UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO DE ELÉTRICA

BACHARELADO EM ENGENHARIA ELÉTRICA

EDUARDO FELIPE CORDEIRO PINTO

APRESENTAÇÃO DO TRÁFEGO DE DADOS DE UM BARRAMENTO CAN EM UMA INTERFACE GRÁFICA DESENVOLVIDA EM JAVA.

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2018

EDUARDO FELIPE CORDEIRO PINTO

APRESENTAÇÃO DO TRÁFEGO DE DADOS DE UM BARRAMENTO CAN EM UMA INTERFACE GRÁFICA DESENVOLVIDA EM JAVA.

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Engenharia Elétrica, do Departamento de Elétrica, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Frederic Conrad Janzen

PONTA GROSSA

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento de Elétrica
Bacharelado em Engenharia Elétrica

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

APRESENTAÇÃO DO TRÁFEGO DE DADOS DE UM BARRAMENTO CAN EM UMA INTERFACE GRÁFICA DESENVOLVIDA EM JAVA.

por

EDUARDO FELIPE CORDEIRO PINTO

Este Trabalho de Conclusão de Curso foi apresentado em 21 de novembro de 2018 como requisito parcial para a obtenção do título de Bacharel em Engenharia Elétrica. O(A) candidato(a) foi arguido(a) pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Dr. Frederic Conrad Janzen
Orientador

Prof. Dr. Max Mauro Dias Santos
Membro Titular

Profa. Dra. Mônica Hoeldtke Pietruchinski
Membro Titular

Prof. Dr. Josmar Ivanqui
Responsável pelos TCC

Prof. Dr. Jeferson José Gomes
Coordenador do Curso

A Folha de Aprovação assinada encontra-se no
Departamento de Registros Acadêmicos da UTFPR – Câmpus Ponta Grossa

Dedico este trabalho a minha família e aos meus amigos, pelos momentos de ausência.

AGRADECIMENTOS

A realização deste trabalho não seria possível sem o apoio dos meus pais, Eni e Joani, que sempre buscaram oferecer à mim as oportunidades que não tiveram.

Agradeço a Universidade Tecnológica Federal do Paraná, pela troca de conhecimento gerada desde minha formação técnica.

A *UTForce E-Racing*, pelos dois anos de pesquisa e desenvolvimento, que me tornaram apto a discorrer este trabalho de conclusão de curso.

A minha namorada, Vivian, por me apoiar na hora que mais precisei no decorrer deste trabalho.

Aos meus amigos Felipe, Gabriel, Guilherme, Lucas, Manoel, Miguel e Walace, pela grande parceria desde o início do curso.

Ao meu orientador, Frederic, que durante minha graduação se mostrou mais que um professor: um grande amigo.

Enfim, a todos os que de alguma forma contribuíram para a realização deste trabalho.

RESUMO

PINTO, Eduardo Felipe Cordeiro. **Apresentação do tráfego de dados de um barramento CAN em uma interface gráfica desenvolvida em Java.**. 2018. 188 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Elétrica) – Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

Devido ao baixo número de referências sobre a conexão entre interfaces gráficas com microcontroladores PIC, conectados a um barramento CAN, optou-se por desenvolver um documento que exponha as principais características desenvolvidas neste projeto. Para isto, realizou-se uma revisão bibliográfica com relação aos modelos de comunicação de dados utilizados, bem como uma breve introdução as linguagens de programação. Como resultado desta pesquisa, desenvolveu-se uma interface gráfica, possibilitando o envio e recebimento de dados ao barramento CAN. Os códigos utilizados são mostrados por meio de fluxogramas e algoritmos.

Palavras-chave: *Interface* gráfica. Microcontroladores. CAN. Linguagem de programação.

ABSTRACT

PINTO, Eduardo Felipe Cordeiro. **Presentation of the data traffic of a CAN bus in a graphical interface developed in Java.**. 2018. 188 p. Final Coursework (Bachelor's Degree in Electrical Engineering) – Federal University of Technology – Paraná. Ponta Grossa, 2018.

Due to the low number of references about the connection between graphic interfaces and PIC microcontrollers, connected to a CAN bus, it was decided to develop a document which exposes the main developed features at this project. For this, it was realized a bibliographic revision with relation to the data communication models used, as well as a brief introduction to the programming languages. As a result of this research, has developed a graphic interface, allowing the data sending and receiving to the CAN bus. The used codes are shown by flowcharts and algorithms.

Keywords: Graphic Interfaces. Microcontrollers. CAN. Programming Languages.

LISTA DE ALGORITMOS

Algoritmo 1 – Estrutura de decisão 56
Algoritmo 2 – Estrutura de repetição 57
Algoritmo 3 – Ponteiros 58
Algoritmo 4 – Protótipo função adc_Init 70
Algoritmo 5 – Protótipo função adc_Read 72
Algoritmo 6 – Seleção do canal analógico 72
Algoritmo 7 – Retorno da função adc_Read 73
Algoritmo 8 – Protótipo função Can_Set_Mode 75
Algoritmo 9 – Requisição do modo de operação do módulo CAN 75
Algoritmo 10 – Protótipo função Can_Init 77
Algoritmo 11 – Parametrização da temporização do módulo CAN 77
Algoritmo 12 – Protótipo da função Can_Set_Mask 78
Algoritmo 13 – Atribuição da máscara ao buffer n 79
Algoritmo 14 – Protótipo da função Can_Set_Filter 80
Algoritmo 15 – Atribuição do filtro ao filtro n 80
Algoritmo 16 – Protótipo da função Can_Set_Id 81
Algoritmo 17 – Atribuição do id ao buffer n 82
Algoritmo 18 – Protótipo da função Can_Write 83
Algoritmo 19 – Número de dados enviados 83
Algoritmo 20 – Atribuição do vetor data para o registrador de envio 84
Algoritmo 21 – Protótipo da função Can_Read 85
Algoritmo 22 – Lógica de recepção do identificador 85
Algoritmo 23 – Atribuição do registrador de recebimento para o vetor data 85
Algoritmo 24 – Número de dados recebidos 86
Algoritmo 25 – Protótipo da função toCharArray 87
Algoritmo 26 – Protótipo da função toCharArrayFixedDigits 88
Algoritmo 27 – Protótipo da função stringToInt 89
Algoritmo 28 – Protótipo da função split 91
Algoritmo 29 – Protótipo da função Send_Nibble 93
Algoritmo 30 – Protótipo da função Send_Byte 93
Algoritmo 31 – Protótipo da função Lcd_Init 94
Algoritmo 32 – Protótipo da função Lcd_Set_Cursor 95
Algoritmo 33 – Protótipo da função Lcd_Write_Char 96
Algoritmo 34 – Protótipo da função Lcd_Write_String 97
Algoritmo 35 – Protótipo da função Lcd_Cmd 98
Algoritmo 36 – Protótipo da função Usart_Init 100
Algoritmo 37 – Protótipo da função Usart_Write 100
Algoritmo 38 – Protótipo da função Usart_Read 102

LISTA DE ILUSTRAÇÕES

Figura 1 – Os cinco componentes da comunicação de dados 21
Figura 2 – Topologia de malha 23
Figura 3 – Topologia de estrela 23
Figura 4 – Topologia de barramento 24
Figura 5 – Topologia de anel 24
Figura 6 – Fluxo de dados simplex 25
Figura 7 – Fluxo de dados half-duplex 25
Figura 8 – Fluxo de dados full-duplex 25
Figura 9 – Transmissão paralela 26
Figura 10 – Transmissão serial 26
Figura 11 – Transmissão serial síncrona 27
Figura 12 – Transmissão serial assíncrona 27
Figura 13 – Barramento do protocolo de comunicação CAN 29
Figura 14 – Níveis lógicos do barramento CAN 30
Figura 15 – Quadro de dados 31
Figura 16 – Campo de arbitragem padrão 32
Figura 17 – Campo de arbitragem estendido 32
Figura 18 – Fase de arbitragem 33
Figura 19 – Campo de controle padrão 34
Figura 20 – Campo de controle estendido 34
Figura 21 – Campo CRC 35
Figura 22 – Campo de reconhecimento 35
Figura 23 – Quadro remoto 36
Figura 24 – Quadro de erro 37
Figura 25 – Quadro de sobrecarga 37
Figura 26 – Segmentos da temporização CAN 39
Figura 27 – Relação entre frequência de operação, divisor e segmentos 40
Figura 28 – CAN no modelo OSI 41
Figura 29 – Módulo convesor USART para USB 43
Figura 30 – Taxa de variação em função da resistência entre o pino RS e GND . 52
Figura 31 – Simbologia dos fluxogramas 53
Figura 32 – Processo de compilação 54
Figura 33 – Exemplo de vetor em linguagem C 58
Figura 34 – Vetor após a modificação do seu primeiro elemento 58
Figura 35 – Exemplo de união em linguagem C 59
Figura 36 – Gravador PICkit 3 60
Figura 37 – Linguagem procedural x Linguagem orientada à objetos 62
Figura 38 – Topologia utilizada 67
Figura 39 – Diagrama de ligação nó Sensor 1 e nó Sensor 2 67
Figura 40 – Diagrama de ligação nó LCD 68
Figura 41 – Diagrama de ligação nó Gerador de Mensagens 68
Figura 42 – Diagrama de ligação nó CANALL 69
Figura 43 – Formatação da conversão analógico-digital 71
Figura 44 – Fluxograma da função adc_Init 72

Figura 45 – Fluxograma da função adc_Read 73
Figura 46 – Fluxograma da função Can_Set_Mode 76
Figura 47 – Fluxograma da função Can_Init 78
Figura 48 – Fluxograma da função Can_Set_Mask 79
Figura 49 – Fluxograma da função Can_Set_Filter 81
Figura 50 – Fluxograma da função Can_Set_Id 82
Figura 51 – Fluxograma da função Can_Write 84
Figura 52 – Fluxograma da função Can_Read 86
Figura 53 – Fluxograma da função toCharArray 88
Figura 54 – Fluxograma da função toCharArrayFixedDigits 89
Figura 55 – Fluxograma da função stringToInt 90
Figura 56 – Fluxograma da função split 91
Figura 57 – Exemplo de funcionidade da função split 92
Figura 58 – Fluxograma da funçãoSend_Nibble 93
Figura 59 – Fluxograma da função Send_Byte 94
Figura 60 – Fluxograma da função Lcd_Init 95
Figura 61 – Fluxograma da função Lcd_Set_Cursor 96
Figura 62 – Fluxograma da função Lcd_Write_Char 97
Figura 63 – Fluxograma da função Lcd_Write_String 98
Figura 64 – Fluxograma da função Lcd_Cmd 99
Figura 65 – Fluxograma da função Usart_Init 100
Figura 66 – Fluxograma da função Usart_Write 101
Figura 67 – Fluxograma da função Usart_Read 102
Figura 68 – Variáveis CANALL 103
Figura 69 – Rotina de inicialização nó CANALL 106
Figura 70 – Fluxograma do laço de repetição do nó CANALL 107
Figura 71 – Rotina de inicialização nó Sensor 1 110
Figura 72 – Fluxograma do laço de repetição do nó Sensor 1 110
Figura 73 – Variáveis nó LCD 111
Figura 74 – Cabeçalho display LCD 112
Figura 75 – Rotina de inicialização nó LCD 112
Figura 76 – Display LCD completo 113
Figura 77 – Fluxograma do laço de repetição do nó LCD 114
Figura 78 – Variáveis nó Gerador de Mensagens 114
Figura 79 – Rotina de inicialização nó Gerador de Mensagens 115
Figura 80 – Fluxograma do laço de repetição do nó Gerador de Mensagens . . . 115
Figura 81 – Tela principal CANALL 116
Figura 82 – Pacotes e classes desenvolvidos 117
Figura 83 – Interface CANALL: desconectada 118
Figura 84 – Interface CANALL: conectando 118
Figura 85 – Interface CANALL: conectada 119
Figura 86 – Tabela de recebimento de mensagens 120
Figura 87 – Lógica da lista de identificadores 120
Figura 88 – Formato de envio de mensagens da interface gráfica 121
Figura 89 – Cabeçalho do arquivo salvo 122
Figura 90 – Calculadora de temporização 123
Figura 91 – Lógica da calculadora de temporização 124
Figura 92 – Calculadora de filtro 125
Figura 93 – Componentes desenvolvidos no Eagle 126

Figura 94 – Implementação física do projeto 129
Figura 95 – Ampliação nó CANALL 130
Figura 96 – Panel USB com console serial habilitado 130
Figura 97 – Tabela de recebimento de mensagens, sem filtro no formato decimal 131
Figura 98 – Tabela de recebimento de mensagens, com filtro no formato decimal 131
Figura 99 – Tabela de recebimento de mensagens, sem filtro no formato hexadecimal 132
Figura 100 – Exibição de gráfico habilitada 132
Figura 101 – Gráfico proveniente do arquivo de texto 133
Figura 102 – Envio de valores através da interface gráfica 133
Figura 103 – Exemplo de funcionamento da calculadora de temporização 134
Figura 104 – Exemplo de funcionamento da calculadora de filtro 135
Figura 105 – Esquemático nó CANALL 142
Figura 106 – Esquemático nó Sensor 143
Figura 107 – Esquemático nó LCD 144

LISTA DE TABELAS

Tabela 1 – Modelo OSI 28
Tabela 2 – Tabela verdade para filtragem de mensagens 31
Tabela 3 – Data Length Code 34
Tabela 4 – Valores possíveis de Tq 40
Tabela 5 – Taxas transmissão em função do comprimento do barramento . . . 41
Tabela 6 – Pinos FT232R 43
Tabela 7 – Funcionalidades do microcontrolador PIC18F258 44
Tabela 8 – Principais pinos do microcontrolador 45
Tabela 9 – T_{AD}, em μs, em função de F_{OSC} 46
Tabela 10 – Comparação entre ISO11898-2 e MCP2551 50
Tabela 11 – Principais pinos do transceiver 51
Tabela 12 – Modos de operação MCP2551 51
Tabela 13 – Operadores aritiméticos 55
Tabela 14 – Operadores relacionais 55
Tabela 15 – Operadores lógicos 55
Tabela 16 – Operadores em nível de bit 56
Tabela 17 – Variáveis suportadas pelo compilador XC8 61
Tabela 18 – Variáveis suportadas pela linguagem Java 61
Tabela 19 – Comparativo entre as versões disponíveis do software EAGLE . . . 65
Tabela 20 – Bits de configuração 74
Tabela 21 – Constantes do modo de operação 75
Tabela 22 – Constantes da seleção do filtro 80
Tabela 23 – Correspondência entre pinos do display e pinos do microcontrolador 92
Tabela 24 – Correspondência entre pinos do display e pinos do microcontrolador 99
Tabela 25 – Comandos da interrupção nó CANALL 108

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

ABREVIATURAS

art.	Artigo
cap.	Capítulo
sec.	Seção

SIGLAS

ACK	*Acknowlegement Field*
ADC	*Analog-Digital Converter*
CRC	*Cyclic Redundancy Check*
DLC	*Data Length Code*
EMI	*Electromagnetic interference*
ICSP	*In-Circuit Serial Programming*
ID	*Identifier*
IDE	*Integrated Development Enviroment*
JDK	*Java Development Kit*
JVM	*Java Virtual Machine*
LCD	*Liquid Crystal Display*
LLC	*Logical Link Control*
PC	*Personal Computer*
PLS	*Physical Layer Signaling*
POO	Programação Orientada a Objetos
RTR	*Remote Transmission Request*
SOF	*Start of Frame*
SMD	*Surface Mounting Device*
USB	*Universal Serial Bus*

ACRÔNIMOS

CAN	*Controller Area Network*
COM	*Communication Port*
ISO	*International Organization for Standardization*
MAC	*Medium Acess Control*
OSI	*Open Systems Interconnection*
RISC	*Reduced Instruction Set Computer*
SAE	*Society of Automotive Engineers*
USART	*Universal Serial Bus*

LISTA DE SÍMBOLOS

LETRAS LATINAS

BR_{calc}	Taxa de transmissão calculada	[bauld]
BR_{des}	Taxa de transmissão desejada	[bauld]
BRP	*Bauld Rate Prescaller*	[bit]
F_{osc}	Frequência de oscilação	[Hz]
I_{Rs}	Corrente no pino RS	[μA]
NBR	*Nominal Bit Rate*	[bit/s]
$SP(\%)$	Ponto de amoostragem	[%]
T_{AD}	Período da conversão ADC	[bit]
t_{bit}	Tempo de *bit*	[s]
T_q	Tempo de quantização	[s]
V_{dif}	Tensão diferencial	[V]
V_{CAN_L}	Tensão linha CAN_L	[V]
V_{CAN_H}	Tensão linha CAN_H	[V]
V_{Rs}	Tensão no pino RS	[V]

SUMÁRIO

1 INTRODUÇÃO . . . 18
1.1 DELIMITAÇÃO DO TEMA . . . 18
1.2 JUSTIFICATIVA . . . 18
1.3 OBJETIVOS . . . 19
1.3.1 Objetivo Geral . . . 19
1.3.2 Objetivos Específicos . . . 19
1.4 ESTRUTURA DO TRABALHO . . . 19
2 REFERENCIAL TEÓRICO . . . 21
2.1 COMUNICAÇÃO DE DADOS . . . 21
2.1.1 Topologias física . . . 22
2.1.1.1 Malha . . . 22
2.1.1.2 Estrela . . . 22
2.1.1.3 Barramento . . . 23
2.1.1.4 Anel . . . 24
2.1.2 Modos de operação . . . 25
2.1.3 Modos de transmissão . . . 25
2.1.3.1 Paralela . . . 26
2.1.3.2 Serial . . . 26
2.1.3.2.1 Síncrona . . . 26
2.1.3.2.2 Assíncrona . . . 27
2.1.4 Modelo OSI . . . 27
2.2 CONTROLLER AREA NETWORK . . . 29
2.2.1 Características . . . 29
2.2.2 Quadros . . . 31
2.2.2.1 Quadro de dados . . . 31
2.2.2.1.1 Início do quadro . . . 31
2.2.2.1.2 Campo de arbitragem . . . 32
2.2.2.1.3 Campo de controle . . . 33
2.2.2.1.4 Campo de dados . . . 34
2.2.2.1.5 Campo da verificação cíclica de redundância . . . 35
2.2.2.1.6 Campo de reconhecimento . . . 35
2.2.2.1.7 Fim do quadro . . . 36
2.2.2.2 Quadro remoto . . . 36
2.2.2.3 Quadro de erro . . . 36
2.2.2.3.1 Flag de erro . . . 37
2.2.2.3.2 Delimitador de erro . . . 37
2.2.2.4 Quadro de sobrecarga . . . 37
2.2.2.5 Espaço entre quadros . . . 38
2.2.3 Temporização . . . 38
2.2.4 Modelo OSI do protocolo CAN . . . 40
2.3 ADDRESSABLE UNIVERSAL SYNCHRONOUS ASYNCHRONOUS RECEIVER TRANSMITTER . . . 42
2.3.1 Conversor USART para USB . . . 42
2.4 MICROCONTROLADOR PIC18F258 . . . 44

2.4.1 Módulo ADC 44
2.4.2 Módulo CAN 46
2.4.3 Módulo USART 47
2.5 TRANSCEIVER MCP2551 50
2.6 LINGUAGEM DE PROGRAMAÇÃO 53
2.6.1 Linguagem C 53
2.6.1.1 Elementos básicos da linguagem C 54
2.6.1.2 MPLAB X IDE 59
2.6.1.3 Gravador PICkit 3 59
2.6.1.4 Compilador XC8 60
2.6.2 Linguagem Java 61
2.6.2.1 Programação orientada a objetos 62
2.6.2.2 Kit de desenvolvimento Java (JDK) 63
2.6.2.3 NetBeans 64
2.7 EAGLE 65
3 DESENVOLVIMENTO 66
3.1 DESCRIÇÃO DO PROJETO 66
3.1.1 Nó Sensor 1 e nó Sensor 2 67
3.1.2 Nó LCD 68
3.1.3 Nó Gerador de Mensagens 68
3.1.4 Nó CANALL 69
3.2 PROGRAMAÇÃO PIC18F258 70
3.2.1 Bibliotecas 70
3.2.1.1 Biblioteca adc.h 70
3.2.1.1.1 Função adc_Init 70
3.2.1.1.2 Função adc_Read 72
3.2.1.2 Biblioteca config.h 73
3.2.1.3 Biblioteca CAN.h 74
3.2.1.3.1 Função Can_Set_Mode 74
3.2.1.3.2 Função Can_Init 76
3.2.1.3.3 Função Can_Set_Mask 78
3.2.1.3.4 Função Can_Set_Filter 80
3.2.1.3.5 Função Can_Set_Id 81
3.2.1.3.6 Função Can_Write 82
3.2.1.3.7 Função Can_Read 84
3.2.1.4 Biblioteca conversions.h 86
3.2.1.4.1 Função toCharArray 86
3.2.1.4.2 Função toCharArrayFixedDigits 87
3.2.1.4.3 Função strintToInt 89
3.2.1.4.4 Função split 90
3.2.1.5 Biblioteca lcd.h 92
3.2.1.5.1 Função Send_Nibble 92
3.2.1.5.2 Função Send_Byte 93
3.2.1.5.3 Função Lcd_Init 94
3.2.1.5.4 Função Lcd_Set_Cursor 95
3.2.1.5.5 Função Lcd_Write_Char 96
3.2.1.5.6 Função Lcd_Write_String 97
3.2.1.5.7 Função Lcd_Cmd 98
3.2.1.6 Biblioteca USART.h 99

3.2.1.6.1 Função Usart_Init 99
3.2.1.6.2 Função Usart_Write 100
3.2.1.6.3 Função Usart_Read 101
3.2.2 Programação nó CANALL 102
3.2.2.1 Função Principal nó CANALL 103
3.2.2.1.1 Laço de repetição nó CANALL 105
3.2.2.2 Interrupção nó CANALL 108
3.2.3 Programação nó Sensor 1 108
3.2.3.1 Função principal nó Sensor 1 109
3.2.3.1.1 Laço de repetição nó Sensor 1 109
3.2.4 Programação nó Sensor 2 111
3.2.5 Programação nó LCD 111
3.2.5.1 Função principal nó LCD 111
3.2.5.1.1 Laço de repetição nó LCD 112
3.2.6 Programação nó Gerador de Mensagens 113
3.2.6.1 Função principal nó Gerador de Mensagens 113
3.2.6.1.1 Laço de repetição nó Gerador de Mensagens 114
3.3 INTERFACE GRÁFICA 116
3.3.1 Painel USB 117
3.3.2 Painel CAN 119
3.3.3 Painel Dados 121
3.3.4 Menu Ferramentas 122
3.3.4.1 Submenu Temporização 122
3.3.4.2 Submenu Filtro 124
3.4 PLACAS DE CIRCUITO IMPRESSO 126
3.4.1 Itens comuns as placas de circuito impresso 127
3.4.2 Placa para o nó Sensor 1 e nó Sensor 2 128
3.4.3 Placa para o nó LCD 128
3.4.4 Placa para o nó CANALL 128
3.4.5 Placa para o nó Gerador de Mensagens 128
4 RESULTADOS E DISCUSSÕES 129
5 CONCLUSÃO E PERSPECTIVAS 136
REFERÊNCIAS 137
GLOSSÁRIO 141
APÊNDICES 141
APÊNDICE A – ESQUEMÁTICO DOS NÓS 142
APÊNDICE B – BIBLIOTECAS DESENVOLVIDAS 145
B.0.1 adc.h 145
B.0.2 can.h 146
B.0.3 config.h 152
B.0.4 conversions.h 153
B.0.5 lcd.h 155
B.0.6 USART.h 157
APÊNDICE C – CÓDIGOS FONTE DOS NÓS 159
C.0.1 CANALL.c 159
C.0.2 NO_GERADOR.c 163
C.0.3 NO_LCD.c 164
C.0.4 NO_SENSOR1.c 165
C.0.5 NO_SENSOR2.c 166

ANEXOS 167
ANEXO A – REGISTRADORES PIC18F258 . . . . **168**
A.1 INTCON 168
A.2 PIR1 169
A.3 PIE1 170
A.4 TXSTA 171
A.5 RCSTA 172
A.6 CANCON 173
A.7 CANSTAT 174
A.8 TXBNCON 175
A.9 TXBNSIDH 176
A.10 TXBNSIDL 176
A.11 TXBNDM 177
A.12 TXBNDLC 177
A.13 RXB0CON 178
A.14 RXB1CON 179
A.15 RXBNSIDH 180
A.16 RXBNSIDL 180
A.17 RXBNDLC 181
A.18 RXBNDM 182
A.19 RXFNSIDH 182
A.20 RXFNSIDL 183
A.21 RXMNSIDH 183
A.22 RXMNSIDL 184
A.23 BRGCON1 184
A.24 BRGCON2 185
A.25 BRGCON3 186
A.26 CIOCON 186
A.27 ADCON0 187
A.28 ADCON1 188

1 INTRODUÇÃO

Após o desenvolvimento do primeiro microprocessador, o Intel 4004, no ano de 1971, os sistemas computacionais se tornaram cada vez mais itens indispensáveis nos diversos setores industriais.

Processos que eram apenas monitorados através de painéis indicadores visuais, como lâmpadas e medidores, passaram a ser não só monitorados, mas também controlados pelos sistemas computacionais. Isto só pode ser obtido com o desenvolvimento dos protocolos de comunicação e dos sistemas supervisórios.

O estudo de um protocolo de comunicação largamente utilizado na área automobilística e industrial como o CAN (*Controller Area Network*) é extremamente relevante na formação de um engenheiro eletricista.

Relacionar diferentes áreas da tecnologia, como o desenvolvimento de interfaces gráficas, conhecimento de redes industriais e programação de microcontroladores permite que o profissional em formação adquira habilidades de integração de diversos sistemas.

1.1 DELIMITAÇÃO DO TEMA

Este trabalho consiste no desenvolvimento de uma rede CAN utilizando microcontroladores de fácil acesso no mercado nacional. Este desenvolvimento será fortemente embasado em normas (ISO 11898-1 e ISO 11898-2) e folhas de dados de fabricantes.

Será também desenvolvida uma *interface* gráfica, cuja finalidade é apresentar o tráfego de dados da rede CAN.

1.2 JUSTIFICATIVA

Não foi encontrado nenhum trabalho que comtemple as características deste projeto: desenvolvimento de um barramento CAN, utilizando microcontroladores PIC em uma *interface* gráfica desenvolvida em Java.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

O objetivo geral deste trabalho de conclusão de curso consiste na apresentação de dados provenientes de um barramento CAN em uma *interface* gráfica.

1.3.2 Objetivos Específicos

- Desenvolver bibliotecas em linguagem C relacionadas com a comunicação CAN e USART, conversão entre formatos numéricos e textuais e apresentação de dados em um *display* LCD.
- Desenvolver uma *interface* gráfica, utilizando a linguagem Java, que tenha a capacidade de receber e enviar dados de um computador pessoal para o barramento CAN.
- Elaborar fluxogramas e diagramas que representem visualmente a lógica de programação utilizada.
- Desenvolver calculadoras que auxiliem na temporização do protocolo de comunicação CAN.
- Projetar e confeccionar as placas de circuito impresso utilizadas neste projeto.
- Salvar dados do barramento CAN e mostra-los graficamente.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho apresenta cinco capítulos: Introdução, Referencial Teórico, Desenvolvimento, Resultados e Discussões e Conclusões e Perspectivas.

O segundo capítulo, Referencial Teórico, aborda os princípios de comunicações de dados, um embasamento teórico sobre o protocolo CAN e modelo USART, expõe as principais características do microcontrolador e *tranceiver* e aborda conceitos de linguagem de programação.

O terceiro capítulo, Desenvolvimento, descreve as características do projeto como um todo, das bibliotecas desenvolvidas e da interface gráfica. É também realizada uma abordagem sobre o *hardware* deste projeto.

O quarto capítulo, Resultados e Discussões, evidencia o funcionamento do sistema desenvolvido, o expondo por meio de fotografias e figuras as suas características.

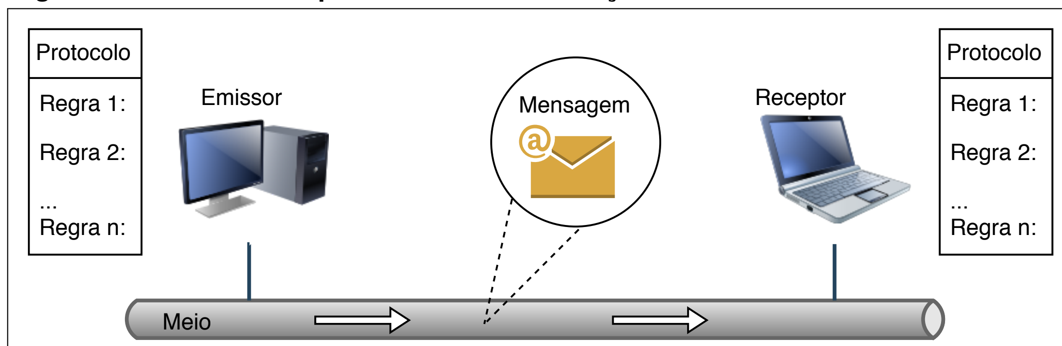
O quinto capítulo, Conclusões e Perspectivas, apresenta as conclusões deste trabalho assim como sugestões para trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 COMUNICAÇÃO DE DADOS

Comunicação de dados são as trocas de dados entre dois dispositivos por intermédio de algum tipo de meio de transmissão, como um condutor formado por fios (FOROUZAN, 2009). A comunicação de dados é formada basicamente por cinco componentes: mensagem, emissor, receptor, meio e protocolo (figura 1).

Figura 1 – Os cinco componentes da comunicação de dados



Fonte: Adaptado de Forousan

As mensagens podem conter textos, números, figuras, áudios e vídeos. Para generalizar o seu conteúdo, as mensagens podem ser chamadas simplesmente de dados. São os dados os responsáveis pela troca de informação entre os emissores e receptores.

Os dispositivos emissores são aqueles que enviam os dados, enquanto que os receptores são os que recebem os dados. Esses dispositivos são comumente chamados de nós ou estações.

Para que a troca de informação entre os emissores e receptores ocorra, é necessário o meio de transmissão. Ele é o caminho físico pelo qual a mensagem trafega. Os meios de transmissão podem ser guiados (cabos) e não guiados (sem fio).

A conexão entre um emissor e um receptor através do meio físico não garante a troca de informação. Esta somente é garantida se emissor e receptor seguirem um conjunto de regras em comum. Este conjunto de regras é denominado protocolo. O protocolo controla a comunicação de dados, representando um acordo entre os dispositivos de comunicação.

Existem vários protocolos relacionados com a comunicação de dados, sendo que cada um deles procura atender critérios como desempenho, confiabilidade e segurança. É por causa disto que não existe somente um tipo de rede, pois a sua aplicabilidade é quem define tais critérios. Uma comunicação entre um teclado e um computador pessoal não precisa ser tão segura quanto a comunicação entre um computador industrial e um braço robótico.

Como exemplo de protocolos é possível citar alguns aplicados à automação industrial (*DeviceNet*, *Modbus*, *Profibus*, HART e *EtherCAT*) e os utilizados em aplicações automobilísticas (CAN, *FlexRay* e LIN).

2.1.1 Topologias física

A topologia física é um termo que refere ao modo que uma rede é disposta fisicamente. As topologias podem ser do tipo malha, estrela, barramento e anel. Existe também a combinação destes tipos de topologias, que são denominadas redes mistas.

Quando apenas dois dispositivos estão conectados entre si, a conexão é dita ponto a ponto. Já quando mais de dois dispositivos estão conectados entre si, a conexão é chamada de multiponto.

2.1.1.1 Malha

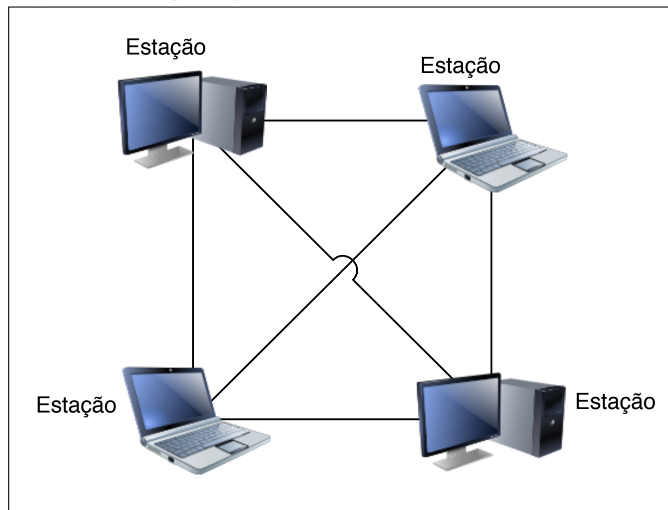
A topologia malha apresenta uma conexão ponto a ponto entre todos os dispositivos. É uma topologia muito robusta, pois a falha em um *link* não compromete toda a rede. A topologia malha é apresentada na figura 2.

2.1.1.2 Estrela

Em uma topologia estrela, todo dispositivo é conectado à um controlador central. É uma topologia mestre-escravo, onde o controlador central é o mestre e os demais, escravos. A comunicação entre os escravos só pode ser realizada com a passagem pelo mestre. O controlador central também pode ser chamado de *hub*.

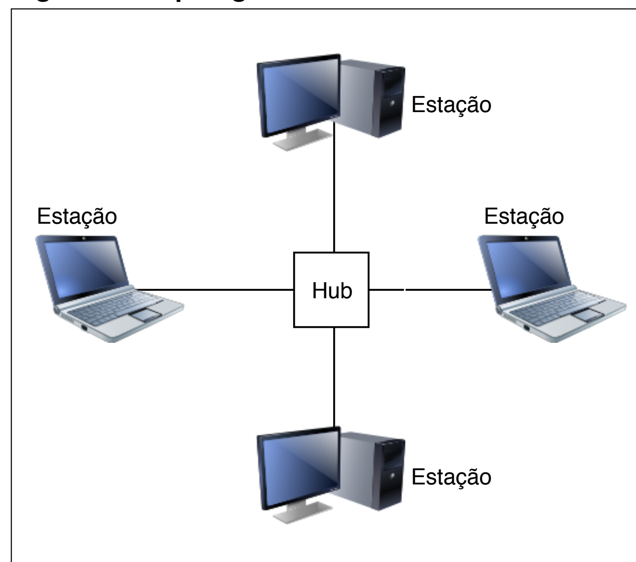
Esta topologia é mais barata que a de malha, pois utiliza menos cabos. Possui o inconveniente de que se o controlador centrar falhar, toda a rede para de funcionar. Tal topologia é mostrada na figura 3.

Figura 2 – Topologia de malha



Fonte: Adaptado de Forousan

Figura 3 – Topologia de estrela



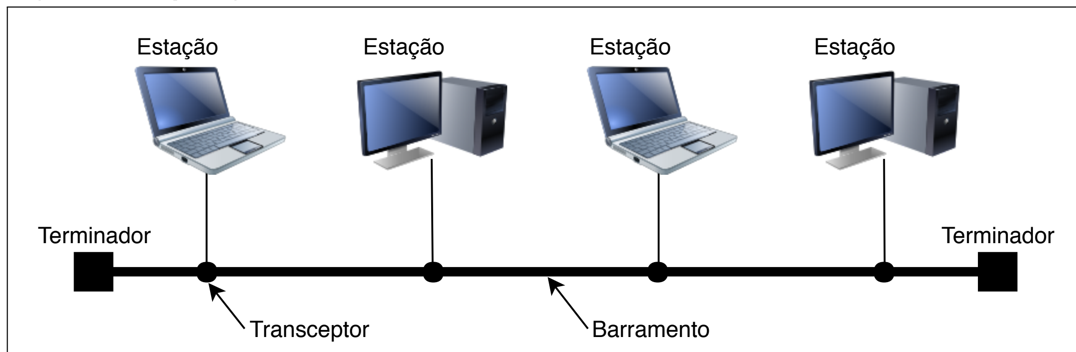
Fonte: Adaptado de Forousan

2.1.1.3 Barramento

A topologia barramento apresenta uma conexão multiponto entre os dispositivos. Um longo cabo, denominado *backbone* interliga todos os dispositivos da rede. Nas extremidades do backbone é necessário um terminador, que tem como função impedir a reflexão de sinal (figura 4).

Possui facilidade de instalação e menor quantidade de cabos, quando comparado as demais topologias. Como inconveniente, uma ruptura ou falha do *backbone* interrompe toda a rede.

Figura 4 – Topologia de barramento

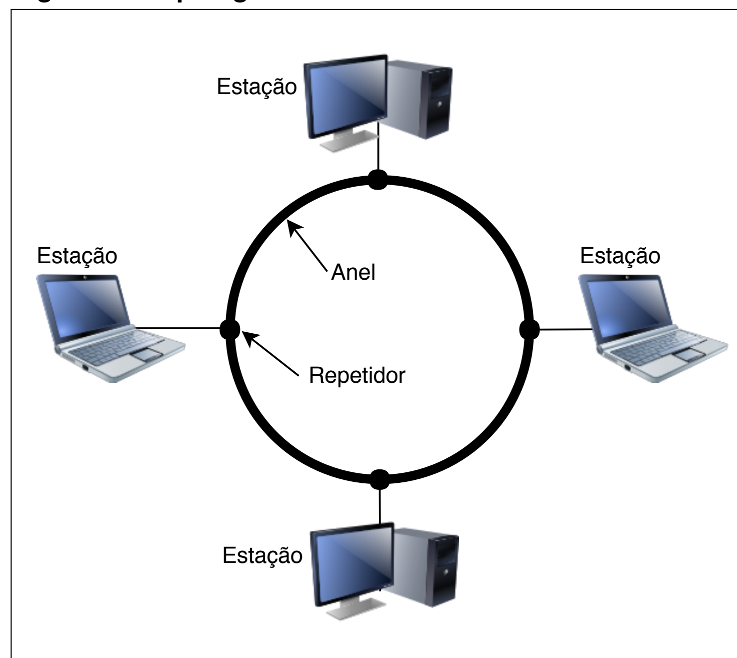


Fonte: Adaptado de Forousan

2.1.1.4 Anel

Na topologia anel, todos os dispositivos possuem conexão ponto a ponto com os dispositivos conectados de cada lado. Uma mensagem enviada por uma estação passa por outras estações até o destino final por meio de retransmissões. A topologia em anel é mostrada na figura 5.

Figura 5 – Topologia de anel



Fonte: Adaptado de Forousan

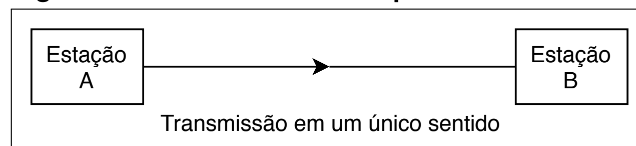
Esta topologia é facilmente instalada e reconfigurada, porém há maior custo envolvido com o cabeamento.

2.1.2 Modos de operação

A comunicação de dados entre dois ou mais dispositivos pode ser *simplex*, *half-duplex* ou *full-duplex*. Esta classificação define se a transmissão e recepção ocorre simultaneamente ou não.

Uma rede *simplex* permite a comunicação unidirecional, ou seja, apenas um dispositivo transmite enquanto apenas um dispositivo recebe (figura 6). Isto é evidenciando na comunicação entre um teclado e um computador.

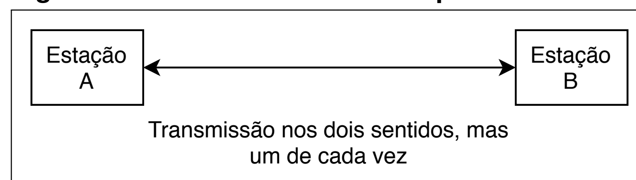
Figura 6 – Fluxo de dados simplex



Fonte: Autoria própria

Uma rede half-duplex permite a comunicação bidirecional, mas não ao mesmo tempo, como o que o ocorre nos *walkie-talkies* (figura 7).

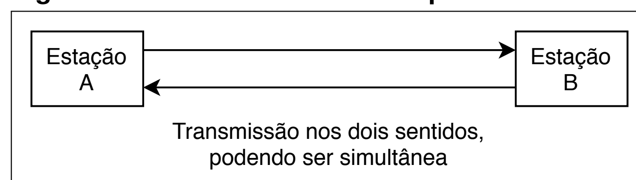
Figura 7 – Fluxo de dados half-duplex



Fonte: Autoria própria

Uma rede *full-duplex* permite a comunicação bidirecional e ao mesmo tempo (figura 8). A rede telefônica é um exemplo de rede *full-duplex*.

Figura 8 – Fluxo de dados full-duplex



Fonte: Autoria própria

2.1.3 Modos de transmissão

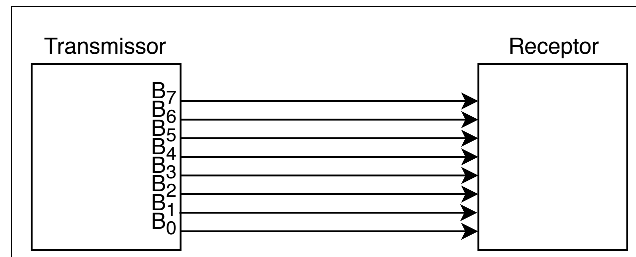
O modo de transmissão designa o número de unidades elementares de informação (*bit*) que podem ser transmitidas simultaneamente pelo canal de comunicação.

Os modos de transmissão podem ser: transmissão paralela e transmissão série.

2.1.3.1 Paralela

Na transmissão paralela ocorre o envio simultâneo de mais de um *bit*. Para x *bits* enviados simultaneamente, são necessários x meios de transmissão (figura 9).

Figura 9 – Transmissão paralela



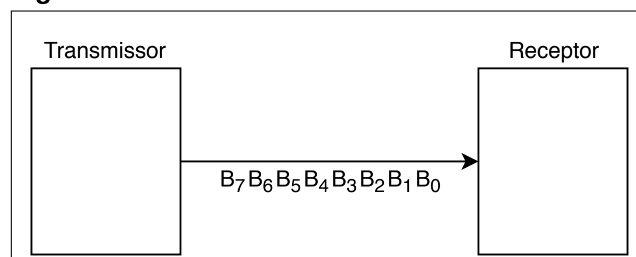
Fonte: Adaptado de Forousan

A vantagem de uma transmissão paralela é a sua velocidade de transmissão, mas requer mais linhas de comunicação.

2.1.3.2 Serial

Na transmissão serial, os dados são enviados *bit* por *bit* pelo meio de transmissão (figura 10). As transmissões seriais se dividem em síncronas e assíncronas.

Figura 10 – Transmissão serial



Fonte: Adaptado de Forousan

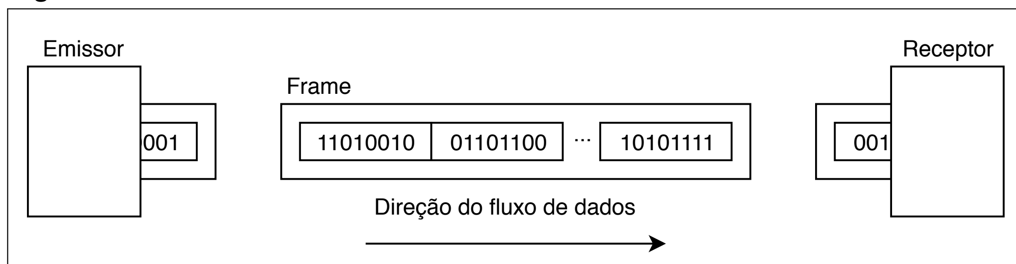
Possui a vantagem de utilizar apenas um meio de comunicação, mas sua velocidade de transmissão é menor.

2.1.3.2.1 Síncrona

Na transmissão serial síncrona, emissor e receptor devem estar sincronizados no mesmo *clock*. O receptor recebe, de forma contínua, as informações no ritmo em

que o emissor envia. Para isto é necessária uma linha denominada *clock*.

Figura 11 – Transmissão serial síncrona



Fonte: Adaptado de Forousan

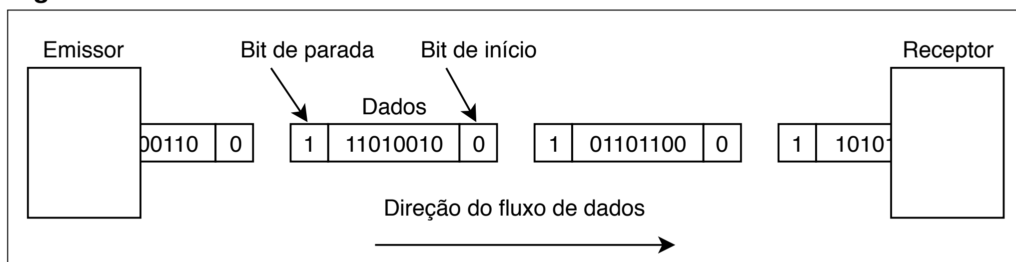
2.1.3.2.2 Assíncrona

Na transmissão assíncrona o intervalo entre as mensagens não é importante. Sem sincronização o receptor não é capaz de prever quando o próximo dado chegará.

Para alertar sobre a chegada de dados, existe um *bit* extra, anterior aos dados denominado *bit* de início (*start bit*). Geralmente possui nível lógico baixo.

Quando todo o dado é transmitido, é necessário alertar ao receptor que não há mais dados. Isto é feito com anexando *bits* extras ao final dos dados. Tais *bits* tem o nome de *bits* de parada (*stop bits*).

Figura 12 – Transmissão serial assíncrona



Fonte: Adaptado de Forousan

2.1.4 Modelo OSI

O modelo de referência da ISO (International Organization for Standardization), tem como principal objetivo ser um modelo padrão para protocolos de comunicação entre diversos tipos de sistema. O modelo OSI (Open Systems Interconnection) foi publicado em 1984 pela ISO (ZIMMERMANN, 1980).

Trata-se de uma arquitetura modelo que divide as redes em sete camadas hierárquicas, sendo que cada uma delas realiza determinadas funções. As camadas são: física, enlace de dados, rede, transporte, sessão, apresentação e aplicação. As camadas bem como suas funções são mostradas na tabela 1.

Tabela 1 – Modelo OSI

Camada		Função
Número	**Nome**	
7	Aplicação	Funções especializadas, como terminais virtuais
6	Apresentação	Formatação de dados e conversão de caracteres e códigos
5	Sessão	Negociação e estabelecimento de conexão com outro nó
4	Transporte	Meios e métodos para a entrega de dados ponta-a-ponta
3	Rede	Roteamento de pacotes através de uma ou várias redes
2	Enlace	Detecção e correçao dos erros proveninente do meio de transmissão
1	Física	Transmissão dos bits através do meio de transmissão

Fonte: Autoria própria

2.2 CONTROLLER AREA NETWORK

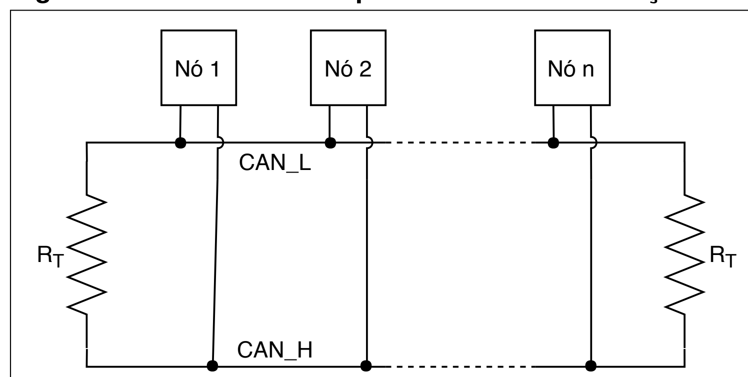
Em fevereiro de 1986, a Robert Bosch GmbH introduziu o Controller Area Network (CAN) no congresso da Sociedade de Engenheiros Automotivos (SAE). Era a hora do nascimento de um dos protocolos de comunicação mais bem-sucedidos de todos os tempos (CAN IN AUTOMATION, 2015).

O protocolo CAN foi originalmente idealizado em aplicações automobilísticas, mas o seu uso na automação industrial ganhou destaque desde a sua concepção, pois além da rede CAN de ser muito robusta, possui eficiente detecção de erros.

2.2.1 Características

O CAN baseia-se na topologia de barramento, onde apenas dois fios são necessários para a comunicação de dados ocorrer (figura 13). Este barramento é multimestre, onde qualquer dispositivo que esteja conectado a ele é capaz de transmitir e receber dados. O tráfego destes dados é feito de maneira serial, onde cada dispositivo conectado ao barramento é denominado nó.

Figura 13 – Barramento do protocolo de comunicação CAN



Fonte: Autoria própria.

Assim como mostrado na figura 4, a figura 13 possui terminadores para evitar a reflexão de sinal. O número máximo de estações (ou nós) conectados ao barramento pe limitado pela capacidade do transceptor (INTERNATIONAL STANDARDIZATION ORGANIZATION, 2003a).

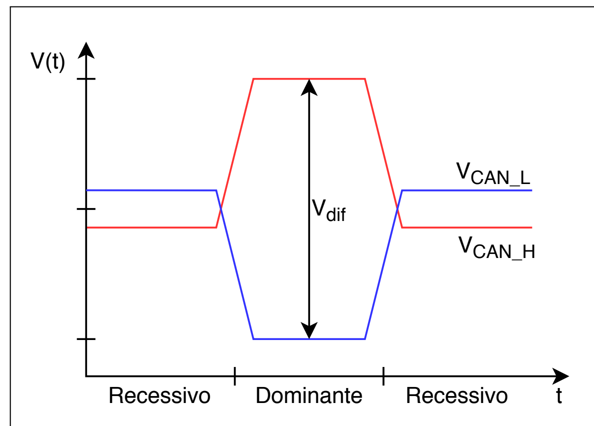
Os dados são transmitidos com nível de tensão diferencial entre os dois fios do barramento, denominados CANH (CAN *High*) e CANL (CAN *Low*). A tensão diferencial

é dada pela expressão 1.

$$V_{dif} = V_{CAN_H} - V_{CAN_L} \tag{1}$$

No CAN os dados são representados por *bits* dominantes e *bits* recessivos. O estado dominante é relacionado com o nível lógico 0, enquanto o recessivo com o nível lógico 1. O gráfico presente na figura 14 mostra a tensão diferencial entre as linhas CANH e CANL do barramento CAN, bem como o nível lógico relacionado à esta diferença de tensão. Segundo a ISO 11898-2, quando a tensão diferencial é maior que $0{,}9V$, o estado é dominante. Para uma tensão diferencial menor $0{,}5V$ o estado é recessivo (INTERNATIONAL STANDARDIZATION ORGANIZATION, 2003b).

