Leonardo Cayesse Zeferino Miúdo

Bacharel em Engenharia Eléctrica - Universidade Agostinho Neto

Automação distribuída com protocolos CAN e Modbus

Dissertação apresentada para obtenção do Grau de Mestre em Engenharia Electrotécnica, Sistemas e de Computadores na especialidade de Energia Eléctrica e Automação

Orientador : Doutor Luis Filipe Figueira de Brito Palma, Professor Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júris:

Presidente:	Prof. Associado Doutor Fernando Vieira do Coito
Arguente:	Prof. Auxiliar Doutor João Almeida das Rosas
Vogal:	Prof. Auxiliar Doutor Luis Filipe de Brito Palma

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2014

Automação distribuida com protocolos CAN e Modbus

Copyright © Leonardo Cayesse Zeferino Miúdo, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

"Página deixada em branco"

Ao meu irmão Wilson Miúdo.

"Página deixada em branco"

Agradecimentos

Agradeço primeiramente a Deus por me dar saúde e ter estado sempre presente durante esta caminhada, só Ele sabe as dificuldades que foram superadas para alcançar o objectivo da conclusão desta dissertação.

Quero agradecer especialmente à minha família pelo incentivo e pelo apoio que têm dado para a conclusão do curso, e por estarem sempre presente durante toda a minha formação académica.

Agradeço ao senhor Jorge Maneco, por acreditar em mim e pelo incentivo dado para esta formação.

Ao Professor Doutor Luis Filipe de Brito Palma, orientador desta dissertação, agradeço a oportunidade e o apoio dado para a realização deste trabalho científico, a dedicação, a motivação e por incansavelmente estar sempre disponível para solucionar dúvidas, dar sugestões e indicar correções.

Agradeço à empresa *Schneider Electric* Portugal pelo apoio e pela disponibilidade de documentos sobre o protocolo Modbus, em particular à Engª Fátima Borges.

Gostaria de apresentar os meus agradecimentos aos meus amigos, pelo apoio e acreditarem que fosse possível em especial ao Abreu Liliano, Debs Tavares e ao José Teixeira. Agradeço também a todas as pessoas que directa ou indirectamente contribuiram na execução deste trabalho, em particular ao Mestre Fábio Januário e ao Mestre João Cruz. Muito obrigado.

"Página deixada em branco"

Resumo

As redes industriais têm um papel importante no controlo e supervisão de processos industriais. Modbus e CAN são protocolos abertos que permitem uma comunicação rápida e fiável entre equipamentos de diferentes fornecedores. O objectivo desta dissertação resume-se num estudo pormenorizado destes dois protocolos e implementar algoritmos de controlo e automação utilizando estes protocolos. O protocolo Modbus foi implementado em linha série RS-485 no modo RTU, utilizando autómatos programáveis Twido da *Schneider Electric*, para o controlo de velocidade de dois ventiladores com um controlador PID. Foi criada uma interface para a monitorização do processo utilizando o padrão OPC®, tendo como servidor OPC o *software* MatrikonOPC e como cliente OPC o *software* Scilab®. Quanto ao protocolo CAN, implementou-se num sistema de controlo de nível de água (processo PCT9), utilizando *shields* CAN de placas com microcontroladores com um controlador PID com *anti-windup* e como camada de aplicação utilizou-se o IDE do Arduino; integrou-se novamente o Scilab® para a criação de uma interface. Foram obtidos resultados experimentais na análise dos protocolos e da aplicação no controlo dos processos. Para o protocolo Modbus foram realizados testes para comunicação em *unicast* sem erro e para comunicação em *excepction response*; analisou-se a comunicação em *broadcast* e erros no barramento. O padrão OPC® apresentou bons resultados por permitir uma comunicação em boas condições entre os dispositivos. Para o CAN, obteve-se resultados das tramas de 11 e 29 *bits*, analisou-se detalhes dos campos da trama de 11 *bits*, as tramas com alta e baixa prioridade, o *bit stuffing* e o *bit* de confirmação *ACK slot*. Com os resultados experimentais obtidos, concluiu-se a grande importância destes protocolos para os sistemas de automação distribuída; e a importância do padrão OPC® em unificar a comunicação entre diversos dispositivos. Concluiu-se também o porquê de estes dois protocolos apresentarem grande relevância no campo das redes industriais e a nível de outros serviços.

Palavras Chave: Modbus, CAN, Microcontroladores, PLCs, OPC®, Scilab®.

”Página deixada em branco”

Abstract

Industrial networks have an important role on the control and supervision of industrial processes. Modbus and CAN are open protocols which allow a fast and reliable communication between different suppliers and equipment. The overall goal of this dissertation is to study in detail each protocol and to implement control and automation algorithms over the protocols. Modbus protocol was implement in serial line RS-485 in the RTU mode, using Schneider Electric programmable logic controller Twido, for two fans speed control. An interface was also created for process monitoring, using OPC® standard, as OPC server the software MatrikonOPC and a Scilab® program as OPC client. Regarding CAN protocol, it was implemented on a water tank process (Process PCT9) for water level control, using Arduino shields CAN and as application layer the Arduino's integrated development environment; Scilab® was integrated for the creation of an interface.

Protocol application experimental results on the process control were obtained, for Modbus protocol tests were taken for the communication on unicast without error and for communication in exception response; Communication broadcast and bus errors were analyzed. OPC® showed good results by allowing the communication between equipment in good conditions. Regarding CAN, results were obtained concerning 11 and 29 bits frames, it was analyzed 11 bits frames fields, frames with high and low priority, bit stuffing and the ACK slot bit confirmation. From the experimental results obtained, it was concluded the great importance of these protocols on the distributed automation systems; and the relevance of the OPC® standard in unifying the communication between several equipments. It was also concluded why these two protocols are so important on the field of industrial networks and other services.

Keywords: Modbus Protocol, CAN Protocol, Microcontrollers, Programmable Logic Controller, OPC® standard, Scilab®.

"Página deixada em branco"

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

ACK *Acknowledgement*

ADU *Unidade de Dados de Aplicação*

ASCII *Código Padrão Americano para o Intercâmbio de Informação*

CAL *Camada de Aplicação CAN*

CAN *Controller Area Network*

CD-AMP *Collision Detection and Arbitration on Message Priority*

CiA *CAN in Automation*

CR *Resolução de Colisão*

CRC *Verificação de Redundância Cíclica*

CSMA *Carrier Sense Multiple Acess*

DLC *Data Lenght Code*

EIA *Aliança das Indústrias Eletrônicas*

EOF *Fim da Trama*

HDLC *High Level Data Link Control*

HMI *Interface Homem-Máquina*

IEEE *Instituto de Engenheiros Electricistas e Electrónicos*

IL *Lista de Instruções*

ISA *Instrument Society of America*

ISO *International Standardization Organization*

I/O *Input/Output*

LLC *Controlo de ligação lógica*

LRC *Verificação de Redundância Longitudinal*

MAC *Controlo de acesso ao meio*

NRZ *Non Return to Zero*

OLE *Object Linking and Embedding*

OPC *OLE Process Control*

OSI *Open System Interconnection*

PDU *Unidade de Dados do Protocolo*

PID *Proporcional, Integral e Derivativo*

PLC *Controlador Lógico Programável*

RTR *Pedido de Transmissão Remota*

RTU *Unidade Terminal Remota*

SDS *Smart Distributed System*

SOF *Início da trama*

SPI *Serial Peripherical Interface*

SRR *Substitute Remote Request*

TCP/IP *Protocolo de Controlo de Transmissão/Protocolo de Internet*

TIA *Associação das Indústrias de Telecomunicações*

Capítulo 1

Introdução

Neste capítulo introdutório é feito um enquadramento que conduziu à execução desta dissertação. Na secção 1.1 apresenta-se a introdução da tese, dando uma visão da importância das redes industriais mais precisamente dos protocolos no desenvolvimento do sector industrial.

Na secção 1.2 é apresentado o contexto e o que motivou a implementação destes dois protocolos CAN (*Controller Area Network*) e Modbus nesta dissertação. Em seguida na secção 1.3 são apresentadas as contribuições e os objectivos a alcançar com este trabalho. Finalmente na secção 1.4 é dada uma visão de como está estruturada a dissertação.

1.1 Introdução

Automação é a tecnologia relacionada com a aplicação de sistemas mecânicos, eléctricos e electrónicos, apoiados em meios computacionais, na operação e controlo dos sistemas de produção [24]. Para se ter eficiência na produção em qualquer indústria é necessário que as máquinas e sistemas de controlo estejam integrados e que troquem constantemente informação de maneira rápida e segura. Assim sendo, é possível controlar e supervisionar processos na indústria com maior agilidade e facilidade utilizando redes industriais.

As redes industriais são sistemas distribuídos onde os seus componentes comunicam entre si com a finalidade de controlar processos. Um sistema distribuído é um conjunto de componentes localizados numa rede que comunicam entre si e coordenam as suas acções apenas pela troca de mensagens.

Os protocolos de comunicação industrial têm um papel importante nas redes industriais porque representam o "idioma"utilizado pelos componentes do sistema de forma a conseguirem comunicar. Um protocolo é um conjunto de regras que define como os equipamentos trocam os dados, ou seja, protocolos são regras que controlam a comunicação entre dois ou mais equipamentos. Para aplicação em automação distribuída existem vários protocolos como: Profibus, AS-I, Fieldbus, Ethernet, CANopen, Modbus, CAN (*Controller Area Network*), DeviceNet, entre outros. Os protocolos que serão estudados e implementados nesta dissertação são o Modbus e o CAN por serem os protocolos mais populares em sistemas de automação e controlo industrial.

O protocolo Modbus foi criado em 1979 pela *Modicon Industrial Automation Systems* (actual *Schneider Electric*) e continua a permitir que vários dispositivos de automação comunicam entre si, no campo das redes industriais é talvez o protocolo mais utilizado.

O protocolo CAN foi introduzido pela empresa *Robert Bosch GmbH* em 1986, sendo um dos protocolos mais bem sucedidos de sempre. Inicialmente foi criado para o mercado automóvel mas actualmente também é utilizado em muitas aplicações industriais.

1.2 Contexto e Motivação

Actualmente em sistemas de automação industrial e em outros níveis de aplicação, as redes de comunicação industrial, ou seja, os protocolos industriais constituem um tema de crescente importância porque vários dispositivos na indústria possuem interfaces de algum tipo de rede industrial.

Diversos fornecedores possuem soluções de redes de comunicação proprietárias, fazendo com que o cliente fique dependente de produtos, serviços e manutenção de um único fabricante. Com o objectivo principal da interoperabilidade[1] e flexibilidade[2] de operação, grupos de desenvolvedores definem normas de padrão aberto (sistemas que su-

[1]Interoperabilidade é a capacidade para comunicar inteligentemente com outros dispositivos.

[2]Flexibilidade é a capacidade para substituir um ou outro dispositivo (possivelmente de fornecedor diferente).

portam equipamentos e dispositivos de diferentes fabricantes) para o desenvolvimento de redes de comunicação por todos os interessados. Com isso, os desenvolvedores ganham com a flexibilidade de desenvolvimento de linhas de produtos em função da procura, e o cliente ganha o facto de não ficar totalmente preso a apenas um fornecedor [22].

Para além da motivação de que ambos os protocolos serem protocolos abertos, Modbus e CAN são dois dos barramentos mais populares em sistemas de automação e controlo industrial.

A motivação que levou o estudo do protocolo CAN é por ser um padrão popular fora da indústria automóvel, possuindo aplicações na medicina, marinha e em qualquer área em que seja necessário uma rede simples mas robusta; é também o barramento de comunicação padrão nos veículos de hoje. Foi definido que até 2008, todos os veículos vendidos nos EUA seriam obrigados a implementar CAN com o padrão OBD-II [3] , eliminando o problema de se ter cinco protocolos [4] [19].

A tabela 1.1 apresenta os anos de início do padrão OBD-II com protocolos CAN. Outra motivação do barramento CAN, é que permite a aplicação de vários protocolos de alto nível como CANopen, DeviceNet, SDS (*Smart Distributed System*) entre outros, permitindo um maior número de aplicações do barramento.

Tabela 1.1: Padrão OBD-II por país [19].

Países	Ano de início obrigatório OBD
Estados Unidos da America	1996
Canada	1998
Europa	2000/2001
Europa (Diesel)	2004
Australia	2006
Todos os outros países	Opcional

A motivação do estudo do protocolo Modbus deve-se ao facto de ser um dos protocolos mais antigos e por ainda hoje ser no ambiente de produção industrial o mais utilizado, permitindo a implementação em Modbus padrão (linha série RS-485 e RS-232), Modbus TCP/IP (*Protocolo de Controlo de Transmissão/Protocolo de Internet*) e Modbus *Plus* (esta versão é mantida sob domínio da Schneider Electric e só pode ser implementada sob licença deste fabricante).

Actualmente, o Modbus não é apenas um protocolo industrial, pois muitas outras aplicações no ramo da energia, transporte, construção fazem o uso dos seus benefícios. Analistas do sector de integração de vários fornecedores têm relatado que existem mais de

[3] OBD (do inglês On-Board Diagnostic) designa um sistema de autodiagnóstico disponível actualmente na maioria dos automóveis.

[4] ISO 9141, SAE J1850 PWM e SAE J1850 VPW são protocolos utilizados pela Chrysler, GM e Ford. O protocolo ISO14230 e o ISO9141 [19].

7 milhões de nós com protocolo Modbus na América do Norte e na Europa [25].

1.3 Objectivos e Contribuições

Os objectivos e contribuições desta dissertação resume-se no estudo detalhado dos protocolos CAN e Modbus para a área de controlo e automação industrial; com base nestes protocolos foram implementados algoritmos de controlo distribuído em rede com PLCs (*Controladores Lógicos Programáveis*) e placas Arduino com controladores CAN (*shield* CAN).

Para além do estudo destes dois barramentos que são os mais populares da indústria, para o protocolo Modbus também se incluirá o padrão OPC® (*OLE Process Control*) onde OLE significa *"Object Linking and Embedding"*. OPC® é um padrão de interoperabilidade para troca de dados segura e confiável no espaço de automação industrial e em outros sectores.

Para o protocolo Modbus os objectivos são:

- Um estudo pormenorizado do protocolo Modbus em RTU (*Remote Terminal Unit*) com barramento série RS-485;
- Apresentar uma proposta de aplicação do protocolo com algoritmos de controlo para a área industrial com PLC´s e controlador PID;
- Utilizar o padrão OPC® para a criação de uma interface no *software* Scilab® para a monitorização das variáveis do processo a controlar.

Para o protocolo CAN os objectivos serão os seguintes:

- Um estudo detalhado do protocolo CAN;
- Apresentar uma proposta de aplicação do protocolo com algoritmos de controlo para a área industrial com arduinos acoplados a *shield* CAN e um controlador PID com *anti windup*;
- Visualizar a trama de dados detalhando os campos que a constituem, e a utilização do *software* Scilab para apresentação das variáveis do processo a controlar.

1.4 Organização da tese

Esta dissertação é constituída por 5 capítulos, incluindo este capítulo da introdução e está organizada da seguinte forma:

Capítulo 2 - Estado da Arte

Neste capítulo é feito o estudo detalhado dos protocolos CAN e Modbus recorrendo aos documentos disponibilizados pelas organizações que gerem estes protocolos. Na primeira secção é feita uma introdução às redes de comunicação industrial e as principais considerações em qualquer protocolo industrial tendo em conta aos escolhidos para esta dissertação. O estudo dos protocolos foi centralizado nas três camadas (camada de aplicação, camada de ligação de dados e camada física) da ISO/OSI (*Open System Interconnection*) elaborado pela ISO (*International Standardization Organization*). Para o protocolo Modbus na camada de ligação de dados, considerou-se o modo Modbus em linha série por ser o tipo de implementação mais utilizada.

Capítulo 3 - Arquitecturas, Tecnologias e Implementação

O capítulo 3 é dividido em duas secções, em cada uma delas é descrita um protocolo. Para o Modbus é apresentado a arquitectura proposta de implementação do protocolo e de seguida é feita a análise de cada secção da arquitectura, terminando com a análise das funções Modbus no PLC utilizadas na implementação, exemplos de leitura e escrita de *words* e a implementação da rede Modbus. O protocolo CAN segue a mesma filosofia do Modbus, arquitectura e análise dos seus respectivos blocos. De seguida são apresentadas as funções da biblioteca da *shield* CAN do arduino e a implemetação da rede CAN.

Capítulo 4 - Resultados experimentais

Neste capítulo serão apresentados os resultados dos testes dos barramentos e da implementação dos algoritmos de controlo com ambos os protocolos. Para o protocolo Modbus, os resultados da análise da comunicação serão apresentados a partir da tabela de animação do PLC cliente Modbus ou supervisor, e um gráfico para análise do processo a controlar. Para o protocolo CAN, os resultados da análise do barramento serão obtidos pelo osciloscópio analisando a trama de dados e será apresentado um gráfico do comportamento do processo a controlar.

Capítulo 5 - Conclusão

No quinto capítulo é elaborada a conclusão da dissertação e são apresentadas propostas de trabalho futuro para os estudos das redes e protocolos de comunicação industrial.

Capítulo 2

Estado da arte

Neste capítulo, é realizado o estudo dos protocolos Modbus e CAN, recorrendo a documentação disponibilizada pelas suas organizações (Organização Modbus e a CiA). Na secção 2.1, é feita uma introdução aos protocolos de comunicação industrial destacando o modelo da ISO e as principais considerações num protocolo de comunicação industrial.

Na secção 2.2, é feita a introdução ao protocolo Modbus.

A secção 2.3 descreve a camada de aplicação, destacando o seu modelo de endereçamento e os códigos das funções do protocolo Modbus. O protocolo Modbus pode ser implementado em linha série, TCP/IP e Modbus *Plus*, nesta dissertação, o estudo baseou-se somente no Modbus linha série.

Por isso, na secção 2.4 é feita a descrição do Modbus com barramento série, subsecção é apresentado o modo de transmissão mais utlizado, transmissão RTU, terminando a descrição do protocolo com a sua camada física na secção 2.5.

A secção 2.6 está reservada para o protocolo CAN, sendo feita a introdução ao protocolo CAN e descrita uma das principais características do protocolo que se chama arbitragem.

Na secção 2.7, é apresentada a descrição da camada de ligação de dados, destacando as duas principais tramas (*frames*), trama de dados e trama remota. A secção termina com a característica filtragem.

As duas últimas secções 2.8 e 2.9 estão reservadas para a camada física do protocolo e para a camada de aplicação.

2.1 Protocolos de comunicação industrial

A rápida evolução da electrónica, da engenharia de *software* e da miniaturização de componentes são os principais factores para o desenvolvimento dos sistemas de automação distribuída com redes de campo [26], [27].

Nos anos 40, a instrumentação de processos industriais operava com sinais de pressão de 3 - 15 psi[1] (*libra força por polegada ao quadrado*) para monitorar os dispositivos de controlo na instalação fabril. Já nos anos 60, os sinais analógicos de 4 - 20 mA foram introduzidos na indústria para medição e controlo de dispositivos. Com o desenvolvimento de processadores nos anos 70, surgiu a ideia de utilizar computadores para controlo de processos e proceder ao seu controlo. Nos anos 80, iniciou-se o desenvolvimento dos primeiros sensores inteligentes, bem como os sistemas de controlo digital [28], [27].

Tendo-se os instrumentos digitais, era necessário algo que pudesse interligá-los. Então, nasceu a ideia de uma rede[2] que conectasse todos os dispositivos de campo e disponibilizasse todos os sinais do processo num mesmo meio físico [28], [27]. A partir daí, a necessidade de uma rede era clara, assim como um padrão que pudesse deixá-lo compatível com o controlo de instrumentos digitais. As primeiras redes surgiram durante a decada de 80.

Uma rede de controlo industrial é um sistema de equipamentos interconectados utilizados para monitorarizar e controlar equipamentos físicos em ambientes industriais. A tecnologia em uso nas redes industriais também está começando a mostrar uma maior dependência dos padrões de *ethernet*, especialmente nos níveis mais altos da arquitetura de rede como se observa na figura 2.1 [29].

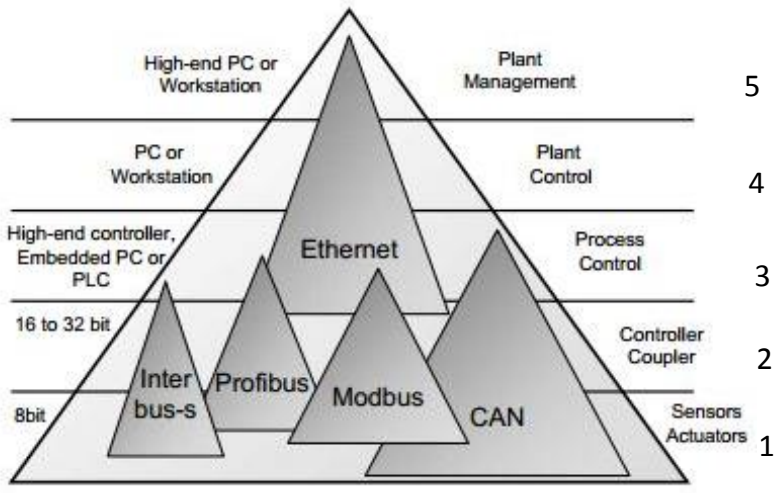


Figura 2.1: Protocolos na pirâmide de automação industrial [1].

[1]psi x 14.50368 = 1 bar — 1 psi = 0.068948 bar. O mesmo é igual a 6894,801 Pa (1 bar = 100.000 Pa).
[2]Também conhecida como *fieldbus* ou rede de campo

A figura 2.1 mostra a pirâmide de automação industrial, ela simboliza a hierarquia num sistema de automação industrial, e nela observa-se que os protocolos em estudo (Modbus e CAN) ocupam o nível dos sensores-actuadores, nível do controlador e o nível de supervisão [3]. O padrão *ethernet* ocupa os níveis superiores. A troca de dados entre o nível 1 (dispositivos de campo) e o nível 2 (controlo) pode também ser feita conectando directamente os dispositivos de campo nos pinos de entrada e saída I/O (*Input/Output*) de um controlador, tal como será visto mais adiante na arquitectura de implementação do protocolo Modbus com PLCs e na arquitectura de implementação do protocolo CAN com dispositivos Arduino.

Actualmente, a principal referência para a maioria dos protocolos actuais é o modelo de referência da ISO/OSI (*Open System Interconection*) elaborado pela ISO (*International Standardization Organization*) para apoiar o desenvolvimento e implementação de protocolos abertos. Com base nisto, as normas foram aprovadas pela ISO e IEEE (*Institute of Electrical and Electronic Engineers*), formando a base para uma comunicação aberta em áreas de escritórios e de indústrias [2].

De acordo com o modelo OSI, os sistemas de comunicação de dados são descritos por um modelo hierárquico que consiste em sete camadas de diferentes funcionalidades conforme a figura 2.2.

Figura 2.2: Modelo de referência da ISO.

Apenas três camadas (camada física, camada de ligação de dados e camada de aplicação) são geralmente relevantes para a comunicação de dados em aplicações de barramento. A camada de aplicação faz a interface entre os protocolos de comunicação e a aplicação que pediu ou recebe a informação pela rede. A camada de ligação de dados assegura que

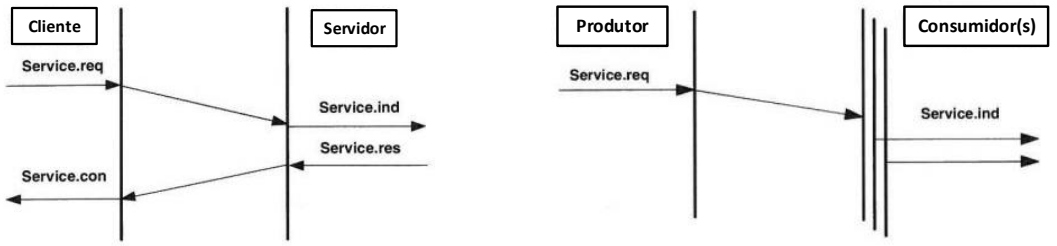
[3] O nível 1 é o nível de sensores e actuadores que contém sensores simples e actuadores utilizados no processo. O nível 2 é onde se encontram os equipamentos que executam o controlo automático das actividades do processo. O nível 3 permite a supervisão do processo, normalmente, possuindo base de dados com informações relativas ao processo.

o conteúdo da mensagem no local de destino seja exactamente igual à origem. A camada física é responsável pela transferência de *bits* num circuito de comunicação, a sua concepção deve relacionar-se com a definicão das interfaces eléctricas e mecânicas.

Principais considerações nos protocolos de comunicação industrial

Existem algumas considerações que se deve levar em conta na escolha de um protocolo de comunicação industrial. As considerações aqui descritas serão aquelas que mais se enquadram nos protocolos em estudo que são: modelo de comunicação de dados (cliente - servidor e produtor - consumidor), protocolos orientados por nós e protocolos orientados por mensagem, controlo de acesso ao meio MAC (*Medium Access Control*), topologia da rede e a forma de utilização do meio físico.

Os modelos de comunicação de dados cliente - servidor e produtor - consumidor, (figura 2.3) são os mais importantes paradigmas de comunicação de dados. O modelo cliente-servidor, (figura 2.3(a)) descreve uma relação de comunicação ponto a ponto (mesmo que haja vários servidores), é utilizado pelo protocolo Modbus e o padrão OPC. O modelo produtor-consumidor, (figura 2.3(b)) descreve uma relação de comunicação multiponto entre um produtor e um ou mais consumidores, sendo utilizado no protocolo CAN.



(a) Modelo Cliente - Sevidor. (b) Modelo Produtor - Consumidor.

Figura 2.3: Modelos de comunicação de dados [2].

Os protocolos de comunicação de dados podem ser distinguidos entre protocolos orientados por nós (*node-oriented protocols*) e protocolos orientados por mensagem (*message-oriented protocols*), conforme ilustrado na figura 2.4.

Nos protocolos orientados por nós, o caso do protocolo Modbus, a troca de dados entre dois ou mais nós é baseada no endereçamento do nó. Geralmente a trama (*frame*) de dados

transmitida sobre o meio de comunicação contém o endereço do nó. Assim uma trama é enviada para um nó específico ou grupo de nós (*broadcasting*) [2], [6].

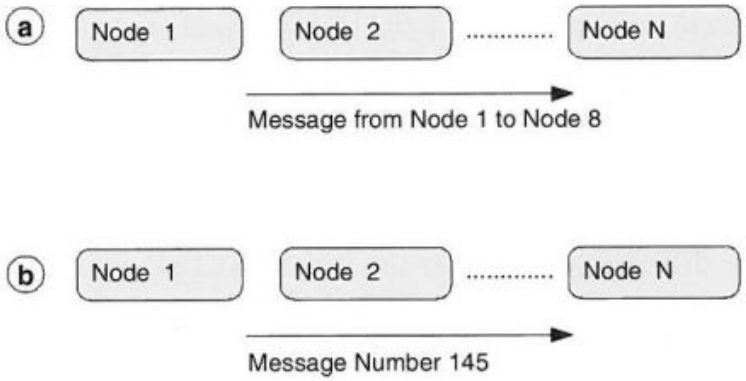


Figura 2.4: Comunicação de dados orientado por nó e por mensagem [2].

Nos protocolos orientados por mensagem, o caso do protocolo CAN, a troca de dados é baseada na trama ou no identificador da mensagem. A mensagem transmitida por um nó identifica-se com um único e específico identificador (identificador de mensagem), o destino de uma mensagem não é definido. É unicamente a decisão dos nós, ligados ao barramento, aceitar, ou não, uma mensagem transmitida. É possível por este meio que uma mensagem não seja aceite por nenhum nó, ou seja aceite por um ou muitos nós, conforme a figura 2.3(b). Todos os sistemas de comunicação de dados em automóveis, excepto aqueles para transferência de alto volume de dados, são baseados nos protocolos orientados por mensagem [2].

O controlo de acesso ao meio (MAC) é utilizado para arbitrar qual o transmissor que em uma rede ganha o acesso ao meio de transmissão. Ele é utilizado para evitar que dois ou mais nós transmitam dados ao mesmo tempo e, assim, evita que os sinais interfiram. O método do MAC, portanto, pode ser um critério decisivo para a seleção do protocolo de comunicação. Existem diferentes tipo de MAC, os principais e os utilizados nos dois protocolos em estudo são: Mestre - Escravo para o protocolo Modbus[4] e CSMA (*Carrier-Sense Multiple Access*) para o protocolo CAN. A descrição do MAC mestre - escravo será encontrada no protocolo Modbus (secção 2.2) e para o CSMA no protocolo CAN (secção 2.6).

A topologia de uma rede descreve a estrutura de conexão física entre os nós. As topolo-

[4]Modbus em TCP/IP utiliza o controlo de acesso ao meio CSMA-CD (*Carrier Sense Multiple Access with Collision Detection*)

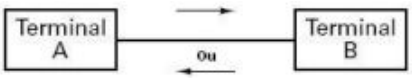
gias mais importantes são: estrela, barramento, anel e árvore. A topologia barramento é a que será aplicada na implementação de ambos os protocolos. A forma da utilização do meio físico que conecta os nós, representada na figura 2.5, dá origem à seguinte classificação:

Simplex: a ligação é utilizada apenas em um dos dois possíveis sentidos de transmissão figura 2.5(a).

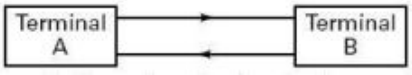
Half-duplex: a ligação é utilizada nos dois possíveis sentidos de transmissão, porém apenas um de cada vez, figura 2.5(b), protocolo CAN e Modbus.

Full-duplex: a ligação é utilizada simultâneamente nos dois possíveis sentidos de transmissão figura 2.5(c).

(a) Transmissão em somente uma direcção.



(b) Transmissão nas duas direcções, mas não simultâneamente.



(c) Transmissão nas duas direcções simultâneamente.

Figura 2.5: Comunicação *Simplex* (a), *Half-duplex* (b) e *Full-duplex* (c).

Transmissão série assíncrona os seus dados podem ser transmitidos de forma irregular, através do intervalo entre 2 *bits* fixos, protocolo Modbus.

Transmissão série síncrona os dados são transmitidos continuamente. Um sinal de sincronização é transmitido em paralelo com o sinal de dados.

Ambos os protocolos possuem o método de transmissão digital do tipo série, ou seja, os *bits* em um *byte* são enviados um depois do outro. Por razões de custo e durabilidade, muitas redes de comunicação usam transmissão digital série assíncrona *half-duplex*.

Nas secções seguintes serão feitas as descrições dos protocolos Modbus e CAN tendo em conta as principais considerações de um protocolo, as três camadas relevantes da ISO e as investigações feitas em volta destes dois protocolos. Seguindo uma ordem cronológica, será descrito no início o Modbus e em seguida o CAN.

2.2 Protocolo Modbus

No objectivo de criar um protocolo de comunicação industrial para os seus PLCs, a empresa Modicon criou o protocolo Modbus. No início o protocolo foi criado de forma aberta. Com essa intensão criou-se uma organização conhecida como Modbus-IDA. O propósito da Modbus-IDA é oferecer ajuda aos membros da organização, credenciando-os, divulgar o protocolo Modbus e os seus documentos de implementação. As informações relacionadas com o protocolo e a organização estão disponíveis no site `http://www.modbus.org/`.

Apesar de o protocolo ser antigo, muitos equipamentos actualmente na indústria têm suporte Modbus, o qual já passou por várias actualizações [25]. Modbus é um protocolo de aplicações de mensagens, posicionado no nível 7 do modelo OSI, que permite a comunicação cliente - servidor entre dispositivos conectados em diferentes tipos de topologias [3].

O protocolo Modbus foi criado em 1979 pela *Modicon Industrial Automation Systems* (actual *Schneider Electric*) e continua a permitir que vários dispositivos de automação comunicam entre si. O apoio para a estrutura simples do Modbus continua a crescer, geralmente o seu meio de comunicação é pela porta série, mas hoje a comunidade da internet pode aceder ao Modbus em TCP/IP, onde o seu paradigma é baseado em mestre - escravo [3].

A figura 2.6 elucida a estrutura das camadas do protocolo Modbus, sendo normalmente implementado utilizando:

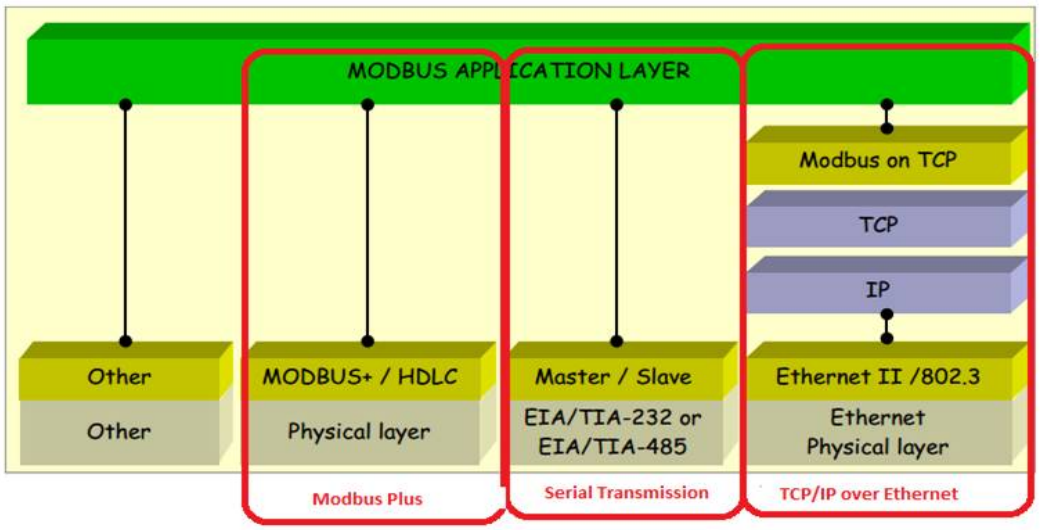


Figura 2.6: Estrutura das camadas do protocolo Modbus [3].

- TCP/IP sobre *ethernet*;

- Transmissão série assíncrona sobre vários meios (cabos, EIA/TIA-232-E, EIA-422, EIA/TIA-485-E);
- Modbus *Plus*, uma rede de alta velocidade com taxas de transmissão de 1 Mbps. O meio físico é o RS-485, com controlo de acesso ao meio por HDLC (*High Level Data Link Control*).

Uma das aplicações do protocolo Modbus em TCP/IP é apresentada em [4] (figura 2.7), em que se utiliza uma rede sem fios (*wireless*) em larga escala, uma arquitectura que foi implementada pela empresa brasileira Petrobrás em sua rede de distribuição de gás numa cidade do Brasil. A aplicação desta arquitectura embora sendo para uma comunicação *wireless*, os dispositivos de campo estão ligados ao *gateway* através de um barramento Modbus em linha série.

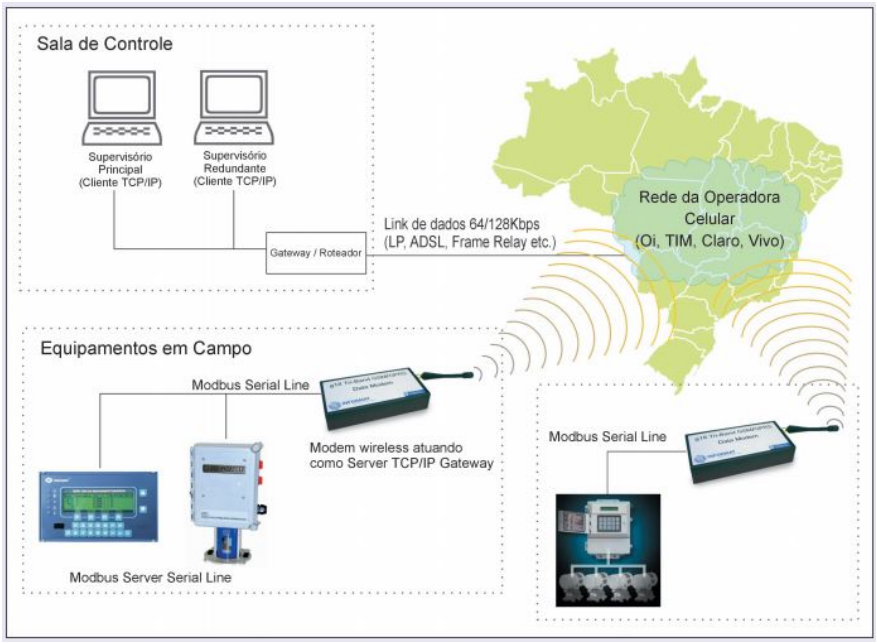


Figura 2.7: Aplicação do protocolo Modbus em comunicação sem fio [4].

No campo das redes industriais, o Modbus é o protocolo mais utilizado, já que diversos controladores e *softwares* para desenvolvimento de sistemas de supervisão utilizam este protocolo e por facilmente se adequar a diversos meios físicos. Isto deve-se à sua grande simplicidade e facilidade de implementação [30].

Salientando a grande utilização do protocolo Modbus para a criação de sistemas de supervisão, em [5] apresenta-se uma arquitectura Modbus em TCP/IP de um sistema de

supervisão utilizando o padrão OPC®[5] (figura 2.8). Muitas vezes, para o desenvolvimento de um sistema de supervisão recorre-se ao padrão OPC® porque facilita a comunicação de dados de distintos fabricantes. Nesta arquitectura (figura 2.8), o *software WinCC* foi utilizado como OPC cliente e como servidor um módulo OPC servidor e o cliente Modbus TCP/IP. O padrão OPC® será implementado nesta dissertação para a criação de um sistema de supervisão na rede de comunicação de dados Modbus.

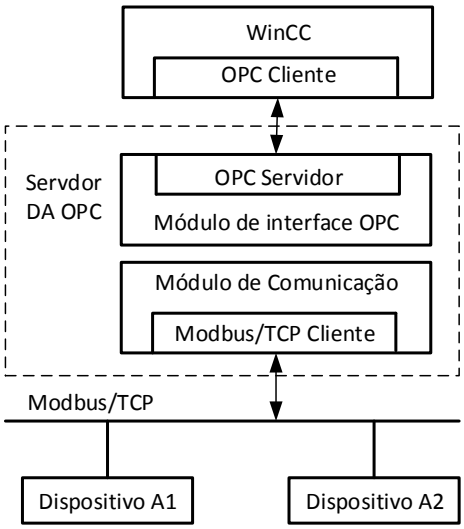


Figura 2.8: Arquitectura de uma comunicação Modbus em TCP/IP com o padrão OPC [5].

Nesta dissertação será feito o estudo do protocolo Modbus em linha série com uma transmissão série assíncrona RS-485, em *half-duplex*.

2.3 Protocolo de Aplicação Modbus

Uma comunicação Modbus é sempre iniciada pelo mestre (cliente Modbus). Os escravos (servidores Modbus) nunca transmitem uma mensagem sem receber uma requisição do mestre e também nunca comunicam entre si. O mestre inicia uma transação Modbus e aguarda o término desta para, então, iniciar uma nova transação [20]. O mestre emite requisições para os escravos de 2 modos:

- Modo *unicast*: o mestre endereça um escravo individualmente como se pode ver na figura 2.9. Depois de receber e processar a requisição, o escravo retorna uma resposta

[5]Padrão OPC® utiliza um protocolo de comunicação entre cliente e servidor.

para o mestre. Nesse modo, uma transação Modbus consiste em 2 mensagens: uma requisição do mestre e uma resposta do escravo [6];

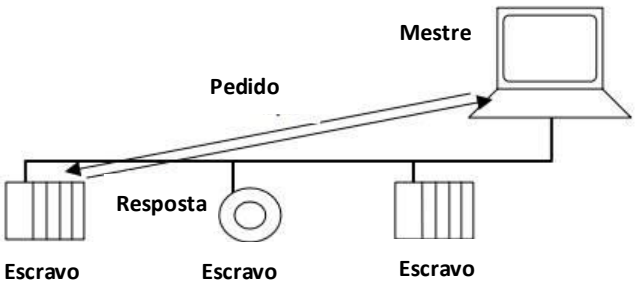


Figura 2.9: Modo *unicast* [6].

- Modo *broadcast*: o mestre pode enviar uma requisição para todos os escravos. Nenhuma resposta é retornada, pois as requisições são necessariamente comandos de escrita. Todos os dispositivos devem aceitar a mensagem *broadcast*, figura 2.10 [6].

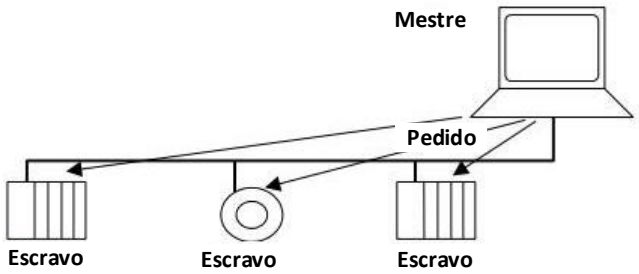


Figura 2.10: Modo *broadcast* [6].

O espaço de endereçamento Modbus compreende 256 endereços diferentes (tabela 2.1). O endereço 0 (zero) é reservado como endereço *broadcast*. Todos os escravos devem reconhecer o endereço *broadcast*. O mestre Modbus não possui endereço específico, apenas os escravos devem ter endereços. O endereço de cada escravo deve ser único em um barramento série Modbus [31].

Tabela 2.1: Regras de endereçamento Modbus [6].

0	ISO : [1 ; 247]	[248 ; 255]
Endereço *Broadcast*	Endereço Individual do Escravo	Reservado

2.3.1 Descrição do Protocolo Modbus

O protocolo Modbus define uma simples unidade de dados do protocolo, PDU (*Protocol Data Unit*), independentemente das outras camadas utilizadas na comunicação. O protocolo define três tipos de PDUs, que são:

1. PDU requisição Modbus;

2. PDU resposta Modbus;

3. PDU resposta de excepção (*Exception Response*) Modbus.

A unidade de dados de aplicação ADU (*Application Data Unit*) é construída pelo cliente que inicia a transação Modbus (figura 2.11).

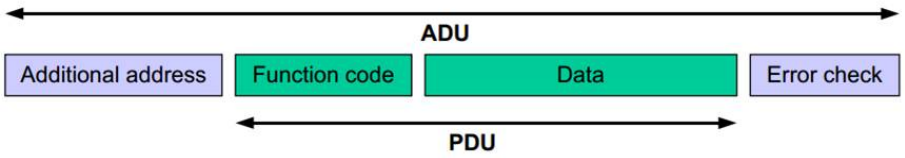


Figura 2.11: Trama geral Modbus [3].

