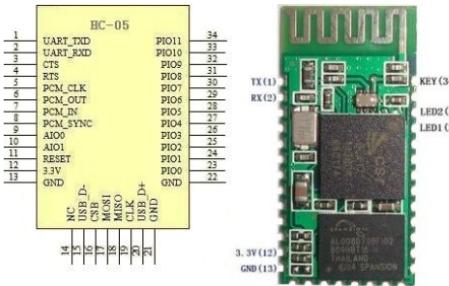


Figura 5.21: HC-05

Face á aplicação propriamente dita, esta foi elaborada em código nativo Android, no IDE Android Studio, sendo que uma imagem ilustrativa deste ambiente de desenvolvimento integrado apresenta-se na figura 5.22.

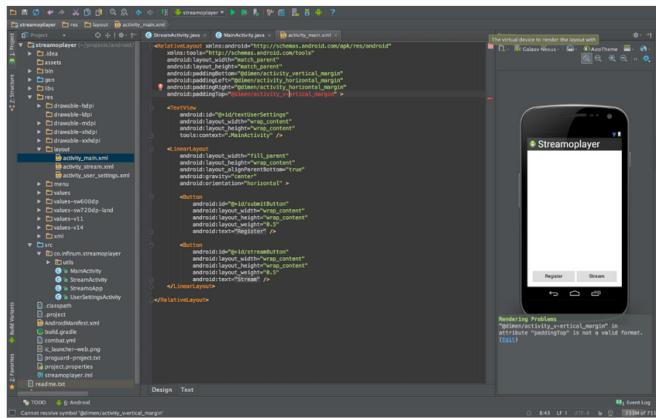


Figura 5.22: IDE Android Studio

Através do uso deste software é possível integrar todas as funcionalidades inerentes a um *smartphone* Android de uma maneira simples e organizada, incluindo o código que corre no *Back-end*, maioritariamente em JAVA, e o código de *Front-end*, em XML.

A Aplicação desenvolvida apresenta dois ecrãs principais com funções específicas:

- O Primeiro ecrã apresenta todas as configurações do Bluetooth no

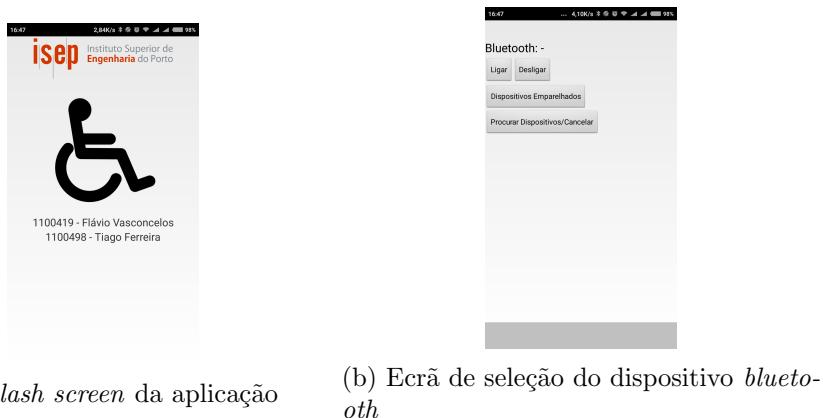


Figura 5.23

*smartphone*, de maneira a existir o emparelhamento entre este e o módulo HC-05;

- O Segundo ecrã é responsável por enviar comandos de direção para o módulo HC-05, que por sua vez, será interpretado nos motores do sistema;

Em seguida são apresentadas figuras ilustrativas da aplicação de maneira a criar uma espécie de mapa de navegação sobre a mesma. Na figura 5.23a é apresentado um simples ecrã *splash* de apresentação da Aplicação.

Na figura 5.23b é apresentado um menu para selecionar o dispositivo *bluetooth* a emparelhar. Relativamente ao ecrã das configurações de Bluetooth, é possível observar a existência de três botões:

- Botão ”Ligar”: Liga o Bluetooth do *smartphone*
- Botão ”Desligar”: Desliga o Bluetooth do *smartphone*
- Botão ”Dispositivos Emparelhados”: Efetua uma procura de todos os dispositivos anteriormente emparelhados com o *smartphone*

Na elaboração do código, e de maneira ao software tem conhecimento

que deve renderizar uma vista ou layout, é necessário estender a classe ”bluetooth1” para uma Atividade (linha 1).

Assim que uma Atividade é declarada deve-se utilizar as funções que definem o seu ciclo de vida (onCreate(), onPause(), onResume(), entre outras ...), daí ser definida a função onCreate na linha 3, que indica que todo o código presente nesta função deve correr assim que a Atividade é criada. De seguida é definido que o layout que deve mostrar ao utilizador enquanto esta Atividade permanecer ativa é o “bluetooth3” (linha 5).

```

1 public class bluetooth1 extends Activity{
2     Override
3     protected void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.bluetooth3);
6         overridePendingTransition(R.anim.fade_in, R.anim.fade_out);

```

Em seguida é feita a configuração do Bluetooth através da estrutura presente no SDK Android BluetoothAdapter

```

1 myBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
2 if(myBluetoothAdapter == null) {
3     onBtn.setEnabled(false);
4     offBtn.setEnabled(false);
5     listBtn.setEnabled(false);
6     findBtn.setEnabled(false);
7     text.setText("Status: nao suportado");
8     Toast.makeText(getApplicationContext(), "Dispositivo nao suporta
9         Bluetooth",
10        Toast.LENGTH_LONG).show();
}