Figura 14 – Níveis lógicos do barramento CAN



Fonte: Adaptado de ISO 11898-2.

Apenas um dispositivo pode enviar dados, enquanto todos os outros, recebem. Quando mais de um dispositivo tenta enviar dados ao mesmo tempo, o dispositivo com maior prioridade terá a preferência na transmissão. Esta prioridade é definida pelo identificador (ID). Quanto menor for o valor do identificador, maior será a prioridade da mensagem 18.

Como definido em Bosch et al. (1991), é possível receber mensagens com valor de identificador pré-definidos pelo programador do controlador CAN. A tabela 2 mostra a lógica pertinente à máscara, filtro e identificador. Isto também é observado mais adiante, na equação 16.

Tabela 2 – Tabela verdade para filtragem de mensagens

Máscara	Filtro	Identificador	Aceita ou rejeita o bit
0	x	x	Aceita
1	0	0	Aceita
1	0	1	Rejeita
1	1	0	Rejeita
1	1	1	Aceita

Fonte: Datasheet PIC18F258.

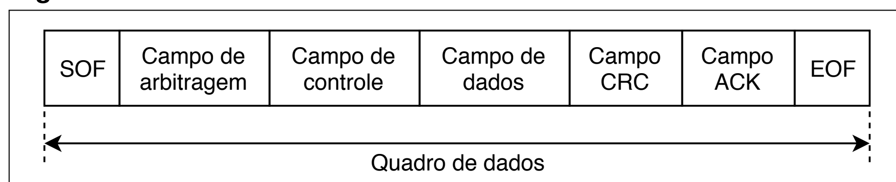
2.2.2 Quadros

A comunicação entre os nós do barramento CAN é realizada através de frames, também denominados quadros. Existem basicamente quatro tipos diferentes de quadros: quadro de dados, quadro remoto, quadro de erro e quadro de sobrecarga. Os dois primeiros permitem programação enquanto os demais não permitem.

2.2.2.1 Quadro de dados

Um quadro de dados é composto de sete diferentes campos de *bit*, sendo eles: início do quadro (SOF), campo de arbitragem, campo de controle, campo de dados, campo da verificação cíclica de redundância (CRC), campo de reconhecimento (ACK) e fim do quadro (EOF). Isto pode ser observado na figura 15.

Figura 15 – Quadro de dados



Fonte: Adaptado de Bosch.

2.2.2.1.1 Início do quadro

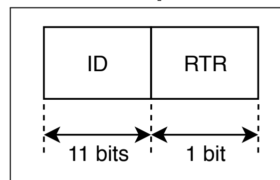
O início do quadro (SOF – *Start of Frame*) indica o início do quadro de dados e do quadro remoto. É composto por um único *bit* dominante. Trata-se do primeiro *bit* transmitido em uma comunicação CAN.

2.2.2.1.2 *Campo de arbitragem*

O campo de arbitragem (Arbitration Field) é necessário para resolver colisões de mensagem no barramento. Ele apresenta diferenças no formato padrão (standard) e estendido (extended).

É composto pelos 11 *bits* do identificador (ID) mais 1 *bit* da requisição de transmissão remota (RTR – Remote Transmission Request) no formato padrão (figura 16). Para o quadro de dados, o *bit* RTR deve ser dominante.

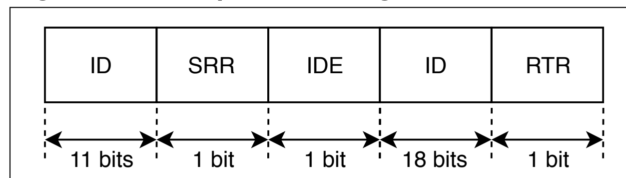
Figura 16 – Campo de arbitragem padrão



Fonte: Adaptado de Bosch.

No formato estendido, o campo de arbitragem possui 29 *bits* do identificador, separando os 11 mais significativos pelos *bits* SSR e IDE. Há também o *bit* RTR no fim do campo de arbitragem, totalizando 32 *bits* (figura 17).

Figura 17 – Campo de arbitragem estendido



Fonte: Adaptado de Bosch.

O *bit* SSR tem a mesma função do *bit* RTR, ou seja, a requisição de transmissão remota. O *bit* IDE define se será utilizado o formato padrão (dominante) ou estendido (recessivo).

Quando mais de um dispositivo tenta enviar mensagem, simultaneamente, o campo de arbitragem define qual será o único dispositivo (nó) que irá transmitir seus dados. Essa definição é feita durante a fase de arbitragem.

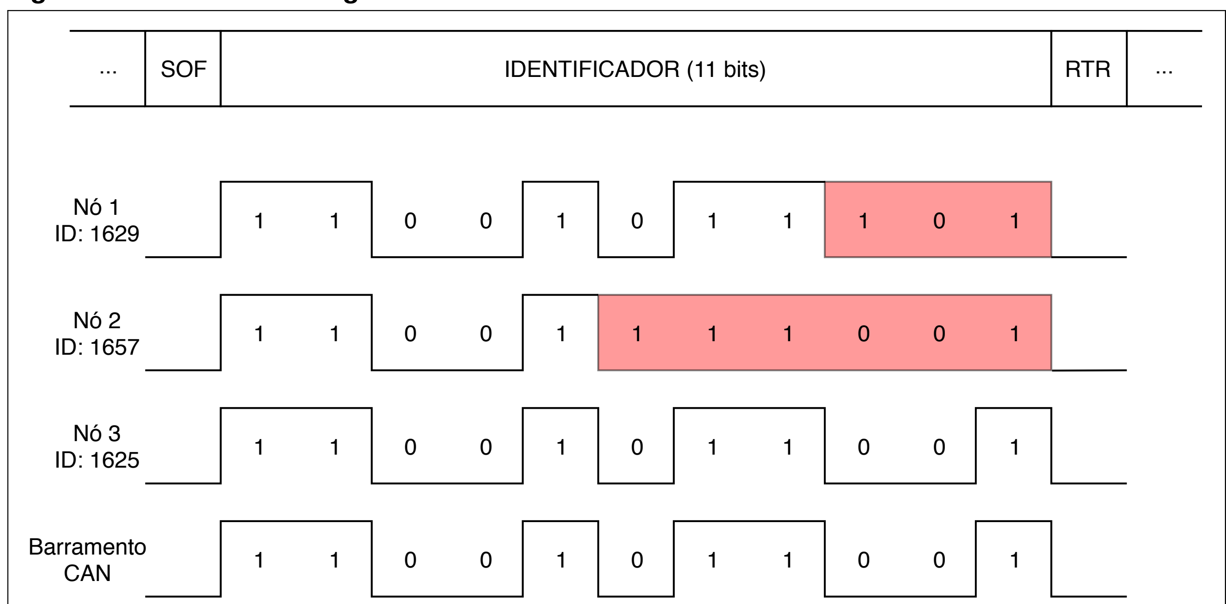
O Identificador (ID) define a prioridade que o nó possui, e quanto menor for o seu valor, maior é a sua prioridade. No formato padrão o Identificador é composto por 11 *bits* enquanto no formato estendido, 29 *bits*. Em ambos os formatos o *bit* mais

significativo é transmitido primeiro. É necessário que os sete *bits* mais significativos não sejam todos recessivos (BOSCH et al., 1991).

Durante a fase de arbitragem, os dispositivos transmissores transmitem seus identificadores e comparam seu estado (recessivo ou dominante) no barramento. Esta comparação é feita *bit* a *bit*. Quando os estados são iguais, os dispositivos continuam a transmissão. Quando diferentes, o dispositivo com estado dominante continuará transmitindo, enquanto o com estado recessivo cessa a transmissão. No final da fase de arbitragem apenas um dispositivo estará transmitindo.

A figura 18 mostra três dispositivos, tentando transmitir dados. É possível observar que o nó com menor valor de Identificador (Nó 3) ganhará a preferência de transmissão. Como falado anteriormente, o nível dominante é relacionado com nível lógico 0, enquanto o recessivo é relacionando com o nível lógico 1. Isto justifica um ID com valor menor ter maior prioridade que um ID maior, visto que o primeiro possui maior número de *bits* dominantes (0), quando comparado com o segundo.

Figura 18 – Fase de arbitragem



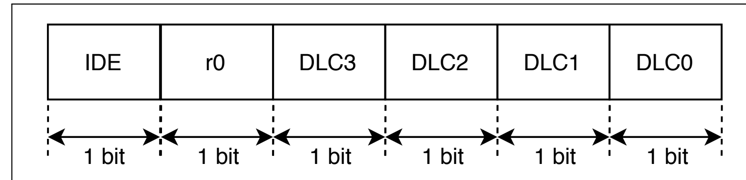
Fonte: Adaptado de Bosch.

2.2.2.1.3 Campo de controle

O campo de controle (*Control Field*) é composto de seis *bits*, que incluem quatro necessários para definir o número de *bytes* da mensagem (DLC – *Data Lenght Code*). Os dois *bits* restantes são reservados no formato estendido (figura 20). No

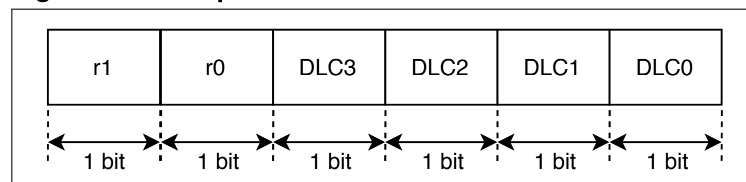
modo padrão um *bit* é reservado enquanto o outro é o bit IDE (figura 19), presente no campo de arbitragem no formato estendido (figura 17).

Figura 19 – Campo de controle padrão



Fonte: Adaptado de Bosch.

Figura 20 – Campo de controle estendido



Fonte: Adaptado de Bosch.

A definição do número de *bytes* presentes no campo de dados é feita com base na tabela 3.

Tabela 3 – Data Length Code

Número de bytes da mensagem	DLC3	DLC2	DLC1	DLC0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0

Fonte: ISO 11898-2.

2.2.2.1.4 Campo de dados

O campo de dados (*Data Field*) é o responsável pela troca de dados numa comunicação CAN. Possui tamanho variável, de 0 a 8 *bytes*, sendo o *bit* mais significativo transmitido primeiro.

2.2.2.1.5 *Campo da verificação cíclica de redundância*

O campo da verificação cíclica de redundância (CRC – *Cyclic Redundancy Check*) é utilizado para verificar se há erro de transmissão no quadro recebido. É composto pela sequência CRC procedida pelo delimitador CRC.

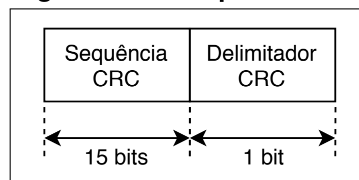
A sequência CRC armazena o resto da divisão entre os campos anteriores e o polinômio gerador, mostrado na equação 2.

$$x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + 1 \quad (2)$$

O dispositivo receptor compara a sequência CRC recebida com a calculada por ele, com base com campos e polinômio gerador. Caso eles difiram, há erro de transmissão.

A sequência CRC é composta por 15 *bits* enquanto o delimitador CRC é composto por um único *bit* recessivo, como mostrado na figura 21.

Figura 21 – Campo CRC

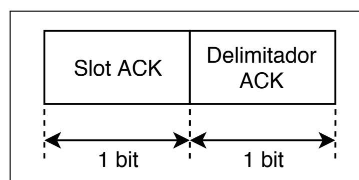


Fonte: Adaptado de Bosch.

2.2.2.1.6 *Campo de reconhecimento*

O campo de reconhecimento (ACK – *Acknowledgement Field*) indica se o quadro foi recebido normalmente, este campo possui dois *bits*, um para o *slot* ACK e outro delimitador ACK (figura 22).

Figura 22 – Campo de reconhecimento



Fonte: Adaptado de Bosch.

Quando a mensagem é transmitida, o *bit slot* ACK é posto como recessivo, para que quando a mensagem é recebida e nenhum erro é verificado através do CRC, o *bit slot* ACK passa a ser dominante.

O *bit* delimitador é sempre recessivo. Como consequência, o *bit slot* ACK está entre dois *bits* recessivos (delimitador CRC e delimitador ACK).

2.2.2.1.7 Fim do quadro

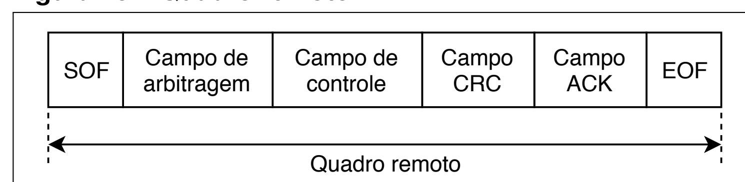
O fim do quadro (EOF – *End of Frame*) marca o final dos quadros de dados e remoto. Esta marca de finalização é feita por meio de 7 *bits* recessivos. Estes *bits* são os últimos *bits* a serem transmitidos.

2.2.2.2 Quadro remoto

O quadro remoto possui os mesmos campos que o quadro de dados, exceto a ausência do campo de dados.

É utilizado quando há a necessidade de um nó receptor solicitar o envio de dados de um nó transmissor (IBRAHIM, 2014). Este tipo de *frame* é raramente utilizado em aplicações automobilísticas. A estrutura de um quadro remoto é mostrada na figura 23.

Figura 23 – Quadro remoto

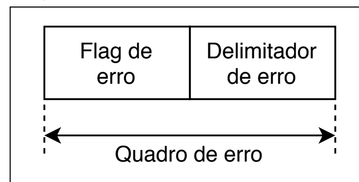


Fonte: Adaptado de Bosch.

2.2.2.3 Quadro de erro

O quadro de erro é necessário para notificar se um erro foi detectado durante a transmissão. Este quadro é composto de dois campos de *bit*: *flag* de erro e delimitador de erro (figura 24). Os quadros de erro são gerados e transmitidos automaticamente pelo hardware do protocolo CAN quando um erro é detectado no barramento.

Figura 24 – Quadro de erro



Fonte: Adaptado de Bosch.

2.2.2.3.1 Flag de erro

O campo de *bit flag* de erro é composto de no mínimo 6 *bits* e no máximo 12 *bits*. Ele pode indicar dois tipos diferentes de erros: erro ativo e erro passivo. Um erro ativo consiste de seis *bits* consecutivos em estado dominante, enquanto um erro passivo consiste de seis *bits* consecutivos em estado recessivo.

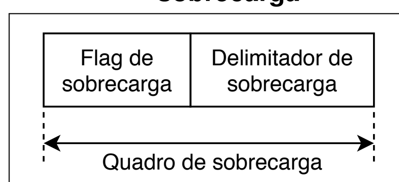
2.2.2.3.2 Delimitador de erro

O campo delimitador de erro complementa o quadro de erro. Após o término do quadro de erro, a atividade no barramento volta ao normal. O nó interrompido pelo erro tenta retransmitir a mensagem abortada. Este campo de *bit* é composto por 8 *bits* recessivos.

2.2.2.4 Quadro de sobrecarga

O quadro de sobrecarga é utilizado somente pelos nós receptores. Ele é necessário para informar aos nós que transmitem dados que o nó receptor ainda não está apto à recebe-los. É composto de dois campos: *flag* de sobrecarga e delimitador de sobrecarga (figura 25).

Figura 25 – Quadro de sobrecarga



Fonte: Adaptado de Bosch.

2.2.2.5 Espaço entre quadros

A separação entre um quadro predecessor, de qualquer tipo, ao quadro de dados ou quadro remoto é feita pelo espaço entre quadros. É composto por pelo menos três *bits* recessivos, denominados de Intermissão. Este tempo é necessário para permitir aos nós um tempo a mais de processamento antes do próximo quadro.

2.2.3 Temporização

Para haver comunicação de dados entre os nós de um barramento CAN é necessário que todos estejam com mesma taxa de transmissão, ou seja, mesmo Nominal Bit Rate (NBR). Esta grandeza mede o número de *bits* transmitidos por segundo, sem ressincronização. O inverso desta grandeza é o tempo nominal de *bit* (Nominal Bit Time), representado pela sigla t_{bit}.

A temporização do barramento CAN é composta de quatro segmentos não sobrepostos, que, quando somados, formam o tempo de *bit* nominal (t_{bit}). Há uma relação entre o tempo de *bit* e a taxa de transmissão, conforme mostrado na equação 3 (RICHARDS, 2005).

$$NBR = \frac{1}{t_{bit}} \tag{3}$$

Os quatro segmentos não sobrepostos são: segmento de sincronização (SyncSeg), segmento de propagação (PropSeg), primeiro segmento de fase (PhaseSeg1) e segundo segmento de fase (PhaseSeg2).

Tais segmentos são necessários para sincronizar os nós (SyncSeg), compensar os atrasos físicos da transmissão (PropSeg) e compensar erros na transição de borda do *bit* (PhaseSeg1 e PhaseSeg2).

Outros elementos que compõe a temporização são a ponto de amostragem (SP) e o SJW (*Synchronization Jump Width*).

O ponto de amostragem define o local onde o nível lógico é lido e interpretado. Está localizado no final do primeiro segmento de fase. Seu valor é avaliado em porcentagem.

A relação entre o ponto de amostragem e os segmentos não sobrepostos da

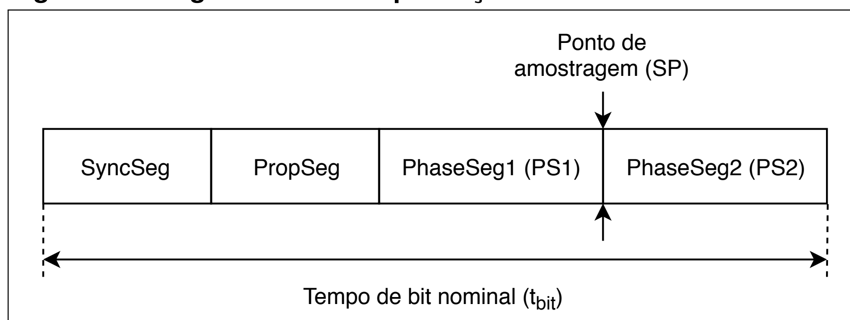
temporização é mostrada na equação 4.

$$SP(\%) = \frac{SyncSeg + PropSeg + PS1}{SyncSeg + PropSeg + PS1 + PS2} \cdot 100 \tag{4}$$

O SJW tem por finalidade a ressincronização. É programável e com ele o primeiro segmento de fase pode ser alongado e o segundo, encurtado.

Os segmentos não sobrepostos, bem como o ponto de amostragem são melhores compreendidos com o auxílio da figura 26.

Figura 26 – Segmentos da temporização CAN



Fonte: AN754.

A expressão que relaciona o tempo de *bit* com os segmentos não sobrepostos é mostrada na equação 5.

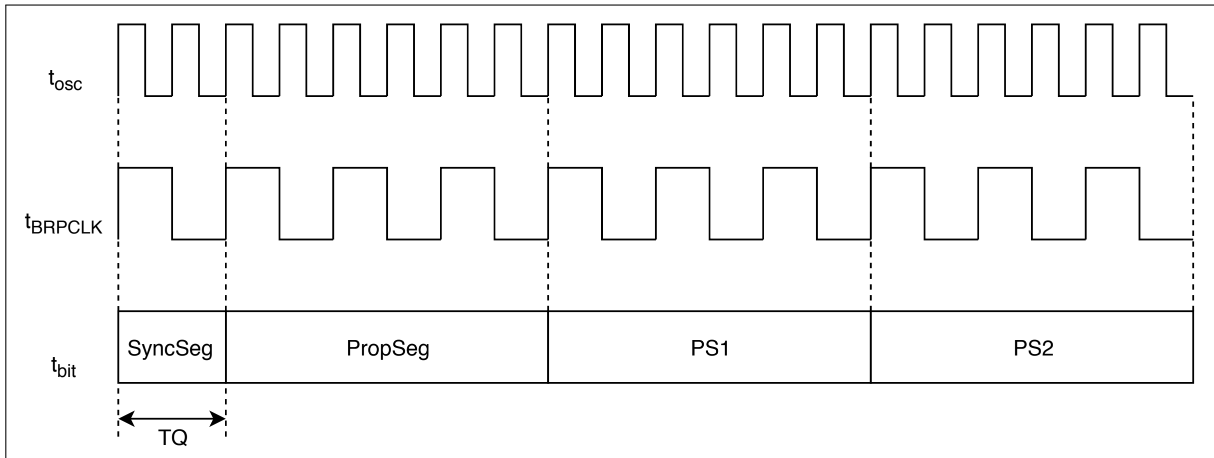
$$t_{bit} = t_{SyncSeg} + t_{PropSeg} + t_{PS1} + t_{PS2} \tag{5}$$

Há outra relação possível entre o tempo de *bit* (t_{bit}) e os quatros segmentos (SyncSeg, PropSeg, PhaseSeg1 e PhaseSeg2). Ao invés de utilizar o tempo de cada segmento, é mais comum utilizar unidades inteiras positivas múltiplas do período de oscilação do microcontrolador. Tais unidades inteiras positivas recebem o nome de tempo de quantização (T_q - *Time quantum*). Para haver a multiplicidade existe um divisor, denominado *Bauld Rate Prescaller* (BRP) nos microcontroladores da Microchip. A definição de T_q bem como a relação de t_{bit} é mostrada na equação 6 e 7. Isto também pode ser visualizado na figura 27.

$$T_q = \frac{2 \cdot (BRP + 1)}{F_{osc}} \tag{6}$$

$$t_{bit} = T_q \cdot (SyncSeg + PropSeg + PS1 + PS2) \tag{7}$$

Figura 27 – Relação entre frequência de operação, divisor e segmentos



Fonte: AN754.

Tabela 4 – Valores possíveis de Tq

Elemento	**Número de Tq's**
SyncSeg	1
PropSeg	1 - 8
PS1	1 - 8
PS2	2 - 8
SJW	1 - 4

Fonte: Adaptado de ISO 11898-1.

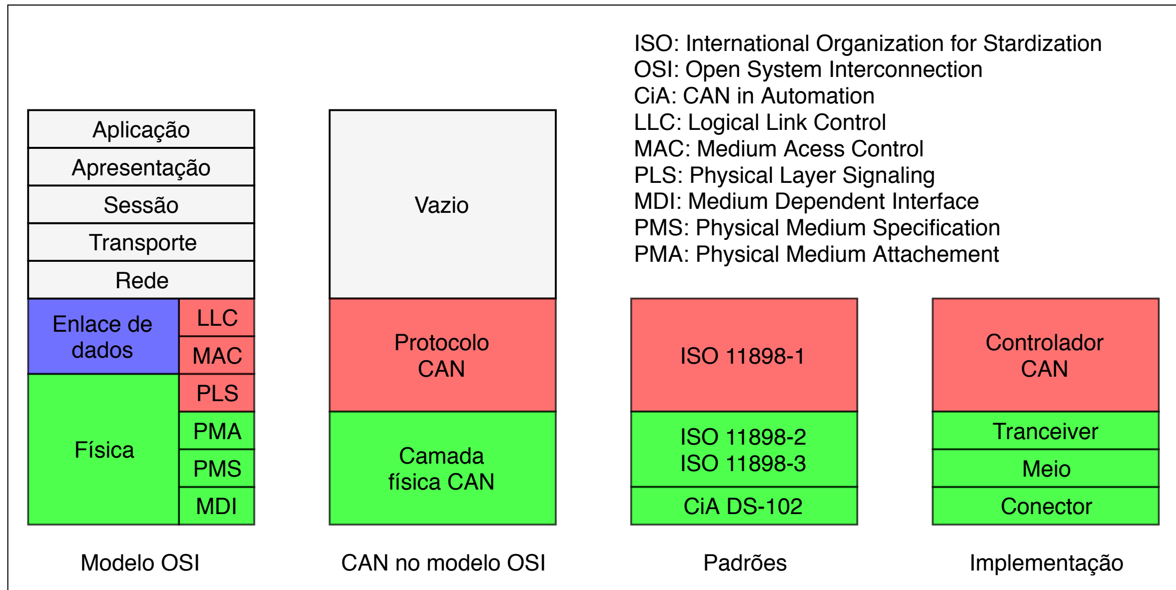
Utilizando desta segunda abordagem, que relaciona o tempo de *bit* (t_{bit}) e o tempo de quantização (T_q), define-se cada elemento da temporização como um número múltiplo de T_q. Tais intervalos de valores são padronizados, conforme a tabela 4. O primeiro segmento é fixo, sendo os demais programáveis (INTERNATIONAL STANDARDIZATION ORGANIZATION, 2003a).

2.2.4 Modelo OSI do protocolo CAN

Diversos protocolos de comunicação são descritos utilizando o modelo OSI, apresentado na tabela 1. O protocolo CAN implementa a camada física e parte da camada de enlace de dados do modelo OSI, conforme mostra a figura 28. As camadas acima da enlace de dados podem ser implementadas, de acordo com a necessidade da aplicação.

É possível observar que o padrão ISO 11898-1 abrange os protocolos *Physical Layer Signaling* (PLS), *Medium Acess Control* (MAC) e *Logical Link Control* (LLC) definidos no modelo OSI (INFORMATIK, 2018 (Acesso em 21 jun. 2018)). O protocolo CAN abrange apenas o controlador CAN (implementado no microcontrolador),

Figura 28 – CAN no modelo OSI



Fonte: Adaptado de Vector Informatik.

enquanto que o padrão ISO 11898-2 é implementado no *tranceiver*.

O barramento e a conexão do barramento CAN com o nó não é padronizada por nenhuma ISO, porém a organização CAN in Automation (1996) padroniza os mesmos. Neste protocolo há também a padronização do comprimento máximo do barramento em função da taxa de transmissão (tabela 5).

Tabela 5 – Taxas transmissão em função do comprimento do barramento

Taxa de transmissão ($kbps$)	Tempo nominal de bit (μs)	Comprimento do barramento (m)
1000	1	40
500	2	100
100	10	500
50	20	1000

Fonte: Adaptado de CiA DS-102.

2.3 ADDRESSABLE UNIVERSAL SYNCHRONOUS ASYNCHRONOUS RECEIVER TRANSMITTER

A Universal Synchronous Asynchronous Receiver Transmitter (USART) é um formato de comunicação serial, que permite a transmissão e recepção de dados de forma síncrona ou assíncrona, porém, o modo assíncrono é o mais utilizado.

O modo de comunicação serial síncrona utiliza uma linha de *clock* e outra de dados, enquanto o modo de comunicação serial assíncrono utiliza duas linhas de dados, fazendo que o modo assíncrono seja *full-duplex*.

A aplicação mais comum da interface USART é a transferência de dados entre um periférico com um computador pessoal (PC), usando uma porta serial ou mesmo a emulando em uma porta USB, utilizando *chips* dedicados.

Da sigla USART, o “U” representa Universal, ou seja, genérico. Isto não o caracteriza como um protocolo. Esta universalidade faz com que seja compatível com diferentes protocolos seriais, como o RS-232. A maior diferença entre a USART e o protocolo RS-232 é com relação aos níveis lógicos de tensão. Enquanto A USART utiliza níveis lógicos de tensão variando de $0V$ à $5V$, o protocolo RS-232 emprega valores que vão desde $-15V$ até $+15V$ (STRANGIO, 2006).

A transmissão de dados ocorre por meio de quadros, onde cada quadro contém o *bit* de início, os *bits* de dados, o *bit* de paridade e o *bit* de parada. É possível que o *bit* de paridade seja convertido em um *bit* adicional de parada. O formato do quadro é o mesmo mostrado na figura 12.

2.3.1 Conversor USART para USB

Realizar a comunicação de dados entre um dispositivo microprocessado e um computador pessoal, utilizando uma porta Universal Serial Bus (USB) era algo que amedrontava muitos desenvolvedores em um passado não muito distante, devido a sua complexidade. Muitas vezes a escolha mais vaiável era a utilização de um conversor USART para RS232, conectando o PC com o microcontrolador via porta COM.

Com a grande popularização da porta USB, a porta COM caiu em desuso nos computadores pessoais, chegando até não estar presente nos computadores mais recentes. Porém, a necessidade da comunicação entre microcontroladores e compu-

tadores fez surgir o FT232R, um chip conversor USART para USB, produzida pela FTDI (FT232R, 2005).

Após o desenvolvimento deste *chip*, a comunicação micocontrolador – PC tornou-se muito mais fácil, fazendo possível o surgimento de plataformas como o Arduino.

Com um oscilador interno de $12MHz$, é possível que a transferência de dados seja de $183baud$ até $3Mbaud$, compatíveis com as versões USB 1.1 à USB 2.0.

Dentre os pinos presentes no chip FT232R, destacam-se os referentes à alimentação, comunicação serial e comunicação USB, como mostrado na tabela 6.

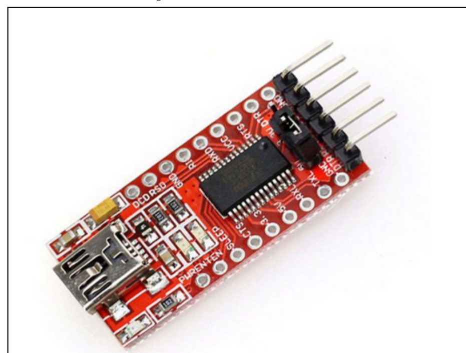
Tabela 6 – Pinos FT232R

Pino	Nome	Tipo	Descrição
5	RXD	Entrada	Entrada de dados da transmissão assíncrona
1	TXD	Saída	Saída de dados da transmissão assíncrona
15	D+	I/O	Sinal diferencial positivo USB
16	D-	I/O	Sinal diferencial negativo USB
20	VCC	Alimentação	Alimentação de $+3{,}3V$ até $+5{,}25V$
7,18 e 21	GND	Alimentação	Pinos de terra do dispositivo

Fonte: Adaptado de FT232R.

Afim de facilitar o uso do chip FT232, existem módulos disponíveis à venda, contendo os resistores, capacitores e conectores necessários para que a comunicação ocorra. Um destes módulos vendido pela SparkFun, é mostrado na figura 29.

Figura 29 – Módulo convesor USART para USB



Fonte: SparkFun

2.4 MICROCONTROLADOR PIC18F258

Uma das grandes fabricantes de microcontroladores é a Microchip, também responsável pela produção de microcontroladores PIC. Estes microcontroladores possuem tecnologia Reduced Instruction Set Computer (RISC) (ZANCO, 2006).

Os PICs são populares, tanto industrialmente como para hobbystas graças ao seu baixo custo, ampla disponibilidade, grande base de usuários, extensa coleção de notas de aplicação, disponibilidade de ferramentas de desenvolvimento de baixo custo ou grátis, e capacidade de programação serial e reprogramação com memória *flash*.

O microcontrolador utilizado no projeto é um PIC18F258, pois além de possuir um controlador CAN em seus periféricos, é facilmente encontrado à venda no mercado nacional. Além do controlador CAN, há outros importantes periféricos, que estão mostrados juntamente com as características deste dispositivo, na tabela 7.

Tabela 7 – Funcionalidades do microcontrolador PIC18F258

Funcionalidades	Descrição
Frequencia de operação	DC - $40MHz$
Memória de dados	768 bytes
Portas I/O	A, B e C
Conversores A/D (10 bits)	5
Comunicações seriais	MSSP, CAN e USART
Fontes de interrupções	17
ICSP	Presente
Encapsulamento	SPDIP-28 e SOIC-28

Fonte: Datasheet PIC18F258.

Este circuito integrado possui vinte e oito pinos, podendo estar presente em dois tipos de encapsulamento: PDIP-28 e SOIC-28 (MICROCHIP TECHNOLOGY INC., 2006). A tabela 8 lista os pinos utilizados neste projeto.

2.4.1 Módulo ADC

Um conversor analógico-digital, muitas vezes denominado por A/D ou ADC, é utilizado na interface entre dispositivos digitais, como microcontroladores, e transdutores. Eles transformam um sinal analógico, contínuo no tempo, num sinal amostrado, discreto no tempo, quantizado dentro de um número finito de valores inteiros, determinado pela resolução característica do conversor em *bits* (8, 10, 12, 16, etc).

O módulo conversor analógico para digital do PIC18F258 possui cinco entra-

Tabela 8 – Principais pinos do microcontrolador

Pino	Nome	Tipo	Descrição
1	!MCLR/Vpp	Entrada/Alimentação	Reset/Tensão de programação ICSP
2	RA0	Entrada	Entrada analógica 0
3	RA1	Entrada	Entrada analógica 1
8,19	Vss	Alimentação	Pinos de terra do dispositivo
9	OSC1	Entrada	Entrada da fonte de oscilação
10	OSC2	Saída	Saída da fonte de oscilação
17	TX	Saída	Saída de dados da transmissão UART
18	RX	Entrada	Entrada de dados da transmissão UART
20	Vdd	Alimentação	Alimentação positiva
23	CANTX	Saída	Saída de dados da transmissão CAN
24	CANRX	Entrada	Entrada de dados da transmissão CAN
27	PGC	Programação	Clock da programação ICSP
28	PGD	Programação	Dados da programação ISCP

Fonte: Datasheet PIC18F258.

das, localizadas nos pinos 2, 3, 4, 5 e 7. Elas convertem níveis de tensão contínuos, situados no intervalo de $0V$ até $5V$, em sinais digitais, com resolução de $10bits$.

Existem quatro registradores relacionados à conversão A/D, dois relacionados com a configuração (ADCON0 e ADCON1) e dois relacionados com o valor digital convertido (ADRESH e ADRESL). O formato dos registradores destinados à configuração são mostrados no anexo (A.27 e A.28). Estes registradores estão documentados na seção 20.0 do *datasheet* do microcontrolador.

Em uma conversão A/D, a temporização é importante. Segundo o datasheet do microcontrolador PIC18F258, o tempo de conversão por *bit* é chamado de T_{AD}. Uma conversão com resolução de 10 *bits* demanda um tempo de 12 T_{AD}. Para uma correta conversão, a fonte de clock deve ser no mínimo igual ao valor do T_{AD}, que é de $1{,}6\mu s$.

De acordo com a tabela 9, é possível que os divisores 2, 4, 8, 16, 32 e 64 sejam selecionados, de forma a servir como o *clock* do módulo ADC. O tempo de conversão por *bit* é relacionado com a frequência de oscilação de acordo com expressão 8.

$$T_{AD} = \frac{1}{F_{osc}} \cdot divisor \tag{8}$$

Tabela 9 – T_{AD}, em μs, em função de F_{OSC}

	Opção	Frequência de oscilação (MHz)						
Divisor	**ADCS2:ADCS0**	**2**	**4**	**8**	**10**	**16**	**20**	**32**
2	000	1	0,5	0,25	0,2	0,125	0,1	0,0625
4	100	2	1	0,5	0,4	0,25	0,2	0,125
8	001	4	2	1	0,8	0,5	0,4	0,25
16	101	8	4	2	1,6	1	0,8	0,5
32	010	16	8	4	3,2	2	1,6	1
64	110	32	16	8	6,4	4	3,2	2

Fonte: Colocar o datasheet.

2.4.2 Módulo CAN

O módulo CAN implementa o CAN 2.0A e CAN 2.0B, ambos definidos na especificação da BOSCH (BOSCH et al., 1991). Todos os formatos de quadros da comunicação CAN são aceitas por este módulo, usando formato de identificador padrão ou estendido.

O PIC18F258 é dotado de três *buffers* de transmissão e dois *buffers* de recepção, onde os *buffers* de transmissão podem ter prioridades diferentes (quatro níveis de prioridade). Para cada *buffer* de recepção há uma máscara de aceitação (*acceptance mask*). Há no total seis filtros associados à recepção de mensagens, sendo dois filtros para um *buffer* (RXB0) e quatro para o outro (RXB1).

Os registradores relacionados com as máscaras são RXMnSIDH e RXMnSIDL, mostrados no Anexo A (A.21 e A.22). Os relacionados com os filtros são RXFnSIDH e RXFnSIDL, também mostrados no Anexo A (A.19 e A.20). A letra "n" informa qual dos dois *buffers* de recepção é utilizado.

Como as mensagens transmitidas e recebidas neste projeto usam apenas 11 *bits* em seu identificador, os registradores TXBnSIDH e TXBnSIDL são utilizados, onde a letra "n" denota um dos três buffers de transmissão utilizados, podendo ser 0, 1 ou 2. As seções A.9 e A.10, presentes no Anexo A, mostram estes registradores.

O microcontrolador PIC18F258 possui três registradores destinados à configuração da temporização do barramento CAN. Eles estão documentados na seção 19.2.4 do *datasheet* deste microcontrolador e nas seções A.23, A.24 e A.25 do anexo deste documento.

Tais registradores recebem os valores dos segmentos não sobrepostos, mostrados na figura 26, com base na tabela 4.

Os seis *bits* menos significativos do registrador BRGCON1 recebem o valor do divisor Bald Rate Prescaler (BRP). Portanto, o valor do BRP no microcontrolador deve estar situado entre os valores 0 e 63.

Afim de relacionar a taxa de transmissão, frequência de oscilação e os segmentos de temporização não sobrepostos, obtém-se a equação 9, que é obtida de acordo com os seguintes passos:

- Passo 1: substitui-se a equação 6 na equação 7.
- Passo 2: da equação resultante no passo 1, substitui-se na equação 3.
- Passo 3: isola-se a variável BRP.

$$BRP = \frac{F_{osc}}{2 \cdot NBR \cdot (SyncSeg + PropSeg + PS1 + PS2)} - 1 \tag{9}$$

A transmissão dos dados ocorre com grupo de registradores de 8 *bits* cada um denominados TXBnDm (seção A.11), enquanto a recepção ocorre com o grupo de registradores RXBnDm (seção A.18).

2.4.3 Módulo USART

O módulo USART é um dos três módulos de comunicação serial presentes no microcontrolador PIC18F258. Nele é possível a seleção do modo assíncrono e modo síncrono, onde o primeiro é *full-duplex* e o segundo é *half-duplex*.

Para o projeto descrito neste documento, o modo de operação assíncrono é o escolhido, devido à sua compatibilidade com conversores UART para USB disponíveis. A sigla UART difere da USART pela ausência da letra ‘S’, que denota síncrono.

Neste modo, a USART usa o formato de *bit Non-Return-to-Zero* (NRZ). Sua estrutura compõe de um *bit* de início (*Start bit*), oito ou nove *bits* de dados e um *bit* de parada (*Stop bit*). É mais comum o uso de oito *bits* de dados. A interface USART transmite o *bit* menos significativo (LSb) do byte primeiro.

Não há nenhum método de detecção de erros via *hardware*, mas é possível implementá-lo via *software*. Isto pode ser feito utilizando o formato de 9 *bits* de dados sendo o nono *bit* representando a paridade.

De acordo com a folha de dados do microcontrolador PIC18F258, o módulo USART consiste de quatro elementos:

- Gerador da taxa de dados: Ajusta a taxa de transmissão da comunicação USART.
- Circuito de amostragem: Os dados recebidos são amostrados três vezes por este circuito.
- Transmissor assíncrono: Envia a mensagem.
- Receptor assíncrono: Recebe a mensagem.

Toda a comunicação serial assíncrona é feita basicamente por três registradores, sendo dois referentes à configuração (TXSTA e RCSTA) e um terceiro que recebe o valor da taxa de comunicação (SPBRG). Estes registradores estão documentados na seção 18.0 do *datasheet*, sendo também expostos no Anexo A deste documento (seções A.4 e A.5).

O segundo *bit* (BRGH) do registrador TXSTA em conjunto com o registrador SPBRG, determinam a velocidade da comunicação, como mostra a equação 10. O valor de SPBRG deve estar entre 0 e 255, pois é um registrador de 8 *bits*. As diferentes fórmulas para BRGH = 0 e BRGH = 1 são necessárias para adequar o valor de SPBRG entre a faixa já citada, pois o primeiro refere-se à baixas velocidades, enquanto o segundo o oposto.

$$SPBRG = \begin{cases} \dfrac{F_{osc} - 64 \cdot BR_{des}}{64 \cdot BR_{des}}, & se \quad BRGH = 0 \\ \\ \dfrac{F_{osc} - 16 \cdot BR_{des}}{16 \cdot BR_{des}}, & se \quad BRGH = 1 \end{cases} \tag{10}$$

A variável BR_{des} é denominada *bauld rate* desejada, ou seja, a taxa de transmissão requerida na comunicação USART. Alguns dos valores comumente utilizados são: $2400bps$, $9600bps$, $19200bps$, 38400 bps e $57600bps$.

Com o cálculo do registrador SPBRG, percebe-se que nem sempre um valor inteiro é encontrado. Para contornar isso, arredonda-se este valor para o inteiro mais

próximo. Porém, isso torna a taxa de transmissão diferente que a desejada, conforme mostra a equação 11, onde BR_{calc} é denominada taxa de transmissão calculada.

$$BR_{calc} = \begin{cases} \dfrac{F_{osc}}{64 \cdot SPBRG + 64}, & se \quad BRGH = 0 \\ \\ \dfrac{F_{osc}}{16 \cdot SPBRG + 16}, & se \quad BRGH = 1 \end{cases} \tag{11}$$

Para garantir uma boa comunicação de dados, é necessário que a diferença entre a taxa de transmissão calculada e a desejada seja a menor possível. De acordo com testes práticos, um valor de erro (equação 12) menor que $1{,}5\%$ é aceitável.

$$ERRO(\%) = \frac{|BR_{calc} - BR_{des}|}{BR_{des}} \cdot 100 \tag{12}$$

Quando é utilizada interrupção referente a transmissão e/ou recepção de dados da USART, torna-se necessário a configuração de três registradores (INTCON, PIR1 e PIE1). Estes registradores estão documentados na seção 8.0 do datasheet, além de serem anexados ao trabalho nas seções A.1, A.2 e A.3.

Os dados são transmitidos pelo registrador TXREG enquanto os dados recebidos são armazenados no registrador RCREG.

2.5 TRANSCEIVER MCP2551

A conexão entre o microcontrolador PIC18F258 e o barramento CAN é feito com um *transceiver* (transceptor). Este dispositivo combina as funções de um receptor e um transmissor em um único dispositivo.

O *transceiver* escolhido foi o Microchip MCP2551, pois além de ser da mesma marca do microcontrolador utilizado, é o mais acessível no mercado nacional.

O MCP2551 é um *tranceiver* totalmente compatível com padrão ISO-11898-2 (INTERNATIONAL STANDARDIZATION ORGANIZATION, 2003b). Algumas de suas características elétricas são até melhores que as exigidas neste padrão (RICHARDS, 2002), garantindo uma comunicação de dados mais robusta. Isto é mostrado na tabela 10.

Tabela 10 – Comparação entre ISO11898-2 e MCP2551

Parâmetro	ISO11989-2		MCP2551		Unidade	Comentário
	Mínimo	Máximo	Mínimo	Máximo		
Tensão CC no barramento CAN	-3	+32	-40	+40	V	Excede a ISO
Tensão transitória no barramento	-150	+100	-250	+250	V	Excede a ISO
Tensão de modo comum no barramento	-2	+7	-12	+12	V	Excede a ISO
Tensão dominante de saída (CANH)	+2,75	+4,50	+2,75	+4,50	V	Compatível com a ISO
Tensão dominante de saída (CANL)	+0,5	+2,25	+0,5	+2,25	V	Compatível com a ISO
Tensão diferencial recessiva	-500	+50	-500	+50	mV	Compatível com a ISO

Fonte: Adaptado de AN228

Este circuito integrado possui oito pinos, podendo estar presente em dois tipos de encapsulamento: PDIP-8 e SOIC-8. A tabela 11 lista o número do pino, seu nome e a sua função.

Segundo o *datasheet* do MCP2551 (MICROCHIP TECHNOLOGY INC., 2010a), o pino VREF é definido como $VDD/2$, onde VDD deve estar entre $4{,}5V$ e $5{,}5V$.

Os pinos CANL e CANH devem estar conectados diretamente ao barramento CAN, podendo estar com terminador de 120Ω em paralelo (caso seja primeiro ou último

Tabela 11 – Principais pinos do transceiver

Pino	Nome	Tipo	Descrição
1	TXD	Entrada	Entrada de dados transmitidos
2	Vss	Alimentação	Pino de terra do dispositivo
3	Vdd	Alimentação	Alimentação positiva
4	RXD	Saída	Saída dos dados recebidos
5	Vref	Alimentação	Referência de tensão
6	CANL	Entrada/Saída	CAN Low
7	CANH	Entrada/Saída	CAN High
8	Rs	Entrada	Controle da taxa de variação

Fonte: Adaptado de MCP2551.

nó).

Ao contrário do que se espera à primeira impressão, o pino TXD é um pino de entrada, enquanto o pino RXD é um pino de saída.

O pino RS é usado para selecionar entre os modos *High-Speed*, *Slope-Control* ou *Standby*. A seleção é feita com base no nível de tensão aplicado neste pino, conforme mostra a tabela 12.

Tabela 12 – Modos de operação MCP2551

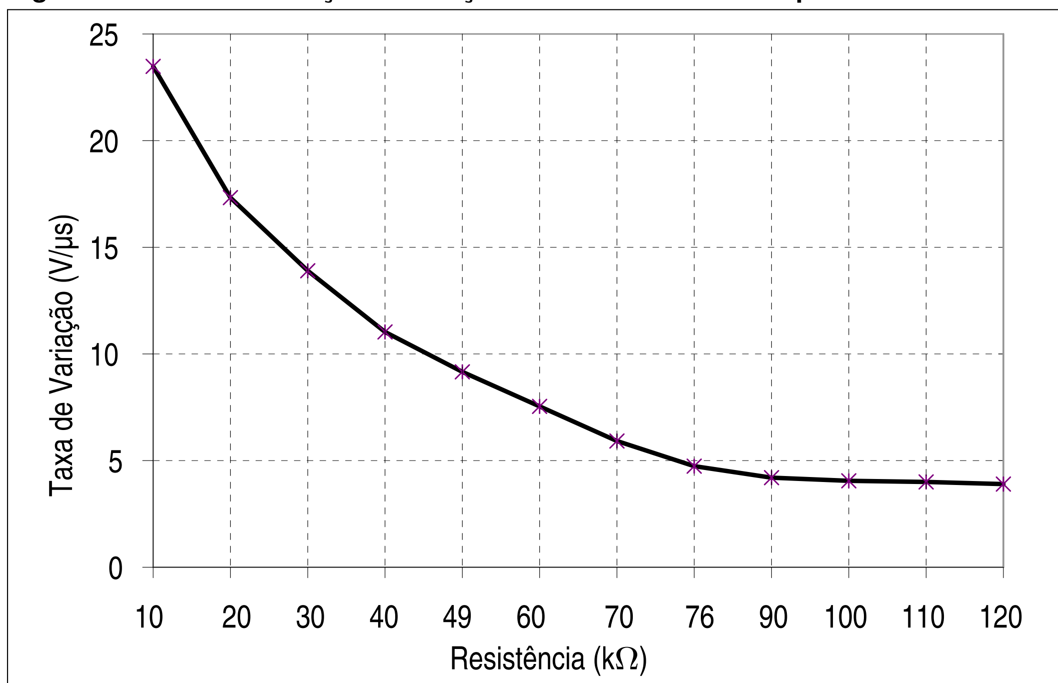
Modo	Corrente no pino Rs	Tensão resultante no pino Rs
Standby	$-I_{Rs} < 10\mu A$	$V_{Rs} > 0{,}75Vdd$
Slope-Control	$10\mu A < -I_{Rs} < 200\mu A$	$0{,}4Vdd < V_{Rs} < 0{,}6Vdd$
High-Speed	$I_{Rs} < 610\mu A$	$0 < V_{Rs} < 0{,}3Vdd$

Fonte: Adaptado de MCP2551.

O modo *High-Speed* é utilizado quando se deseja a máxima taxa de transmissão ($1Mbps$), já o modo *Slope-Control* é necessário quando se deseja reduzir a interferência eletromagnética (EMI). No modo *Standby* o transceptor é desativado, consumindo menos energia.

A taxa de variação (*Slew Rate*) é afetada com a seleção do modo *Slope-Control*. Isto pode ser melhor observado na figura 30.

Figura 30 – Taxa de variação em função da resistência entre o pino RS e GND



Fonte: Adaptado de MCP2551.

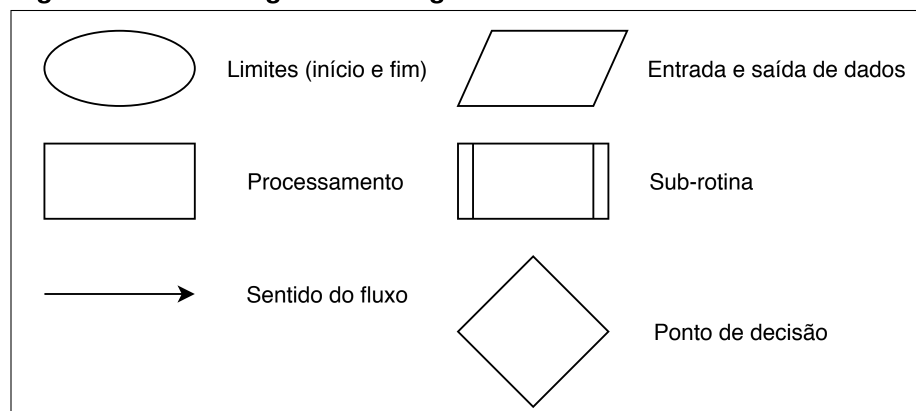
2.6 LINGUAGEM DE PROGRAMAÇÃO

Uma linguagem de programação é uma forma abstrata de especificar instruções ou regras para o computador (ou outro dispositivo microprocessado). Escrever diretamente na forma que o computador interpreta (linguagem máquina) é extremamente difícil, o que é sanado com as linguagens de programação.