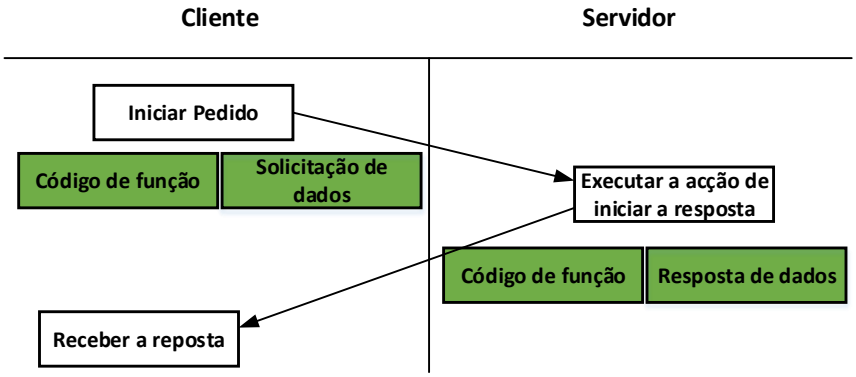
O código de função (*function code*) indica ao servidor qual o tipo de acção que deve realizar, sendo codificado em 1 *byte*. Os códigos válidos estão na gama de 1 a 255 decimal. A gama de 128 a 255 é reservada e utilizada para a *exception responses*. Quando uma mensagem é enviada de um cliente para um dispositivo servidor, o campo do código de função informa ao servidor que tipo de acção deve ser executada. O código de função 0 (zero) não é válido.

O campo dados (*data*) da mensagem enviada de um cliente para o dispositivo servidor existe informação adicional, que o servidor utiliza para tomar as medidas definidas pelo código de função. Isto pode incluir itens como endereços discretos e de registro, quantidade de itens a serem tratados, e a contagem de *bytes* de dados[6].

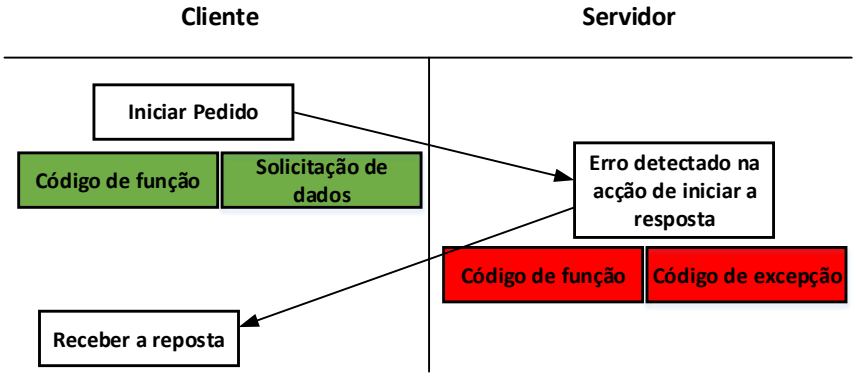
Se nenhum erro ocorrer em uma comunicação Modbus, o campo de dados da resposta da ADU enviada pelo servidor contém os dados requisitados, conforme a figura 2.12(a). Na ocorrência de algum erro, o campo de dados da resposta da ADU contém um *exception*

[6]No capítulo 3 (secção 3.1.5) o campo de endereço do escravo, o código de função e o campo de dados corresponderá a tabela de transmissão

code[7] que a aplicação do servidor pode utilizar para determinar a próxima acção a ser tomada, (figura 2.12(b)) [3].



(a) Comunicação Modbus sem erro.



(b) Comunicação Modbus com *exception response*.

Figura 2.12: Comunicação Modbus [3].

Quando o servidor responde ao cliente, este utiliza o campo de código de função para indicar uma resposta normal (livre de erros), ou que algum tipo de erro ocorreu (chamado de *exception response*). Para uma resposta normal, o servidor simplesmente copia para a resposta o código da função original.

O protocolo Modbus utiliza uma representação *"big-Endian"* para endereços e itens de dados. Isto significa que, quando uma quantidade numérica maior do que um único *byte* é transmitido, o *byte* mais significativo é enviado primeiro [8] [3].

[7] A tabela dos códigos de excepção será apresentada na implementação do protocolo (capítulo 3).
[8] Por exemplo no valor 0x1234 (16 *bits*) o primeiro *byte* a ser enviado é o 0x12 e em seguida o 0x34.

2.3.2 Modelo de dados e de endereçamento Modbus

O protocolo Modbus de aplicação baseia o seu modelo de dados numa série de tabelas que possuem características distintas. Tais características são elucidadas na tabela 2.2.

Tabela 2.2: Características das tabelas Modbus [3].

Tabela Primaria	Tipo de Objectivo	Tipo	Comentários
Entrada discreta	Um *bit*	Somente Leitura	Este tipo de dado pode ser fornecido por um sistema I/O
Bobinas	Um *bit*	Leitura e Escrita	Este tipo de dado pode ser alterado por um programa de aplicação
Entrada de registros	16-*bit word*	Somente Leitura	Este tipo de dado pode ser fornecido por um sistema I/O
Registros *Holding*	16-*bit word*	Leitura e Escrita	Este tipo de dado pode ser alterado por um programa de aplicação

Para cada uma das tabelas primárias, o protocolo permite a selecção individual de 65536[9] itens de dados, e as operações de leitura ou escrita desses itens são desenhadas para abrangirem vários itens de dados consecutivos até um limite de tamanho de dados que é dependente do código de função de comunicação.

Os dados manipulados via Modbus estão localizados na memória do dispositivo. Endereços físicos não devem ser confundidos com a referência dos dados, que é utilizada nas funções Modbus. A referência lógica é do tipo inteiro sem sinal começando em 0 (zero).

O protocolo Modbus define precisamente regras de endereçamento da PDU: em uma PDU Modbus cada dado é endereçável de 0 a 65535[10]. No modelo de dados Modbus cada elemento dentro de um bloco de dados é numerado de 1 a *n*. Um dado Modbus numerado como x é endereçável em uma PDU Modbus como $(x\text{-}1)$[11], figura 2.13.

Os principais códigos de funções Modbus estão listados na tabela 2.3. Detalhes dos códigos de funções é possivel encontrar em [21].

[9] Por exemplo 1- Entrada discreta; 2 - Bobinas; sucessivamente 3 e 4. 1:1-65536; 2:1-65536

[10] Isto significa que as PDU´s Modbus, ou seja, os *bits* e as *words* no PLC são endereçáveis no intervalo de 0 a 65535

[11] Por exemplo uma entrada de registros memória *word* %MW9 é endereçável como 3:10

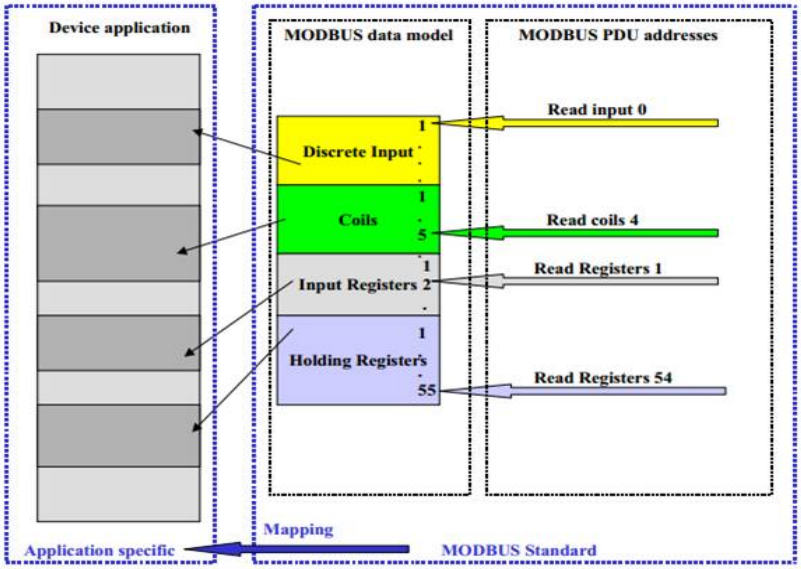


Figura 2.13: Modelo de endereçamento Modbus [3].

Tabela 2.3: Principais Códigos de funções Modbus [3].

Códigos de Funções		**Descrição**			
Dec	Hex				
02	02	Read Discrete Inputs	Physical Discrete Inputs	Bit acess	Data Acesss
01	01	Read Coils			
05	05	Write Single Coil	Internal Bits Or Physical coils		
15	0F	Write Multiple Coils			
04	04	Read Input Registers	Physical Input Registers	16 bits acess	
03	03	Read Holding Registers			
06	06	Write Single Register	Internal Registers Or Physical Output Registers		
16	10	Write Multiple Register			
07	07	Read Exception status	Diagnostic		
08	08	Diagnostic			

Uma vez que o pedido tenha sido processado por um servidor, uma resposta é construída. Dependendo do resultado do processamento, dois tipos de resposta são construídas:

1. Uma resposta Modbus : a resposta do código da função será igual ao pedido do código da função;

2. Uma Modbus *exception response* : o objetivo é proporcionar ao cliente informações relevantes sobre o erro detectado durante o processamento; O código da função *exception* será igual ao código da função da requisição adicionado por 0x80 [12]. Um código de *exception* é fornecido para indicar o tipo de erro.

2.4 Protocolo Modbus em Linha Série

Tal como já se referiu nas secções anteriores, nesta dissertação versa sobre o estudo e a implementação foi realizado com um protocolo Modbus em linha série, utilizando a transmissão em RTU (*Remote Terminal Unit*), porque a sua maior densidade de caracteres permite melhores rendimentos em comparação com o modo de transmissão ASCII (*American Standard Code for Information Interchange*).

O protocolo Modbus em linha série é um protocolo mestre - escravo. Isto significa que existe apenas um mestre ligado ao barramento. Quanto aos escravos (máximo de 247), dependendo da interface física que se esteja a utilizar, um ou mais podem estar simultâneamente ligados ao mesmo barramento [6].

Na tradicional arquitectura de 7 camadas da ISO, (figura 2.2 em secção 2.1), o protocolo Modbus sobre linha série está posicionado em 3 camadas, tabela 2.4.

Tabela 2.4: Modbus em linha série utiliza um modelo de 3 camadas [20].

Camada	Funções ISO/OSI	Funções Modbus
7	Aplicação	Protocolo de aplicação Modbus
3-6	Vários	Nulo
2	Ligação de dados	Protocolo Modbus em linha Série
1	Física	EIA-485, EIA-232

No topo está a camada de aplicação, esta é denominada de protocolo de aplicação Modbus ou simplesmente Protocolo Modbus, tal como foi descrito na secção 2.3. As camadas 3 a 6 não são utilizadas. A ligação de dados (camada 2) é ocupada pelo protocolo

[12] Por exemplo código de função 01 + 80 = 81. Para funções como 10 será igual a 10 + 80 = 90.

Modbus linha série. Finalmente, a camada física (camada 1) permite implementação, tanto para o EIA-232 ou EIA-485 [20].

Como camada física pode-se utilizar RS-232 ou RS-485. A interface RS-485 de 2 fios é a mais comum e é possível ter vários escravos, e, apenas um único mestre. A interface RS-232 pode ser utilizada para comunicação ponto a ponto de curta distância, com apenas um mestre e um escravo. No protocolo Modbus em linha série, o papel do cliente é proporcionado pelo mestre do barramento e os dispositivos escravos actuam como servidores [20].

Já foi descrito na subsecção 2.3.1 que a aplicação Modbus define uma simples PDU independentemente das outras camadas utilizadas na comunicação. O mapeamento do protocolo Modbus para barramentos específicos ou redes pode introduzir alguns campos adicionais na PDU.

A figura 2.14 representa o protocolo Modbus sobre linha série, o campo de endereço (*adress field*) contém somente o endereço do escravo. E como se viu na tabela 2.1, os endereços dos escravos estão no intervalo de [0 ; 247] em decimal, aos dispositivos individuais do escravo são atribuídos endereços na gama de [1 ; 247]. O mestre endereça um escravo, colocando o seu respectivo endereço de escravo no campo de endereço da ADU. Quando o escravo retorna a resposta, ele coloca o seu próprio endereço no campo de endereço da resposta fazendo com que o mestre saiba qual o escravo que está respondendo.

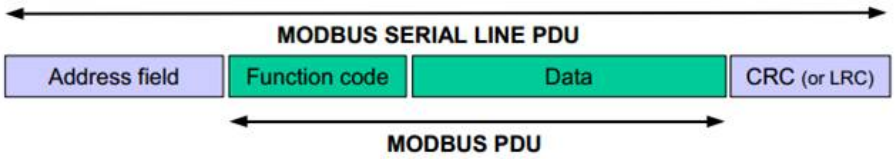


Figura 2.14: Trama Modbus sobre linha série [6].

O código de função (*Function code*) indica ao servidor que tipo de acção a ser executada. O código de função pode ser seguido por um campo de dados que contém os parâmetros de requisição ou solicitação (*request*) e resposta (*response*).

O campo de verificação de erros (*Error Checking*) é o resultado de uma verificação de redundância, cálculo que é realizado sobre os conteúdos da mensagem. Dois métodos de cálculo são utilizados, dependendo do modo de transmissão que está sendo utilizado (RTU ou ASCII).

2.4.1 Diagrama de estado Mestre/Escravo

A camada de ligação de dados em linha série compreende 2 sub-camadas separadas: protocolo mestre - escravo e modo de transmissão (RTU e ASCII), ambas as subcamadas serão descritas adiante.

Para a subcamada protocolo mestre - escravo, a figura 2.15 representa o diagrama de estado mestre. O estado *"Idle"* é o estado inicial depois do *power-up*, uma requisição somente pode ser enviada neste estado. Quando uma requisição *unicast* é enviada para um escravo, o mestre entra em estado de espera da resposta (*"Waiting for reply"*), e um tempo de resposta (*"Response Time-out"*) é iniciado. O valor deste tempo é dependente da aplicação.

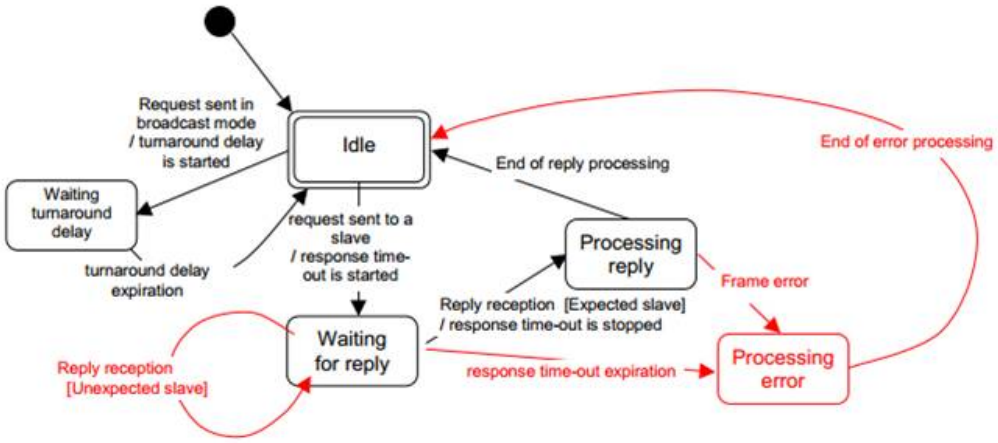


Figura 2.15: Diagrama de estado mestre [6].

Quando uma resposta é recebida, o mestre verifica a resposta antes de iniciar o processamento. A verificação pode resultar num erro da trama ou de uma resposta recebida de um escravo inesperado. Se não houver resposta, o tempo de espera expira e um erro é gerado, e o mestre entra em estado *"Idle"*. Quando uma requisição *broadcast* é enviada no barramento série (figura 2.15) nenhuma resposta é retornada a partir dos escravos. No entanto, um atraso é respeitado pelo mestre, a fim de permitir que qualquer escravo processe a requisição atual antes de enviar uma nova. Este atraso é chamado de *"Turnaround delay"*. Portanto, o mestre vai para o estado *"Waiting Turnaround delay"* antes de voltar ao estado "*Idle*"e de enviar uma outra requisição.

O diagrama de estado do escravo comporta-se em função das mensagens solicitadas pelo mestre, como observado na figura 2.16.

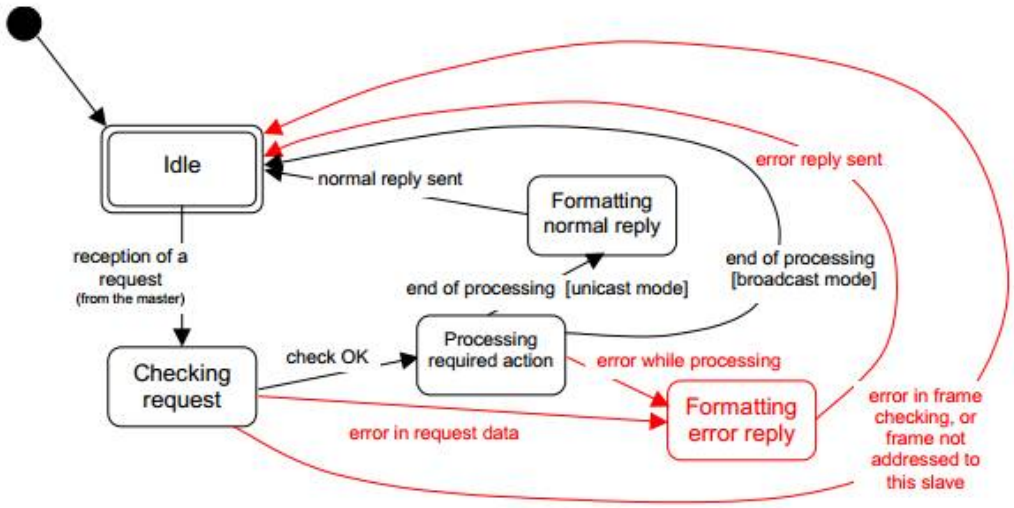


Figura 2.16: Diagrama de estado escravo [6].

No estado *"Idle"* nenhuma requisição encontra-se pendente, este é o estado inicial depois do *power-up*. Quando uma requisição é recebida, o escravo verifica os dados recebidos antes de executar a acção solicitada. Diferentes erros podem ocorrer: erro de formato da requisição, acção inválida. Em caso de erro, a resposta deve ser enviada ao mestre. Uma vez que a acção necessária tenha sido concluída, ou seja, nenhum erro na verificação da requisição, uma mensagem *unicast* requer que uma resposta seja formatada e enviada para o mestre. Se o escravo detecta um erro na trama recebida, nenhuma resposta é devolvida para o mestre. A figura 2.17 representa o diagrama temporal da comunicação mestre - escravo.

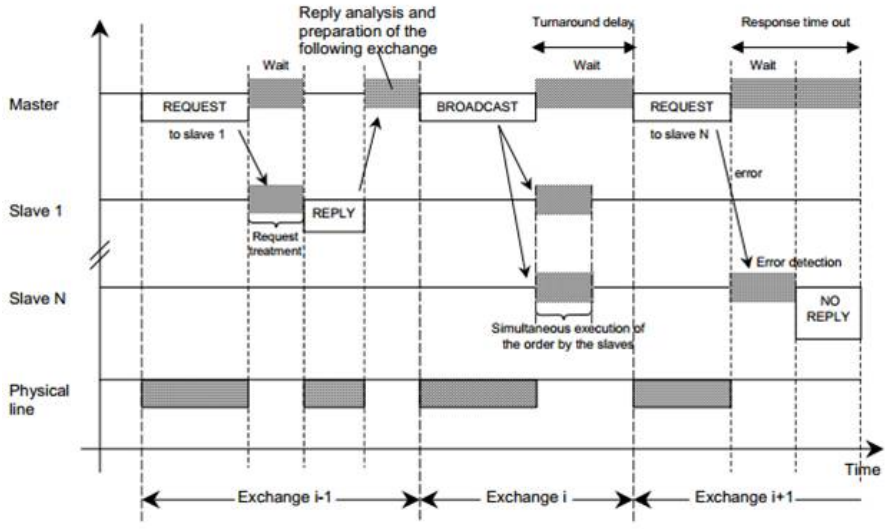


Figura 2.17: Diagrama temporal da comunicação mestre - escravo [6].

Em *unicast* o tempo de resposta deve ser definido tal que o escravo possa processar a requisição e retornar a resposta. Em *broadcast* o *Turnaround delay* deve ser longo de modo que qualquer escravo possa processar apenas a requisição e ser capaz de receber uma nova. Portanto, o *Turnaround delay* deve ser menor do que o tempo de resposta. Normalmente, o tempo de resposta varia entre 1 *s* e vários segundos, a 9600 *bps*, e o *Turnaround delay* é de 100 a 200 *ms*.

2.4.2 Modo de Transmissão RTU

Existem dois modos definidos de transmissão série: o modo RTU e o ASCII. O modo de transmissão define o *bit* no campo do conteúdo da mensagem transmitida em linha série, e determina como a informação é empacotada e descodificada dentro do campo da mensagem. O modo de transmissão deve ser o mesmo para todos os dispositivos no barramento série Modbus.

Embora o modo ASCII seja necessário em algumas aplicações específicas, a interoperabilidade entre os dispositivos Modbus só pode ser alcançada se cada dispositivo tiver o mesmo modo de transmissão. A configuração padrão deve ser o modo RTU [6].

A segurança do barramento série Modbus é baseada em dois tipos de verificação de erros [6]:

1. Verificação de paridade (par ou ímpar) deve ser aplicada a cada caracter.
2. Verificação da trama (LRC ou CRC) deve ser aplicada à mensagem inteira.

Quando os dispositivos comunicam entre si em linha série Modbus, utilizando o modo RTU, cada *byte* (8 *bits*) em uma mensagem, contém dois caracteres hexadecimais de 4 *bits*. A principal vantagem deste modo é que a sua maior densidade de dados de caracteres permite melhores rendimentos que o modo ASCII para a mesma gama de transmissão. Cada mensagem tem de ser transmitida em um fluxo contínuo de caracteres. O formato (11 *bits*) para cada *byte* em modo RTU é [21]:

1 start bit + 8 bits de dados + 1 bit de paridade + 1 stop bit

Para além da paridade par, (tabela 2.5) também podem ser utilizados outros modos: paridade ímpar ou sem paridade. A fim de garantir uma compatibilidade elevada com

outros produtos, recomenda-se o uso do modo sem paridade. 0 uso do modo RTU sem paridade requer 2 *bit* stop (tabela 2.6).

Tabela 2.5: Sequência de *bits* no modo RTU, com paridade par [21].

Verificação com Paridade										
Start	1	2	3	4	5	6	7	8	Par	Stop

Tabela 2.6: Sequência de *bits* no modo RTU, caso sem paridade [21].

Verificação sem Paridade										
Start	1	2	3	4	5	6	7	8	Stop	Stop

O modo padrão de paridade deve ter paridade par [6]. Os caracteres ou *bytes* são serialmente transmitidos ou enviados na seguinte ordem (esquerda para direita)[13]: *bit menos significativo ... bit mais significativo*

No modo de transmissão RTU o campo de verificação da trama é o CRC e a sua descrição pode ser observada na tabela 2.7. O tamanho máximo da trama Modbus RTU é de 256 *bytes*.

Tabela 2.7: Trama da mensagem RTU [6].

Endereço do Escravo	Código de função	Dados	CRC
1 *byte*	1 *byte*	0 até 252 *byte*(s)	2 *bytes* CRC baixo/CRC alto

Uma mensagem Modbus é colocada pelo transmissor em uma trama que possui pontos de início e fim bem definidos. Assim, os dispositivos que recebem mensagens podem detectar o início das mensagens quando elas são completadas. No modo RTU, as tramas das mensagens são separadas por um silêncio na linha de no mínimo 3,5 caracteres, conforme ilustrado na figura 2.18

[13]Embora os *bits* dos *bytes* sejam transmitidos do menos significativo ao mais significativo os bytes são transmitidos do mais signifivo ao menos significavo.

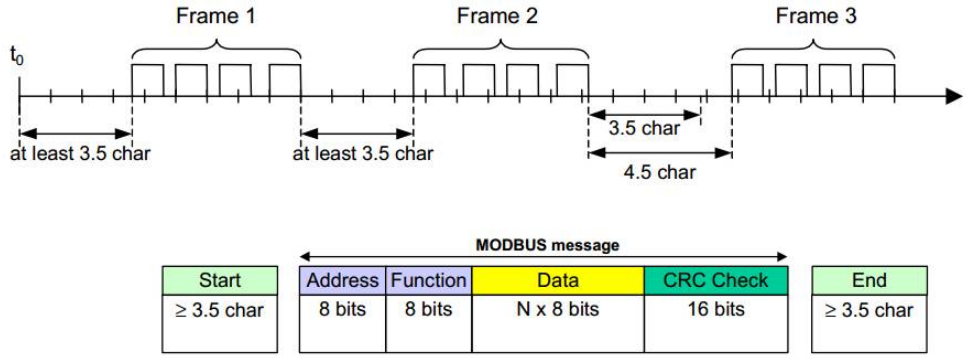


Figura 2.18: Trama da mensagem RTU [6].

A trama da mensagem deve ser transmitida como um fluxo contínuo de caracteres. Se um silêncio maior que 1,5 caracteres ocorrer entre 2 caracteres, a trama de mensagem é considerada incompleta e deve ser descartada pelo receptor, conforme a figura 2.19 [6].

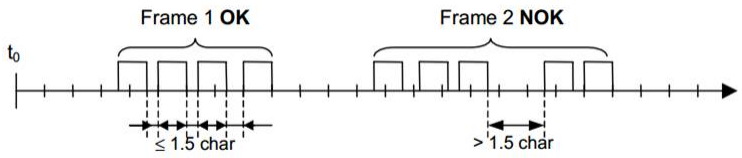


Figura 2.19: Silêncio entre caracteres [6].

O modo RTU inclui um campo de verificação de erros, que se baseia no método CRC realizado sobre o conteúdo da mensagem. O campo CRC verifica o conteúdo da mensagem inteira. É aplicado independentemente de qualquer método de verificação de paridade utilizado para os caracteres individuais da mensagem [21].

O campo CRC compreende um valor de 16 *bits* implementado como dois *bytes* de 8 *bits* e é anexado à mensagem no último campo. Quando isso é feito o campo do *byte* de baixa ordem (*byte low-order*) é colocado primeiro, seguido do *byte* de alta ordem (*byte de high-order*). O *byte* de alta ordem CRC é o último a ser enviado na mensagem.[14] O valor CRC é calculado pelo dispositivo de envio, que acrescenta o CRC na mensagem. O dispositivo receptor recalcula o CRC durante a recepção da mensagem e compara o valor calculado com o valor real que recebeu no campo CRC. Se os dois valores não forem iguais, ocorrerá um erro.

Tanto a verificação de caracteres e a verificação da trama da mensagem são geradas

[14]Por exemplo o cálculo CRC da trama 02 07 em hexadecimal resulta em 12 41 em hexadecimal, mas o CRC enviado é 41 12 em hexadecimal.

no dispositivo (mestre ou escravo) que emite e é aplicado ao conteúdo das mensagens antes da transmissão. O dispositivo (mestre ou escravo) verifica cada caracter e toda a trama da mensagem durante o recepção da trama. A figura 2.20 fornece uma descrição da transmissão em diagrama de estado do modo RTU do mestre do escravo.

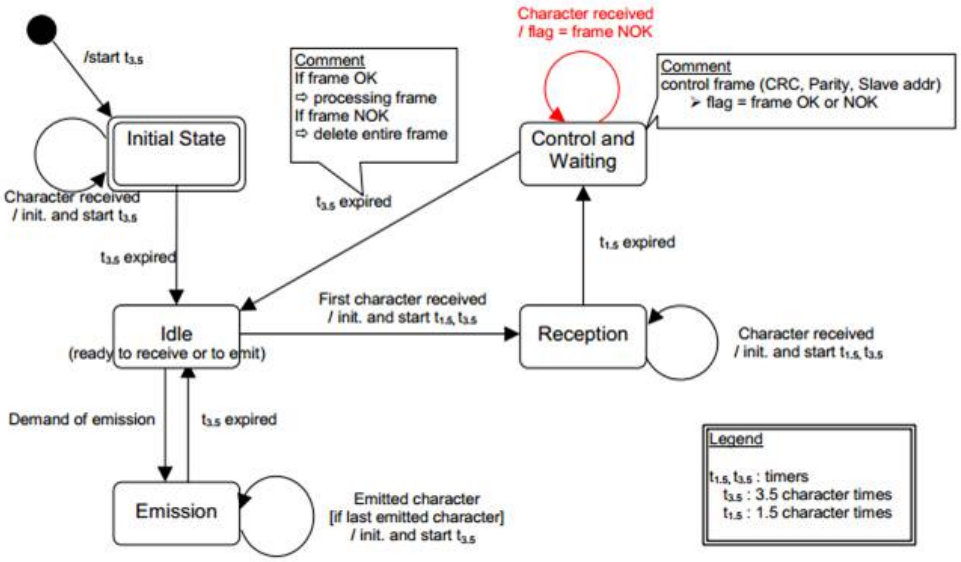


Figura 2.20: Diagrama de estados da transmissão RTU [6].

Em [7], o protocolo Modbus no modo RTU é utilizado no sistema de controlo do nível de líquido na flotação de minerais. Com base no protocolo Modbus RTU, segundo o autor, obteve-se um bom desempenho de controlo e assegurou-se uma boa separação do mineral. A arquitectura é constituida por um PLC S7-300 da Siemens, como mestre Modbus, e por vários escravos, figura 2.21. Este PLC permite ainda uma comunicação *ethernet*.

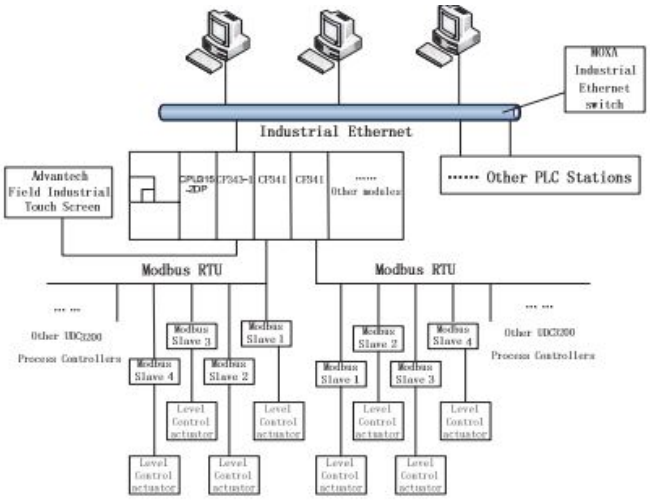


Figura 2.21: Arquitectura de uma comunicação Modbus RTU e *Ethernet* [7].

2.5 Camada física do protocolo Modbus

O protocolo Modbus previa uma conexão ponto-a-ponto EIA-232 entre um computador e um PLC. Essa opção permanece até hoje. Mas o Modbus sobre linha série incentiva o uso do padrão EIA-485 multiponto - com suporte até 32 dispositivos através de um barramento comum. Este padrão permite sistemas ponto a ponto e multiponto, em uma configuração de dois fios (*two-wire*). Além disso, alguns dispositivos podem implementar uma interface RS-485 a quatro fios (*four-wire*) [20].

A figura 2.22 mostra uma interface de dois fios recomendado pela EIA-485, onde existe um mestre e vários escravos ligados a um barramento comum de dois fios (D1 e D0) com resistores terminadores de 120 Ohms. Embora seja chamado de barramento de dois fios, existe uma conexão comum. O protocolo Modbus sobre linha série suporta interface de 2 e 4 fios, a implementação de 2 fios é a mais popular. É possível ter uma conexão *full-duplex* com 4 fios, mas o protocolo Modbus é estritamente *half-duplex*. O mestre emite comandos para um determinado escravo, enquanto aguarda a resposta do escravo. E isso é feito de forma bastante eficaz com uma implementação a 2 fios [20].

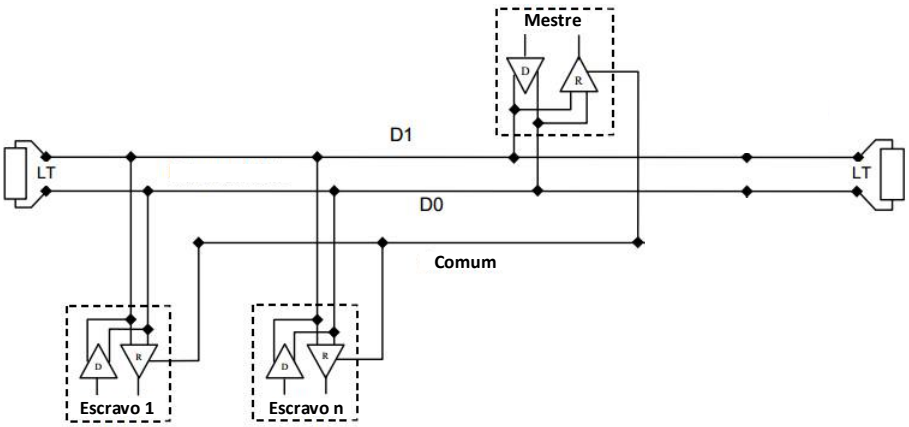


Figura 2.22: Topologia geral com dois fios [6].

No sistema Modbus padrão todos os dispositivos estão conectados (em paralelo) em um cabo constituído por 3 condutores. Dois desses condutores (configuração a "dois fios") formam um par equilibrado, em que os dados bi-direccionais são transmitidos. RJ45 ou conectores D-shell 9 (porta série) podem ser utilizados nos dispositivos para conectar os cabos. Por norma, a taxa de transferencia no protocolo Modbus é 19,2 kbps. Mas outras taxas podem ser utilizadas, como por exemplo: 1200, 2400, 4800, ... 38400 bps, 56 kbps,

115 kbps [6].

Muitos têm sido os trabalhos envolvendo PLCs, redes industriais e o padrão OPC®. A figura 2.23 apresenta uma arquitectura desenvolvida em [8] para o controlo de um motor de indução com uma rede *Profinet*, *Profibus* e incluindo o padrão OPC®. O motor foi controlado por um PLC baseado num controlador PID. Os resultados obtidos desta arquitectura, segundo o autor, demonstraram a possibilidade de estabelecer um controlo local e remoto, assegurando boa estabilidade e eficiência do sistema.

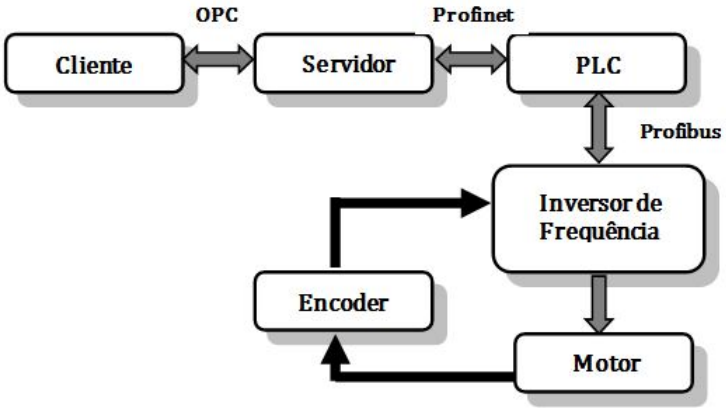


Figura 2.23: Arquitectura de um sistema de controlo com Profinet, Profibus e o padrão OPC [8].

O protocolo Modbus é muito utilizado a nível industrial mas muitas vezes há a necessidade de que um certo protocolo troque dados com um outro protocolo. Já existem interfaces Modbus - CAN e muitos estudos têm sido feito em volta destas interfaces (Modbus - CAN ou CAN - Modbus) porque estes dois protocolos representam os barramentos mais populares em sistemas de controlo industrial. Nas próximas secções será feita a descrição do protocolo CAN, e no final serão apresentados alguns estudos feitos em volta das interfaces entre o CAN e o Modbus.

2.6 Protocolo CAN

Em fevereiro de 1986, a empresa Robert Bosch GmbH introduziu o sistema de barramento série CAN (*Controller Area Network*) no congresso da SAE (*Society of Automotive Engineers*). Era a hora do nascimento de um dos protocolos de rede mais bem sucedidos de sempre. CAN foi especificamente projectado para ser robusto em ambientes electromagneticamente ruidosos [32]. Embora inicialmente criado para o mercado automóvel para reduzir o peso, o custo da cablagem e adicionar alguns recursos, actualmente CAN também é utilizado em muitas aplicações de automação industrial, medicina, marinha, serviço militar e em áreas em que seja necessário uma rede simples mas robusta [11], [33].

No início de 1992, os utilizadores e fabricantes estabeleceram a CiA (*CAN in Automation*), associação internacional dos utilizadores e fabricantes. Uma das primeiras tarefas da CiA foi a especificação da CAL (*CAN Application Layer*) camada de aplicação CAN [1]. As especificações do protocolo CAN são definidas pela associação CiA e estão parcialmente acessíveis no site `http://www.can-cia.org` [34].

Hoje quase todos os novos autocarros fabricados na Europa estão equipados com pelo menos uma rede CAN. Também utilizado em outros veículos, bem como em controlo industrial, CAN é um dos protocolos mais dominantes [1]. Vários têm sido os trabalhos para o mercado automóvel baseados em CAN, isso mostra a grandiosidade deste protocolo nesta área. A taxa de transmissão é limitada pelo comprimento do barramento utilizado, conforme a tabela 2.8 e as características do barramento CAN são resumidas de seguida [35]:

Tabela 2.8: Taxa de transmissão *versus* distância para barramento CAN [2].

Taxa (Kbit/s)	Distância máx. (m)	Nominal *bit time* (μs)
1000	25	1
800	50	1.25
500	100	2
250	250	4
125	500	8
50	1000	20
20	2500	50
10	5000	100

- Atribui prioridade às mensagens e apresenta flexibilidade de configuração;
- Possui capacidade multimestre;

- Obedece ao conceito de *broadcast*;
- Tem um esquema de arbitragem não destrutiva (*bitwise arbitration*) descentralizada, baseada na adopção dos níveis "dominantes"e "recessivos", sendo utilizado para controlar o acesso ao barramento;
- As mensagens de dados são pequenas (no máximo oito *bytes* de dados);
- Não há endereço explícito nas mensagens. Em vez disso, cada mensagem carrega um identificador que controla a sua prioridade no barramento e que pode servir como uma identificação do seu conteúdo;
- Utiliza um elaborado esquema de tratamento de erros que resulta na retransmissão das mensagens que não são apropriadamente recebidas;
- Fornece meios efectivos para isolar falhas e remover nós com problemas do barramento;
- O meio físico de transmissão pode ser escolhido de acordo com as necessidades. O mais comum é o par trançado[15], mas também podem ser utilizados outros meios de transmissão, tais como fibra óptica e radiofrequência.

O protocolo CAN foi internacionalmente padronizado em 1993 como ISO 11898 no modelo de referência de 7 camadas ISO/OSI (tabela 2.9). Na primeira edição de 2003 da ISO 11898-1, juntamente com ISO 11898-2 substituiram a ISO 11898 de 1993 [36]. Tal como no protocolo Modbus, o protocolo CAN define apenas três das camadas de referência da ISO, sendo estas a camada física, camada de ligação de dados e a camada de aplicação, conforme ilustrado na tabela 2.9.

Tabela 2.9: Modelo OSI/ISO do protocolo CAN [22].

Camada	Função	Especificação
7	Aplicação	Especificado pelo Projectista
6	Apresentação	Vazio
5	Sessão	Vazio
4	Transporte	Vazio
3	Rede	Vazio
2	Ligação de dados	Coberto pelo CAN e padrão ISO
1	Física	Coberto pela ISO e parcialmente pela CAN

[15]Também denominado de par enterlaçado.

Inicialmente, somente as camadas de ligação de dados e física eram utilizadas no CAN, mas devido ao desenvolvimento do protocolo, CAN sofreu várias actualizações e consequentemente surgiu uma nova camada, a camada de aplicação. Várias empresas desenvolveram aplicações de *software* para esta nova camada e apesar do desenvolvimento das aplicações de *software* dessas empresas, vários programadores acrescentaram as suas camadas de aplicação, visto que microcontroladores implementam a camada de ligação de dados e a camada física.

Nesta dissertação serão utilizadas placas Arduinos acoplados a *shields* CAN e como camada de aplicação foi utilizado o ambiente de desenvolvimento do arduino e a linguagem C, como descrito em capítulo 3.

O protocolo CAN foi desenvolvido com a versão 1.2, seguido pela versão 2.0A e 2.0B. As versões são compatíveis. Diferentes versões do protocolo podem ser, com algumas limitações, operadas em uma única rede. CAN 2.0A é idêntico ao 1.2, os identificadores possuem 11 *bits* de comprimento enquanto o CAN 2.0B pode ter tanto identificadores de 11 *bits* (Controladores CAN 2.0B passivos) como de 29 *bits* (Controladores CAN 2.0B activos) chamados de trama estendida [37].

CAN v2.0A. Nesta versão a estrutura é constituida por camada física, camada de ligação de dados e camada de aplicação (ver figura 2.24).

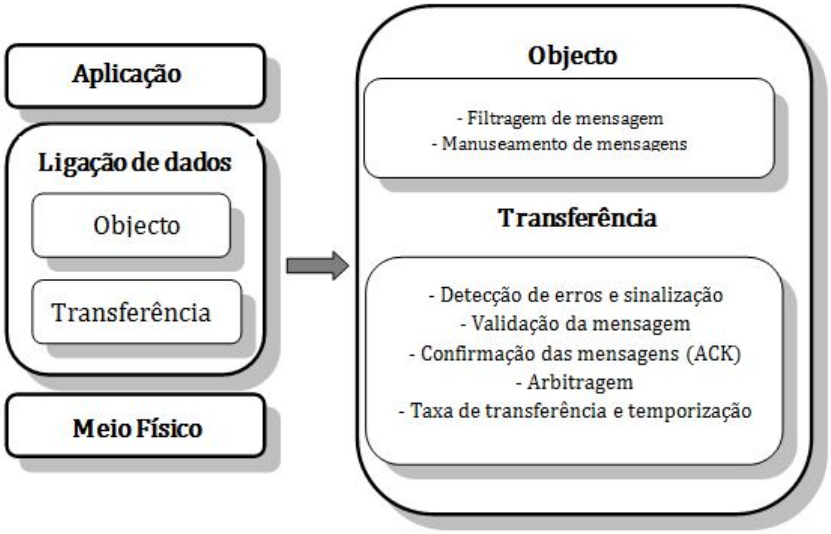


Figura 2.24: CAN 2.0A [9].

A camada física define como os sinais são realmente transmitidos e tem as seguintes características: nível do sinal, *bit* de representação e transmissão ao meio. A camada de transferência representa o núcleo do protocolo CAN. Apresenta as mensagens recebidas para a camada de objecto e recebe as mensagens que serão enviadas a partir da camada de objecto. A camada de objecto responsabiliza-se pela filtragem e manuseamento das mensagens. A camada de aplicação está totalmente vazia relativamente ao que interessa no protocolo CAN.

CAN v2.0B. A versão CAN 2.0B surge depois da versão CAN 2.0A. No entanto, teve algumas alterações nas camadas mais baixas (ligação de dados e camada física). A figura 2.25 mostra estas alterações. Na estrutura desta versão o meio físico foi dividido em três subcamadas: PLS (*Physical Signalling*), PMA (*Physical Medium Attachment*) e MDI (*Medium Dependent Interface*).