```

Linha 1 – Inicialização do BluetoothAdapter;

Linhas 2 a 10 - Verificação se o *smartphone* possui *Bluetooth* incorporado, e caso este não possua todos os botões presentes no *layout* devem ficar

desativados e deverá ser mostrada uma mensagem ao utilizador alertando que o *smartphone* não possui suporte para tecnologia *Bluetooth*;

Após passar positivamente nesta verificação deverão ser executadas as funções referentes a cada um dos botões. No caso do botão ”Ligar”, o código da função que deve executar apresenta-se a seguir.

```
1 public void on(View view){  
2     if (!myBluetoothAdapter.isEnabled()) {  
3         Intent turnOnIntent = new  
4             Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);  
5         startActivityForResult(turnOnIntent, REQUEST_ENABLE_BT);  
6  
6     Toast.makeText(getApplicationContext(), "Bluetooth ligado" ,  
7         Toast.LENGTH_LONG).show();  
8 }  
9 else{  
10     Toast.makeText(getApplicationContext(), "Bluetooth ja esta  
11         ligado" ,  
12         Toast.LENGTH_LONG).show();  
13 }
```

Ao ser clicado, o botão ”Ligar” deve correr a função on() que efetua a verificação se o *Bluetooth* se encontra desligado ou se este já se encontra ligado.

Linhas 2 a 8 – Se o Bluetooth estiver desligado é efetuado um pedido via Intent para a ação “ACTION\_REQUEST\_ENABLE” que ativa o *Bluetooth* no *smartphone*, e é despoletada uma mensagem para o ecrã alertando que o *Bluetooth* foi ligado;

Linhas 9 a 12 – Caso o Bluetooth já se encontre ligado e o utilizador carregue no Botão, apenas é despoletada uma mensagem avisando que o Bluetooth já se encontra ligado, logo não é necessário proceder a mais ne-

nhuma ação;

No caso do Botão ”Desligar”, o código da função que deve executar encontra-se a seguir.

```
1 public void off(View view){  
2     myBluetoothAdapter.disable();  
3     text.setText("Status: Desconectado");  
4  
5     Toast.makeText(getApplicationContext(), "Bluetooth Desligado",  
6         Toast.LENGTH_LONG).show();  
7 }
```

Linha 2 – A variável corresponde ao BluetoothAdapter é desativada, fazendo com que o *Bluetooth* se desligue;

Linhas 3 a 6 – Despoletar mensagem avisando o utilizar que o *Bluetooth* foi desligado

No caso do Botão ”Dispositivos Emparelhados”, o código da função que deve executar encontra-se a seguir.

```
1 public void list(View view){  
2     // get paired devices  
3     pairedDevices = myBluetoothAdapter.getBondedDevices();  
4  
5     // put it's one to the adapter  
6     for(BluetoothDevice device : pairedDevices)  
7         BTArrayAdapter.add(device.getName() + "\n" +  
8             device.getAddress());  
9  
9     Toast.makeText(getApplicationContext(), "Mostrar Dispositivos  
10    Emparelhados",  
11        Toast.LENGTH_SHORT).show();  
12 }
```

Linha 3 – Através da função presente na SDK Android “getBondedDevices()” é feita a procura de todos os dispositivos anteriormente emparelhados com o *smartphone*; Linhas 6 e 7 – Por cada dispositivo encontrado é associado o seu nome, através da função “getName()”, e o seu endereço, através da função “getAddress()”

Após isto é criada uma lista de todos os dispositivos emparelhados e o utilizador deve escolher qual deles é o correto para o sistema em causa. O código relativo a este método apresenta-se no excerto seguinte, e é importante referir que este código deve permanecer dentro da função `onCreate()`.

```
1 myListView = (ListView)findViewById(R.id.listView1);
2 BTArrayAdapter = new ArrayAdapter<String>(this,
3     android.R.layout.simple_list_item_1);
4 myListView.setAdapter(BTArrayAdapter);
5
6 myListView.setClickable(true);
7 myListView.setOnItemClickListener(new
8     AdapterView.OnItemClickListener() {
9
10    public void onItemClick(AdapterView<?> arg0, View arg1, int
11        position, long arg3) {
12
13        Object o = myListview.getItemAtPosition(position);
14        switch(position){
15            default:
16                Intent i = new Intent(getApplicationContext(),
17                    MainActivity.class);
18                startActivity(i);
19                break;
20            }
21        }
22    });
23});
```

Linha 1 – Associação da variável “myListView” á lista criada no layout;

Linhas 2 e 3 – Criação de um *Array* que contêm todos os dispositivos emparelhados com o *smartphone* e associação deste *Array* á variável da lista criada na linha 1;

Linhas 5 e 6 – Aplica-se a ação de item clicável a cada um dos elementos da lista;

Linhas 8 a 15 – Assim que haja um click em algum dos itens da lista é obtida a posição do item na lista e executado um Intent que faz com que seja aberta a Atividade MainActivity;