Existem diversas linguagens de programação, sendo as mais usadas: JavaScript, HTML e CSS. A linguagem Java aparece na quinta posição, enquanto que a linguagem C aparece na décima primeira posição (STACK OVERFLOW, 2018).

Afim de facilitar o entendimento da lógica de programação, foi escolhido o uso de fluxogramas para a representação dos programas. A simbologia utilizada é mostrada na figura 31.

Figura 31 – Simbologia dos fluxogramas



Fonte: Autoria própria.

2.6.1 Linguagem C

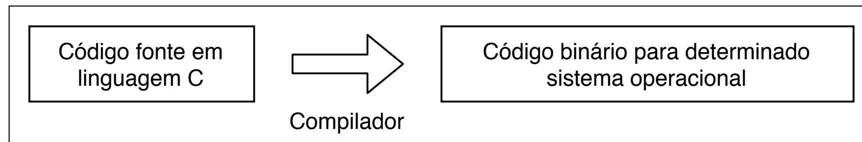
A linguagem C é a principal representante da programação estruturada (procedural), sendo amplamente aplicada em sistemas embarcados, onde o conhecimento do hardware se faz necessário para a elaboração de um bom programa.

Em uma linguagem procedural é possível invocar procedimentos (funções, métodos, rotinas e sub-rotinas), fornecendo argumentos aos mesmos, podendo obter valores de retorno. Isto evidencia o conceito de modularidade presente neste modelo de linguagem.

Um programa escrito em linguagem C deve ser traduzido para uma linguagem

de mais baixo nível, como Assembly ou código máquina. Esta tradução é feita com um programa, ou grupo de programas denominados compiladores (figura 32).

Figura 32 – Processo de compilação



Fonte: Caelum.

Afim de tornar o código mais modular, existe o conceito de bibliotecas. As bibliotecas são caracterizadas por um conjunto de rotinas, contendo funções de entrada, saída e processamento de informações. Existem bibliotecas padrão da linguagem C, como a stdio.h, mas é possível também o desenvolvimento de bibliotecas pelo programador.

As principais características da linguagem de programação C são listadas abaixo.

- Simplicidade;
- Facilidade de uso;
- Confiabilidade.

A primeira versão desta linguagem de programação é datada do ano de 1978, sendo desenvolvida por Kemighan e Ritchie (1978), nos laboratórios Bell. Afim de garantir maior portabilidade, no ano de 1985 estabeleceu-se o padrão oficial da linguagem C, denominado C ANSI.

2.6.1.1 Elementos básicos da linguagem C

Na linguagem C existem os identificadores, que são utilizados para nomear as constantes, variáveis e funções definidos pelo programador. Como regra, eles devem começar por uma letra, maiúscula ou minúscula. Não é permitido o uso de caracteres especiais, com exceção do caractere sublinhado. Somente a partir do segundo caractere é permitido o uso de números. Exemplos de identificadores da linguagem C: j, count, Can_ Init, aux1.

Os tipos de dados dependem do compilador utilizado, onde há também variação do seu tamanho (em *bits*). Os modificadores informam se o tipo de dado é sem sinal (*unsigned*), com sinal (*signed*), reduzida (*short*) ou estendida (*long*).

É possível realizar operações aritméticas, relacionais, lógicas e operações em nível de *bit*, conforme mostrado nas tabelas 13, 14, 15 e 16.

Tabela 13 – Operadores aritiméticos

Operação	Símbolo
Adição	$+$
Subtração	$-$
Divisão	$/$
Multiplicação	$*$
Resto	$\%$
Atribuição	$=$

Fonte: The C programming language.

Os operadores relacionais retornam verdadeiro, ou seja nível lógico alto (1) ou falso (0). Qualquer valor diferente de zero é considerado como verdadeiro.

Tabela 14 – Operadores relacionais

Operação	Símbolo
Menor que	$<$
Maior que	$>$
Menor ou igual	$<=$
Maior ou igual	$>=$
Igualdade	$==$
Desigualdade	$!=$

Fonte: The C programming language.

Os operadores lógicos realizam comparações entre expressões, sendo muito utilizados em estruturas de decisão.

Tabela 15 – Operadores lógicos

Operação	Símbolo
e (conjunção)	$\&\&$
ou (disjunção)	$\vert\vert$
não (negação)	$!$

Fonte: The C programming language.

As operações em nível de *bit* são utilizadas para testar, atribuir e deslocar *bits*. Um deslocamento de *bits* à esquerda equivale a dobrar o valor, enquanto que um deslocamento à direita equivale a operação inversa. Tais operações permitem que sejam implementadas máquinas de estados, seleções de itens em um menu e manipulações de variáveis.

Somente é possível realizar operações em nível de *bit* nas variáveis cujo tipo de dados não seja de ponto flutuante (*float* e *double*).

Tabela 16 – Operadores em nível de bit

Operação	Símbolo
Deslocamento à esquerda	<<
Deslocamento à direita	>>
e (and)	&
ou (or)	|
ou exclusivo (xor)	∧
não (not)	!

Fonte: The C programming language.

A condição, passada como parâmetro nas estruturas condicionais (ou de decisão), é qualquer expressão que possa ser avaliada com o valor verdadeiro ou falso. Isto pode ser melhor observado no algoritmo 1, que apresenta a estrutura *if-else* e *switch*, respectivamente. É possível observar que é utilizado chaves ({}) para delimitar o bloco de comandos.

Algoritmo 1 – Estrutura de decisão

```
if(condicao)
{
  \\ Sequencia de comandos 1
}
else
{
  \\ Sequencia de comandos 2
}

switch(expressao)
{
  case valor1:
    \\ Sequencia de comandos 1
    break;
  case valor2:
    \\ Sequencia de comandos 2
    break;
  case valorN:
    \\ Sequencia de comandos N
    break;
  default:
    \\ Sequencia padrao
}
```

Fonte: The C programming language.

Se a condição é avaliada como verdadeira, é executado o bloco de comandos 1, na estrutura *if*. É executada o bloco de comandos 2, caso seja falsa. A estrutura

switch difere da estrutura *if-else* por apresentar um valor padrão (*default*) para caso nenhuma condição (*case*) seja verdadeira.

Existem três diferentes estruturas de repetição: *while*, *do-while* e *for* (algoritmo 2). São compostas pela inicialização, condição e incremento. A estrutura *do-while* difere das demais pelo fato que o bloco de comandos é sempre executado na primeira execução.

Algoritmo 2 – Estrutura de repetição

```
\\ inicializacao
while(condicao)
{
  \\ Sequencia de comandos
  \\ Incremento
}

do{
  \\ Sequencia de comandos
}while(condicao);

for(inicializacao; condicao; incremento)
{
  \\ Sequencia de comandos
}
```

Fonte: The C programming language.

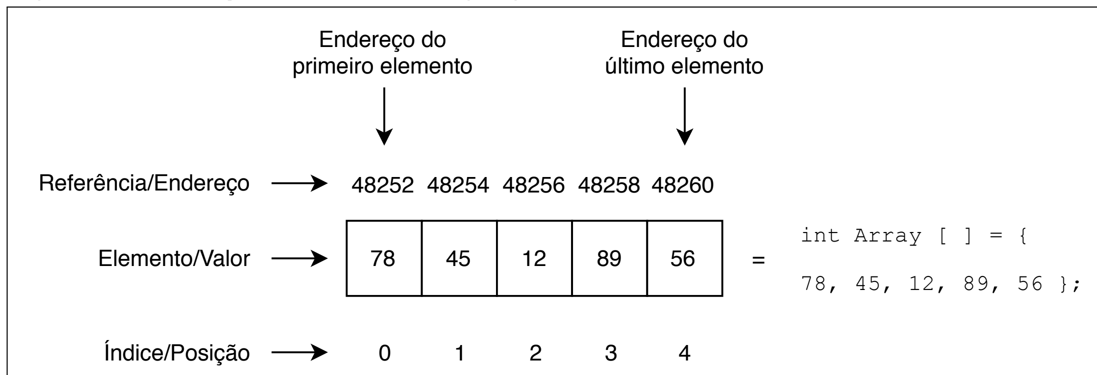
As funções, também chamadas de sub-rotinas, apresentam a característica de retornar um tipo de dado, podendo ser fornecido argumentos (parâmetros) para que sejam processados. As funções são as grandes responsáveis pelo conceito de modularidade na liguagem de programação C.

Nas funções possuem um valor retornado, nome, parâmetros e corpo. Porém, em linguagem C é possível declarar um função evidenciando seu tipo de retorno, nome e parâmetros, otimindo o seu corpo. Isto é chamado de protótipo de função.

É possível armazenar um conjunto de dados, de mesmo tipo, com um mesmo nome (mesmo identificador). A individualização é feita por um índice, que começa pelo índice 0. A esta característica se dá o nome de vetor. A declaração de um vetor unidimensional é mostrado abaixo. Um exemplo de vetor é mostrado na figura 33.

Uma das características que tornam a linguagem C flexível é a utilização de ponteiros. Os ponteiros ou apontadores são variáveis que armazenam o endereço de memória de outras variáveis. Eles são úteis para a manipulação de vetores e retorno de mais de um valor em uma função. O operador $\&$ retorna o endereço da variável

Figura 33 – Exemplo de vetor em linguagem C



Fonte: Autoria própria.

enquanto o operador $*$ retorna o conteúdo da variável de endereço $\&$. No algoritmo 3, o valor da variável ptr é o endereço de memória (48252) da primeira posição do Array[0] (figura 33).

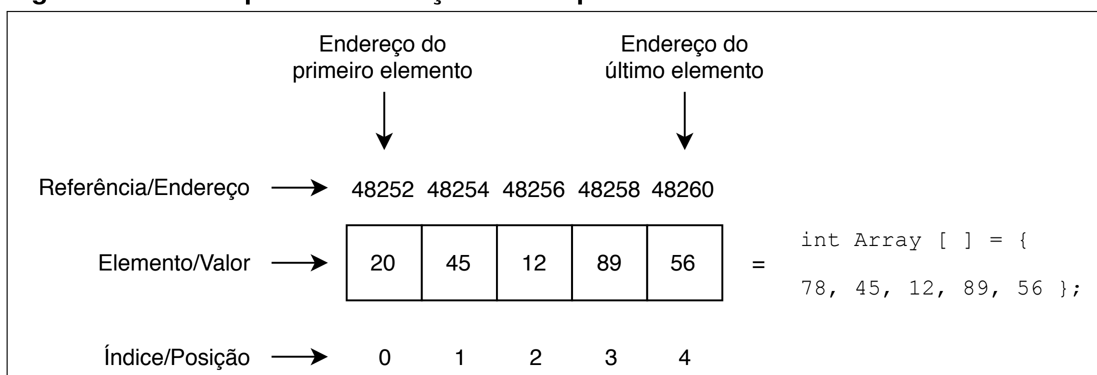
Algoritmo 3 – Ponteiros

```
int *ptr;
ptr = &Array[0];
*ptr = 20;
```

Fonte: Autoria própria.

Para modificar o valor da variável Array[0] (78) por meio de ponteiros, é necessário utilzar o operador $*$, como mostra o algoritmo 3. A figura 34 mostra como o vetor Array foi modificado com o ponteiro ptr.

Figura 34 – Vetor após a modificação do seu primeiro elemento

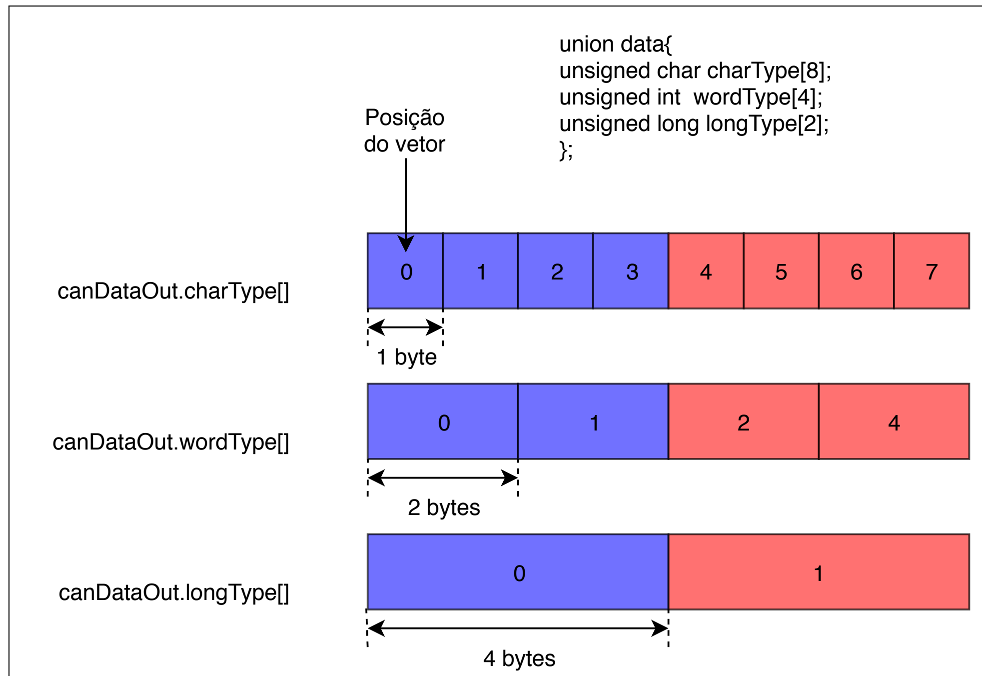


Fonte: Autoria própria.

Uma união (*union*) é uma variável que pode armazenar dados de diferentes tipos e tamanhos juntos. Esta característica permite que diferentes tipos de dados sejam armazenados em um único local de memória. Um exemplo de como é alocada

a memória em uma união é mostrada na figura 35. Este tipo de dado será utilizado no recebimento e envio de mensagens para o barramento CAN.

Figura 35 – Exemplo de união em linguagem C



Fonte: Autoria própria.

2.6.1.2 MPLAB X IDE

O MPLAB X IDE (Integrated Development Enviroment) é um *software* que pode ser utilizado em Windows, MAC OS e Linux para o desenvolvimento de aplicações utilizando microcontroladores e controladores digitais da Microchip. É baseado no *open source NetBeans IDE* da Oracle (MICROCHIP TECHNOLOGY INC., 2015).

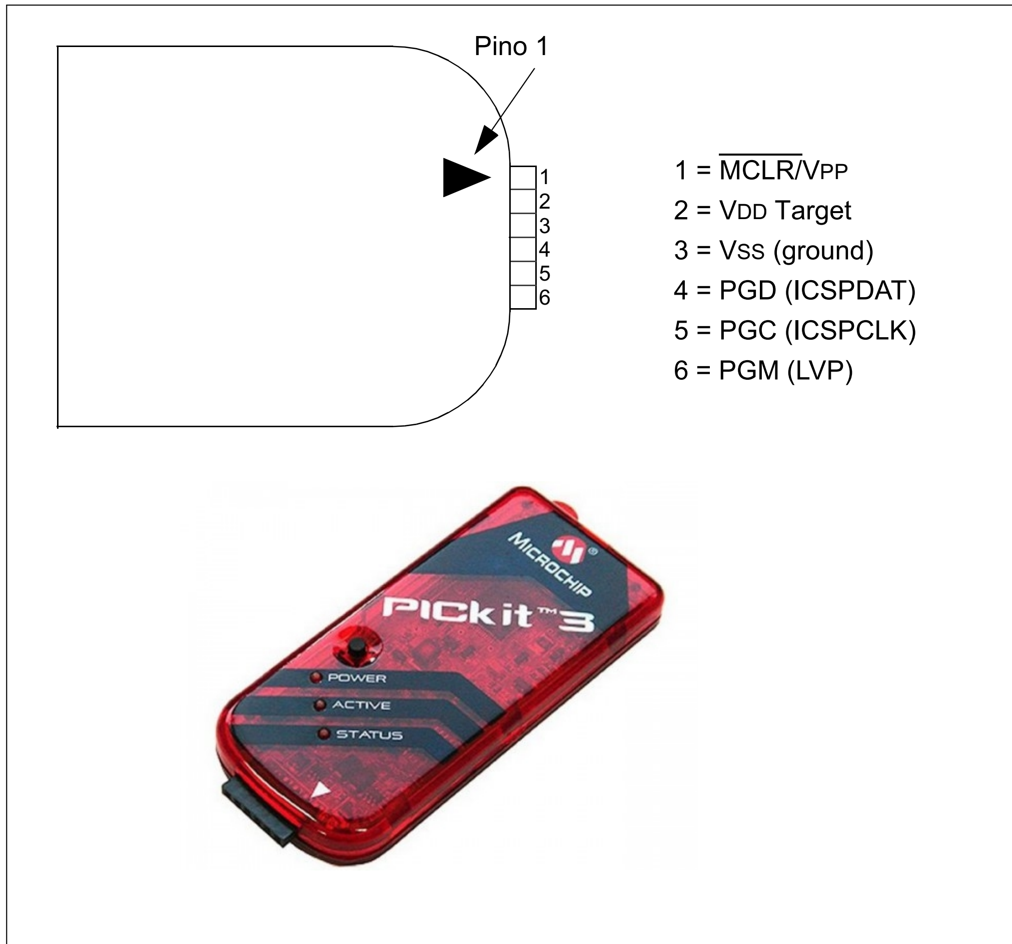
Este ambiente de desenvolvimento é gratuito e possui uma grande variedade de documentos que auxiliam na sua utilização, sendo facilmente encontradas no site da Microchip. A versão utilizada neste projeto é MPLAB X IDE v3.55.

2.6.1.3 Gravador PICkit 3

Para a gravação do microcontrolador PIC18F258 foi utilizado o PICkit 3. Este dispositivo é produzido pela Microchip, sendo compatível com os microcontroladores da linha PIC16F, PIC18F, dsPIC33F, PIC24 e PIC32 (MICROCHIP TECHNOLOGY INC., 2010b)

A programação é feita via *In-Circuit Serial Programming* (ICSP), ligando o PICkit 3 em uma porta USB (MICROCHIP TECHNOLOGY INC., 1997). É totalmente compatível com o MPLAB X IDE. A figura 36 mostra este dispositivo, bem como o nome dos pinos referentes a gravação ICSP.

Figura 36 – Gravador PICkit 3



Fonte: Microchip

2.6.1.4 Compilador XC8

O compilador escolhido para a programação do microcontrolador é o MPLAB XC8 C Compiler. Ele possui versão gratuita, mas sem a otimização de código das versões pagas. É o responsável pela conversão do código fonte (.c) em linguagem máquina (.hex).

Sua escolha é justificada pela compatibilidade com o microcontrolador utilizado e integração com a interface MPLAB X IDE. A versão utilizada é v1.41.

A tabela 17 mostra os principais tipos de variáveis suportadas pelo compilador

XC8 (MICROCHIP TECHNOLOGY INC., 2012).

Tabela 17 – Variáveis suportadas pelo compilador XC8

Tipo	Tamanho (bits)	Tipo aritimérico
bit	1	Inteiro
char	8	Inteiro
short	8	Inteiro
int	8	Inteiro
long	32	Inteiro
float	24	Real
double	32	Real

Fonte: Microchip.

2.6.2 Linguagem Java

Existem oito diferentes tipos de dados primitivos na linguagem Java (ORACLE CORPORATION, 2017), sendo seis diferentes representações para números inteiros e duas para números de ponto flutuante (tabela 18). É possível observar que há diferença de tamanho para os tipos de dados suportados pelo compilador XC8 (tabela 17).

Tabela 18 – Variáveis suportadas pela linguagem Java

Tipo	Tamanho (bits)	Tipo aritimérico
boolean	1	Inteiro
byte	8	Inteiro
short	16	Inteiro
char	16	Inteiro
int	32	Inteiro
long	64	Inteiro
float	32	Real
double	64	Real

Fonte: Oracle.

Em uma linguagem de programação como C e Pascal, a situação presente na figura 32 ocorre. O código fonte é compilado para uma linguagem de baixo nível (linguagem máquina), sendo compatível com um sistema operacional em específico. Caso seja necessária a execução do código em outro sistema operacional, é necessário compila-lo novamente.

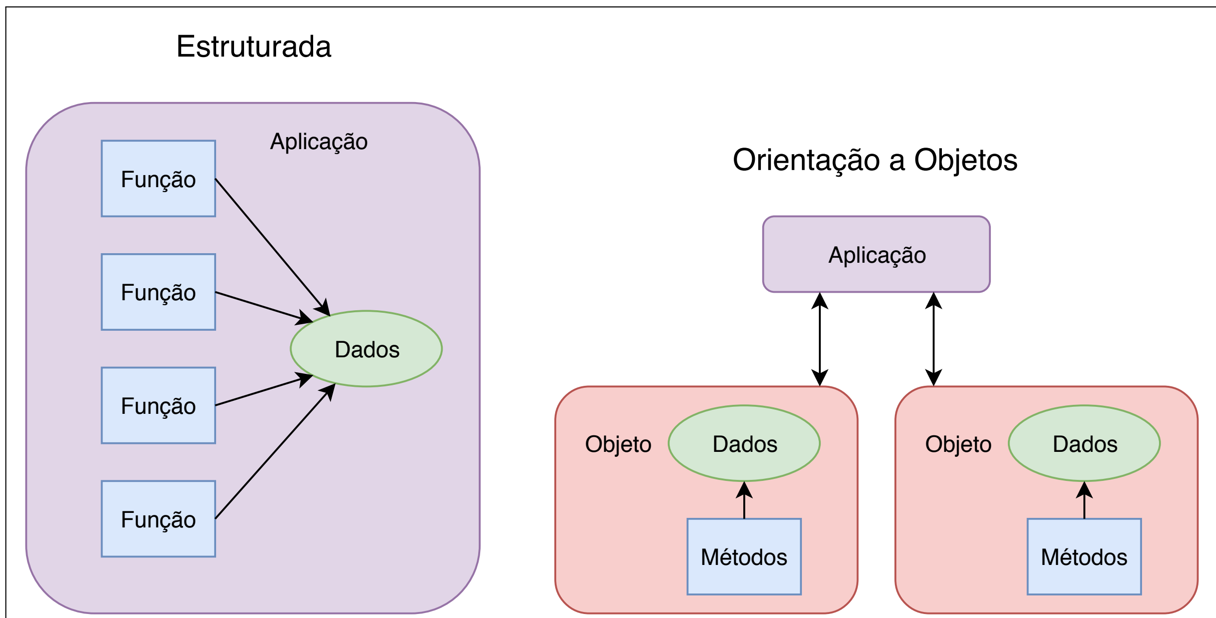
Isso não ocorre na linguagem Java, pois a mesma utiliza o conceito de máquina virtual, denominada *Java Virtual Machine* (JVM) (ORACLE CORPORATION, 2013). Um *software* executado em Windows será igualmente representado em Linux (DEVMEDIA, 2013a).

Muito das características da linguagem de programação em C está presente na linguagem Java, mas com nomes diferentes. As funções em C são equivalentes aos métodos em Java. Também existem as estruturas de decisão, repetição e vetores, já citados na seção anterior.

2.6.2.1 Programação orientada a objetos

Em C e em outras linguagens de programação procedurais, a programação tende a ser orientada a ações, enquanto que em Java tende a ser orientada a objetos (DEITEL; DEITEL, 2010). A figura 37 mostra um comparativo entre uma linguagem procedural e uma orientada a objetos.

Figura 37 – Linguagem procedural x Linguagem orientada à objetos



Fonte: DevMedia.

A Orientação a Objetos é o paradigma de programação mais utilizado para o desenvolvimento de sistemas que necessitem como principal característica o planejamento e implementação do *software* a partir da representação de “coisas” da vida real por meio de objetos.

A Programação Orientada a Objetos (POO) é composta por alguns itens, dentre os quais é possível destacar: classes, objetos, atributos e métodos.

A palavra classe vem da biologia, onde os seres que possuem atributos e comportamentos em comum, mas não iguais, pertencem a mesma classe. A classe é um modelo para múltiplos objetos com características semelhantes.

Um objeto é uma coisa material ou abstrata que pode ser percebida pelos sentidos e descria por meio de suas características, comportamentos e estado atual. Um objeto é uma abstração de conjunto de coisas do mundo real, onde é uma variável cujo tipo de dado é uma classe.

Os atributos são o conjunto de propriedades de uma classe, sendo representado pelas variáveis, já os métodos são o conjunto de funcionalidades de uma classe, podendo ser comparado com as funções (em outras linguagens de programação).

Para uma linguagem ser orientada a objetos, ela deve atender quatro requisitos, que por muitos é chamado os quatro pilares da programação orientada a objetos. Tais pilares são: abstração, herança, polimorfismo e encapsulamento (DEVMEDIA, 2013b).

A abstração é utilizada para a definição de entidades do mundo real, ou seja, é a representação de um objeto real em um sistema computacional. É nesse conceito que são criadas as classes.

O encapsulamento é a técnica utilizada para não expor detalhes internos do programa para o usuário, tornando partes do sistema mais independentes.

Afim da reutilização de código, o conceito de herança é inserido. Na programação orientada a objetos é permitido que uma classe herde atributos e métodos da outra.

O termo polimorfismo significa muitas formas, permitindo que o desenvolvedor use o mesmo elemento, mas de formas diferentes. Este pilar da Orientação a Objetos que tem forte ligação com o pilar da herança.

2.6.2.2 Kit de desenvolvimento Java (JDK)

O *Java Development Kit* (JDK) é um kit de desenvolvimento Java fornecido livremente pela Oracle. Constitui um conjunto de programas que engloba compilador, interpretador e utilitários, fornecendo um pacote de ferramentas básicas para o desenvolvimento de aplicações Java.

A versão utilizada neste trabalho de conclusão de curso é a JDK 8, sendo que a mais recente (JDK 11) é disponibilizada para download pelo site da Oracle (ORACLE CORPORATION, 2018b).

2.6.2.3 NetBeans

O NetBeans é um ambiente de desenvolvimento, *Interface Development Environment* (IDE), sendo totalmente escrito em linguagem Java. Trata-se de uma aplicação de código aberto, onde existe ferramentas para programação em Java, C/C++, PHP e JavaScript (NETBEANS, 2018).

O NetBeans dá o suporte necessário para a criação de *interfaces* gráficas de maneira visual, bem como o desenvolvimento para aplicações em *smartphones* e internet. Esta *interface* é a *interface* oficial para o Java 8 (NETBEANS, 2018).

Como o NetBeans é escrito em Java, é independente de plataforma, funciona em qualquer sistema operacional que suporte a máquina virtual Java (JVM). A versão utilizada neste projeto é NetBeans IDE 8.0.

2.7 EAGLE

O Eagle é um *software* da Autodesk, proprietária de *softwares* como AutoCad e Inventor. O Eagle é um *software* pago, mas que possui versão estudantil (*Light*), com limitação no número de *layers*, tamanho de placa e número de folhas (*sheets*) utilizadas para o esquemático (AUTODESK).

Tabela 19 – Comparativo entre as versões disponíveis do software EAGLE

	Eagle Premium	Eagle Standard	Eagle Free
Área máx. da PCI	Ilimitado	160 cm^2	80 cm^2
Número máx. de layers	16	4	2
Número máx. de sheets	999	99	2
Preço	US$ 510,00	US$ 100,00	US$ 0,00

Fonte: Autodesk.

Apesar destas limitações, este *software* é suficiente para a maioria dos projetos estudantis. Não há limitação quanto ao número de trilhas e uso de componentes SMD.

Outra característica importante deste *software* são as bibliotecas, que já vêm pré-instaladas. Com elas é possível incluir desde simples resistores até complexos circuitos integrados.

A *interface* do *software* Eagle permite que novos componentes sejam criados. Isto pode ser feito facilmente, pois é possível aproveitar encapsulamentos de outros componentes encontrados nas bibliotecas pré-instaladas.

3 DESENVOLVIMENTO

3.1 DESCRIÇÃO DO PROJETO

Este projeto de conclusão de curso consiste de um barramento CAN, composto por cinco nós, denominados Nó Sensor 1, Nó Sensor 2, Nó LCD, Nó Gerador de Mensagens e Nó CANALL.

Os nós Sensor 1, Sensor 2 e Gerador de Mensagens, somente enviam dados ao barramento CAN, enquanto os nós CANALL e LCD são capazes de receber mensagens provenientes do barramento. Há também a possibilidade de envio de mensagens pelo Nó CANALL.

A conexão entre o barramento CAN e um computador pessoal é feita via comunicação serial (USART-USB) a partir do nó CANALL.

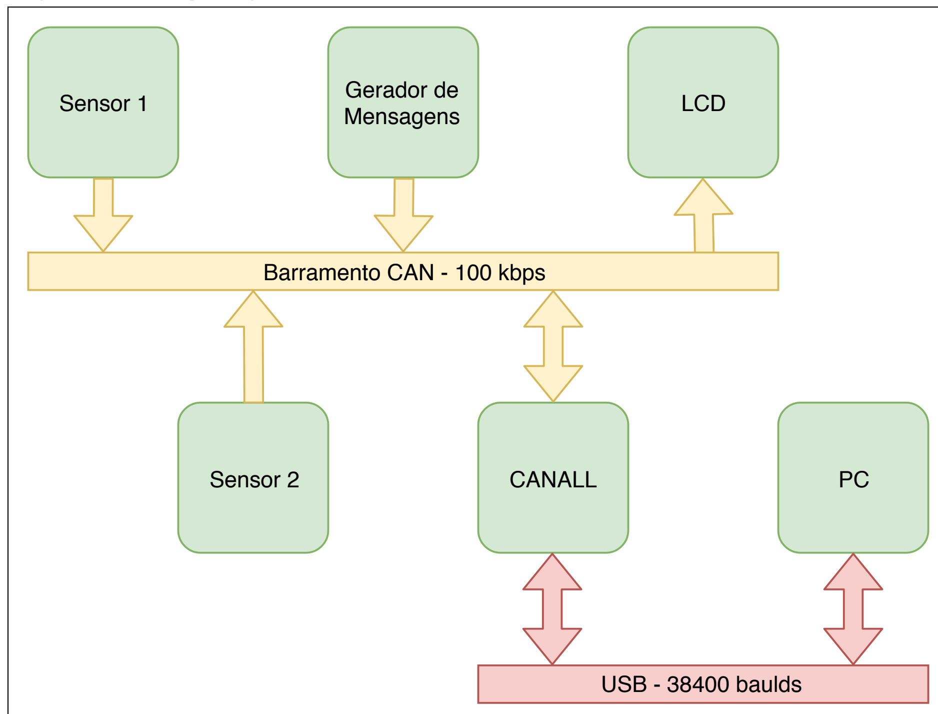
Nos nós Sensor 1 e Sensor 2 é realizada a leitura de um valor analógico, proveniente de um potenciômetro. Esse valor é convertido para um valor digital, com resolução de 10 *bits* e então enviado para o barramento CAN.

A adequação dos níveis lógicos do microcontrolador (PIC18F258) para o barramento CAN é realizada pelo transceptor MCP2551, enquanto que a adequação dos níveis do microcontrolador para o computador pessoal é feita pelo conversor FTDI FT232R.

No computador pessoal foi desenvolvida uma *interface*, em linguagem Java, também denominada CANALL. Nesta *interface* é possível analisar o tráfego de mensagens no barramento CAN, podendo filtrá-las de acordo com a necessidade do usuário da mesma.

A figura 38 mostra o barramento CAN com os nós presentes neste projeto, bem como a sua estrutura interna e a conexão com o computador pessoal. A taxa de transferência do barramento CAN é de $100kbps$, enquanto a taxa de transferência entre o microcontrolador e o computador é de $38400bps$.

Figura 38 – Topologia utilizada

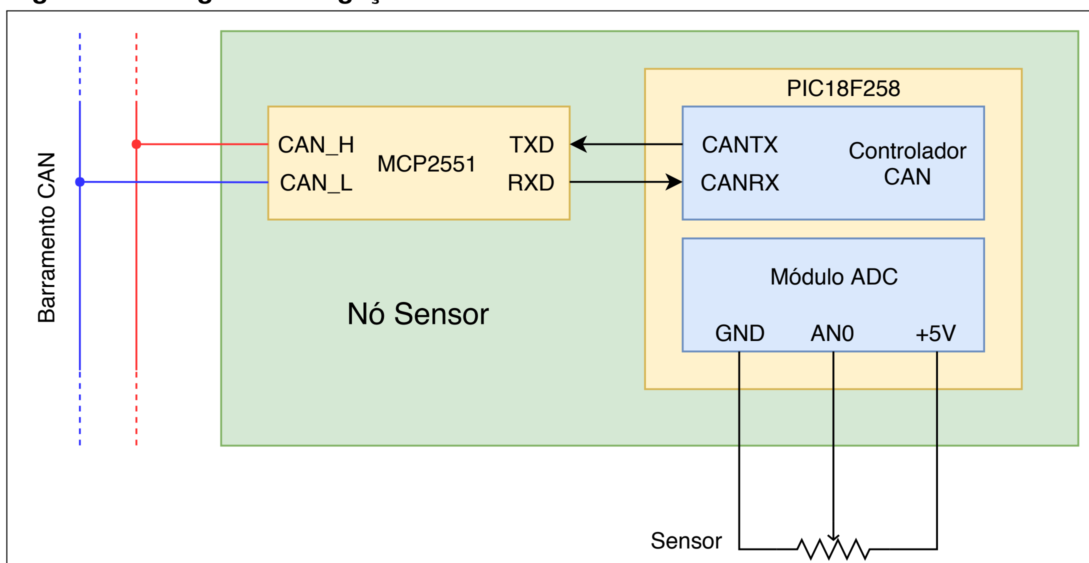


Fonte: Autoria própria.

3.1.1 Nó Sensor 1 e nó Sensor 2

Este nó deve possuir conexão com o barramento CAN para o envio de dados relacionados com a leitura de uma grandeza analógica proveniente de um potenciômetro. A figura 39 mostra o diagrama de ligação requeridos para o nó Sensor 1 e Sensor 2.

Figura 39 – Diagrama de ligação nó Sensor 1 e nó Sensor 2



Fonte: Autoria própria.

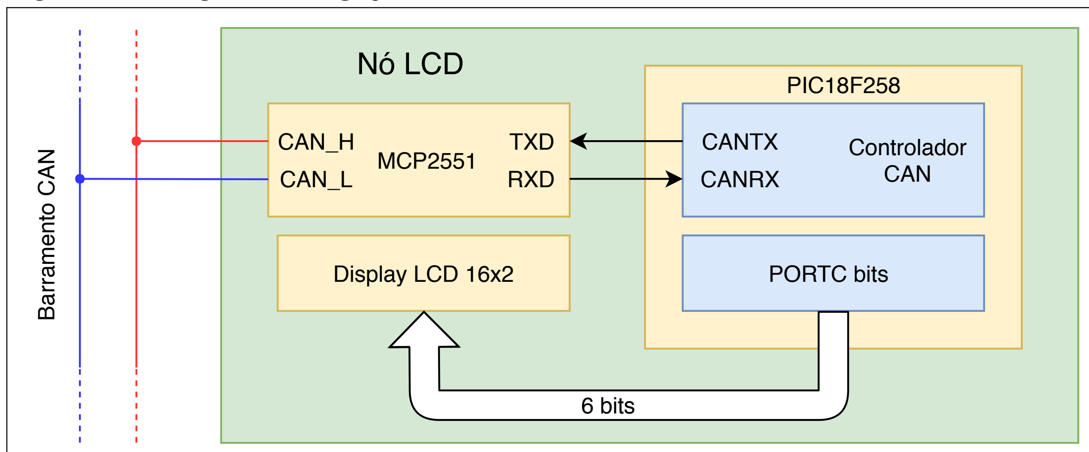
3.1.2 Nó LCD

A placa de circuito impresso desenvolvida para o nó LCD deve possuir conexão com o barramento CAN e apresentar de dados em um *display* LCD 16x2.

Visando utilizar uma menor quantidade de pinos do microcontrolador, optou-se por utilizar o modo de operação de 4 *bits* no *display*, sendo acrescido mais 2 *bits* para a habilitação e seleção de instrução do *display*.

A conexão entre o LCD e o microcontrolador é feita pelos pinos do PORTC, conforme mostrado na figura 40.

Figura 40 – Diagrama de ligação nó LCD

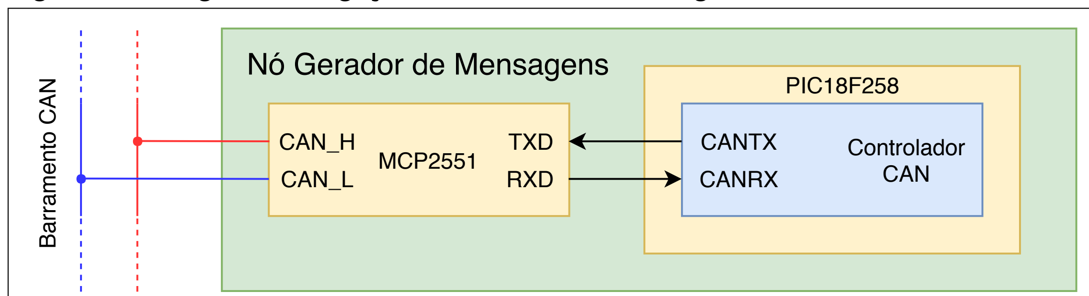


Fonte: Autoria própria.

3.1.3 Nó Gerador de Mensagens

O único pré-requisito necessário para o nó Gerador de Mensagens é possuir conexão com o barramento CAN, conforme mostra a figura 41.

Figura 41 – Diagrama de ligação nó Gerador de Mensagens

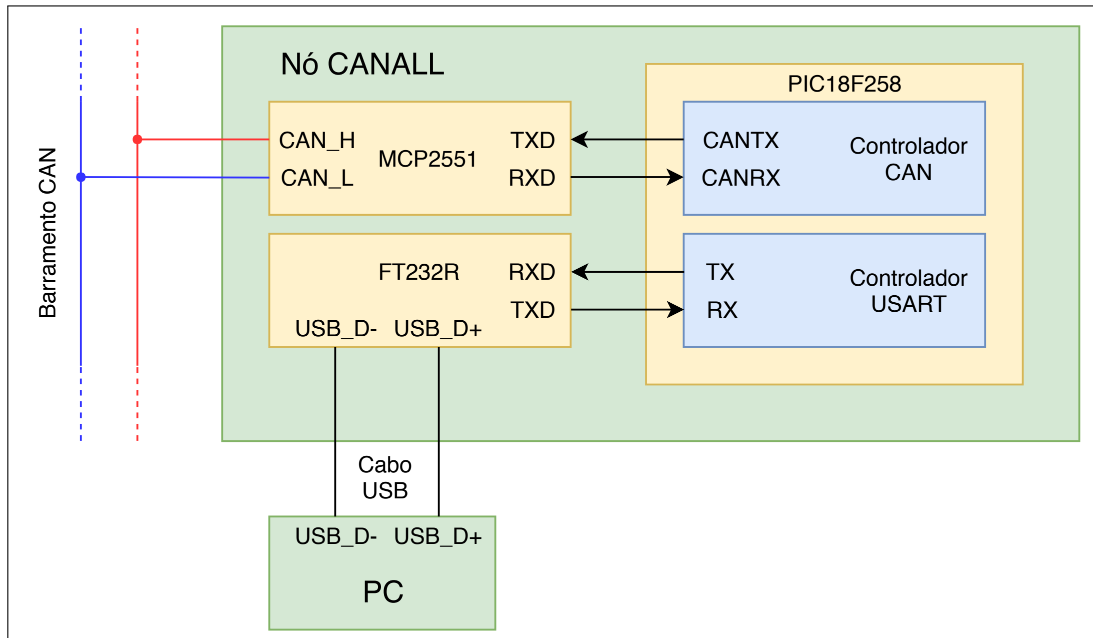


Fonte: Autoria própria.

3.1.4 Nó CANALL

A placa de circuito impresso desenvolvida para o nó CANALL deve possuir conexão com o barramento CAN e conexão USB com um computador pessoal, conforme mostrado na figura 42.

Figura 42 – Diagrama de ligação nó CANALL



Fonte: Autoria própria.

3.2 PROGRAMAÇÃO PIC18F258

3.2.1 Bibliotecas

Devido à grande popularidade dos microcontroladores PIC da família 18F, existe um vasto acervo de bibliotecas que podem ser utilizadas. Porém, para um melhor entendimento da programação em microcontroladores, optou-se por desenvolver todas as bibliotecas, tomando como base algumas já existentes.

Para a execução do projeto foram desenvolvidas 6 bibliotecas: adc.h, CAN.h, config.h, conversions.h, lcd.h e USART.h. Para a execução do projeto foram desenvolvidas 6 bibliotecas: adc.h, CAN.h, config.h, conversions.h, lcd.h e USART.h. Sua descrição de funcionamento está localizado nas próximas seções, enquanto que o código fonte das mesmas pode ser encontrado no apêndice deste documento.

3.2.1.1 Biblioteca adc.h

Esta biblioteca tem como função a conversão de um valor analógico, situado na faixa entre $0V$ e $5V$, para um valor digital, com resolução de 10 *bits*. Possui basicamente duas funções: inicialização e leitura.

3.2.1.1.1 Função adc_Init

A inicialização e configuração da conversão AD é feita por esta função, que não possui nenhum valor de parâmetro e retorno. Seu protótipo é mostrado abaixo.

Algoritmo 4 – Protótipo função adc_Init

```
void adc_Init(void);
```

Fonte: Autoria própria.

Na função adc_Init todo o PORTA é declarado como entrada digital fazendo todos os bits do registrador TRISA como 1. Isso evita que um nível de tensão seja fornecido do microcontrolador ao pino, evitando curto-circuitos.

Como a frequência de oscilação é de $10MHz$, a fonte de *clock* do conversor AD deve possuir período de oscilação de 16 vezes o período de oscilação do micro-

controlador ($16 \cdot T_{osc}$), o que resulta em $1{,}6\mu s$. É utilizada a equação 8 para isto.

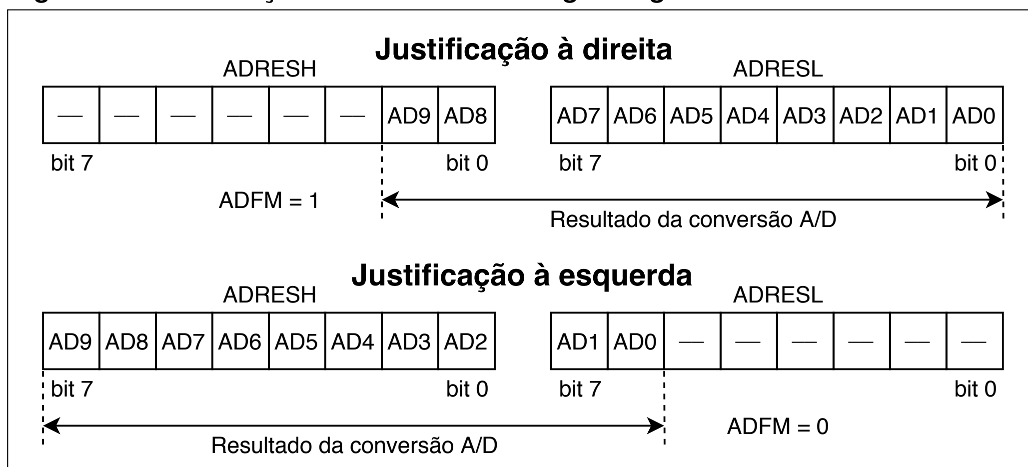
$$TAD = \frac{1}{10000000} \cdot 16 = 1{,}6\mu s$$

A seleção deste divisor (16) é feita com base na tabela 9 , logo, o valor de ADCS2:ADCS0 é 101. Com base nas seções anexas A.27 e A.28, ADCS2 é o sexto bit do registrador ADCON1, enquanto ADCS1 e ADCS0 são os *bits* mais significativos do registrador ADCON0.

É necessário ativar o módulo conversor A/D (ADON = 1), mas ainda não converter o valor analógico (GO_DONE = 0), pois este ainda está em fase de inicialização.

Como a resolução é de 10 *bits* mas os registradores comportam 8 *bits*, dois registradores são necessários para a conversão A/D. Tais registradores são o ADRESH e ADRESL. Eles tornam necessário a formatação ser à direita, pois assim a leitura será mais fácil, como será abordada na próxima seção. A formatação à direita é feita com ADFM = 1. A formatação à direita e à esquerda são mostradas na figura 43.

Figura 43 – Formatação da conversão analógico-digital

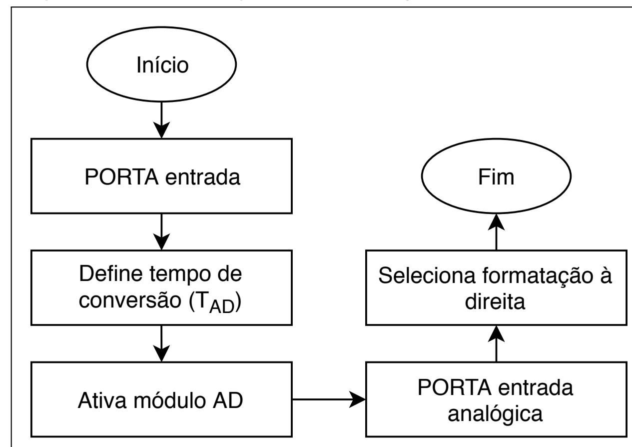


Fonte: Autoria própria.

Fazendo os bits PCFG3:PCFG0 serem iguais a 0000, temos uma referência de tensão positiva (Vref+) como o próprio VDD e a negativa (Vref-) como o GND. Também todos os pinos são entradas analógicas.

O fluxograma presente na figura 44 mostra o processo de inicialização desta função.

Figura 44 – Fluxograma da função adc_Init



Fonte: Autoria própria.

3.2.1.1.2 Função adc_Read

Esta função realiza a leitura do valor analógico situado na faixa de $0V$ até $5V$. Possui valor de retorno, de 16 *bits*, sendo que apenas 10 *bits* são utilizados. A seleção do canal de conversão utilizado é feita via parâmetro. Isto é mostrado no protótipo da função, mostrada abaixo.

Algoritmo 5 – Protótipo função adc_Read

```
unsigned int adc_Read(char channel);
```

Fonte: Autoria própria.

Os canais analógicos AN5, AN6 e AN7 não são implementados no PIC18F258, portanto existem cinco canais utilizáveis (AN4:AN0) para a conversão A/D. O canal utilizado é selecionado com o parâmetro, em *char*, da função adc_Read. A escolha do canal é dada pela seguinte expressão booleana:

Algoritmo 6 – Seleção do canal analógico

```
ADCON0bits.CHS = channel & 0x07;
```

Fonte: Autoria própria.

Após a escolha do canal, é iniciada a conversão A/D (GO_DONE = 1). Quando a conversão termina, o valor de GO_DONE passa automaticamente a ser 0. Sabendo disto, para aguardar a conversão ocorrer, uma instrução *while* é utilizada.

O retorno da função adc_Read é um número inteiro, proveniente da operação booleana dos registradores ADRESH e ADRESL, como mostrado abaixo.

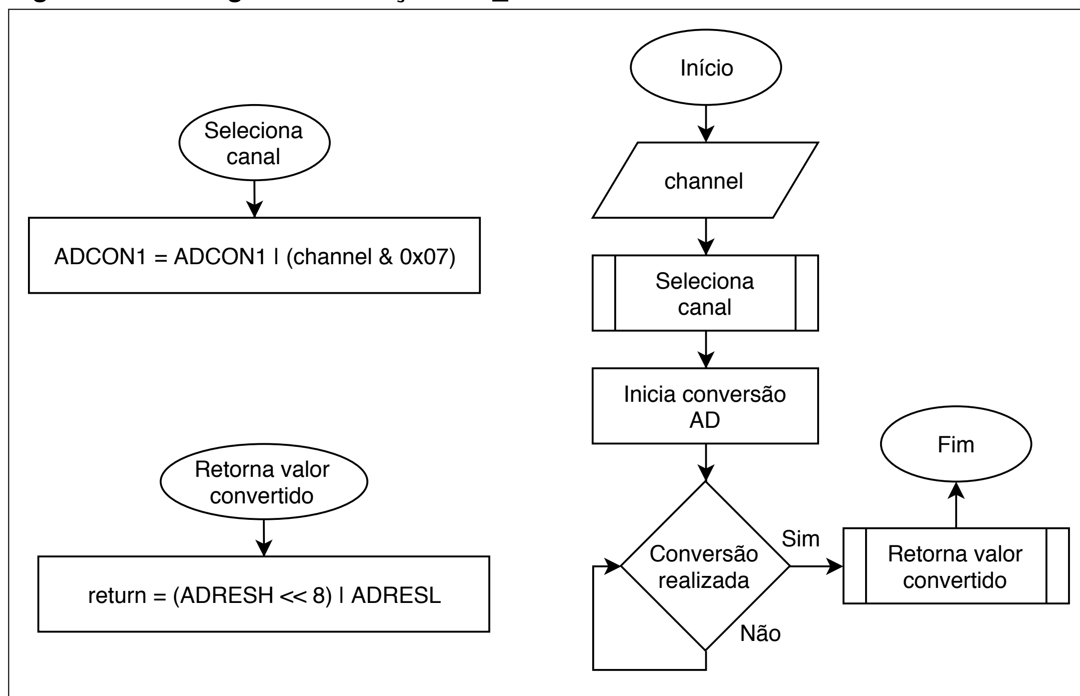
Algoritmo 7 – Retorno da função adc_Read

```
return (ADRESH << 8) | ADRESL;
```

Fonte: Autoria própria.