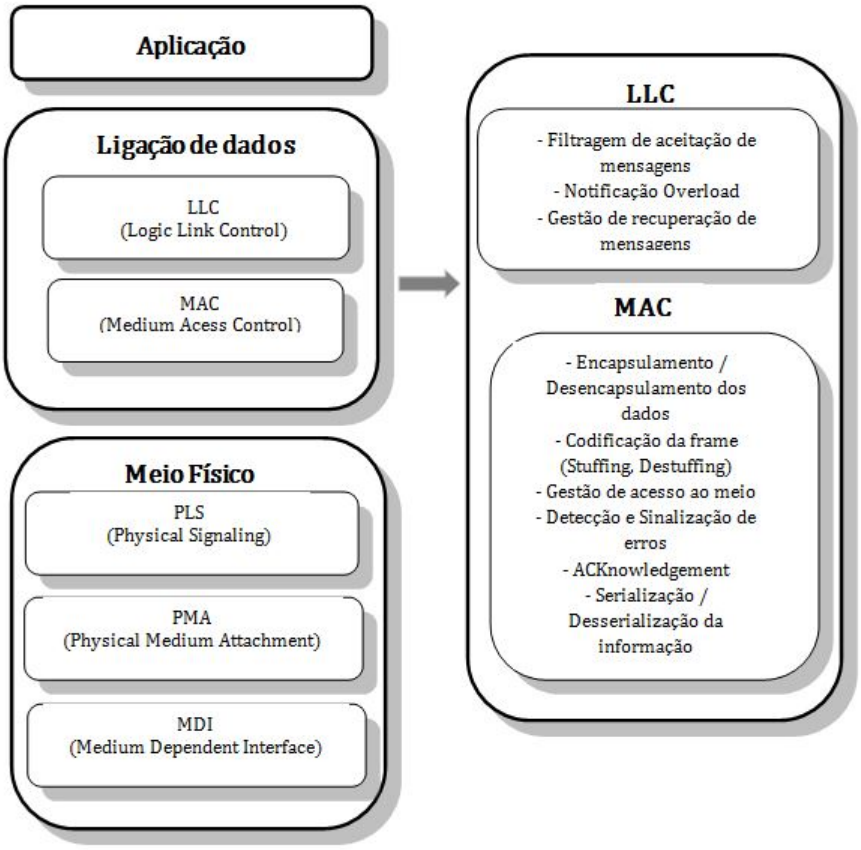


Figura 2.25: CAN 2.0B [10].

A camada física define o nível de transmissão e a representação dos *bits* recebidos através do barramento, além de ser responsável pelo ajuste do tempo de *bit* (*bit timing*)

e sincronização entre os nós que participarão do processo de arbitragem (define qual nó terá acesso ao meio físico para transmitir a sua mensagem) [22]. A camada de ligação de dados divide-se em controlo de ligação lógica LLC (*Logical Link Control*) e controlo de acesso ao meio MAC.

O LLC é responsável pelo controlo da aceitação de mensagens que são recebidas e a notificação de sobrecarga do nó conectado à rede. MAC é considerado o *kernel*[16] do protocolo CAN, que tem como função principal o controlo de acesso ao meio físico, além da detecção/sinalização de erros, reconhecimento de mensagens recebidas e desencapsulamento de mensagens [22]. As *shields* implementadas nos arduinos possuem um controlador CAN com a versão v2.0B o que significa que possuem a versão CAN mais recente e podem ser implementadas com identificadores de 11 ou 29 *bits*.

Processo de arbitragem. O barramento CAN é um protocolo de comunicação série desenvolvido para aplicações na indústria automóvel que recentemente vem sendo utilizado em sistemas de automação industrial. O protocolo CAN é baseado na técnica CSMA/CR (*Carrier Sense Multiple Access/Collision Resolution*), também denominado de CSMA/CD-AMP (*Carrier Sense Multiple Access/Collision Detection and Arbitration on Message Priority*), de acesso ao meio de transmissão [37]. Isso significa que possui um CSMA não destrutivo, e que sempre que ocorrer uma colisão entre duas ou mais mensagens, a de mais alta prioridade terá o acesso ao meio físico assegurado e prosseguirá a transmissão.

Uma das propriedades mais importante do barramento é o esquema de arbitragem binária (*bitwise arbitration*) que fornece uma boa maneira de resolver a colisão de mensagens. Sempre que dois nós começarem a transmitir mensagens ao mesmo tempo, o mecanismo de arbitragem garante que a mensagem de mais alta prioridade será enviada. Isso é conseguido pela definição de dois níveis de barramentos chamados recessivo e dominante [22].

Os níveis exactos de tensão utilizados nos cabos de transmissão são mostrados na figura 2.26. No estado recessivo, tanto CAN-L (CAN *Low*) e CAN-H (CAN *High*) estão em 2.5V (nível neutro). No estado dominante, CAN-L é "puxado"para baixo 1V e CAN-H é "puxado"para cima 1V. Assim, os níveis são 1.5V para CAN-L e 3.5V para CAN-H com

[16] Componente central do sistema operativo da maioria dos computadores; ele serve de ponte entre aplicações e o processamento real de dados feito a nível de *hardware*.

uma diferença de 2V entre eles [1].

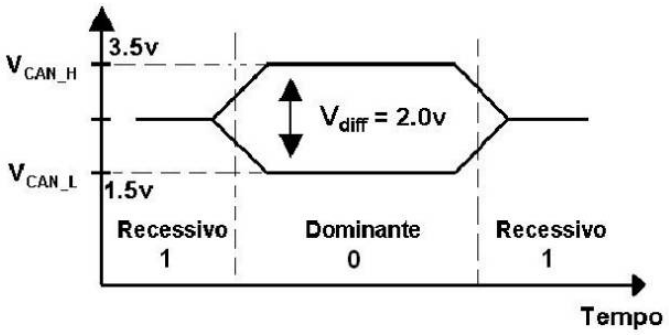


Figura 2.26: Tensão eléctrica da camada física CAN [11].

Qualquer nó no barramento só pode fazer o sinal ir de "1"para "0". Ele não pode forçar um "0"para "1". Ou seja, em repouso, o barramento está em um "1". Portanto, um nó faz um "0", "puxando"o barramento separados por 2 volts, e "1"por não fazer nada, permitindo que o barramento se une à sua diferença ociosa de 0 (zero) V. Os termos dominantes e recessivos referem-se não só aos níveis de tensão/lógica, mas também dá sentido ao esquema de prioridade do CAN. Dominante é "0"e recessivo é "1"[11]. Rever a figura 2.26 para verificar estes dois estados possíveis do barramento CAN.

A figura 2.27 ilustra a arbitragem CAN. Se ambas as opções estão abertas, a diferença de tensão através da lâmpada é zero - este é o estado recessivo. Ou o interruptor pode colocar uma diferença de tensão através da lâmpada fazendo com que ilumine. Este é o estado dominante e um voltímetro mede a diferença de tensão de 2 volts na lâmpada. Neste caso, figura 2.27, o interruptor da mão direita é incapaz de desligar a lâmpada - cada chave só pode ligar a lâmpada : não desligará, se o outro já está ligado. Se ambas as opções estão abertas, a diferença de tensão vai se tornar 0 volts e a lâmpada apaga-se [11].

Um nível dominante sempre sobrepõem um nível recessivo. Assim, enquanto um nó está enviando uma mensagem, ele compara o nível do *bit* transmitido com o nível monitorado no barramento. Se um nó tenta enviar um nível recessivo e detecta um dominante, ele perde a arbitragem e interrompe o processo de transmissão [22].

Por vezes, se numa rede existir um nó com muita baixa prioridade e sempre que tentar enviar uma mensagem "perder"o acesso ao barramento, esta longa latência da mensagem

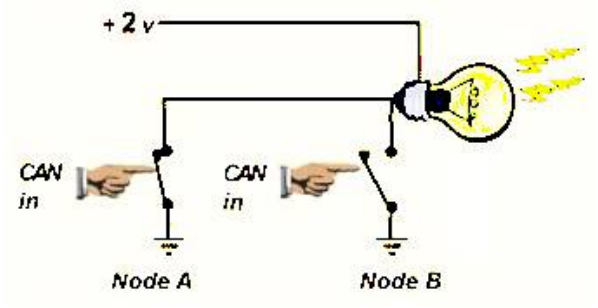


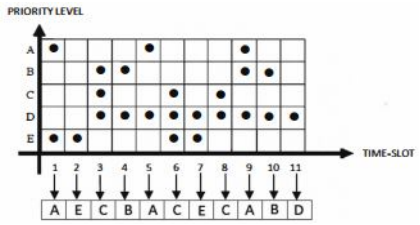
Figura 2.27: Circuito para arbitragem da saída CAN [11].

do nó de baixa prioridade é problemática em situações em que este nó está a tentar enviar uma mensagem crítica supondo que se esteja presente em um sistema de segurança crítica.

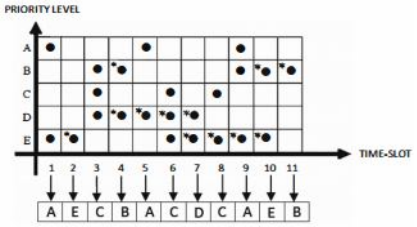
Em [12] é mencionado o *Counter* CAN (C-CAN) para resolução deste problema de performance do tempo e da latência da mensagem do nó de baixa prioridade. Na figura 2.28(a) estão representados os nós e os seus níveis de prioridade. A figura 2.28(b) apresenta a prioridade baseada na arbitragem CAN e como se ver pela figura o nó D o de menor nível de prioridade só consegue ter acesso ao barramento quando nenhum outro nó estiver a enviar uma mensagem.

Node Name	Priority Level	Priority Number
A	Highest	2
C		4
B		6
E		8
D	Lowest	10

(a) Nível de prioridade dos nós.



(b) Prioridade baseada na arbitragem CAN.



(c) Prioridade baseada na arbitragem C-CAN.

Figura 2.28: Modelos de comunicação de dados [12].

A resolução para o problema de longa latência do nó D, o C-CAN figura 2.28(c), subtrai por 1 o nível de prioridade do nó ou dos nós sempre que perder o acesso ao barramento

o que fará com que diminua com o tempo de latência. Para mais informação consultar a referência [12].

2.7 Camada de ligação de dados CAN

Existem quatro tipos de tramas no protocolo CAN: trama de dados, trama de erro, trama remota e trama de sobrecarga. Apenas a trama de dados transporta os dados da mensagem. De acordo com a especificação CAN dois diferentes formatos de tramas são especificados: "formato base"com identificadores de 11 *bits* e o "formato estendido"com identificadores de 29 *bits*.

2.7.1 Trama de dados

Na figura 2.29 está representado o formato de uma trama de dados e da trama remota com identificador de 11 *bits* ("formato base") com os seus respectivos campos de comprimento e a quantidade de *bits*.

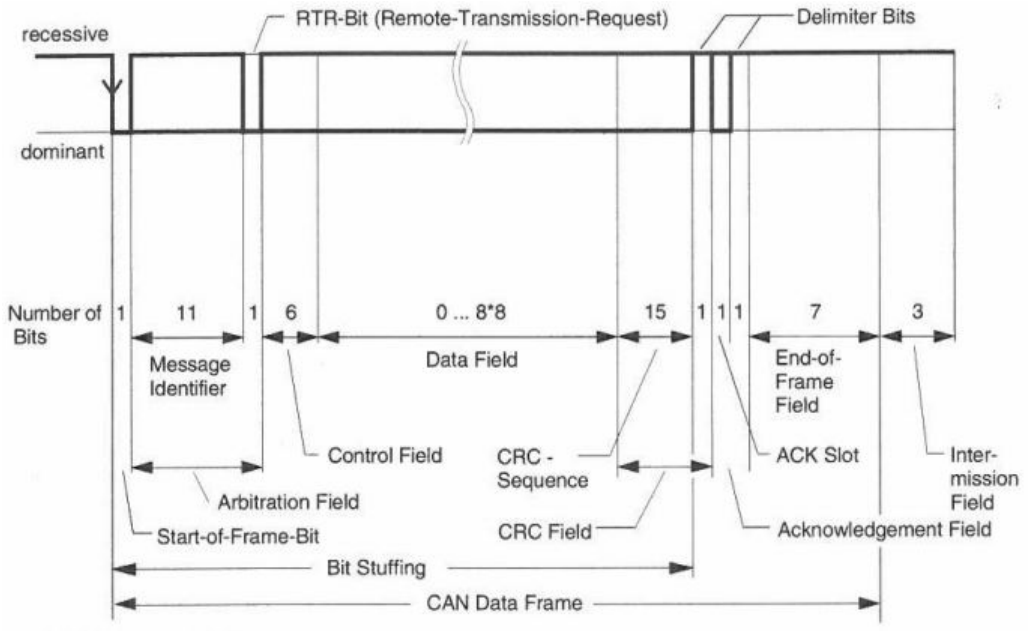


Figura 2.29: Formato da trama de dados e da trama remota ("formato base") [2].

Cada mensagem CAN é dividida em diferentes campos de comprimento específico, estes campos são: início da trama(*Start Of Frame*), campo de arbitragem (*Arbitration Field*), campo de controlo (*Control Field*), campo de dados (*Data Field*), campo de verificação de redundância cíclica (CRC field), campo de confirmação (*Acknowledgement Field*) e fim da

trama (*End Of Frame*). O campo de arbitragem combina, no identificador, a prioridade da mensagem, com o endereço lógico da informação, ou seja para se saber se se trata de uma trama de dados ou uma trama remota.

Início da trama (*SOF*) corresponde a 1 *bit* dominante. Este *bit* marca o início da trama de dados ou trama remoto e é representado por um único *bit* dominante.

Campo de arbitragem (*Arbitration Field*) corresponde a 12 *bits*. Consiste no campo identificador de 11 *bits* e o *bit* RTR (*Pedido de Transmissão Remota*), figura 2.30. Quanto menor fôr o valor númerico do identificador maior é a prioridade. O *bit* RTR é utilizado para distinguir uma trama de dados (RTR *bit* dominante) da trama remota (RTR *bit* recessivo).

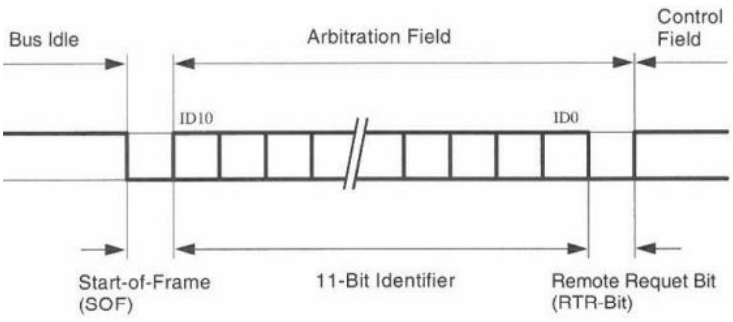


Figura 2.30: Formato do campo de arbitagem [2].

Campo de Controlo (*Control Field*) corresponde a 6 *bits*. O primeiro *bit* no campo de controlo é o identificador de extensão IDE (*Identifier Extension Flag*). Nas tramas de identificadores com 11 *bits* este *bit* é dominante, o que indica que o identificador está concluído. O *bit* seguinte *r0* é reservado.

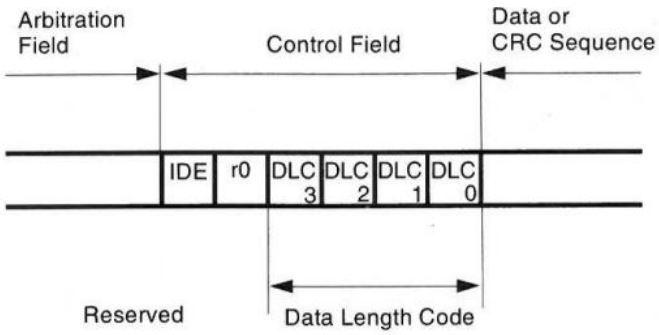


Figura 2.31: Formato do campo de controlo ("formato base") [2].

Os últimos quatro *bits* do campo de controlo, figura 2.31, correspondem ao comprimento dos dados a serem transferidos, o código do tamanho dos dados DLC (*Data Lenght Code*) varia de acordo com a tabela 2.10.

Tabela 2.10: Valores aceitáveis no campo DLC [23].

Bytes	DLC3	DLC2	DLC1	DLC0
0	D	D	D	D
1	D	D	D	R
2	D	D	R	D
3	D	D	R	R
4	D	R	D	D
5	D	R	D	R
6	D	R	R	D
7	D	R	R	R
8	R	D	D	D
D = *Bit* Dominate		R = *Bit* Recessivo		

Campo de dados (*Data field*) corresponde de 0 a 8 *bytes*. Este campo contém os dados a serem transferidos dentro da trama CAN e podem ser entre 0 a 8 *bytes*. O *bit* mais significativo é transferido primeiro, ver figura 2.29.

Campo CRC (*CRC field*) corresponde a 16 *bits*. O campo CRC consiste numa sequência de verificação de 15 *bits* e um *bit* delimitador transmitido recessivamente. Por meio da sequência CRC o receptor verificará se a sequência de *bits* recebido foi corrompido por alguma perturbação (figura 2.32).

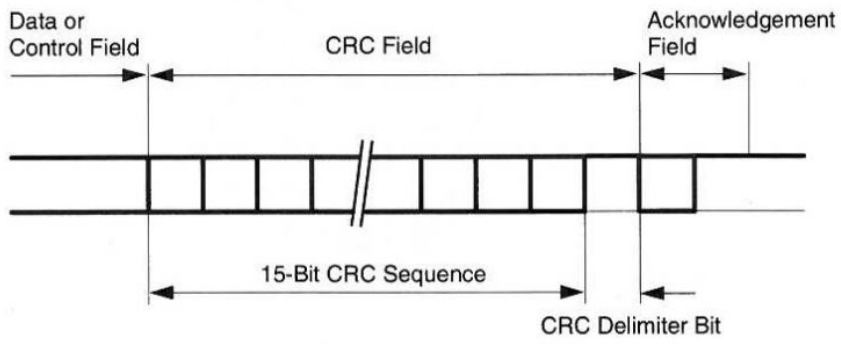


Figura 2.32: Formato do campo CRC [2].

Campo de confirmação (*ACK field*) corresponde a 2 *bits*. Os dois *bits* do campo de confirmação (figura 2.33), consistem nos chamados *bit "ACK Slot"*e *bit "ACK Delimiter"*ou *bit "ACK delimitador"*.

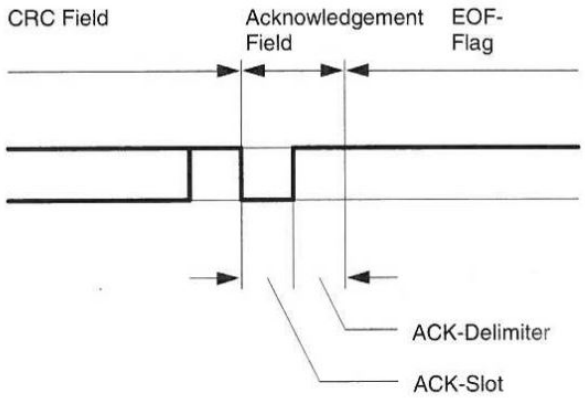


Figura 2.33: Formato do campo do confirmação [2].

No campo de verificação o transmissor envia para o barramento dois *bits* recessivos enquanto que o receptor envia para o transmissor uma trama totalmente recessiva mas com o *bit ACK Slot* em dominante. A figura 2.33 representa a trama do barramento com o formato do campo de confirmação *bit ACK Slot* em dominante e o *bit ACK Delimiter* em recessivo, representando a confirmação de uma mensagem por parte do nó receptor.

Fim da trama (*EOF*) corresponde a 7 *bits* recessivos. A trama de dados é delimitada por uma sequência de 7 *bits* recessivos. Juntos com o *bit* recessivo delimitador ACK resulta numa sequência total de 8 *bits* recessivos no final de uma trama de dados ou trama remota.

O espaço entre as tramas (*Inter Frame Space*) corresponde a 3 *bits* recessivos. Separa a trama actual da trama seguinte. Dentro do nó CAN, este período é também utilizado para transferir uma mensagem correctamente recebida do controlador do protocolo no espaço apropriado do *buffer* de recepção, ou para transferir uma mensagem a partir da memória intermédia de transmissão para o controlador do protocolo. As tramas de dados e tramas remotas são separadas de tramas precedentes independentemente do tipo destas (dados, remota, erro ou sobrecarga).

Tramas com identificador de 29 *bits* diferem da trama com identificador de 11 *bits* no campo de arbitragem e no primeiro *bit* do campo de controlo (figura 2.34).

Campo de arbitragem (*Arbitration field*) corresponde a 32 *bits*. O campo de arbitragem na trama com identificador de 29 *bits* consiste em 3 partes. Começa com 11

bits mais significativos do identificador base, seguido por 2 *bits* recessivos: o *bit* de pedido substituto de requisição remota SRR (*Substitute Remote Request*) e o *bit* IDE. Em seguida surgem os 18 *bits* menos significativos do identificador de extensão, conforme figura 2.34.

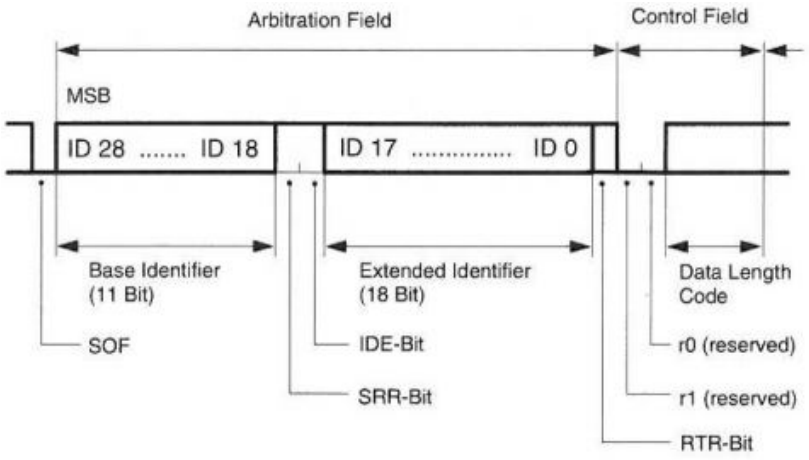


Figura 2.34: Formato da trama estendida [2].

Os valores concatenados do identificador base e do identificador de extensão são os endereços lógicos e prioridade da mensagem. O último *bit* do campo de arbitragem é o *bit* RTR.

Campo de controlo (*Control field*) corresponde a 6 *bits*. Em contraste com as tramas com identificadores de 11 *bits*, o campo de controlo da trama com identificadores de 29 *bits* não é iniciado com a IDE e um *bit* reservado, mas com dois *bits* reservados, *r1* e *r0*. Os últimos quatro *bits* do campo de controlo contêm o DLC e em seguida o campo de dados.

2.7.2 Trama remota

Um nó que seja receptor de determinados dados pode iniciar a transmissão dos mesmos através de pedido ao nó de origem, enviando uma trama remota, ou seja, um pedido de dados. A trama remota é constituida por 6 campos. Os campos constituintes de uma trama remota são idênticos aos de uma trama de dados, com a excepção do valor do *bit* RTR do campo de arbitragem, que agora é recessivo e da inexistência de campo de dados. Os *bits* DLC do campo de controlo da trama remota devem possuir valor igual ao da trama de dados correspondente.

2.7.3 Trama de erro

A trama de erro (2.35) é transmitida por qualquer nó quando é detectado um erro no barramento e é constituida por dois campos distintos. O primeiro campo é dado pela sobreposição de *flags* de erro provenientes de diferentes estações. O segundo campo é o delimitador de erro.

Flag de erro: este campo indica a existência do erro informando se é passivo ou activo. Para erro activo a *flag* de erro conterá *bits* dominantes e para erro passivo conterá *bits* recessivos. Delimitador de erro: tem a função de activar a comunicação no barramento CAN após uma falha. Ele é enviado pelo nó que gerou o erro [22].

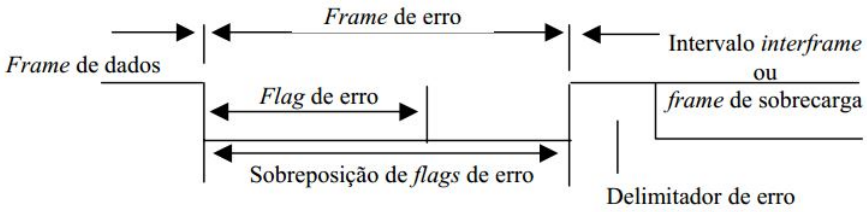


Figura 2.35: Trama de erro [13].

2.7.4 Trama de sobrecarga

A função é similar à da trama de erro (figura 2.36), sinalizando sobrecarga num nó, impossibilitando-o de receber a trama de dados (mensagens). O transmissor precisa de aguardar o próximo processo de arbitragem. A trama de sobrecarga contém dois campos de *bits*: *flag* de sobrecarga e delimitador de sobrecarga. O formato da *flag* de sobrecarga corresponde à da *flag* do erro. O delimitador de sobrecarga tem formato idêntico ao do delimitador de erro.

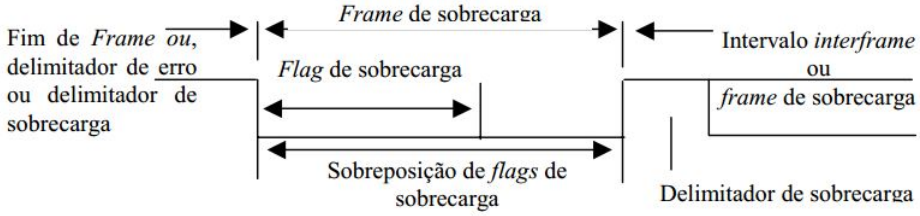


Figura 2.36: Trama de sobrecarga [13].

A trama de sobrecarga é utilizada para provocar um atraso extra entre uma trama de dados, ou remota, e a trama posterior. As tramas de erro e de sobrecarga não são precedidas por um espaço entre as as tramas.

2.7.5 *Bit* de codificação de fluxo

No protocolo CAN os *bits* de uma trama são fisicamente representados de acordo com o código *Non-Return-to-Zero* (NRZ). Isto significa que, durante um intervalo de tempo de um *bit* são gerados *bits* de nível dominante ou recessivo [2].

Uma longa sequência de *bits* da mesma polaridade (no máximo podem ser transmitidos 5 *bits* com a mesma polaridade) causará perdas de sincronização entre controladores CAN. Esta desvantagem da codificação em NRZ pode ser eliminada quando um *bit* adicional de polaridade diferente (*"stuff bit"*) é inserido na sequência de *bits* enviados pelo transmissor. O receptor remove estes *bits* inseridos em conformidade antes de fazer o processmento do sinal. Este processo é conhecido como *bit stuffing*, como observado na figura 2.37, com as seguintes sequências de *bits*:

1. Sequência de *bits* a serem transmitido;
2. Sequência de *bits* após ter ocorrido o *bit stuffing* (S = *Stuff bit*);
3. Sequência de *bits* recebidos pelo receptor após o filtro do *bit stuff*.

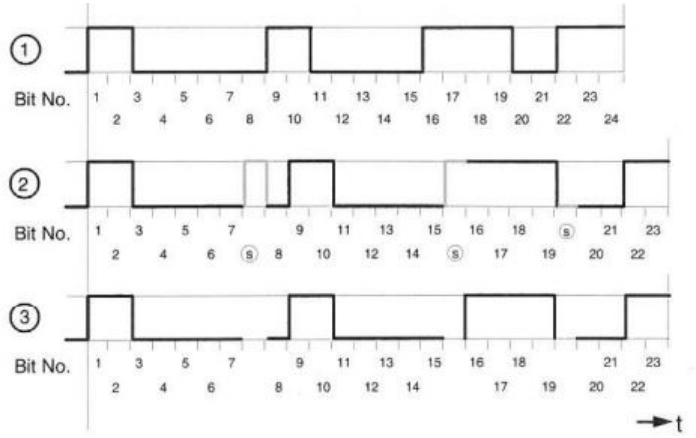


Figura 2.37: Processo de *bit stuffing* [2].

Em trama de dados e remota, o campo SOF, o campo de arbitragem, o campo de controlo, o campo de dados e a sequência CRC são codificados com o metódo *bit stuffing*.

2.7.6 Temporização de um *bit* CAN

O protocolo CAN difere de outros protocolos pela utilização de uma transmissão síncrona de dados em vez de uma transmissão assíncrona[17] Enquanto a sincronização

[17]Protocolo Modbus utiliza uma transmissão assíncrona.

do *bit* numa transmissão assíncrona é facilmente feita durante a recepção do *start-bit* dum caracter, em uma transmissão síncrona não existe um *start-bit* disponível para iniciar a trama. Isto normalmente não é suficiente para manter a amostragem do receptor suficientemente sincronizada com o transmissor. Para permitir que o receptor tenha uma amostragem correcta do fluxo de *bits* recebidos é necessário uma ressincronização contínua do receptor.

Cada nó em uma rede CAN é cronometrado individualmente por um gerador de *clock*. Os segmentos de tempo de *bit* (*bit time*), t_{bit}, são individualmente ajustados em cada nó, em que o tempo de *bit* é o inverso do *nominal bit rate* (*NBR*), como se indica na equação (2.1).

$$NBR = f_{bit} = \frac{1}{t_{bit}} \tag{2.1}$$

Os parâmetros do tempo de *bit* estão escritos nos registros de configuração lógica da temporização *bit BTL* (*bit timing logic*). A taxa de transmissão *prescaler BRP* (*Baud rate prescaler*) determina a duração do *time quantum* t_q, que é a unidade básica do tempo de *bit*, enquanto que os segmentos de temporização determinam o número de *time quantum* no tempo de *bit*. As frequências de relógio dos osciladores nos nós CAN não são absolutamente estáveis, devido à temperatura ou à variação de tensão e ao envelhecimento dos componentes. Enquanto os desvios permanecem dentro de uma certa gama de tolerância do oscilador, os nós são capazes de compensar as diferenças de sincronização numa trama CAN. De acordo com a especficação CAN descrita na figura 2.38 e pela equação (2.2), o tempo do *bit* (*bit time*), t_{bit}, é dividido em quatro segmentos:

1. Segmento de sincronização (*Synchronization Segment*) - tem a duração de 1 t_q e é utilizado para a sincronização dos nós;

2. Segmento do tempo de propagação (*Propagation Time Segment*) - serve para fornecer tempo à propagação do sinal no meio físico da rede CAN e pode ir de 1 até 8 t_q;

3. Segmento 1 *Phase Buffer* (*Phase Buffer Segment 1*) - este segmento, juntamente com o segmento 2 servem para compensar erros de fase e cercam o tempo de amostragem, O ponto de amostragem é o ponto de tempo em que o nível do barramento é lido e interpretado como um valor do respectivo *bit*. A sua localização é no final de segmento 1. Por ressincronização estes segmentos podem ser alargados ou

encurtados. O segmento 1 pode ir de 1 a 8 t_q;

4. Segmento 2 *Phase Buffer* (*Phase Buffer Segment 2*) - é o valor máximo do segmento 1 e é denominado "tempo de processamento da informação".

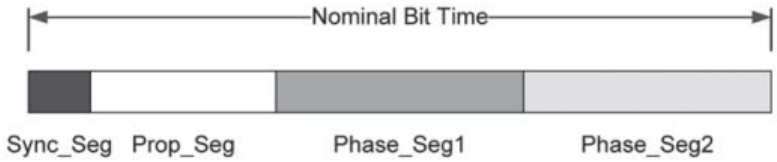


Figura 2.38: Tempo de *bit* [2].

$$t_{bit} = t_{Sync\_Seg} + t_{Prop\_Seg} + t_{Phase\_Seg1} + t_{Phase\_Seg2} \tag{2.2}$$

O número total de *time quantum* t_q, em um tempo de *bit* tem de ser maior de 8 e menor de 25. Para mais informação da temporização de um *bit* CAN deve-se consultar as referências [2], [10] e [37].

2.7.7 Verificação e sinalização de erros

A aplicação em veículos inicialmente prevista pelo protocolo CAN exigia elevado desempenho no que se refere à segurança do resultado de transmissão de dados. Para atender a essa especificação, vários mecanismos são fornecidos pelo protocolo para detectar erros. Estes mecanismos são os seguintes: verificação do *bit* e da trama, CRC, ACK e *bit stuffing*, sendo descritos de seguida [22].

Verificação do *bit*: o emissor tem a capacidade para detectar erros de *bit* baseado na monitorização dos sinais do barramento. Assim cada nó que transmite também monitora o barramento, detectando, caso existam, diferenças entre o valor do *bit* transmitido e o valor do *bit* monitorizado.

Verificação da trama: este tipo de erro é detectado quando um campo da trama contiver um ou mais *bits* incongruentes.

Bit Stuffing esta técnica é utilizada para detecção de erros nas tramas transmitidas. Após cada conjunto de 5 *bits* iguais é inserido um *bit stuffing*. Esta técnica é aplicada

somente nas tramas de dados e nas tramas remotas. Os campos ACK e EOF estão sujeitos a esta técnica. As tramas de sobrecarga e erro não são sujeitas a esta técnica.

CRC: antes da transmissão de um trama o seu CRC é calculado e o seu resultado inserido no campo CRC. Na recepção dessa trama, o cálculo é refeito e comparado com o resultado no campo CRC da trama recebida. No caso de erro, uma trama é transmitida imediatamente. O cálculo de CRC é utilizado nas tramas de dados e é calculado a partir do campo SOF até o último campo, EOF.

ACK: um transmissor ao enviar uma mensagem na rede CAN, envia o campo ACK com *bits* recessivos. O receptor, ao receber essas mensagens, retorna esses *bits* como dominantes. Quando o transmissor recebe o ACK em dominante significa que a mensagem foi recebida com sucesso.

O protocolo CAN define os contadores de erro, TXCnt (contador de erro de transmissão) e RXCnt (contador de erro de recepção). Esses contadores definem os estados de erro de um determinado nó, podendo ser erro passivo ou activo, como se observa na figura 2.39.

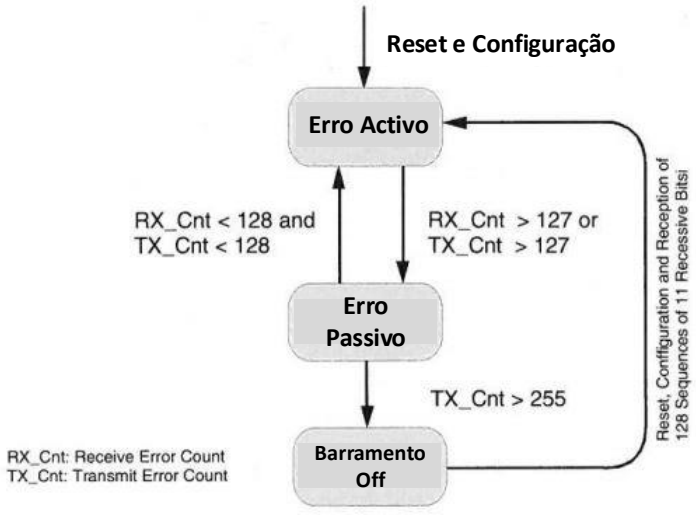


Figura 2.39: Diagrama de estado de erro de um nó CAN [2].

Erro passivo: representa o estado normal de um nó, podendo transmitir ou recebeer

mensagens pela rede. Na detecção de erro em uma recepção, a *flag* de erro passivo é enviada.

Erro activo: indica que o nó está com erros frequentes. Quando os contadores TXCnt e RXCnt ultrapassam 127 erros, o nó passa de estado passivo para o estado activo. Nesta situação, *flags* de erro activo são enviadas. Isso garante maior eficiência do barramento CAN, impedindo que nós com erros frequentes possam utilizar o barramento.

Barramento Off: um nó que atinge 255 erros será desconectado do barramento e somente será iniciado por um *reset*.

2.7.8 Filtragem

O protocolo CAN é implementado utilizando controlador que usualmente também comunica com qualquer um microcontrolador. Assim, a maioria dos controladores de protocolo oferece um serviço de filtragem de mensagens que faz com que somente as mensagens com o padrão de identificação pré-programado sejam armazenadas e sinalizadas ao microcontrolador.

Isso possibilita uma economia de tempo de leitura e processamento das mensagens recebidas, libertando o microprocessador para tarefas mais importantes. Essa operação normalmente envolve a configuração de duas máscaras para o identificador das mensagens de forma a seleccionar as mensagens ou grupo de mensagens desejadas, descartando as não desejadas.

2.8 Camada física do protocolo CAN

Um nó é geralmente conectado a um barramento constituido por dois fios terminados, em ambas as pontas por dois resistores de um valor recomendado de 120 *Ohms* para evitar sinal de reflexão. O controlador CAN pode estar directamente conectado a qualquer microcontrolador. A ligação do nó com o meio físico normalmente faz-se através de um transceptor, o qual comunica serialmente com o controlador através do pino de saída Tx e do pino de entrada Rx.

A função do transceptor é transformar os *bits* que entram e saem do controlador em tensão diferencial a ser aplicada ao barramento. O modo de transmissão diferencial é

utilizado para proporcionar imunidade a interferências electromagnéticas ao barramento. O nível recessivo corresponde a uma diferença de tensão menor do que 0,5V entre CAN-H e CAN-L, com a tensão em CAN-H sendo normalmente maior do que CAN-L. Já o nível dominante é detectado quando a diferença de tensão fôr de no mínimo 0,9V, sendo a tensão nominal nesse estado de 3,5V para CAN-H e 1,5 para CAN-L.

A arquitectura implementada para a camada física do protocolo CAN nesta dissertação é igual à representada na figura 2.40, onde o produtor CAN - supervisor será um arduino com um microcontrolador ATmega328 e o consumidor CAN - controlador um microcontrolador ATmega2560, ambos com controladores CAN MCP2515 e transceptores CAN MCP2551. As *shields* CAN utilizadas já possuem resistor terminal e o barramento implementado é constituido por um cabo de par trançado. Com maiores detalhes no capítulo 3 serão apresentados os nós a implementar.

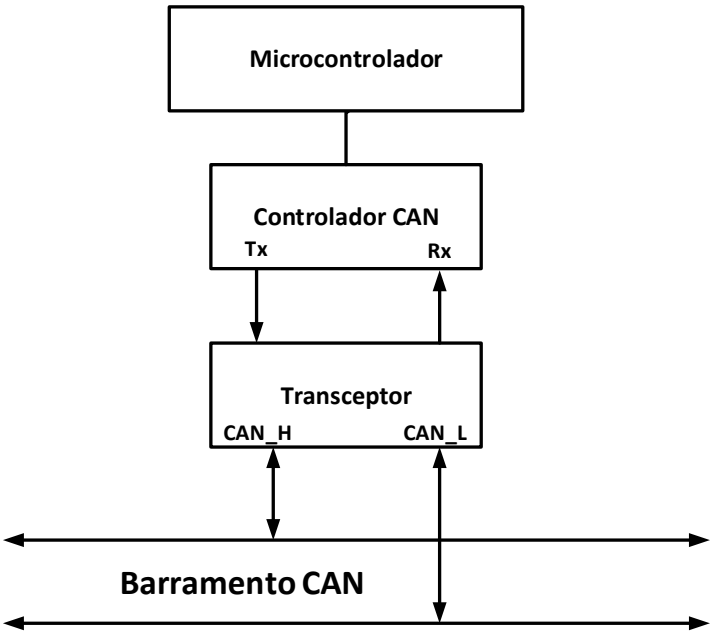


Figura 2.40: Barramento de conexao de um controlador CAN e um CAN transceptor [2].

O barramento de dois fios permite a transmissão do sinal diferencial e é tolerante a erros dentro de certos limites. A interferência de campos induzidos electromagneticamente pode ser compensada por um par de fios enterlaçados, aumentando assim a imunidade à interferência.

A realização de uma rede CAN é suportada pela recomendação DS-102 da CiA para

acesso ao meio e ao barramento. Esta proposta é baseada na ISO 11898-1 e 11898-2 e especifica: taxas padronizadas de 10 kbps a 1 *Mbps*, recomendações dos fios do barramento e atribuição dos pinos e conectores. A recomendação DS-102 especifica um cabo de par enterlaçados terminado em ambas as extremidades com a impedância da linha (figura 2.41) [38].

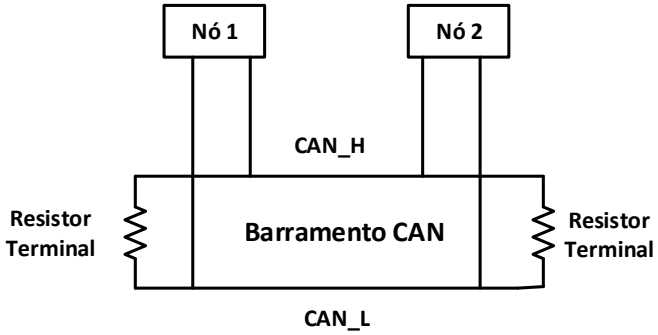


Figura 2.41: Forma recomendada de conexão ao barramento [2].

Uma aplicação do protocolo CAN é descrita em [14], sendo o seu diagrama de blocos elucidado na figura 2.42. O objectivo do desenvolvimento deste sistema consiste em monitorizar vários parametros de um veículo tais como temperatura, nível de CO (monóxido de carbono), tensão entre os terminais da bateria e o LDR (fotoresistência) através do protocolo CAN utilizando como camada de aplicação um programa desenvolvido em MPLab IDE.

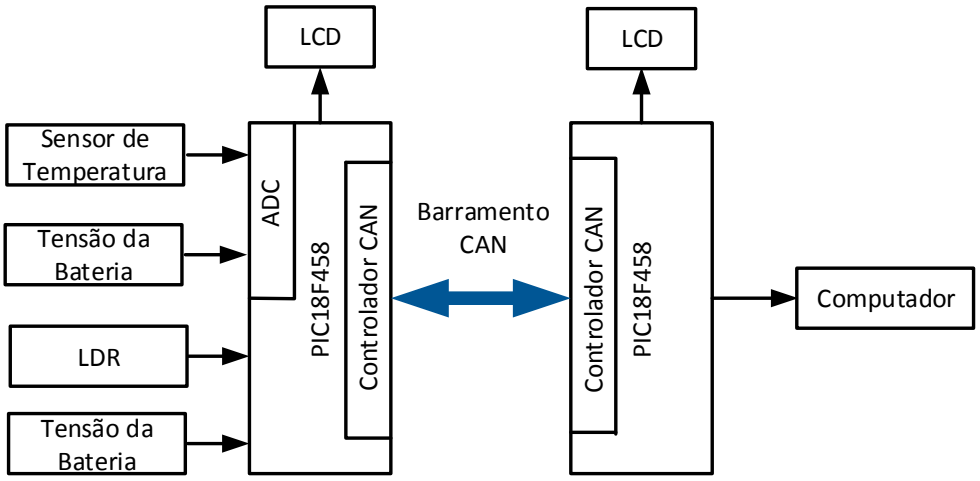


Figura 2.42: Arquitectura baseada na monitorização de um veículo usando CAN [14].