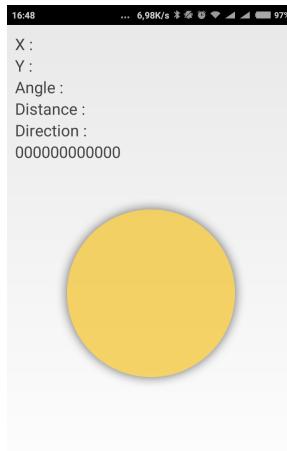


Figura 5.24: Ecrã de comando

Relativamente ao ecrã do envio de comandos (figura 5.24) para o HC-05, é possível observar a existência de diversas informações:

- *TextView* ”Trama”: Neste texto deverá aparecer a trama em tempo real que é enviada através do *joystick* para o HC-05, trama esta responsável pelo controlo dos motores da cadeira de rodas;
- *Joystick*: Interface gráfica responsável por enviar uma trama de valores para o módulo Bluetooth;

Tal como na Atividade “bluetooth1”, na “MainActivity” também é necessário estender a classe para uma Atividade, associar o layout (neste caso denominado ”main”) e criar a função `onCreate()`.

Dentro da função `onCreate()`, a classe deverá correr o seguinte código:

```
1 try {
2     findBT();
3     openBT();
4 } catch (IOException e) {
5     // TODO Auto-generated catch block
6     e.printStackTrace();
7 }
```

Pelo facto da transmissão de dados por Bluetooth ser uma operação de I/O, é necessário usar um ciclo *try/catch*.

Linhas 2 e 3 – Caso o *smartphone* consiga correr o código da maneira correta deve percorrer a função “`findBT()`” e, posteriormente, a função “`openBT()`”;

Linha 4 a 6 – Caso exista um erro de I/O, o software deverá gerar uma exceção alertando para um erro externo a este e não percorrer nenhuma função;

Relativamente à função “`findBT()`”, esta somente efetua as configurações e resultados vindos da Atividade “bluetooth1”, e indica se o dispositivo Bluetooth escolhido da lista é o correto para controlar a cadeira de rodas ou não.

A função “`openBT()`” acaba por ser a mais importante desta atividade pois é a responsável por efetuar a ligação e o emparelhamento entre o *smartphone* e o HC-05. O seu código apresenta-se em seguida.

```

1 void openBT() throws IOException {
2
3     UUID uuid =
4         UUID.fromString("00001101-0000-1000-8000-00805f9b34fb");
5
6     mmSocket = mmDevice.createRfcommSocketToServiceRecord(uuid);
7     mmSocket.connect();
8     mmOutputStream = mmSocket.getOutputStream();
9     mmInputStream = mmSocket.getInputStream();
10    joystick();
11}

```

Linha 3 – Associação de um UUID (do inglês *Universally unique identifier*) que é responsável por representar o ID de um protocolo de porta série usado no HC-05;

Linhas 5 e 6 – Criação e ligação de um servidor rfcomm através de Sockets ao UUID da linha 3;

Linha 7 – Criação de um canal de escrita;

Linha 8 – Criação de um canal de leitura;

Na linha 9 é executada a função *joystick* responsável por toda a comunicação entre o *joystick* virtual criado e os comandos enviados para o HC-05, sendo que o código mais importante correspondente a esta função apresenta-se no seguinte excerto:

```

1 layout_joystick.setOnTouchListener(new View.OnTouchListener() {
2     public boolean onTouch(View arg0, MotionEvent arg1) {
3         js.drawStick(arg1);
4         if(arg1.getAction() == MotionEvent.ACTION_DOWN
5             || arg1.getAction() == MotionEvent.ACTION_MOVE) {
6             textView1.setText("X : " + String.valueOf(js.getX()));
7             textView2.setText("Y : " + String.valueOf(js.getY()));

```

```
8 textView3.setText("Angulo : " + String.valueOf(js.getAngle()));
9
10 textView4.setText("Distancia : " +
11     String.valueOf(js.getDistance()));
12
13 int direction = js.get8Direction();
14 if(direction == JoyStickClass.STICK_UP) {
15     textView5.setText("Direction : Up");
16     if (js.getDistance()<100.0 && js.getDistance()>50.0) {
17         textView6.setText("075000000000");
18         sendData("075000000000");
19     }
20     else if (js.getDistance()<50.0 && js.getDistance()>0.0) {
21         textView6.setText("000000000000");
22         sendData("000000000000");
23     }
24     else if (js.getDistance()<150 && js.getDistance()>100.0) {
25         textView6.setText("125000000000");
26         sendData("125000000000");
27     }
28     else if (js.getDistance()>200) {
29         textView6.setText("255000000000");
30         sendData("255000000000");
31     }
32 }
```

Linha 1 e 2 – Criação de uma ação de toque em relação ao layout do *joystick* virtual criado

Linha 3 – Após existir um toque no *joystick* deverá ser desenhado o *stick* na posição respetiva;

Linhas 4 e 5 – Sempre que existir uma ação de toque com movimento no joystick é despoletada a evento ACTION\_MOVE do MotionEvent;

Linhas 6 a 9 – São retiradas informações relativas ao *joystick* em tempo real, assim que há ações de *touch*.