Como dito na seção anterior, uma formatação à direita torna a conversão de 10 *bits* mais fácil. Isto porque o *bit* menos significativo da conversão AD coincide com o *bit* menos significativo do registrador ADRESL, tornando necessária a rotação à esquerda do registrador ADRESH, em oito unidades.

Figura 45 – Fluxograma da função adc_Read



Fonte: Autoria própria.

3.2.1.2 Biblioteca config.h

Uma das funcionalidades do ambiente de desenvolvimento MPLAB X é a geração das diretivas de pré-processamento. Elas são geradas de acordo com uma tabela dinâmica, presente no *software*, que permite a rápida configuração destes *bits*. Estas configurações estão apresentadas na tabela 20. É também nesta biblioteca que a frequência de oscilação é definida.

Tabela 20 – Bits de configuração

Nome	Descrição
OSC = HS	Seleção de oscilador externo com alta velocidade
OSCS = OFF	Seleciona o oscilador principal como fonte
PWRT = ON	Habilita o *power-up timer*
BOR = ON	Habilita o *brown-out*
BORV = 25	Define a tensão de *brown-out* em $2{,}5V$
WTD = OFF	Desabilita o *watchdog timer*
STVR = ON	Habilita a reinicialização com o estouro de pilha
LVP = OFF	Desabilita a gravação com baixa tensão

Fonte: Autoria própria.

A ativação do *power-up timer* faz com que o microcontrolador aguarde alguns ciclos de máquina durante a fase de energização, garantindo que todos os seus periféricos estejam operantes quando o processamento estiver sendo executado.

O *brown-out* monitora a diferença de tensão entre os pinos VDD e VSS. Quando tal valor é inferior ao estipulado, ocorre a reinicialização do microcontrolador. A configuração estipulada nesta biblioteca define este valor em $2{,}5V$.

O *watchdog* é um recurso de detecção de erros, onde um temporizador (*timer*) é incrementado de acordo com um divisor escolhido. O valor deste temporizador deve ser zerado antes de atingir o tempo de estouro. Esta desabilitado o *watchdog timer*, visto que o mesmo deverá ser avaliado a cada período de tempo, o que pode causar erros nas comunicações USART e CAN.

3.2.1.3 Biblioteca CAN.h

A biblioteca CAN.h tem por finalidade a configuração da taxa de transmissão, modo de operação, leitura de dados, envio de dados e filtragem de mensagens do protocolo de comunicação CAN.

Possui sete funções: Can_Set_mode, Can_Init, Can_Set_Mask, Can_Set_-Filter, Can_Set_ID, Can_Write e Can_Read.

3.2.1.3.1 Função Can_Set_Mode

A presente função permite a seleção do modo de operação do módulo CAN. As opções disponíveis são: Modo de configuração, modo de escuta, modo remoto, modo desabilitado e modo normal. É passado por parâmetro um valor de 8 *bits*, sem

sinal (*unsigned char*), que corresponde ao modo de operação. Não há retorno nesta função. O protótipo da função é mostrado no algoritmo 8

Algoritmo 8 – Protótipo função Can_Set_Mode

```
void Can_Set_Mode(unsigned char mode);
```

Fonte: Autoria própria.

Como um valor numérico é passado por parâmetro à função, um correspondente na forma textual é melhor interpretado pelo programador que utiliza esta biblioteca. Esta correspondência é obtida pela diretiva #define, conforme mostrado na tabela 21.

Tabela 21 – Constantes do modo de operação

Modo de operação	Nome da constante	Valor da constante
Modo normal	NORMAL_MODE	0
Modo desabilitado	DISABLE_MODE	1
Modo remoto	LOOPBACK_MODE	2
Modo de escuta	LISTEN_ONLY_MODE	3
Modo de configuração	CONFIG_MODE	4

Fonte: Autoria própria.

Para que a seleção do modo de operação ocorra, primeiramente é necessário que nenhuma transmissão esteja ocorrendo, tornando necessário abortar todas as possíveis transmissões em andamento (CANCONbits.ABAT=1).

Então, deve ser feita a requisição do modo de operação ao módulo CAN. Tal requisição é feita pelos *bits* REQOP2:REQOP0, do registrador CANCON. Esses *bits* recebem o valor do parâmetro da função, conforme a seguinte expressão lógica mostrada no algoritmo 9.

Algoritmo 9 – Requisição do modo de operação do módulo CAN

```
CANCONbits.REQOP2 = (0x04 & mode) >> 2;
CANCONbits.REQOP1 = (0x02 & mode) >> 1;
CANCONbits.REQOP0 = 0x01 & mode;
```

Fonte: Autoria própria.

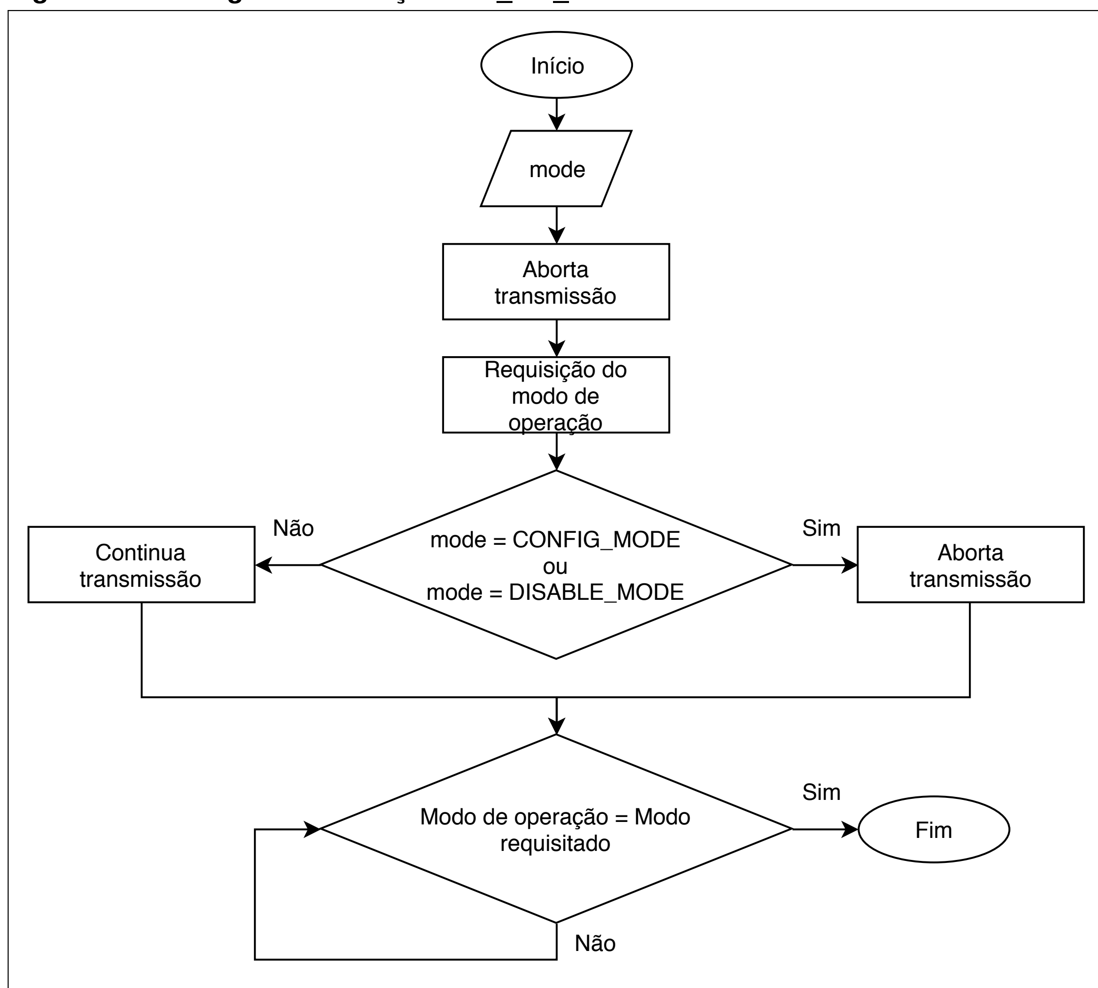
É feita uma comparação, com o comando *if*, afim de saber se o modo requerido é o modo de configuração ou modo desabilitado. Se verdadeiro, é necessário

abortar todas as transmissões (CANCONbits.ABAT=1). Caso for falso, as transmissões ocorrerão normalmente (CANCONbits.ABAT=0).

Como garantia que o modo requerido é o presente no módulo CAN, os *bits* OPMODE2:OPMODE0, do registrador CANSTAT, são checados. É necessário aguardar que o modo seja selecionado no módulo. A função somente é encerrada quando o módulo estiver com o modo de operação requerido.

O fluxograma presente na figura 46 mostra o funcionamento da função Can_Set_Mode.

Figura 46 – Fluxograma da função Can_Set_Mode



Fonte: Autoria própria.

3.2.1.3.2 Função Can_Init

A função Can_Init é responsável pela configuração da temporização da comunicação CAN, predefinições de filtros e máscaras, além das definições de prioridade

nos *buffers* de transmissão e recepção. É dotada de cinco parâmetros, que estão intimamente ligados com a temporização, como mostrado em seu protótipo (algoritmo 10).

Algoritmo 10 – Protótipo função Can_Init

```
void Can_Init(char sjw, char brp, char propseg,
char phseg1, char phseg2);
```

Fonte: Autoria própria.

Os registradores BRGCON1, BRGCON2 e BRGCON3 recebem os valores dos parâmetros da função, porém, não diretamente. É necessária uma manipulação matemática para ajustar o seu valor aos registradores, conforme mostrado na equação abaixo (equação 13).

$$\begin{gathered} sjw = 64 \cdot (sjw - 1) \\ propseg = propseg - 1 \\ phseg1 = 8 \cdot (phseg1 - 1) \\ phseg2 = phseg2 - 1 \end{gathered} \tag{13}$$

Após esta correção, os registradores recebem os valores de acordo com a lógica mostrada no algoritmo 11.

Algoritmo 11 – Parametrização da temporização do módulo CAN

```
BRGCON1 = sjw | brp
BRGCON2 = phseg1 | propseg
BRGCON3 = phseg2
```

Fonte: Autoria própria.

É definido que o pino 23 (CANTX) é uma saída digital (TRISBbits.RB2 = 0), enquanto que o pino 24 (CANRX), uma entrada (TRISBbits.RB3 = 1). Também é desabilitada a interrupção relacionada à CAN (PIE3 = 0).

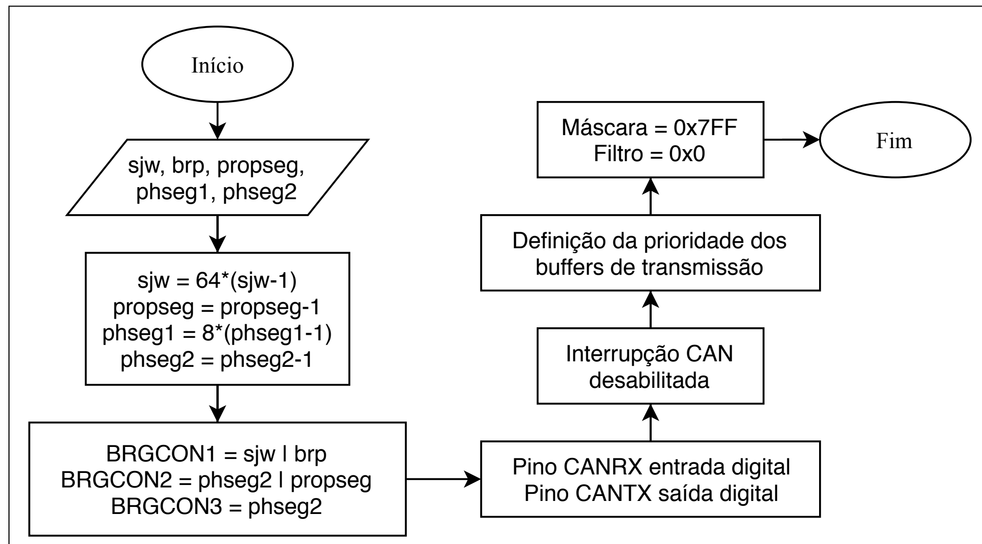
Com relação aos *buffers* de transmissão, é definido que o *buffer* 0 é o que possui maior prioridade, seguido pelo *buffer* 1 e pelo *buffer 2*, que possui a menor prioridade possível.

Nos *buffers* de recepção, é definido que somente mensagens válidas com identificador padrão são aceitas. Em ambos os *buffers* de recepção, as máscaras são

predefinidas com todos os *bits* em nível alto (0x7FF), enquanto os filtros possuem todos os *bits* em nível baixo (0x0), tornando necessária as funções Can_Set_Mask e Can_Set_Filter, apresentadas nas seguintes seções.

O fluxograma, presente na figura 47 mostra o funcionamento da função Can_-Init.

Figura 47 – Fluxograma da função Can_Init



Fonte: Autoria própria.

3.2.1.3.3 Função Can_Set_Mask

A presente função tem por finalidade a configuração das máscaras utilizadas no recebimento de mensagens. Esta função não possui valor de retorno, porém nela são passados dois valores por parâmetro. O primeiro parâmetro, de 8 *bits*, é referente a seleção do *buffer*. O segundo, com 16 *bits*, é referente a máscara desejada. O protótipo da função é mostrado no algoritmo 12

Algoritmo 12 – Protótipo da função Can_Set_Mask

```
void Can_Set_Mask(unsigned char rx_buffer, unsigned int mask);
```

Fonte: Autoria própria.

A atribuição da máscara ao *buffer* desejado é feita primeiramente com a estrutura condicional *if*, comparando o valor do *buffer* desejado (rx_buffer) com os dois *buffers* existentes (0 ou 1) no recebimento de mensagens.

Devido existência de dois *buffers* de recebimento, é possível que o valor do parâmetro mask possa ser atribuído aos registradores RXM0SIDH e RXM0SIDL, quando o *buffer* 0 é selecionado, ou para os registradores RXM1SIDH e RXM1SIDL, quando o *buffer 1* for o selecionado. Esses registradores recebem o valor do parâmetro mask de acordo com a seguinte expressão lógica abaixo (algoritmo 15, onde n é o buffer selecionado (0 ou 1).

Algoritmo 13 – Atribuição da máscara ao buffer n

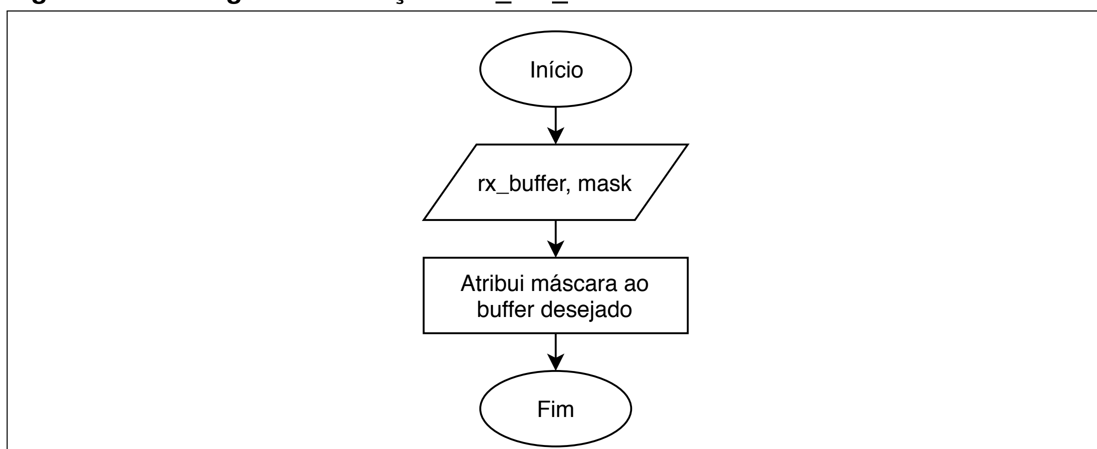
```
RXMnSIDH = (mask & 0x07F8) >> 3
RXMnSIDL = (mask & 0x0007) << 5
```

Fonte: Autoria própria.

A utilização desta expressão lógica é justificada pelo fato que a máscara possui 11 *bits*, enquanto cada registrador comporta 8 *bits*. Além disto, o *bit* menos significativo da máscara encontra-se no quinto *bit* do registrador RXMnSIDL, o que faz necessário a rotação dos *bits*.

Quando a atribuição aos registradores RXMnSIDH e RXMnSIDL é feita, a função é encerrada. O fluxograma presente na figura 48 mostra o funcionamento da função Can_Set_Mask.

Figura 48 – Fluxograma da função Can_Set_Mask



Fonte: Autoria própria.

3.2.1.3.4 *Função Can_Set_Filter*

A presente função tem como finalidade a configuração dos seis filtros utilizados no recebimento de mensagens. Esta função não possui valor de retorno, mas possui dois parâmetros: option e filter, como mostrado no protótipo no algoritmo 14.

Algoritmo 14 – Protótipo da função Can_Set_Filter

```
void Can_Set_Filter(unsigned char option, unsigned int filter);
```

Fonte: Autoria própria.

Assim como ocorre na função Can_Set_Mode, a diretiva #define é utilizada para a seleção de um dos seis filtros, conforme mostra a tabela 22.

Tabela 22 – Constantes da seleção do filtro

Filtro	Nome da constante	Valor da constante
Filtro 0, Buffer 0	F0_B0	0
Filtro 1, Buffer 0	F1_B0	1
Filtro 2, Buffer 1	F2_B1	2
Filtro 3, Buffer 1	F3_B1	3
Filtro 4, Buffer 1	F4_B1	4
Filtro 5, Buffer 1	F5_B1	5

Fonte: Autoria própria.

O parâmetro option recebe uma das seis possibilidades de filtros, onde seu valor é comparado em uma estrutura condicional *if*, afim de saber qual é o filtro selecionado.

Após a definição do qual é o filtro selecionado, os registradores RXFnSIDH e RXFnSIDL recebem o valor do parâmetro filter de acordo com a seguinte expressão lógica mostrada abaixo. Ela possui mesmo formato que a da função Can_Set_Mask, pela mesma justificativa.

Algoritmo 15 – Atribuição do filtro ao filtro n

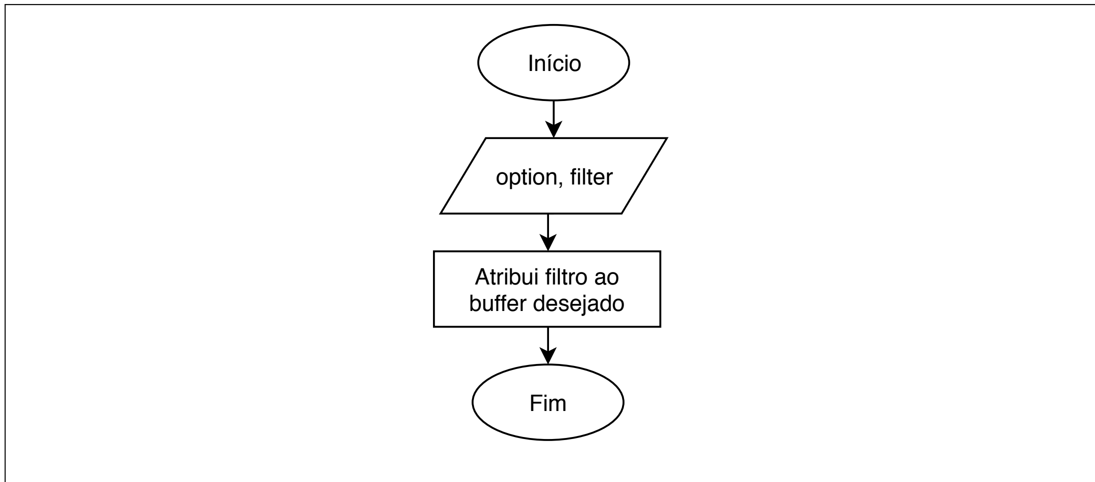
```
RXFnSIDH = (filter & 0x07F8) >> 3
RXFnSIDL = (filter & 0x0007) << 5
```

Fonte: Autoria própria.

Os valores de n, na expressão acima, podem variar de 0 até 5, que corresponde aos seis filtros de recebimento de mensagens.

Quando a atribuição aos registradores RXFnSIDH e RXFnSIDL é feita, a função é encerrada. O fluxograma presente na figura 49 mostra o funcionamento da função Can_Set_Filter.

Figura 49 – Fluxograma da função Can_Set_Filter



Fonte: Autoria própria.

3.2.1.3.5 Função Can_Set_Id

A presente função tem como finalidade a configuração dos identificadores presentes nos três diferentes *buffers* de envio. A função não possui nenhum valor de retorno, mas têm dois parâmetros: tx_buffer, id. O protótipo é mostrado no algoritmo 16.

Algoritmo 16 – Protótipo da função Can_Set_Id

```
void Can_Set_Id(unsigned char tx_buffer, unsigned int id);
```

Fonte: Autoria própria.

A atribuição do valor de filtro ao *buffer* desejado é feita com uma estrutura condicional (*if*), comparando o valor do *buffer* desejado (tx_buffer) com os três *buffers* existentes (0, 1 e 2) destinados ao envio de mensagens.

De forma semelhante ao que ocorre na função Can_Set_Mask, existem diferentes registradores para a configuração do identificador (ID). Com isso há os registradores do *buffer* 0 (TXB0SIDH e TXB0SIDL), *buffer* 1 (TXB1SIDH e TXB1SIDL) e *buffer 2* (TXB2SIDH e TXB2SIDL).

Os registradores TXBnSIDH e TXBnSIDL recebem o valor do parâmetro id de acordo com a seguinte expressão lógica (algoritmo 17), já utilizada e justificada nas seções acima.

Algoritmo 17 – Atribuição do id ao buffer n

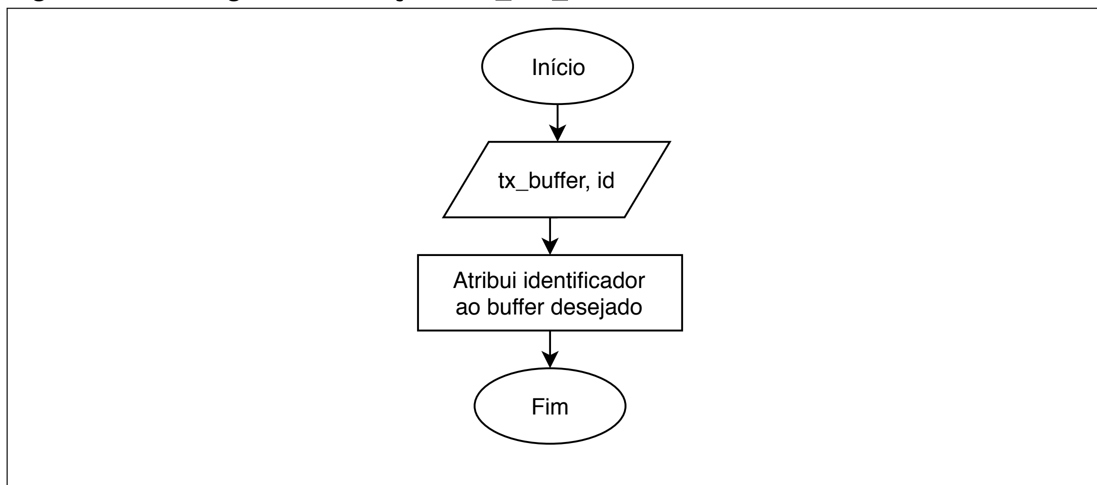
```
TXBnSIDH = (id & 0x07F8) >> 3
TXBnSIDL = (id & 0x0007) << 5
```

Fonte: Autoria própria.

Sobre os valores de n, na expressão acima, podem variar de 0 até 3, que corresponde aos seis três *buffers* de envio de mensagens.

Logo após, é necessário informar ao módulo CAN que o formato de identificador utilizado é o padrão (*standard*), ou seja, de 11 *bits*. O *bit* EXIDE, do registrador TXBnSIDL deve possuir nível lógico zero para isto. Após esta atribuição a função é encerrada. Na figura 50 é apresentado o fluxograma pertinente à função Can_Set_Id.

Figura 50 – Fluxograma da função Can_Set_Id



Fonte: Autoria própria.

3.2.1.3.6 Função Can_Write

A função Can_Write é a responsável pelo envio de dados para os nós do barramento CAN. Ela possui três parâmetros, sendo dois deles passados por valor (tx_buffer e tx_dlc) e um por referência (*data). O protótipo da função é mostrado abaixo (algoritmo 18.

Algoritmo 18 – Protótipo da função Can_Write

```
void Can_Write(char tx_buffer, char tx_dlc, char *data);
```

Fonte: Autoria própria.

A definição de qual dos *buffers* é utilizado no envio da mensagem é dada pelo parâmetro tx_dlc, que pode receber valores de 0 até 2, representando os *buffers* 0, 1 e 2. Este valor é comparado pela estrutura condicional *if*, que faz a seleção do *buffer* desejado.

Com a seleção do *buffer*, é necessário fornecer ao módulo CAN o número de *bytes* de dados que serão enviados. Isto é feito com o valor presente no parâmetro tx_-dlc, que admite valores entre 0 e 8, correspondendo ao número de *bytes* de dados. A atribuição deste valor ao registrador é feita com a seguinte expressão lógica (algoritmo 19), tendo em base a tabela 3, já mostrada no corpo deste documento. O valor de n pode variar entre 0 e 2, representando os 3 *buffers*.

Algoritmo 19 – Número de dados enviados

```
TXBnDLC = tx_dlc & 0x0F;
```

Fonte: Autoria própria.

A passagem por referência desta função é um vetor com 8 posições (*data), cada posição com 8 *bits*, totalizando 64 *bits*. Isto compreende o número máximo de *bits* transmitidos no protocolo CAN 2.0B (BOSCH et al., 1991). O valor deste vetor referenciado é passado ao campo de dados, conforme a lógica presente no algoritmo 20.

Com esta metodologia torna-se mais fácil o envio de dados, pois os registradores possuem tamanho de 8 *bits*, assim como o vetor referenciado. Outro ponto a favor deste método é que o *bit* menos significativo do vetor corresponde ao *bit* menos significativo do campo de dados, evitando o deslocamento de *bits*.

Após atribuir a mensagem ao campo de dados, é necessária uma requisição para o envio dos dados (TXBnCONbits.TXREQ = 1). Com a requisição feita, é necessária que a mesma seja executada. Este controle é feito verificando se o *bit* TXREQ, do registrador TXBnCON tornou-se zero. A função só termina quando a mensagem é

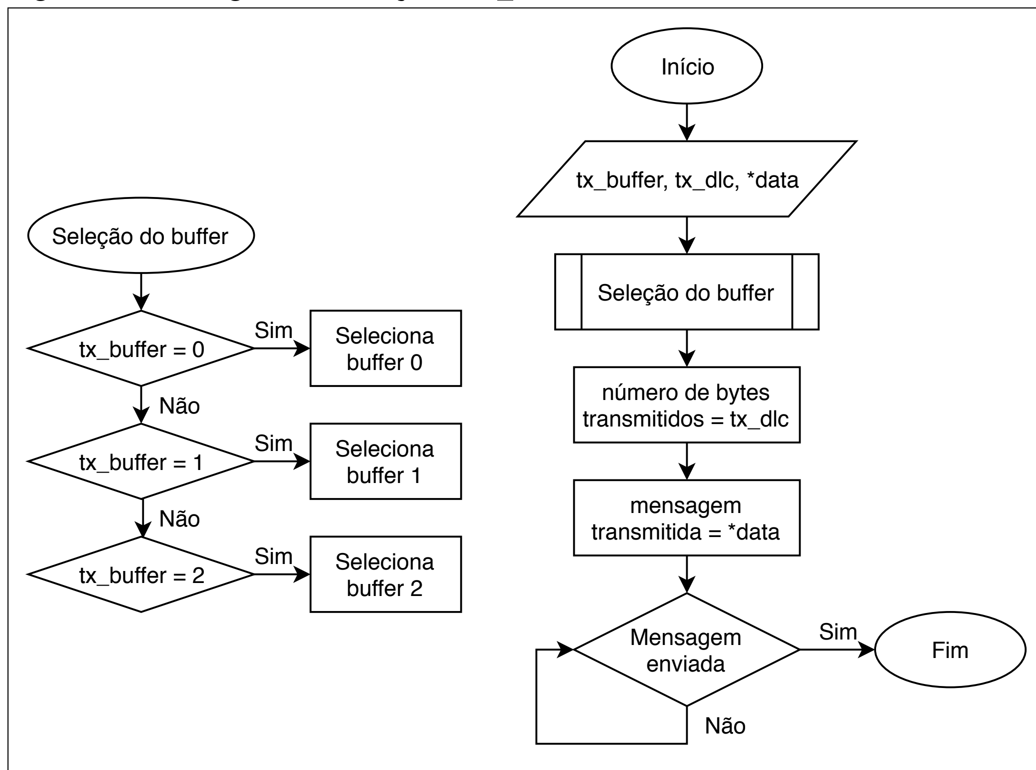
Algoritmo 20 – Atribuição do vetor data para o registrador de envio

```
TXBnD7 = data[7];
TXBnD6 = data[6];
TXBnD5 = data[5];
TXBnD4 = data[4];
TXBnD3 = data[3];
TXBnD2 = data[2];
TXBnD1 = data[1];
TXBnD0 = data[0];
```

Fonte: Autoria própria.

enviada com sucesso, ou seja, quando TXREQ = 0. A figura 51 mostra o fluxograma desta função.

Figura 51 – Fluxograma da função Can_Write



Fonte: Autoria própria.

3.2.1.3.7 Função Can_Read

A função Can_Read é a responsável pela leitura dos dados provenientes do barramento CAN. É composta de quatro parâmetros, sendo todos passados por referência, conforme mostrado no protótipo abaixo (figura 21).

Algoritmo 21 – Protótipo da função Can_Read

```
void Can_Read(char *data, int rx_id,
int *dlc, char *error_count);
```

Fonte: Autoria própria.

Como existem dois *buffers* destinados à recepção de mensagens, é necessário saber qual o *buffer* está em operação no momento de aquisição de dados. Isto é feito com a estrutura de condicional *if*.

A partir deste momento é informado ao módulo CAN que mensagens com o formato padrão de identificadores serão recebidas (RXBnSIDLbits.EXID = 0). O valor de identificador recebido é armazenado na posição de memória que é passada pelo parâmetro da função. Esta atribuição é feita com base na expressão abaixo (algoritimo 22), onde id_high e id_low são variáveis locais da função.

Algoritmo 22 – Lógica de recepção do identificador

```
id_high = (RXBnSIDH << 3) & 0x7F8
id_low = (RXBnSIDL >> 5) & 0x0007
*rx_id = id_high | id_low
```

Fonte: Autoria própria.

Os dados são atribuídos ao vetor *data, onde cada posição deste vetor ocupa 8 *bits*. Uma leitura de até 8 *bytes* pode ser executada. Esta atribuição é mostrada no algoritmo 23.

Algoritmo 23 – Atribuição do registrador de recebimento para o vetor data

```
data[7] = RXBnD7 ;
data[6] = RXBnD6 ;
data[5] = RXBnD5 ;
data[4] = RXBnD4 ;
data[3] = RXBnD3 ;
data[2] = RXBnD2 ;
data[1] = RXBnD1 ;
data[0] = RXBnD0 ;
```

Fonte: Autoria própria.

O número de *bytes* recebidos é armazenado no parâmetro *dlc, que recebe tal valor conforme a expressão lógica mostrada no algoritmo 24.

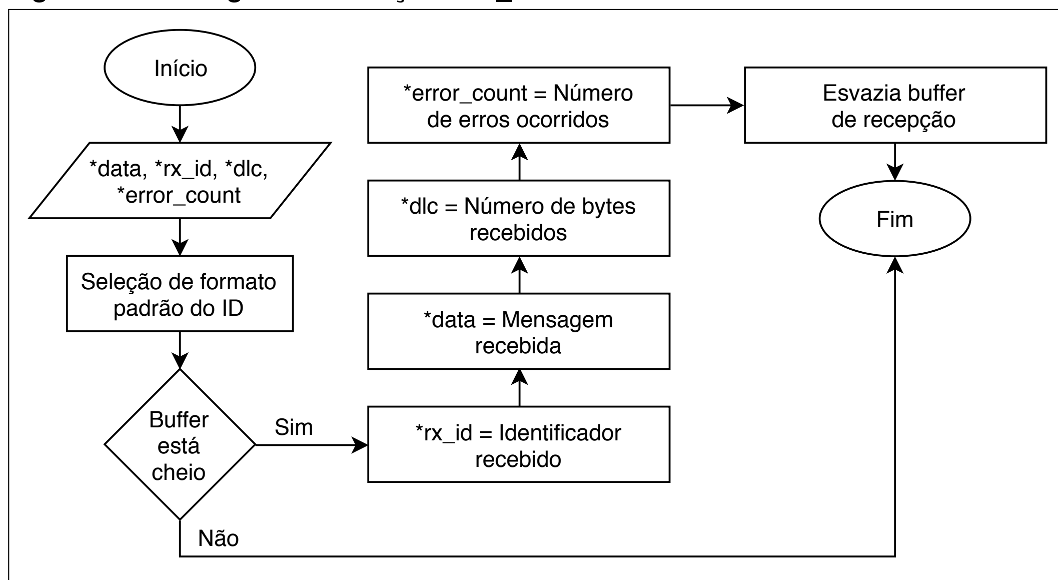
Algoritmo 24 – Número de dados recebidos

```
*dlc =RXBnDLC & 0x0F;
```

Fonte: Autoria própria.

A definição do estado da comunicação (erro ativo e erro passivo) é fornecido ao parâmetro *error_count, que recebe o valor do registrador RXERRCNT. O fluxograma desta função é apresentado na figura 52.

Figura 52 – Fluxograma da função Can_Read



Fonte: Autoria própria.

3.2.1.4 Biblioteca conversions.h

A biblioteca conversions.h tem por finalidade a conversão de valores inteiros para vetores de caracteres, vetor de caracteres para números inteiros e separação e a quebra de um vetor de caracteres em dois vetores, a partir de um caractere separador. É baseada na biblioteca stdlib.h.

3.2.1.4.1 Função toCharArray

A função toCharArray é baseada na função itoa, da biblioteca stdlib.h. Possui três parâmetros, sendo um passado por referência (*numArray) e dois passados por valor (num e base). Seu protótipo é mostrado no algoritmo 25.

Algoritmo 25 – Protótipo da função toCharArray

```
void toCharArray(char *numArray, long num, int base);
```

Fonte: Autoria própria.

Esta função é destinada à conversão números inteiros, de até 32 *bits*, em um vetor de caracteres, de 8 *bits* cada, padronizados pela tabela ASCII.

É possível a conversão em diversos valores de bases, como binária, octal, decimal e hexadecimal. A escolha da base é proveniente do parâmetro base, fornecido pelo usuário desta função.

O número a ser convertido é fornecido à função pelo parâmetro num. Este valor é atribuído à variável local value, que receberá todas as expressões matemáticas. Há também a definição da varável local rem, de 32 *bits*, que tem por finalidade armazenar o resto da divisão.

Se o valor da variável value for nulo, o vetor de caracteres atribui o caractere zero na primeira posição do vetor, enquanto a segunda posição recebe o caractere nulo, indicando o fim do vetor. Se isto ocorre, a função é encerrada.

Quando o valor a ser convertido é diferente de zero, é necessário um algoritmo para a conversão, como mostrado na figura 53.

De acordo com a tabela ASCII, o caractere ‘0’ possui valor decimal 48, enquanto o caractere ‘A’ possui valor decimal 65. Exemplificando, um resto de divisão com valor 5 é somado com o 48, fornecendo um valor decimal de 53, que corresponde ao caractere ‘5’ na tabela ASCII. O mesmo é válido quando o resto é maior que 9, mas ao invés de ser somado 48, é somado 65.

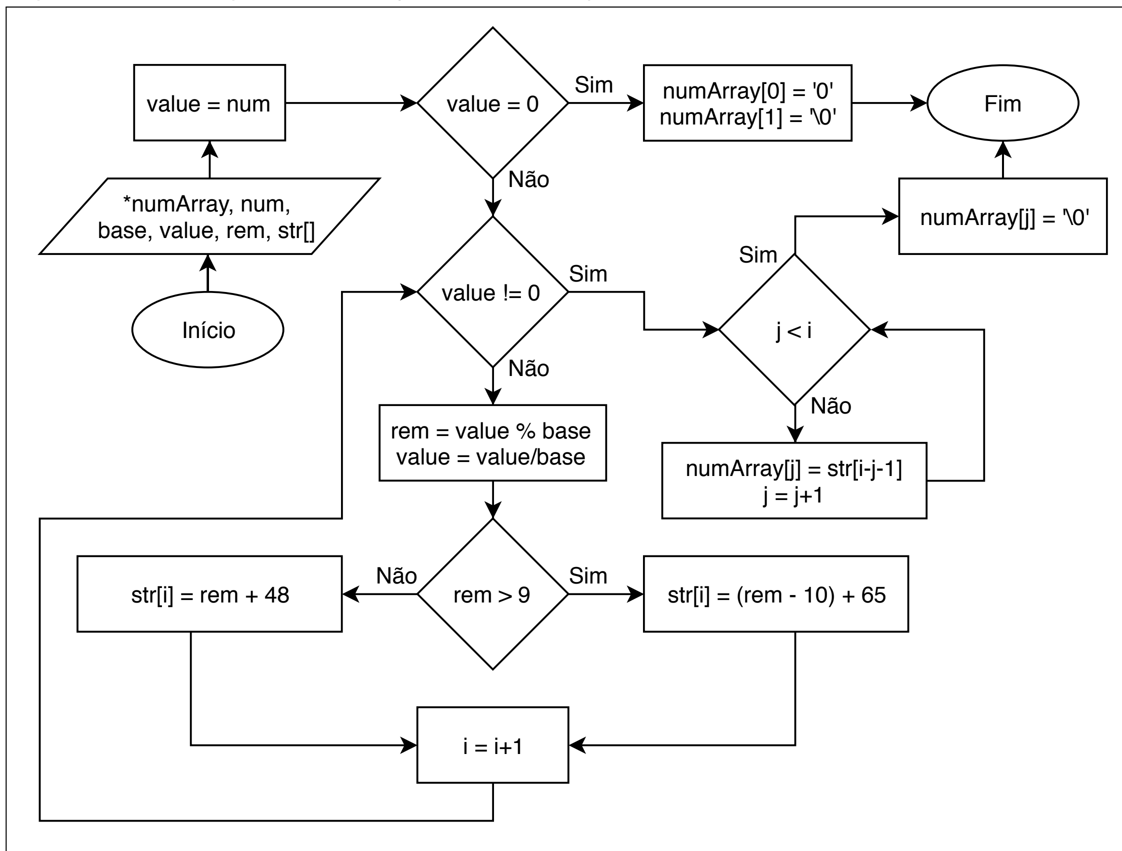
É importante notar que o resto só é maior que 9 quando a conversão é em uma base maior que 10, como na hexadecimal, o que gera a presença de letras.

Esta operação é repetida sucessivamente, até que não haja valor à ser dividido. No final desta operação, todo o vetor deve ser invertido, finalizando a função.

3.2.1.4.2 Função toCharArrayFixedDigits

A presente função é semelhante à função acima (toCharArray), porém, nela o número de dígitos presentes no vetor de caracteres é fixo. Isto significa que é possível

Figura 53 – Fluxograma da função toCharArray



Fonte: Autoria própria.

a representação do número 0001, algo impossível na função toCharArray. Apresenta quatro parâmetros e não possui valor de retorno, como mostrado em seu protótipo (algoritmo 26.

Algoritmo 26 – Protótipo da função toCharArrayFixedDigits

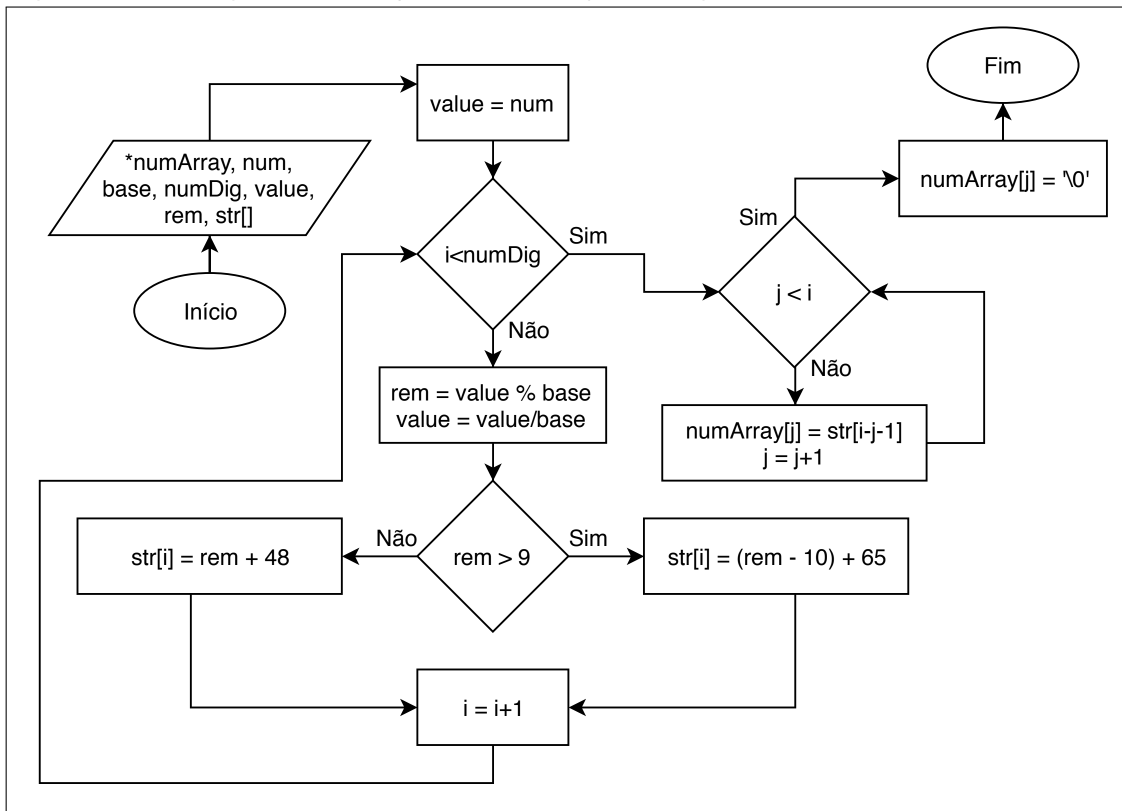
```
void toCharArray(char *numArray, long num,
int base, int numDig);
```

Fonte: Autoria própria.

Ao invés da divisão ocorrer sucessivamente até que não haja valor à ser dividido, nesta função a divisão ocorre de acordo com o número de dígitos, conforme passado por parâmetro (numDig). Se o número de dígitos é 3, ocorrerão 3 divisões sucessivas. O fluxograma desta função é apresentado na figura 54.

É necessário que o vetor de caracteres seja invertido, assim como ocorre na função toCharArray. Quando o número de divisões sucessivas é igual ao número de dígitos a função é encerrada.

Figura 54 – Fluxograma da função toCharArrayFixedDigits



Fonte: Autoria própria.

Este formato de conversão, com o número de dígitos fixos é muito utilizado em *displays* LCD, pois caso seja utilizado o formato de dígitos variáveis, o caractere de maior magnitude irá apresentar um valor diferente de zero, mesmo que o mesmo seja zero, conforme observado em testes práticos desta função.

3.2.1.4.3 Função strintToInt

A função relatada nesta seção tem a finalidade de converter um vetor de caracteres em um número inteiro de 16 *bits*. Possui dois parâmetros e um valor de retorno, conforme mostra o algoritmo 27.

Algoritmo 27 – Protótipo da função stringToInt

```
unsigned int stringToInt(char *numArray, int base);
```

Fonte: Autoria própria.

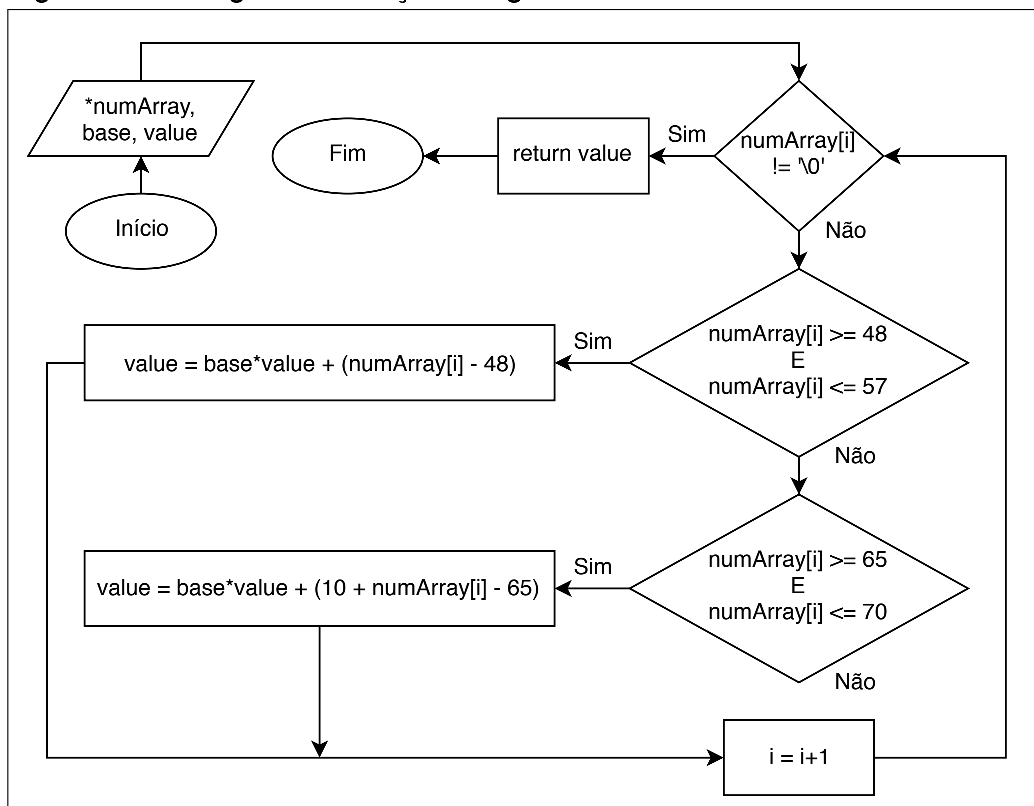
Com a estrutura de repetição *for*, é possível percorrer todo o vetor de caracteres, desde sua primeira posição (0) até a última, que contém o caractere nulo ($' \backslash 0'$).

A alteração da posição do vetor é dada pelo incremento da variável i.

Com base na tabela ASCII, o decimal 48 corresponde ao caractere ‘0’ e o decimal 57 corresponde ao caractere ‘9’. Quando o decimal está entre os valores 65 e 70, os caracteres entre ‘A’ e ‘F’ são os envolvidos.

Após o laço de repetição percorrer todo o vetor de caracteres, a função retorna o valor convertido armazenado na variável value, encerrando assim a função, conforme mostra a figura 55.

Figura 55 – Fluxograma da função stringToInt



Fonte: Autoria própria.

3.2.1.4.4 Função split

A presente função tem por finalidade separar um vetor de caracteres em dois números. Isto é feito através de um caractere delimitador (‘#’), que deve estar presente no vetor de caracteres. Esta função possui três parâmetros, todos passados por referência, como mostra seu protótipo (algoritmo 28).

O vetor a ser separado em dois números é proveniente do parâmetro *str. Com o auxílio de uma estrutura de repetição, todo o vetor é percorrido, onde a posição que

Algoritmo 28 – Protótipo da função split

```
void split(char *str, int num1, int num2);
```

Fonte: Autoria própria.

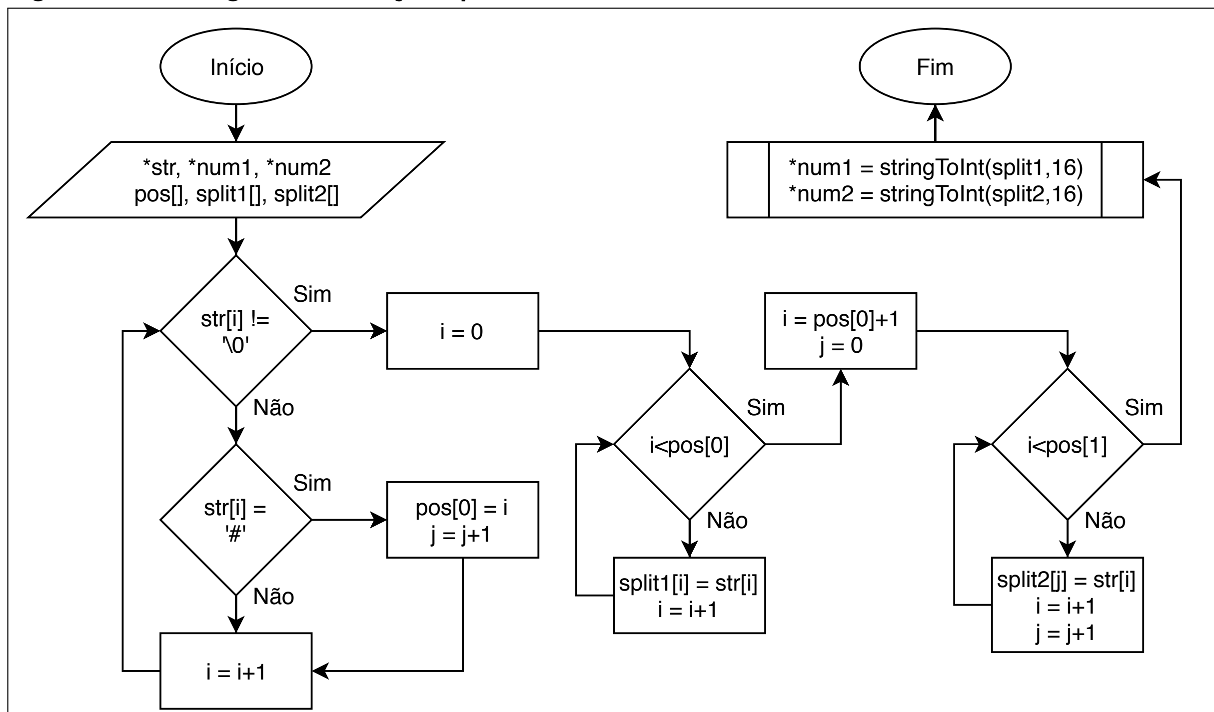
o caractere delimitador se encontra é armazenada na posição zero do vetor pos[].