2.9 Camada de aplicação CAN

O CAN, inicialmente, consistia apenas nas camadas de meio físico e de ligação de dados. Após a sua imposição com sucesso no mercado e com o seu desenvolvimento devido à estabilidade que proporcionou nas redes industriais, foram desenvolvidos pacotes de *software* por várias companhias com a finalidade de conseguir implementar serviços na camada de aplicação. São aqui mencionados alguns protocolos de alto nível baseados no barramento CAN [39]:

- SAE J1939 utilizado em aplicações de redes em automóveis, com especial atenção para os autocarros e camiões, desenvolvido por SAE International (*Society of Automotive Engineers International*);
- CANKingdom utilizado em robots, máquinas, etc. Desenvolvido por Kvaser (Kvaser AB);
- Devicenet utilizado em automação industrial, desenvolvido inicialmente pela Allen-Bradley, agora Rockwell Automation, Inc.;
- CANOpen utilizado para diversos tipos de redes industriais (indústria automóvel, automação na agricultura, electrónica marítima, equipamento médico, etc), desenvolvido pela CiA (CAN-in Automation);
- CAL utilizado para controlo e monitorização de dispositivos industriais (ex. PLC), desenvolvido por CiA (CAN-in Automation);
- SDS utilizado em sensores inteligentes e actuadores ligados directamente a uma rede, desenvolvido pela Micro Switch Honeywell;
- TT-CAN utilizado essencialmente na indústria automóvel (controlo de sistemas do motor, chassis e transmissão), desenvolvido pela CiA (CAN - in Automation);

Uma camada de aplicação baseada no barramento CAN pode ser observada em [15]. Foi criada uma camada de aplicação para um sistema de controlo *Beam* que pode controlar antenas *beam* para uma orientação espacial desejada. Devido à capacidade de um rápido varrimento da antena *beam*, o sistema de controlo *beam*, ilustrado na figura 2.43 precisa de uma rede fiável e de alto desempenho em tempo real. O estudo deste sistema baseou-se

na versão CAN 2.0B padrão e como referência nas especificaçõs da camada de aplicação CANopen[18].

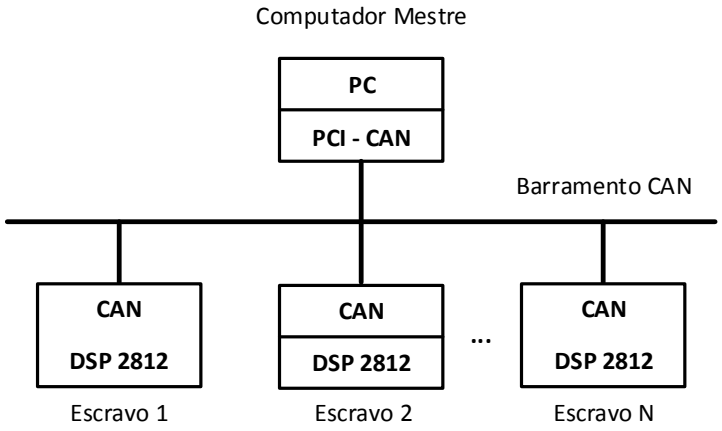


Figura 2.43: Arquitectura da camada de aplicação Modbus-CAN [15].

O sistema de controlo *beam* consiste em um computador industrial mestre e um número variavel de escravos. O computador mestre é conectado à rede de controlo CAN via PCI para um adaptador CAN. O escravo é um sistema DSP2812 que tem uma interface para o barramento CAN. Um escravo controla um determinado número de antenas. O mestre recebe os comandos de controlo ou parametros de controlo via uma interface HMI.

O protocolo CAN e o protocolo Modbus são dois tipos de barramentos muito populares nos sistemas de controlo. Vários estudos têm sido feitos para uma interface de conversão entre esses dois protocolos. Em [16] apresenta-se uma arquitectura de interface CAN - Modbus, elucidada na figura 2.44.

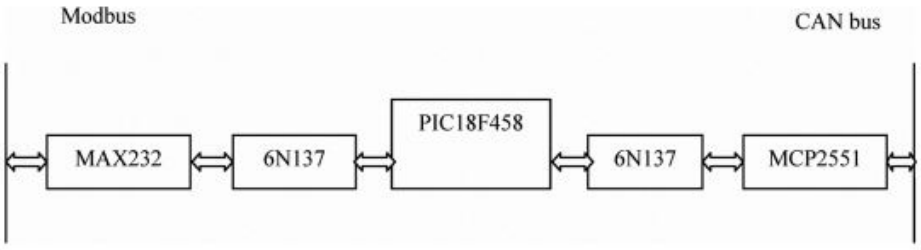


Figura 2.44: Arquitectura da interface de conversão Modbus/CAN [16].

[18]A camada de aplicação CANopen é baseada no controlo de acesso ao meio mestre - escravo.

Nesta arquitectura o CAN foi definido como mestre e o Modbus como escravo. O mestre envia os dados para a interface de conversão, após a recepção dos dados (comando, endereço, etc) a partir da interface. Em seguida a interface encapsula os dados em formato do protocolo Modbus para enviar para o barramento Modbus; as informações de resposta são enviadas de volta para a interface de conversão. Depois de a interface receber a mensagem, analisa os dados e depois converte para o formato do protocolo CAN e envia para o mestre. Quando os dados enviados pelo protocolo Modbus for superior a 8 *bytes* este será enviado muitas vezes até completar os dados enviados.

Em [17] é apresentada uma filosofia diferente em relação à anterior, criando uma camada de aplicação do protocolo CAN a qual se chama de Modbus-CAN. Segundo o autor os resultados obtidos utilizando esta interface atingem um melhor desempenho em comparação com o protocolo Modbus TCP/IP (2.45). Para maiores informações relacionada com as interfaces Modbus - CAN consultar a referência [40].

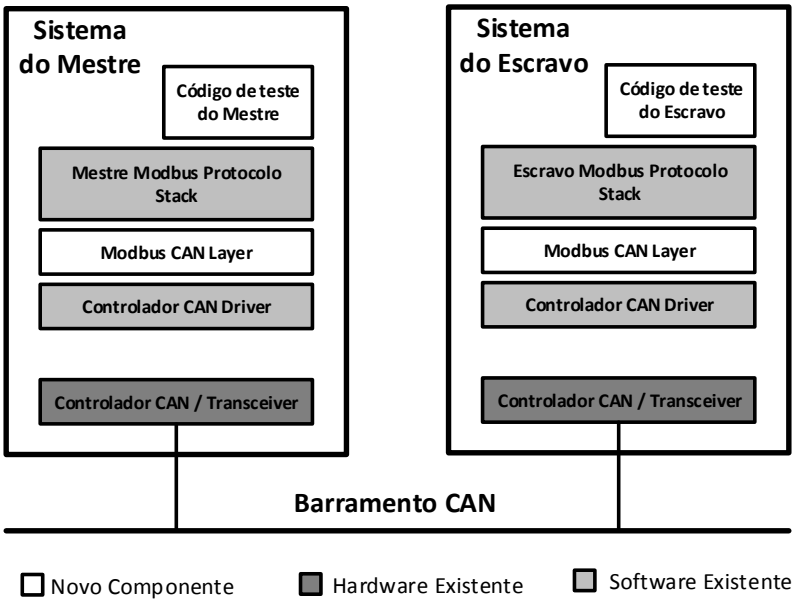


Figura 2.45: Arquitectura da camada de aplicação Modbus-CAN [17].

O estudo do protocolo Modbus foi direccionado para uma transmissão em linha série RS-485 no modo RTU e uma camada física com dois fios em *half-duplex*. Na camada de aplicação viu-se que o protocolo Modbus é baseado no controlo de acesso ao meio mestre e escravo. O tamamho máximo do campo de dados em Modbus RTU é de 252 *bytes*.

O protocolo CAN será implementado com placas Arduino na versão CAN v2.0B, com um processo de arbitragem baseado em CAN e um controlo de acesso ao meio baseado na técnica CSMA/CR. Como camada de aplicação será um programa criado ambiente de desenvolvimento integrad do arduino; com identificador de 11 *bits*. No protocolo CAN o tamanho máximo do campo de dados é de 8 *bytes*. A *shield* CAN da placa Arduino actualmente não suporta uma biblioteca para uma trama remota.

Capítulo 3

Arquitecturas, Tecnologias e Implementação

Neste capítulo serão apresentadas as arquitecturas e as tecnologias utilizadas na implementação dos protocolos. Na secção 3.1 é apresentada a arquitectura de implementação do protocolo Modbus e de seguida é feito a análise dos blocos que constituem essa mesma arquitectura. Nas secções seguintes são analisadas as funções Modbus no PLC utilizado para a implementação; são apresentados alguns exemplos de leitura e escrita de *words* utilizando a programação padrão e a programação com funções de definições macro da *Schneider Electric* com o protocolo Modbus terminando com a secção de implementação da arquitectura.

A secção 3.2 é reservada ao protocolo CAN que segue a mesma filosofia da secção do protocolo Modbus, onde se apresenta a arquitectura e a análise dos principais blocos que a constituem. Nas secções seguintes são analisadas as funções CAN na *shield* do Arduino, e, por último, é feita a implementação do protocolo CAN no controlo do processo PCT9.

3.1 Protocolo Modbus

Nesta secção será apresentado o trabalho de implementação realizado com o protocolo Modbus. A implementação foi feita com PLCs da Schneider Electric e com o *software* Twidosuite. Os dois PLCs (mestre e escravo, ou cliente e servidor Modbus) são ligados sobre linha série com o padrão RS-485 em uma configuração de 2 fios, usando portas mini-Din de 8 pinos e programados na linguagem lista de instruções IL (*Instruction List*).

O PLC servidor Modbus funcionará como um controlador PID de um processo e estará directamente ligado ao processo através de entradas digitais, saídas de relé, entradas e saídas analógicas representando a ligação do nível 1 ao nível 2 da pirâmide da automação. O PLC supervisor, ou seja, cliente Modbus (nível 3 da pirâmide) definirá a referência para o controlador e de retorno monitoriza as variáveis (saída e acção de controlo). Devido à ausência de uma HMI no PLC supervisor, recorreu-se ao padrão OPC®, juntamente com o *software* Scilab® para supervisionar o sistema. O PLC supervisor e o Scilab® serão ligados através do padrão OPC®, permitindo a criação de uma interface HMI que converterá as solicitações genéricas-OPC de leitura e escrita em solicitações específicas do PLC e vice-versa.

O OPC® é um padrão de comunicação desenvolvido para as necessidades da indústria de automação, que necessitava unificar a comunicação entre os diversos dispositivos, reduzindo os custos de integração e de desenvolvimento de *software* para as áreas de automação e controlo. Foi um desenvolvimento conjunto de diversos fornecedores de automação industrial para facilitar a comunicação de dados entre dispositivos de distintos fabricantes. Utiliza um protocolo universal de comunicação entre clientes e servidores [41]. Para maiores informações consultar a página: `https://opcfoundation.org/`.

O padrão OPC® utiliza uma comunicação cliente - servidor. Nesta dissertação o servidor OPC® será criado no *software* da MatrikonOPC e o cliente OPC® será realizado um programa desenvolvido em Scilab®.

3.1.1 Arquitectura

A arquitectura proposta para a implementação do protocolo Modbus está representada na figura 3.1. Basicamente, encontra-se dividida em três partes: cliente Modbus - supervisor ou mestre, servidor Modbus - controlador ou escravo e o processo a controlar.

O supervisor e o controlador são constituidos por um PLC[1] e um módulo de entradas e saídas analógicas[2]. O PLC cliente Modbus é monitorizado pelo Scilab®, a conexão entre o PLC cliente e a interface HMI do Scilab® é feita com o padrão OPC®.

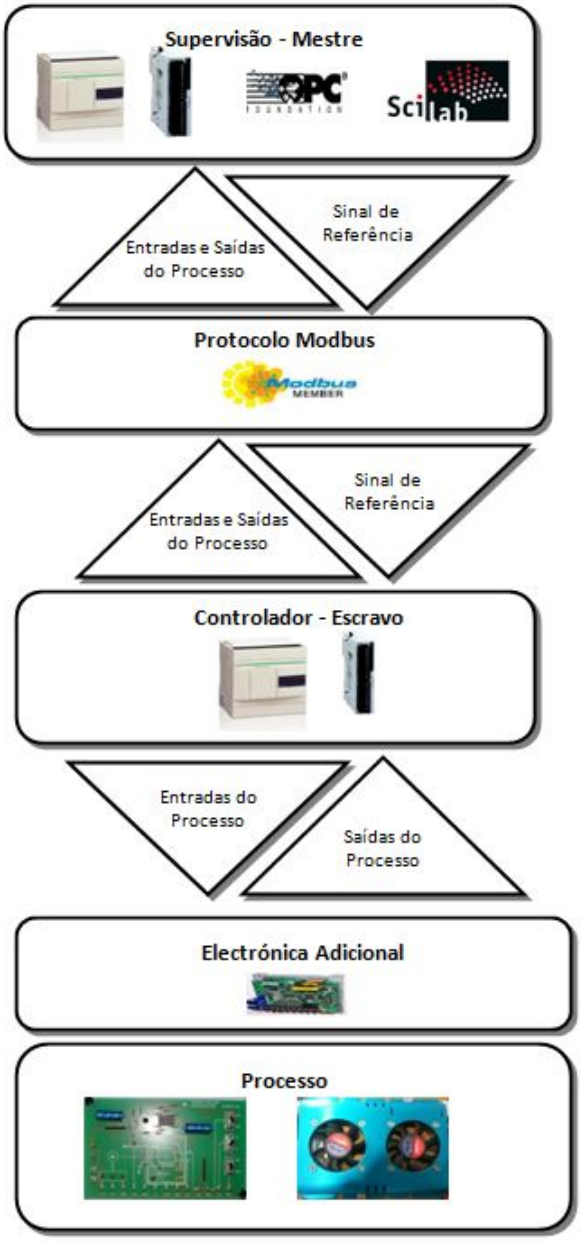


Figura 3.1: Arquitectura do sistema desenvolvido com o protocolo Modbus.

O padrão OPC® utiliza uma comunicação cliente - servidor, no entanto, na interface HMI do Scilab® (OPC cliente) será definida a referência dos ventiladores considerando uma certa gama de temperatura [30 ºC ; 60 ºC] para o simulador de um forno eléctrico,

[1]PLC Twido TWDLCAA24DRF da Schneider Electric
[2]Twido TWDAMM6HT da Schneider Electric

sendo este dado, sinal de referência do processo, enviado ao OPC servidor (MatrikonOPC). Por sua vez, o OPC servidor enviará a referência do processo ao PLC cliente Modbus ou supervisor, tudo isto utilizando a comunicação com o padrão OPC®.

O PLC cliente Modbus (supervisor ou mestre) enviará o sinal de referência ao PLC servidor Modbus (controlador ou escravo), que por sua vez fará o controlo do processo com um controlador PID (*Proporcional, Integral e Derivativo*). A troca de dados entre o PLC cliente Modbus e o PLC servidor Modbus é feita através do protocolo Modbus. No sentido inverso são enviados os sinais do processo (saída dos ventiladores, acção de controlo dos ventiladores e a temperatura do simulador do forno).

A placa electrónica adicional faz parte do processo e é constituida por amplificadores operacionais, transistores, resistores e condensadores.

A figura 3.2 ilustra o diagrama de blocos do controlador ligado ao processo térmico (simulador de um forno eléctrico) e ao processo dos ventiladores. O sinal de referência, acção de controlo e saída dos ventiladores serão as variáveis a apresentar na interface do Scilab®.

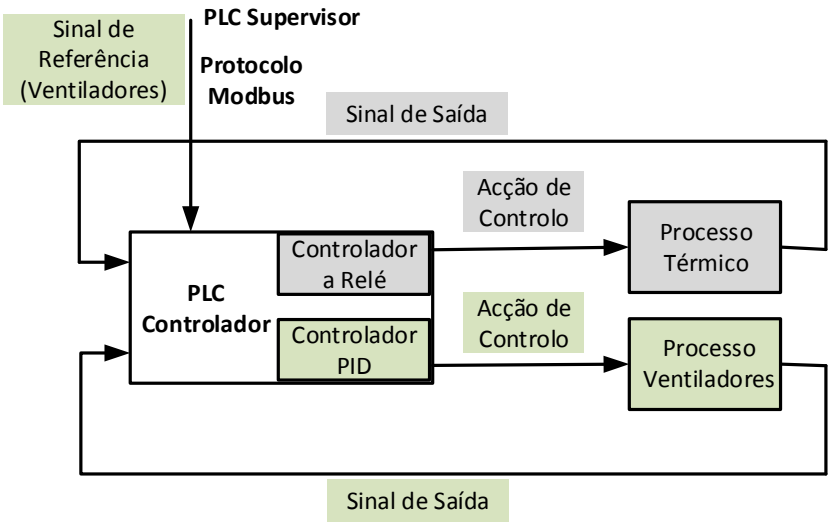
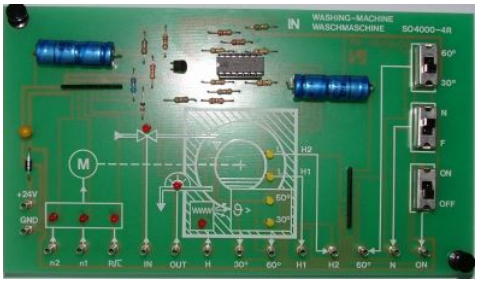


Figura 3.2: Arquitectura de controlo multivariável.

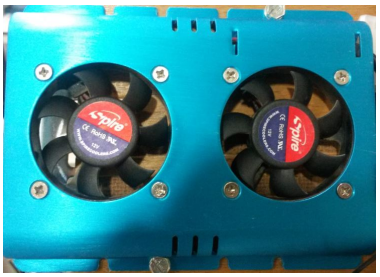
3.1.2 Análise do Processo

O processo é constituido por um simulador de um forno eléctrico e um conjunto de dois ventiladores, tal como se pode observar na figura 3.3. Admitindo que a temperatura

ambiente é inferior a 30 ºC, o forno aquece desde a temperatura ambiente até uma temperatura que se situa na gama [30 ºC; 60 ºC]. Foi implementado um controlador liga - desliga com histerese para o controlo de temperatura do simulador do forno.



(a) Forno eléctrico.



(b) Ventiladores.

Figura 3.3: Processo com simulador de temperatura e ventiladores.

Está acessível a temperatura do forno e a velocidade de rotação dos ventiladores mas devido à ausência de sensores considerou-se como sinal da temperatura e velocidade de rotação, a tensão entre os terminais do simulador e do par de ventiladores. O sensor de temperatura do forno dá-nos valor na gama de [3 V; 8 V] que corresponde às temperaturas de 30 e 60 ºC.

O objectivo neste processo é controlar a velocidade de rotação dos ventiladores definindo uma referência considerando a temperatura do forno na gama [30 ºC; 60 ºC].

Fez-se uma aquisição dos dados nomeadamente à entrada e saída dos ventiladores com ajuda do Matlab® e obteve-se uma função transferência de primeira ordem com ganho estático $G(s = 0) = 0.94$ e constante de tempo 0.17 s. Tendo como base esta função transferência (equação 3.1), projectou-se um controlador PI.

$$G(s) = \frac{K}{\tau s + 1} = \frac{0.94}{0.17s + 1} \tag{3.1}$$

3.1.3 Análise do Servidor Modbus - Controlador do Processo

Esta é a subsecção destinada à análise do controlador, ou servidor Modbus. O controlador do processo é constituido por um PLC e um módulo de entradas e sáidas analógicas, vide figura 3.4. O PLC já tem um módulo de controlador PID embutido, sendo apenas necessário projectar o controlador de modo a definir os ganhos (K_p, K_i e K_d).

A implementação do controlador foi baseada nas análises de Visioli e Kuo (*Pratical PID - Advances in Industrial Control* e *Automatic Control Systems*): "Embora as novas

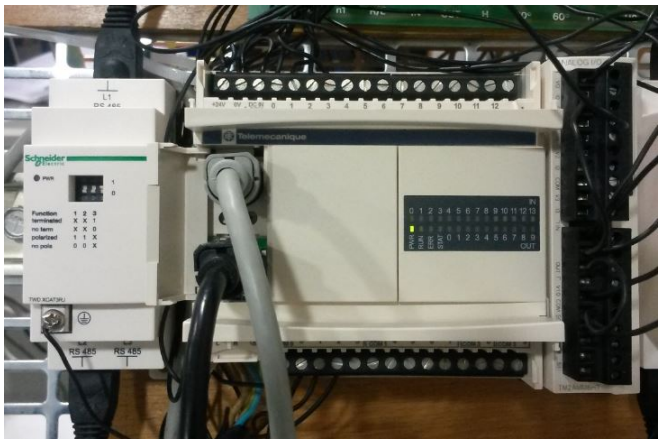


Figura 3.4: Servidor Modbus - Controlador - Escravo.

e eficazes teorias e metodologias de projeto que estão sendo continuamente desenvolvidas no campo de controlo automático, os controladores PID ainda são de longe os controladores mais amplamente adoptados na indústria. Na verdade, embora sejam relativamente simples de utilizar, são capazes de proporcionar um desempenho satisfatório em muitas tarefas de controlo de processo [42] e [43].

Com esta análise, chegou-se à conclusão da escolha da utilização de um controlador PID para o controlo automático dos ventiladores.

O controlador PID tem uma estrutura com 3 componentes: proporcional, integral e derivativa. A versão mais clássica da equação de um controlador PID está descrita na equação 3.2, e na figura 3.5 observa-se a sua arquitectura.

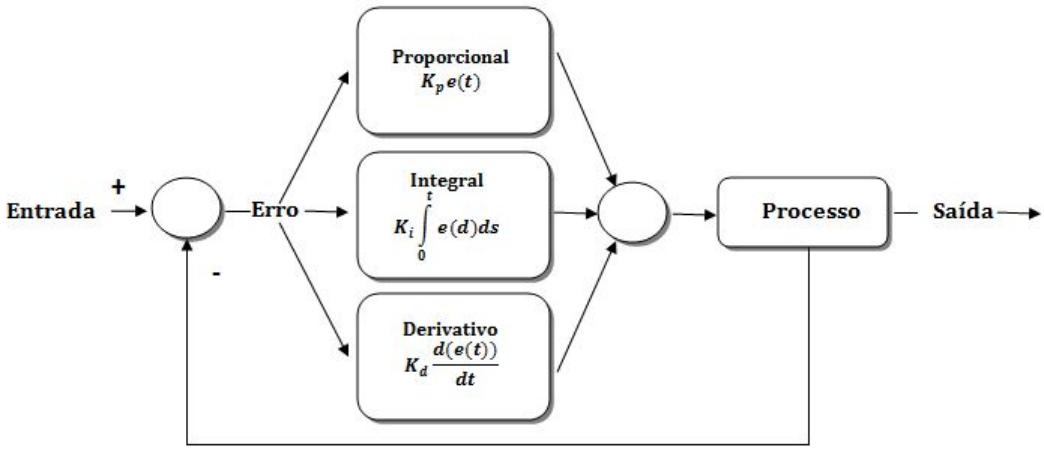


Figura 3.5: Arquitectura clássica de um controlador PID.

$$u(t) = K_p\Big[e(t) + \frac{1}{T_i}\int_0^t e(t)dt + T_d\frac{d}{dt}e(t)\Big] \tag{3.2}$$

Em (3.2), u é a variável de controlo e e é o erro ($e(t) = r(t) - y(t)$), ou seja, a diferença entre a referência r e o valor medido y. A variável de controlo é assim a soma dos três termos: termo **P** (proporcional ao erro), termo **I** (proporcional a integral do erro) e o termo **D** (proporcional à derivada do erro). Os parâmetros do controlador são K_p ganho proporcional, T_i tempo integral e T_d tempo derivativo [18].

A acção proporcional é proporcional ao erro de controlo, de acordo com a equação (3.3) onde K_p é o ganho proporcional.

$$u(t) = K_p \cdot e(t) \tag{3.3}$$

A acção integral é proporcional ao integral do erro, isto é:

$$u(t) = K_i\int_0^t e(t)dt \tag{3.4}$$

$$K_i = \frac{K_p}{T_i} \tag{3.5}$$

onde K_i é o ganho integral. A acção integral está relacionada com os valores anteriores do erro de controlo, ou seja a acção integral permite proporcionar o seguimento de referência e a rejeição de perturbações.

Enquanto que a acção proporcional é depende do valor actual do erro e a acção integral é baseada nos valores passados do erro, a acção derivativa se baseia-se nos valores futuros do erro e pode ser expressa por:

$$u(t) = K_d\frac{d}{dt}e(t) \tag{3.6}$$

$$K_d = K_p \cdot T_d \tag{3.7}$$

onde K_d é o ganho derivativo. A acção derivativa pode proporcionar uma resposta mais rápida às variações do sistema por estimar o valor futuro do erro. No entanto, pode tornar a dinâmica, muitas vezes, agressiva às variações rápidas proporcionando oscilações e até instabilidade no sistema, provocando resultados contrários aos desejados [18].

Após o estudo do controlador PID houve a necessidade de se seleccionar uma arquitectura de controlador PID, optou-se pela versão PI. A acção derivativa muitas vezes não é utilizada. Uma observação interessante é que muitos controladores industriais só têm acção PI e que em outros, a ação derivativa pode ser (e frequentemente é) desligada em muitas malhas de controlo [18].

Após estes estudos o controlador foi projectado com ajuda do Matlab® utilizando o metódo de cancelamento de zeros e pólos, o diagrama de localização das raizes (*rootlocus*) recorrendo as referências [44], [45], [46] e [47], obtendo-se os seguintes resultados para os ganhos do controlador PI:

A equação 3.8 representa o controlador e a figura 3.6 representa o diagrama de blocos do controlador e o processo.

$$C(s) = Kp \cdot (1 + \frac{1}{t_i s}) = Kp \cdot (\frac{t_i s + 1}{t_i s}) \tag{3.8}$$

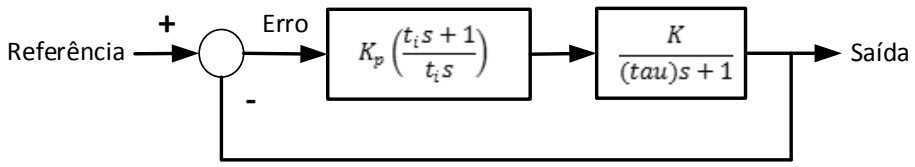


Figura 3.6: Arquitectura de controlo univariável.

Utilizando o método de cancelamento de zero/pólo resultou num t_i igual a:

$$t_i s + 1 = \tau s + 1$$

$$t_i = \tau$$

$$t_i = 0.17$$

Considerando t_i igual a τ e multiplicando a função de transferência do controlador pela função de transferência do processo, obtém-se a equação 3.9:

$$(\frac{t_i s + 1}{t_i s} \cdot \frac{K}{\tau s + 1}) = \frac{0.94}{0.17 s} \tag{3.9}$$

Admitindo que se pretende o pólo do anel fechado numa certa zona do plano z, em -0.3, e aplicando o *rootlocus* à função de transferência (3.9) obteve-se um K_p igual a 0.0562, conforme ilustrado na figura 3.7.

$$K_p = 0.05 \ ; \ T_i = 0.2 \ ; \ T_d = 0$$

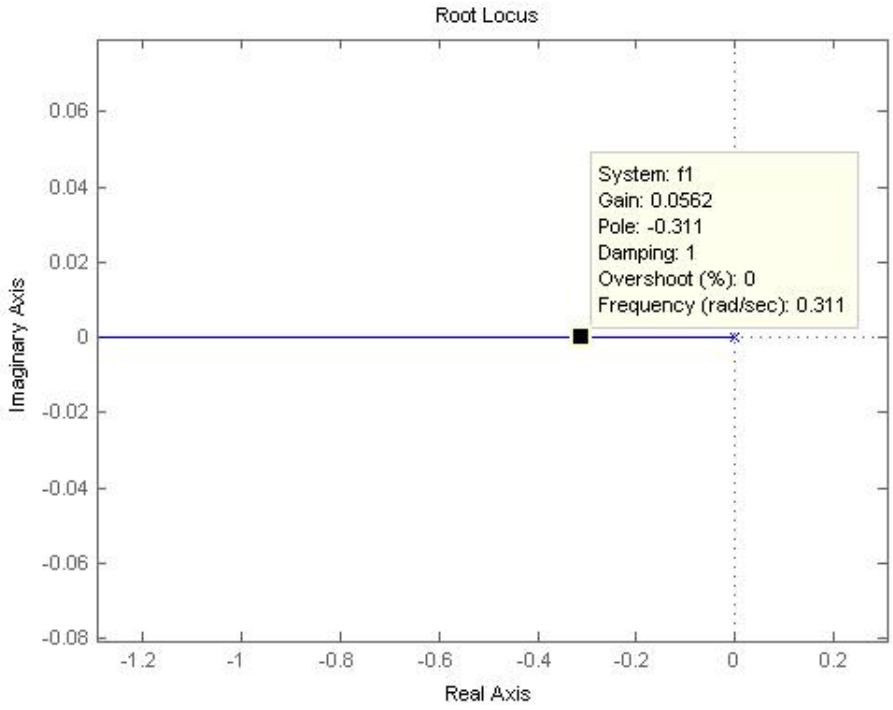


Figura 3.7: *Rootlocus* para posicionamento do pólo.

Para este controlador, a acção de controlo u não foi saturada mas é de salientar que quando é saturada, o fenómeno de *wind-up* pode ocorrer. Isto será visto na implementação do controlador do processo no caso do protocolo CAN na secção 3.2.3.

3.1.4 Análise do Cliente Modbus - Supervisor do Processo

Esta subsecção está reservada à descrição do supervisor, ou seja, o cliente Modbus. Tal como no controlador, o supervisor também é constituido por um PLC e um módulo de entradas e saídas analógicas, conforme a figura 3.8.

Na sequência da implementação de uma interface HMI, recorreu-se ao padrão OPC® porque o PLC mestre está dotado de comunicação OPC®, ou seja, estão disponibilizados

Figura 3.8: Cliente Modbus - Supervisor - Mestre.

os dados internos em uma outra interface, o que resulta que aplicações externas podem interagir com a leitura e escrita das memórias internas do PLC.

Padrão OPC. OPC® como já se fez referência é um padrão de interoperabilidade para troca de dados segura e fiável na área de automação industrial e em outros sectores. Na figura 3.9 está representada a topologia do padrão OPC®.

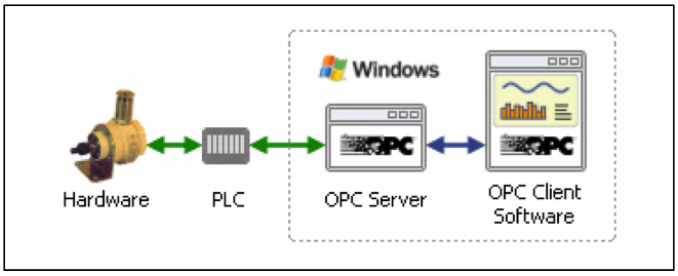


Figura 3.9: Topologia típica do padrão OPC.

O padrão foi lançado pela primeira vez em 1996, sendo que o seu objectivo era criar interfaces padronizadas dos protocolos nos PLCs permitindo que os sistemas HMI/SCADA consigam fazer a interface com o homem que iria converter requisiçoes genéricas OPC® de leitura e escrita em requisições específicas do dispositivo e vice-versa [48].

O padrão OPC® utiliza um protocolo de comunicação entre cliente e servidor, a arquitectura do padrão OPC® adoptada para esta dissertação está representada na figura 3.10. O Scilab® funciona como um OPC cliente e o OPC servidor é o *software* da MatrikonOPC.

Servidor OPC da Matrikon. A conexão de comunicação entre o OPC cliente e o PLC cliente Modbus, ou seja, PLC supervisor, é realizada por um servidor OPC (Matrikon

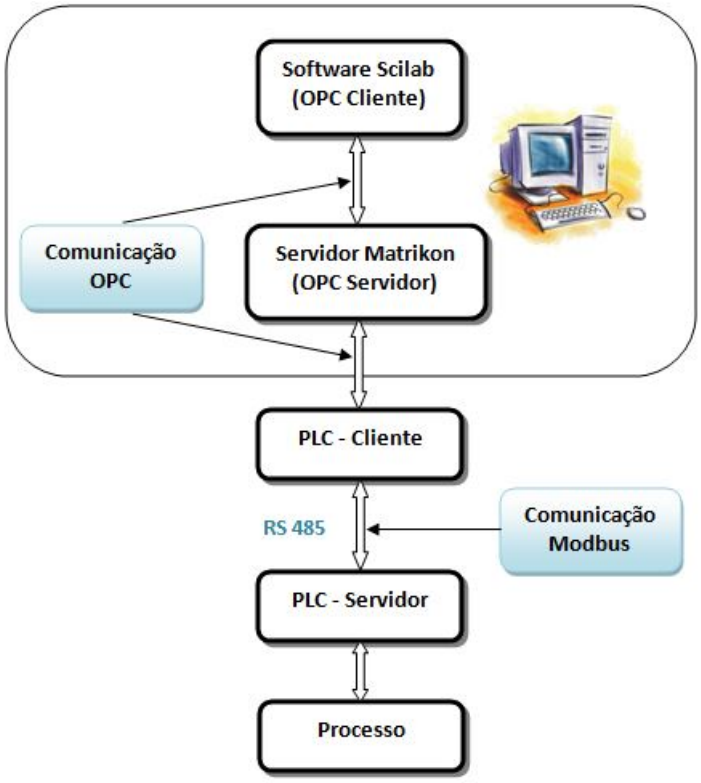


Figura 3.10: Arquitectura OPC a implementar.

OPC). Este servidor possui os *drivers* referentes ao PLC e disponibiliza uma região de dados específica onde é possível ler ou escrever valores das memórias internas, ou até mesmo ler estado de entradas e saídas. O servidor OPC foi criado a partir do *software* *Modbus OPC Server for Modbus Devices* da Matrikon OPC, vide figura 3.11 **A**.

Basicamente, o servidor OPC está subdividido em 3 partes: a) o servidor contendo todos os grupos; b) o grupo que é a camada de organização dos itens OPC; c) item é o elemento principal, o item é o objecto que carrega a informação desejada. Tais partes serão descritas em pormenor na secção 3.1.7.

Cliente OPC em Scilab®. O *software* Scilab® (figura 3.11) **B**, começa na década de 80, como Blaise, *software* criado no INRIA (Instituto Francês de Pesquisa em Informática e Automação) e desenvolvido com a finalidade de promover uma ferramenta de controlo automático para os investigadores. No início da década de 90, o *software* tornou-se Scilab® e de lá para cá já se teve várias versões [49].

O Scilab® surge como uma solução para controlo de processos industriais pois permite a comunicação com o padrão OPC®, e surge também como uma solução grátis e

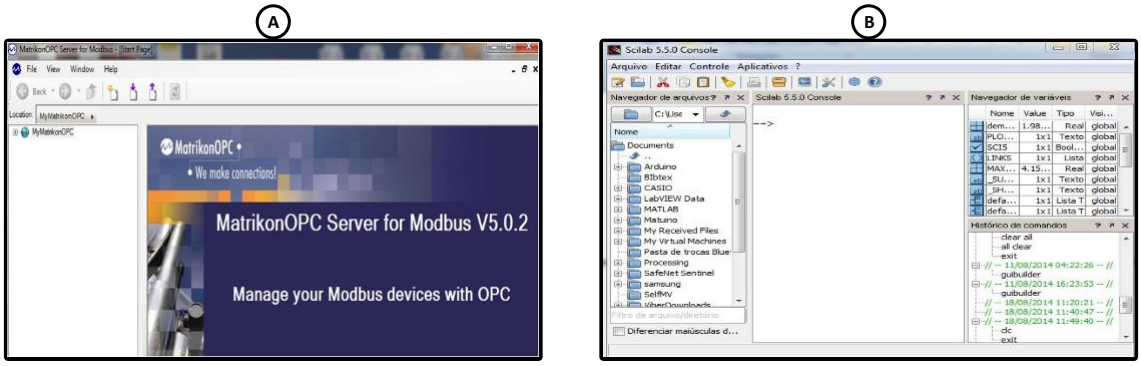


Figura 3.11: Pacotes *Software* utilizados para a comunicação OPC. Em **A** servidor OPC Matrikon OPC e em **B** cliente OPC Scilab.

open-source. Existem outros *softwares* clientes OPC como Matlab® e o Labview®. O Scilab® também fornece muitas *toolboxes* para a área de controlo e optimização. Através da biblioteca GUI[3] do Scilab® é possível criar-se uma interface HMI.

As funções necessárias de acordo com a sintaxe do Scilab OPC cliente, são as seguintes:
Conexão com o servidor OPC:

- *opc_browse()* - comando que procura os servidores OPC disponíveis no computador;
- *opc_connect()* - comando que realiza a conexão com o servidor encontrado criando um cliente, mas sem grupo ou item.

Criação de um grupo no Cliente OPC

- *opc_add_group()* - comando que cria um grupo no cliente, porém sem qualquer item.

Adicionar itens escolhidos ao grupo:

- *opc_add_item()* - comando que adiciona os itens.

Leitura dos dados:

- *opc_item_read(n,'i' ou 'f')* - permite a leitura dos n primeiros itens do servidor em valores inteiros i ou em vírgula flutuante f;

Escrita de dados:

[3]Graphical User Interface - Interface Gráfica com o utilizador.

- *opc_item_write(x,y,'i' ou 'f')* - permite a escrita de um valor y no item x, na forma inteira ou de vírgula flutuante.

Tendo-se apresentado a arquitectura a implementar e analisado o processo, o servidor Modbus - controlador e o cliente Modbus - supervisor juntamente com o padrão OPC, na próxima secção serão apresentadas as principais funções a implementar no PLC TWDLCAA24DRF para o protocolo Modbus.

3.1.5 Funções Modbus no PLC Twido

O PLC utilizado para a implementação, figura 3.12 apresenta as seguintes características:

- 24 I/O discretas das quais 14 são entradas (24V em corrente contínua) e 10 são saídas de relé;
- Permite expandir no máximo 4 módulos;
- Tem ligação Modbus série com Mini Din;
- Tem 2 portas de comunicação;
- Tem modo mestre - escravo com transmissão RTU-ASCII em RS-485 *half-duplex*;
- Função de um controlador PID embutido.

Quanto ao módulo analógico, possui 4 entradas e 2 saídas, de 0 a 10V (corrente contínua) ou 4 a 20 mA.

As varáveis no PLC são definidas nos endereços respeitando a seguinte sintaxe [50]:

$$\%ObjectoFormatoRack.m.c$$

onde: *Objecto* indica se se trata de uma variável de entrada(I), saída de dados (Q), variável interna (M) ou constante interna (K) e *Formato* indica se a variável é do tipo boolena (X), *word* (W), *double* (D) ou *floating* (F). *Rack* contém o endereço do rack, m a posição do módulo no rack e c o canal ao qual se pretende aceder.

Tal como foi descrito nas secções anteriores, o protocolo Modbus tem a limitação de que quando se está a enviar uma requisição para um escravo, o barramento tem de estar somente disponível para essa comunicação, ou seja, o protocolo Modbus é estritamente

(a) TWD LCAA24DRF. (b) TWD AMM6HT.

Figura 3.12: PLC twido LCAA24DRF e módulo de entradas e saídas analógicas AMM6HT.

half-duplex. Para evitar conflitos na comunicação, o PLC em referência para a implementação deste protocolo oferece dois serviços de comunicação:

- Instrução EXCHx: para transmitir/receber mensagens.
- Bloco da função %MSGx: para controlar a troca de mensagem.

A instrução EXCHx e o bloco da função %MSGx podem ser acedidos anexando o número da porta (1 ou 2). O PLC no modo mestre em RTU tem o suporte da instrução EXCHx. O tamanho máximo da trama transmitida e recebida é de 250 *bytes*.

A instrução EXCHx permite ao controlador mestre enviar e/ou receber informação do escravo. Para uma requisição Modbus é definido uma tabela de *words*, (%MWi:L), contendo informações de controlo e os dados a serem transmitidos e/ou recebidos. A troca de mensagens é feita usando a instrução EXCHx:

Sintaxe : [EXCHx %MWi:L] onde:

x = número da porta de comunicação(1 ou 2);

i = memória inicial;

L = número de *words* de controlo, transmissão e recepção.

O uso do bloco de função %MSGx é opcional, este bloco tem duas entradas e duas saídas associadas a ele e pode ser utilizado para gerir a troca de dados, conforme tabela 3.1. O bloco de função %MSGx tem dois objectivos: verificação de erro na comunicação e coordenação de múltiplas mensagens.

Uma requisição Modbus no PLC é definida pela tabela de *words* associada à instrução EXCHx, sendo constituida pelas tabelas de controlo, transmissão e recepção (vide tabela 3.2).

Tabela 3.1: Bloco da função MSGx.

Entrada/Saída	Definição	Descrição
R	Entrada *Reset*	Definido 1 : reinicializa a comunicação (%MSGx.E = 0 e %MSGx.D = 1)
%MSGx.D	Comunicação completa	0 : requisição em progresso 1 : comunicação completa no fim da transmissão,fim do caracter recebido, erro, ou reset do bloco
%MSGx.E	Erro	0 : tamanho da mensagem OK e conexão OK. 1 : mau comando, tabela incorrectamente configurada, caracter recebido incorrecto, ou tabela de recepção cheia.

Tabela 3.2: Tabela de *words*.

	Byte Mais Significativo	*Byte* Menos Significativo
Tabela de Controlo	Comando	Tamanho (Transmissão/Recepção)
	Recepção(Rx) *offset*	Transmissão(Tx) *offset*
Tabela de Transmissão	1 *Byte* Transmitido	2 *Byte* Transmitido
	—	—
	—	N *Byte* Transmitido
	N+1 *Byte* Transmitido	
Tabela de Recepção	1 *Byte* Recebido	2 *Byte* Recebido
	—	—
	—	P *Byte* Recebido
	P+1 *Byte* Recebido	

Na tabela de controlo o *byte* comando, em caso de uma requisição *unicast* (excepto para *broadcast*) Modbus RTU deve ser sempre 01, o *byte* de comando no caso de uma requisição *broadcast* deve ser 00, enquanto o endereço do escravo deve ser definido para 00. O *byte* tamanho contém o tamanho de transmissão (máximo 250 *bytes*). Este parâmetro é o tamanho em *bytes* da tabela de transmissão. Se Tx *offset* é igual a 0, este parâmetro é igual ao comprimento da trama de transmissão. Se o parâmetro Tx *offset* não é igual a 0, um *byte* da trama de transmissão (indicado pelo valor Tx *offset*) não irá ser transmitido e este parâmetro é igual ao tamanho da trama em si, menos 1.