- função “getX()”: dá o valor do eixo do X corresponde ao sítio do *joystick* em que há um toque;
- função “getY()”: dá o valor do eixo do Y corresponde ao sítio do *joystick* em que há um toque;
- função “getAngle()”: dá o valor do ângulo do toque do *joystick*;
- função “getDistance()”: dá o valor da distância do sítio onde existiu o toque até ao centro do *joystick*;

Linha 11 – é obtida a direção na qual se está a exercer o toque do *joystick*, esta pode ser uma de 8 direções possíveis, como mostra a figura 5.25.

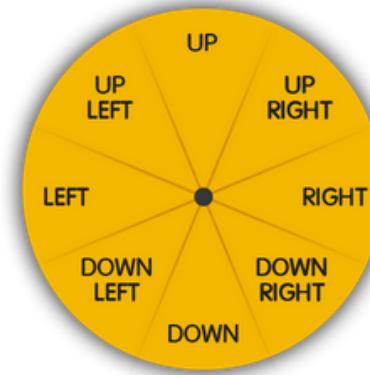


Figura 5.25: 8 possíveis direções de controlo

A trama de valores a enviar para o HC-05 compreende uma estrutura específica de 12 dígitos separados por 4 grupos de valores de 0 a 255, como por exemplo ”255.000.127.000”, sendo que:

- Os Primeiros 3 dígitos correspondem ao valor de PWM para o motor andar para a Frente;
- Os Segundos 3 dígitos correspondem ao valor de PWM para o motor andar para a Trás;

- Os Terceiros 3 dígitos correspondem ao valor de PWM para o motor andar para a Esquerda;
- Os Últimos 3 dígitos correspondem ao valor de PWM para o motor andar para a Direita;

Linha 12 e 13 – Caso o toque exercido no *joystick* esteja presente na área designada “UP”, é despoletada uma mensagem indicando que o utilizador quer andar com o carrinho para a frente;

Linhas 14 a 29 – É calculada a distância do toque ao centro do *joystick* e dividida em 4 patamares de 0 a 255:

- Caso a Distância esteja compreendida entre 50 e 100 – Deve ser enviada a trama ”075000000000”
- Caso a Distância esteja compreendida entre 0 e 50 – Deve ser enviada a trama ”000000000000”
- Caso a Distância esteja compreendida entre 100 e 150 – Deve ser enviada a trama ”125000000000”
- Caso a Distância seja superior a 200 – Deve ser enviada a trama ”255000000000”

As restantes 7 direções possíveis do *joystick* fazem uso da mesma abordagem da direção explicada anteriormente e daí não haver necessidade de se proceder á sua explicação mais em detalhe.

Esta página foi intencionalmente deixada em branco.

# Capítulo 6

## Resultados

Aqui são discutidos aspectos inerentes ao sistema final proposto e quais os caminhos a seguir para uma possível integração comercial do mesmo, sendo que neste caso, todo o projeto desenvolvido apenas serve como prova de conceito unicamente académica.

Posto isto, e de uma forma meramente informativa mostra-se na figura 6.1, o protótipo funcional.

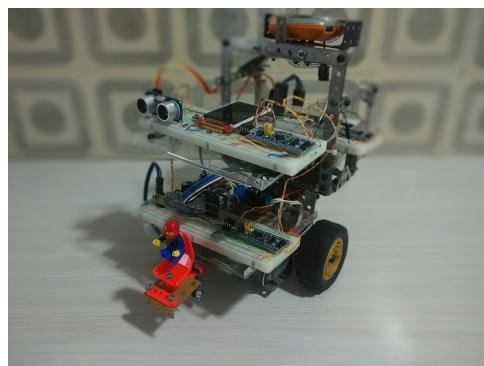


Figura 6.1: Protótipo funcional

Pode-se perceber a complexidade do sistema criado e como a rede CAN se interliga de forma ideal entre todos os nós. O número elevado de componentes interligados no sistema é uma das principais razões para a sua elevado

complexidade mas existe a noção de que uma rede CAN implementada em situações reais, como por exemplo em automóveis, tem um número muito superior de componentes interligados e necessita de uma ainda melhor gestão da rede.

O controlo da locomoção do sistema criado através de uma aplicação móvel foi o caminho seguido devido a esta ser uma das abordagens mais utilizada para este tipo de problemas, além de que atualmente o uso de *smartphones* apresenta uma escala global, fazendo com que um sistema deste tipo esteja disponível a qualquer utilizador.

De modo a ver as mensagens circulantes na rede foi colocado um analisador lógico e numa situação ideal em que a cadeira de rodas está parada é obtido os seguintes valores. É de salientar que é apenas um excerto das mensagens capturadas.