Sabendo qual a posição do vetor a ser separado se encontra o caractere delimitador, é possível separar este vetor em dois vetores, onde um vai da posição zero até a posição anterior ao caractere delimitador, e o outro, que inicia na posição posterior ao caractere delimitador e vai até o caractere nulo. Estes dois vetores são denominados split1 e split2.

Os vetores são inseridos como parâmetro da função stringToInt, desta mesma biblioteca, já abordada neste documento (seção 3.2.1.4.3). Com isto, é a separação dos números é concluída.

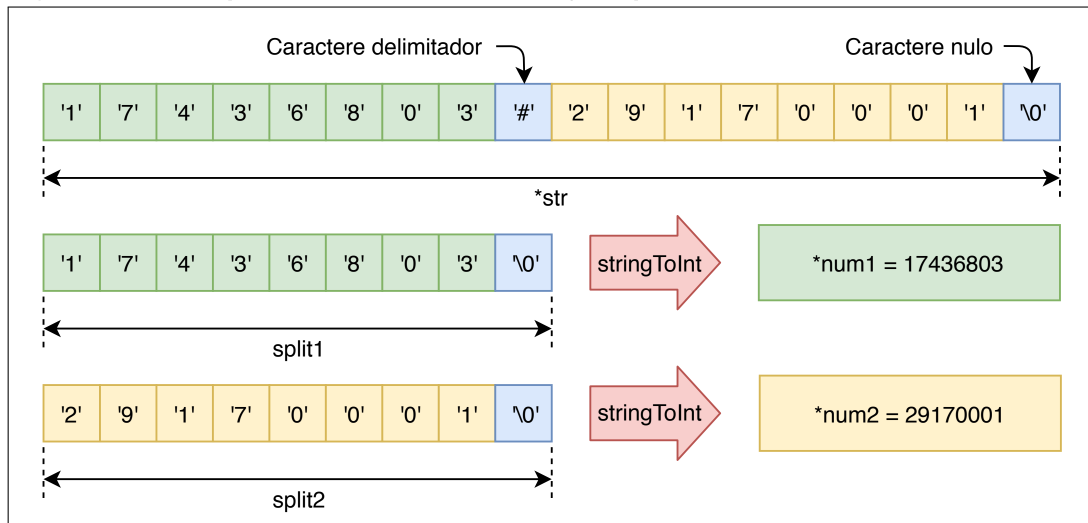
O fluxograma presente na figura 56 mostra o funcionamento da função split, bem como a figura 57 mostra um exemplo de separação, onde os dois números (17436803 e 29170001) são divididos pelo caractere delimitador (‘#’). Em ambas imagens, é possível notar a presença da função stringToInt.

Figura 56 – Fluxograma da função split



Fonte: Autoria própria.

Figura 57 – Exemplo de funcionidade da função split



Fonte: Autoria própria.

3.2.1.5 Biblioteca lcd.h

A biblioteca lcd.h é destinada à inicialização, apresentação e posicionamento de caracteres no *display* LCD. Nesta biblioteca existem 7 funções, que serão descritas nas seções seguintes.

Afim de utilizar uma menor quantidade de pinos no microcontrolador, a opção de 4 *bits* de dados é utilizada. As diretivas #define, mostradas na tabela 23, representam a correspondência entre os pinos do *display* e do microcontrolador.

Tabela 23 – Correspondência entre pinos do display e pinos do microcontrolador

Nome da constante	Porta digital
RS	RC5
EN	RC4
D4	RC3
D5	RC2
D6	RC1
D7	RC0

Fonte: Autoria própria.

Esta biblioteca é compatível com vários tamanhos de *display* LCD, que vão de $8x1$ até $40x4$.

3.2.1.5.1 Função Send_Nibble

A função é responsável pelo envio de 4 *bits* (1 *nibble*) de dados para o *display* LCD. Seu protótipo de função é mostrado abaixo (algoritmo 29).

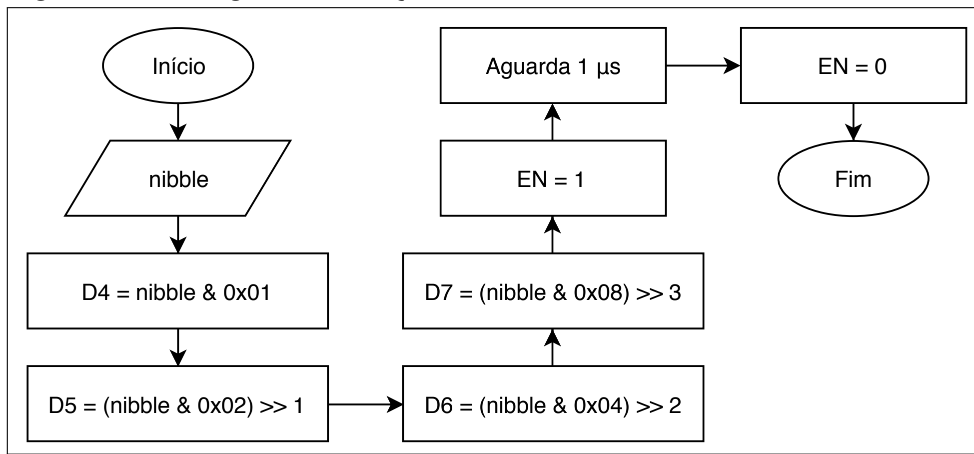
Algoritmo 29 – Protótipo da função Send_Nibble

```
void Send_Nibble(char nibble);
```

Fonte: Autoria própria.

Os 4 *bits* de dados são enviados de forma paralela ao microcontrolador, pelos pinos definidos na tabela 23. O fluxograma desta função é mostrado na figura 58.

Figura 58 – Fluxograma da funçãoSend_Nibble



Fonte: Autoria própria.

3.2.1.5.2 Função Send_Byte

Esta função tem como finalidade o envio de um *byte* de dados para o *display*. Tal envio é feito enviando os 4 *bits* mais significativos primeiro, sendo separados dos menos significativos pela transição de nível lógico do pino Enable (EN). Seu protótipo é mostrado no algoritmo 30.

Algoritmo 30 – Protótipo da função Send_Byte

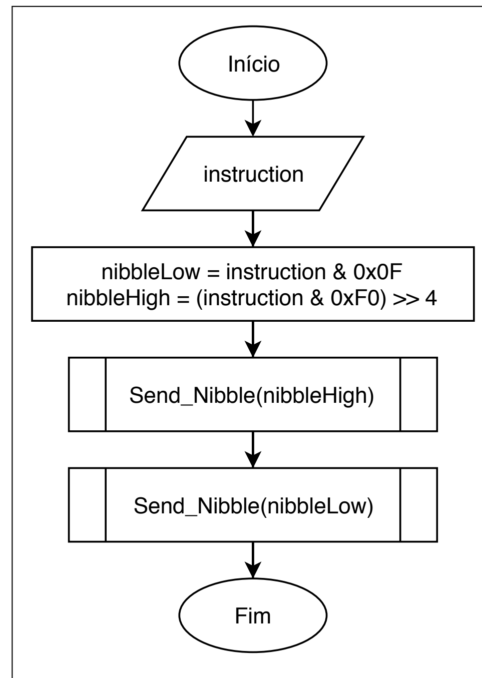
```
void Send_Byte(char instruction);
```

Fonte: Autoria própria.

Os 4 *bits* mais significativos são denominados nibbleHigh, enquanto os menos significativos são nomeados nibbleLow. Os níveis lógicos das variáveis acima são atribuídos aos pinos do microcontrolador de acordo com a tabela 23. O fluxograma

desta função é mostrada na figura 59. É possível observar que esta função emprega a função Send_Nibble.

Figura 59 – Fluxograma da função Send_Byte



Fonte: Autoria própria.

3.2.1.5.3 Função Lcd_Init

A função relatada nesta seção é responsável pela configuração e inicialização do *display* LCD. Não possui nenhum valor de retorno, bem como nenhum parâmetro, conforme mostrado no algoritmo 31.

Algoritmo 31 – Protótipo da função Lcd_Init

```
void Lcd_Init(void);
```

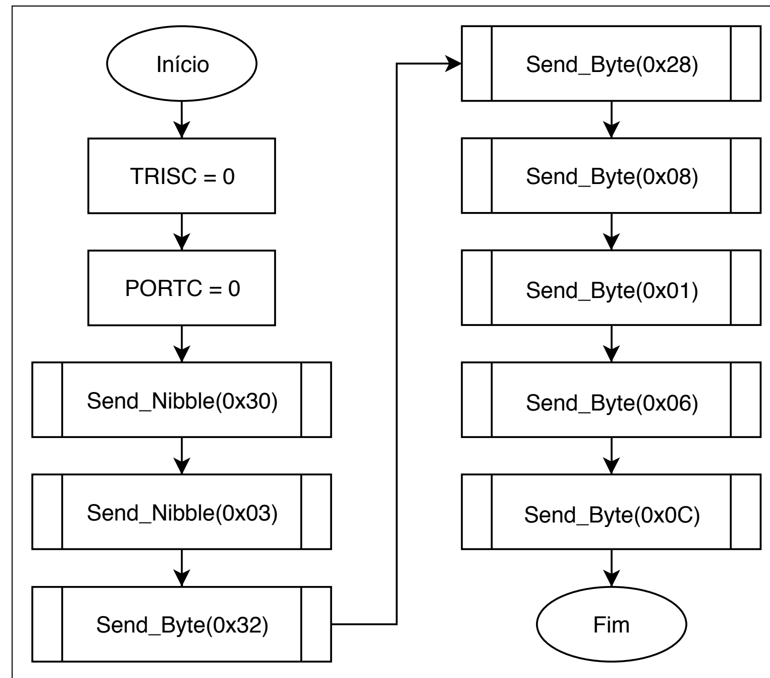
Fonte: Autoria própria.

Todos os pinos definidos pela tabela 23 são configurados com saídas digitais (TRISC = 0) inicialmente com nível lógico baixo (PORTC = 0).

É feita então a rotina de inicialização do *display*, seguindo os passos descritos no datasheet para que o mesmo funcione no modo de 4 *bits*, sem exibição do cursor

e que o mesmo não esteja piscando. Toda a temporização descrita na folha de dados também é respeitada. A sequência de inicialização é mostrada na figura 60.

Figura 60 – Fluxograma da função Lcd_Init



Fonte: Autoria própria.

3.2.1.5.4 Função Lcd_Set_Cursor

A presente função tem como finalidade o posicionamento do cursor no *display* LCD. É fornecido como parâmetro desta função o número da linha e da coluna, passados pelo usuário desta biblioteca. Seu protótipo é mostrado a seguir (algoritmo 32.

Algoritmo 32 – Protótipo da função Lcd_Set_Cursor

```
void Lcd_Set_Cursor(char row, char column);
```

Fonte: Autoria própria.

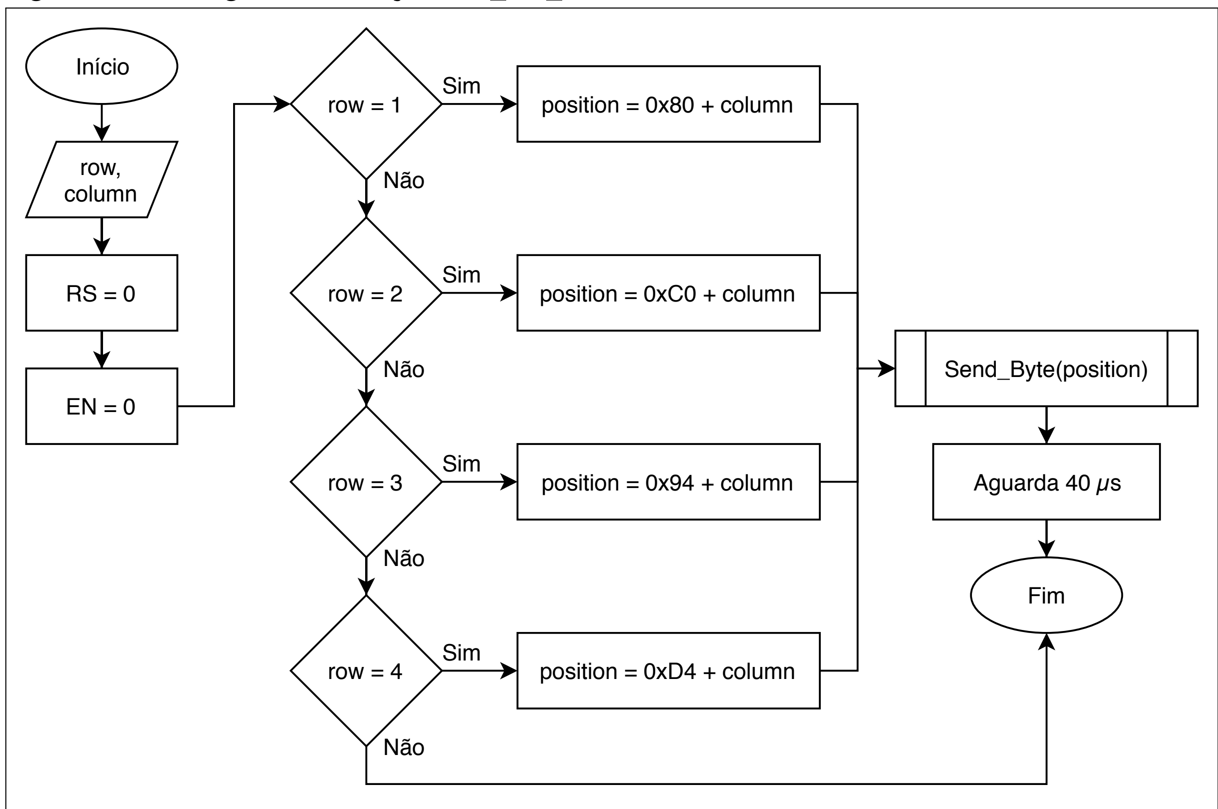
São então colocados o pino RS e o EN em nível lógico baixo, pois esta função é de configuração, não de escrita de caracteres.

A posição do cursor é obtida pela expressão 14, que leva em conta a linha e coluna requerida. O fluxograma (figura 61), presente na figura abaixo mostra o funcio-

namento da função.

$$position = \begin{cases} 0x08 + column, & se \quad row = 0 \\ 0xC0 + column, & se \quad row = 1 \\ 0x94 + column, & se \quad row = 2 \\ 0xD4 + column, & se \quad row = 3 \end{cases} \tag{14}$$

Figura 61 – Fluxograma da função Lcd_Set_Cursor



Fonte: Autoria própria.

3.2.1.5.5 Função Lcd_Write_Char

A presente função tem como objetivo escrever um único caractere no *display* LCD. Seu protótipo é mostrado abaixo (algoritmo 33.

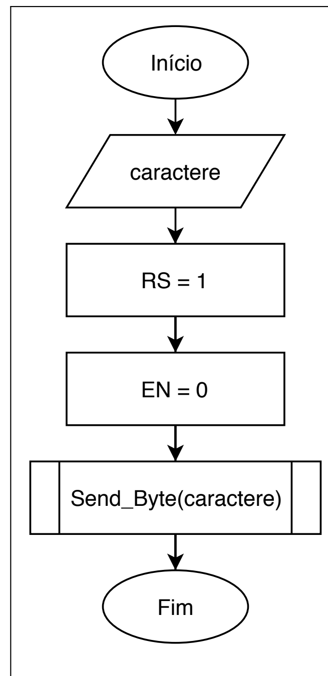
Algoritmo 33 – Protótipo da função Lcd_Write_Char

```
void Lcd_Write_Char(char caractere);
```

Fonte: Autoria própria.

Esta função sozinha possui pouca utilidade, mas quando combinada com a função Lcd_Write_String, torna-se uma poderosa ferramenta de escrita. Seu fluxograma é mostrado na figura 62.

Figura 62 – Fluxograma da função Lcd_Write_Char



Fonte: Autoria própria.

3.2.1.5.6 Função Lcd_Write_String

Esta função foi desenvolvida com a finalidade de escrever uma cadeia de caracteres, fornecida pelo usuário desta biblioteca, para o *display*. Esta cadeia é passada por referência à função, conforme mostrado abaixo.

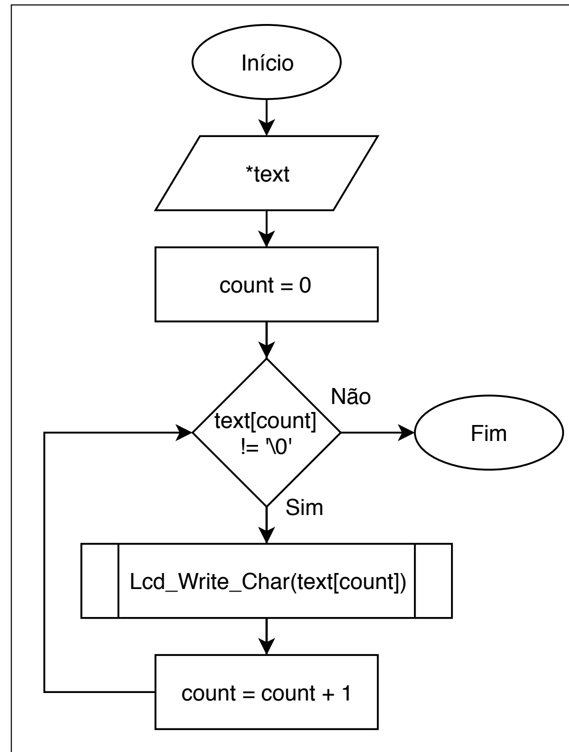
Algoritmo 34 – Protótipo da função Lcd_Write_String

```
void Lcd_Write_String(char *text);
```

Fonte: Autoria própria.

A função Lcd_Write_Char está inserida dentro de um laço de repetição (*for*), onde todas as posições deste vetor são percorridas, até encontrar o caractere nulo ($' \backslash 0'$), que marca o final da *string*. O fluxograma é mostrado na figura 63.

Figura 63 – Fluxograma da função Lcd_Write_String



Fonte: Autoria própria.

3.2.1.5.7 Função Lcd_Cmd

A função Lcd_Cmd é muito semelhante a função Lcd_Write_Char, porém esta função é destinada para configurações. Seu protótipo é mostrado no algoritmo 35.

Algoritmo 35 – Protótipo da função Lcd_Cmd

```
void Lcd_Cmd(char instruction);
```

Fonte: Autoria própria.

De acordo com o *datasheet* do *display* LCD, os comandos de configurações requerem que o pino RS esteja em nível lógico baixo. Esta é a principal diferença para a função Lcd_Write_Char.

Esta função permite que comando pré-definidos, mostrados na tabela 24, sejam executados. O comando é fornecido pelo usuário, sendo passado como parâmetro desta função.

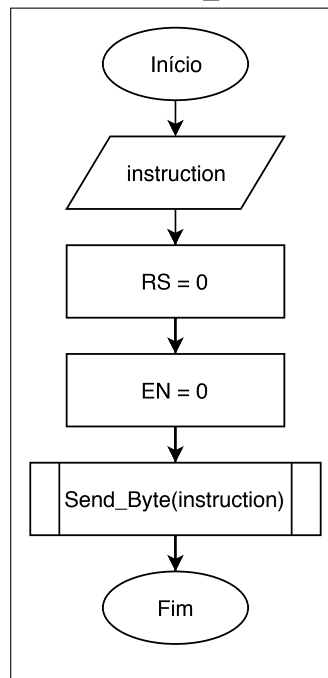
A sequência de execução desta função é exposta pelo fluxograma presente na figura 64.

Tabela 24 – Correspondência entre pinos do display e pinos do microcontrolador

Comando	Nome da constante	Valor da constante (hex)
Limpa o display	LCD_CLEAR	0x01
Cursor volta para primeira posição	LCD_RETURN_HOME	0x02
Liga o cursor	LCD_CURSOR_ON	0x0E
Desliga o cursor	LCD_CURSOR_OFF	0x0C
Faz cursor piscar (se ligado)	LCD_BLINK_CURSOR_ON	0x0F
Desativa o cursor piscante	LCD_BLINK_CURSOR_OFF	0x0E
Rotaciona para esquerda	LCD_SHIFT_LEFT	0x18
Rotaciona para direita	LCD_SHIFT_RIGHT	0x1C

Fonte: Autoria própria.

Figura 64 – Fluxograma da função Lcd_Cmd



Fonte: Autoria própria.

3.2.1.6 Biblioteca USART.h

A biblioteca USART.h é destinada à configuração, envio e recepção de mensagens entre a porta serial do microcontrolador e a porta serial do computador pessoal. É dotada de quatro funções, que serão melhores abordadas na sequência.

3.2.1.6.1 Função Usart_Init

A função Usart_Init possui dois parâmetros, passados por valor, referidos a taxa de transmissão de comunicação. O protótipo é mostrado abaixo (algoritmo 36).

Algoritmo 36 – Protótipo da função Usart_Init

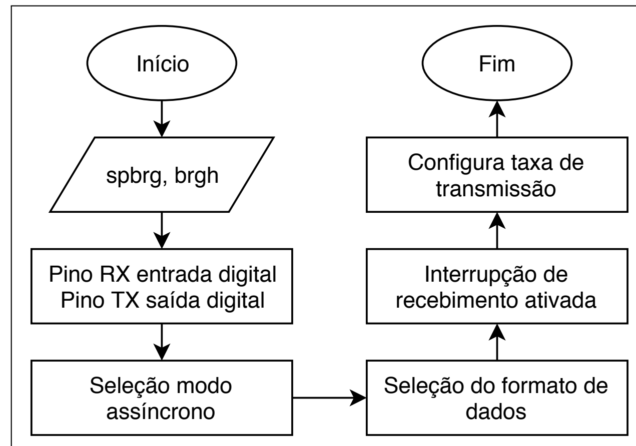
```
void Usart_Init(char spbrg, char brgh);
```

Fonte: Autoria própria.

É configurado o pino 17 como saída e o pino 18 como entrada através do registrador TRISC. Tais pinos correspondem as portas seriais do microcontrolador.

A taxa de transmissão é configurada pelos registradores SPBRG e TXSTA (*bit* BRGH), que recebem os valores fornecidos pelos parâmetros.

Figura 65 – Fluxograma da função Usart_Init



Fonte: Autoria própria.

3.2.1.6.2 *Função Usart_Write*

Esta função tem apenas um parâmetro, passado por referência, que contém a mensagem à ser transmitida. Seu protótipo é mostrado no algoritmo 37.

Algoritmo 37 – Protótipo da função Usart_Write

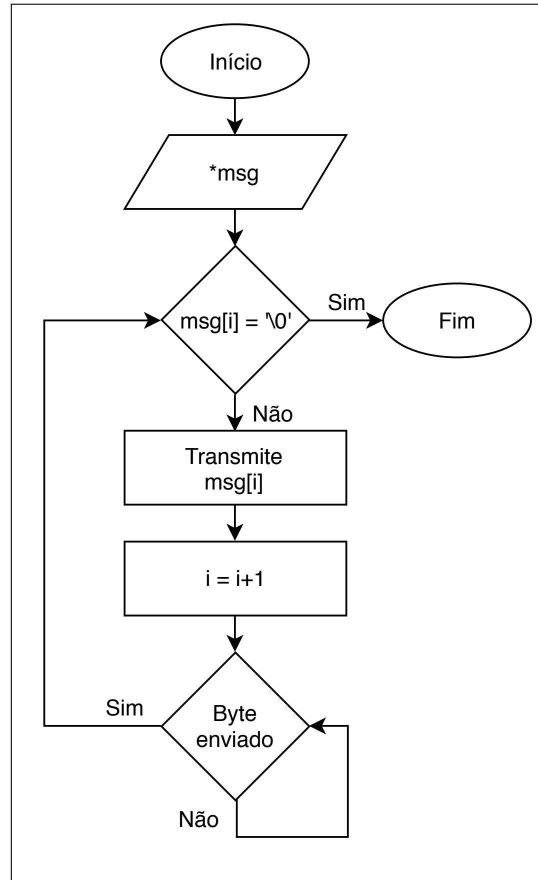
```
void Usart_Write(char *msg);
```

Fonte: Autoria própria.

A mensagem deve estar no formato de um vetor, com 8 *bits* cada posição, de qualquer comprimento. A última posição deste vetor deve ser o caractere nulo ($' \backslash 0'$), marcando o fim do vetor.

Uma estrutura de repetição percorre todo o vetor, enviando 1 *byte* por vez, até encontrar o caractere nulo. Quando isto ocorre, toda a mensagem foi transmitida, encerrando a presente função. Isto é mostrado na figura 66.

Figura 66 – Fluxograma da função Usart_Write



Fonte: Autoria própria.

O *bit* TRMT, do registrador TXSTA, indica se o caractere (8 *bits*) foi enviado. Caso verdadeira a afirmação, seu valor é 1, caso contrário, 0.

3.2.1.6.3 Função Usart_Read

De forma semelhante à função de envio, a presente função possui apenas um parâmetro, passado por referência, destinado ao armazenamento da mensagem recebida, como mostra o protótipo abaixo (algoritmo 38).

Uma estrutura de repetição permite a atribuição do *byte* recebido ao vetor referenciado enquanto o caractere nulo não for encontrado. Quando isto ocorre, a função é encerrada. O *bit* RCIF, do registrador PIR1, permite informar se o *buffer* de

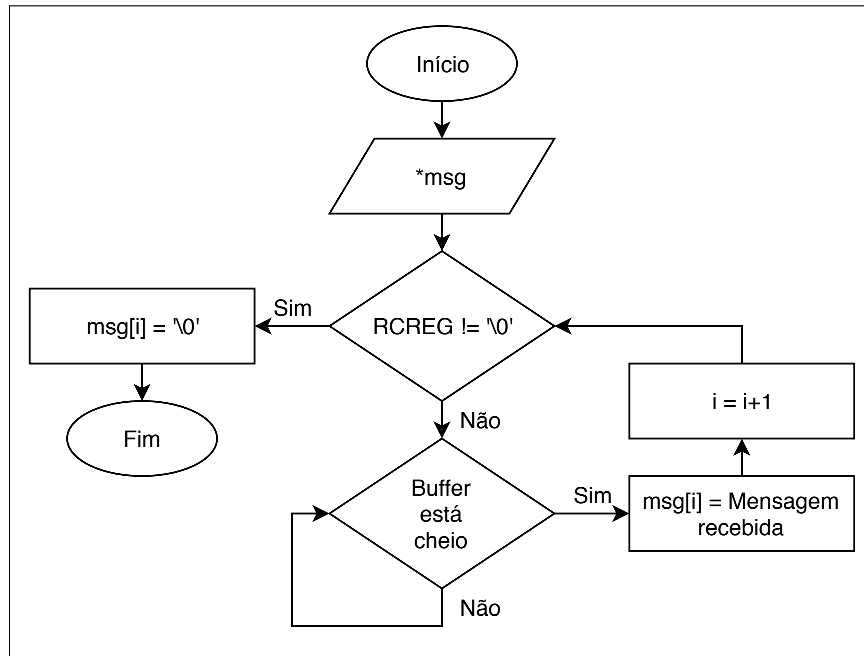
Algoritmo 38 – Protótipo da função Usart_Read

```
void Usart_Read(char *msg);
```

Fonte: Autoria própria.

recebimento (RCREG) está vazio. Quando uma mensagem é recebida, o *buffer* fica cheio, permitindo a sua leitura (figura 67).

Figura 67 – Fluxograma da função Usart_Read



Fonte: Autoria própria.

3.2.2 Programação nó CANALL

O nó CANALL é o responsável pela conexão entre o barramento CAN e a *interface* gráfica. Seu *software* foi desenvolvido em linguagem C, onde o programa principal é denominado NO_CANALL.c. São incluídas as bibliotecas conversions.h, config.h, CAN.h e USART.h. A frequência de oscilação deste nó é de $8MHz$.

Neste programa são utilizadas apenas variáveis globais, visto que existem interrupções referentes a recepção de dados via módulo USART. Tais variáveis são divididas em variáveis CAN e variáveis USART. Essas variáveis são mostradas na figura 68.

Figura 68 – Variáveis CANALL

Variáveis USART		Variáveis CAN	
char spbrg	char txFlag	char sjw	int canIdOut
char brgh	char aux	char brp	int canIdIn
char usartDataOut[9]		char propseg	char canDlc
char usartIdOut[5]		char phseg1	char canErrorCount
char usartRx[9]		char phseg2	char canMask
char usartMask[5]		union data canDataIn	char canFilter
char usartFilter		union data canDataOut	

Fonte: Autoria própria.

3.2.2.1 Função Principal nó CANALL

A primeira tarefa da função principal (*main*) é habilitar as interrupções relacionadas ao recebimento de mensagens provenientes do módulo USART (INTCONbits.GIE = 1 e INTCONbits.PEIE = 1), onde seus registradores são mostrados na no anexo A.1.

O módulo USART é inicializado com a função Usart_Init, sendo passados os parâmetros necessários para que sua temporização seja $38400baulds$. A equação 10, mostrada na seção 2.4.3, é utilizada para isto. Escolhendo o modo de alta velocidade (BRGH = 1) e substituindo as grandezas conhecidas, frequência de oscilação ($F_{osc} = 8MHz$) e taxa de transmissão desejada ($BR_{des} = 38400$), obtêm-se o valor passado ao registrador SPBRG.

$$SPBRG = \frac{F_{osc} - 16 \cdot BR_{des}}{16 \cdot BR_{des}}$$

$$SPBRG = \frac{8000000 - 16 \cdot 38400}{16 \cdot 38400}$$

$$SPBRG = 12{,}0208 \approx 12$$

Devido ao valor de SPBRG ser fracionário, é necessário aproximá-lo ao inteiro mais próximo ($SPBRG = 12$) e avaliar a diferença entre o valor calculado e o valor desejado. A equação 11 mostra a expressão da taxa de transferência calculada.

$$BR_{calc} = \frac{F_{osc}}{16 \cdot SPBRG + 16}$$

$$BR_{calc} = \frac{8000000}{16 \cdot 12 + 16}$$

$$BR_{calc} = 38461{,}5385 baulds$$

O cálculo do erro leva em base a diferença relativa entre a taxa de transferência calculada e a desejada. Seu valor é inferior a 1,5%, sendo aceitável.

$$ERRO(\%) = \frac{|BR_{calc} - BR_{des}|}{BR_{des}} \cdot 100$$

$$ERRO(\%) = \frac{|38461{,}5385 - 38400|}{38400} \cdot 100$$

$$ERRO(\%) = 0{,}1602\%$$

Para a correta inicialização do módulo CAN é necessário que o mesmo esteja em modo de configuração. Isto é feito com a função Can_Set_Mode, fornecendo como parâmetro a constante CONFIG_MODE, definida na biblioteca CAN.h. A inicialização ocorre quando a função Can_Init é chamada, sendo passado os parâmetros relacionados com a sua temporização (NBR) de $100kbps$.

Utilizando da calculadora desenvolvida em Java (seção 3.3.4.1), presente na *interface* gráfica, foram obtidos os seguintes valores de segmentos de temporização mostrados abaixo.

$$SyncSeg = 1$$
$$PropSeg = 6$$
$$PS1 = 6$$
$$PS2 = 7$$
$$SJW = 1$$

Estes valores são então substituídos na equação 9, afim de determinar o valor de BRP.

$$BRP = \frac{F_{osc}}{2 \cdot NBR \cdot (SyncSeg + PropSeg + PS1 + PS2)} - 1$$

$$BRP = \frac{8000000}{2 \cdot 100000 \cdot (1+6+6+7)} - 1$$

$$BRP = 1$$

Conhecendo os valores de SJW, BPR, PropSeg, PS1 e PS2, a função Can_Init pode ser chamada.

É recomendável que o valor do ponto de amostragem (SP) seja em torno de 70% do tamanho do *bit*. Seu valor, calculado conforme a equação 4, é mostrado abaixo.

$$SP(\%) = \frac{SyncSeg + PropSeg + PS1}{SyncSeg + PropSeg + PS1 + PS2} \cdot 100$$

$$SP(\%) = \frac{1+6+6}{1+6+6+7} \cdot 100$$

$$SP(\%) = 65\%$$

É definido também que o nó recebe qualquer mensagem válida passando o valor zero como parâmetro da função Can_Set_Mask, fazendo com que qualquer valor de identificador seja aceito.

A finalização da configuração do módulo CAN ocorre com a seleção do modo normal de operação. Isto é feito passando o parâmetro NORMAL_MODE na função Can_Set_Mode.

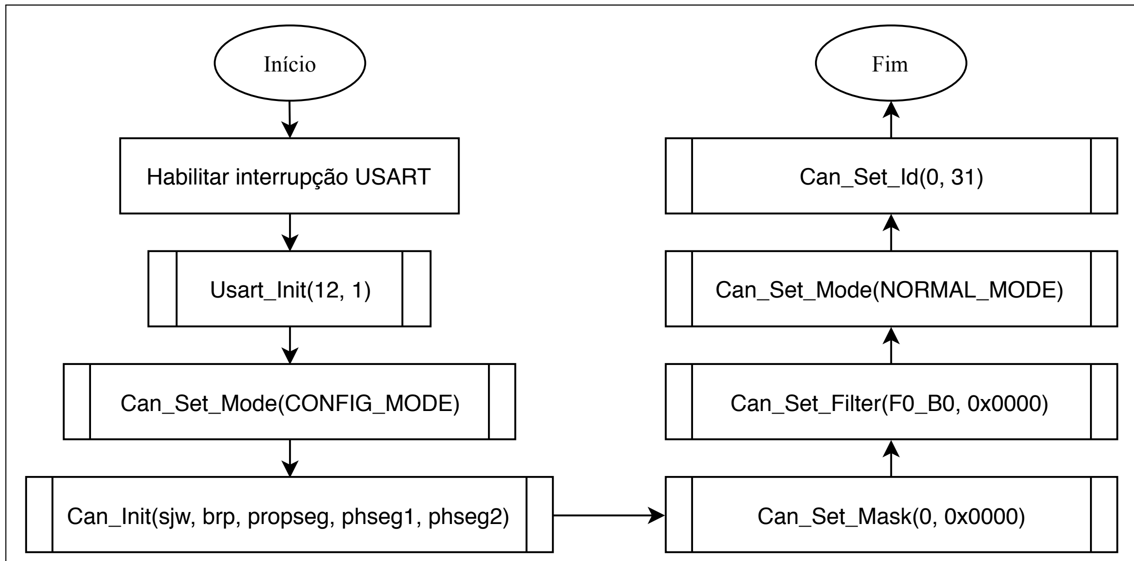
O valor do identificador é posto como 31 através da função Can_Set_Id em primeiro instante. Este valor pode ser alterado pela interface gráfica, conforme abordado nas próximas seções.

A rotina de inicialização, ou seja, aquela que é executada apenas uma vez é mostrada na figura 69.

3.2.2.1.1 Laço de repetição nó CANALL

É realizada a leitura dos dados referentes ao barramento CAN através da função Can_Read. Nela são passadas as variáveis destinadas ao recebimento de

Figura 69 – Rotina de inicialização nó CANALL



Fonte: Autoria própria.

dados (canDataIn.charType), ao recebimento do identificador (canIdIn), número de *bytes* da mensagem recebida (canDlc) e número de erros (canErrorCount).

O identificador recebido é mostrado na *interface* gráfica. Para isto é necessário enviar este valor para o computador pessoal. Seu valor (canIdIn) é convertido (função toCharArray) para um vetor de caracteres (usartIdOut), onde a base hexadecimal é a utilizada. Este valor convertido é enviado para o computador pessoal através da função Usart_Write.

Afim de separar o identificador dos dados recebidos, um caractere separador ('#') é enviado para o computador pessoal. Eles serão divididos com a função split, em java, na interface gráfica.

O envio dos dados é diferente do envio do identificador. É necessário enviá-los de acordo com o número de bytes dos mesmos. Uma estrutura condicional switch avalia as oito opções possíveis de tamanho de mensagem.

Quando a mensagem é de no máximo 4 *bytes* de tamanho, é convertido o valor numérico em um vetor de caracteres, através da função toCharArray. Este valor pode estar nas variáveis canDataIn.charType (1 *byte*), canDataIn.wordType (2 *bytes*) ou canDataIn.longType (3 e 4 *bytes*).

Devido a estrutura do compilador XC8 possuir variável de no máximo 32 *bits* (4 *bytes*), para uma mensagem maior que 4 *bytes* de tamanho, é necessário enviar os *bits* mais significativos excedentes aos 4 *bytes*, somente então enviar os 4 *bytes* me-

nos significativos. Os *bits* mais significativos excedentes aos 4 *bytes* são convertidos para o vetor de caracteres através da função toCharArray, enquanto que os 4 *bytes* menos significativos são convertidos com a função toCharArrayFixedDigits. Isso pode ser melhor observado no apêndice.

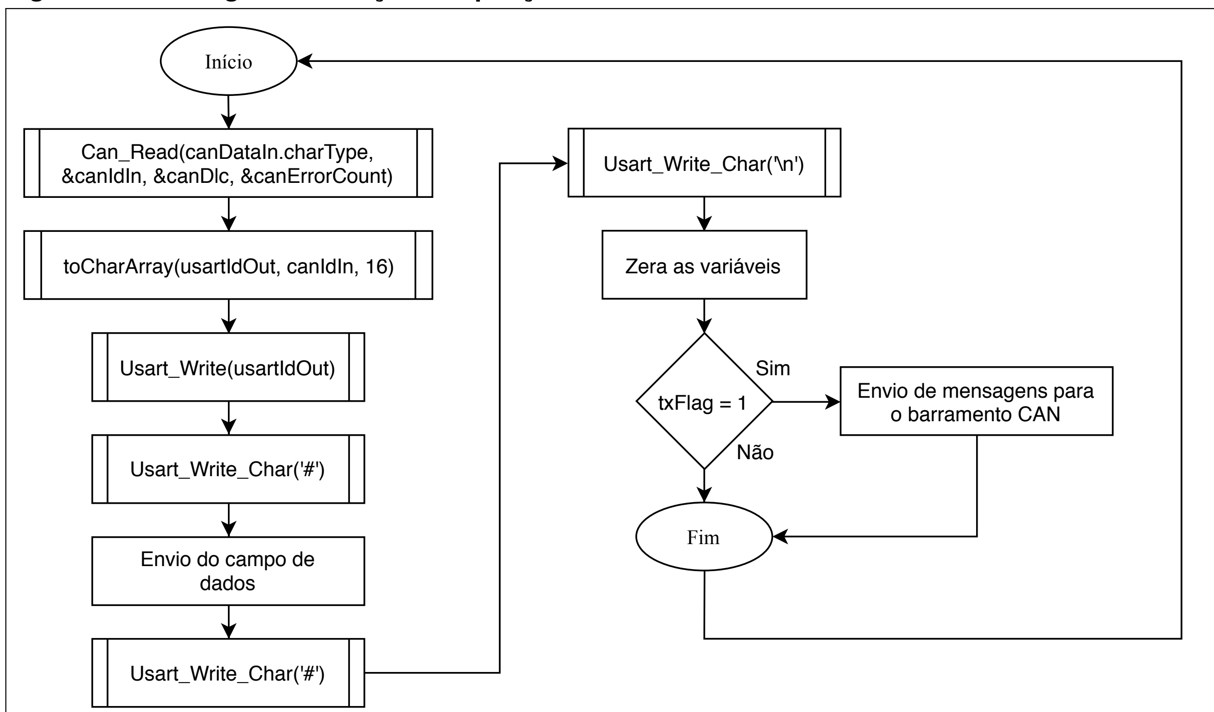
Após o envio dos dados é enviado novamente o caractere separador ('#'), sendo seguido pela quebra de linha ('$\backslash n$'), visando uma separação dos valores de identificador e dados mais fácil na interface em Java.

As variáveis de recebimento de mensagens CAN (canDataIn) são zeradas, para que não sejam sobrepostas na próxima leitura.

O envio de mensagens para o barramento CAN somente ocorre se o usuário da interface enviar um requerimento de envio, sendo realizado na interrupção do microcontrolador, conforme será abordado na sequência.

O fluxograma pertinente ao laço de repetição do nó CANALL é mostrado na figura 70

Figura 70 – Fluxograma do laço de repetição do nó CANALL



Fonte: Autoria própria.

3.2.2.2 Interrupção nó CANALL

Sempre que uma mensagem proveniente do computador chega ao microcontrolador via USART, ocorre a interrupção. Esta mensagem é primeiramente composta de um único caractere, afim de realizar as tarefas mostradas na tabela 25.

Tabela 25 – Comandos da interrupção nó CANALL

Caractere recebido	Comando
R	Aceita mensagens válidas com máscara e filtro especificados pelo usuário
S	Aceita todas mensagens válidas
T	Habilita o modo de envio de mensagens
U	Desabilita o modo de envio de mensagens

Fonte: Autoria própria.

Sabendo qual o comando requisitado pelo usuário, é necessário ler toda a mensagem recebida. Esta leitura é feita com a função Usart_Read, onde a variável que armazena os dados recebidos é denominada UsartRx.

Conforme visto pela tabela 25, quando o caractere de requisição ‘R’ é recebido, o usuário passa os valores de máscara e filtro, sendo estes valores separados pelo caractere separador (‘#’). Os valores de máscara e de filtro são convertidos de vetor de caracteres para números inteiros, através da função stringToInt (canMask e canFilter). Após esta conversão, tais valores são atribuídos ao módulo CAN através das funções Can_Set_Mask e Can_Set_Filter.

Para receber todos os valores de identificadores novamente, o caractere de requisição ‘S’ deve ser recebido, atribuindo o valor zero para a máscara e filtro.

Se o caractere de requisição ‘T’ for o selecionado, é habilitado o envio de dados da interface para o barramento CAN. Isto é possível fazendo txFlag = 1, onde este valor é comparado em uma estrutura condicional (if), que quando verdadeira, envia os dados presentes na variável canDataOut.

Quando caractere de requisição ‘U’ for recebido, é limpa a variável de txFlag, desabilitando o envio de dados.

3.2.3 Programação nó Sensor 1

Este nó, denominado Nó Sensor 1, é o responsável pela leitura e conversão do valor analógico de um potenciômetro, onde o valor de tensão de seu divisor resistivo

está situado no intervalo de $0V$ até $5V$. Seu software foi desenvolvido em linguagem C, onde o programa principal é denominado NO_SENSOR1.c. São incluídas as bibliotecas config.h, CAN.h e adc.h. A frequência de oscilação deste nó é de $8MHz$.

A única variável utilizada na programação deste nó é denominada canDataOut, onde o tipo de dado é *union data*. Sua abrangência é a função principal (*main*).

3.2.3.1 Função principal nó Sensor 1

A primeira tarefa da função principal é zerar todas a variáveis, com o intuito de evitar problemas com lixo de memória. É definido que todo os pinos que compõe o PORTC são saídas digitais (TRISC = 0), bem como o seu nível lógico é posto em nível baixo (PORTC = 0).

O módulo de conversão analógica para digital é inicializado com a função adc_Init(), presente na biblioteca adc.h. Suas características são mostradas na seção 3.2.1.1.1 deste documento.

Assim como no Nó CANALL (seção 3.2.2), para a inicialização do módulo CAN, é necessário pôr o mesmo em modo de configuração, com a função Can_Set_Mode. Toda a parte de inicialização neste nó é a mesma do Nó CANALL, além de apresentar a mesma taxa de transferência.

O valor do identificador é posto como 9 através da função Can_Set_Id, onde seu valor permanece imutável durante toda a comunicação.

A rotina de inicialização do nó Sensor 1 é mostrada na figura 71.

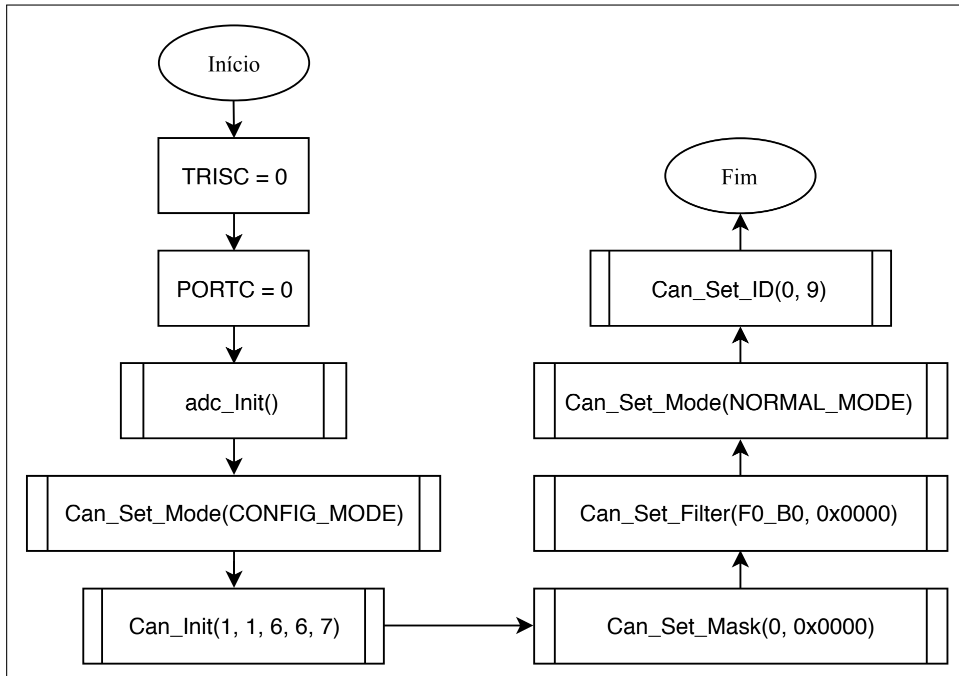
3.2.3.1.1 Laço de repetição nó Sensor 1

Durante a execução do laço de repetição infinito (*loop* infinito) do presente programa, ocorre a leitura do valor analógico presente no canal 0 do microcontrolador (pino 2). Isto é feito com a função adc_Read, passando como parâmetro o canal de leitura, neste caso, o zero.

O valor convertido possui resolução de 10 *bits*, sendo armazenado na primeira posição da variável canDataOut.wordType, que possui tamanho de 16 *bits*.

Para ocorrer o envio deste valor para o barramento CAN é necessário passa-lo como parâmetro na função Can_Write, bem como o seu tamanho de 2 *bytes* e o

Figura 71 – Rotina de inicialização nó Sensor 1



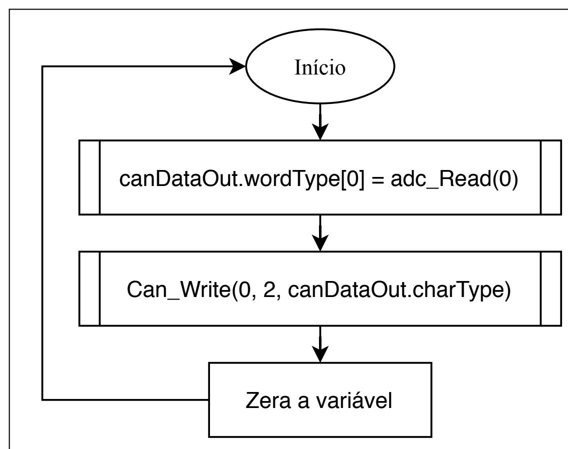
Fonte: Autoria própria.

buffer utilizado (0).

Ao final do envio desta mensagem para o barramento, as varáveis são zeradas, impedindo que valores errôneos sejam enviados nas futuras execuções do loop infinito deste programa.

A figura 72 mostra o fluxograma de funcionamento de todo o programa presente no Nó Sensor 1.

Figura 72 – Fluxograma do laço de repetição do nó Sensor 1



Fonte: Colocar fonte

3.2.4 Programação nó Sensor 2

A programação deste nó é a mesma da apresentada pelo Nó Sensor 1, apenas com a diferença do seu identificador (ID) possuir valor 19.

3.2.5 Programação nó LCD

Este nó, denominado Nó LCD, é o responsável por apresentar o valor enviado pela interface serial, através do Nó CANALL. Seu software foi desenvolvido em linguagem C, onde o programa principal é denominado NO_LCD.c. São incluídas as bibliotecas, config.h, CAN.h, lcd.h e conversions.h. A frequência de oscilação deste nó é de $8MHz$.

Neste programa são utilizadas apenas variáveis locais, onde o seu nível de abrangência é a função principal (*main*). Essas variáveis são mostradas na figura 73.

Figura 73 – Variáveis nó LCD

char sjw	int canIdIn
char brp	char canDlc
char propseg	char canErrorCount
char phseg1	char canMask
char phseg2	char canFilter
union data canDataIn	char lcdDataOut

Fonte: Autoria própria.

3.2.5.1 Função principal nó LCD

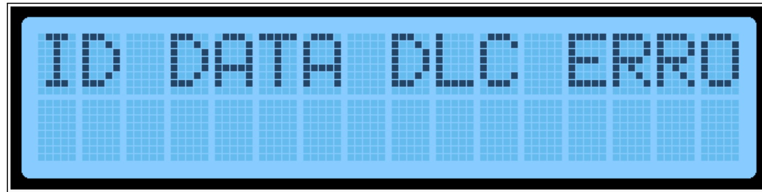
Assim como todos os outros nós, o primeiro processamento da função principal do Nó LCD é zerar todas as variáveis, pela mesma razão já citada.

Com todas as variáveis zeradas, é necessário iniciar o LCD através da função Lcd_Init(), contida na biblioteca lcd.h.