O *byte* Tx *offset* contém o valor da posição do *byte* (1 para o primeiro *byte*, 2 para o segundo *byte*, e assim por diante), dentro da trama de transmissão para ignorar quando transmitir os *bytes*. Por exemplo, se este *byte* contém 3, o terceiro *byte* seria ignorado, fazendo com que o quarto *byte* na trama fosse transmitido como terceiro *byte*.

O *byte* Rx *offset* contém o valor 1 para o primeiro *byte*, 2 para o segundo *byte*, e assim por diante dentro da tabela de recepção para adicionar ao receber uma trama. Por

exemplo, se este *byte* contém 3, o terceiro *byte* na tabela de recepção seria preenchido com um zero, e o terceiro *byte*, que foi actualmente recebido será introduzido na quarta posição da tabela de recepção. Tx *offset* e Rx *offset* são utilizados para lidar com as questões associadas aos valores de *bytes* ou *words* dentro do protocolo Modbus no PLC, estes *offsets* são utilizados para facilitar a interpretação dos *bytes* e *words* de maneira que "caiam"nos limites de *words*.

Na utilização do modo RTU, a tabela de transmissão é preenchida com a requisição antes de executar a instrução EXCHx. Em tempo de execução, o controlador determina que camada de ligação de dados é, e realiza todas as conversões necessárias para processar a transmissão e a resposta. Início, fim da trama e a verificação dos caracteres não são armazenados nas tabelas de transmissão/recepção.

Após todos os *bytes* serem transmitidos, o PLC cliente Modbus ou mestre passa para o modo de recepção e espera para receber qualquer *byte*. A recepção é concluída caso se verifique: o tempo de espera da trama foi detectado ou a tabela de recepção está completa.

A tabela de transmissão contém os dados a serem transmitidos. O primeiro *byte* contém o endereço específico de um servidor Modbus ou *broadcast*, o segundo *byte* contém o código de função e o resto contém as informações associadas a esse código de função [4].

A tabela de recepção contém os dados que estão sendo recebidos. O primeiro *byte* contém o endereço do dispositivo, o segundo *byte* contém o código de função, e o resto contém informação associada a esse codigo de função.

Tendo-se visto as principais funções utilizadas para o protocolo Modbus no PLC Twido e a tabela de *words* de uma requisição Modbus, na secção seguinte serão descritos alguns exemplos de leitura e escrita de *words* da comunicação Modbus, considerando dois PLCs, um cliente (mestre Modbus) e um servidor (escravo Modbus).

3.1.6 Exemplos de Leitura e Escrita de *words* em Modbus no PLC Twido

Nesta subsecção veremos exemplos de programação de leitura e escrita de *words* em um escravo ou servidor utilizando dois tipos de codificação, utilizando o formato de uma trama em Modbus vista no capítulo 2 (endereço, código de função Modbus, dados e ve-

[4] Nestas informações inclui itens como endereços discretos e de registo, quantidade de itens a serem tratados, e a contagem de *bytes* de dados

rificação de erros) e o disponibilizado pelo PLC Twido, utilizando as definições macro (*Macro Definition Function*) da *Schneider Electric*, sendo estas feitas na linguagem lista de instruções. A configuração de uma rede Modbus e dos PLCs será vista na secção 3.1.7. Para esta secção, o objectivo é de somente mostrar como é feito o código de programação para uma requisição Modbus de leitura e escrita de *words*.

O PLC utilizado para a implementação disponibiliza duas linguagens de programação IL e Ladder. Para a implementação do protocolo Modbus, o PLC será programado em Ladder e de seguida se converterá para a linguagem IL para a execução no PLC. A IL é uma linguagem de programação textual de baixo nível semelhante ao *Assembly*. As instruções são identificadas por uma letra ou um pequeno conjunto de letras associadas aos endereços onde se encontram as informações a serem trabalhadas no programa [51]. A linguagem IL tem a vantagem de possuir uma maior velocidade de execução no PLC.

Na figura 3.13 é descrito um exemplo de leitura de 4 *words* em hexadecimal, as %MW correspondem às memórias internas do PLC.

```
1     // MESTRE //                    // ESCRAVO //
2  LD 1                            LD 1
3  [ %MW0 := 16#0106 ]             [ %MW0 := 16#6566 ]
4  [ %MW1 := 16#0300 ]             [ %MW1 := 16#6768 ]
5  [ %MW2 := 16#0203 ]             [ %MW2 := 16#6970 ]
6  [ %MW3 := 16#0000 ]             [ %MW3 := 16#7172 ]
7  [ %MW4 := 16#0004 ]             END
8  LD 1
9  AND %MSG2.D
10 [ EXCH2 %MW0:10 ]
11 LD %MSG2.E
12 END
```

Figura 3.13: Exemplo de leitura de 4 *words*.

No PLC cliente Modbus é programado o código "MESTRE"e no PLC servidor Modbus o código "ESCRAVO". Recorrendo à tabela 3.2 (tabela de *words*), os códigos das memórias %MW0 e %MW1 correspondem à tabela de controlo e da memória %MW2 a memória %MW4 corresponde à tabela de transmissão. A tabela de recepção será vista na tabela de animação do *software* Twidosuite do PLC mestre.

Na memória %MW0 o *byte* 01 corresponde ao comando e como já foi descrito é sempre 1 quando se trata de uma transmissão *unicast* em RTU, o *byte* 06 corresponde ao tamanho da tabela de transmissão. A memória %MW1 corresponde à Rx *offset* sendo igual ao *byte* 03 e a Tx *offset* sendo igual ao *byte* 00. O valor 0 (zero) em Tx *offset* significa que o

parâmetro tamanho é igual ao comprimento da trama de transmissão. A utilização do Rx *offset* definido na memória %MW1 do mestre, o deslocamento 03, adicionará um *byte* (valor = 0) na terceira posição da tabela de recepção. Isso alinha as *words* do mestre para que "fiquem"correctamente nos limites da *word*. Sem essa compensação, cada *word* de dados seria dividida em duas *words*.

O primeiro *byte* 02 da memória %MW2 , corresponde ao endereço do escravo, o *byte* seguinte é o código de função; Na tabela 2.3, viu-se que 03 corresponde a leitura de um *holding registers*. A memória %MW3 indica a primeira memória a ser lida no escravo (%MW0), enquanto que %MW4 indica o número de memórias a serem lidas 4. Antes de executar a instrução EXCH2 que indica o número de *words* da tabela de *words*, a aplicação verifica a comunicação %MSG2.D e finalmente o estado de erro %MSG2.E.

Abrindo a tabela de animação do mestre, resultará na figura 3.14. O primeiro *byte* da memória %MW5 o *byte* 02 corresponde ao endereço do escravo enquanto que o *byte* seguinte, 03, corresponde ao código de função.

```
1 // Tabela de animação do Master //
2 %MW5   0203   0000   Hexadecimal  //Endereço 02, código de função 03
3 %MW6   0008   0000   Hexadecimal  // 00 Rx offset, 08 bytes recebidos
4 %MW7   6566   0000   Hexadecimal
5 %MW8   6768   0000   Hexadecimal
6 %MW9   6970   0000   Hexadecimal
7 %MW10  7172   0000   Hexadecimal
```

Figura 3.14: Tabela de animação do Mestre para a leitura de 4 *words*.

A memória %MW6 indica o número de *bytes* recebidos (08). O primeiro *byte* da memória %MW6, com o valor 00 corresponde ao terceiro *byte* recebido e foi definido com o valor 0 (zero) pelo Rx *offset*. As *words* lidas do escravo (a partir da memória %MW7) estão alinhados corretamente com o limite das *words* do mestre.

Quanto à escrita de *words*, na figura 3.15 está representado um exemplo de escrita de duas *words* no escravo. Para o escravo, programou-se uma única *word* somente para atribuir espaço no escravo para os endereços de memória, sem a atribuição de espaço a requisição Modbus estaria tentando escrever em locais que não existiam no escravo.

O Tx *offset* o segundo *byte* na memória %MW1 igual a 07, o segundo *byte* após o *byte* Rx *offset* igual a 00, irá eliminar o *byte* mais significativo da sexta *word* (valor hexadecimal

```
1    // MESTRE //                    // ESCRAVO //
2  LD 1                           LD 1
3  [ %MW0 := 16#010C ]            [ %MW0 := 16#FFFF ]
4  [ %MW1 := 16#0007 ]            END
5  [ %MW2 := 16#0210 ]
6  [ %MW3 := 16#0010 ]
7  [ %MW4 := 16#0002 ]
8  [ %MW5 := 16#0004 ]
9  [ %MW6 := 16#6566 ]
10 [ %MW7 := 16#6768 ]
11 LD 1
12 AND %MSG2.D
13 [ EXCH2 %MW0:11 ]
14 LD %MSG2.E
15 END
```

Figura 3.15: Exemplo de escrita de duas *words*.

00 em %MW5). Isso funciona para alinhar os valores dos dados na tabela de transmissão da tabela de *words* de modo que eles fiquem corretamente nos limites das *words*.

No mestre, a tabela de *words* da instrução EXCHx é inicializada para escrever 2 *words* (4 *bytes*) no escravo com endereço 02 usando o código de função 10 em hexadecimal, 16 em decimal (escrita de mútiplos registros), memória %MW2.

A memória %MW3 indica a localização da primeira memória a ser escrita no escravo (10 hexadecimal, 16 em decimal). %MW4 indica o número de *words*, enquanto que %MW5 o número de *bytes*. A figura 3.16 mostra a tabela de animação do mestre e do escravo. Da memória %MW8 à memória %MW10 correspondem a tabela de recepção.

```
1  // Tabela de animação do Mestre //      // Tabela de animação do Escravo //
2  %MW0   010C   0000   Hexadecimal        %MW16   6566   0000   Hexadecimal
3  %MW1   0007   0000   Hexadecimal        %MW17   6768   0000   Hexadecimal
4  %MW2   0210   0000   Hexadecimal
5  %MW3   0010   0000   Hexadecimal
6  %MW4   0002   0000   Hexadecimal
7  %MW5   0004   0000   Hexadecimal
8  %MW6   6566   0000   Hexadecimal
9  %MW7   6768   0000   Hexadecimal
10 %MW8   0210   0000   Hexadecimal
11 %MW9   0010   0000   Hexadecimal
12 %MW10  0004   0000   Hexadecimal
```

Figura 3.16: Tabela de animação do Mestre e do Escravo.

Para facilitar a programação destes PLCs para uma comunicação Modbus entre dois equipamentos, a *Schneider Electric* disponibiliza uma abordagem baseada nas funções de definições macro (*Macro Definition Function*). São funções de leitura e escrita tanto de

bits como de *words*. Abordaremos apenas a escrita e leitura de *words* para compararmos com os exemplos anteriores.

Na função ***C_RDNW*** (figura 3.17) é possível ler *N words* no barramento. Nesta função aparecem dois parâmetros, nomeadamente o indicador do número da macro na instância e o indicador do número de *words* para ler. Este exemplo lê 10 *words* do escravo usando a instância 2 começando na memória %MW5.

```
1  LD 1
2  [ C_RDNW_ADDR1_2 := 5 ] //Carrega o endereço %MW5 da instância 2.
3
4  [ C_RDNW 2 10 ] //Pedido ao escravo para ler 10 words usando a instancia 2.
```

Figura 3.17: Exemplo de leitura de 10 *words*.

C_WRNW, figura (3.18) é a função que permite escrever *N words* sobre o barramento, também tem dois parâmetros, o indicador da instância e o número de *words*. Neste exemplo escreveu-se duas *words* utilizando a instância 15 começando na memória %MW7.

```
1  LD 1
2  [ C_WRNW_ADDR1_15 := 7 ] //Carrega o endereço %MW7 da instância 15.
3
4  [ %MW0 := 0 ]
5  [ C_WRNW_VAL1_15[ %MW0] := 16#1234 ] //Carrega 1234 na instância 15.
6
7  [ %MW0 := 1 ]
8  [ C_WRNW_VAL1_15[ %MW0] := 16#5678 ] //Carrega 5678 na instância 15.
9
10 [ C_WRNW 15 2 ]  // Escrever duas words usando a instancia 15.
```

Figura 3.18: Exemplo de escrita de duas *words*.

Embora a programação utilizando as definições macro seja mais simples e curta em comparação com a programação padrão de um protocolo Modbus, nota-se claramente que em termos daquilo que foi o estudo do protocolo Modbus no capítulo 2 esta programação não seria a ideal, por não possuir a estrutura de uma trama em Modbus. No entanto, para a implementação desta dissertação utilizou-se a programação que mais se parece com o formato padrão de uma trama Modbus.

Tendo-se apresentado as principais funções Modbus no PLC e exemplos de programação de leitura e escrita de *words* com o protocolo Modbus, é chegada a altura da implementação do protocolo para monitorização e controlo do processo assim como a criação de uma in-

terface HMI através do padrão OPC.

3.1.7 Implementação da rede Modbus no controlo do processo

Para a criação da rede Modbus, seguiram-se as etapas indicadas na figura 3.19, começando pela configuração do *hardware* do PLC mestre e escravo, conexão dos cabos de ligação (D1, D0 e Comum), em seguida a configuração da porta do PLC (PLC Twido possui duas portas de comunicação), programação da aplicação (em Ladder ou IL) e por último a inicialização da tabela de animação[5].

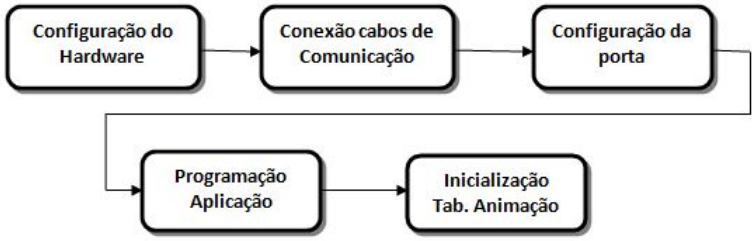
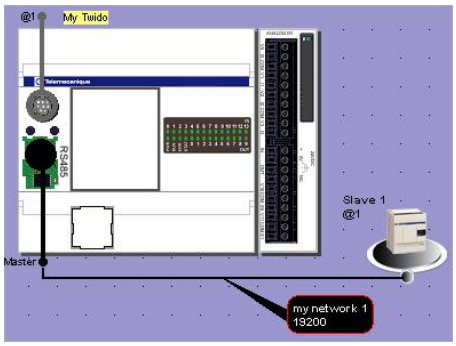
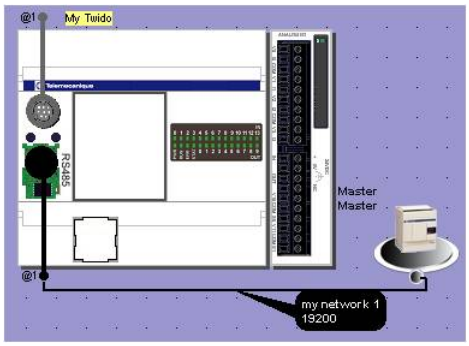


Figura 3.19: Configuração de uma ligação Modbus.

Configuração da rede e dos PLC´s. A implemetação do sistema começa na configuração dos dispositivos e seus módulos. A figura 3.20 representa a configuração do *hardware* para o PLC cliente e servidor ambos com o módulo de entradas e saídas analógicas e a interface de comunicação série RS-485 na porta 2 com ligação mini Din de 8 pinos.



(a) Configuração do hardware mestre.



(b) Configuração do hardware escravo.

Figura 3.20: Configuração do Hardware.

O controlador, PLC escravo figura 3.20(b), foi configurado com o endereço 1. Embora o mestre não seja endereçável na sua configuração foi-lhe atribuido o endereço 2 para que seja acedido no *software* Matrikon OPC (servidor OPC). Tanto o mestre como o escravo

[5]A tabela de animação mostra o resultado de uma comunicação Modbus.

têm de ter a mesma configuração de rede. Assim, foi definido um *baudrate* de 19200 kpbs, modo RTU, um tempo de resposta (*Response Timeout*) de 1 segundo e o tempo entre as tramas (*Time between frames*)[6] de 10 ms (figura 3.21).

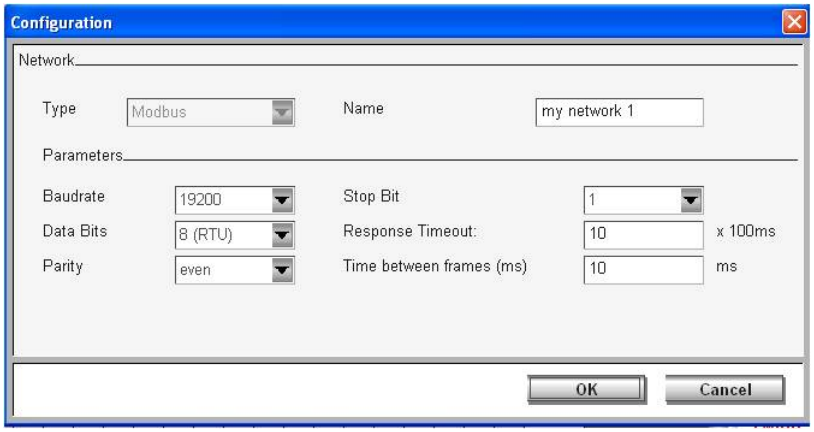


Figura 3.21: Configuração da rede.

Quanto às portas de comunicação, a porta 1 foi utilizada para a programação do PLC e a porta 2 foi utilizada para a comunicação Modbus para ambos os casos tanto mestre como escravo. O protocolo Modbus sugere que se utilize o modo padrão, paridade par; foi assim definido este modo de paridade. Após a configuração da porta, chega-se então à fase de programação da aplicação.

O servidor, ou seja, o controlador Modbus foi programado de maneira a que o processo reaja em função do que foi definido na análise do processo, tendo um controlador liga - desliga para controlar a temperatura do simulador [30 ºC ; 60 ºC] e um outro controlador PI para o controlo da velocidade dos ventiladores (3.22). Por último associou-se as entradas e saídas analógicas às memórias internas (%MW) para que sejam acedidas pelo PLC supervisor[7]. As memórias %MW12, 13 e 14 estarão associadas à saída do processso (ventiladores), acção de controlo e a temperatura do simulador no PLC do controlador. A referência será escrita na memória %MW40[8] do controlador. Em seguida, foi definido o ganho K_p, o tempo integral T_i e o tempo de amostragem de 10 ms enquanto que T_d é igual a 0 porque estamos perante um controlador PI, conforme ilustrado na figura 3.22

A programação do PLC cliente (Mestre) foi feita de maneira a permitir ao mestre consiga ler as memórias associadas as variáveis do processo (saída (velocidade dos venti-

[6] O tempo entre as tramas representa o *Turnaround delay* visto no capítulo 2. Que representa o atraso respeitado pelo mestre a fim de permitir que o escravo processa a requisição actual antes de enviar uma nova.

[7] A troca de dados entre o servidor e o cliente só pode ser feita com memórias internas

[8] 28 em hexadecimal

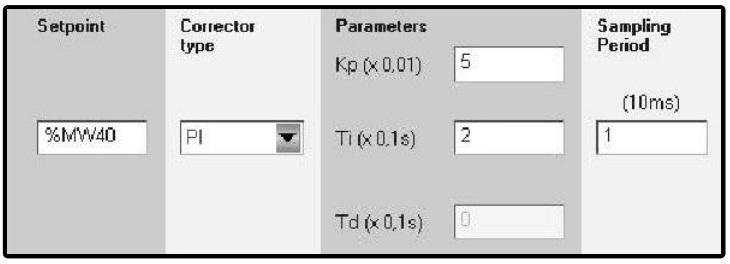


Figura 3.22: Inserção dos valores do ganho e do tempo do controlador.

ladores), acção de controlo dos ventiladores e a temperatura do simulador) e a escrita do valor de referência do processo no PLC escravo.

Testes da comunicação Modbus. Depois da configuração da rede e dos PLCs foram feitos testes na rede de comunicação do protocolo de maneira a verificar o funcionamento da rede Modbus entre o cliente e o servidor Modbus; os resultados destes testes encontram-se no capítulo 4 (resultados experimentais).

No entanto, como se viu no capítulo 2, o mestre emite requisições para o escravo de 2 modos *unicast* e *broadcast*. Foram feitos testes para uma requisição *unicast* onde o servidor pode receber a requisição sem nenhum erro de comunicação e retornar uma resposta normal; outro teste foi o servidor receber uma requisição sem nenhum erro mas não poder manipular o pedido, logo o servidor retornará uma *exception response*.

Em seguida será feito um teste de comunicação em *broadcast* e um teste com objectivo de se detectar erro na comunicação neste caso o bloco de função %MSG2.E e a *word* do sistema %SW64 informarão o tipo de erro. Para além do uso do *bit* do sistema associado a instrução EXCH, a *word* do sistema %SW64 contém o código de erro da porta 2, porta utilizada para a comunicação Modbus. Na tabela 3.3 estão representados os possíveis códigos que a *word* do sistema pode apresentar.

O primeiro teste consistiu em colocar o servidor a receber a requisição de leitura de três memórias associadas a saída do processo (ventiladores), acção de controlo (ventiladores) e a temperatura do simulador, figura 3.23, ser escrita a referência do processso, figura 3.24, sem nenhum erro e manipular os pedidos normalmente, ou seja retornar em ambos os casos uma resposta normal.

As memórias %MW24 e 23 (figuras 3.23 e 3.24) estão relacionadas com o tempo em que o PLC cliente está em modo de leitura e escrita, após estas memórias está representado

Tabela 3.3: Tabela do código de erro para a *word* do sistema %SW64.

%SW64 código de erro para a porta 2
0 - Operação foi bem sucedida
1 - Número de bytes a ser transmitido é demasiado grande
2 - Tabela de transmissão demasiado pequena
3 - Tabela de word demasiado pequena
4 - Tabela de recepção demasiado cheia
5 - Tempo de espera decorrido
6 - Transmissão
7 - Comando incorrecto dentro da tabela
8 - Porta seleccionada não configurada/disponível
9 - Erro de recepção
10 - Não pode usar %KW se estiver a receber
11 - Transmissão offset maior que a tabela de transmissão
12 - Recepção offset maior que a tabela de recepção
13 - Controlador parou o processamento de EXCH

```
1  (*LER AS VARIÁVEIS*)
2  LD  %M24
3  AND  %MSG2.D
4  [ %MW0 := 16#0106 ]  // 01 comando e 06 tamanho (6 bytes)
5  [ %MW1 := 16#0300 ]  // Rx offset 03 e 00 Tx offset
6  [ %MW2 := 16#0103 ]  // 01 endereço do escravo e 03 código da função
7  [ %MW3 := 16#000C ]  // C, 12 em decimal a primeira a memória a ser lida
8  [ %MW4 := 16#0003 ]  // Quantidade de memórias a serem lidas 3
9  [ EXCH2 %MW0:10 ]
10 AND  %MSG2.E         // %MSG2.E e %SW64 relacionadas com possíveis erros
11 [ %SW64 := 13 ]
```

Figura 3.23: Leitura de 3 memórias no PLC escravo.

os códigos de uma requisição ou solicitação Modbus para o PLC servidor. Na memória %MW3 é definida a primeira memória a ser lida e tal como se pode ver na figura 3.23 a letra C hexadecimal corresponde a 12[9]. A memória %MW4 corresponde à quantidade de memórias a serem lidas 3, %MW12, 13 e 14, correspondente à temperatura do simulador, a saída do processo e a acção de controlo.

A memória %MW13 indica a memória a ser escrita no servidor (28 em hexa, 40 em decimal). A memória %MW14 corresponde ao valor a ser enviado ou escrito no PLC servidor Modbus e este valor está associado à memória %MW99 que por sua vez será igual à memória %MW150[10], ou seja, %MW14 := %MW99; %MW99 := %MW150.

A tabela 3.4 resume as memórias associadas ao PLC servidor Modbus - controlador e

[9] %MW12 no servidor está associada à saída do processo (velocidade dos ventiladores)

[10] Esta memória %MW150 será configurada no OPC cliente, para que a referência seja enviada pelo cliente OPC em Scilab®.

```
(*ESCREVER N WORDS*)
LD  %M23
AND  %MSG2.D
[ %MW10 := 16#0106 ]  // 01 comando e 06 tamanho (6 bytes)
[ %MW11 := 16#0000 ]  // Rx offset 00 e 00 Tx offset
[ %MW12 := 16#0106 ]  // 01 endereço do escravo e 06 código da função
[ %MW13 := 16#0028 ] // Memória a ser escrita 28 hexadecimal e 40 em decimal
[ %MW14 := %MW99 ]    // Valor a enviar para a memória do escravo
[ EXCH2 %MW10:8 ]
AND  %MSG2.E
[ %SW64 := 13 ]
```

Figura 3.24: Escrita da variável de referência no PLC escravo.

ao cliente Modbus - supervisor. O padrão OPC será descrito mais adiante, observou-se na tabela que os dados são endereçáveis no PLC cliente Modbus como $x - 1$.

Tabela 3.4: Variáveis do processo associadas aos diferentes itens.

Variáveis do processo associadas aos diferentes itens			
	Servidor Modbus	Cliente Modbus	Padrão OPC
Tª do Simulador	%MW12	%MW7	3:8
Saída do Processo (ventiladores)	%MW13	%MW8	3:9
Acção de Controlo	%MW14	%MW9	3:10
Referência (Leitura)	%MW40	%MW17	3:18
Referência (Escrita)	%MW40	%MW150	4:151

Quanto às diferentes referências na tabela acima, a referência escrita é o valor definido pelo supervisor do processo que é constituído pelo PLC cliente Modbus, MatrikonOPC servidor OPC e o cliente OPC em Scilab®. A referência leitura corresponde à confirmação por parte do PLC servidor Modbus quando responde à requisição por parte do PLC cliente Modbus. Ambas as referências terão os mesmos valores, sendo que o valor de referência leitura foi considerado somente para confirmar se realmente o valor escrito na referência do PLC servidor Modbus é igual ao valor definido pelo cliente OPC em Scilab®.

%SW64 corresponde à *word* do sistema associada à porta 2 do PLC e o número 13 representa a quantidade de possíveis códigos que podem surgir na comunicação, conforme a tabela 3.3. Para além da memória do sistema, a saída do bloco funcional (%MSG2.E) também informará possíveis erros (vide a tabela 3.1). Está sendo apresentado (figuras 3.23 e 3.24) somente o código do PLC cliente Modbus porque uma requisição é sempre iniciada pelo mestre, ou seja, os códigos de pedido e escrita estão sempre colocados no lado do PLC cliente, o servidor ou escravo nunca inicia uma requisição.

Este primeiro teste representará o código de programação que estará em funcionamento para o controlo e monitorização do processo. Mais adiante, a implementação do padrão OPC® será em função do funcionamento deste código de programação e da tabela 3.4.

No segundo teste, o servidor foi colocado a receber uma requisição do mestre sem nenhum erro, mas este não pode manipular o pedido. Neste teste, fez-se o pedido de leitura de uma memória que não foi programada no servidor Modbus (%MW3 := 16#0032 ler a memória 32 em hexadecimal, 50 em decimal), para o qual o servidor retornará uma *exception response*. Na figura 3.25, está representado o código do cliente Modbus para o caso de uma *exception response*.

```
(*LER AS VARIÁVEIS*)
LD  %M24
AND  %MSG2.D
[ %MW0 := 16#0106 ]      // 01 comando e 06 tamanho
[ %MW1 := 16#0300 ]      // Rx offset 03 e 00 Tx offset
[ %MW2 := 16#0103 ]      // 01 endereço do escravo e 03 código da função
[ %MW3 := 16#0032 ]      // 32 em hexa, 50 em decimal a memória a ser lida
[ %MW4 := 16#0001 ]      // Uma memória a ser lidas
[ EXCH2 %MW0:10 ]
AND  %MSG2.E
[ %SW64 := 13 ]
```

Figura 3.25: Comunicação com *exception response*, leitura de uma memória inexistente no PLC servidor.

Para uma comunicação em que o servidor Modbus retorna uma *exception response*, é possível o servidor enviar um dos 9 códigos de excepção. Na tabela 3.5 são referenciados esses códigos [3].

Tabela 3.5: Lista dos códigos de excepção [3].

Códigos de excepção Modbus	
Código	Nome
01	Função inválida
02	Endereço inválido
03	Valor dos dados inválidos
04	Falha no escravo
05	Reconhecimento
06	Escravo ocupado
08	Erro na memória de paridade
0A	*Gateway* indisponível
0B	*Gateway* não respondeu

Como se viu, a requisição em *broadcast* é aquela em que nenhuma resposta é retornada

por parte do escravo; para este teste foi feito a escrita de um valor na memória %MW40 no servidor. Era de esperar que para uma requisição em *broadcast* o endereço do escravo fosse igual a 0, para o comando na tabela de controlo da tabela *words* também tem de ser igual a 0. Na figura 3.26, é representado o código em IL do PLC mestre na requisição em *broadcast*.

```
(*ESCREVER N WORDS*)
LD  %M23
AND  %MSG2.D
[ %MW10 := 16#0006 ]    // Comando 00, tamanho 06
[ %MW11 := 16#0000 ]    // Rx igual a 00, Tx 00
[ %MW12 := 16#0006 ]    // Endereço broadcast 00, 06 código de função
[ %MW13 := 16#0028 ]    // Memória a ser escrita
[ %MW14 := %MW99 ]      // Valor a escrever na memória
[ EXCH2 %MW10:8 ]
AND  %MSG2.E
[ %SW64 := 13 ]
```

Figura 3.26: Comunicação *broadcast* escrita de uma memória no PLC escravo.

E por último, fez-se o teste de um erro de comunicação em que durante a requisição do pedido de leitura das memórias do PLC servidor, removeu-se o cabo de comunicação entre o PLC servidor e o PLC cliente, figura 3.27, com objectivo de verificar que valores surgem na saída do bloco %MSG2.E e na *word* do sistema %SW64.

```
(*LER AS VARIÁVEIS*)
LD  %M24
AND  %MSG2.D
[ %MW0 := 16#0106 ]      // 01 comando e 06 tamanho
[ %MW1 := 16#0300 ]      // Rx offset 03 e 00 Tx offset
[ %MW2 := 16#0103 ]      // 01 endereço do escravo e
[ %MW3 := 16#000C ]      // C, 12 em decimal a primeira a memória a ser lida
[ %MW4 := 16#0003 ]      // 3 memórias a serem lidas
[ EXCH2 %MW0:10 ]
AND  %MSG2.E
[ %SW64 := 13 ]
```

Figura 3.27: Comunicação com erro, leitura das variáveis no PLC escravo.

No capítulo 4 desta dissertação serão apresentados os resultados destes testes a partir da tabela de animação do mestre no programa Twidosuite e será feita a análise destes resultados. No caso de uma requisição de escrita de um certo valor no PLC escravo, achou-se que não seria necessário apresentar a tabela de animação do escravo porque nesta requisição o escravo ou servidor retorna para o mestre o valor que foi escrito.

Servidor OPC. Após os testes da comunicação Modbus, e considerando o primeiro caso em que o servidor Modbus recebe as requisições de leitura e escrita do cliente Modbus sem nenhum erro de comunicação e retorna uma resposta normal, é chegado o momento da implementação da arquitectura do padrão OPC®, representada na figura 3.10.

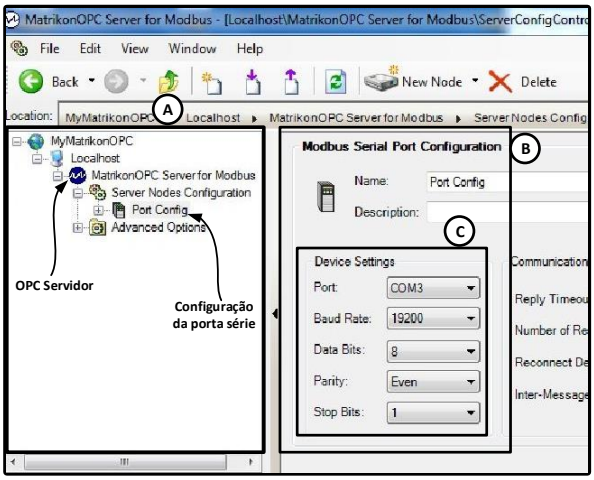


Figura 3.28: Servidor MatrikonOPC para Modbus - Configuração da rede.

Em primeiro lugar, criou-se um servidor OPC para poder trocar dados com o PLC cliente e por sua vez o servidor OPC (MatrikonOPC) trocar dados com o cliente OPC (Scilab®) que será a interface HMI do processo.

Na criação do servidor OPC no *software* da Matrikon, figura 3.28, em primeiro lugar configuram-se as definições da comunicação série do PLC cliente Modbus, definições essas que serão as mesmas com a configuração da rede Modbus[11] que permite a comunicação entre o PLC cliente e o PLC servidor. Em **A** representa-se a criação do servidor, em **B** configuração da porta e em **C** as definições do dispositivo (PLC Modbus cliente).

Na figura 3.29 surge o porquê de se ter atribuido na configuração do PLC cliente um certo endereço, endereço 2, rectângulo **B**. Para a configuração série de um PLC no servidor OPC, é necessário que se defina o endereço do PLC que se quer configurar, sendo por isso que foi atribuido um endereço ao PLC cliente Modbus .

Depois de se ter definido o endereço do PLC cliente fica assim criado o servidor OPC. Continuando no *software* da Matrikon na aplicação *Matrikon Explore*, figura 3.30 será feita a criação de um grupo para conter os itens no OPC servidor, *Group0*. Para adicionar itens ao grupo, é necessário que se cumpram as regras de endereçamento e como já se

[11] *Baudrate* 19200, modo RTU, sem paridade e 1 *stop bit*.

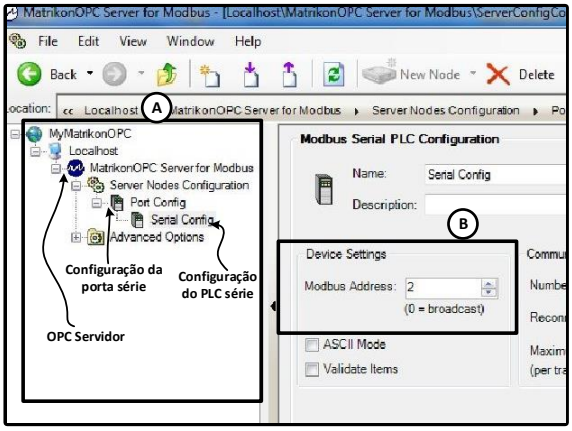


Figura 3.29: Servidor MatrikonOPC para Modbus - Configuração do PLC cliente.

viu, na subsecção 2.3.2, os dados são endereçáveis de 0 a 65535, o que significa que por exemplo a memória %MW7 é endereçável na PDU como 8.

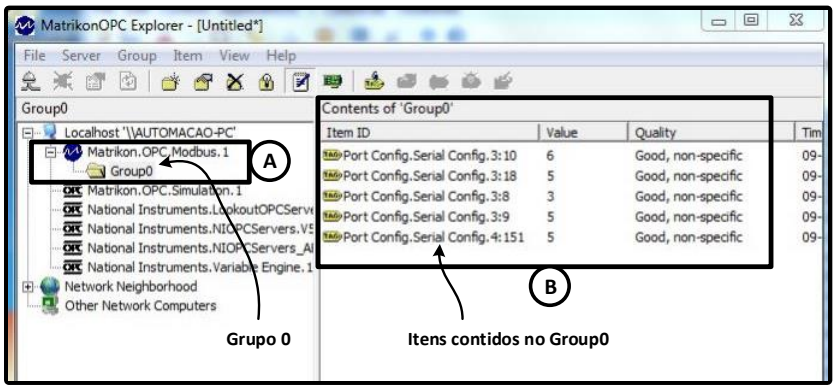


Figura 3.30: Matrikon Explore - Criação do grupo e dos itens OPC.

O modelo de dados da aplicação Modbus baseia-se nas tabelas primárias (tabela 2.2), em que 0 corresponde a uma entrada discreta, 1 a uma bobina (*coil*), 3 a uma entrada de registros e 4 a uma *holding register*. Na figura 3.30 foram adicionados os seguintes itens:

- A temperatura do simulador (*Port Config.Serial Config 3:8*)
- Saída do processo - velocidade dos ventiladores (*Port Config.Serial Config 3:9*);
- Acção de controlo (*Port Config.Serial Config 3:10*)
- Referência para ser escrita (*Port Config.Serial Config 4:151*);

- Referência enviada pelo controlador (*Port Config.Serial Config 3:18*)[12].

A referência para ser escrita no controlador ((*Port Config.Serial Config 4:151*)) tem o número 4, porque trata-se de uma *holding register*, que permite ler e escrever na memória, enquanto que para o número 3 trata-se de uma entrada de registos e só permite leitura de dados e não escrita. A referência enviada pelo controlador será igual à referência para ser escrita, este item foi adicionado simplesmente para confirmar que valor é que o controlador tem como referência.

Interface cliente OPC. No cliente OPC, *software* Scilab®, foi feita uma interface GUI através da função *guibuilder* na linha de comando. Na busca dos servidores OPC disponíveis no computador, foi encontrado o servidor OPC da Matrikon com o nome de *Matrikon.OPC.Modbus*. Após a criação do grupo foram adicionados os cinco itens OPC. Na figura 3.31 está representado o código do editor do Scilab® com o nome do servidor, do grupo e dos itens OPC. O nome do grupo e dos itens do cliente OPC serão os mesmos do OPC servidor criados no *software* da Matrikon.

```
1  opc_server_name = 'Matrikon.OPC.Modbus'
2  opc_group_name = 'Group0'
3  item(1) = 'Port Config.Serial Config.3:18'
4  item(2) = 'Port Config.Serial Config.3:9'
5  item(3) = 'Port Config.Serial Config.3:10'
6  item(4) = 'Port Config.Serial Config.4:151'
7  item(5) = 'Port Config.Serial Config.3:8'
```

Figura 3.31: Itens do cliente OPC.

A figura 3.32 mostra as funções do cliente OPC em Scilab® já mencionadas na secção 3.1.4. E como se pode ver nesta figura, primeiro é feita a conexão com o servidor OPC, em seguida são procurados os itens criados pelo servidor, adiciona-se o grupo e por último adiciona-se os itens OPC acima referidos.

Na figura 3.33, foi criado com GUI uma função *iniciar* que representará um botão na interface em que clicando nele permite ler três itens, referência (referência enviada pelo controlador), saída (velocidade dos ventiladores) e acção de controlo. O tempo t_{max} representa o tempo de ensaio do processo e após este tempo, a comunicação OPC será desconectada.

[12]Para além do servidor Modbus (controlador) enviar o seu endereço e o código de função também envia para o cliente Modbus qual o vslor escrito.

```
1
2   found_server = opc_server_browse()
3     opc_connect(opc_server_name)
4     opc_item_browse()
5     opc_add_group(opc_group_name)
6     opc_add_item(item,item_num)
```

Figura 3.32: Principais funções do Scilab OPC cliente.

```
1   function iniciar()
2   //Write your callback for  obj2  here
3
4   for k = 1:length(t)
5       item_read_value = opc_item_read(item_num,flag);
6       v(k) = item_read_value(1);
7       z(k) = item_read_value(2);
8       w(k) = item_read_value(3);
9       plot2d(t(1:k),[v(1:k) z(1:k) w(1:k)],[1,2,3],
10      leg="Referência@Saída@Acção de Controlo");
11      sleep(twaitms);
12
13  if (k == tmax) then
14          opc_disconnect()
15      end
16  end
17  endfunction
```

Figura 3.33: Código da leitura das variáveis do processo.

O código da figura 3.34 representa uma outra função criada pela GUI que tem como nome *referencia* sendo o botão que permite definir a referência do processo (ventiladores). A *flag* neste código e no programa todo para a interface foi definida como um inteiro porque os dados lidos e escritos no PLC cliente Modbus têm o formato de números inteiros.

```
1  function referencia()
2  //Write your callback for  obj3  here
3  setpoint=evstr(x_dialog('Por favor insere o SETPOINT: ','10'))
4      if tmax==[] then
5          msg=_("ERORR: Nenhum número inserido. ");
6          messagebox(msg, "ERROR", "error");
7          error(msg);
8          return;
9      end
10     item_value= setpoint
11     opc_item_write(item_index,item_value,flag)
12 endfunction
```

Figura 3.34: Código de escrita da referência do processo.

A figura 3.35 representa o código implementado no editor do Scilab®, o resultado deste código será a interface apresentada no capítulo 4.

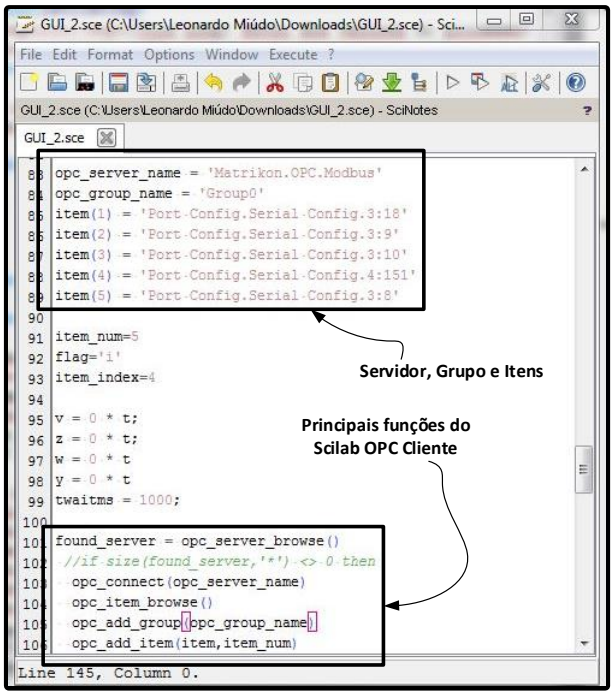


Figura 3.35: O editor e o código da interface.

O PLC cliente Modbus ou supervisor estará constantemente a comunicar de modo a permitir a leitura ou escrita de dados com o PLC servidor Modbus, ou controlador através da rede Modbus e também com o servidor OPC (MatrikonOPC) através do padrão OPC®. Devido a esta permanente comunicação foi necessário definir tempos de leitura e escrita para ambas as comunicações, de maneira que não haja conflitos durante a comunicação de dados, que no PLC cliente resultam em leitura de valores incorrectos.

Assim sendo, na figura 3.36 representa-se o diagrama temporal das duas comunicações Modbus e OPC. Durante a implementação do código no PLC supervisor foi definido um temporizador do tipo liga - desliga que em estado ligado permitirá a escrita em Modbus e no estado desligado a leitura Modbus.

No diagrama temporal a escrita Modbus ocorre quando o temporizador passa para o estado ligado, a escrita de valores para o escravo será feita apenas num intervalo de t_{11} deixando o restante tempo para a comunicação OPC. Quando o temporizador passa para o estado desligado no intervalo de t_{12}, é feita a leitura Modbus. O tempo t_{22} de leitura e t_{32} de escrita Modbus é destinado para a comunicação OPC.