ID: 102	ID: 101	Data: 99	Data: 75	Data: 182	ID: 102	ID: 101	Data: 103
Data: 100	Data: 74	Data: 182	ID: 102	ID: 101	Data: 77	Data: 13	Data: 182
Data: 182	ID: 102	ID: 101	Data: 118	Data: 74	Data: 180	ID: 102	ID: 101
ID: 101	Data: 98	Data: 75	Data: 183	ID: 102	ID: 101	Data: 112	Data: 35
Data: 74	Data: 182	ID: 102	ID: 101	Data: 106	Data: 74	Data: 183	ID: 102
ID: 102	ID: 101	Data: 101	Data: 74	Data: 182	ID: 102	ID: 101	Data: 104
Data: 100	Data: 74	Data: 182	ID: 102	ID: 101	Data: 85	Data: 15	Data: 182
Data: 182	ID: 102	ID: 101	Data: 121	Data: 75	Data: 181	ID: 102	ID: 101
ID: 101	Data: 98	Data: 74	Data: 183	ID: 102	ID: 101	Data: 112	Data: 16
Data: 75	Data: 182	ID: 102	ID: 101	Data: 100	Data: 74	Data: 183	ID: 102
ID: 102	ID: 101	Data: 105	Data: 74	Data: 182	ID: 102	ID: 101	Data: 106
Data: 100	Data: 75	Data: 182	ID: 102	ID: 101	Data: 95	Data: 13	Data: 182
Data: 182	ID: 102	ID: 101	Data: 122	Data: 74	Data: 181	ID: 102	ID: 101
ID: 101	Data: 97	Data: 74	Data: 183	ID: 102	ID: 101	Data: 111	Data: 15
Data: 75	Data: 182	ID: 102	ID: 101	Data: 92	Data: 16	Data: 182	ID: 102
ID: 102	ID: 101	Data: 109	Data: 74	Data: 181	ID: 102	ID: 101	Data: 109
Data: 99	Data: 74	Data: 182	ID: 102	ID: 101	Data: 102	Data: 13	Data: 182
Data: 182	ID: 102	ID: 101	Data: 121	Data: 74	Data: 182	ID: 102	ID: 101
ID: 101	Data: 96	Data: 74	Data: 183	ID: 102	ID: 101	Data: 108	Data: 16
Data: 75	Data: 182	ID: 102	ID: 101	Data: 87	Data: 15	Data: 182	ID: 102
ID: 102	ID: 101	Data: 113	Data: 74	Data: 181	ID: 102	ID: 101	Data: 111
Data: 99	Data: 74	Data: 183	ID: 102	ID: 101	Data: 107	Data: 14	Data: 182
Data: 182	ID: 102	ID: 101	Data: 117	Data: 74	Data: 182	ID: 102	ID: 101
ID: 101	Data: 97	Data: 74	Data: 183	ID: 102	ID: 101	Data: 105	Data: 17
Data: 75	Data: 182	ID: 102	ID: 101	Data: 80	Data: 15	Data: 182	
ID: 102	ID: 101	Data: 116	Data: 74	Data: 181	ID: 102	ID: 101	
Data: 98	Data: 74	Data: 183	ID: 102	ID: 101	Data: 111	Data: 16	
Data: 182	ID: 102	ID: 101	Data: 111	Data: 74	Data: 182	ID: 102	

## *Capítulo 6*

---

É possível verificar assim, que a cadeira estando em *standby* apenas as mensagens 102 e 101 é que circulam na rede que corresponde ao valor lido do LDR e à distancia obtida do sonar respetivamente. Como não há mensagens com ID 100 a circular na rede, é possível concluir que a cadeira está parada.

Esta página foi intencionalmente deixada em branco.

## Capítulo 7

# Conclusão e Trabalho Futuro

Após o término deste projeto e alguma reflexão sobre o trabalho desenvolvido, chega-se à conclusão que a utilização de uma rede CAN num sistema que dispõe de vários sistemas embebidos é muito vantajosa quer pela sua robustez quer pelas sua flexibilidade, simplicidade, elevadas taxas de transferência. A capacidade que o protocolo CAN tem de endereçar mensagens, permite estabelecer prioridades nas mensagens podendo assim definir mensagens de extrema importância, como por exemplo mensagens de emergência, e mensagens com menos importância.

Atualmente este tipo de comunicação é muito utilizada quer em automóveis como em hospitais devido a sua viabilidade. A aplicação deste tipo comunicação numa cadeira de rodas não é ao acaso, na verdade a cadeira de rodas tem vários módulos. Estes têm de ser interligados entre si, o modulo que contem o *joystick* para controlo da cadeira, modulo de gestão da bateria, modulo do acionamento dos motores entre outros, significa que a ligação destes módulos tem de ser feita com mínimo de cablagem de modo a reduzir o peso e aumento da autonomia, a robustez e fiabilidade de modo a ser imune ao ruído eletromagnético , a possibilidade de *plug and play* para que facilmente seja substituída algum modulo, entre outros fazem que o protocolo CAN seja a melhor solução em termos de comunicação para uma

cadeira de rodas.

Com esta comunicação é possível adicionar módulos à rede sem qualquer alteração do software dos dispositivos já existentes na rede tornando assim esta rede muito versátil.

Para trabalho futuro de modo a aproximar o protótipo final de uma cadeira de rodas elétrica real seria interessante desenvolver uma placa (um modulo) para gestão da bateria da cadeira de rodas. Pois o órgão vital de uma cadeira de rodas elétrica é a bateria e atualmente a questão da gestão de energia é muito importante não só em termos de performance mas também em questão ecológica. Outro melhoramento seria a inclusão de um GPS de modo a fazer o *tracking* e em caso de acidente ter uma localização para os serviços de emergência. Inicialmente já tinha pensado mas por questões de prioridade não foi possível implementar esta funcionalidade, fica então como um possível trabalho futuro.