Afim de fornecer um entendimento mais rápido da mensagem recebida, é feito um cabeçalho. Este cabeçalho contém as palavras: ID, DATA, DLC e ERRO. A posição de tais palavras é feita com a função Lcd_Set_Cursor, onde o primeiro parâmetro é linha e o segundo é a coluna. Todas as palavras do cabeçalho estão localizadas na primeira linha, estando na coluna 0, 3, 8 e 12, na ordem em que foram citadas acima.

A figura 74 mostra o a forma como tal cabeçalho é apresentado.

Figura 74 – Cabeçalho display LCD



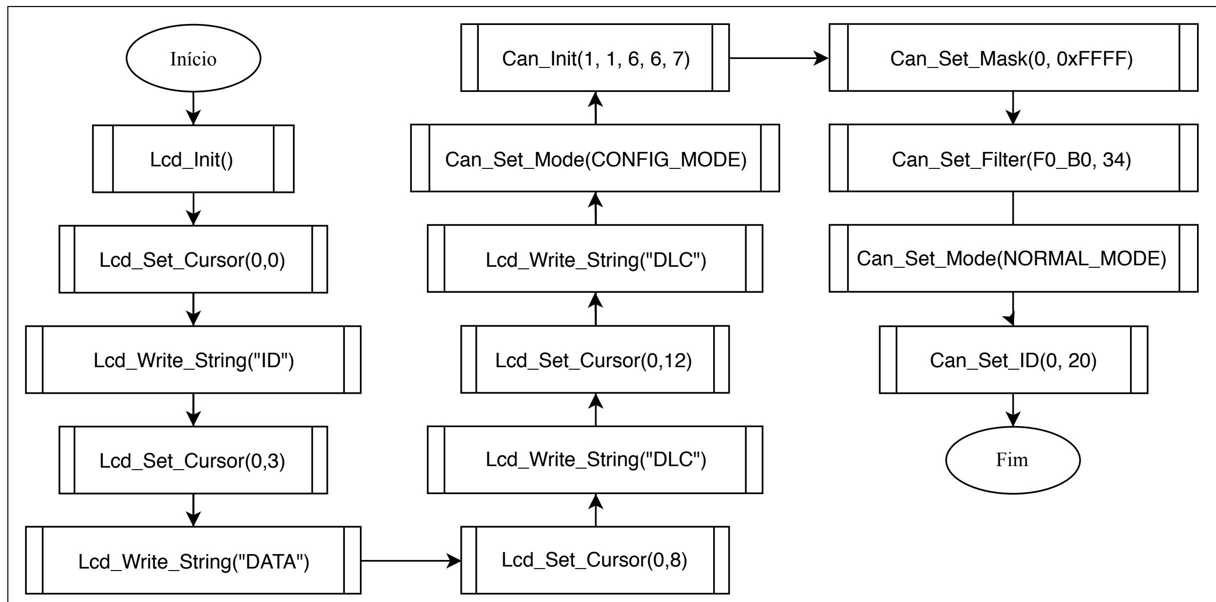
Fonte: Autoria própria.

A inserção do cabeçalho fora do laço de repetição faz com que ele seja inserido apenas uma única vez, reduzindo o tempo de execução do laço de repetição principal.

A inicialização do módulo CAN é a mesma que a apresentada pelo Nó CANALL, apenas com a diferença do seu valor de identificador e filtro. Seu valor de identificador é 20, enquanto que o seu valor de filtro é 11, ou seja, apenas mensagens provenientes deste ID são recebidas pelo nó LCD, pois seu valor de máscara é 2047.

A figura 75 mostra o fluxograma da rotina de inicialização do nó LCD.

Figura 75 – Rotina de inicialização nó LCD



Fonte: Autoria própria.

3.2.5.1.1 Laço de repetição nó LCD

No *loop* infinito deste programa, ocorre a leitura dos dados provenientes do barramento CAN, caso os mesmos possuam identificadores de valor 11. Qualquer

outro valor de identificador é rejeitado, portando não é processado.

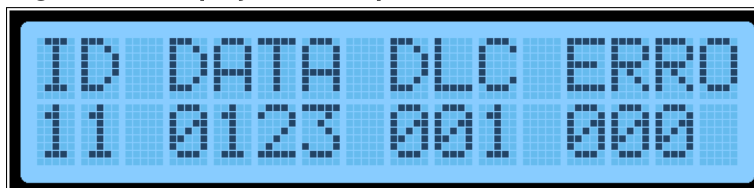
Tais dados são obtidos com a função Can_Read, sendo passados como parâmetro as variáveis canDataIn.charType, canIdIn, canDlc e errorCount.

Todas essas variáveis são convertidas para o formato de vetor de caractere através da função toCharArrayFixedDigits, para que possam ser devidamente apresentadas no *display* LCD.

A utilização da função toCharArrayFixedDigits é justificada pelo motivo que o tamanho da mensagem recebida pode variar. Quando a função toCharArray é utilizada em seu lugar, uma mensagem recebida de valor menor que a mensagem anterior apresenta na tela seu digito de maior significância de maneira estática, fornecendo um valor errôneo ao usuário.

Os valores convertidos apresentados pelo LCD nas posições 0, 3, 8 e 12 da segunda linha do LCD. A figura 76 mostra a combinação do cabeçalho com os valores convertidos. Após a apresentação dos valores, todas as variáveis são novamente zeradas.

Figura 76 – Display LCD completo



Fonte: Autoria própria.

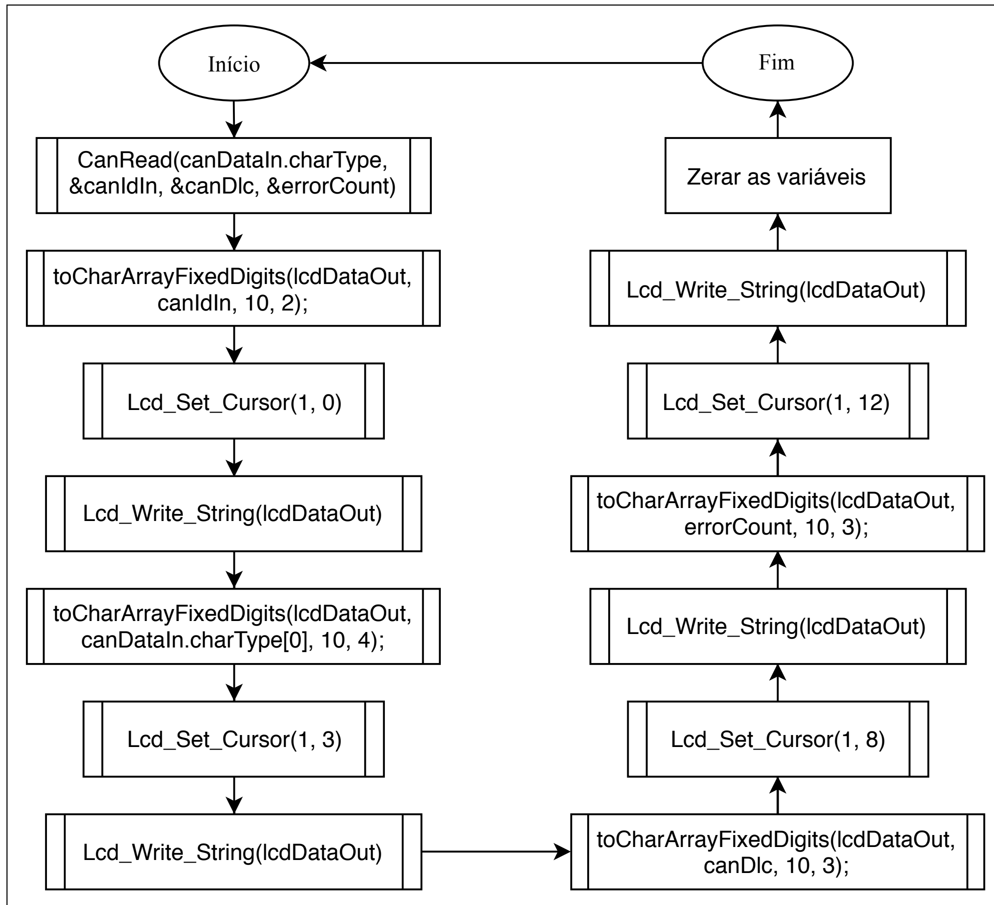
3.2.6 Programação nó Gerador de Mensagens

Este programa tem a finalidade de enviar mensagens com diferentes valores de identificador (ID) para que mais dados trafeguem pelo barramento CAN. As variáveis presentes deste nó podem ser visualizadas na figura 78.

3.2.6.1 Função principal nó Gerador de Mensagens

A rotina de inicialização consiste em configurar a temporização do barramento CAN de mesmo modo que em outros nós. Como não é desejado receber nenhuma mensagem do barramento CAN, sua máscara é configurada com os todos os bits em

Figura 77 – Fluxograma do laço de repetição do nó LCD



Fonte: Autoria própria.

Figura 78 – Variáveis nó Gerador de Mensagens

char sjw	char phseg1
char brp	char phseg2
char propseg	union data canDataOut

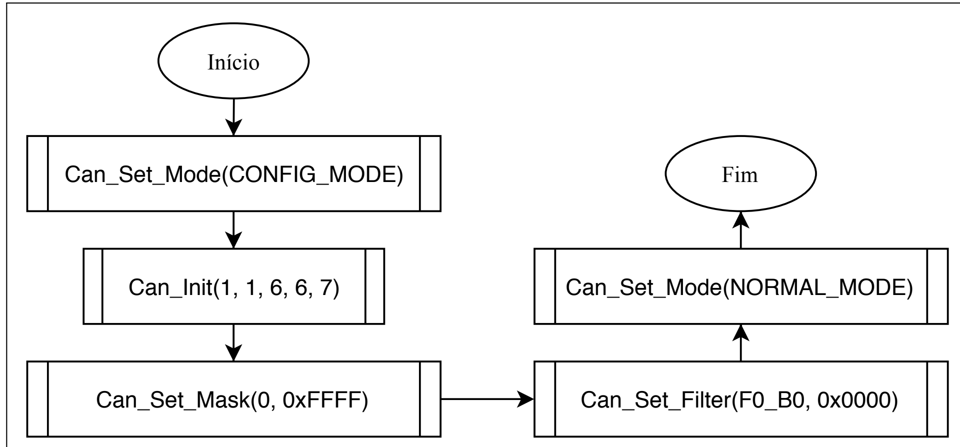
Fonte: Autoria própria.

nível lógico alto (1). O fluxograma, presente na figura 79 mostra a rotina de inicialização.

3.2.6.1.1 Laço de repetição nó Gerador de Mensagens

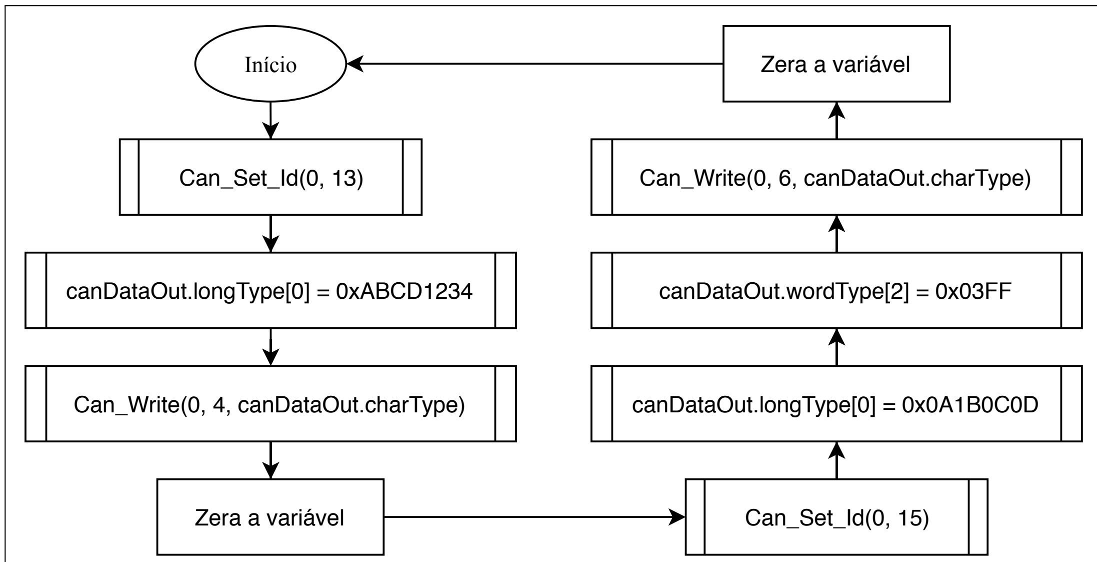
O loop infinito do nó Gerador de Mensagens envia mensagens de 4 *bytes* e 6 *bytes* com os ID's 13 e 15. Isto é mostrado na figura 80.

Figura 79 – Rotina de inicialização nó Gerador de Mensagens



Fonte: Autoria própria.

Figura 80 – Fluxograma do laço de repetição do nó Gerador de Mensagens

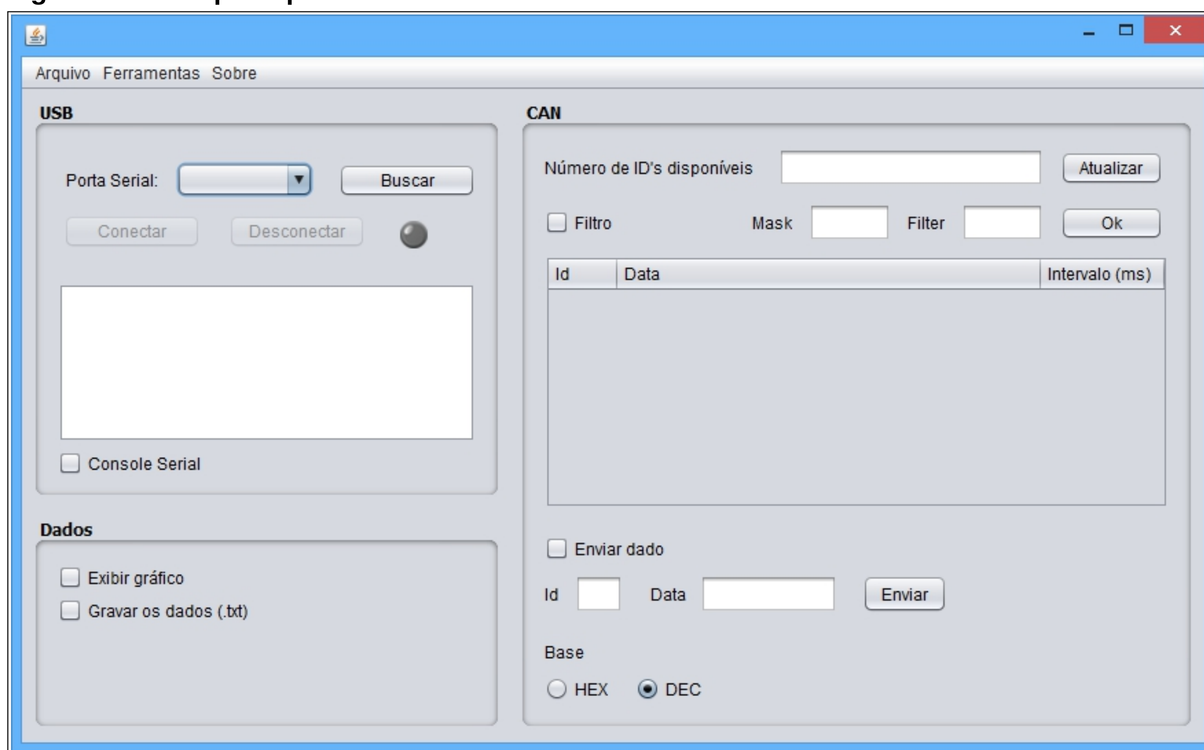


Fonte: Autoria própria.

3.3 INTERFACE GRÁFICA

A *interface* gráfica CANALL foi desenvolvida utilizando a linguagem Java com o auxílio do *software* NetBeans. Ela é a responsável pela exibição e aquisição dos dados circulantes no barramento CAN. A tela principal da *interface* é mostrada na figura 81.

Figura 81 – Tela principal CANALL

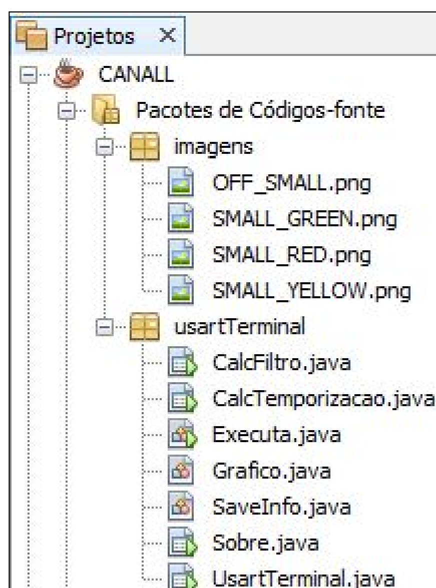


Fonte: Autoria própria.

Nota-se que a mesma possui uma divisão em painéis, denominados: painel USB, painel CAN e painel de dados. Há também a presença de menus, denominados: menu ferramentas e menu ajuda. Tais componentes serão abordados nas seguintes seções.

Para o desenvolvimento da *interface* gráfica CANALL, foram elaborados os pacotes imagens e usartTerminal. O primeiro armazena os ícones utilizados no *software*, enquanto que no segundo estão as classes referentes a comunicação USB, elementos gráficos, calculadoras e salvamento de dados. A imagem 82 mostra os pacotes e as classes do projeto CANALL.

Figura 82 – Pacotes e classes desenvolvidos



Fonte: Autoria própria.

3.3.1 Painel USB

Este painel é responsável pela configuração da conexão entre o computador pessoal e o microcontrolador PIC18F258. É utilizada a biblioteca RXTX (2011), que permite a transmissão e envio de dados utilizando as portas seriais ou USB. Toda a documentação desta biblioteca pode ser encontrada no site da Oracle (ORACLE CORPORATION, 2018a).

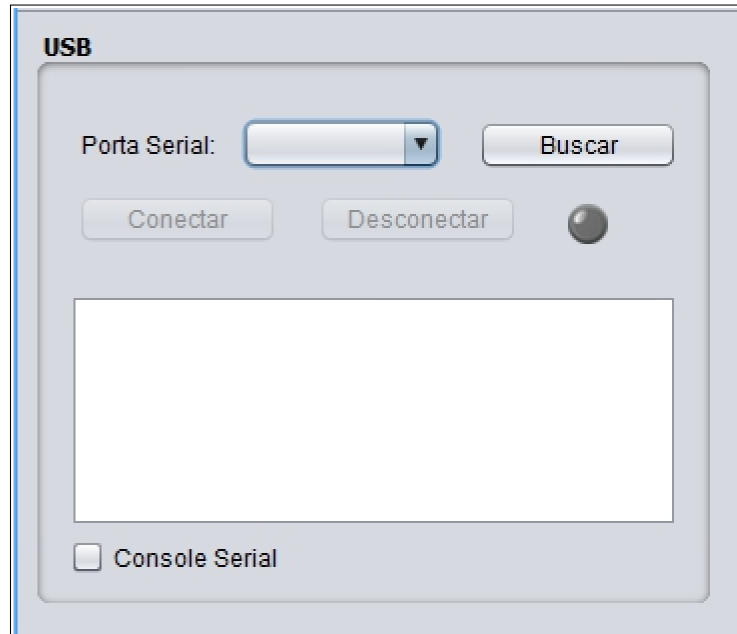
Quando o botão buscar é pressionado, a caixa de combinação é preenchida com as portas seriais disponíveis. A conexão somente ocorre quando o botão conectar é pressionado. Para encerrar a conexão, o botão desconectar é necessário. As figuras 83, 84 e 85 mostram esta operação.

No estado desconectado, o botão conectar e desconectar são bloqueados para o usuário neste instante, visto que é necessário buscar uma porta para haver a conexão. Também é necessário estar conectado para desconectar.

Quando uma porta USB é reconhecida pelo *software*, é possível estabelecer a conexão. Portando, quando o estado conectando é o presente, o botão conectar é então liberado ao usuário. O botão desconectar ainda está bloqueado, pois neste momento não há nenhuma conexão em andamento. Quando há uma conexão em andamento, apenas o botão desconectar é desbloqueado ao usuário.

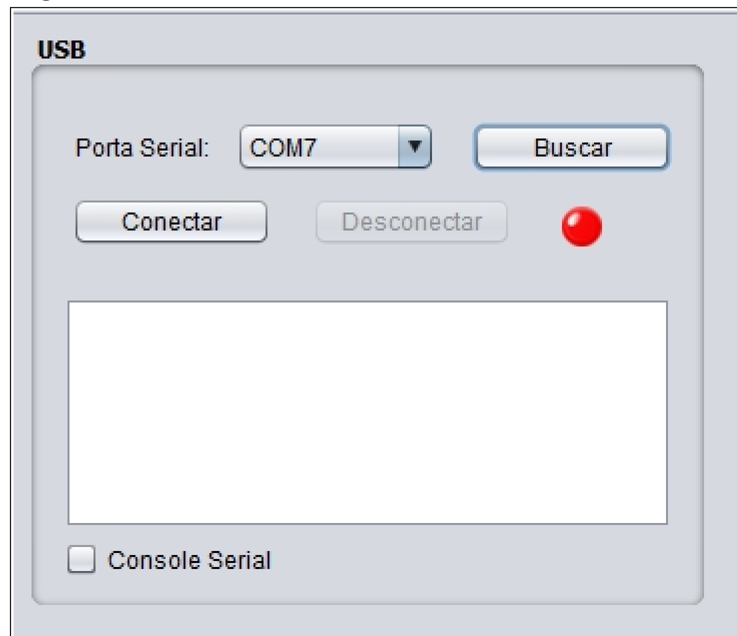
No momento em que uma conexão é estabelecida, o console serial apresenta

Figura 83 – Interface CANALL: desconectada



Fonte: Autoria própria.

Figura 84 – Interface CANALL: conectando

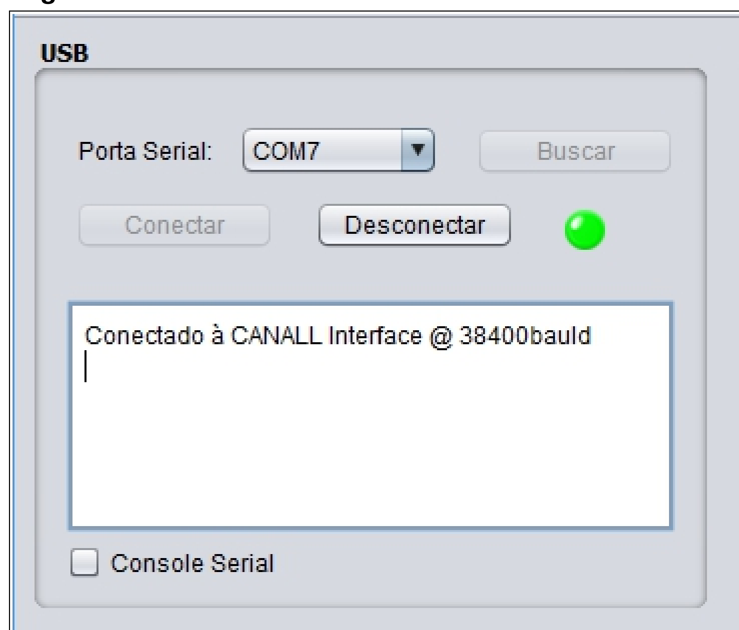


Fonte: Autoria própria.

a mensagem mostrada na figura 85.

Por padrão, a velocidade de comunicação é fixada em $38400bauds$, transmitindo oito *bits* de dados acrescido de um *bit* de parada. Uma pequena *label*, de $20x20pixels$, mostra o estado da conexão USB por meio de três imagens (LED apagado, LED vermelho e LED verde). A primeira é relacionada com o estado desconectado, a segunda com o conectando e a terceira com o conectado.

Figura 85 – Interface CANALL: conectada



Fonte: Autoria própria.

Toda vez que um dado é recebido pela porta serial do computador ocorre uma notificação, que interrompe a execução do programa para a leitura de tais dados. Isto é possível através do método notifyOnDataAvailable, presente na biblioteca RXTX Serial.

A caixa de seleção Console Serial permite a exibição da mensagem recebida via USB por meio da área de texto presente neste painel. Com isto é possível visualizar como o microcontrolador está enviando as mensagens para o computador. Isto será apresentado no capítulo Resultados e Discussões. As mensagens provenientes do microcontrolador estão em formato hexadecimal.

3.3.2 Painel CAN

No Painel CAN é possível configurar máscara e filtro do microcontrolador CAN, enviar e receber dados referente à comunicação CAN além de mudar o sistema de base (decimal ou hexadecimal). É também possível enviar mensagens da *interface* CANALL para o barramento CAN.

Os dados recebidos são colocados em uma tabela que contém o identificador da mensagem, os dados e o intervalo entre cada recebimento, como mostrado na figura 86.

Um vetor de 32 *bits* cada posição, denominado idList, é criado para armaze-

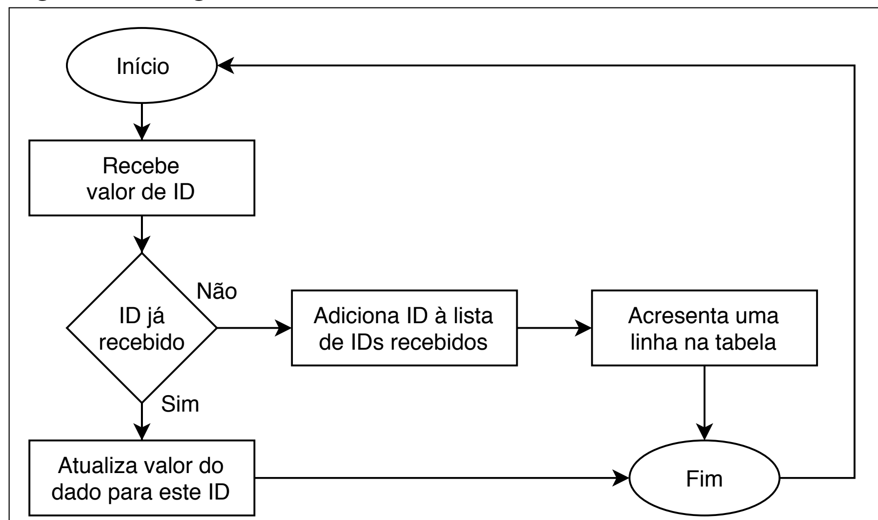
Figura 86 – Tabela de recebimento de mensagens

Id	Data	Intervalo (ms)

Fonte: Autoria própria.

nar todos os diferentes valores de identificadores recebidos. Isto é necessário para a apresentação dos dados na tabela, pois a cada mensagem recebida, o identificador desta mensagem compara se o valor já está presente no vetor idList. Caso esteja, o valor de dados na tabela é apenas atualizado, caso contrário uma nova linha na tabela é criada. O tamanho deste vetor indica quantos identificadores diferentes foram recebidos, sendo mostrado no campo de texto Número de ID's disponíveis. Isto é mostrado no fluxograma presente na figura 87.

Figura 87 – Lógica da lista de identificadores



Fonte: Autoria própria.

A coluna Intervalo é medida em milissegundos, tomando como base também o vetor idList. O tempo inicial de execução do programa, também medido em milissegundos, é fornecido pelo método System.currentTimeMillis(). Para cada ocorrência de um mesmo valor de identificador, é diminuído o valor do tempo de sistema atual pelo tempo de sistema anterior, armazenando no vetor intervalTime.

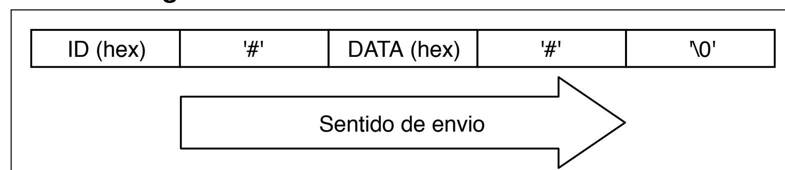
Com relação a filtragem de identificadores, a caixa de diálogo Filtro deve ser selecionada. A sua seleção faz o computador enviar o caractere ‘R’ para o microcontrolador, que faz com que o mesmo entre em seu modo de configuração de máscara e filtro. Caso o mesmo seja desabilitado, o caractere ’S’ é o enviado. Isto é compatível com a tabela 25.

Os valores presentes nos campos de texto filtro e máscara são preenchidos pelo usuário, de forma que seu valor é enviado ao microcontrolador quando o botão Ok é pressionado. O PIC18F258 separa o filtro da máscara com a função split, descrita nas seções acima.

É possível mudar todo o sistema de base do painel CAN, com o botão HEX ou DEC, que converte para o sistema hexadecimal ou decimal, respectivamente. No sistema hexadecimal os dados da tabela são separados a cada *byte*. Isto será mostrado no capítulo Resultados e Discussões.

O envio de mensagens através da interface CANALL para o barramento CAN também é possível. Para isto, a caixa de seleção Enviar dado deve ser selecionada, o que manda para o microcontrolador o caractere ‘T’, que faz com que o mesmo entre no modo de envio de mensagens. A sua desabilitação faz a interface enviar o caractere ’U’ para o microcontrolador. Isto também é compatível com a tabela 25. O formato da mensagem enviada é mostrada na figura 88.

Figura 88 – Formato de envio de mensagens da interface gráfica



Fonte: Autoria própria.

3.3.3 Painel Dados

O Painel Dados permite mostrar graficamente os dados recebidos, bem como salvá-los em um arquivo de texto. Isto permite que o usuário desta *interface* possa relacionar matematicamente os dados de diferentes sensores.

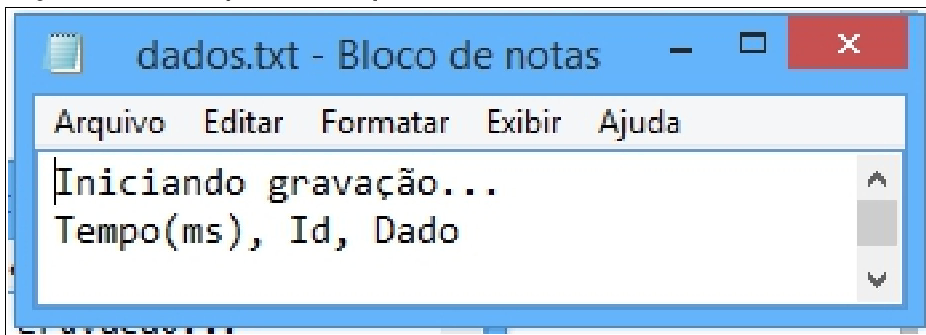
A caixa de seleção Exibir Gráfico abre uma nova janela, em que cada vez que um novo dado é recebido, a mesma é atualizada. Para isto é utilizada a biblioteca

jChart2D (2011).

O gráfico comporta apenas uma variável no eixo das ordenadas, ou seja, é necessário filtrar o valor para apenas um identificador para exibir os dados. O eixo das abscissas é medido em milissegundos.

A caixa de seleção Gravar os Dados salva os dados recebidos em um arquivo com extensão .txt, denominado dados, na mesma pasta em que a *interface* está instalada. Nela não há necessidade de filtrar mensagens. O cabeçalho deste arquivo de texto é mostrado na figura 89.

Figura 89 – Cabeçalho do arquivo salvo



Fonte: Autoria própria.

3.3.4 Menu Ferramentas

Este menu tem por finalidade auxiliar a programação do microcontrolador com módulo CAN. Nele há duas calculadoras, uma referente à temporização e outra destinada à filtragem de mensagens.

3.3.4.1 Submenu Temporização

É fornecido pelo usuário da interface três parâmetros para o cálculo dos segmentos de temporização da rede CAN, sendo eles: frequência de operação do microcontrolador, taxa de transferência e ponto de amostragem, como mostrado na figura 90.

Com o conhecimento destas três grandezas é impossível a determinação de uma única solução para as seis demais, sendo elas: SyncSeg, PropSeg, PS1, PS2,

Figura 90 – Calculadora de temporização

Fonte: Autoria própria.

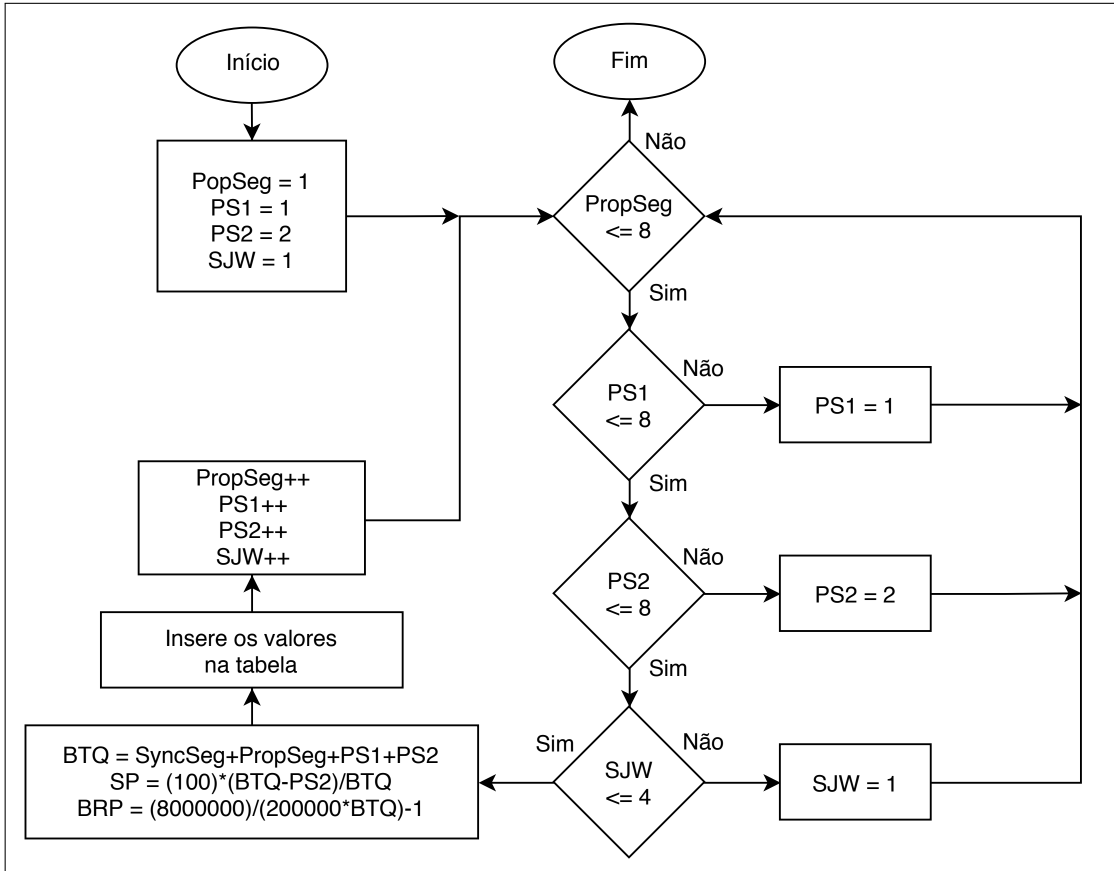
SJW e BRP. As soluções devem atender as condições impostas pela equação 15.

$$
\begin{cases}
SyncSeg = 1 \\
1 \leq PropSeg \leq 8 \\
1 \leq PS1 \leq 8 \\
1 \leq PS2 \leq 8 \\
1 \leq SJW \leq 4 \\
PS2 \geq SJW \\
PropSeg + PS1 \geq PS2 \\
SyncSeg + PropSeg + PS1 + PS2 \leq 20 \\
BRP < 63 \\
BRP \in \mathbb{Z}_+
\end{cases}
\tag{15}
$$

As cinco primeiras condições são definidas pela ISO 11898-1 (INTERNATIONAL STANDARDIZATION ORGANIZATION, 2003a), enquanto as demais são definidas pela folha de dados do microcontolador da Microchip.

Para determinar as grandezas é necessário utilizar estruturas de repetição concatenadas, e no seu final comparar se a solução atende as condições da equação 15, conforme mostrado no fluxograma presente na figura 91.

Figura 91 – Lógica da calculadora de temporização



Fonte: Autoria própria.

3.3.4.2 Submenu Filtro

Afim de saber a faixa de valores de identificadores de formato padrão (11 *bits*) que são aceitos pelo microcontrolador, a calculadora mostrada na figura 92 foi desenvolvida.

São fornecidos os valores da máscara, filtro e identificador, para o cálculo de valores de IDs aceitos. O valor de ID fornecido para o usuário serve para verificar se o mesmo será aceito para os valores de máscara e filtro. Caso afirmativo, a mensagem “Aceita” é mostrada, caso contrário a mensagem “Rejeita” é mostrada. Isto será mostrado no próximo capítulo.

Para determinar a faixa de valores, a expressão 16 é inserida numa estrutura

Figura 92 – Calculadora de filtro

Fonte: Autoria própria.

de repetição, onde o valor de id_i é o incremento. Com isto, os valores de ID são testados de 0 até 2047, que representam os valores possíveis de ID. Todos os valores que satisfazem a expressão acima são mostrados na área de texto, conforme exposto na figura 92.

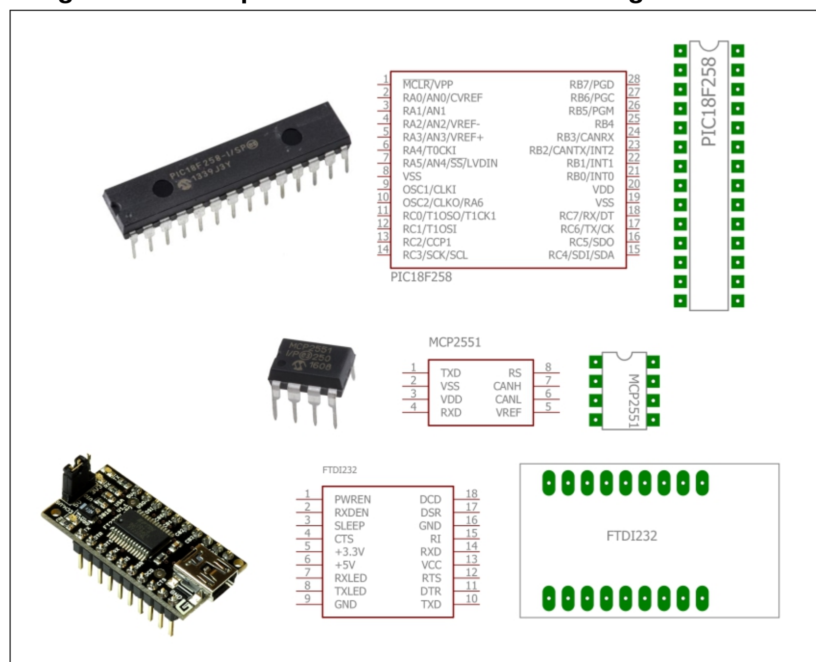
$$\begin{cases} Aceita, se : \prod_{i=0}^{10}((filtro_i \bigoplus id_i) + (!mascara_i)) \ll i = 2047 \\ Rejeita, se : \prod_{i=0}^{10}((filtro_i \bigoplus id_i) + (!mascara_i)) \ll i \neq 2047 \end{cases} \tag{16}$$

3.4 PLACAS DE CIRCUITO IMPRESSO

Para o desenvolvimento do *layout* das placas de circuito impresso (PCI's) foi utilizado o *software* Eagle. Seu uso é justificado pela sua facilidade de utilização, bem como o grande número de usuários deste *software*, sendo *hobbistas*, estudantes ou profissionais.

Para o projeto, foi necessária a criação de três componentes: PIC18F258, MCP2551 e Conversor USART-USB. Os dois primeiros já possuíam encapsulamentos pré-existentes (PDIP-28 e PDIP-8). Para o terceiro foi necessária a consulta do *datasheet* do módulo, afim de saber as dimensões do mesmo. A figura 93 mostra o símbolo e o encapsulamento destes três componentes.

Figura 93 – Componentes desenvolvidos no Eagle



Fonte: Autoria própria.

Conforme mostrado na figura 38, este trabalho de conclusão de curso possui 5 nós, onde alguns compartilham características em comum. As subseções seguintes descrevem as semelhanças e diferenças destas placas de circuito impresso. Os esquemáticos das mesmas podem serem observados no apêndice deste documento, já o seu modelo físico pode ser observado na figura 38.

3.4.1 Itens comuns as placas de circuito impresso

Devido à alimentação das placas de circuito impresso ser $24V$, há a necessidade de utilizar um regulador de tensão (7805). Este regulador fornecerá a tensão ($5V$) e a corrente necessária para o funcionamento de todos os nós.

Capacitores cerâmicos de $100nF$ são colocados próximos aos pinos de alimentação dos circuitos integrados, filtrando ruídos de alta frequência. Estes capacitores também são chamados de capacitores de desacoplamento.

Um resistor de $10k\Omega$ é inserido para limitar a corrente entre $+5V$ e o pino 1 (MCLR) do microcontrolador PIC18F258. Isto garante que ele não será reiniciado.

A fonte de *clock* para o microcontrolador é proveniente de um cristal oscilador, com frequência de $8MHz$. Este cristal fica o mais próximo possível dos pinos 9 e 10 do microcontrolador (OSC1 e OSC2). Nos terminais deste cristal é conectado capacitores cerâmicos de $22pF$, conforme recomendado pelo *datasheet*.

Os pinos 23 e 24 do microcontrolador (CANTX e CANRX) foram colocados o mais próximo possível dos pinos 1 e 4 do *transceiver* (TXD e RXD). Isto é recomendado, pois neste ponto o nível de tensão é lógico ($0 - 5V$), e não diferencial, fazendo suscetível erros causados por interferências eletromagnéticas.

A conexão do *transceiver* MCP2551 com o barramento CAN é feita através dos pinos 6 e 7 (CANL e CANH).

O pino 5 do *transceiver* (VREF) é colocado em um divisor de tensão. Deste divisor de tensão, uma tensão de aproximadamente $2{,}5V$ é empregada neste pino.

O pino 8 do *transceiver* (RS) controla a taxa de variação (*slew rate*). Usando um resistor de $10k\Omega$ entre este pino e o potencial de $0V$ (GND), permitindo uma taxa de variação de $25V/\mu s$ (30).

A gravação do microcontrolador se dá pelo conector ICSP, que conecta os pinos 1 (MCRL), 27 (PGC) e 28 (PGD). Neste conector também há conexão com $+5V$ e GND.

Os conectores de alimentação (GND e +Vcc), barramento CAN (CANH e CANL) e de gravação (ISCP) são colocados próximos as bordas da placa de circuito impresso.

3.4.2 Placa para o nó Sensor 1 e nó Sensor 2

O pino central do potenciômetro de $10k\Omega$ é conectado ao primeiro canal analógico do microcontrolador (AN0), sendo que os demais pinos são conectados com os potenciais de $0V$ (GND) e $5V$.

Existe também um resistor de $100k\Omega$ conectado entre o canal analógico AN0 (pino 2) e o referencial de $0V$. Isto garante que caso o potenciômetro esteja desconectado do circuito, o canal AN0 irá interpretar um valor de $0V$, evitando flutuação de tensão neste pino.

3.4.3 Placa para o nó LCD

A conexão entre o *display* LCD e o microcontrolador é dado pelo PORTC, mais especificamente pelos pinos 11, 12, 13, 14, 15 e 16.

O pino central *trimpot* de $10k\Omega$ é conectado ao pino de contraste do display LCD (pino 2), tornando possível o ajuste de contraste do *display*.

Visto que o *display* será utilizado apenas para escrita de dados, o pino 5 do *display* LCD (R/W) é aterrado, não permitindo realizar leituras sobre o mesmo.

3.4.4 Placa para o nó CANALL

Para conectar o PIC18F258 com o módulo USART-USB (FTDI RS-232), é necessário somente dois pinos do microcontrolador, o 17 (TX) e 18 (RX). A conexão com o GND é também necessária.

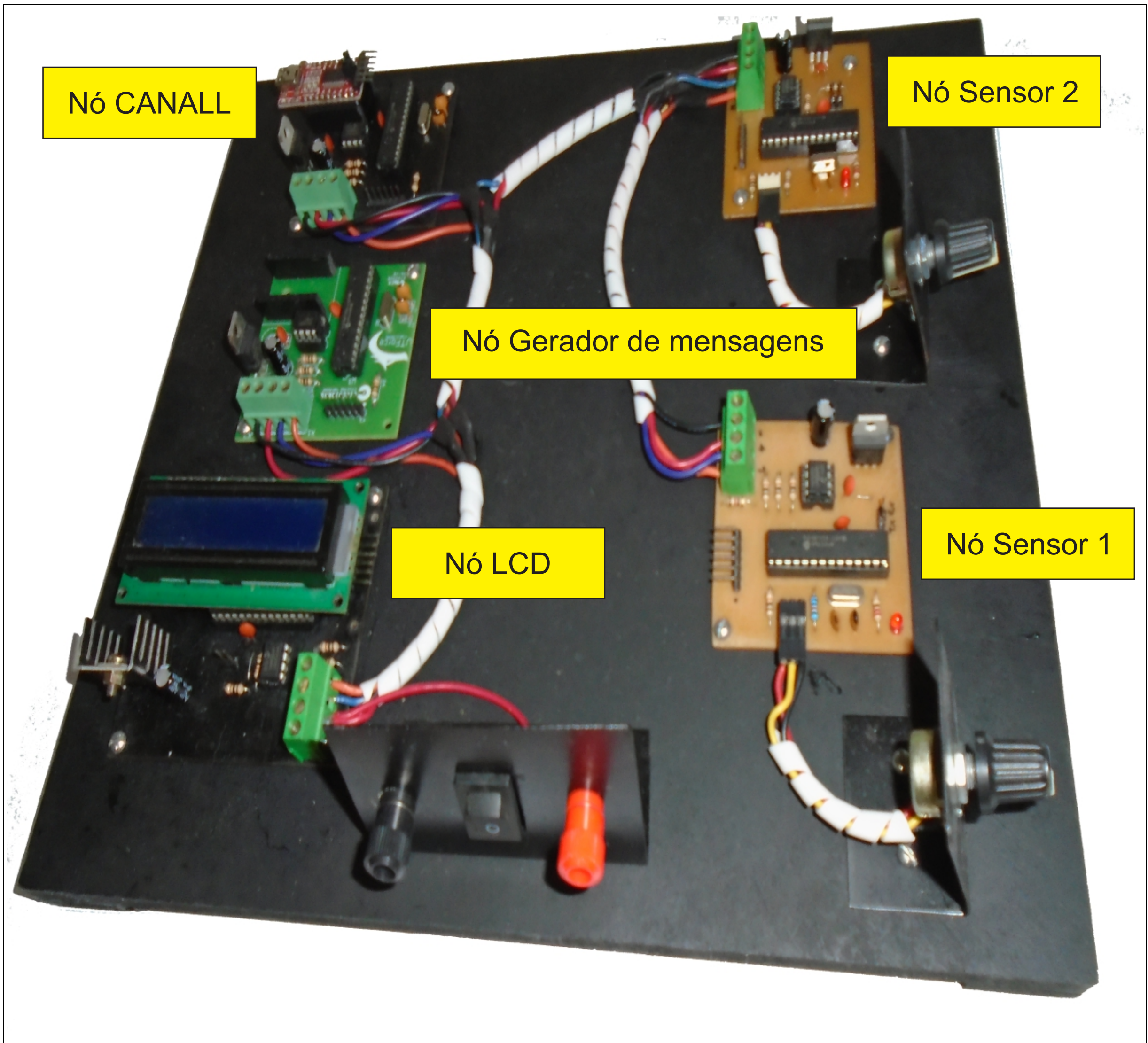
3.4.5 Placa para o nó Gerador de Mensagens

Qualquer uma das placas descritas nas seções acima poderia ser utilizada para o nó Gerador de Mensagens, visto que todas possuem conexão com o barramento CAN.

4 RESULTADOS E DISCUSSÕES

A figura 94 mostra o barramento CAN com todos os nós descritos no capítulo Desenvolvimento. Ele é a implementação física da figura 38.

Figura 94 – Implementação física do projeto



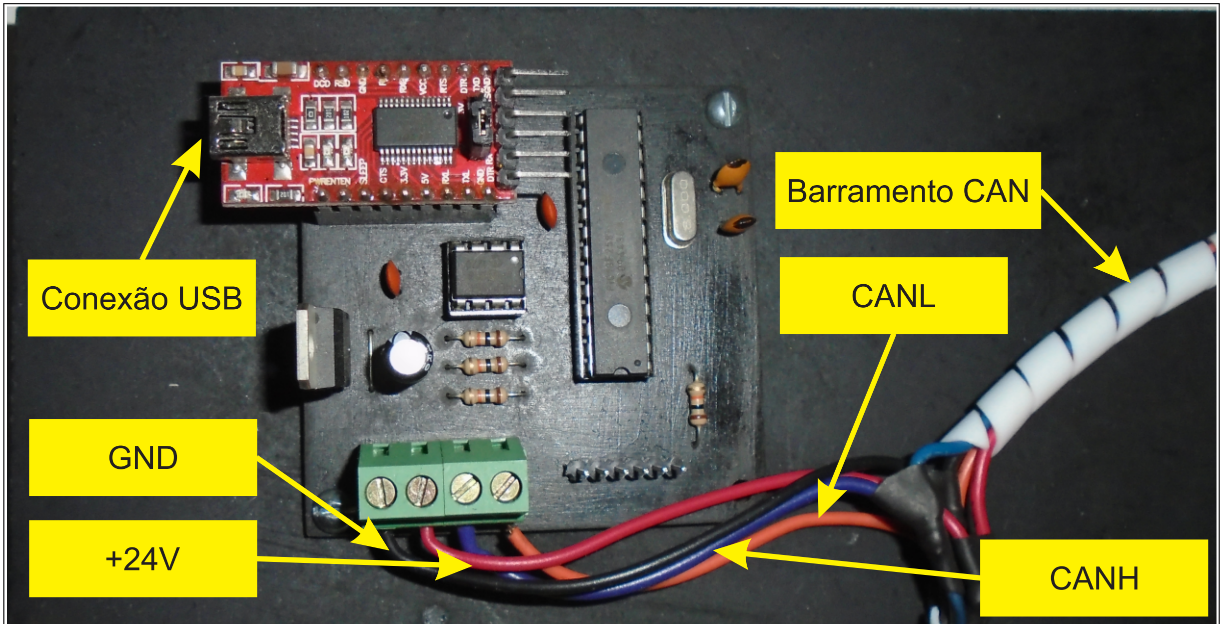
Fonte: Autoria própria.