A comunicação OPC apresenta um tempo superior em relação à comunicação Modbus porque se trata da comunicação utilizada para a apresentação do resultado do processo e

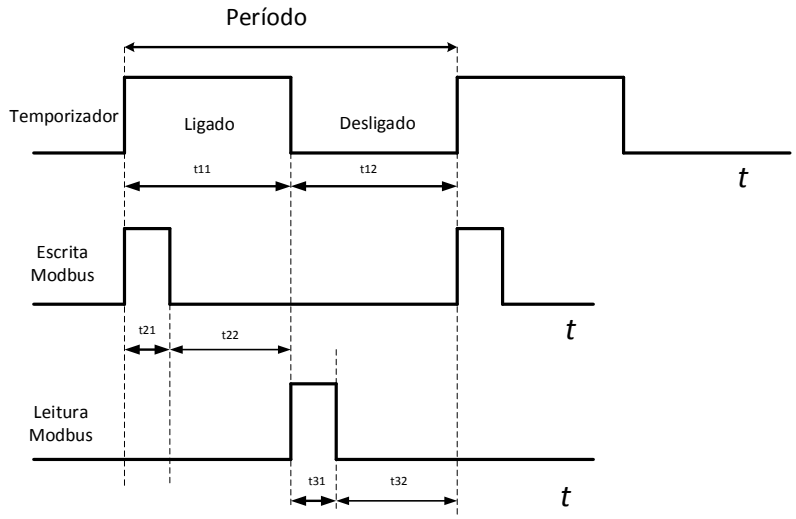


Figura 3.36: Diagrama temporal da comunicação Modbus e do padrão OPC.

também porque não é possível controlar-se a leitura do servidor OPC, ou seja, está constantemente a fazer a leitura ao PLC cliente.

Tendo-se apresentado a arquitectura, a tecnologia utilizada e a implementação do protocolo Modbus, os resultados dos testes efectuados a rede e a apresentação gráfica na interface criada no Scilab® serão apresentados no capítulo 4, resultados experimentais. Nas próximas secções será apresentada a arquitectura, tecnologia e implementação para o protocolo CAN.

3.2 Protocolo CAN

Nesta secção descrita feita a implementação do protocolo CAN, com base nas placas Arduino e suas *shields* CAN. As *shields* são constituidas pou um controlador CAN, MCP2515, e um transceptor MCP2551 que permite a comunicação num barramento CAN.

O protocolo será testado em um processo de controlo de nível da água (PCT9 de Armfield) com um controlador PI com *anti windup* programado num Arduino Mega 2560. A implementação deste protocolo seguirá a mesma filosofia da implementação do protocolo Modbus tendo um processo a controlar, um controlador e um supervisor mas com modelo de comunicação produtor-consumidor trocando dados sobre o protocolo CAN.

Os Arduino controlador e supervisor foram programados no *software* de desenvolvimento integrado Arduino IDE (*Integrated Development Environment*) . Na busca de uma

solução para a representação gráfica dos sinais do processo, utilizou-se o *open source software* Scilab®.

3.2.1 Arquitectura

A arquitectura para os testes com o protocolo CAN obedeceram à representada na figura 3.37. Esta arquitectura basicamente é constituida por um supervisor (placa Arduino UNO e uma *shield* CAN) com interface *open source* Scilab® e uma placa Arduino Mega acoplada a uma *shield* CAN que será o controlador do processo.

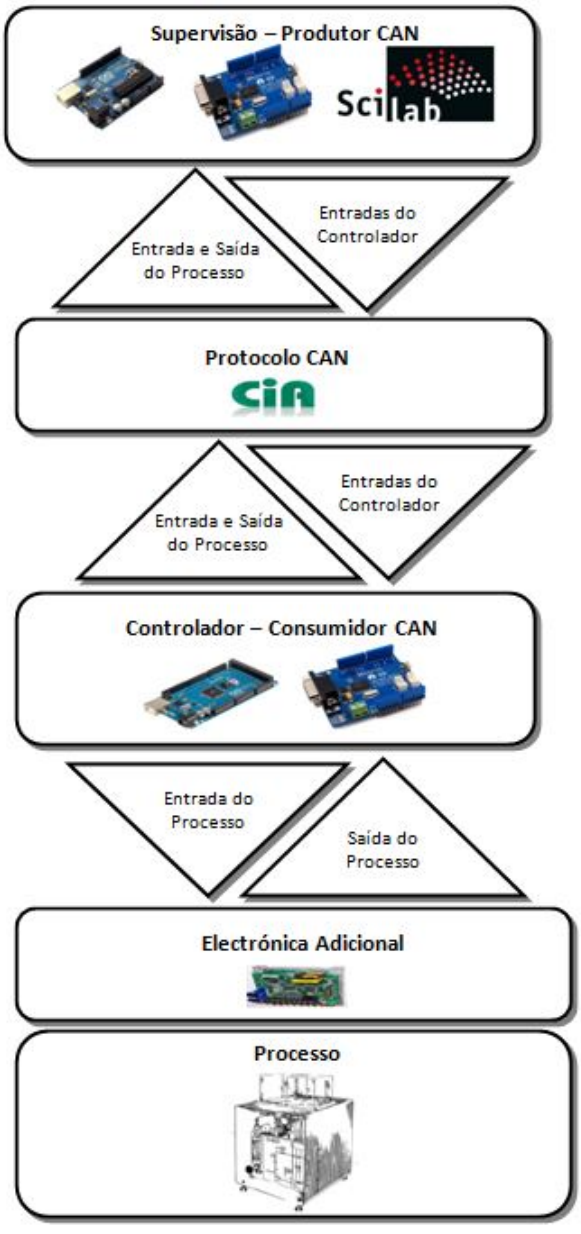


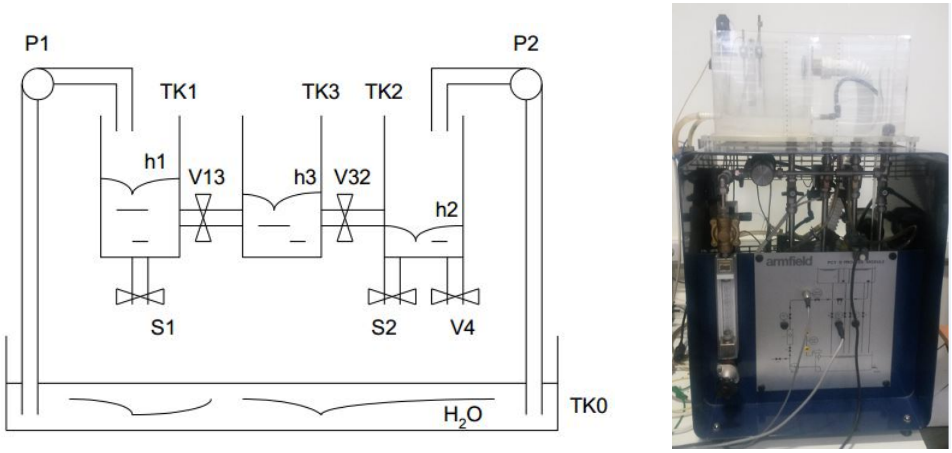
Figura 3.37: Arquitectura do sistema testado com o protocolo CAN.

As *shields* CAN já possuem resistores terminais e estão ligadas a um barramento com cabo par trançado. A comunicação entre as *shields* e as placas arduino é por SPI (*Serial Peripherical Interface*).

O supervisor definirá a referência, os ganhos do controlador K_p, K_i e o tempo de amostragem T_s. De retorno receberá a partir do controlador as variáveis do processo: saída do processo e acção de controlo. A placa electrónica adicional na figura da arquitectura faz parte do processo e é constituida por um filtro passa-baixo para filtrar o sinal pulsado da acção de controlo vindo do arduino controlador e um driver de potência EM-174 DC-Motor Driver 12/24Vdc 8A que estará directamente ligado à bomba de água.

3.2.2 Análise do Processo

O processo PCT9 da Armfield é constituido por três tanques ligados em série (cascata) e um reservatório onde é armazenada água. Na figura 3.38 está representado o processo PCT9, figura 3.38(a) o esquema do processo e na figura 3.38(b) a imagem real do processo. Os três tanques (TK1, TK3 e TK2) estão interligados, existindo válvulas (V13 e V32) de comando de fluxo. As válvulas S1, S2 e S4 permitem o escoamento da água. Existe duas bombas de água P1 e P2. No trabalho apenas esteve acessível a bomba do tanque 1 (P1) como actuador do processo, para encher o tanque 1, e o sensor de nível da água no tanque 1 (h1) como sensor do processo.



(a) Esquema do processo.

(b) Imagem real do processo.

Figura 3.38: Processo PCT9 (modificado) da Armfield.

Fazendo parte do processo, existe uma placa electrónica adicional (filtro passa-baixo) para se filtrar passa-baixo o sinal de saída do modulador PWM da placa Arduino que irá

actuar no driver de potência da bomba de água. Este valor de tensão filtrado será aplicado a um driver de potência EM-174 DC-Motor Driver 12/24Vdc 8A que por sua vez actua directamente na bomba de água. O valor do sensor é lido pela placa Arduino controlador. O sensor só permite leitura de valores de até 3V. A tensão eléctrica aplicada ao driver de potência (actuador) será limitada para limitar o caudal da bomba de água a 2.5V.

3.2.3 Análise do Consumidor CAN - Controlador do Processo

O controlador é constituido por uma *shield* CAN acoplada a uma placa Arduino Mega 2560 (vide figura 3.39). A *shield* comunica com qualquer arduino através da interface periférica série SPI a partir dos pinos 10, 11 e 12. Existe incompatibilidade devido o Arduino Mega 2560 não usar estes pinos para comunicação série.

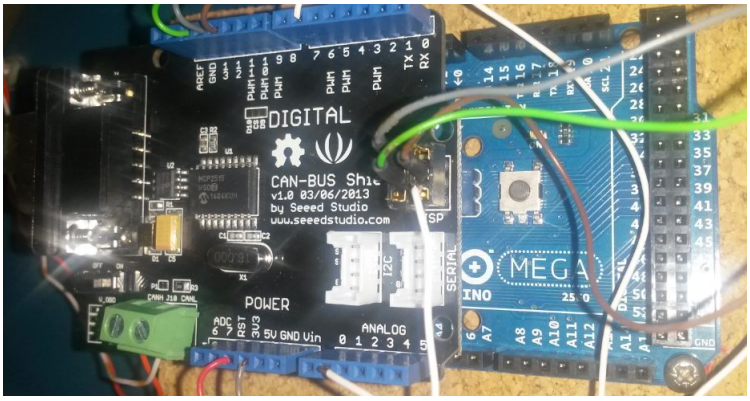


Figura 3.39: Consumidor CAN - controlador do processo.

Esta incompatibilidade deve-se ao controlador CAN (MCP2515) da *shield* estar ligado aos pinos 10, 11 e 12 e não aos pinos de comunicação série da placa e os pinos 10, 11 e 12 no arduino Mega não permitem comunicação série com microcontrolador.

Tal como já se fez referência na descrição da implementação do controlador no protocolo Modbus, para o protocolo CAN, será utilizado um controlador PI com *anti windup*.

Controlador PI *anti-windup*. Com este controlador, pretende-se limitar a acção de controlo em máximo e mínimo, e para isso implementou-se um controlador PI com *anti-windup*, tendo como referências as publicações de Astrom e Visioli [18] e [42].

Quando a variável de controlo atinge o limite máximo (ou mínimo) do actuador ocorre a saturação do sinal de controlo. Este facto faz com que o anel de realimentação seja de certa forma quebrado, pois o actuador permanecerá no seu limite máximo (ou mínimo)

independentemente da saída do processo. Entretanto, se um controlador com a acção integral é utilizado, o erro continuará a ser integrado e o termo integral tende a tornar-se muito grande. Do inglês, diz-se que o termo integral "*winds-up*". Neste caso, para que o controlador volte a trabalhar na região linear (saia da saturação) é necessário que o termo integral recupere [52].

Portanto, dever-se-á esperar que o sinal de erro troque de sinal e, por um longo período de tempo, aplicar na entrada do controlador, um sinal de erro oposto. A consequência disto é que a resposta transitória do sistema tenderá a ficar lenta e oscilatória, característica esta extremamente indesejável em um processo industrial [52].

Existem várias maneiras de se evitar o *wind-up* da acção integral. A ideia básica é impedir que o integrador continue a integrar quando a saturação ocorre. A maneira mais conveniente é a *"back calculation"* elucidada na figura 3.40. Quando a saída do actuador satura, o termo integral é recalculado de forma que seu valor permaneça no valor limite do actuador. É vantajoso fazer essa correção não instantâneamente, mas dinamicamente com uma constante de tempo T_t.

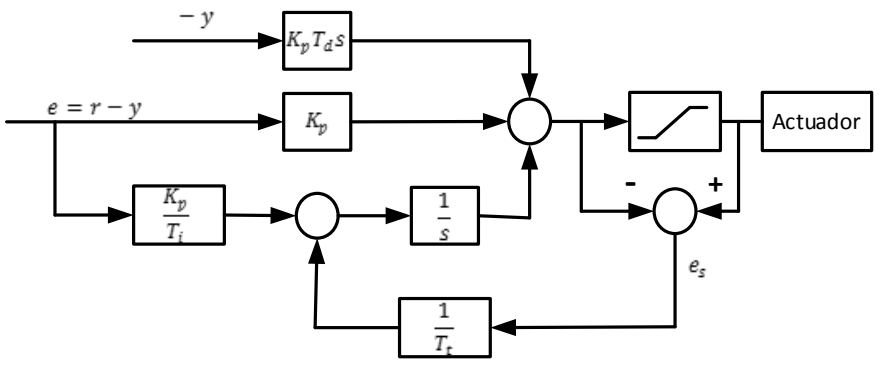


Figura 3.40: Controlador PID com *anti windup* [18].

A figura 3.40 mostra o diagrama de blocos de um controlador PID com *anti windup*, em que a acção de controlo é saturada à entrada do actuador. O sistema apresenta um anel de realimentação adicional ao PID normal. A diferença entre o valor de entrada e de saída da acção de controlo após a saturação constitui um erro e_s que é realimentado à entrada do integrador, com um ganho $1/T_t = K_t$. Quando não há saturação, o erro e_s é igual a zero e, portanto, este anel não tem nenhum efeito, quando o controlador está operando linearmente, ou seja, quando a sua saída não está saturada [42].

Quando ocorre a saturação e_s será diferente de zero e o sinal aplicado na entrada do integrador não mais será eK_p/T_i mas:

$$\frac{1}{T_t}e_s + \frac{K_p}{T_i}e \tag{3.10}$$

sendo que em regime permanente, teremos que:

$$e_s = -\frac{K_p T_t}{T_i}e \tag{3.11}$$

ou seja, a entrada do integrador será igual a zero prevenindo que o mesmo aumente em demasia. O tempo para que a entrada do integrador chegue a zero é determinado pelo ganho $1/T_t$ onde T_t pode ser interpretado como a constante de tempo que determina o quão rápido a entrada do integrador será levada a zero.

A fim de facilitar os cálculos, a regra de projecto para o tempo T_t é sugerida por Astrom e Hagglund [42] como:

$$T_t = \sqrt{T_i T_d} \tag{3.12}$$

A principal desvantagem desta fórmula é que ela não pode ser adoptada no controlo PI onde T_d é igual a 0. Em alternativa Bohn e Atherton [42] sugerem que:

$$T_t = T_i \tag{3.13}$$

Os parâmetros do controlador PI com *anti - windup* são: b_i ganho integral e a_o ganho de *anti - windup* e têm as seguintes expressões:

$$b_i = \frac{K_p T_s}{T_i} \tag{3.14}$$

$$a_o = \frac{T_s}{T_t} \tag{3.15}$$

onde T_s é o tempo de amostragem.

Nesta dissertação utilizou-se a técnica de *back calculation* para se evitar o *windup*. O projecto dos ganhos do controlador e a análise de desempenho do controlador foram realizados em no Matlab®. O ponto de funcionamento do processo, foi estabelecido no nível de água 0.3, em valores normalizados na gama [0 ; 1] Foram obtidos dados em anel

fechado, com um controlador PI, para este ponto de funcionamento. Foi colocado um filtro passa baixo na saída do controlador de forma a prevenir variações demasiado rápidas na entrada do processo. Os valores dos ganhos do controlador PI, obtidos para o ponto de funcionamento (sendo que o valor do pólo do filtro passa baixo foi de 0.8, no plano z):

$$K_p = 30 \; ; \; K_i = 35$$

A referência do processo (nível da água), os ganhos do controlador (K_p e K_i) e o tempo de amostragem T_s foram definidos pelo supervisor, produtor CAN, e enviados pelo barramento CAN para o consumidor CAN (controlador), e de retorno, o controlador enviará os dados de saída do processo e a acção de controlo.

Na implementação da rede CAN na secção 3.2.6, apresenta-se o código do controlador PI com *anti windup* implementado com placas Arduino.

3.2.4 Análise do Produtor CAN - Supervisor do Processo

O supervisor foi implementado por um Arduino UNO e uma *shiled* CAN, figura (vide 3.41). Diferente do arduino Mega 2560, o arduino UNO é totalmente compatível com a *shield* CAN, isto porque o arduino UNO usa os terminais 10, 11 e 12 para comunicação série. No supervisor, é possível definir-se a referência do processo, os ganhos do controlador K_p, K_i e o tempo de amostragem T_s. Na figura 3.41 está representado a imagem da placa Arduino com supervisor.

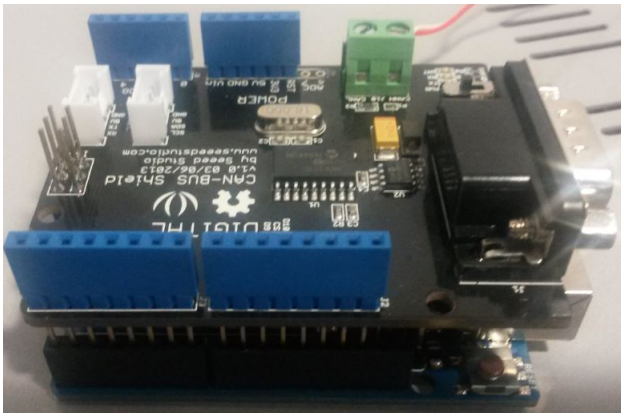


Figura 3.41: Supervisor do processo - Produtor CAN.

Como se está perante um *open source* para a implementação de uma rede CAN, Arduino, o Scilab® foi novamente adoptado como o *software* para a criação de uma interface

e supervisão do processo. O Scilab® possui uma *toolbox*, *Serial Communication Toolbox* permitindo uma comunicação série. Entretanto, acederá a porta série COM do computador com ligação ao arduino e terá a função de ler os dados que se encontram nesta porta. A interface foi criada no GUI, a figura 3.42 ilustra a arquitectura implementada.

As funções necessárias de acordo com a sintaxe do Scilab® (*Serial Commmunication Toolbox*), são as seguintes:

- *openserial(p,"mode")* - comando que realiza a abertura da porta série, onde p indica o número da porta e *mode* indica o *baudrate* da porta.
- *readserial(x)* - comando para ler os caracteres da porta série, x indica a variável atribuída à porta.
- *closeserial(x)* - comando para fechar a porta série, x indica a variável atribuída à porta.

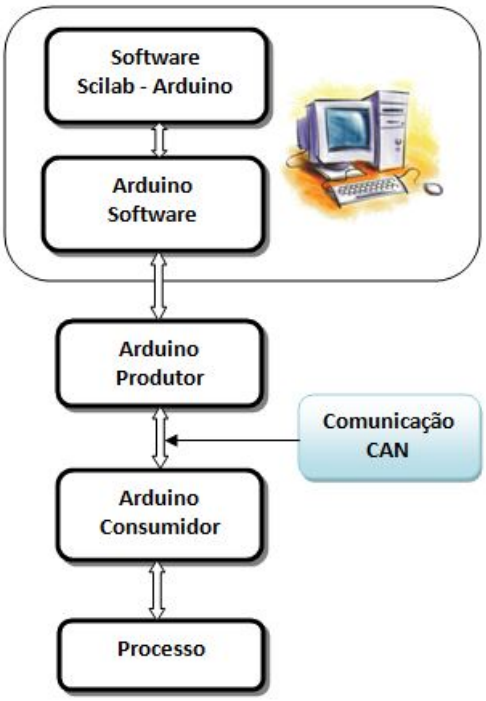


Figura 3.42: Arquitectura Scilab - Arduino.

3.2.5 Funções na biblioteca da *shield* CAN

O CAN é um barramento comum na indústria porque permite uma velocidade de comunicação com média e alta fiabilidade a longa distância. É comumente encontrado em

máquinas modernas e como barramento de diagnóstico de falhas e avarias nos automóveis. Na *shield* tem-se um controlador CAN MCP2515, com interface SPI para ligar a um microcontrolador (arduino ou outro) e um transceptor CAN, MCP2551, que permite comunicar em barramento CAN. O controlador CAN MCP2515 tem as seguintes características:

- Implementa CAN V2.0B até 1 Mbps, interface SPI até 10MHz.
- Trama de dados e remota com identificador de 11 *bits* (*standard* ou base) e 29 *bits* (estendido).
- Dois *buffers* receptores com prioridade no armazenamento das mensagens.

O MCP2551 é um dispositivo CAN de alta velocidade, que funciona como interface entre um controlador de protocolo CAN e o barramento físico. O MCP2551 possui capacidade de transmissão e recepção diferencial para o controlador do protocolo CAN, e é totalmente compatível com a norma ISO-11898. Ele opera em velocidade de até 1 Mbps. A biblioteca do CAN bus *shield* actualmente não suporta trama remota, mas apenas trama de dados e as suas principais funções serão aqui retratadas.

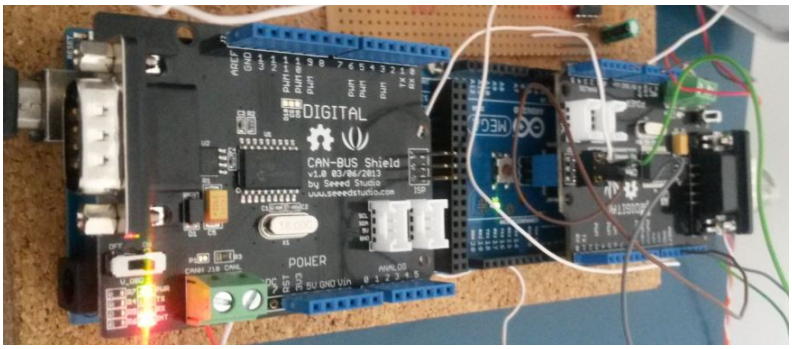


Figura 3.43: Imagem real do produtor e consumidor CAN.

A função *CAN.begin* inicializa o controlador CAN e define a velocidade de comunicação, *baudrate*. Para esta função o retorno será *CAN_OK* inicialização completa ou *CAN_FAILINIT*, inicialização falhada.

Existem no controlador CAN, 2 registos de máscara e 5 registos de filtro. Estes são normalmente utilizados em redes com muitos nós. Para definir estes registos de máscaras e filtros são utilizadas as seguintes funções:

CAN.init_Mask (*unsigned char num, unsigned char ext, unsigned char ulData*);

CAN.init_Filt (*unsigned char num, unsigned char ext, unsigned char ulData*).

"num": representa que registo a utilizar. Podem definir-se 0 e 1 para máscaras e de 0 a 5 para filtros.

"ext": representa o estado da trama. 0 (zero) significa que é uma máscara ou filtro para uma trama com identificador de 11 *bits*. 1 significa que é para uma trama estendida com identificador de 29 *bits*.

"ulData" representa o teor da máscara e do filtro.

A função para enviar dados no barramento é *CAN.sendMsgBuf*(*INT8U id, INT8U ext, INT8U len, data_buf)*, em que:

"id" representa de onde os dados vêm.

"ext" representa o estado da trama. '0' significa trama standard ou base. '1' significa trama estendida.

"len" representa o tamanho da trama.

"data_buf" é o conteúdo da mensagem.

A função *CAN.checkReceive* serve para verificar se a mensagem foi recebida. O MCP2515 pode operar em 2 modos, *software* verificando se a trama foi recebida ou utilizando pinos adicionais para sinalizar que uma trama foi recebida ou com transmissão concluída. A função retornará 1 se chegar uma frame e 0 se não chegar.

A função *CAN.readMsgBuf*(*unsigned char len, unsigned char buf*) é utilizada para receber os dados no nó receptor. Em condições de definição de máscaras e filtros, esta função só pode obter tramas que atendam aos requisitos das máscaras e filtros:

"len" representa o tamanho dos dados.

"buf" é onde estão armazenados os dados.

Quando alguns dados chegam ao receptor, usa-se a função *CAN.getCanId* para obter o identificador do nó emissor.

A função *CAN.checkError* tem o propósito de verificar erros no controlador CAN.

3.2.6 Implementação da rede CAN no controlo do processo

Antes de se fazer o controlo e a monitorização do processo utilizando a rede CAN, será analisado o barramento CAN fazendo testes da trama de dados que se está a enviar. O resultado destes testes são visualizados no osciloscópio com o objectivo de verificar se realmente a trama cumpre com todos os campos analisados no capítulo 2. No esquema de um programa em linguagem C para Arduino existem 3 secções: a primeira consiste na definição das variáveis globais, a segunda chamada de *setup* é onde se define as condiçoes iniciais para o programa e por último a secção *loop* é o ciclo com o programa principal.

Na primeira secção, define-se todas as variáveis utilizadas ao longo do programa, e na segunda secção *setup* define-se como condições iniciais o *baudrate* das comunicações SPI, CAN, as máscaras e os filtros da mensagem (vide figura 3.44).

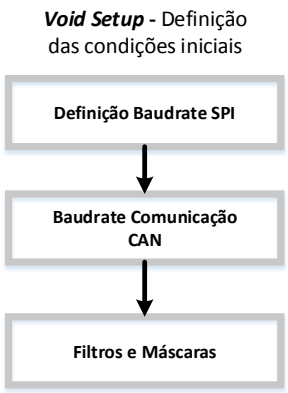


Figura 3.44: Diagrama de blocos *setup*.

O controlo da configuração de registos do tempo de *bit* do controlador CAN de ambas as *shields* para a interface do barramento CAN foram configurados com um *clock* de 16 MHz, para um *baudrate* CAN de 20 kbps foi definido *baud rate prescaler* CFG1 de 0x0F.

Inicialmente, pretende-se fazer testes à rede de maneira a obter-se alguns resultados relacionados com o barramento, ou seja, fazer análise da trama de dados. Estes resultados se encontram no capítulo 4. Os testes foram feitos com um *baudrate* de 20 kbps e o comprimento do cabo par trançado da ligação entre o produtor CAN e o consumidor CAN foi de 100 cm.

Para o primeiro teste, considerou-se uma trama de dados em que são enviados perma-

nentemente dados ao barramento, com um identificador de 0x400 hexadecimal, 1024 em decimal (10000000000 em binário). A escolha deste identificador foi simplesmente por representar metade da quantidade de identificadores que a versão pode ter, 2048 em decimal, (vide figura 3.45).

```
1  // Versão Standard
2   unsigned char msg[0] = {};
3   CAN.sendMsgBuf(0x400, 0, 0, msg);
```

Figura 3.45: Trama versão *standard*.

Em seguida fez-se um teste com o mesmo identificador mas considerando uma versão estendida resultando num identificador 0x00000000000000000010000000000 em binário e o DLC foi novamente definido a 0 (figura 3.46).

```
1  // Versão extendida
2   unsigned char msg[0] = {};
3   CAN.sendMsgBuf(0x400, 1, 0, msg);
```

Figura 3.46: Trama versão estendida.

Trata-se de uma versão *standard* por apresentar 11 *bits* como se pode ver na figura 3.45 o valor 0 (zero) após o identificador da mensagem 0x400, o DLC também é igual a 0 (zero) o que significa que no campo de dados não se encontra qualquer *byte*, ou seja, o campo de dados está vazio (tabela 3.6).

Tabela 3.6: Valores aceitáveis no campo DLC em 0 *byte* [23].

Bytes	DLC3	DLC2	DLC1	DLC0
0	0	0	0	0

O objectivo de não enviar qualquer dado (DLC = 0 *bytes*) na versão *standard* e estendida é simplesmente para não se ter um sinal muito extenso que não fosse possível observa-lo completamente no osciloscópio, numa escala aceitável para ser interpretado. A única diferença entre os dois códigos, trama *standard* e trama estendida, resulta que para o primeiro caso depois do identificador (0x400 hexadecimal) tem-se 0 (zero), enquanto que para definir um identificador de 29 *bits* tem-se 1.

Devido ao facto de não ser possível ter-se o sinal completo da versão estendida no oscilocópio numa escala que fosse possível analisar, para os seguintes testes considerou-se somente a versão *standard* (figura 3.47), e o passo seguinte foi analisar a diferença que

resultaria considerando que o transmissor envia a trama ou mensagem sem receptor e com receptor.

```
1  // Versão Standard
2   unsigned char msg[0] = {};
3   CAN.sendMsgBuf(0x400, 0, 0, msg);
```

Figura 3.47: Trama versão *standard* para o teste de ACK *slot*.

O objectivo deste teste é de verificar o campo de confirmação ACK (o *bit ACK-Slot* tem de ser dominante) que está relacionado com a confirmação do nó receptor depois de ter recebido uma mensagem, como se pode ver na figura 3.48.

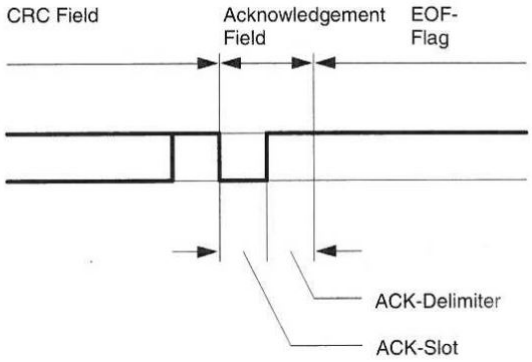


Figura 3.48: Formato do campo do confirmação [2].

Considerando a versão *standard*, em seguida fez-se o teste de verificação dos níveis de tensão do CAN-H e do CAN-L de modo a se comprovar se realmente a rede apresentava os níveis estabelecidos pelo protocolo, e comparar estes mesmos níveis com os termos dominantes e recessivos. Portanto, espera-se que o CAN-H tenha uma tensão de 3,5V e o CAN-L de 1,5V quando se está presente a um *bit* dominante, e ambos em 2,5V na presença de um *bit* recessivo.

E por último foi realizado o teste de verificação das prioridades considerando uma trama de dados com alta prioridade, figura 3.49, em que se tem um identificador 0x0000h (000000000000 em binário) totalmente dominante e outra trama de baixa prioridade, figura 3.50 com identificador 0x7FFh (11111111111 em binário) totalmente recessivo. E é de se esperar que estas tramas apresentem muitos *bits stuffing*.

Após os testes é chegado a altura da implemetação do barramento CAN para o controlo e monitorização do processo de controlo do nível de água. Para o supervisor atribuiu-

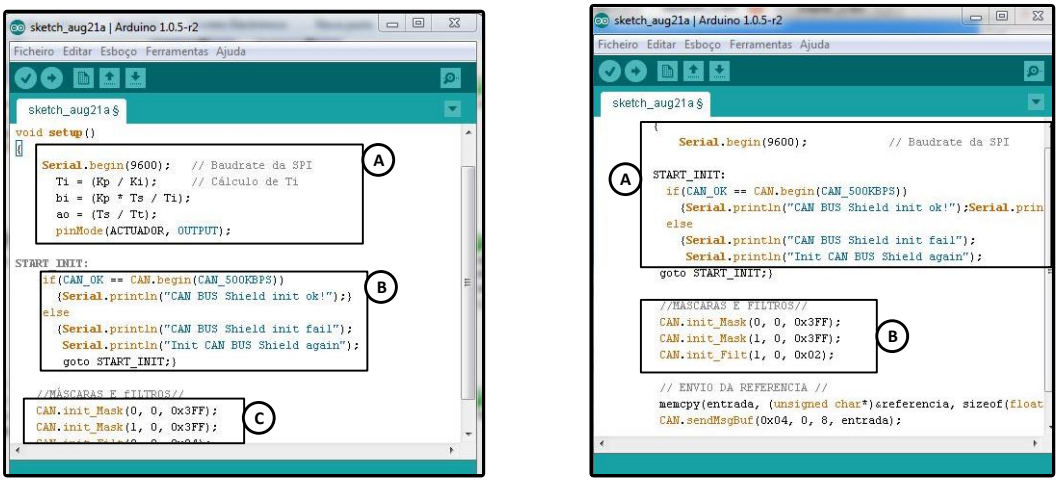
```
// Trama com alta prioridade, identificador 0x0000
unsigned char msg[0] = {};
CAN.sendMsgBuf(0x0000, 0, 0, msg);
```

Figura 3.49: Trama com alta prioridade.

```
// Trama com baixa prioridade, identificador 0x7FF
unsigned char msg[0] = {};
CAN.sendMsgBuf(0x7FF, 0, 0, msg);
```

Figura 3.50: Trama com baixa prioridade.

se o identificador 0x00 (alta prioridade) e para o controlador 0x01. Atribuiu-se a alta prioridade ao supervisor por ser o elemento dentro da arquitectura que permite a atribuição da referência do processo, dos valores dos ganhos e do tempo de amostragem para o controlador. Quanto aos *baudrates* estabeleceu-se o valor de 9600 bps para SPI e 20 kbps para a comunicação CAN. Na figura 3.51 é apresentado o *software* de desenvolvimento integrado para a programação das placas Arduino com os códigos para o controlador e para o supervisor.



(a) Controlador - Consumidor.

(b) Supervisor - Produtor.

Figura 3.51: *Void Setup.*

Em *setup* do Arduino controlador foram definidas as condições iniciais, *baudrate* das comunicações, as máscaras e os filtros da mensagem. Para o supervisor além de se ter definido as condições iniciais já descritas é enviado a referência do processo, os ganhos do controlador (K_p, K_p) e o tempo de amostragem T_s com um intervalo de 200 ms (figura 3.52).

O envio da referência, dos ganhos e do tempo de amostragem T_s pelo supervisor no *setup* deve-se ao facto destes valores serem enviados uma única vez, não tendo necessidade de estar dentro do *loop*, e também para diminuir o tempo de processamento do microcontrolador. Sempre que se fizer uma alteração destes valores o programa deve ser "carregado"para o arduino supervisor e serão enviados estes novos valores.

No protocolo CAN o tamanho máximo da mensagem é de 8 *bytes* e os dados que estão sendo enviados (referência, ganhos e o tempo de amostragem) pelo produtor ou supervisor são constituidos por 4 *bytes* onde tem-se uma unidade 1 *byte*, a vírgula 1 *byte* e um centésimo 2 *bytes*. Poderia-se enviar as mensagens desta forma mas o controlador não saberia distinguir um valor do outro, ou seja, não conseguiria distinguir o valor da referência do tempo de amostragem. Para resolver este problema inseriu-se um identificador no sétimo *byte* da mensagem. Estes identificadores são: e para a referência, p para o ganho K_p, i para o ganho K_i e a para o tempo de amostragem T_s, figura 3.52.

```
1   // Envio da Referência //
2     memcpy(entrada, (unsigned char*)&referencia, sizeof(float));
3     entrada[7] = 'e'
4     CAN.sendMsgBuf(0x00, 0, 8, entrada);
5     delay(200);
6
7      // Envio do Ganho Kp //
8     memcpy(ganhoKp, (unsigned char*)&ganho1, sizeof(float));
9     ganhoKp[7] = 'p'
10    CAN.sendMsgBuf(0x00, 0, 8, ganho1);
11    delay(200);
12
13     // Envio do Ganho Ki //
14    memcpy(ganhoKi, (unsigned char*)&ganho2, sizeof(float));
15    ganhoKi[7] = 'i'
16    CAN.sendMsgBuf(0x00, 0, 8, ganhoKi);
17    delay(200);
18
19     // Envio do Tempo de amostragem Ts //
20    memcpy(tempoTs, (unsigned char*)&amostragem, sizeof(float));
21    tempoTs[7] = 'a'
22    CAN.sendMsgBuf(0x00, 0, 8, tempoTs);
23    delay(200);
```

Figura 3.52: *Setup* do Supervisor envio da referência, dos ganhos e do tempo de amostragem.

A figura 3.53 apresenta o código do controlador para fazer a leitura da referência, dos ganhos e do tempo de amostragem.

No *loop* do controlador primeiro foi lido o valor da referêcia, dos ganhos e do tempo T_s enviado pelo supervisor em seguida é realizado o cálculo das variáveis do controlador

```
1
2  CAN.readMsgBuf(&len, buf);
3
4     if (buf[7] == 'e')
5       {Serial.print("Referencia:");
6       memcpy((unsigned char*)&abc, buf,sizeof(float));
7       Serial.println(abc);
8
9     else if (buf[7] == 'p')
10      {Serial.print("GanhoKp:");
11      memcpy((unsigned char*)&efg, buf, sizeof(float));
12      Serial.println(efg);}
13
14    else if (buf[7] == 'i')
15      {Serial.print("GanhoKi:")
16      memcpy((unsigned char*)&hij, buf, sizeof(float));
17      Serial.println(hij);}
18
19    else if (buf[7] == 'a')
20      {Serial.print("Tamostragem:");
21      memcpy((unsigned char*)&klm, buf, sizeof(float));
22      Serial.println(klm);}
```

Figura 3.53: *Loop* do Controlador leitura dos valores iniciais do processo.

b_i e a_o e por último a implementação do controlador PI com *anti windup*. Como já foi referido, os cálculos para estes ganhos foram feitos para a referência do processo em 0.3.

Os cálculos foram todos feitos com o objectivo de limitar os valores na escala [0;1]. O arduino UNO e Mega apenas lêm valores até 5V, ou seja, o arduino só permite valores de entrada até 5 V que corresponde a 1024. Com a função *analogRead* do arduino o valor lido pelo sensor será dividido por 614 (correspondente a 3V, tal como já foi descrito na análise do processo) para ter-se valores na escala de [0 ; 1].

Está-se perante um controlador PI com *anti windup*, por isso a parte derivativa do controlador será igual a 0 (zero) o que implica que $T_i = T_t$.

O controlador PI com *anti windup* implementado, figura 3.54, é do tipo *back calculation*, a expressão $U[2]$ e a equação 3.16 representam o filtro à saída do actuador em que *pole* representa o polo do filtro com um valor de 0.8, enquanto que $I[2]$ representa a acção integral.

$$u(t) = pole \cdot u(t) + u_o(t) \cdot (1 - pole) \tag{3.16}$$

A implementação de um controlador do tipo PI com *anti windup* deve-se ao facto de ter uma saturação à saída do controlador. Este valor, a acção de controlo, como se encontra no intervalo de [0 ; 1] foi necessário multiplica-lo por 127 que corresponde a 2.5V

```
                    // CONTROLADOR PI ANTI WIND-UP //
LEITURA = analogRead(SENSOR);
   VALOR = LEITURA/614; // Ganho do sensor 1/3, 614 corresponde a 3 volts

    if (VALOR < 0) { VALOR = 0; }
    if (VALOR > 1) { VALOR = 1; }

   ERRO = (REFER - VALOR);  // Sinal do erro
   P = (Kp*ERRO);
   V[2] = (P + I[2] + D[2]);
   Uo[2] = V[2];  // Acção de controlo antes de ser saturada

    if (V[2] < 0) { Uo[2] = 0; }
    if (V[2] > 1) { Uo[2] = 1; }

   // Acção de controlo saturada e filtrada
   U[2] = (pole*U[1] +((1-pole)*Uo[2]));
   I[2] = (I[1] + bi*ERRO + ao*(U[2] - V[2])); //U-V (saturada - filtrada)

   DUTY = U[2];
    if (DUTY < 0) { DUTY = 0; }
    if (DUTY > 1) { DUTY = 1; }

   DUTYCICLE = DUTY*127;
   analogWrite(ACTUADOR, DUTYCICLE);
```

Figura 3.54: *Loop* Controlador PID *anti wind-up*.

(tensão máxima a entrada do actuador) para que seja escrito pelo arduino. A figura 3.54 representa o código do controlador PI com *anti windup*.

Após o cálculo da acção de controlo a aplicar ao processo, os dados de saída do processo e da acção de controlo são enviados do controlador - consumidor CAN ao supervisor - produtor CAN. A mensagem só é enviada se t_{aux} fôr igual a t_{envio}. O t_{envio} foi definido como 4 segundos e o tempo de espera (*delay*) do ciclo (*loop*) foi é de 500 s, o que significa que a mensagem só é enviada de 2 em 2 segundos.

Cada vez que $t_{aux} = t_{envio}$ são enviadas duas mensagens com os valores de entrada e saída do processo, a acção de controlo e o sinal de saída, todos eles com o identificador do controlador 0x01 (figura 3.55).

No supervisor, depois de se enviar a referência do processo no *setup*, entra-se para o *loop* onde se estará constantemente a receber os dados do processo vindo do controlador, figura 3.56.

```
                // COMUNICAÇÃO PROTOCOLO CAN //
 taux = taux + 1;
  if (taux == tenvio)
        {taux = 0;
           // Acção de Controlo
       memcpy(control, (unsigned char*)&DUTY, sizeof(float));
       control[7] = 'c';
       CAN.sendMsgBuf(0x01, 0, 8, control);
            // Sinal de saída
       memcpy(output, (unsigned char*)&VALOR, sizeof(float));
       output[7] = 'o';
       CAN.sendMsgBuf(0x01, 0, 8, output);
     }
```

Figura 3.55: *Loop* envio dos valores do processo.

```
CAN.readMsgBuf(&len, buf);

   if (buf[7] == 'c')
     {Serial.print("Accao_de_Controlo:")
     memcpy((unsigned char*)&def, buf,sizeof(float));
     Serial.println(def);}

   else if (buf[7] == 'o')
     {Serial.print("Saida:");
     memcpy((unsigned char*)&ghi, buf, sizeof(float));
     Serial.println(ghi);}
```

Figura 3.56: *Loop* do Supervisor leitura dos valores do processo.

Interface gráfica em Scilab®.

O *software* Scilab® foi novamente adoptado para a apresentação gráfica das variáveis do processo, acção de controlo e sinal de saída do sistema a controlar e foi implementado lendo a porta de configuração do arduino do supervisor tal como já foi elucidado na arquitectura da figura 3.42. Como se pode ver na figura 3.57, a porta foi acedida na COM3 com um *baudrate* do SPI de 9600 bps.

A figura 3.58 mostra o editor e o código da interface que foi implementado e que será apresentado nos resultados experimentais no capítulo 4. Em **A** é possível ver o código de

leitura da porta série.

```
serial_port=openserial(3,"9600")
disp("Porta Série Aberta");

closeserial(serial_port);
disp("Porta Série Fechada");
```

Figura 3.57: Abertura e o fecho da porta série por parte do Scilab®.