# Bibliografia

- [1] *STM32F101xx, STM32F102xx, ST M32F103xx, STM32F105xx and STM32F107xx advanced ARM -based 32-bit MCUs.*
- [2] Almost 25 years of can. *Control Engineering*, accessed in 2015.
- [3] *DeviceNet*, chapter R2. UFMG – Departamento de Engenharia Eletrônica, accessed in 2015.
- [4] Atmel. *ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet*, 2004.
- [5] BOSH. The configuration of the can bit timing, 2010.
- [6] R. Boys. Can primer: Creating your own network, 2013.
- [7] S. Corrigan. Introduction to the controller area network (can). Technical report, Texas Instruments, 2008.
- [8] J. Costa. Redes de computadores, 2010.
- [9] A. Devices. *ADM3053 Data Sheet*, 2004.
- [10] N. Dias. Tratamento de dados de sensores com uc, 2013.
- [11] S. Electric. Redes de comunicação industrial, 2007.
- [12] HMS. As-interface - actuator sensor interface, 2011.
- [13] E. Hub. Ir sensor, 2015.

- [14] C. in Automation (CiA). Can data link layer, accessed in 2015.
- [15] C. in Automation (CiA). Can implementation, accessed in 2015.
- [16] C. in Automation (CiA). Can physical layer, accessed in 2015.
- [17] N. Instruments. Controller area network (can) overview, 2014.
- [18] M. K. Kiejin Park. Advanced bit stuffing mechanism for reducing can message response time. Master's thesis, Ajou University (Korea), accessed in 2015.
- [19] MAXIM. Max7219/max7221, 2003.
- [20] E. Maye. Reliable data exchange in the automobile with can. 2008.
- [21] Microchip. *MCP25625 Data Sheet*, 2004.
- [22] Microchip. *PIC18FXX8 Data Sheet*, 2004.
- [23] NXP. *Industry's first integrated CAN transceiver microcontroller solution*, 2011.
- [24] Philips. *SJA1000 Stand-alone CAN controller APPLICATION NOTE*, 1997.
- [25] A. J. M. Pires. Comunicação de tempo-real em barramentos can baseados no controlador sja1000. Master's thesis, Universidade Moderna do Porto, 2005.
- [26] P. Portugal. Redes de campo feup, accessed in 2015.
- [27] Raisonance. Stm32-primer schematics, 2007.
- [28] H.-C. Reuss. Extended frame format - a new option of the can protocol. Technical report, Philips Semiconductors, 1993.
- [29] Scalevo. Web site. 2015.

- [30] Siemens. *On-Board Communication via CAN without Transceiver*, 1996.
- [31] SMAR. O que é profibus?, 2003.
- [32] ST. Mems inertial sensor lis3lv02dl. Technical report, ST, 2008.
- [33] STMicroelectronics. St spi protocol. Technical report, STMicroelectronics, 2013.
- [34] D. C. Watterson. Controller area network (can) implementation guide, 2012.

Esta página foi intencionalmente deixada em branco.

# Apêndice A

## Codigos de teste

### A.1 Rede CAN

```
1 void can_init(void){  
2  
3     GPIO_InitTypeDef      GPIO_InitStructure;  
4     CAN_InitTypeDef       CAN_InitStructure;  
5     CAN_FilterInitTypeDef CAN_FilterInitStructure;  
6     NVIC_InitTypeDef      NVIC_InitStructure;  
7  
8     /* Enable GPIO clock */  
9     RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN, ENABLE);  
10    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB , ENABLE);  
11    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO , ENABLE);  
12  
13    /* Configure CAN pin: RX */  
14    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;  
15    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;  
16    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
17    GPIO_Init(GPIOB, &GPIO_InitStructure);  
18  
19    /* Configure CAN pin: TX */
```

```
20 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
21 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
22 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
23 GPIO_Init(GPIOB, &GPIO_InitStructure);
24
25
26 // Remap2 is for PB8 and PB9
27 GPIO_PinRemapConfig(GPIO_Remap1_CAN , ENABLE);
28
29 /* CAN 1 cell init */
30 CAN_InitStructure.CAN_Prescaler = 18; //Valores calculados
31 CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
32 CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
33 CAN_InitStructure.CAN_BS1 = CAN_BS1_13tq;
34 CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
35 CAN_InitStructure.CAN_TTCM = DISABLE;
36 CAN_InitStructure.CAN_ABOM = DISABLE;
37 CAN_InitStructure.CAN_AWUM = DISABLE;
38 CAN_InitStructure.CAN_NART = ENABLE;
39 CAN_InitStructure.CAN_RFLM = DISABLE;
40 CAN_InitStructure.CAN_TXFP = DISABLE;
41 CAN_Init(&CAN_InitStructure );
42
43 /* CAN1 filter init, accept every message */
44 CAN_FilterInitStructure.CAN_FilterNumber = 0;
45 CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;
46 CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
47 CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
48 CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
49 CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0x0000 ;
50 CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
51 CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;
52 CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
```

```

53 CAN_FilterInit(&CAN_FilterInitStructure);

54

55 /* Message Init */
56 canMessage.StdId = 454;
57 canMessage.ExtId = 0;
58 canMessage.RTR = CAN_RTR_DATA;
59 canMessage.IDE = CAN_ID_STD;
60 canMessage.DLC = 1;
61 canMessage.Data[0] = 111;
62 }
```