Foram utilizadas as cores vermelho e preto para simbolizar a alimentação ($24V$ e GND, respectivamente). As cores azul e laranja são referentes à comunicação CAN, sendo a primeira relacionada com CANL e a segunda com CANH.

Ainda com base na figura 38, os sensores (potenciômetros) foram fixos em uma chapa metálica, isolada com tinta de coloração preto fosco. Isto também foi feito para a conexão com a fonte de alimentação.

É possível observar de forma mais detalhada o nó CANALL na figura 95, onde está exposto a conexão USB e barramento CAN.

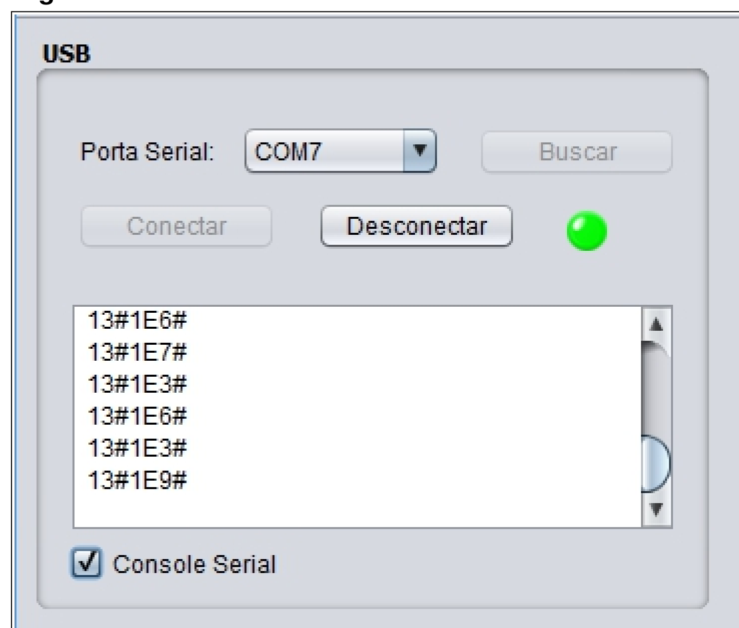
Figura 95 – Ampliação nó CANALL



Fonte: Autoria própria.

Caso o usuário da interface gráfica necessite observar o formato dos dados provenientes do nó CANALL, é necessário ativar a caixa de seleção Console Serial. Na figura 96, é possível observar a presença do caractere delimitador ('#') entre o identificador e o campo de dados.

Figura 96 – Panel USB com console serial habilitado



Fonte: Autoria própria.

Como a interface gráfica CANALL apresenta o fluxo de dados no barramento CAN, é possível receber tal fluxo de maneira integra, ou seja, sem a utilização de filtros. Isto torna a atualização dos valores mostrados na interface mais lenta. A apresentação de tais dados, com formato decimal é mostrada na figura 97.

Figura 97 – Tabela de recebimento de mensagens, sem filtro no formato decimal

CAN

Número de ID's disponíveis: 4 — Atualizar

Filtro — Mask — Filter — Ok

Id	Data	Intervalo (ms)
19	369	62
9	181	48
13	2882343476	2368
15	4393921088525	721

Fonte: Autoria própria.

Com a possibilidade de filtragem de mensagens, optou-se por filtrar as mensagens provenientes do nó Sensor 1 (ID 9). A figura 98 mostra a interface gráfica com valores definidos de máscara (255) e filtro (9).

Figura 98 – Tabela de recebimento de mensagens, com filtro no formato decimal

CAN

Número de ID's disponíveis: 1 — Atualizar

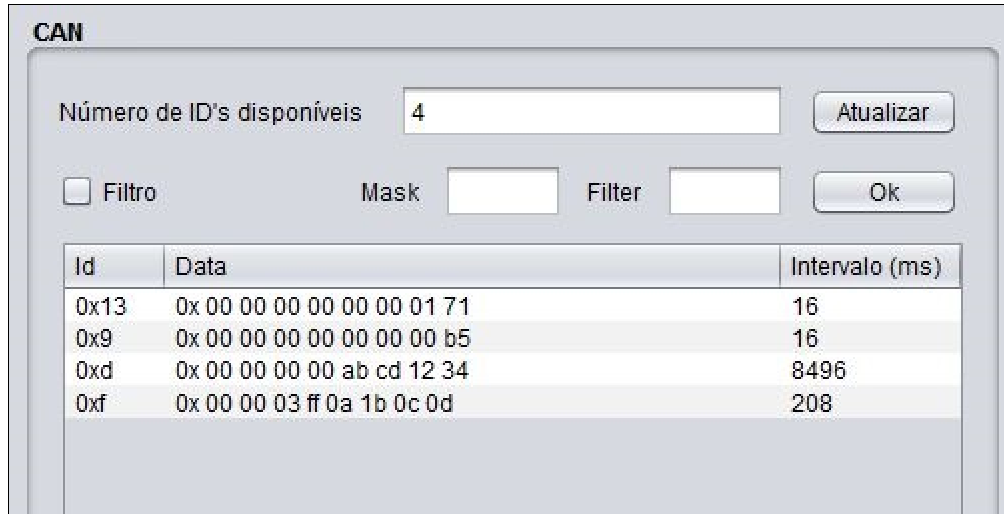
Filtro (marcado) — Mask: 255 — Filter: 9 — Ok

Id	Data	Intervalo (ms)
9	397	18

Fonte: Autoria própria.

Caso o formato numérico selecionado seja o hexadecimal, a tabela de recebimento de dados terá o formato da apresentada na figura 99.

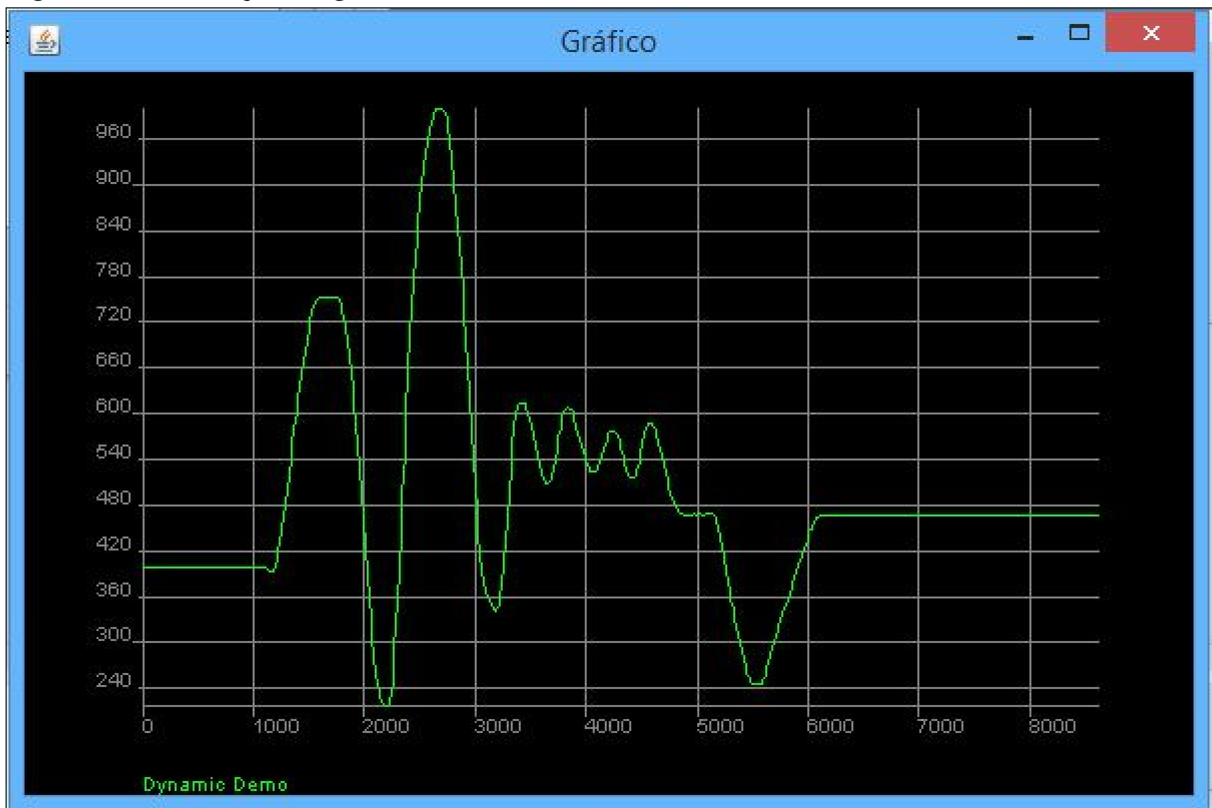
Figura 99 – Tabela de recebimento de mensagens, sem filtro no formato hexadecimal



Fonte: Autoria própria.

Os dados provenientes do barramento CAN podem ser apresentados de forma gráfica, conforme mostra a figura 100.

Figura 100 – Exibição de gráfico habilitada

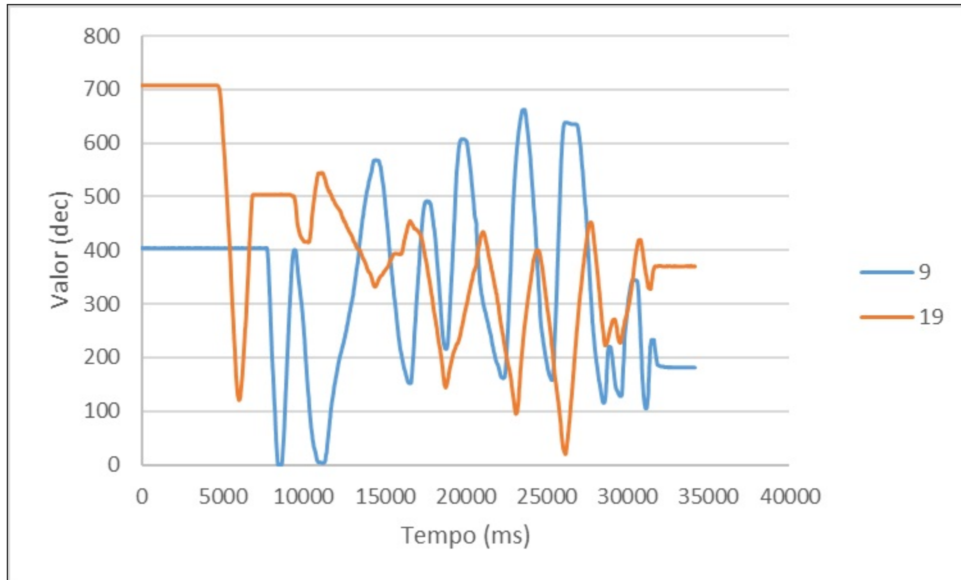


Fonte: Autoria própria.

Conforme dito no capítulo anterior, o tráfego de dados do barramento pode ser salvo em um arquivo de texto, que pode ser mostrado graficamente com o *software*

Excel. Isto é mostrado na figura 101, que apresenta os valores gráficos provenentes dos nós Sensor 1 e Sensor 2.

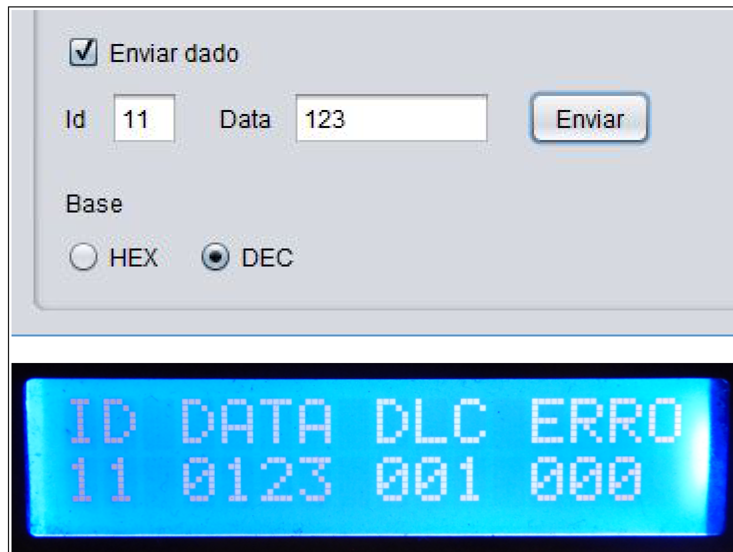
Figura 101 – Gráfico proveniente do arquivo de texto



Fonte: Autoria própria.

O nó LCD recebe apenas mensagens com o ID 11. A figura 102 mostra o valor enviado pela *interface* gráfica e o mesmo valor recebido pelo nó LCD.

Figura 102 – Envio de valores através da interface gráfica



Fonte: Autoria própria.

Afim de mostrar o funcionamento da calculadora de temporização, presente na interface gráfica CANALL, foi incluída a imagem 103 neste documento. Ela apresenta os valores de temporização utilizados por todos os nós do barramento deste projeto.

Figura 103 – Exemplo de funcionamento da calculadora de temporização

Temporização

Frequência (MHz): 8 | Calcular
Bit rate (kbps): 100 | Limpar
Sample point (%): 65

Sync	Prop	PS1	PS2	BTQ	SP(%)	SJW	BRP
1	4	8	7	20	65.0	1	1
1	4	8	7	20	65.0	2	1
1	4	8	7	20	65.0	3	1
1	4	8	7	20	65.0	4	1
1	5	7	7	20	65.0	1	1
1	5	7	7	20	65.0	2	1
1	5	7	7	20	65.0	3	1
1	5	7	7	20	65.0	4	1
1	6	6	7	20	65.0	1	1
1	6	6	7	20	65.0	2	1
1	6	6	7	20	65.0	3	1
1	6	6	7	20	65.0	4	1
1	7	5	7	20	65.0	1	1
1	7	5	7	20	65.0	2	1
1	7	5	7	20	65.0	3	1
1	7	5	7	20	65.0	4	1
1	8	4	7	20	65.0	1	1

Fonte: Autoria própria.

O cálculo da temporização utilizando os valores provenientes da calculadora acima (valores grifados em azul) foi demonstrado na seção 3.2.2.1.

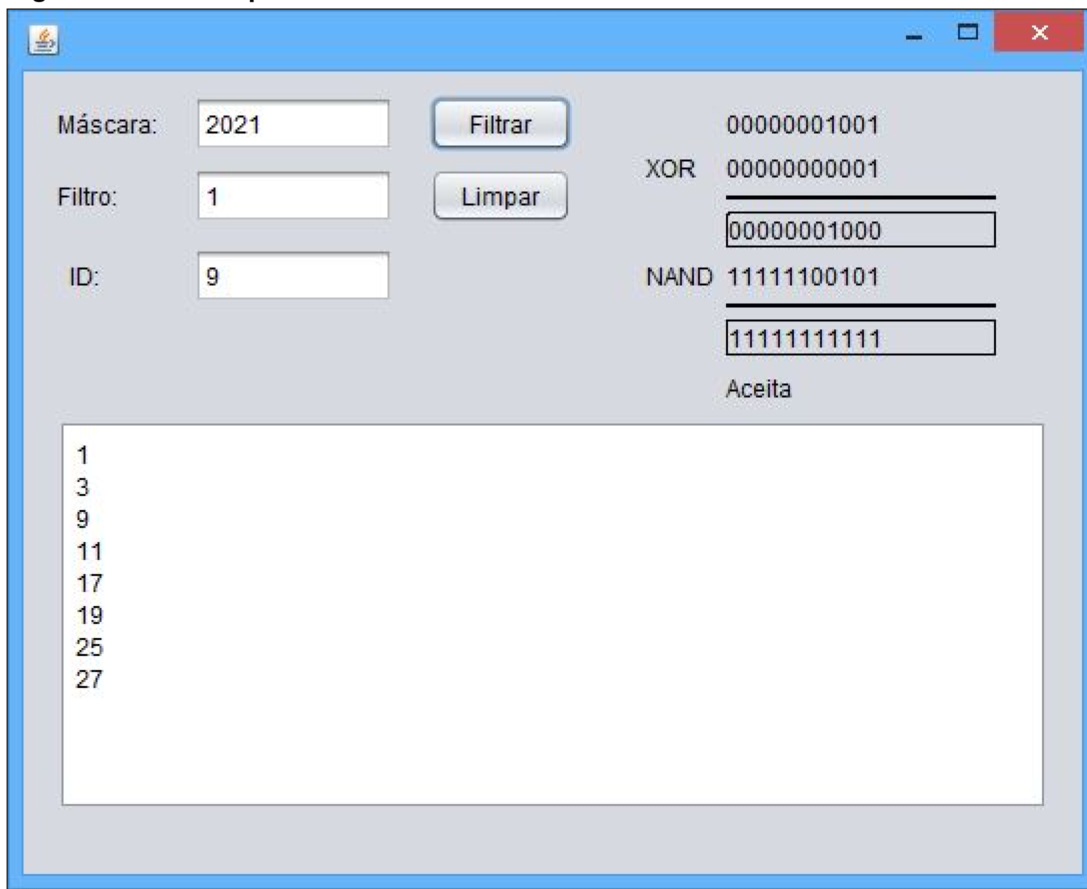
Os valores obtidos na imagem 103 são compatíveis com calculadoras de temporização CAN, como a kvaser (KVASER, 2018).

A calculadora de filtro é colocada à prova testando o valor de máscara e filtro capaz de aceitar as mensagens provenientes do nó Sensor 1 (ID 9) e nó Sensor 2 (ID 19). Uma máscara com valor 2021 e um filtro com valor 1 é capaz disto. A figura 104 mostra que para tais valores, a mensagem é aceita.

A direta dos campos de entrada de dados fornecido pelo usuário da *interface* gráfica, é mostrada a expressão lógica 16, detalhada no capítulo anterior. Pela mesma equação, nota-se que uma mensagem é aceita somente se todos os bits resultantes da expressão lógica tem valor 1, o que resulta na grandeza decimal de 2047 ($2^{11} = 2047$).

Conforme escrito no capítulo desenvolvimento, toda a interface CANALL é compatível com o formato padrão de identificadores, ou seja, 11 *bits*. Valores de iden-

Figura 104 – Exemplo de funcionamento da calculadora de filtro



Fonte: Autoria própria.

tificadores com magnitude maior que 11 *bits* não terão correto funcionamento nesta *interface* gráfica.

5 CONCLUSÃO E PERSPECTIVAS

O objetivo deste trabalho de conclusão de curso era a apresentação do tráfego de dados de um barramento CAN, de autoria própria, em uma interface gráfica utilizando linguagem Java. Como observado no capítulo anterior (Resultados e Discussões), tal objetivo foi alcançado, visto que a *interface* gráfica se mostrou capaz de ser um bom sistema de diagnóstico.

Em primeiro instante, toda a comunicação do barramento CAN foi estabelecida, utilizando apenas o nó LCD e o nó Sensor 1. Estes dois nós foram os grandes responsáveis pelo todo o desenvolvimento da linguagem de programação embarcada nos microcontroladores. Somente após isto, foi desenvolvido o nó CANALL.

O desenvolvimento de todas a bibliotecas utilizadas na programação dos nós permitiu um grande conhecimento de como é o funcionamento do protocolo CAN, além de permitir um grande controle sobre as características da transmissão.

A possibilidade de salvar o tráfego de dados do barramento CAN em um arquivo de texto permite que dados de diferentes identificadores possam ser relacionados.

No desenvolvimento deste projeto, observou-se que a conexão entre a interface gráfica e nó CANALL é o gargalo do projeto, devido a sua baixa taxa de transmissão. Porém o sistema de filtragem de mensagens no hardware (filtro e máscara) permitiu a aquisição de dados em processos rápidos sem perda significativa de rendimento.

A interface gráfica desenvolvida, denominada CANALL, pode ser conectada ao barramento CAN não somente com o conversor FTDI FT232, mas também a um dispositivo USB com conversor à rádio, como os utilizados em telemetria de drones.

O barramento CAN pode ser conectado a qualquer dispositivo que seja compatível com o protocolo CAN 2.0B. Isto foi comprovado com a conexão de um barramento CAN de mesmas características à um conversor de frequência da WEG, modelo CVW300.

Como consideração final, é possível tornar o sistema abordado neste trabalho, ou parte do mesmo em um sistema embarcado para diagnóstico automotivo, seguindo padrões definidos pela Sociedade dos Engenheiros Automotivos (SAE).

REFERÊNCIAS

AUTODESK. **EAGLE Vs. EAGLE Premium**. [S.l.], 2018 (Acesso em 1 set. 2018). Disponível em: <https://www.autodesk.com/products/eagle/compare>. Citado na página 65.

BOSCH, Robert et al. Can specification version 2.0. **Rober Bousch GmbH, Postfach**, v. 300240, p. 72, 1991. Citado 4 vezes nas páginas 30, 33, 46 e 83.

CAN IN AUTOMATION. Cia ds-102. **CAN Physical Layer for CiA DS-201, CAN Reference Model**, 1996. Citado na página 41.

____. **History of CAN technology**. 2015. Disponível em: <https://www.can-cia.org/can-knowledge/can/can-history/>. Citado na página 29.

DEITEL, Harvey M; DEITEL, Paul J. **Java: Como Programar, 6a. Edição**. São Paulo: Prentice Hall, 2010. Citado na página 62.

DEVMEDIA. **Introdução ao Java Virtual Machine (JVM)**. 2013. Disponível em: <https://www.devmedia.com.br/introducao-ao-java-virtual-machine-jvm/27624>. Citado na página 61.

____. **Os 4 pilares da Programação Orientada a Objetos**. 2013. Disponível em: <https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>. Citado na página 63.

FOROUZAN, Behrouz A. **Comunicação de dados e redes de computadores**. São Paulo: AMGH Editora, 2009. Citado na página 21.

FT232R, USB. Uart ic datasheet. **Future Technology Devices International Ltd**, 2005. Citado na página 43.

IBRAHIM, Dogan. **PIC Microcontroller Projects in C**. London: Newnes, 2014. Citado na página 36.

INFORMATIK, Vector. **Introduction to CAN - Standardization**. 2018 (Acesso em 21 jun. 2018). Disponível em: <https://elearning.vector.com/mod/page/view.php?id=335>. Citado na página 40.

INTERNATIONAL STANDARDIZATION ORGANIZATION. **ISO 11898 - 1: Road vehicles — Controller area network (CAN) - Part 1**. 2003. Citado 3 vezes nas páginas 29, 40 e 123.

____. **ISO 11898 - 2: Road vehicles — Controller area network (CAN) - Part 2**. 2003. Citado 2 vezes nas páginas 30 e 50.

JCHART2D. **What is JChart2D?** 2011. Disponível em: <http://jchart2d.sourceforge.net/index.shtml>. Citado na página 122.

KEMIGHAN, Brian W; RITCHIE, Dennis M. **The C programming language**. Englewood Cliffs, New Jersey: Prentice Hall, 1978. Citado na página 54.

KVASER. **CAN Bus Bit Timing Calculator**. 2018. Disponível em: <https://www.kvaser.com/support/calculators/bit-timing-calculator/>. Citado na página 134.

MICROCHIP TECHNOLOGY INC. **In-Circuit Serial Programming (ICSP)**. 1997. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/31028a.pdf>. Citado na página 60.

____. **PIC18FXX8 Datasheet: 28/40-Pin High-Performance, Enhanced Flash Microcontrollers with CAN Module**. 2006. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/41159e.pdf>. Citado na página 44.

____. **MCP2551 Datasheet: High-Speed CAN Transceiver**. 2010. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/20001667G.pdf>. Citado na página 50.

____. **PICkit 3 Programmer/Debugger Users Guide**. 2010. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/51795B.pdf>. Citado na página 59.

____. **MPLAB XC8 C Compiler User's Guide**. 2012. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/50002053G.pdf>. Citado na página 61.

____. **MPLAB X IDE User's Guide**. 2015. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/50002027D.pdf>. Citado na página 59.

NETBEANS. **NetBeans IDE Features**. 2018. Disponível em: <https://netbeans.org/features/index.html>. Citado na página 64.

ORACLE CORPORATION. **The Java Virtual Machine Specification**. 2013. Disponível em: <https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>. Citado na página 61.

____. **Primitive Data Types**. 2017. Disponível em: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>. Citado na página 61.

____. **Class SerialPort**. 2018. Disponível em: <https://docs.oracle.com/cd/E17802_01/products/products/javacomm/reference/api/javax/comm/SerialPort.html>. Citado na página 117.

____. **Java SE Development Kit 11 Downloads**. 2018. Disponível em: <https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.htmll>. Citado na página 63.

RICHARDS, Pat. An28: A can physical layer discussion. **Microchip Technology Incorporated**, 2002. Citado na página 50.

____. An754: Understanding microchip can module bit timing. **Microchip Technology Incorporated**, v. 13, 2005. Citado na página 38.

RXTX. **Welcome to the RXTX wiki**. 2011. Disponível em: <http://rxtx.qbang.org/wiki/index.php/Main_Page>. Citado na página 117.

STACK OVERFLOW. **Stack Overflow Developer Survey 2018**. 2018. Disponível em: <https://insights.stackoverflow.com/survey/2018/>. Citado na página 53.

STRANGIO, Christopher E. The rs232 standard. **Online Document**, v. 10, 2006. Citado na página 42.

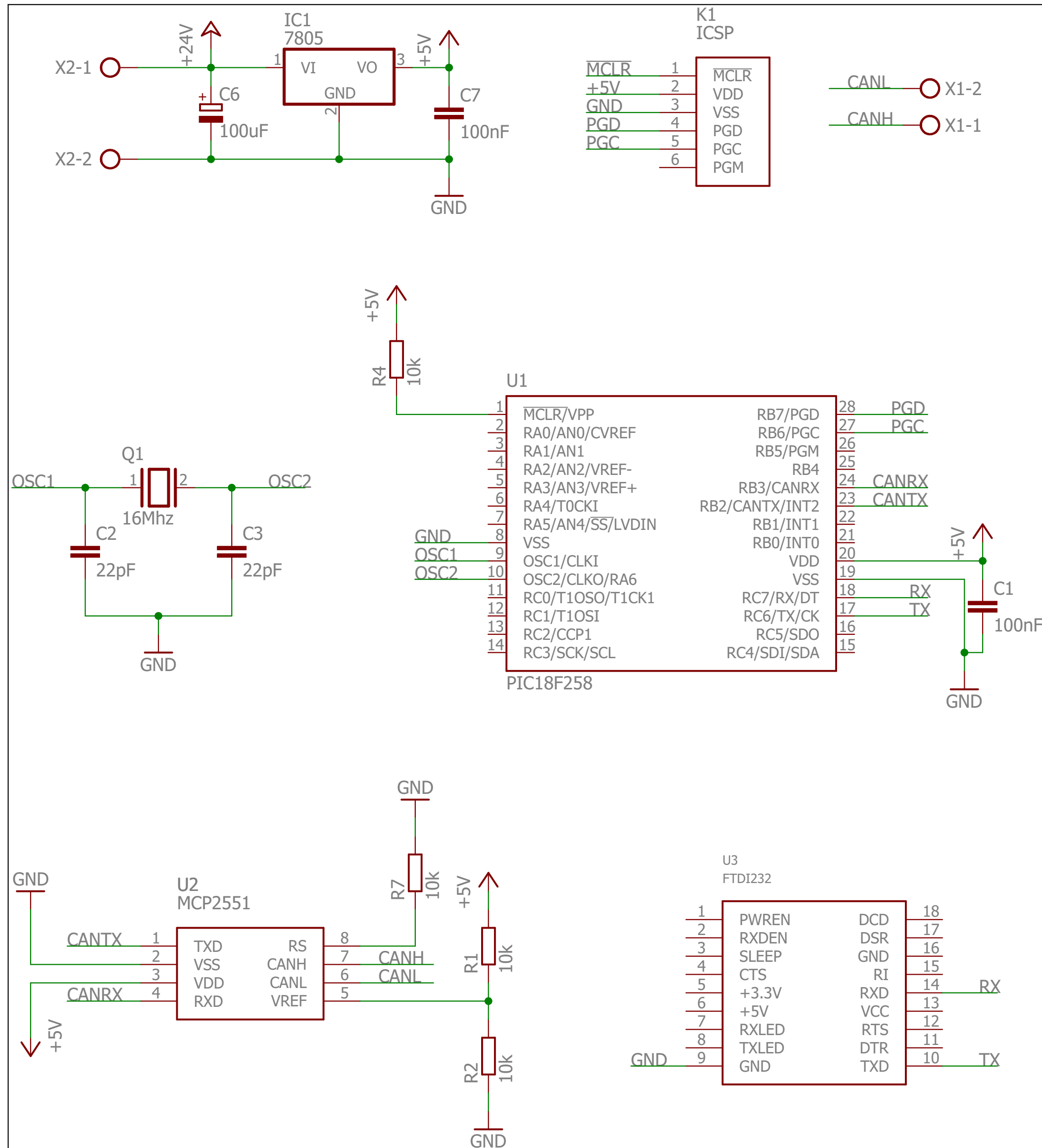
ZANCO, Wagner da Silva. **Microcontroladores PIC – Técnicas de software e hardware para projetos de circuitos eletrônicos**. São Paulo: Érica, 2006. Citado na página 44.

ZIMMERMANN, HUBERT. Osi reference model—the iso model of architecture for open systems interconnection. **IEEE TRANSACTIONS ON COMMUNICATIONS**, v. 28, n. 4, p. 425, 1980. Citado na página 27.

APÊNDICES

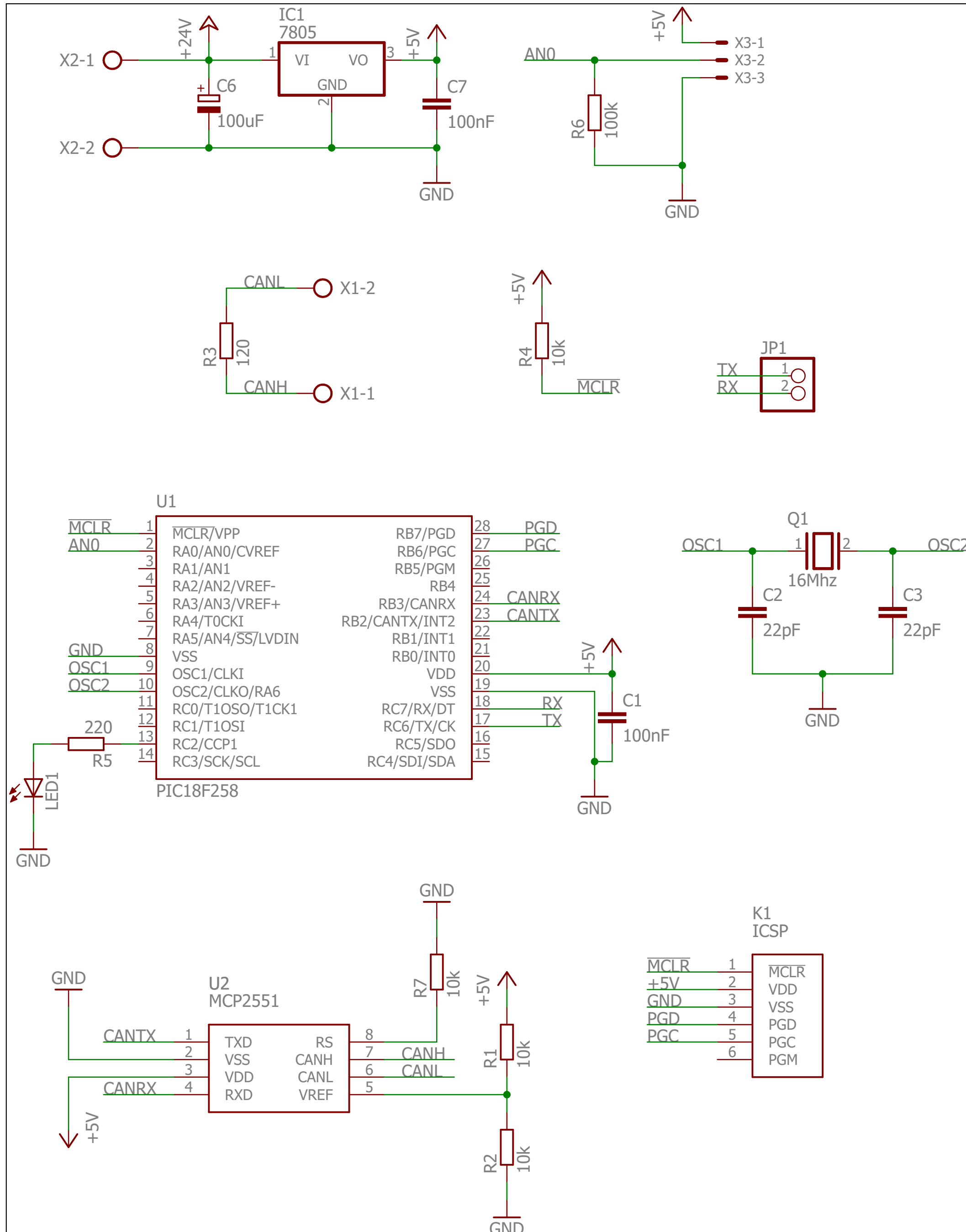
APÊNDICE A – ESQUEMÁTICO DOS NÓS

Figura 105 – Esquemático nó CANALL



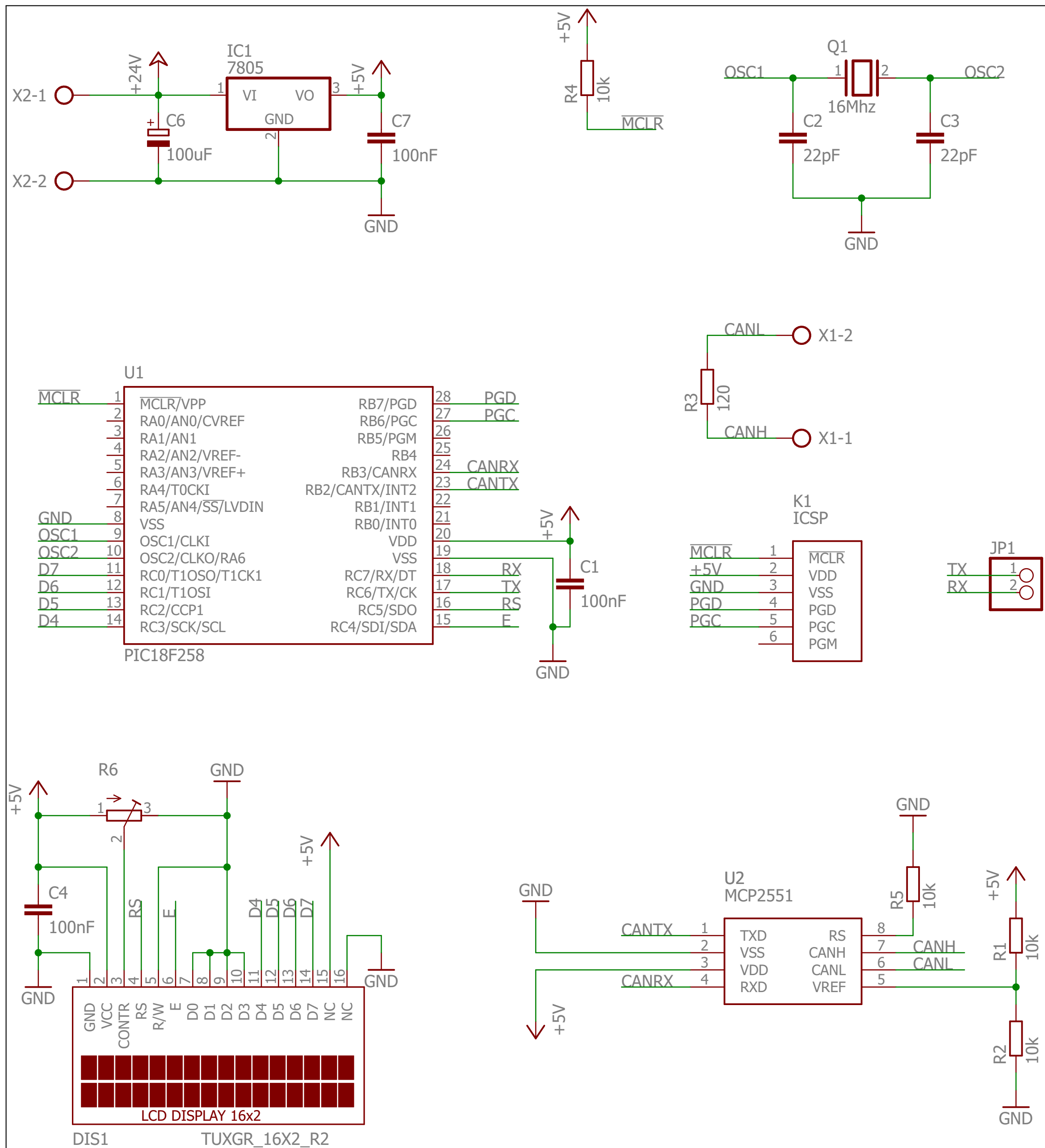
Fonte: Colocar fonte

Figura 106 – Esquemático nó Sensor



Fonte: Colocar fonte

Figura 107 – Esquemático nó LCD



Fonte: Colocar fonte

APÊNDICE B – BIBLIOTECAS DESENVOLVIDAS

B.0.1 adc.h

```c
void adc_Init()
{
    TRISA = 0xFF;
    ADCON1bits.ADCS2 = 1;
    ADCON0bits.ADCS1 = 0;
    ADCON0bits.ADCS0 = 1;
    ADCON0bits.GO_DONE = 0;
    ADCON0bits.ADON = 1;
    ADCON1bits.PCFG = 0b0000;
    ADCON1bits.ADFM = 1;
}

unsigned int adc_Read(char channel)
{
    ADCON0bits.CHS = channel & 0x07;
    __delay_us(10);
    ADCON0bits.GO_DONE = 1;
    while(ADCON0bits.GO_DONE);
    __delay_us(10);
    return  (ADRESH << 8) | ADRESL;
}
```

B.0.2 can.h

```c
#define CONFIG_MODE         0x04
#define LISTEN_ONLY_MODE    0x03
#define LOOPBACK_MODE       0x02
#define DISABLE_MODE        0x01
#define NORMAL_MODE         0x00
#define F0_B0               0x00
#define F1_B0               0x01
#define F2_B1               0x02
#define F3_B1               0x03
#define F4_B1               0x04
#define F5_B1               0x05

void Can_Set_Mode(unsigned char param)
{
    CANCONbits.REQOP2 = (0x04 & param) >> 2;
    CANCONbits.REQOP1 = (0x02 & param) >> 1;
    CANCONbits.REQOP0 = 0x01 & param;
    if(param == CONFIG_MODE)
    {
        while(CANSTATbits.OPMODE2 != 1);
    }
    else
    {
        while(CANSTATbits.OPMODE2 != 0);
    }
}

void Can_Init(void)
{
    // I/O
    TRISBbits.RB3 = 1;
    TRISBbits.RB2 = 0;

    // CAN CONTROL AND STATUS REGISTERS
    // Abort All Pending Transmissions bit
    CANCONbits.ABAT = 1;

    // CAN BAUD RATE REGISTERS - 100 kbps
    // BRP = 1 (Valor do registrador)
    BRGCON1bits.BRP5 = 0;
    BRGCON1bits.BRP4 = 0;
    BRGCON1bits.BRP3 = 0;
    BRGCON1bits.BRP2 = 0;
    BRGCON1bits.BRP1 = 0;
    BRGCON1bits.BRP0 = 1;
    // SJW = 1
    BRGCON1bits.SJW1 = 0;
    BRGCON1bits.SJW0 = 0;
    // Sync_Seg = 1, Fixo
    // Prop_Seg = 6
    BRGCON2bits.PRSEG2 = 1;
    BRGCON2bits.PRSEG1 = 0;
    BRGCON2bits.PRSEG0 = 1;
    // Phase_Seg1 = 6
    BRGCON2bits.SEG1PH2 = 1;
    BRGCON2bits.SEG1PH1 = 0;
```

```
57      BRGCON2bits.SEG1PH0 = 1;
        // Sample Point Occurs Here
59      // Phase_Seg2 = 7
        BRGCON3bits.SEG2PH2 = 1;
61      BRGCON3bits.SEG2PH1 = 1;
        BRGCON3bits.SEG2PH0 = 0;
63
        // OTHER BAUD RATE CONFIGURATION
65      // Phase_Seg2 Freely programmable
        BRGCON2bits.SEG2PHTS = 1;
67      // Bus line is sampled three times
        BRGCON2bits.SAM = 1;
69      // Wake-up filter is not used
        BRGCON3bits.WAKFIL = 0;
71      // CANTX pin will drive VDD when recessive
        CIOCONbits.ENDRHI = 1;
73      // Disable CAN capture
        CIOCONbits.CANCAP = 0;
75
        // CAN INTERRUPT REGISTERS
77      // All interrups disabled
        PIE3 = 0x00;
79
        // CAN TRANSMIT BUFFER 0 REGISTERS
81      // Buffer 0 Priority Level 3
        TXB0CONbits.TXPRI1 = 1;
83      TXB0CONbits.TXPRI0 = 1;

85      // CAN TRANSMIT BUFFER 1 REGISTERS
        // Buffer 1 Priority Level 2
87      TXB1CONbits.TXPRI1 = 1;
        TXB1CONbits.TXPRI0 = 0;
89
        // CAN TRANSMIT BUFFER 2 REGISTERS
91      // Buffer 2 Priority Level 0
        TXB2CONbits.TXPRI1 = 0;
93      TXB2CONbits.TXPRI0 = 0;

95      // CAN RECEIVE BUFFER 0 REGISTERS
        // Receive only valid messages with standard identifier
97      RXB0CONbits.RXM1 = 0;
        RXB0CONbits.RXM0 = 1;
99      // No receive Buffer 0 overflow to receive Buffer 1
        RXB0CONbits.RXB0DBEN = 0;
101     // Mask and filters inicialization
        // Mask 0, Buffer 0
103     RXM0SIDH = 0xFF;
        RXM0SIDL = 0xFF;
105     // Filter 0, Buffer 0, standard messages
        RXF0SIDH = 0x00;
107     RXF0SIDL = 0x00;
        // Filter 1, Buffer 0, standard messages
109     RXF1SIDH = 0x00;
        RXF1SIDL = 0x00;
111
        // CAN RECEIVE BUFFER 1 REGISTERS
113     // Receive only valid messages with standard identifier
        RXB1CONbits.RXM1 = 0;
115     RXB1CONbits.RXM0 = 1;
        // Mask and filters inicialization
```

```
117     // Mask 1, Buffer 1
        RXM1SIDH = 0xFF;
119     RXM1SIDL = 0xFF;
        // Filter 2, Buffer 1, standard messages
121     RXF2SIDH = 0x00;
        RXF2SIDL = 0x00;
123     // Filter 3, Buffer 1, standard messages
        RXF3SIDH = 0x00;
125     RXF3SIDL = 0x00;
        // Filter 4, Buffer 1, standard messages
127     RXF4SIDH = 0x00;
        RXF4SIDL = 0x00;
129     // Filter 5, Buffer 1, standard messages
        RXF5SIDH = 0x00;
131     RXF5SIDL = 0x00;
    }
133
    void Can_Set_Mask(unsigned char rx_buffer, unsigned int mask)
135 {
        if(rx_buffer == 0)
137     {
            // Mask 0, Buffer 0
139         RXM0SIDH = (mask & 0x7F8) >> 3;
            RXM0SIDL = (mask & 0x0007) << 5;
141     }
        if(rx_buffer == 1)
143     {
            // Mask 1, Buffer 1
145         RXM1SIDH = (mask & 0x7F8) >> 3;
            RXM1SIDL = (mask & 0x0007) << 5;
147     }
    }
149
    void Can_Set_Filter(unsigned char option, unsigned int filter)
151 {
        // BUFFER 0
153     if(option == F0_B0)
        {
155         // Filter 0, Buffer 0, standard messages
            RXF0SIDH = (filter & 0x7F8) >> 3;
157         RXF0SIDL = (filter & 0x0007) << 5;
        }
159     if(option == F1_B0)
        {
161         // Filter 1, Buffer 0, standard messages
            RXF1SIDH = (filter & 0x7F8) >> 3;
163         RXF1SIDL = (filter & 0x0007) << 5;
        }
165     // BUFFER 1
        if(option == F2_B1)
167     {
            // Filter 2, Buffer 1, standard messages
169         RXF2SIDH = (filter & 0x7F8) >> 3;
            RXF2SIDL = (filter & 0x0007) << 5;
171     }
        if(option == F3_B1)
173     {
            // Filter 3, Buffer 1, standard messages
175         RXF3SIDH = (filter & 0x7F8) >> 3;
            RXF3SIDL = (filter & 0x0007) << 5;
```

```
177     }
        if(option == F4_B1)
179     {
            // Filter 4, Buffer 1, standard messages
181         RXF4SIDH = (filter & 0x7F8) >> 3;
            RXF4SIDL = (filter & 0x0007) << 5;
183     }
        if(option == F5_B1)
185     {
            // Filter 5, Buffer 1, standard messages
187         RXF5SIDH = (filter & 0x7F8) >> 3;
            RXF5SIDL = (filter & 0x0007) << 5;
189     }
    }
191
    void Can_Set_ID(unsigned char tx_buffer, unsigned int id)
193 {
        if(tx_buffer == 0)
195     {
            TXB0SIDH = (id & 0x7F8) >> 3;
197         TXB0SIDL = (id & 0x0007) << 5;
            // Buffer 0 Standard Identifier
199         TXB0SIDLbits.EXIDE = 0;
        }
201     if(tx_buffer == 1)
        {
203         TXB1SIDH = (id & 0x7F8) >> 3;
            TXB1SIDL = (id & 0x0007) << 5;
205         // Buffer 1 Standard Identifier
            TXB1SIDLbits.EXIDE = 0;
207     }
        if(tx_buffer == 2)
209     {
            TXB2SIDH = (id & 0x7F8) >> 3;
211         TXB2SIDL = (id & 0x0007) << 5;
            // Buffer 2 Standard Identifier
213         TXB2SIDLbits.EXIDE = 0;
        }
215 }

217 void Can_Write(unsigned char tx_buffer, unsigned char tx_dlc,
       unsigned char *data)
    {
219     // PIC 18f258 has three transmit buffers
        if(tx_buffer == 0)
221     {
            // Data Length Code, TXRTR bit cleared
223         TXB0DLC = tx_dlc & 0x0F;

225         // Transmit Buffer 0 Data field
            TXB0D7 = data[7];
227         TXB0D6 = data[6];
            TXB0D5 = data[5];
229         TXB0D4 = data[4];
            TXB0D3 = data[3];
231         TXB0D2 = data[2];
            TXB0D1 = data[1];
233         TXB0D0 = data[0];

235         // Transmit Request Status Bit
```

```
        TXB0CONbits.TXREQ = 1;
        while(TXB0CONbits.TXREQ == 1);
    }
    if(tx_buffer == 1)
    {
        // Data Length Code, TXRTR bit cleared
        TXB1DLC = tx_dlc & 0x0F;

        // Transmit Buffer 1 Data field
        TXB1D7 = data[7];
        TXB1D6 = data[6];
        TXB1D5 = data[5];
        TXB1D4 = data[4];
        TXB1D3 = data[3];
        TXB1D2 = data[2];
        TXB1D1 = data[1];
        TXB1D0 = data[0];

        // Transmit Request Status Bit
        TXB1CONbits.TXREQ = 1;
        while(TXB1CONbits.TXREQ == 1);
    }
    if(tx_buffer == 2)
    {
        // Data Length Code, TXRTR bit cleared
        TXB2DLC = tx_dlc & 0x0F;

        // Transmit Buffer 1 Data field
        TXB2D7 = data[7];
        TXB2D6 = data[6];
        TXB2D5 = data[5];
        TXB2D4 = data[4];
        TXB2D3 = data[3];
        TXB2D2 = data[2];
        TXB2D1 = data[1];
        TXB2D0 = data[0];

        // Transmit Request Status Bit
        TXB2CONbits.TXREQ = 1;
        while(TXB2CONbits.TXREQ == 1);
    }
}

void Can_Read(unsigned char *data, unsigned int *rx_id,
   unsigned char *dlc, unsigned char *error_count)
{
    // PIC 18f258 has two receive buffers

    unsigned int id_high, id_low;

    // RECEIVE BUFFER 0
    if(RXB0CONbits.RXFUL == 1)
    {
        RXB0SIDLbits.EXID = 0;
        id_high = (RXB0SIDH << 3) & 0x7F8;
        id_low = (RXB0SIDL >> 5) & 0x0007;
        *rx_id = id_high | id_low;

        data[7] = RXB0D7;
        data[6] = RXB0D6;
```

```
295         data[5] = RXB0D5;
            data[4] = RXB0D4;
297         data[3] = RXB0D3;
            data[2] = RXB0D2;
299         data[1] = RXB0D1;
            data[0] = RXB0D0;
301
            *dlc = RXB0DLC & 0x0F;
303         *error_count = RXERRCNT;

305         RXB0CONbits.RXFUL = 0;
        }
307
        // RECEIVE BUFFER 1
309     if(RXB1CONbits.RXFUL == 1)
        {
311         RXB1SIDLbits.EXID = 0;
            id_high = (RXB1SIDH << 3) & 0x7F8;
313         id_low = (RXB1SIDL >> 5) & 0x0007;
            rx_id = id_high | id_low;
315
            data[7] = RXB1D7;
317         data[6] = RXB1D6;
            data[5] = RXB1D5;
319         data[4] = RXB1D4;
            data[3] = RXB1D3;
321         data[2] = RXB1D2;
            data[1] = RXB1D1;
323         data[0] = RXB1D0;

325         *dlc = RXB0DLC & 0x0F;
            *error_count = RXERRCNT;
327
            RXB1CONbits.RXFUL = 0;
329     }
}
```