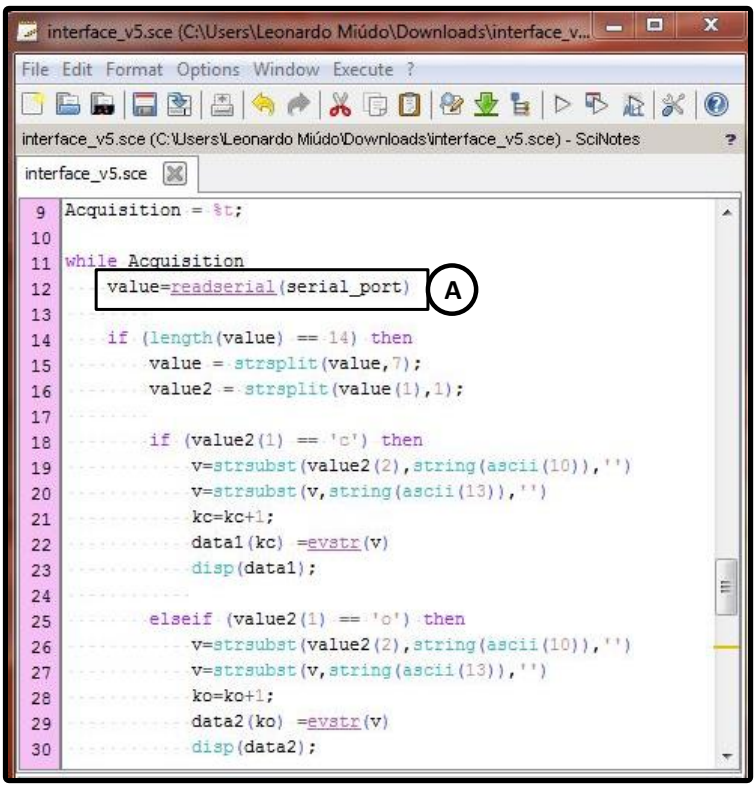


Figura 3.58: O editor e o código da interface CAN.

Capítulo 4

Resultados experimentais

Neste capítulo são apresentados os resultados dos testes efectuados no capítulo 3 e apresentando-se também a interface de monitorização e controlo dos sinais dos processos.

O capítulo está dividido em duas secções, na secção 4.1 apresentam-se os resultados para o protocolo Modbus. Nesta secção são apresentadas as tabelas de animação do PLC cliente - supervisor para os diferentes testes de comunicação, o resultado da configuração OPC, os níveis de tensão do barramento. A interface HMI criada no Scilab® OPC cliente também é descrita.

Na secção 4.2 são apresentados os resultados para o protocolo CAN analisando os campos que constituem uma trama de dados e os níveis de tensão do barramento. Os sinais do processo a controlar são apresentados em Scilab®.

Na última secção, secção 4.3 é apresentada uma breve comparação do protocolo Modbus *versus* protocolo CAN onde o objectivo não será de impulsionar a imagem ou a tecnologia de um ou outro protocolo, mas sim apresentar os pontos fortes e fracos de cada protocolo.

4.1 Protocolo Modbus

Nesta secção são apresentados os resultados dos testes feitos no capítulo anterior, da secção 3.1.7 referentes ao protocolo Modbus. Estes resultados são apresentados através de uma tabela de animação criada no PLC cliente Modbus que é constituida pela tabela de controlo, tabela de transmissão e a tabela de recepção.

Na figura 4.1 é apresentado o resultado através de uma tabela de animação do cliente Modbus, de uma requisição sem nenhum erro em que o objectivo era fazer a leitura de três memórias, que correspondem a %MW7 (a temperatura do simulador), %MW8 (saída do processo - ventiladores) e %MW9 (acção de controlo). %MSG2.D e %MSG2.E representam o bloco da função %MSGx enquanto que %SW64 corresponde a memória do sistema.

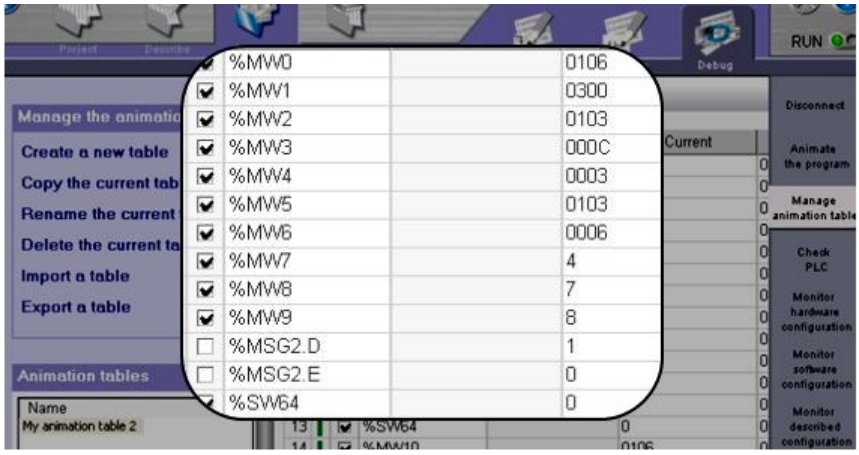


Figura 4.1: Tabela de animação do PLC cliente Modbus - supervisor na leitura das variáveis do processo.

Tal como era de esperar, da memória %MW5 à memória %MW9 corresponde a tabela de resposta. O servidor retorna, ou seja, responde à requisição do mestre inserindo na tabela de resposta, memória %MW5, o seu endereço 01 e o código de função Modbus 03. A memória %MW6 da tabela de resposta, com o valor de 06 corresponde ao número de *bytes* dos dados enviados pelo servidor.

Na figura 4.2 é apresentado o resultado do pedido de escrita da referência com valor de 10 na memória %MW40 (0028 hexadecimal) do servidor ou controlador. Como se tratou novamente de uma requisição sem nenhum erro, o servidor retorna ao mestre o seu endereço (01) e o código da função (06). Diferente da requisição de leitura, na escrita

de uma memória o servidor para além de retornar o seu endereço e o código de função também envia para o cliente ou mestre a memória que foi escrita %MW16; e o valor escrito %MW17 igual a 10.

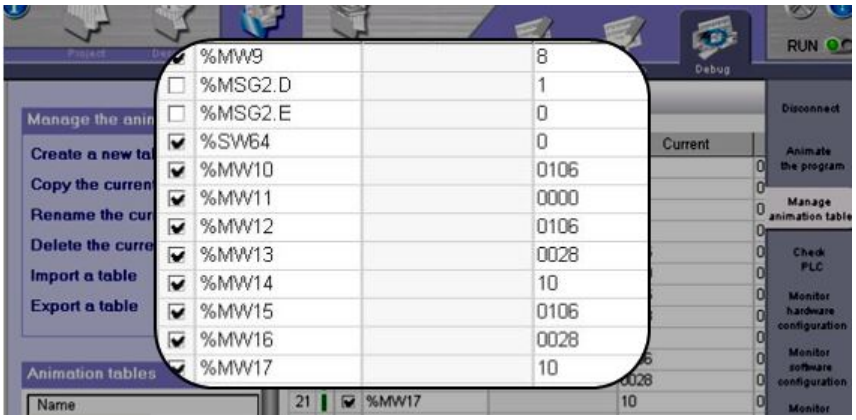


Figura 4.2: Tabela de animação do PLC cliente Modbus - supervisor na escrita da referência.

Em ambos os casos das figuras 4.1 e 4.2 pode-se ver que os dois identificadores de erro estão a 0 (%MSG2.E e %SW64) o que significa que não há nenhum erro na comunicação, enquanto que %MSG2.D está a 1 significando que a comunicação está completa, como se pode ver na tabela 4.1. Para a memória do sistema %SW64 rever a tabela 3.3.

Tabela 4.1: Bloco da função MSGx.

Entrada/Saída	Definição	Descrição
R	Entrada *Reset*	Definido 1 : reinicializa a comunicação (%MSGx.E = 0 e %MSGx.D = 1)
%MSGx.D	Comunicação completa	0 : requisição em progresso 1 : comunicação completa no fim da transmissão, fim do caracter recebido, erro, ou reset do bloco
%MSGx.E	Erro	0 : tamanho da mensagem OK e conexão OK. 1 : mau comando, tabela incorrectamente configurada, caracter recebido incorrecto, ou tabela de recepção cheia.

No segundo teste foi realizada uma requisição sem nenhum erro mas o servidor não pode manipular este pedido, porque o pedido de leitura foi feito de uma memória que não foi programada (%MW3 := 16#0032) e a resposta foi uma *exception response*, na tabela de resposta (%MW5 e %MW6) o servidor inseriu o seu endereço (01) e o código da função 03 somado com 0x80 o que resulta em 83 (vide figura 4.3).

E em seguida na memória %MW6 é fornecido o código *exception* 02 que segundo

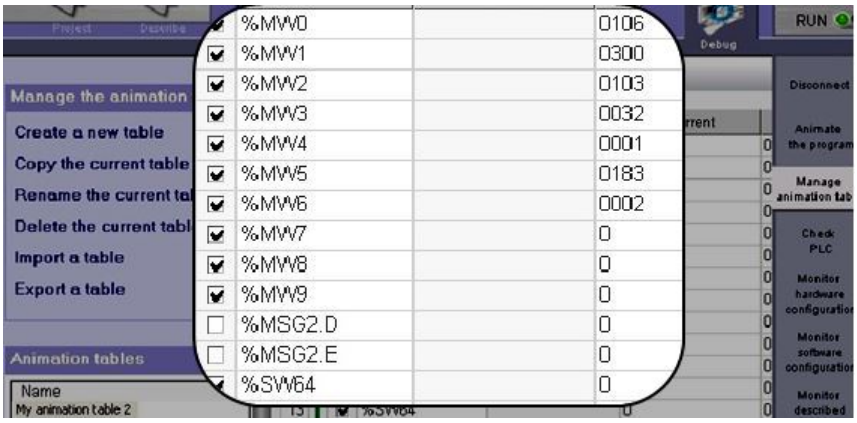


Figura 4.3: Tabela de animação de uma requisição em *exception response.*

a organização Modbus [3], corresponde a um endereço de dados inválido (*"illegal data address"*), como se observa na tabela 4.2. As memórias %MW7, 8 e 9 não apresentam nenhum valor porque nenhuma leitura foi feita por parte do servidor Modbus. Mesmo sendo uma *exception response* não há nenhum erro na comunicação entre o servidor e o cliente Modbus, o valor 0 em %MSG2.D significa apenas que a requisição está em progresso e o valor 0 em %MSG2.E indica que o tamanho e a conexão estão OK (rever tabela 4.1).

Tabela 4.2: Lista dos códigos de excepção. [3]

Códigos de excepção Modbus	
Código	Nome
01	Função inválida
02	Endereço inválido
03	Valor dos dados inválidos
04	Falha no escravo
05	Reconhecimento
06	Escravo ocupado
08	Erro na memória de paridade
0A	*Gateway* indisponível
0B	*Gateway* não respondeu

Na figura 4.4 está representada a tabela de animação do mestre que corresponde ao resultado de uma requisição em *broadcast.* Nesta requisição foi pedido aos servidores embora o teste foi feito com somente um servidor, que coloque ou que seja escrito nas suas memórias %MW40, que corresponde à memória para a referência do processo no servidor Modbus ou controlador, o valor 5. Era de esperar que nenhuma resposta fosse retornada do servidor, por isso a tabela de resposta constituida pelas memórias %MW15, %MW16 e %MW17 estão a 0.

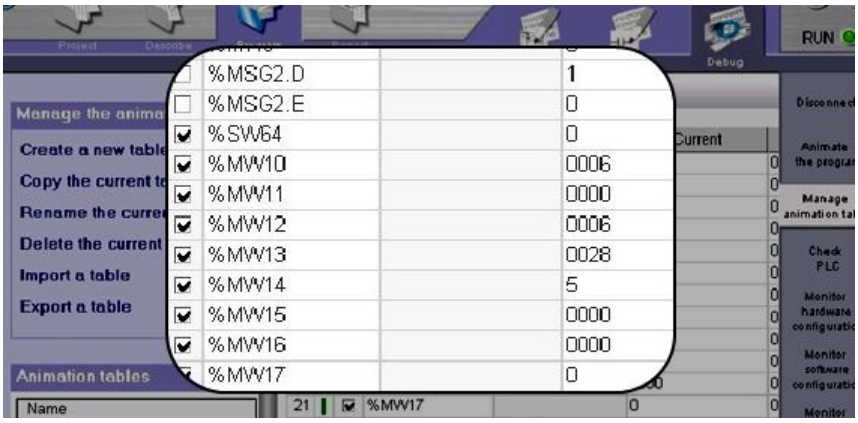


Figura 4.4: Tabela de animação da comunicação *broadcast*.

O facto de o servidor não responder por tratar-se de uma requisição em *broadcast* notou-se que não existe nenhum erro na comunicação, analisando a *word* do sistema %SW64 e as saídas do bloco %MSG2, rever as suas respectivas tabelas.

E por último, na figura 4.5, um teste em que surge um erro na comunicação. O código digitado no mestre é totalmente igual ao que foi digitado na figura 4.1 diferindo na tabela de animação a memória do sistema %SW64 e na saída do erro %MSG2.E. Enquanto que %MSG2.D tem a requisição em progresso igual a 1, %MSG2.E está em 1 o que significa que existe algum erro e a *word* do sistema %MW64 com o valor 5 indica que o tempo de resposta (*response time out*) expirou (rever tabela 3.3).

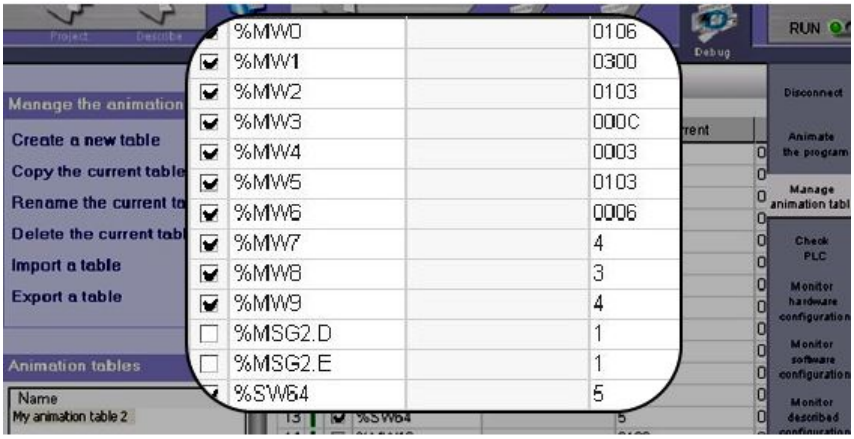


Figura 4.5: Erro numa comunicação Modbus.

Os resultados dos testes da comunicação Modbus foram os esperados tendo em conta o estudo que se fez do protocolo no Capítulo 2.

Embora não se tenha dado tanta abordagem ao padrão EIA-485, em seguida o objectivo será a apresentação dos níveis de tensão do barramento considerando este padrão.

Tal como já foi dito nos capítulos anteriores o protocolo Modbus foi implementado utilizando o padrão EIA-485 ou simplesmente RS-485 em *half-duplex*, ligado a um barramento comum de dois fios (D1 e D0) e uma conexão comum. O padrão RS-485 permite uma transmissão de dados com tensões de -7V até +12V. Baseando-se nesta análise, resolveu-se apresentar os níveis de tensão dos dois fios do barramento e o sinal diferencial que é transmitido entre D1 e D0.

Para apresentação destes sinais foi utilizado um osciloscópio em que o canal 1 foi ligado entre D1 e o ponto comum (terra), canal 2 ligado entre D0 e ponto comum. Na figura 4.6 são apresentados os níveis de tensão dos dois fios referenciados à terra, e vê-se pelo osciloscópio que o sinal D0 é complementar do sinal D1.

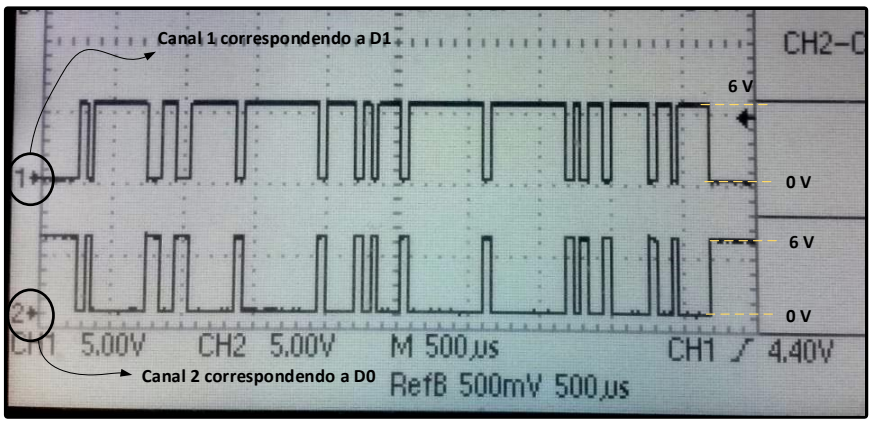


Figura 4.6: Níveis de tensão de D1 e D0.

A figura 4.7 apresenta os mesmos níveis de tensão de D1 e D0 já apresentadas na figura anterior, mas desta vez com a inclusão do sinal diferencial, ou seja da subtracção entre D1 e D0. Este sinal diferencial apresenta tensões entre -6 e +6V, estando dentro das tensões de transmissão de dados de um padrão RS-485. Quando o sinal diferencial (D1-D0, ou seja CH1-CH2), figura 4.7 fôr igual a -6V o sinal apresenta o nível lógico 0 e quando fôr +6V nível lógico 1.

No Capítulo 3, durante a configuracção da rede Modbus, definiu-se um *baudrate* de 19200 bps, significando que em 1 segundo são enviados 19200 *bits*. Recorrendo à regra de três simples com objectivo de se saber em quanto tempo é enviado um *bit* chegou-se a este resultado:

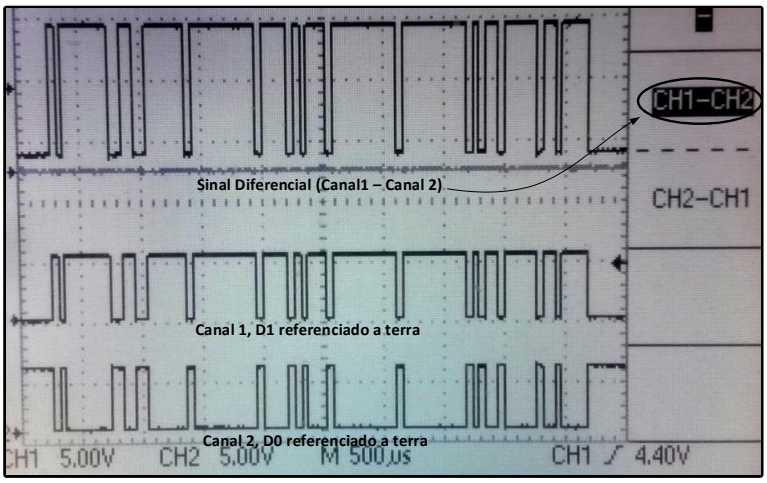


Figura 4.7: Sinal diferencial (D1-D0).

$$time\ bit = \frac{1\ bit \cdot 1\ s}{19200\ bit} \Rightarrow time\ bit = 52\ \mu s$$

Arredondando este valor obtem-se 50 μs, foi possível ser apresentado no osciloscópio, como se pode ver na figura 4.8. O que significa que embora se tenha desprezado os 2 μs, os dados estão sendo enviados conforme se esperava.

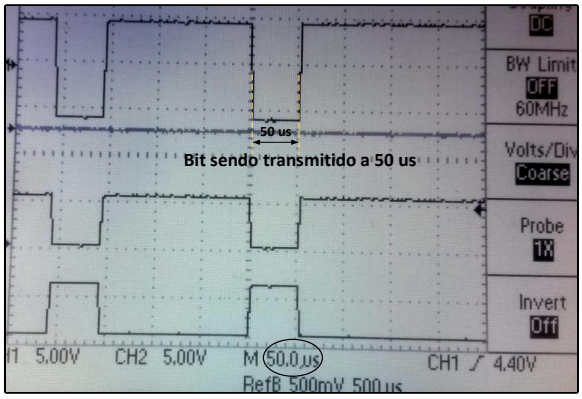


Figura 4.8: Tempo de bit.

Após a consideração da camada física Modbus e de se ter analisado os sinais no osciloscópio. Continuando e a considerar o teste em que as requisições do cliente Modbus foram feitas sem nenhum erro e o servidor Modbus responder ao pedido de leitura (memórias associadas a temperatura do simulador, sinal de saída (ventiladores) e acção de controlo) e a escrita da referência do processo, de seguida será apresentado o resultado da comunicação OPC.

Durante a implementação do protocolo Modbus e do padrão OPC® para se criar a interface no Scilab®, um dos principais itens foi o tempo em que o PLC cliente comunicava com o PLC servidor e o tempo que o PLC cliente estivesse disponível para ser lido pelo servidor OPC. Definiu-se o diagrama temporal representado na figura 4.9 em que para a comunicação Modbus foi definido 0.5 s para escrita, 0.5 s para leitura e 3 s para a comunicação OPC. Só se obteve bom desempenho do sistema com os tempos definidos, para valores inferiores a estes obteve-se valores incorrectos.

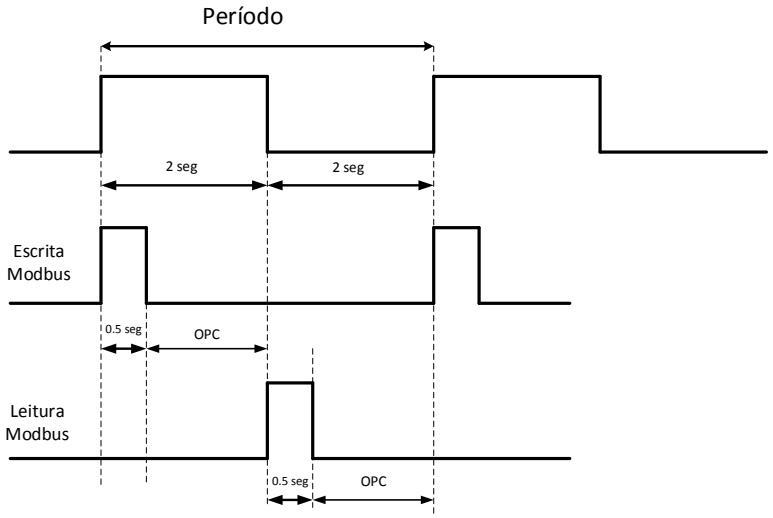


Figura 4.9: Diagrama temporal da comunicação Modbus e do padrão OPC.

Na figura 4.10 está representado o servidor OPC a partir da sua aplicação *MatrikonOPC Explore* onde é definido o grupo e os itens OPC. Reparando em **B**, vê-se claramente os itens definidos e com uma boa qualidade da comunicação OPC. É possível alterar-se a referência do processo através do item 4:151.

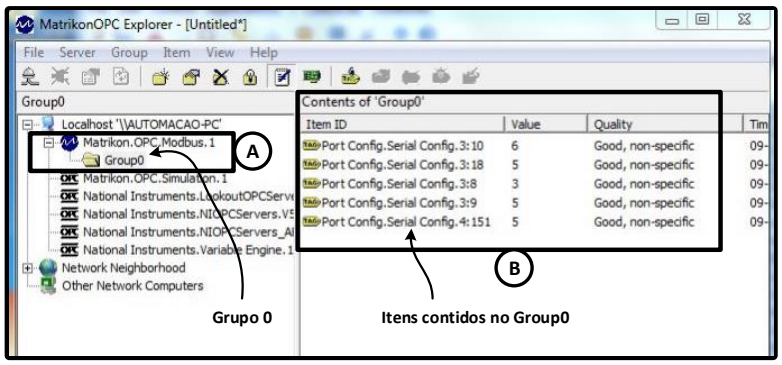
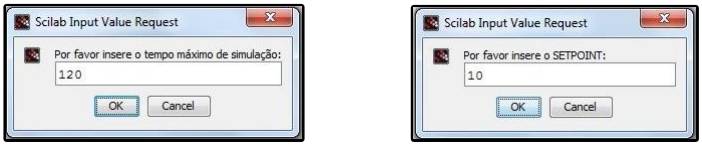


Figura 4.10: Matrikon Explore - Criação do grupo e dos itens OPC.

Nas figuras 4.11 e 4.12 são apresentados os resultado do código do editor do cliente OPC em Scilab criado para a interface. No início da simulação depois da conexão do OPC cliente (Scilab®) ao OPC servidor (MatrikonOPC) abrirá a janela de definição do tempo de simulação. Após a definição do tempo de simulação e premindo no botão iniciar a simulação, será aberto o gráfico de monitorização das variáveis do processo. Em seguida, premindo no botão *setpoint* permite-se que o utilizador insira a referência para o sistema de controlo.



(a) Janela de definição do tempo de simulação.

(b) Janela de definição da referência do processo.

Figura 4.11: Janelas de definição para a simulação.

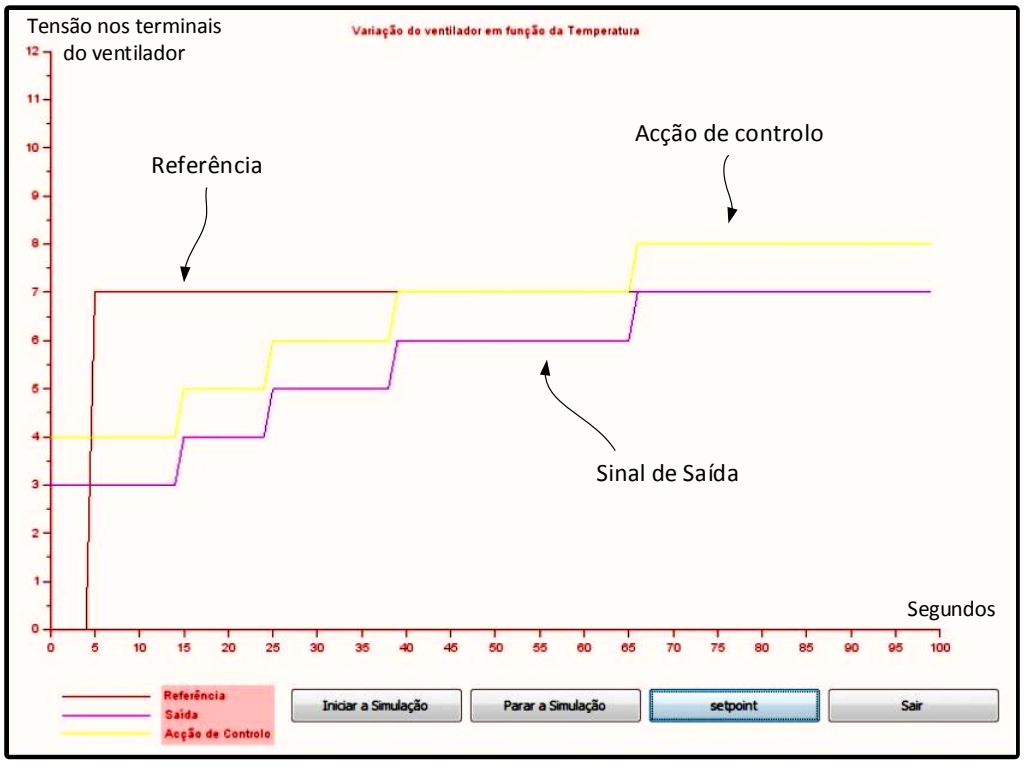


Figura 4.12: Interface no Scilab, com sinais de referência, saída e acção de controlo.

Na figura 4.11 são apresentadas as duas janelas de definição dos parâmetros de si-

mulação onde a figura 4.11(a) corresponde à janela de definição do tempo de simulação e a 4.11(b) à janela de definição da referência.

Nos primeiros instantes, quando a referência está a 0 (zero), o sinal de saída e a acção de controlo encontram-se em 2 e 4 respectivamente. Foi assim definido logo no início porque o objectivo dos ventiladores é de estarem a refrigerar um outro sistema, por isso logo que o sistema é activado o sinal de saída estará no seu nível mínimo que corresponde a 2.

Após a definição da referência com o valor 7 aproximadamente no instante 5 s, o controlador começa a fazer o controlo do processo e só aproximadamente no instante 85 s o sinal de saída será igual à referência.

4.2 Protocolo CAN

Nesta secção são apresentados os resultados dos testes feitos no capítulo 3 para o protocolo CAN. Para análise dos sinais no barramento será apresentado e verificado no osciloscópio cada detalhe dos campos que constituem a trama de dados.

O resultado do primeiro teste é apresentado na figura 4.13 em que está sendo continuamente enviado uma trama de dados. A figura apresenta três tramas a serem enviadas pelo nó emissor. O objectivo deste teste é somente de verificar como as tramas são enviadas e o espaçamento temporal entre elas.

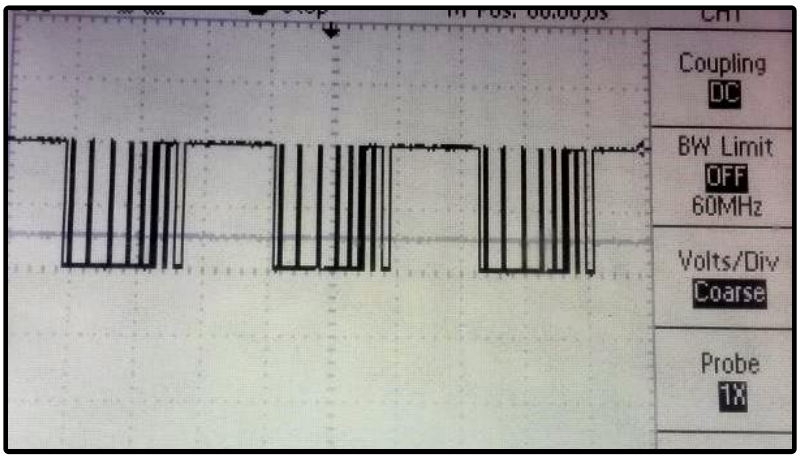
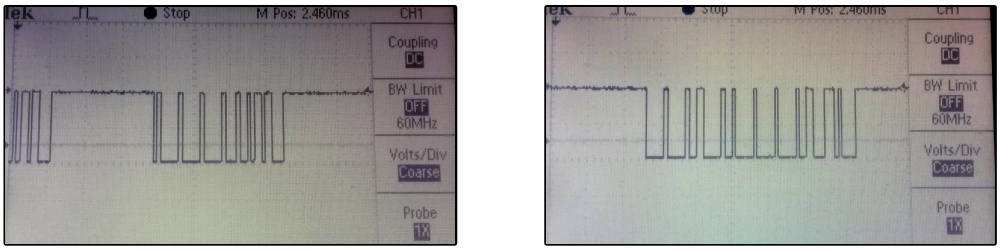


Figura 4.13: Nó transmissor enviando tramas repetidamente.

Na figura 4.14 considera-se uma trama estendida de 29 *bits* (figura 4.14(b)), nota-se

claramente que a trama de identificador de 29 *bits* apresenta um número maior de *bits* que a do identificador de 11 *bits* figura 4.14(a). As duas tramas na figura 4.14 foram colocadas à mesma escala no osciloscópio.



(a) Trama com identificador de 11 *bits*.

(b) Trama com identificador de 29 *bits*.

Figura 4.14: Trama e dados de 11 e 29 *bits*.

Na análise da figura de identificador de 11 *bits* 4.14(a) verificou-se que o *bit* RTR é dominante por estar a tratar-se de uma trama de dados e o *bit* IDE é dominante por possuir um identificador de 11 *bits*. Para além disto viu-se também a existência de vários *bits stuffing* isto porque o identificador 0x400 hexadecimal (0x100000000000 em binários) apresenta muitos *bits* dominantes.

Ao contrário do identificador de 11 *bits* o identificador de 29 *bits*, figura 4.14(b), apresentou dois *bits* recessivos SRR e IDE e entre esses dois *bits* encontravam-se os *bits* do identificador base (0x00000000000) e do identificador de extensão (0x000000010000000000), o que resultou em muitos *bits stuffings* maior do que com identificador de 11 *bits*.

Devido à dimensão do ecrã do osciloscópio utilizado para a apresentação destes resultados entrou-se em maiores detalhes apenas no identificador de 11 *bits*, isto porque, foi possível ter o sinal completo com uma maior escala no ecrã do osciloscópio e desta forma ser possível analisar as tramas, e também porque será este utilizado no controlo do processo.

Na figura 4.15 está representada a trama de dados com identificador de 11 *bits*, com os campos de arbitragem, de controlo, CRC e o campo de confirmação (ACK). No início da trama tem-se um *bit* dominante (SOF), o campo de arbitragem é constituido por 11 *bits* tendo 2 *bits stuffing* terminando com o *bit* RTR que é dominante porque estamos perante uma trama de dados.

O campo de controlo é totalmente dominante tendo um *bit stuffing* depois de 5 *bits* dominantes. O campo CRC apresenta 16 *bits* tendo o último *bit* (*bit* delimitador) transmi-

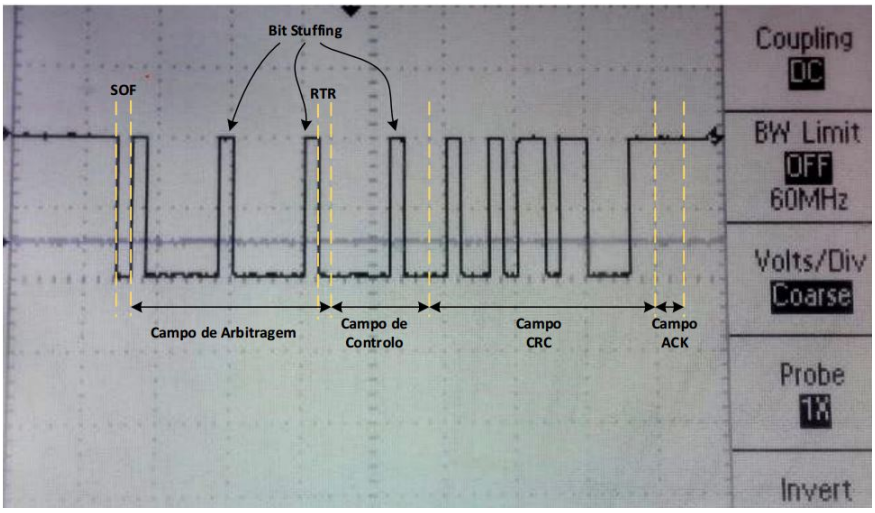


Figura 4.15: Transmissor da trama por parte do nó transmissor sem nenhum receptor.

tido recessivamente. E por último o campo de confirmação com dois *bits* recessivos. Com esta figura 4.15, é possível confirmar-se os campos de uma trama de dados, devendo-se salientar que o campo de dados que fica entre o campo de controlo e o campo CRC é 0, significando que não foi enviado qualquer dado e que o DLC também é igual a 0.

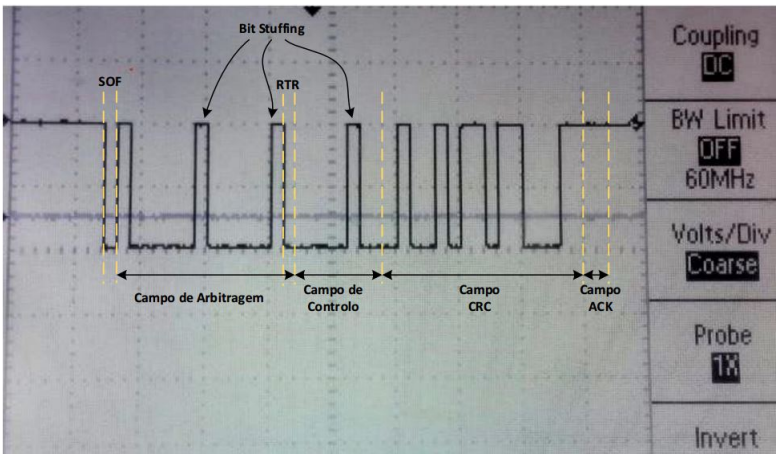
No campo de confirmação ACK tendo-se os dois *bits* (*ACK Slot* e *ACK Delimiter*) no estado recessivos significa que a requisição não está sendo recebida por nenhum outro nó.

A análise do *bit stuffing* é feita na figura 4.15 em que é apresentada uma trama de 11 *bits* com um identificador 400 em hexadecimal (0x10000000000). Após o *start bit*, nota-se que o primeiro *bit* é 1 e de seguida zeros mas depois de 5 *bits* a 0 surge claramente um *bit* em 1 (*bit stuffing*), depois deste *bit stuffing* surgem novamente 5 zeros e obrigatoriamente o bit seguinte tem de ser um *bit stuffing*.

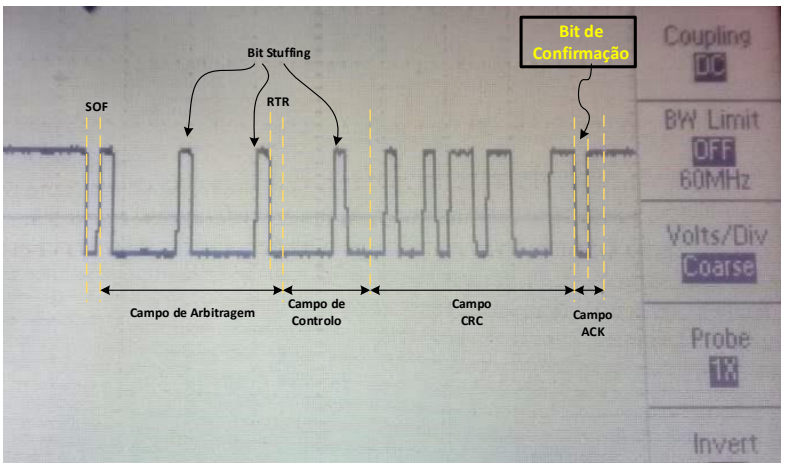
A figura 4.16 apresenta duas tramas uma representa a trama transmitida sem receptor (figura 4.16(a)), ou seja, a trama está sendo enviada pelo nó emissor mas sem nenhum dispositivo receptor. Enquanto que na figura (4.16(b)) é apresentado o sinal enviado pelo transmissor tendo-se um nó receptor. Viu-se durante os estudos do protocolo CAN que durante uma transmissão, os dois *bits* ACK são recessivos, e o nó receptor envia para o transmissor uma trama totalmente recessiva mas na posição do *bit ACK slot* será um *bit* dominante confirmando a recepção da mensagem, como se pode verificar na figura 4.16. As figuras 4.16(a) e 4.16(b) representam as tramas do barramento.

A única diferença entre as duas tramas consiste no *bit ACK slot* em que na figura com

o receptor apresenta um *bit* dominante confirmando a recepção dos dados.



(a) Trama transmitida sem nenhum receptor.



(b) Trama com um receptor e a inserção do bit de confirmação.

Figura 4.16: Envio e resposta com trama de 11 *bits*.

Na figura 4.17 os resultados obtidos foram os esperados para a análise do nível de tensão no barramento. O transceptor MCP2551 converte os sinais em tensões diferenciais que para o CAN H corresponde a 3,5V e para o CAN L 1,5V.

A diferença entre as duas tensões é de 2V, considerando que no estado recessivo tem-se uma tensão de 2,5V. Na figura 4.18 confirma-se os níveis de tensão com os *bits* recessivos e dominantes numa trama de dados. No osciloscópio o CAN H foi ligado ao canal 1 enquanto que CAN L ao canal 2.

A trama de dados certa para a análise do barramento surge na subtracção do canal 2 ao canal 1 (CH2 - CH1) e não do canal 1 ao 2 (CH1 - CH2) que a príncipio seria o que se esperava, mas a subtracção (CH1 - CH2) resulta numa trama invertida, isto porque o

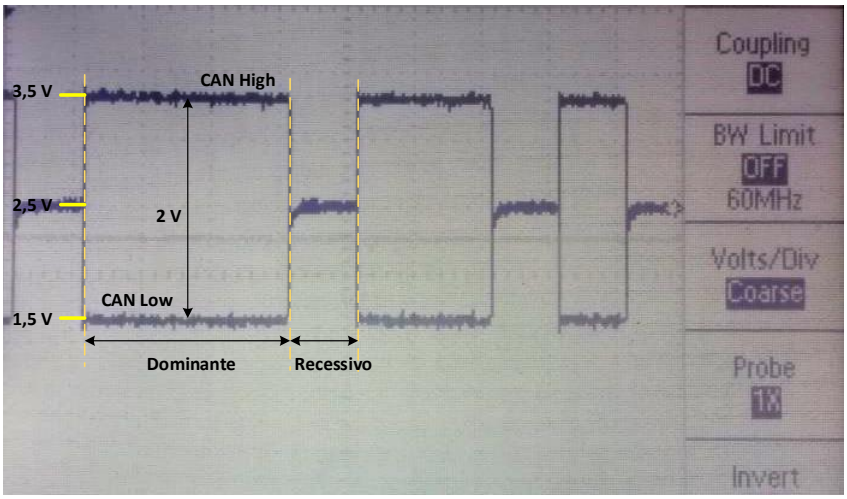


Figura 4.17: Nível de tensão do CAN_H e CAN_L.

barramento sem nenhuma trama a ser transmitida já se encontra em "1" *bit* recessivo.

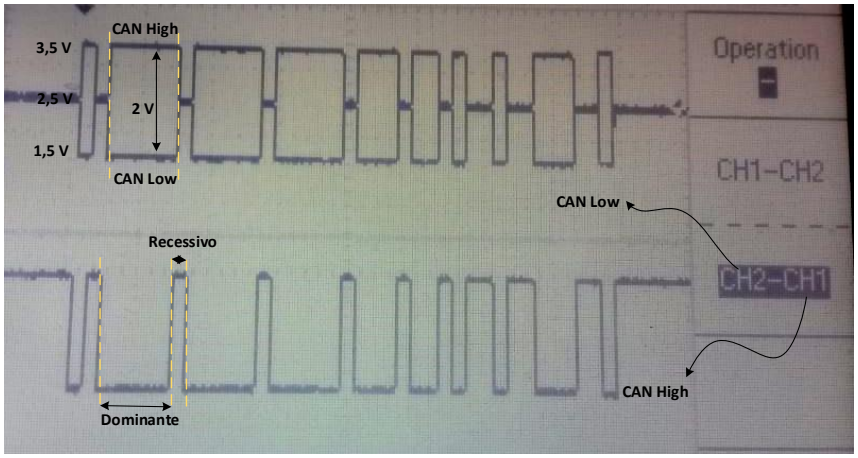


Figura 4.18: CAN_H, CAN_L, *Bit* Dominante e Recessivo.