## A.2 SPI

```

1 void max7219_init(void){
2     SPI_InitTypeDef SPI_InitStructure;
3     GPIO_InitTypeDef GPIO_InitStructure;
4     GPIO_StructInit (& GPIO_InitStructure);
5     SPI_StructInit (& SPI_InitStructure);
6
7     RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1 , ENABLE);
8     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA , ENABLE );
9     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC , ENABLE );
10    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB , ENABLE );
11
12    GPIO_DeInit(GPIOA);
13    GPIO_DeInit(GPIOB);
14    GPIO_DeInit(GPIOC);
15 //led debug
16    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_8 ;
17    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
18    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
19    GPIO_Init(GPIOC, &GPIO_InitStructure);
```

```
20 //MOSI PA7
21 //miso PA6
22 //sck PA5
23 //cs PB12
24 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2; //cs
25 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
26 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
27 GPIO_Init(GPIOB, &GPIO_InitStructure);
28
29 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_5; //mosi clk
30 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
31     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
32 GPIO_Init(GPIOA, &GPIO_InitStructure);
33
34 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
35 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; //miso
36 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
37 GPIO_Init(GPIOA, &GPIO_InitStructure);
38
39 GPIO_WriteBit(GPIOA, GPIO_Pin_5,0);
40
41 SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
42 SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
43 SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
44 SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
45 SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
46 SPI_InitStructure.SPI_NSS = SPI_NSS_Hard;
47 SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
48 SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
49 SPI_InitStructure.SPI_CRCPolynomial = 7;
50 SPI_I2S_DeInit(SPI1);
51 SPI_Init(SPI1,&SPI_InitStructure);
52 SPI_Cmd(SPI1,ENABLE);
```

52

53

54 }

Esta página foi intencionalmente deixada em branco.

## Apêndice B

# Funções de Inicialização

### B.1 can\_init

```
1 void can_init(void) {  
2     /* Enable perifericos */  
3     RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);  
4     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);  
5     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);  
6     /* RX */  
7     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;  
8     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;  
9     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
10    GPIO_Init(GPIOB, &GPIO_InitStructure);  
11    /* TX */  
12    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;  
13    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  
14    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
15    GPIO_Init(GPIOB, &GPIO_InitStructure);  
16    /*Remap1 PB8 e PB9*/  
17    GPIO_PinRemapConfig(GPIO_Remap1_CAN1, ENABLE);  
18    /* CAN 1 init */  
19    CAN_InitStructure.CAN_Prescaler = 18; //64
```

```

20 CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
21 CAN_InitStructure.CAN_SJW = CAN_SJW_1tq; //2
22 CAN_InitStructure.CAN_BS1 = CAN_BS1_13tq; //11
23 CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq; //4
24 CAN_InitStructure.CAN_TTCM = DISABLE;
25 CAN_InitStructure.CAN_ABOM = DISABLE;
26 CAN_InitStructure.CAN_AWUM = DISABLE;
27 CAN_InitStructure.CAN_NART = ENABLE;
28 CAN_InitStructure.CAN_RFLM = DISABLE;
29 CAN_InitStructure.CAN_TXFP = DISABLE;
30 CAN_Init(CAN1, &CAN_InitStructure );
31 /* CAN1 filtros */
32 CAN_FilterInitStructure.CAN_FilterNumber = 0; // 0..13 for CAN1,
33 // 14..27 for CAN2
34 CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdList;
35 //http://www.cse.dmu.ac.uk/~eg/tele/CanbusIDandMask.html
36 CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_16bit;
37 //
38 CAN_FilterInitStructure.CAN_FilterIdHigh = (0x0 << 5); ////
39 // substituir 0x0 por Id a
40 CAN_FilterInitStructure.CAN_FilterIdLow = (0x0 << 5); // receber
41 //CAN_FilterInitStructure.CAN_FilterMaskIdHigh = (0x0<<5); //
42 //CAN_FilterInitStructure.CAN_FilterMaskIdLow =(0x0<<5) ; //
43 CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;
44 CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
45 CAN_FilterInit(&CAN_FilterInitStructure);
46 }

```

## B.2 sonar\_init

```

1 void sonar_init(void) //timer 3 usado 25ms
2 {

```

```

3   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB , ENABLE);

4 //-----TRIG-----
5   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
6   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
7   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
8   GPIO_Init(GPIOB, &GPIO_InitStructure);

9
10 //-----ECHO-----
11  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
12  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
13  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
14  GPIO_Init(GPIOB, &GPIO_InitStructure);

15
16  GPIO_WriteBit(GPIOB, GPIO_Pin_11, 0); //desativar o pino trigg
17 // -----TIMER-----
18  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
19  TIM_TimeBaseStructure.TIM_Period = 24999; //auto-reload de 24999
20  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; //nao
21    tem divisao de clk
22  TIM_TimeBaseStructure.TIM_Prescaler = 35; //prescaler de 35
23  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
24    //modo de contagem crescente
25  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); //Enviar a
26    estrutura preenchida
27  TIM_Cmd(TIM3, DISABLE); //desliga o timer
28 }

```

### B.3 sonar

```

1 void sonar(void) {
2   TIM_Cmd(TIM3, ENABLE); //liga o timer
3   TIM_SetCounter(TIM3, 0); //coloca o timcont a 0