B.0.3 config.h

```
// CONFIG1H
#pragma config OSC = HS
#pragma config OSCS = OFF

// CONFIG2L
#pragma config PWRT = ON
#pragma config BOR = ON
#pragma config BORV = 25

// CONFIG2H
#pragma config WDT = OFF
#pragma config WDTPS = 2

// CONFIG4L
#pragma config STVR = ON
#pragma config LVP = OFF

// CONFIG5L
#pragma config CP0 = OFF
#pragma config CP1 = OFF
#pragma config CP2 = OFF
#pragma config CP3 = OFF

// CONFIG5H
#pragma config CPB = OFF
#pragma config CPD = OFF

// CONFIG6L
#pragma config WRT0 = OFF
#pragma config WRT1 = OFF
#pragma config WRT2 = OFF
#pragma config WRT3 = OFF

// CONFIG6H
#pragma config WRTC = OFF
#pragma config WRTB = OFF
#pragma config WRTD = OFF

// CONFIG7L
#pragma config EBTR0 = OFF
#pragma config EBTR1 = OFF
#pragma config EBTR2 = OFF
#pragma config EBTR3 = OFF

// CONFIG7H
#pragma config EBTRB = OFF

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#define _XTAL_FREQ 8000000
#include <xc.h>
```

B.0.4 conversions.h

```
void toCharArray(unsigned char *numArray, unsigned long num,
   unsigned int base)
{
    unsigned long value;
    unsigned long rem = 0;     // Resto (remainder)
    unsigned int i = 0;       // Contador
    unsigned int j = 0;       // Contador
    unsigned char str[16];
    value = num;
    if(value == 0)
    {
        numArray[0] = '0';
        numArray[1] = '\0';
    }
    if(value != 0)
    {
        while(value != 0)
        {
            rem = value%base;
            value = value/base;
            if(rem > 9)
            {
                str[i] = (rem-10) + 65; // 65 = 'A'
            }
            if(rem <= 9)
            {
                str[i] = rem + 48; // 48 = '0'
            }
            i++;
        }
        for(j=0; j<i; j++)
        {
            numArray[j] = str[i-j-1];
        }
        numArray[j] = '\0';
    }
}

void toCharArrayFixedDigits(unsigned char *numArray, unsigned
   long num, unsigned int base, unsigned int numDig)
{
    unsigned long value;
    unsigned long rem = 0;     // Resto (remainder)
    unsigned int i = 0;       // Contador
    unsigned int j = 0;       // Contador
    unsigned char str[16];
    value = num;
    while(i < numDig)
    {
        rem = value%base;
        value = value/base;
        if(rem > 9)
        {
            str[i] = (rem-10) + 65; // 65 = 'A'
        }
        if(rem <= 9)
```

```
            {
                str[i] = rem + 48; // 48 = '0'
            }
            i++;
            for(j=0; j<i; j++)
            {
                numArray[j] = str[i-j-1];
            }
            numArray[j] = '\0';
        }
}

unsigned int stringToInt(unsigned char *numArray, unsigned int
    base)
{
    unsigned int value = 0;
    unsigned int i=0;
    for(i=0; numArray[i] != '\0'; i++)
    {
      if(numArray[i] >= 48 && numArray[i] <= 57){
            value = base*value + (numArray[i] - 48);
   }
           if(numArray[i] >= 65 && numArray[i] <= 70){
            value = base*value + (10 + numArray[i] - 65);
   }
    }
    return value;
}

void split(unsigned char *idData, unsigned int *idNumber,
    unsigned int *dataNumber)
{
  unsigned int i = 0;
  unsigned int j = 0;
  unsigned int pos[2];
  unsigned char id[5];
  unsigned char data[16];
  for(i=0; idData[i]!='\0'; i++)
  {
    if(idData[i]=='#')
    {
      pos[j]=i;
      j++;
    }
  }
  for(i=0;i<pos[0];i++)
  {
    id[i]=idData[i];
  }
  id[i]='\0';
  for(i=pos[0]+1, j=0;i<pos[1];i++, j++)
  {
    data[j]=idData[i];
  }
  data[j]='\0';
  *idNumber = stringToInt(id,16);
  *dataNumber = stringToInt(data,16);
}
```

B.0.5 lcd.h

```
#define RS RC5
#define EN RC4
#define D4 RC3
#define D5 RC2
#define D6 RC1
#define D7 RC0

#define LCD_CLEAR                  0x01
#define LCD_RETURN_HOME            0x02
#define LCD_CURSOR_ON              0x0E
#define LCD_CURSOR_OFF             0x0C
#define LCD_BLINK_CURSOR_ON        0x0F
#define LCD_BLINK_CURSOR_OFF       0x0E
#define LCD_SHIFT_LEFT             0x18
#define LCD_SHIFT_RIGHT            0x1C

void Send_Nibble(char nibble)
{
    D4 = nibble & 0x01;
    D5 = (nibble & 0x02) >> 1;
    D6 = (nibble & 0x04) >> 2;
    D7 = (nibble & 0x08) >> 3;
    EN = 1;
    __delay_us(40);
    EN = 0;
}

void Send_Byte(char instruction)
{
    char nibbleHigh, nibbleLow;
    nibbleLow = instruction & 0x0F;
    nibbleHigh = (instruction & 0xF0) >> 4;
    Send_Nibble(nibbleHigh);
    Send_Nibble(nibbleLow);
}

void Lcd_Init(void)
{
    TRISCbits.RS = 0;
    TRISCbits.EN = 0;
    TRISCbits.D4 = 0;
    TRISCbits.D5 = 0;
    TRISCbits.D6 = 0;
    TRISCbits.D7 = 0;

    PORTCbits.RS = 0;
    PORTCbits.EN = 0;
    PORTCbits.D4 = 0;
    PORTCbits.D5 = 0;
    PORTCbits.D6 = 0;
    PORTCbits.D7 = 0;

    __delay_ms(15);

    Send_Nibble(0x30);__delay_us(4500);
    Send_Nibble(0x03);__delay_us(100);
```

```
    Send_Byte(0x32);__delay_ms(3);
    Send_Byte(0x28);__delay_us(40);
    Send_Byte(0x08);__delay_us(40);
    Send_Byte(0x01);__delay_ms(2);
    Send_Byte(0x06);__delay_us(40);
    Send_Byte(0x0C);__delay_us(40);
}

void Lcd_Set_Cursor(char row, char column)
{
    char position = 0;
    RS = 0;
    EN = 0;
    if(row == 0)
    {
        position = 0x80 + column;
        Send_Byte(position);
    }
    if(row == 1)
    {
        position = 0xC0 + column;
        Send_Byte(position);
    }
    if(row == 2)
    {
        position = 0x94 + column;
        Send_Byte(position);
    }
    if(row == 3)
    {
        position = 0xD4 + column;
        Send_Byte(position);
    }
    __delay_us(40);
}

void Lcd_Write_Char(char caractere)
{
    RS = 1;
    EN = 0;
    Send_Byte(caractere);
}

void Lcd_Write_String(char *text)
{
    char count;
    for(count = 0; text[count] != '\0'; count++)
    {
        Lcd_Write_Char(text[count]);
    }
}

void Lcd_Cmd(char instruction)
{
    RS = 0;
    EN = 0;
    Send_Byte(instruction);
}
```

B.0.6 USART.h

```
void Usart_Init(unsigned char spbrg, unsigned char brgh)
{
    // I/O
    TRISCbits.RC6 = 0;
    TRISCbits.RC7 = 1;

    // TRANSMIT STATUS AND CONTROL REGISTER
    // Clock Source Select bit
    TXSTAbits.CSRC = 0;
    // 8 bit transmission
    TXSTAbits.TX9 = 0;
    // Transmit enabled
    TXSTAbits.TXEN = 1;
    // Asynchronous mode
    TXSTAbits.SYNC = 0;
    // 9th bit without use
    TXSTAbits.TX9D = 0;

    // RECEIVE STATUS AND CONTROL REGISTER
    // Serial Port enabled
    RCSTAbits.SPEN = 1;
    // 8-bit reception
    RCSTAbits.RX9 = 0;
    // Enable continuous receive
    RCSTAbits.CREN = 1;
    // Disable address detection
    RCSTAbits.ADDEN = 0;

    // USART INTERRUPTS
    // Disable transmit, turn on the receive interrupt
    PIE1bits.RCIE = 1;
    PIE1bits.TXIE = 0;

    // BAUD RATE
    SPBRG = spbrg;
    TXSTAbits.BRGH = brgh;

}

void Usart_Write(char *msg)
{
    unsigned int i;

    for(i = 0; msg[i] != '\0'; i++)
    {
        // USART Transmit Register
        TXREG = msg[i];
        // Wait the Transmit Shift Register Status bit (TSR)
   stays empty
        while(TXSTAbits.TRMT == 0);
    }
}

void Usart_Read(unsigned char *msg)
{
    unsigned int i = 0;
```

```
56      while(RCREG != '\0')
        {
58          while(PIR1bits.RCIF == 0);
            msg[i] = RCREG;
60          i++;
        }
62      msg[i] = '\0';
    }
64
    unsigned char Usart_Char_Read()
66  {
        unsigned char msg;
68    if(PIR1bits.RCIF == 1)
      {
70      msg = RCREG;
      }
72    return msg;
    }
```

APÊNDICE C – CÓDIGOS FONTE DOS NÓS

C.0.1 CANALL.c

```
#include "conversions.h"
#include "config.h"
#include "CAN.h"
#include "USART.h"

// Variaveis USART
// Temporizacao - 38400 bauds
char spbrg = 12;
char brgh = 1;
// Dados
unsigned char usartDataOut[9];
unsigned char usartIdOut[5];
unsigned char usartRx[9];
unsigned char usartMask[5];
unsigned char usartFilter[5];
unsigned char txFlag = 0;
unsigned char aux = 0;

// Variaveis CAN
// Temporizacao - 100 kbps
char sjw = 1;
char brp = 1;
char propseg = 6;
char phseg1 = 6;
char phseg2 = 7;
// Dados
union data{
  unsigned char charType[8]; // 8x8 bits
  unsigned int  wordType[4]; // 4x16 bits
    unsigned long longType[2]; // 2x32 bits
};
union data canDataIn;
union data canDataOut;
unsigned int canIdOut;
unsigned int canIdIn = 0;
unsigned char canDlc = 0;
unsigned char canErrorCount = 0;
unsigned int  canMask = 0;
unsigned int  canFilter = 0;

interrupt void getData(void)
{
    if(PIR1bits.RCIF == 1)
    {
        switch (RCREG)
        {
            case 'R':
                Can_Set_Mode(CONFIG_MODE);
                Usart_Read(usartRx);
                usartMask[0] = usartRx[0];
                usartMask[1] = usartRx[1];
                usartMask[2] = usartRx[2];
```

```
                usartMask[3] = usartRx[3];
                usartMask[4] = '\0';
                usartFilter[0] = usartRx[4];
                usartFilter[1] = usartRx[5];
                usartFilter[2] = usartRx[6];
                usartFilter[3] = usartRx[7];
                usartFilter[4] = '\0';

                canMask = stringToInt(usartMask, 10);     l
                canFilter = stringToInt(usartFilter, 10);
                Can_Set_Mask(0, canMask);
                Can_Set_Filter(0, canFilter);
                Can_Set_Mode(NORMAL_MODE);
                break;

            case 'S':
                Can_Set_Mode(CONFIG_MODE);
                Can_Set_Mask(0, 0x0000);
                Can_Set_Filter(0, 0x0000);
                Can_Set_Mode(NORMAL_MODE);
                break;

            case 'T':
                txFlag = 1;
                Usart_Read(usartRx);
                split(usartRx, &canIdOut, &aux);
                break;

            case 'U':
                txFlag = 0;
                break;
        }
    }
}

void main(void)
{
    // Habilita interrupcao do USART
    INTCONbits.GIE = 1;
    INTCONbits.PEIE = 1;

    canDataIn.longType[0] = 0;
    canDataIn.longType[1] = 0;

    Usart_Init(spbrg, brgh);

    Can_Set_Mode(CONFIG_MODE);
    Can_Init(sjw, brp, propseg, phseg1, phseg2);
    Can_Set_Mask(0, 0x0000);
    Can_Set_Filter(F0_B0, 0x0000);
    Can_Set_Mode(NORMAL_MODE);
    Can_Set_ID(0, 31);

    while(1)
    {
        Can_Read(canDataIn.charType, &canIdIn, &canDlc, &
   canErrorCount);

        toCharArray(usartIdOut, canIdIn, 16); // Base 16 -
   Hexadecial
```

```
111         Usart_Write(usartIdOut);

113         TXREG = '#'; while(TXSTAbits.TRMT == 0); // Separador

115         switch (canDlc)
            {
117             case 1:
                    toCharArray(usartDataOut,canDataIn.charType
    [0],16);
119                 Usart_Write(usartDataOut);
                    break;
121
                case 2:
123                 toCharArray(usartDataOut,canDataIn.wordType
    [0],16);
                    Usart_Write(usartDataOut);
125                 break;

127             case 3:
                    toCharArray(usartDataOut,canDataIn.longType[0]
    & 0x00FFFFFF,16);
129                 Usart_Write(usartDataOut);
                    break;
131
                case 4:
133                 toCharArray(usartDataOut,canDataIn.longType
    [0],16);
                    Usart_Write(usartDataOut);
135                 break;

137             case 5:
                    toCharArray(usartDataOut,canDataIn.charType
    [4],16);
139                 Usart_Write(usartDataOut);
                    toCharArrayFixedDigits(usartDataOut,canDataIn.
    longType[0],16,8);
141                 Usart_Write(usartDataOut);
                    break;
143
                case 6:
145                 toCharArray(usartDataOut,canDataIn.wordType
    [2],16);
                    Usart_Write(usartDataOut);
147                 toCharArrayFixedDigits(usartDataOut,canDataIn.
    longType[0],16,8);
                    Usart_Write(usartDataOut);
149                 break;

151             case 7:
                    toCharArray(usartDataOut,canDataIn.longType[1]
    & 0x00FFFFFF,16);
153                 Usart_Write(usartDataOut);
                    toCharArrayFixedDigits(usartDataOut,canDataIn.
    longType[0],16,8);
155                 Usart_Write(usartDataOut);
                    break;
157
                case 8:
159                 toCharArray(usartDataOut,canDataIn.longType
    [1],16);
```

```
                    Usart_Write(usartDataOut);
161                 toCharArrayFixedDigits(usartDataOut,canDataIn.
    longType[0],16,8);
                    Usart_Write(usartDataOut);
163                 break;
            }
165
            TXREG = '#'; while(TXSTAbits.TRMT == 0);
167         TXREG = '\n'; while(TXSTAbits.TRMT == 0);

169         if(txFlag == 1)
            {
171             canDataOut.charType[0] = aux;
                Can_Set_ID(0, canIdOut);
173             Can_Write(0, 1, canDataOut.charType);
            }
175     }
}
```

C.0.2 NO_GERADOR.c

```
#include "config.h"
#include "CAN.h"

void main(void)
{
    // Variaveis CAN
    // Temporizacao - 100 kbps
    char sjw = 1;
    char brp = 1;
    char propseg = 6;
    char phseg1 = 6;
    char phseg2 = 7;
    union data
    {
        unsigned char charType[8]; // 8x8 bits
        unsigned int  wordType[4]; // 4x16 bits
        unsigned long longType[2]; // 2x32 bits
    };
    union data canDataOut;
    canDataOut.longType[0] = 0;
    canDataOut.longType[1] = 0;

    Can_Set_Mode(CONFIG_MODE);
    Can_Init(sjw, brp, propseg, phseg1, phseg2);
    Can_Set_Mask(0, 0x0000);
    Can_Set_Filter(F0_B0, 0x00000);
    Can_Set_Mode(NORMAL_MODE);
    int var = 0;
    while(1)
    {
        Can_Set_ID(0, 13);
        canDataOut.longType[0] = 0xABCD1234;
        Can_Write(0, 4, canDataOut.charType);
        __delay_ms(50);

        Can_Set_ID(0, 15);
        canDataOut.longType[0] = 0x0A1B0C0D;
        canDataOut.wordType[2] = 0x03FF;
        Can_Write(0, 6, canDataOut.charType);
        __delay_ms(50);

        canDataOut.longType[0] = 0;
        canDataOut.longType[1] = 0;
    }
}
```

C.0.3 NO_LCD.c

```
#include "config.h"
#include "CAN.h"
#include "lcd.h"
#include "conversions.h"

void main(void)
{
    // Variaveis CAN
    // Temporizacao - 100 kbps
    char sjw = 1;
    char brp = 1;
    char propseg = 6;
    char phseg1 = 6;
    char phseg2 = 7;
    // Dados
    unsigned int canIdIn = 0;
    unsigned char canErrorCount = 0;
    unsigned char canDlc = 0;
    union data{
        unsigned char charType[8]; // 8x8 bits
        unsigned int  wordType[4]; // 4x16 bits
        unsigned long longType[2]; // 2x32 bits
    };
    union data canDataIn;
    canDataIn.longType[0] = 0;
    canDataIn.longType[1] = 0;
    // Variaveis LCD
    unsigned char lcdDataOut[9];

    Lcd_Init();
    Lcd_Set_Cursor(0, 0);
    Lcd_Write_String("ID");
    Lcd_Set_Cursor(0, 3);
    Lcd_Write_String("DATA");
    Lcd_Set_Cursor(0, 8);
    Lcd_Write_String("DLC");
    Lcd_Set_Cursor(0, 12);
    Lcd_Write_String("ERRO");

    Can_Set_Mode(CONFIG_MODE);
    Can_Init(sjw, brp, propseg, phseg1, phseg2);
    Can_Set_Mask(0, 0xFFFF);
    Can_Set_Filter(F0_B0, 11);
    Can_Set_Mode(NORMAL_MODE);
    Can_Set_ID(0, 20);

    while(1)
    {
        Can_Read(canDataIn.charType, &canIdIn, &canDlc, &
   canErrorCount);
        toCharArrayFixedDigits(lcdDataOut, canIdIn, 10, 2);
        Lcd_Set_Cursor(1, 0);
        Lcd_Write_String(lcdDataOut);
        toCharArrayFixedDigits(lcdDataOut, canDataIn.charType
   [0], 10, 4);
        Lcd_Set_Cursor(1, 3);
```

```
55          Lcd_Write_String(lcdDataOut);
            toCharArrayFixedDigits(lcdDataOut, canDlc, 10, 3);
57          Lcd_Set_Cursor(1, 8);
            Lcd_Write_String(lcdDataOut);
59          toCharArrayFixedDigits(lcdDataOut, canErrorCount, 10,
    3);
            Lcd_Set_Cursor(1, 12);
61          Lcd_Write_String(lcdDataOut);
        }
63  }
```

C.0.4 NO_SENSOR1.c

```
1   #include "config.h"
    #include "CAN.h"
3   #include "adc.h"

5   void main(void)
    {
7       // Variaveis CAN
        // Temporizacao - 100 kbps
9       char sjw = 1;
        char brp = 1;
11      char propseg = 6;
        char phseg1 = 6;
13      char phseg2 = 7;
        // Dados
15      union data
        {
17          unsigned char charType[8]; // 8x8 bits
            unsigned int  wordType[4]; // 4x16 bits
19          unsigned long longType[2]; // 2x32 bits
        };
21      union data canDataOut;
        canDataOut.longType[0] = 0;
23      canDataOut.longType[1] = 0;

25      TRISC = 0x00;
        PORTC = 0x00;
27      adc_Init();

29      Can_Set_Mode(CONFIG_MODE);
        Can_Init(sjw, brp, propseg, phseg1, phseg2);
31      Can_Set_Mask(0, 0x0000);
        Can_Set_Filter(F0_B0, 0x00000);
33      Can_Set_Mode(NORMAL_MODE);
        Can_Set_ID(0, 9);
35
        while(1)
37      {
            canDataOut.wordType[0] = adc_Read(0);
39          Can_Write(0, 2, canDataOut.charType);

41          canDataOut.longType[0] = 0;
```

```
        canDataOut.longType[1] = 0;
    }
}
```

C.0.5 NO_SENSOR2.c

```c
#include "config.h"
#include "CAN.h"
#include "adc.h"

void main(void)
{
    // Variaveis CAN
    // Temporizacao - 100 kbps
    char sjw = 1;
    char brp = 1;
    char propseg = 6;
    char phseg1 = 6;
    char phseg2 = 7;
    // Dados
    union data
    {
        unsigned char charType[8]; // 8x8 bits
        unsigned int  wordType[4]; // 4x16 bits
        unsigned long longType[2]; // 2x32 bits
    };
    union data canDataOut;
    canDataOut.longType[0] = 0;
    canDataOut.longType[1] = 0;

    TRISC = 0x00;
    PORTC = 0x00;
    adc_Init();

    Can_Set_Mode(CONFIG_MODE);
    Can_Init(sjw, brp, propseg, phseg1, phseg2);
    Can_Set_Mask(0, 0x0000);
    Can_Set_Filter(F0_B0, 0x00000);
    Can_Set_Mode(NORMAL_MODE);
    Can_Set_ID(0, 19);

    while(1)
    {
        canDataOut.wordType[0] = adc_Read(0);
        Can_Write(0, 2, canDataOut.charType);

        canDataOut.longType[0] = 0;
        canDataOut.longType[1] = 0;
    }
}
```

ANEXOS

ANEXO A – REGISTRADORES PIC18F258

A.1 INTCON

REGISTER 8-1: INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

bit 7 **GIE/GIEH:** Global Interrupt Enable bit
When IPEN (RCON<7>) = 0:
1 = Enables all unmasked interrupts
0 = Disables all interrupts
When IPEN (RCON<7>) = 1:
1 = Enables all high priority interrupts
0 = Disables all priority interrupts

bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit
When IPEN (RCON<7>) = 0:
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts
When IPEN (RCON<7>) = 1:
1 = Enables all low priority peripheral interrupts
0 = Disables all low priority peripheral interrupts

bit 5 **TMR0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4 **INT0IE:** INT0 External Interrupt Enable bit
1 = Enables the INT0 external interrupt
0 = Disables the INT0 external interrupt

bit 3 **RBIE:** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **TMR0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INT0IF:** INT0 External Interrupt Flag bit
1 = The INT0 external interrupt occurred (must be cleared in software by reading PORTB)
0 = The INT0 external interrupt did not occur

bit 0 **RBIF:** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Note: A mismatch condition will continue to set this bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared.

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	x = Bit is unknown

A.2 PIR1

REGISTER 8-4: PIR1: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF[(1)]	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

bit 7 **PSPIF:** Parallel Slave Port Read/Write Interrupt Flag bit[(1)]
1 = A read or a write operation has taken place (must be cleared in software)
0 = No read or write has occurred

bit 6 **ADIF**: A/D Converter Interrupt Flag bit
1 = An A/D conversion completed (must be cleared in software)
0 = The A/D conversion is not complete

bit 5 **RCIF**: USART Receive Interrupt Flag bit
1 = The USART receive buffer, RCREG, is full (cleared when RCREG is read)
0 = The USART receive buffer is empty

bit 4 **TXIF**: USART Transmit Interrupt Flag bit
1 = The USART transmit buffer, TXREG, is empty (cleared when TXREG is written)
0 = The USART transmit buffer is full

bit 3 **SSPIF**: Master Synchronous Serial Port Interrupt Flag bit
1 = The transmission/reception is complete (must be cleared in software)
0 = Waiting to transmit/receive

bit 2 **CCP1IF**: CCP1 Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused in this mode.

bit 1 **TMR2IF:** TMR2 to PR2 Match Interrupt Flag bit
1 = TMR2 to PR2 match occurred (must be cleared in software)
0 = No TMR2 to PR2 match occurred

bit 0 **TMR1IF:** TMR1 Overflow Interrupt Flag bit
1 = TMR1 register overflowed (must be cleared in software)
0 = TMR1 register did not overflow

Note 1: This bit is only available on PIC18F4X8 devices. For PIC18F2X8 devices, this bit is unimplemented and reads as ‘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	x = Bit is unknown

A.3 PIE1

REGISTER 8-7: PIE1: PERIPHERAL INTERRUPT ENABLE REGISTER 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE[(1)]	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

bit 7 **PSPIE:** Parallel Slave Port Read/Write Interrupt Enable bit[(1)]
1 = Enables the PSP read/write interrupt
0 = Disables the PSP read/write interrupt

bit 6 **ADIE**: A/D Converter Interrupt Enable bit
1 = Enables the A/D interrupt
0 = Disables the A/D interrupt

bit 5 **RCIE**: USART Receive Interrupt Enable bit
1 = Enables the USART receive interrupt
0 = Disables the USART receive interrupt

bit 4 **TXIE**: USART Transmit Interrupt Enable bit
1 = Enables the USART transmit interrupt
0 = Disables the USART transmit interrupt

bit 3 **SSPIE**: Master Synchronous Serial Port Interrupt Enable bit
1 = Enables the MSSP interrupt
0 = Disables the MSSP interrupt

bit 2 **CCP1IE**: CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1 **TMR2IE**: TMR2 to PR2 Match Interrupt Enable bit
1 = Enables the TMR2 to PR2 match interrupt
0 = Disables the TMR2 to PR2 match interrupt

bit 0 **TMR1IE**: TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

Note 1: This bit is only available on PIC18F4X8 devices. For PIC18F2X8 devices, this bit is unimplemented and reads as ‘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	x = Bit is unknown

A.4 TXSTA

REGISTER 18-1: TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D
bit 7							bit 0

bit 7 **CSRC:** Clock Source Select bit

Asynchronous mode:
Don't care.

Synchronous mode:
1 = Master mode (clock generated internally from BRG)
0 = Slave mode (clock from external source)

bit 6 **TX9**: 9-bit Transmit Enable bit

1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN**: Transmit Enable bit

1 = Transmit enabled
0 = Transmit disabled

Note: SREN/CREN overrides TXEN in Sync mode.

bit 4 **SYNC:** USART Mode Select bit

1 = Synchronous mode
0 = Asynchronous mode

bit 3 **Unimplemented:** Read as '0'

bit 2 **BRGH**: High Baud Rate Select bit

Asynchronous mode:
1 = High speed
0 = Low speed

Synchronous mode:
Unused in this mode.

bit 1 **TRMT**: Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** 9th bit of Transmit Data
Can be address/data bit or a parity bit.

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	x = Bit is unknown

A.5 RCSTA

REGISTER 18-2: RCSTA: RECEIVE STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

bit 7 **SPEN:** Serial Port Enable bit

1 = Serial port enabled (configures RX/DT and TX/CK pins as serial port pins)
0 = Serial port disabled

bit 6 **RX9**: 9-bit Receive Enable bit

1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN**: Single Receive Enable bit

Asynchronous mode:
Don't care.

Synchronous mode – Master:
1 = Enables single receive
0 = Disables single receive (this bit is cleared after reception is complete)

Synchronous mode – Slave:
Unused in this mode.

bit 4 **CREN**: Continuous Receive Enable bit

Asynchronous mode:
1 = Enables continuous receive
0 = Disables continuous receive

Synchronous mode:
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

bit 3 **ADDEN**: Address Detect Enable bit

Asynchronous mode 9-bit (RX9 = 1):
1 = Enables address detection, enables interrupt and load of the receive buffer when RSR<8> is set
0 = Disables address detection, all bytes are received and ninth bit can be used as parity bit

bit 2 **FERR**: Framing Error bit

1 = Framing error (can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

bit 1 **OERR**: Overrun Error bit

1 = Overrun error (can be cleared by clearing bit CREN)
0 = No overrun error

bit 0 **RX9D:** 9th bit of Received Data

Can be address/data bit or a parity bit.

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	x = Bit is unknown

A.6 CANCON

REGISTER 19-1: CANCON: CAN CONTROL REGISTER

R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0
REQOP2	REQOP1	REQOP0	ABAT	WIN2	WIN1	WIN0	—
bit 7							bit 0

bit 7-5 **REQOP2:REQOP0:** Request CAN Operation Mode bits

1xx = Request Configuration mode
011 = Request Listen Only mode
010 = Request Loopback mode
001 = Request Disable mode
000 = Request Normal mode

bit 4 **ABAT:** Abort All Pending Transmissions bit

1 = Abort all pending transmissions (in all transmit buffers)
0 = Transmissions proceeding as normal

bit 3-1 **WIN2:WIN0:** Window Address bits

This selects which of the CAN buffers to switch into the Access Bank area. This allows access to the buffer registers from any data memory bank. After a frame has caused an interrupt, the ICODE2:ICODE0 bits can be copied to the WIN2:WIN0 bits to select the correct buffer. See Example 19-1 for code example.

111 = Receive Buffer 0
110 = Receive Buffer 0
101 = Receive Buffer 1
100 = Transmit Buffer 0
011 = Transmit Buffer 1
010 = Transmit Buffer 2
001 = Receive Buffer 0
000 = Receive Buffer 0

bit 0 **Unimplemented:** Read as ‘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	x = Bit is unknown

A.7 CANSTAT

REGISTER 19-2: CANSTAT: CAN STATUS REGISTER

R-1	R-0	R-0	U-0	R-0	R-0	R-0	U-0
OPMODE2	OPMODE1	OPMODE0	—	ICODE2	ICODE1	ICODE0	—
bit 7							bit 0

bit 7-5 **OPMODE2:OPMODE0:** Operation Mode Status bits

111 = Reserved
110 = Reserved
101 = Reserved
100 = Configuration mode
011 = Listen Only mode
010 = Loopback mode
001 = Disable mode
000 = Normal mode

Note: Before the device goes into Sleep mode, select Disable mode.

bit 4 **Unimplemented:** Read as ‘0’

bit 3-1 **ICODE2:ICODE0:** Interrupt Code bits

When an interrupt occurs, a prioritized coded interrupt value will be present in the ICODE2:ICODE0 bits. These codes indicate the source of the interrupt. The ICODE2:ICODE0 bits can be copied to the WIN2:WIN0 bits to select the correct buffer to map into the Access Bank area. See Example 19-1 for code example.

111 = Wake-up on interrupt
110 = RXB0 interrupt
101 = RXB1 interrupt
100 = TXB0 interrupt
011 = TXB1 interrupt
010 = TXB2 interrupt
001 = Error interrupt
000 = No interrupt

bit 0 **Unimplemented:** Read as ‘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	x = Bit is unknown

A.8 TXBNCON

REGISTER 19-4: TXBnCON: TRANSMIT BUFFER n CONTROL REGISTERS

U-0	R-0	R-0	R-0	R/W-0	U-0	R/W-0	R/W-0
—	TXABT	TXLARB	TXERR	TXREQ	—	TXPRI1	TXPRI0
bit 7							bit 0

bit 7 **Unimplemented:** Read as ‘0’

bit 6 **TXABT:** Transmission Aborted Status bit
1 = Message was aborted
0 = Message was not aborted

bit 5 **TXLARB:** Transmission Lost Arbitration Status bit
1 = Message lost arbitration while being sent
0 = Message did not lose arbitration while being sent

bit 4 **TXERR:** Transmission Error Detected Status bit
1 = A bus error occurred while the message was being sent
0 = A bus error did not occur while the message was being sent

bit 3 **TXREQ:** Transmit Request Status bit
1 = Requests sending a message. Clears the TXABT, TXLARB and TXERR bits.
0 = Automatically cleared when the message is successfully sent

Note: Clearing this bit in software while the bit is set will request a message abort.

bit 2 **Unimplemented:** Read as ‘0’

bit 1-0 **TXPRI1:TXPRI0:** Transmit Priority bits
11 = Priority Level 3 (highest priority)
10 = Priority Level 2
01 = Priority Level 1
00 = Priority Level 0 (lowest priority)

Note: These bits set the order in which the Transmit Buffer will be transferred. They do not alter the CAN message identifier.

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	x = Bit is unknown

A.9 TXBNSIDH

REGISTER 19-5: TXBnSIDH: TRANSMIT BUFFER n STANDARD IDENTIFIER, HIGH BYTE REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

bit 7-0 **SID10:SID3:** Standard Identifier bits if EXIDE = 0 (TXBnSID Register) or Extended Identifier bits EID28:EID21 if EXIDE = 1

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	x = Bit is unknown

A.10 TXBNSIDL

REGISTER 19-6: TXBnSIDL: TRANSMIT BUFFER n STANDARD IDENTIFIER, LOW BYTE REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

bit 7-5 **SID2:SID0:** Standard Identifier bits if EXIDE = 0 or Extended Identifier bits EID20:EID18 if EXIDE = 1

bit 4 **Unimplemented:** Read as ‘0’

bit 3 **EXIDE:** Extended Identifier enable bit
1 = Message will transmit extended ID, SID10:SID0 becomes EID28:EID18
0 = Message will transmit standard ID, EID17:EID0 are ignored

bit 2 **Unimplemented:** Read as ‘0’

bit 1-0 **EID17:EID16:** Extended Identifier bits

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	x = Bit is unknown

A.11 TXBNDM

REGISTER 19-9: TXBnDm: TRANSMIT BUFFER n DATA FIELD BYTE m REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
TXBnDm7	TXBnDm6	TXBnDm5	TXBnDm4	TXBnDm3	TXBnDm2	TXBnDm1	TXBnDm0
bit 7							bit 0

bit 7-0 **TXBnDm7:TXBnDm0:** Transmit Buffer n Data Field Byte m bits (where 0 ≤ n < 3 and 0 < m < 8)

Each Transmit Buffer has an array of registers. For example, Transmit Buffer 0 has 7 registers: TXB0D0 to TXB0D7.

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	x = Bit is unknown

A.12 TXBNDLC

REGISTER 19-10: TXBnDLC: TRANSMIT BUFFER n DATA LENGTH CODE REGISTERS

U-0	R/W-x	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x
—	TXRTR	—	—	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

bit 7 **Unimplemented:** Read as ‘0’

bit 6 **TXRTR:** Transmission Frame Remote Transmission Request bit
1 = Transmitted message will have TXRTR bit set
0 = Transmitted message will have TXRTR bit cleared

bit 5-4 **Unimplemented:** Read as ‘0’

bit 3-0 **DLC3:DLC0:** Data Length Code bits
1111 = Reserved
1110 = Reserved
1101 = Reserved
1100 = Reserved
1011 = Reserved
1010 = Reserved
1001 = Reserved
1000 = Data Length = 8 bytes
0111 = Data Length = 7 bytes
0110 = Data Length = 6 bytes
0101 = Data Length = 5 bytes
0100 = Data Length = 4 bytes
0011 = Data Length = 3 bytes
0010 = Data Length = 2 bytes
0001 = Data Length = 1 bytes
0000 = Data Length = 0 bytes

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	x = Bit is unknown

A.13 RXB0CON

REGISTER 19-12: RXB0CON: RECEIVE BUFFER 0 CONTROL REGISTER

R/C-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
RXFUL(1)	RXM1(1)	RXM0(1)	—	RXRTRRO	RXB0DBEN	JTOFF	FILHIT0
bit 7							bit 0

bit 7 **RXFUL:** Receive Full Status bit(1)

1 = Receive buffer contains a received message
0 = Receive buffer is open to receive a new message

Note: This bit is set by the CAN module and must be cleared by software after the buffer is read.

bit 6-5 **RXM1:RXM0:** Receive Buffer Mode bits(1)

11 = Receive all messages (including those with errors)
10 = Receive only valid messages with extended identifier
01 = Receive only valid messages with standard identifier
00 = Receive all valid messages

bit 4 **Unimplemented:** Read as ‘0’

bit 3 **RXRTRRO:** Receive Remote Transfer Request Read-Only bit

1 = Remote transfer request
0 = No remote transfer request

bit 2 **RXB0DBEN:** Receive Buffer 0 Double-Buffer Enable bit

1 = Receive Buffer 0 overflow will write to Receive Buffer 1
0 = No Receive Buffer 0 overflow to Receive Buffer 1

bit 1 **JTOFF:** Jump Table Offset bit (read-only copy of RXB0DBEN)

1 = Allows jump table offset between 6 and 7
0 = Allows jump table offset between 1 and 0

Note: This bit allows same filter jump table for both RXB0CON and RXB1CON.

bit 0 **FILHIT0:** Filter Hit bit

This bit indicates which acceptance filter enabled the message reception into Receive Buffer 0.

1 = Acceptance Filter 1 (RXF1)
0 = Acceptance Filter 0 (RXF0)

Note 1: Bits RXFUL, RXM1 and RXM0 of RXB0CON are not mirrored in RXB1CON.

Legend:			
R = Readable bit	W = Writable bit	C = Clearable bit	U = Unimplemented bit, read as ‘0’
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

A.14 RXB1CON

REGISTER 19-13: RXB1CON: RECEIVE BUFFER 1 CONTROL REGISTER

R/C-0	R/W-0	R/W-0	U-0	R-0	R-0	R-0	R-0
RXFUL(1)	RXM1(1)	RXM0(1)	—	RXRTRRO	FILHIT2	FILHIT1	FILHIT0
bit 7							bit 0

bit 7 **RXFUL:** Receive Full Status bit(1)

1 = Receive buffer contains a received message
0 = Receive buffer is open to receive a new message

Note: This bit is set by the CAN module and should be cleared by software after the buffer is read.

bit 6-5 **RXM1:RXM0:** Receive Buffer Mode bits(1)

11 = Receive all messages (including those with errors)
10 = Receive only valid messages with extended identifier
01 = Receive only valid messages with standard identifier
00 = Receive all valid messages

bit 4 **Unimplemented:** Read as ‘0’

bit 3 **RXRTRRO:** Receive Remote Transfer Request bit (read-only)

1 = Remote transfer request
0 = No remote transfer request

bit 2-0 **FILHIT2:FILHIT0:** Filter Hit bits

These bits indicate which acceptance filter enabled the last message reception into Receive Buffer 1.

111 = Reserved
110 = Reserved
101 = Acceptance Filter 5 (RXF5)
100 = Acceptance Filter 4 (RXF4)
011 = Acceptance Filter 3 (RXF3)
010 = Acceptance Filter 2 (RXF2)
001 = Acceptance Filter 1 (RXF1), only possible when RXB0DBEN bit is set
000 = Acceptance Filter 0 (RXF0), only possible when RXB0DBEN bit is set

Note 1: Bits RXFUL, RXM1 and RXM0 of RXB1CON are not mirrored in RXB0CON.

Legend:			
R = Readable bit	W = Writable bit	C = Clearable bit	U = Unimplemented bit, read as ‘0’
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

A.15 RXBNSIDH

REGISTER 19-14: RXBnSIDH: RECEIVE BUFFER n STANDARD IDENTIFIER, HIGH BYTE REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

bit 7-0 **SID10:SID3:** Standard Identifier bits if EXID = 0 (RXBnSIDL Register) or Extended Identifier bits EID28:EID21 if EXID = 1

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	x = Bit is unknown

A.16 RXBNSIDL

REGISTER 19-15: RXBnSIDL: RECEIVE BUFFER n STANDARD IDENTIFIER, LOW BYTE REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	U-0	R/W-x	R/W-x
SID2	SID1	SID0	SRR	EXID	—	EID17	EID16
bit 7							bit 0

bit 7-5 **SID2:SID0:** Standard Identifier bits if EXID = 0 or Extended Identifier bits EID20:EID18 if EXID = 1

bit 4 **SRR:** Substitute Remote Request bit

This bit is always ‘0’ when EXID = 1 or equal to the value of RXRTRRO (RXnBCON<3>) when EXID = 0.

bit 3 **EXID:** Extended Identifier bit

1 = Received message is an extended data frame, SID10:SID0 are EID28:EID18
0 = Received message is a standard data frame

bit 2 **Unimplemented:** Read as ‘0’

bit 1-0 **EID17:EID16:** Extended Identifier bits

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	x = Bit is unknown

A.17 RXBNDLC

REGISTER 19-18: RXBnDLC: RECEIVE BUFFER n DATA LENGTH CODE REGISTERS

U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	RXRTR	RB1	RB0	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

bit 7 **Unimplemented:** Read as ‘0’

bit 6 **RXRTR:** Receiver Remote Transmission Request bit
1 = Remote transfer request
0 = No remote transfer request

bit 5 **RB1:** Reserved bit 1
Reserved by CAN spec and read as ‘0’.

bit 4 **RB0:** Reserved bit 0
Reserved by CAN spec and read as ‘0’.

bit 3-0 **DLC3:DLC0:** Data Length Code bits
1111 = Invalid
1110 = Invalid
1101 = Invalid
1100 = Invalid
1011 = Invalid
1010 = Invalid
1001 = Invalid
1000 = Data Length = 8 bytes
0111 = Data Length = 7 bytes
0110 = Data Length = 6 bytes
0101 = Data Length = 5 bytes
0100 = Data Length = 4 bytes
0011 = Data Length = 3 bytes
0010 = Data Length = 2 bytes
0001 = Data Length = 1 bytes
0000 = Data Length = 0 bytes

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	x = Bit is unknown

A.18 RXBNDM

REGISTER 19-19: RXBnDm: RECEIVE BUFFER n DATA FIELD BYTE m REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
RXBnDm7	RXBnDm6	RXBnDm5	RXBnDm4	RXBnDm3	RXBnDm2	RXBnDm1	RXBnDm0
bit 7							bit 0

bit 7-0 **RXBnDm7:RXBnDm0:** Receive Buffer n Data Field Byte m bits (where $0 \leq n < 1$ and $0 < m < 7$)
Each receive buffer has an array of registers. For example, Receive Buffer 0 has 8 registers: RXB0D0 to RXB0D7.

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	x = Bit is unknown

A.19 RXFNSIDH

REGISTER 19-21: RXFnSIDH: RECEIVE ACCEPTANCE FILTER n STANDARD IDENTIFIER FILTER, HIGH BYTE REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

bit 7-0 **SID10:SID3:** Standard Identifier Filter bits if EXIDEN = 0 or
Extended Identifier Filter bits EID28:EID21 if EXIDEN = 1

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	x = Bit is unknown

A.20 RXFNSIDL

REGISTER 19-22: RXFnSIDL: RECEIVE ACCEPTANCE FILTER n STANDARD IDENTIFIER FILTER, LOW BYTE REGISTERS

R/W-x	R/W-x	R/W-x	U-0	R/W-x	U-0	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDEN	—	EID17	EID16
bit 7							bit 0

bit 7-5 **SID2:SID0:** Standard Identifier Filter bits if EXIDEN = 0 or
Extended Identifier Filter bits EID20:EID18 if EXIDEN = 1

bit 4 **Unimplemented:** Read as ‘0’

bit 3 **EXIDEN:** Extended Identifier Filter Enable bit
1 = Filter will only accept extended ID messages
0 = Filter will only accept standard ID messages

bit 2 **Unimplemented:** Read as ‘0’

bit 1-0 **EID17:EID16:** Extended Identifier Filter bits

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	x = Bit is unknown

A.21 RXMNSIDH

REGISTER 19-25: RXMnSIDH: RECEIVE ACCEPTANCE MASK n STANDARD IDENTIFIER MASK, HIGH BYTE REGISTERS

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

bit 7-0 **SID10:SID3:** Standard Identifier Mask bits or Extended Identifier Mask bits EID28:EID21

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	x = Bit is unknown

A.22 RXMNSIDL

REGISTER 19-26: RXMnSIDL: RECEIVE ACCEPTANCE MASK n STANDARD IDENTIFIER MASK, LOW BYTE REGISTERS

R/W-x	R/W-x	R/W-x	U-0	U-0	U-0	R/W-x	R/W-x
SID2	SID1	SID0	—	—	—	EID17	EID16
bit 7							bit 0

bit 7-5 **SID2:SID0:** Standard Identifier Mask bits or Extended Identifier Mask bits EID20:EID18

bit 4-2 **Unimplemented:** Read as ‘0’

bit 1-0 **EID17:EID16:** Extended Identifier Mask bits

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	x = Bit is unknown

A.23 BRGCON1

REGISTER 19-29: BRGCON1: BAUD RATE CONTROL REGISTER 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
bit 7							bit 0

bit 7-6 **SJW1:SJW0:** Synchronized Jump Width bits

11 = Synchronization Jump Width Time = 4 x TQ
10 = Synchronization Jump Width Time = 3 x TQ
01 = Synchronization Jump Width Time = 2 x TQ
00 = Synchronization Jump Width Time = 1 x TQ

bit 5-0 **BRP5:BRP0:** Baud Rate Prescaler bits

111111 = TQ = (2 x 64)/FOSC
111110 = TQ = (2 x 63)/FOSC
:
:
000001 = TQ = (2 x 2)/FOSC
000000 = TQ = (2 x 1)/FOSC

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	x = Bit is unknown

Note: This register is accessible in Configuration mode only.

A.24 BRGCON2

REGISTER 19-30: BRGCON2: BAUD RATE CONTROL REGISTER 2

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SEG2PHTS	SAM	SEG1PH2	SEG1PH1	SEG1PH0	PRSEG2	PRSEG1	PRSEG0
bit 7							bit 0

bit 7 **SEG2PHTS:** Phase Segment 2 Time Select bit
1 = Freely programmable
0 = Maximum of PHEG1 or Information Processing Time (IPT), whichever is greater

bit 6 **SAM:** Sample of the CAN bus Line bit
1 = Bus line is sampled three times prior to the sample point
0 = Bus line is sampled once at the sample point

bit 5-3 **SEG1PH2:SEG1PH0:** Phase Segment 1 bits
111 = Phase Segment 1 Time = 8 x T_Q
110 = Phase Segment 1 Time = 7 x T_Q
101 = Phase Segment 1 Time = 6 x T_Q
100 = Phase Segment 1 Time = 5 x T_Q
011 = Phase Segment 1 Time = 4 x T_Q
010 = Phase Segment 1 Time = 3 x T_Q
001 = Phase Segment 1 Time = 2 x T_Q
000 = Phase Segment 1 Time = 1 x T_Q

bit 2-0 **PRSEG2:PRSEG0:** Propagation Time Select bits
111 = Propagation Time = 8 x T_Q
110 = Propagation Time = 7 x T_Q
101 = Propagation Time = 6 x T_Q
100 = Propagation Time = 5 x T_Q
011 = Propagation Time = 4 x T_Q
010 = Propagation Time = 3 x T_Q
001 = Propagation Time = 2 x T_Q
000 = Propagation Time = 1 x T_Q

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	x = Bit is unknown

Note: This register is accessible in Configuration mode only.

A.25 BRGCON3

REGISTER 19-31: BRGCON3: BAUD RATE CONTROL REGISTER 3

U-0	R/W-0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0
—	WAKFIL	—	—	—	SEG2PH2(1)	SEG2PH1(1)	SEG2PH0(1)
bit 7							bit 0

bit 7 **Unimplemented:** Read as ‘0’

bit 6 **WAKFIL:** Selects CAN bus Line Filter for Wake-up bit
1 = Use CAN bus line filter for wake-up
0 = CAN bus line filter is not used for wake-up

bit 5-3 **Unimplemented:** Read as ‘0’

bit 2-0 **SEG2PH2:SEG2PH0:** Phase Segment 2 Time Select bits(1)
111 = Phase Segment 2 Time = 8 x TQ
110 = Phase Segment 2 Time = 7 x TQ
101 = Phase Segment 2 Time = 6 x TQ
100 = Phase Segment 2 Time = 5 x TQ
011 = Phase Segment 2 Time = 4 x TQ
010 = Phase Segment 2 Time = 3 x TQ
001 = Phase Segment 2 Time = 2 x TQ
000 = Phase Segment 2 Time = 1 x TQ

Note 1: Ignored if SEG2PHTS bit (BRGCON2<7>) is clear.

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	x = Bit is unknown

A.26 CIOCON

REGISTER 19-32: 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	CANCAP	—	—	—	—
bit 7							bit 0

bit 7-6 **Unimplemented**: Read as ‘0’

bit 5 **ENDRHI:** Enable Drive High bit
1 = CANTX pin will drive VDD when recessive
0 = CANTX pin will 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’

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	x = Bit is unknown

A.27 ADCON0

REGISTER 20-1 ADCON0: A/D CONTROL REGISTER 0

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON
bit 7							bit 0

bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	**00**	Fosc/2
0	**01**	Fosc/8
0	**10**	Fosc/32
0	**11**	Frc (clock derived from the internal A/D RC oscillator)
1	**00**	Fosc/4
1	**01**	Fosc/16
1	**10**	Fosc/64
1	**11**	Frc (clock derived from the internal A/D RC oscillator)

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits

000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)(1)
110 = Channel 6 (AN6)(1)
111 = Channel 7 (AN7)(1)

Note 1: These channels are unimplemented on PIC18F2X8 (28-pin) devices. Do not select any unimplemented channel.

bit 2 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit

When ADON = 1:

1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)

0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit

1 = A/D converter module is powered up

0 = A/D converter module is shut-off and consumes no operating current

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	x = Bit is unknown

A.28 ADCON1

REGISTER 20-2: ADCON1: A/D CONTROL REGISTER 1

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0

bit 7 bit 0

bit 7 **ADFM:** A/D Result Format Select bit

1 = Right justified. Six (6) Most Significant bits of ADRESH are read as ‘0’.
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as ‘0’.

bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	FOSC/2
0	01	FOSC/8
0	10	FOSC/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	FOSC/4
1	01	FOSC/16
1	10	FOSC/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-4 **Unimplemented:** Read as ‘0’

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

PCFG	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references

Note: Shaded cells indicate channels available only on PIC18F4X8 devices.

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	x = Bit is unknown

Note: On any device Reset, the port pins that are multiplexed with analog functions (ANx) are forced to be analog inputs.