Só passa a dominante quando a diferença entre CAN H e CAN L fôr de 2V, ou seja, o barramento possui lógica negativa o que significa que quando a diferença de tensão entre CAN L (CH2) e o CAN H (CH1) fôr 0 (2,5V) o barramento estará em 1 (recesssivo) e não em 0. E quando a diferença entre os níveis de tensão fôr de 2 V o barramento estará em 0 (dominante) e não em 1. O nível dominante (*bit* 0) tem maior prioridade ou sobreescreve o nível recessivo (*bit* 1).

Depois de se ter apresentado os níveis de tensão em comparação com os *bits* dominantes e recessivos é chegado a altura da apresentação das tramas de alta prioridade (totalmente

dominante) e baixa prioridade (totalmente recessiva). O resultado do teste com mais alta prioridade, identificador igual a 0x0000 hexadecimal, é representado na figura 4.19 e com baixa prioridade (0x7FF hexadecimal) é mostrado na figura 4.20.

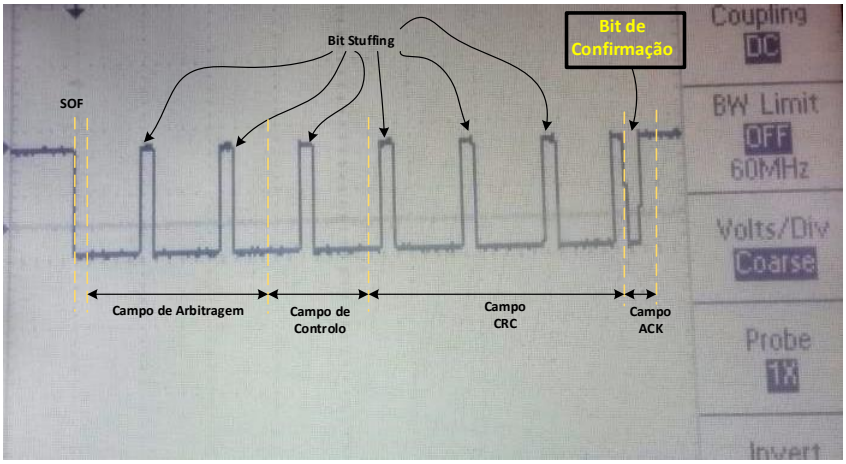


Figura 4.19: Trama com alta prioridade.

Considerando a trama de mais alta prioridade, figura 4.19, não foi enviado nenhum dado, DLC é igual a 0, o que resulta numa trama de dados com todos os seus *bits* dominantes, à excepção do *ACK delimiter* e o campo de fim da trama que têm de ser recessivos.

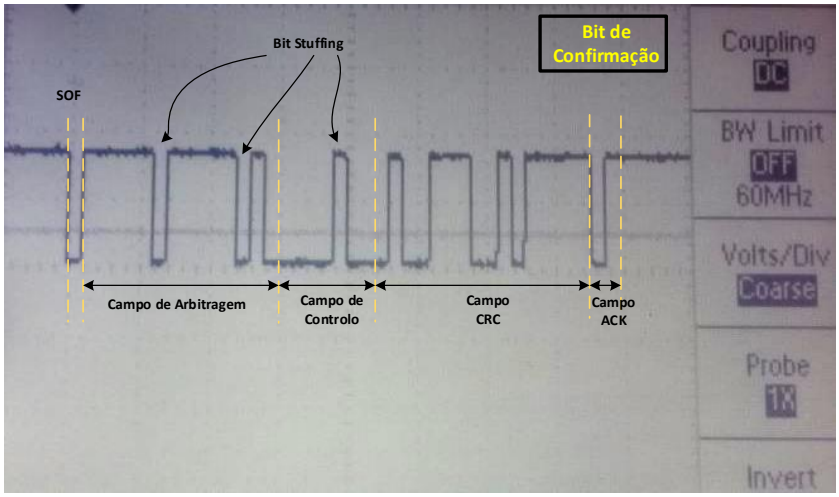


Figura 4.20: Trama com baixa prioridade.

E devido a esta sequência de muitos *bits* dominantes, resultou em vários *bits stuffing*, no campo de arbitragem aparecem 2 *bits stuffing*, campo de controlo 1 e no campo CRC 3. Uma trama de dados com o identificador 0x000h terá a máxima prioridade, porque

apresenta todos os seus *bits* em dominantes. Foi este o identificador escolhido para o nó supervisor por este ter maior prioridade no barramento devido às alterações de dados que podem surgir para o processo a controlar.

A figura 4.20 resulta ao contrário do que sucedeu no caso anterior. Nesta figura tem-se uma trama com os *bits* do identificador todos a recessivos (0x7FF hexadecimal, 11111111111 em binário), resultando em três *bits stuffing*.

Em todos os casos utilizou-se um *baudrate* de 20 kbps e verificou-se um *bit* nominal, t_{bit} de 50 μs como se pode ver na figura 4.21

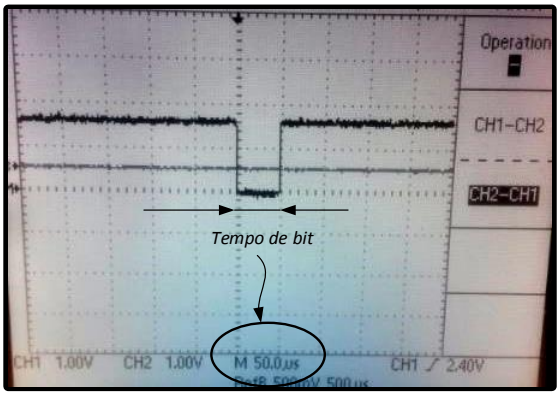


Figura 4.21: Tempo de bit.

Através da análise da configuração do controlador CAN, MCP2515, foi possível calcular os parâmetros do *bit* nominal ou *time bit*. Para se fazer o cálculo destes segmentos primeiramente foi feito o cálculo do *time quantum* e recorreu-se à equação 4.1:

$$t_q\ [\mu s] = \frac{2 \cdot (BRP + 1)}{f_{osc}} \tag{4.1}$$

Os registos da configuração do controlador CAN foram utilizados para efectuar o cálculo do *time quantum*. Para o caso da configuração do *baudrate* de 20 *kbps*, a taxa de transmissão *prescaler BRP* foi de 0x0F que corresponderá a 15 em decimal e foi utilizado uma frequência para o oscilador de 16 MHz. O resultado para o cálculo do *time quantum* é apresentado de seguida:

$$t_q = \frac{2 \cdot (15 + 1)}{16} \ \Rightarrow \ t_q = 2\ \mu s$$

Não confundir a taxa de transmissão *prescaler* com a taxa de transmissão de dados.

A taxa de transmissão *prescaler BRP*, igual a 15 em decimal determinam a duração do *time quantum* enquanto que os 20 kbps determina a duração de um *bit*.

Para verificar a quantidade necessária de *time quantum* que foi necessária para se ter um *time bit* de 50 μs com um *time quantum* correspondente a 2 μs recorreu-se a equação 4.2:

$$n = \frac{t_{bit}}{tq} \tag{4.2}$$

O resultado do cálculo da quantidade de *time quantum* necessário é de:

$$n = \frac{50\ [\mu s]}{2\ [\mu s]} \ \Rightarrow \ n = 25\ unidades$$

O número total de *time quantum* necessário é de 25, o que signfica que para um comprimento máximo de 2500 metros para um nominal *bit time* de 50 μs e também para diminuir a latência é necessário 25 unidades de *time quantum*.

$$t_{Sync\_Seg} = 1 \cdot t_q \tag{4.3}$$

O tempo para o segmento de sincronização tem a duração de 1 t_q e calculou-se através da equação 4.3:

$$t_{Sync\_Seg} = 1 \cdot 2 \ \Rightarrow \ t_{Sync\_Seg} = 2\ \mu s$$

Como já se sabe que foram necessárias 25 unidades logo o segmento de propagação será igual à expressão 4.4:

$$t_{Prop\_Seg} = 8 \cdot t_q \tag{4.4}$$

$$t_{Prop\_Seg} = 8 \cdot 2 \ \Rightarrow \ t_{Prop\_Seg} = 16\ \mu s$$

O resultado da equação 4.5, y, dirá a quantidade necessária para o segmento de fase 1 e para o segmento de fase 2.

$$y = t_{bit} - t_{Prop\_Seg} - t_{Sync\_Seg} \ \Rightarrow \ 50 - 16 - 2 = 32\ \mu s \tag{4.5}$$

Se a divisão de y pelo *time quantum*, equação 4.6, fôr um resultado par significa que os segmentos 1 e 2 terão o mesmo tamanho.

$$m = \frac{y}{tq} \tag{4.6}$$

$$m = \frac{(t_{bit} - t_{Prop\_Seg} - t_{Sync\_Seg})}{tq} \;\Rightarrow\; m = 16\ \mu s$$

Sendo par o resultado da divisão e igual a 16, logo teremos para cada segmento de fase a 16 [μs]. A soma de todos os parâmetros do *bit time* será igual a 25 t_q.

$$t_{Phase\_Seg1} = 8 \cdot t_q \;\Rightarrow\; t_{Phase\_Seg2} = 8 \cdot t_q \tag{4.7}$$

$$t_{Phase\_Seg1} \;=\; t_{Phase\_Seg2} \;= 16\ \mu s$$

Após os cálculos dos segmentos que constituem um *bit* de 50 μs, que resultou em 2 μs para segmento de sincronização, 16 μs para o segmento do tempo de propagação, e 16 μs para o segmento 1 e 2 totalizando os 50 μs, a figura 4.22 apresenta o tempo destes quatro segmentos com uma escala de tempo do osciloscópio de 10 μs, e definiu-se o *time quantum* tal como foi calculado.

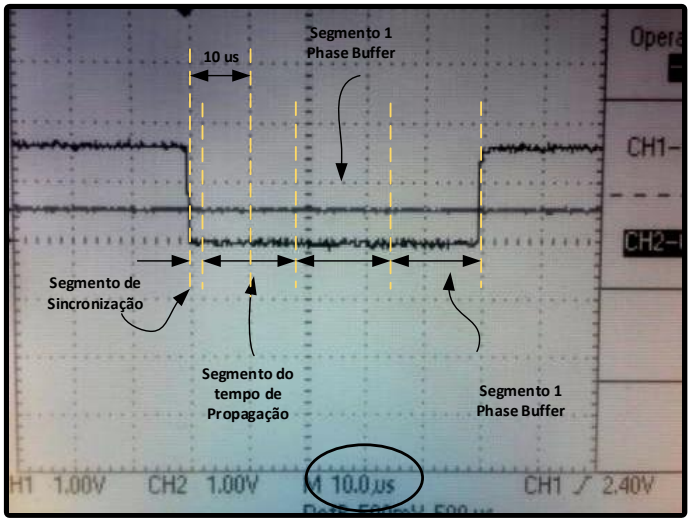


Figura 4.22: Segmentos do tempo de um *bit*.

Na figura 4.23 é apresentado o resultado das duas tramas que estão sendo constantemente enviadas pelo nó controlador para o nó supervisor, e como se pode verificar tem-se a acção de controlo, o sinal de saída e o espaço temporal entre as duas tramas.

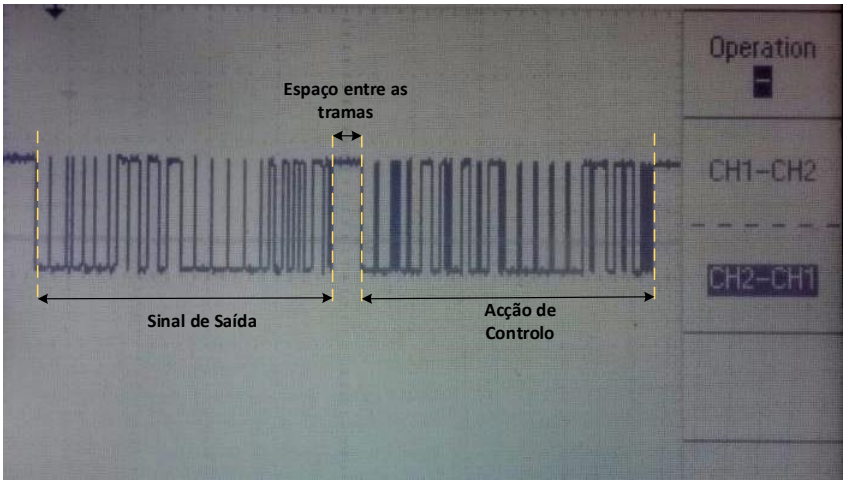


Figura 4.23: Trama transmitida pelo nó controlador com os dados da saída do processo e a acção de controlo.

Estes sinais apresentados na figura 4.23 são possíveis de serem acedidos na porta COM configurada para o Arduino. Abrindo o monitor da porta série a partir do arduino resulta na figura 4.24.

Nesta figura, 4.24, em **A** é apresentada a mensagem de que o controlador foi bem configurado, em **B** é apresentada a mensagem de que foi inicializada a configuração da máscara e do filtro, e que foram definidos com sucesso.

Em **C** e **D** é apresentado o identificador do nó que enviou a mensagem e tal como se esperava é igual a 0x01, e em seguida os valores da acção de controlo com 0.22 e do sinal de saída com 0.30.

Na figura 4.25 são apresentados os três principais sinais do processo: referência, sinal de saída e acção de controlo.

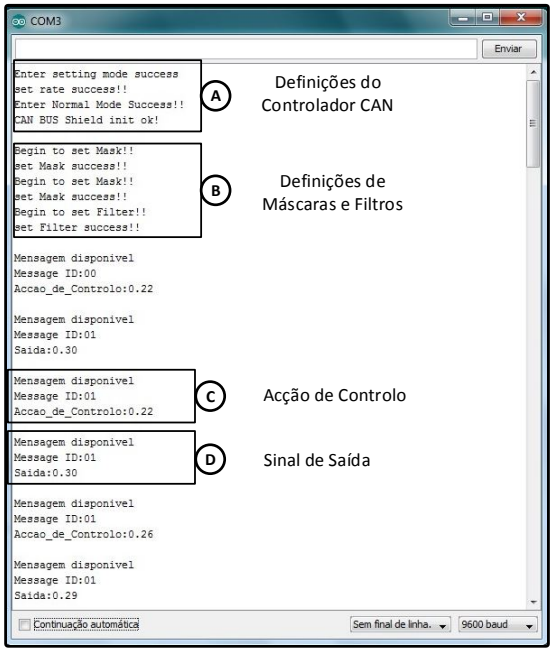


Figura 4.24: Resultado de leitura da porta COM ligada à placa Arduino Supervisor.

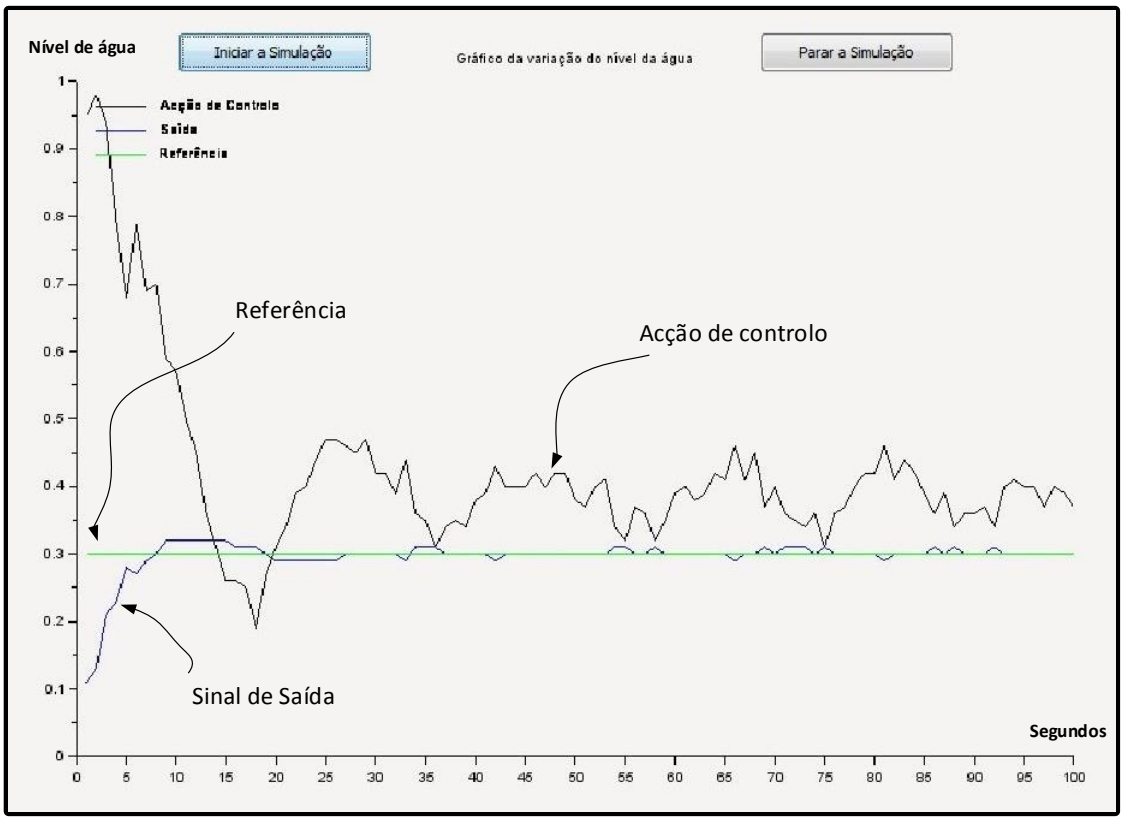


Figura 4.25: Interface para apresentação das variáveis do processo.

Como se pode ver na figura 4.25, o controlador PI com *anti windup* apresenta um bom desempenho no controlo do processo, tentando manter o sinal de saída próximo da referência.

Na interface criada o botão *iniciar a simulação*, permite que seja mostrado o gráfico abrindo a porta série. O sinal de saída começa em 0 (zero), mas devido ao atraso da abertura da porta é visto no gráfico já no instante 0.1 s.

O botão *parar a simulação* permite fechar a porta COM e parar a gráfico de simulação. O sinal de referência e os ganhos são inseridos no IDE do arduino e para isso antes terá de se fechar a porta COM aberta pelo arduino.

Após a análise dos resultados dos dois protocolos na próxima secção será feita uma comparação geral analisando os seus pontos fortes e fracos dependendo da aplicação.

4.3 Protocolo Modbus *versus* protocolo CAN

Em primeiro lugar CAN e Modbus são duas tecnologias diferentes de rede, e cada um é adequado para certas aplicações no mundo da automação.

De um lado tem-se o CAN com uma taxa de transmissão de até 1 Mbps, comprimento da rede física limitada com a taxa de 1 Mbps em 40 metros e um comprimento máximo de dados de até 8 *bytes* por mensagem. Comparando com o Modbus em linha série, taxa de transmisssão de até 115 kbps ou mais, no PLC implementado é possível somente definir até 38400 bps. Modbus em linha série possui um comprimento de dados de até 252 *bytes*, e o comprimento da rede física em RS485 é de 1200 metros.

Vale recordar que o estudo e a comparação de estes protocolos não é uma tentativa de impulsionar a imagem ou a tecnologia de um ou de outro. O importante é que ambos os protocolos têm os seus pontos fortes e fracos dependendo do tipo de aplicação. CAN é, definitivamente, uma escolha adequada para controlo de pequeno porte, ou seja, soluções embutidas que exigem a comunicação de multiprocessador. O desempenho do Modbus desenvolve-se especificamente na comunicação de dispositivos inteligentes, sensores e instrumentos, bem como para aplicação de monitorização de dispositivos de campo usando PCs e interfaces HMI. A tabela 4.3 ilustra uma comparação uma comparação entre o protocolo CAN e Modbus.

O protocolo CAN apresenta várias camadas de aplicação como CANopen, Devicenet, SAE J1939 entre outros, o que permite aplicações maiores para o CAN mas isto não deixa de limitar a camada física do barramento CAN. CAN é um sistema de alta velocidade e é superior às tecnologias convencionais série tais como RS232, RS485 em relação a funcionalidade e fiabilidade, pois previne colisões de mensagens e proporciona excelente detecção e recuperação de erros.

Tabela 4.3: Modbus *versus* CAN.

Modbus *versus* CAN		
	Modbus	CAN
Empresa da Tecnologia	Modicon	Robert Bosch Gmbh
Topologia da rede	Linha, estrela árvore, barramento	Barramento
Meio físico	Par trançado	Par enterlaçado
Máximo nº de dispositivos	32	127
Distância máxima	350 metros	25 - 5000 metros (dependendo do baudrate)
Método de comunicação	Cliente/Servidor	Produtor/Consumidor
Propriedade da transmissão	300 bps - 115 kbps	10 kbps - 1 Mbps
Tamanho de dados	0 - 252 *bytes*	0 - 8 *bytes*
Método de arbitragem	-	Não destrutiva
Verificação de erros	CRC - 16 *bits*	CRC - 15 *bits*
Tipo de transmissão	Série - Half/Duplex	Série - Half/Duplex
Tipo de transmissão série	Assíncrona	Síncrona
Controlo de acesso ao meio	Mestre - Escravo	CSMA/CD

Durante anos houve um grandde equívoco sobre o CAN foi que ele era apenas utilizado em automóveis. A verdade é que desde a sua introdução, provou ser uma tecnologia robusta, simples e versátil e, por conseguinte, pode ser aplicado em várias áreas onde microprocessadores precisam comunicar entre si.

Em conjunto com a sua utilização em automóveis, as aplicações com o CAN não só incluem tarefas de automação industrial, mas qualquer aplicação de controlo e sistema de barramento série, eliminando a fiação excessiva. CAN é superior a qualquer outro sistema de barramento de campo, no que diz respeito a baixo custo, à capacidade de funcionar num ambiente electromagneticamente ruidoso, têm um elevado grau de capacidade de tempo real, uma excelente capacidade de detecção de erros, falhas e facilidade de utilização.

CAN foi projetado com requisitos de tempo real, mesmo com a sua baixa taxa de transmissão de 1 Mbps e um comprimento máximo de dados de 8 *bytes*, CAN vence uma

conexão Modbus TCP/IP de 100 Mbps quando se trata de tramas/segundo[1] e colisão de mensagens. No entanto, CAN está limitado a um comprimento de rede física de cerca de 40 metros em 1 *Mbps*.

O protocolo Modbus apresenta 3 versões: Modbus em linha série, Modbus TCP/IP sobre *ethernet* e Modbus *Plus* o que permite uma dimensão e aplicação maior para o protocolo. A versão Modbus *Plus* possui vários recursos adicionais de roteamento, diagnóstico, endereçamento e consistência de dados. Esta versão ainda é mantida sob domínio da Schneider Electric e só pode ser implementada sob licença deste fabricante.

Modbus é um dos mais antigos e até hoje um dos protocolos de rede mais utilizado em PLCs, embora a versão Modbus em linha série não permita grandes taxas de transmissão, com um comprimento de rede física limitado. Por ser muito mais aplicado em nível de sensores e actuadores a sua versão em TCP/IP permite taxas de transmissão altíssimas chegando até aos 100 Mbps e comprimento de rede física ilimitada mas isto não deixa de limitar o paradigma cliente-servidor. Na versão Modbus TCP/IP o controlo de acesso ao meio é o CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*) o que significa que se fôr detectado uma colisão durante uma transmissão, a transmissão é interrompida.

Modbus é também um protocolo ideal para aplicações RTU em que é necessária a comunicação sem fios, desenvolvendo aplicações especificamente nas grandes redes de automação e controlo de máquinas de produção, incluindo a monitorização da produção. Mas Modbus não é apenas um protocolo industrial, e aplicações de infra-estruturas, contrução, transporte e energia também fazem uso dos seus benefícios. Modbus em particular na versão TCP/IP exige recursos de hardware significativamente maiores do que CAN.

[1] É um método comum de classificar o desempenho de uma rede

Capítulo 5

Conclusões

Neste capítulo são apresentadas as conclusões da dissertação na secção 5.1. A proposta de uma arquitectura para trabalho futuro, para a continuidade dos estudos das redes industriais nas áreas de controlo e automação, é apresentada na secção 5.2.

5.1 Conclusões

Nesta secção são apresentadas as conclusões do presente trabalho. A implementação do trabalho proposto permitiu obter conhecimentos das redes industriais com particular ênfase dos protocolos CAN e Modbus. Surgiram dificuldades na aquisição de material para a implementação dos protocolos, tanto a nível de *hardware* como de *software*.

No estudo para a implementação de ambos os protocolos, capítulo 2, constatou-se que existe muita informação relativa a estes protocolos e isso mostra que ambos os protocolos têm uma aplicação muito vasta. Durante o estudo, recorreu-se principalmente às páginas das organizações que gerem os protocolos por ser uma informação mais credível. Neste estudo, foi possível ter-se um bom conhecimento relativo aos protocolos e à área industrial no geral.

No estudo do protocolo Modbus, deu-se maior enfâse ao protocolo sobre linha série, modo RTU com RS-485, por ser o modo de implementação mais utilizado. Para o protocolo CAN o estudo centrou-se na camada de ligação de dados e na camada física, uma vez que a sua camada de aplicação permite vários protocolos.

Quanto aos métodos e às tecnologias que se utilizaram para a implementação das arquitecturas, optou-se pelos PLCs por estarem disponíveis no laboratório e permitirem comunicação em Modbus linha série com RS-485; optou-se pelas placas Arduino para o protocolo CAN pela existência de *shileds* CAN. A solução do padrão OPC foi obtida com o *software* da MatrikonOPC por apresentar uma versão gratuita durante 30 dias, e quanto ao Scilab por ser um *open source* gratuito.

Não se teve muita dificuldade para a implementação da arquitectura e verificou-se, para o caso do protocolo Modbus, que a programação mais parecida com o padrão de uma requisição Modbus é a melhor solução para aplicação do protocolo visto que apresenta a mesma estrutura analisada no capítulo 2.

Os testes efectuados às redes serviram para compreender melhor o modo de funcionamento dos protocolos. No caso do Modbus efectuaram-se testes para uma requisição *unicast* onde o servidor recebeu a requisição sem nenhum erro e retornou uma resposta normal. Em seguida foi realizado um teste em que o servidor recebeu a requisição mas não pode manipular e retornou uma *exception response*, tal como se esperava. Os testes seguintes foram a comunicação em *broadcast* onde nenhuma resposta é retornada, sendo a verificação de erros na rede feita analisando o bloco %MSG2.E e a *word* do sistema

%SW64.

Nos testes efectuados para o protocolo CAN foi possível verificar as tramas de dados com um identificador de 11 *bits* onde se analisou os seus respectivos campos, também se verificou os níveis de tensão do CAN H e CAN L, a alta prioridade e a baixa prioridade de uma trama, e ainda o *bit stuffing* e o *bit* de confirmação (*ACK slot*).

Na comunicação com o padrão OPC verificou-se a grande aplicabilidade deste padrão em permitir a interligação de dispositivos de diferentes fabricantes. Os controladores PI e PI com *anti windup* apresentaram um bom desempenho no controlo dos processos.

Concluindo, com os testes efectuados e os resultados obtidos mostra-se a grande importância dos protocolos no sector industrial, e entende-se a grande aplicação que estes protocolos têm a nível industrial e a nível de serviços.

5.2 Trabalho futuro

Nesta secção são apresentadas propostas para trabalhos futuros a realizar, para se dar continuidade aos estudos das redes industriais e dos protocolos em particular. Na indústria, muitas vezes, não se tem apenas um cliente e um servidor ou um produtor e um consumidor, tem-se quase sempre a necessidade de se ter vários dispositivos a comunicar. Por exemplo, o caso do protocolo CAN nos automóveis, quando se pressiona um botão para trancar as portas do carro, uma unidade de controlo da porta lê a entrada e transmite os comandos de bloqueio para as outras unidades de controlo das outras portas; estes comandos são enviados como dados na mensagem CAN.

Com esta visão é apresentado como trabalho futuro a inserção de mais um nó, com o objectivo de verificar o comportamento da rede CAN e permitir que três nós troquem mensagens. Para a proposta da arquitectura de trabalho futuro é necessário o Matlab® com *toolbox Vehicle Network* e um dispositivo CAN como por exemplo *CANcaseXL*. É importante que o dispositivo escolhido suporte a *toolbox Vehicle Network*. A arquitectura está representada na figura 5.1.

Continuando com a mesma visão de se ter vários nós numa arquitectura de uma rede industrial, a figura 5.2 mostra a segunda arquitectura para a realização de trabalho futuro. Trata-se de uma ligação entre a rede Modbus e uma rede CANopen com ajuda

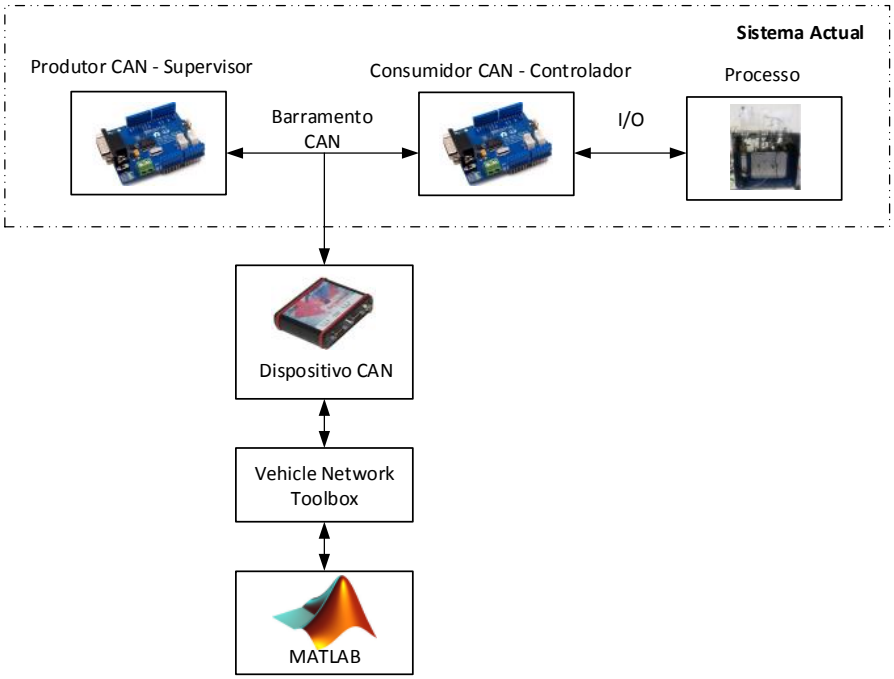


Figura 5.1: Arquitectura proposta para a inserção de mais um nó na rede CAN.

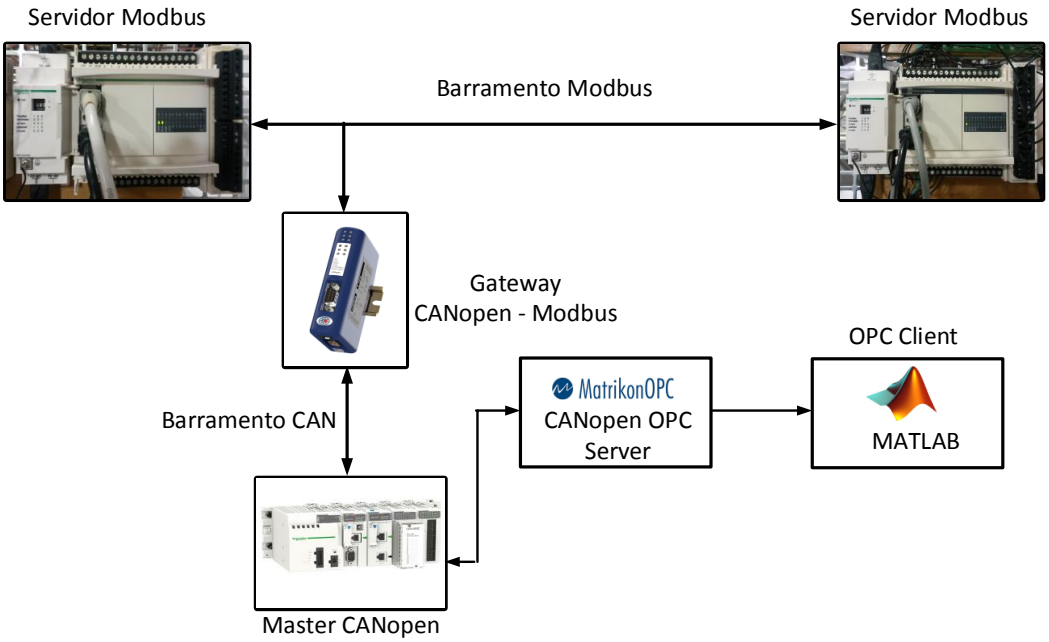


Figura 5.2: Arquitectura proposta para uma ligação entre Modbus e o CANopen.

de um *gateway* CANopen - Modbus actualmente disponível no mercado e a inserção do padrão OPC, em que o servidor OPC pode ser o *software* MatrikonOPC (CANopen OPC server) e como cliente OPC o Matlab®.

Ainda para trabalho futuro pretende-se que se desenvolva uma ferramenta de decisão para a escolha de uma rede industrial entre Modbus e CAN considerando a tabela 4.3.

As redes industriais em sistemas de automação e controlo são responsáveis por fábricas, sistemas de distribuição de electricidade, gás, centrais eléctricas, etc. Com o crescimento de guerras cibernéticas (guerra onde a conflitualidade não ocorre com armas físicas, mas através da confrontação com meios eletrónicos e informáticos) é de fundamental importância o estudo de técnicas de defesa dessas redes. Com isso é proposto como trabalho futuro um estudo de técnicas de ataque e defesa em redes industriais.

Bibliografia

[1] O. Pfeiffer, A. Ayre, and C. Keydel, *Embedded Networking with CAN and CANopen*, 1st ed. Greenfield, MA: Copperhill Technologies Corporation, November 2003.

[2] K. Etschberger, *Controller Area Network Basics, Protocols, Chips and Applications*, 2nd ed. Weingarten, Germany: IXXAT Automation GmbH, July 2001.

[3] Modbus-IDA, "Modbus application protocol specification v1.1b," Organização Modbus, Tech. Rep., December 2006.

[4] N. Rozas, "Aplicação do protocolo modbus em comunicação wireless," *Gás Brasil*, pp. 38–39, November 2004.

[5] L. Dongjiang and S. Ruiqi, "Implement of communication between configuration software and opc server based on modbus/tcp," in *Electronic Measurement Instruments (ICEMI), 2011 10th International Conference on*, vol. 1, Aug 2011, pp. 218–221.

[6] Modbus-IDA, "Modbus over serial line - specification and implementation guide v1.02," Organização Modbus, Tech. Rep., December 2006.

[7] Z. Li, S. Zhang, J. Lang, and H. Shao, "The application and research of the liquid level control technology used in mineral flotation process which based on the modbus communication protocol," in *Control and Decision Conference (CCDC), 2013 25th Chinese*, May 2013, pp. 3603–3606.

[8] J. Velagic, A. Kaknjo, N. Osmic, and T. Dzananovic, "Networked based control and supervision of induction motor using opc server and plc," in *ELMAR, 2011 Proceedings*, Sept 2011, pp. 251–255.

[9] C. in Automation, "Can specification version 2.0 part a," CAN in Automation, Erlangen, Germany, Tech. Rep., September 1991.

[10] CiA, "Can specification version 2.0 part b," CAN in Automation, Erlangen, Germany, Tech. Rep., September 1991.

[11] R. Boys, "Can: Controller area network introduction and primer," Dearborn Group, Inc., Tech. Rep., September 2004.

[12] A. Farahani, G. Latif-Shabgahi, and F. Tajarrod, "On the priority problem of can protocol: A new idea," in *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, vol. 2, June 2010, pp. V2–500–V2–505.

[13] U. do Porto, "Protocolo de comunicações CAN, disponível em http://paginas.fe.up.pt/~ee99058/projecto/pdf/can.pdf, [Online]; Consult: 2014-May-20."

[14] T. Presi, "Design and development of pic microcontroller based vehicle monitoring system using controller area network (can) protocol," in *Information Communication and Embedded Systems (ICICES), 2013 International Conference on*, Feb 2013, pp. 1070–1076.

[15] W. Bangji, L. Qingxiang, Y. Yi, Z. Liu, Z. Zhengquan, L. Xiangqiang, and Z. Jianqiong, "Development of can bus application layer protocol for beam control system," in *Intelligent Computation Technology and Automation (ICICTA), 2011 International Conference on*, vol. 2, March 2011, pp. 519–522.

[16] L. Guohuan, Z. Hao, and Z. Wei, "Research on designing method of can bus and modbus protocol conversion interface," in *BioMedical Information Engineering, 2009. FBIE 2009. International Conference on Future*, Dec 2009, pp. 180–182.

[17] G. Cena, I. Bertolotti, T. Hu, and A. Valenzano, "Design, verification, and performance of a modbus-can adaptation layer," in *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*, May 2014, pp. 1–10.

[18] K. J. Astrom and T. Hagglund, *Automatic Tuning of PID controllers*, 2nd ed. North Carolina: Library of Congress Cataloging-in-Publication Data, July 1995.

[19] C. T. ObdDiag.net, "Adapters for vehicle on-board diagnostic," Disponível em http://www.obddiag.net/adapter.html, [Online]; Consult: 2014-May-20.

[20] G. Thomas, *Introduction to Modbus Serial and Modbus TCP*, Contemporary Control Systems,Inc., Illinois, USA, September-October 2008.

[21] I. Modicon, "Modbus protocol reference guide," Organização Modbus, North Andover, Massachusetts, Tech. Rep., June 1996.

[22] A. Lugli and M. Santos, *Sistemas Filedbus para automação industrial Devicenet, CANopen, SDS e Ethernet*, primeira ed. São Paulo: Câmara Brasileira do Livro, SP, Brasil, 2010.

[23] I. S. I. 11898-1, "Road vehicles - controller area network (can) - part 1: Data link layer and physical signalling," International Standard Organization, Switzerland, Tech. Rep., 2003.

[24] C. Pinto, *Técnicas de Automação*, segunda ed., E. T. e Profissionais ETEP, Ed. Portugal: LIDEL - Edições Técnicas, Março 2004.

[25] O. Modbus, "About modbus organization," Disponível em http://www.modbus.org/about-us.php, [Online]; Consult: 2013-November-12.

[26] L. Alexandre and M. Santos, *Redes Industriais para automação industrial: AS-I, PROFIBUS e PROFINET*, primeira ed. São Paulo: Câmara Brasileira do Livro, SP, Brasil, 2010.

[27] A. B. Lugli and D. A. Lemes, "Estudo de padrões para configuração de instrumentos remotos em redes industriais," in *Transactions on Industrial Electronics*, vol. 57. IEEE, November 2010, pp. 3585–3595p.

[28] T. Sauter, "The three generations of field-level networks-evolution and compatibility issues," in *Transactions on Industrial Electronics*, vol. 57. IEEE, November 2010, pp. 3585–3595p.

[29] B. Galloway and G. Hancke, "Introduction to industrial control networks," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 2, pp. 860–880, Second 2013.

[30] J. Nascimento and P. Lucena, "Protocolo modbus," in *Protocolo Modbus*. Brasil: Universidade Federal do Rio Grande do Norte, Julho 2003.

[31] G. Strack, "Módulo de entrada e saída remoto modbus," Master's thesis, Universidade Federal do Rio Grande do Sul, Brasil, Julho 2011.

[32] C. Dressler and O. Fischer, "Can basics," in *CAN in Automation history and CAN*, November 2007.

[33] R. Li, C. Liu, and F. Luo, "A design for automotive can bus monitoring system," in *Vehicle Power and Propulsion Conference, 2008. VPPC '08. IEEE*, Sept 2008, pp. 1–5.

[34] O. CiA, "About can in automation," Disponível em http://www.can-cia.org/index.php?id=aboutcia, [Online]; Consult: 2014-August-7.

[35] R. B. GmbH, "Can specification version 2.0," Robert Bosch GmbH, Stuttgart, Germany, Tech. Rep., September 1991.

[36] O. S. Interconection, "International standard. road vehicles - controller area network," Open System Interconection, Switzerland, Tech. Rep., December 2003.

[37] W. Lawrenz, *CAN System Engineering From Theory to Practical Applications*, 2nd ed. London: Springer, July 2013.

[38] C. Organization, *CiA Draft Standard 102 version 2.0 - CAN Physical Layer for Industrial Applications*, CiA Organization, Nuremberg, Germany, February 2010.

[39] A. M. da Silva de Carvalho, "Integração de rede de telemetria em ambientes industriais." Master's thesis, Universidade de Trás-os-Montes e Alto Douro, Portugal.

[40] L. Wang, T. Zhang, K. Li, and B. Ren, "Study of can-modbus communication adapter for low-voltage distribution system," in *Mechatronic Sciences, Electric Engineering and Computer (MEC), Proceedings 2013 International Conference on*, Dec 2013, pp. 3500–3503.

[41] A. F. dos Santos Neto, "Aplicação do protocolo aberto opc e do foss scilab no desenvolvimento de um móulo laboratorial para controle de processos industriais." Master's thesis, Universidade Federal de Juiz de Fora, Brasil, 2013.

[42] A. Visioli, *Pratical PID - Advances in Industrial Control*, 2nd ed. London: Springer, July 2006.

[43] F. Golnaraghi and B. C. Kuo, *Automatic Control Systems*, 9th ed., I. John Wiley e Sons, Ed. Danvers, Massachussets: Don Fowley and Daniel Sayre, 2010.

[44] N. S. Nise, *Control Systems Engineering*, 6th ed., I. John Wiley e Sons, Ed. Danvers, Massachusetts: Don Fowley and Daniel Sayre, 2011.

[45] B. C. Kuo, *Automatic Control Systems*, 3rd ed., P. Hall, Ed. New Jersey: Prentice Hall, 1975.

[46] Ogata, *System Dinamics*, 4th ed., D. A. George and S. Disanno, Eds. New Jersey: Pearson Prentice Hall, 2004.

[47] K. Ogata, *Modern Control Engineering*, 5th ed., P. Hall, Ed. New Jersey: Pearson, 2010.

[48] F. OPC, "What is opc?" Disponível em https://opcfoundation.org/about/what-is-opc/, [Online]; Consult: 2014-August-13.

[49] E. Scilab, "History," Disponível em http://www.scilab.org/scilab/history, [Online]; Consult: 2014-August-18.

[50] A. Francisco, *Autómatos Programáveis*, segunda ed. Portugal: LIDEL - Edições Técnicas, Junho 2003.

[51] E. Pérez, J. Acevedo, C. Silva, and J. Quiroga, *Autómatas Programables y sistemas de automatización*, segunda ed., M. E. oes Técnicas, Ed., Espanha, Septiembre 2009.

[52] K. J. Astrom, M. Athans, J. Baillieul, R. R. Bitmead, P. Kokotovic, M. J. Piovoso, and W. J. Rugh, *The Control HandBook*, 1st ed., W. S. Levine, Ed. Florida: CRC Press HandBook, month 2000, vol. I.