```

```

4   GPIO_WriteBit(GPIOB, GPIO_Pin_11, 1); //trig a 1
5   while (TIM_GetCounter(TIM3) <= 10); //espera 10us
6   GPIO_WriteBit(GPIOB, GPIO_Pin_11, 0); //trig a 0
7   while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_10) == 0);
8   TIM_SetCounter(TIM3, 0);
9   while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_10) == 1 &&
10      TIM_GetCounter(TIM3) < 17400);
11  TIM_Cmd(TIM3, DISABLE);
12  distancia = TIM_GetCounter(TIM3);
13  distancia = distancia / 116;
14  //mensagem can a ser enviada para a rede
15  canMessage.StdId = 0x101;
16  canMessage.ExtId = 0;
17  canMessage.RTR = CAN_RTR_DATA;
18  canMessage.IDE = CAN_ID_STD;
19  canMessage.DLC = 1;
20  canMessage.Data[0] = distancia;
21  CAN_Transmit(CAN1, &canMessage);
22  if (ecra == 2)
23  {
24      TM_ILI9341_DrawFilledRectangle(distancia + 40, 50, 40 + 150,
25          70, ILI9341_COLOR_WHITE);
26      TM_ILI9341_DrawFilledRectangle(40, 50, 40 + distancia, 70,
27          ILI9341_COLOR_BLUE);
28  }
29 }
```

## B.4 adc\_init

```

1 void adc_init(void) {
2
3     /* Ativar o periferico adc */
```

```
4     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
5     /* Carregar as configs padrao */
6     ADC_DeInit(ADC1);
7     /* PB0 (ADC Channel8) configurar como entrada analogica */
8     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
9     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
10    GPIO_Init(GPIOB, &GPIO_InitStructure);

11
12    //ADC1 Configuracao
13    //modo independente
14    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
15    // so um canal, sem scan
16    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
17    //unica conversao
18    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
19    //iniciar a conversao por software e nao por hardware
20    ADC_InitStructure.ADC_ExternalTrigConv =
21        ADC_ExternalTrigConv_None;
22    //12bits alinhados a direita
23    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
24    // um canal
25    ADC_InitStructure.ADC_NbrOfChannel = 1;
26    //inicia o ADC com a estrutura
27    ADC_Init(ADC1, &ADC_InitStructure);
28    // e ativa
29    ADC_Cmd(ADC1, ENABLE);

30
31    ADC_ResetCalibration(ADC1); //reinicia a calibracao
32    while (ADC_GetResetCalibrationStatus(ADC1)); //espera que seja
33        resetada
34    ADC_StartCalibration(ADC1); //feita uma nova calibracao
```

```

34     while (ADC_GetCalibrationStatus(ADC1)); //espera que seja feita
35         a calibracao
36 }
```

## B.5 max7219\_init e max7219\_config

```

1 void max7219_init(void) {
2
3     RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1 |
4                             RCC_APB2Periph_GPIOA, ENABLE);
5
6     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4; //cs
7     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
8     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
9     GPIO_Init(GPIOA, &GPIO_InitStructure);
10
11    GPIO_SetBits(GPIOA, GPIO_Pin_4);
12
13    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_7 |
14        GPIO_Pin_6; //miso mosi clk
15    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
16    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
17    GPIO_Init(GPIOA, &GPIO_InitStructure);
18
19    SPI_I2S_DeInit(SPI1);
20    SPI_InitStructure.SPI_Direction = SPI_Direction_1Line_Tx; // so tx
21    SPI_InitStructure.SPI_Mode = SPI_Mode_Master; // configurado como
22        master
23    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; //tramas de
24        8bits
```

```

22 SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low; // clock is high
23   when idle
24 SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge; // data sampled at
25   first edge
26 SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
27 SPI_InitStructure.SPI_BaudRatePrescaler =
28   SPI_BaudRatePrescaler_4; //Prescaler de 4
29 SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;// transmissao
30   com os bits mais signficativos primeiro
31 SPI_InitStructure.SPI_CRCPolynomial = 7;
32 SPI_Init(SPI1, &SPI_InitStructure);
33 SPI_Cmd(SPI1, ENABLE); // ativar SPI1
34 }
```

```

1 #define REG_DECODE      0x09
2 #define REG_INTENSITY   0x0a
3 #define REG_SCAN_LIMIT  0x0b
4 #define REG_SHUTDOWN    0x0c
5 #define REG_DISPLAY_TEST 0x0f
6 #define INTENSITY_MIN   0x00
7 #define INTENSITY_MAX   0x0f
8
9 void max7219_config(void) {
10   int teste = 0;
11   SPI_send(REG_SHUTDOWN, 0x00); //desativar matriz
12   SPI_send(REG_DECODE, 0x00); //sem codificacao
13   SPI_send(REG_INTENSITY, 0x05); //intensidade max
14   SPI_send(REG_SCAN_LIMIT, 0x07); //usar as linhas todas 8
15   SPI_send(REG_SHUTDOWN, 0x01); //ativar matriz
16   max7219_draw(MAX7219_CLEAR);
17
18   for (teste = 0; teste < 22; ++teste) {
19     Delay(1000000);
```

```
20     max7219_draw(teste);  
21 }  
22  
23 }
```