

Comunicação de tempo-real em barramentos CAN baseados no controlador SJA1000

**Análise e implementação de uma solução
para o escalonamento de mensagens**

António Júlio Morais Pires

Licenciado em Engenharia de Automação e Controlo
pela Universidade Moderna do Porto

Dissertação submetida para satisfação parcial dos
requisitos do grau de mestre
em
Engenharia Electrotécnica e de Computadores
(Área de especialização Informática e Automação)

Faculdade de Engenharia da Universidade do Porto

Mestrado em Engenharia Electrotécnica e de Computadores
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal

Dezembro 2005

Dissertação realizada no âmbito do programa do
Mestrado em Engenharia Electrotécnica e de Computadores da
Faculdade de Engenharia da Universidade do Porto

Dissertação realizada sob a supervisão do
Professor Doutor Francisco Manuel Madureira e Castro Vasques de Carvalho
Professor Associado do Departamento de Engenharia Mecânica e Gestão Industrial
e Professor Doutor João Paulo Filipe de Sousa
Professor Auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da
Faculdade de Engenharia da Universidade do Porto

Resumo

Esta dissertação apresenta a análise e implementação de uma solução para o escalonamento de mensagens de tempo-real, em barramentos CAN baseados em controladores CAN Philips SJA1000. Esta solução pretende resolver alguns dos problemas existentes no que diz respeito à perda de metas temporais (“*deadlines*”) no envio de fluxos de mensagens, quando é usado o protocolo CAN na sua forma convencional.

O trabalho realizado no âmbito desta dissertação aborda, numa primeira fase o protocolo CAN e a forma como este é implementado; depois quais os tipos de interfaces, quer hardware quer ambiente de software de desenvolvimento comercial, que existem para este tipo de barramento.

Posteriormente foram analisados alguns dos vários tipos de algoritmos de escalonamento que podem ser aplicados no barramento CAN, os quais funcionaram como base para a realização das várias implementações de algoritmos de escalonamento para a análise experimental realizada.

Verificou-se que, através de dois tipos de escalonamento diferentes, um local e um global, se consegue melhorar as características de tempo-real no barramento CAN, e diminuir fortemente a perda de metas temporais de mensagens em sistemas de comunicação de tempo-real. O software que foi inicialmente desenvolvido, aplicando um escalonamento por prioridades local na fila, permitiu observar experimentalmente que a simples ordenação na fila dos fluxos de mensagens, conforme a sua periodicidade, permite melhorar em muito os resultados no envio de fluxos de mensagens prioritários para a rede, evitando perdas de metas temporais. Fazendo uma pequena alteração no software implementado, de forma a poder incluir o cálculo das metas temporais, obteve-se um escalonamento que permitiu baixar a quantidade de metas temporais perdidas também nos fluxos de mensagens menos prioritárias.

Desta forma, o objectivo a que tinha sido proposto atingir esta dissertação, o qual era fazer o desenvolvimento de um pequeno software que melhorasse a eficácia do envio de fluxos de mensagens de tempo-real através de um barramento com o uso do protocolo CAN, foi alcançado, demonstrando-se assim que o barramento CAN ainda pode ter muitas mais potencialidades do que as que já são usadas, e pode ser bastante melhorado.

Abstract

This thesis presents the analysis and implementation of a solution for the scheduling of real-time messages in a CAN bus, based on CAN controllers Philips SJA1000.

This solution intends to solve some of the problems related to the loss of deadlines in messages streams, when the CAN protocol is used in its conventional setup.

It was verified that, using two different scheduling approaches, one local and one global, it is possible to improve the real-time characteristics of the CAN bus, and to strongly reduce the loss of deadlines for the real-time message streams.

The work presented this thesis is structured as follows: in a first phase, we present the CAN protocol and how it can be used; later we describe the available type of interfaces, commercial hardware and development software. Afterwards, we present a theoretical analysis of real-time communication in the CAN bus.

The experimental analysis was carried out considering the maximum load that can be placed in the network, to be possible to verify the maximum values of schedulability in each node. It is important to say that this kind of load model contains sufficiently more messages than a real load model, allowing us to get experimental results for the use in extreme situations.

All the software developed during the experimental analysis was tested successfully in the hardware testbed, validating the results we have got.

Agradecimentos

Ao longo do meu trabalho, alguns foram os que contribuíram com a sua ajuda e motivação, sem as quais o presente trabalho não teria sido possível.

A realização desta Dissertação de Mestrado contou em primeiro lugar com a preciosa ajuda e orientação do Professor Francisco Vasques e do Professor João Paulo Sousa, que com a sua dedicação, disponibilidade, enorme experiência, constante motivação, numerosas sugestões e críticas, foram uma mais valia para que esta dissertação, marco importante na minha vida académica, se tornasse uma realidade. Por tudo isto, Professores – Bem-hajam.

Quero também agradecer ao Departamento de Engenharia Mecânica e Gestão Industrial da Faculdade de Engenharia da Universidade do Porto – FEUP, pela cedência do laboratório onde desenvolvi o meu trabalho.

A todos aqueles que de alguma forma contribuíram para a realização deste trabalho, quero deixar aqui os meus sinceros agradecimentos.

Finalmente, quero agradecer à minha família e amigos, o apoio e a motivação que sempre me deram desde a primeira hora e por terem compreendido os vários momentos em que não pude estar presente.

António Júlio Morais Pires

Índice

Resumo	v
Abstract	vii
Agradecimentos	ix
Índice	xi
Lista de Figuras	xiv
Lista de Tabelas	xvi
Capítulo 1 - Introdução.....	3
1 - Problema.....	3
2 - Objectivos.....	3
3 - Estrutura da dissertação.....	4
Capítulo 2 - O Protocolo CAN.....	9
1 - Introdução.....	9
2 - O protocolo.....	11
2.1 - Transmissão de tempo-real.....	12
2.2 - Os formatos das tramas das mensagens.....	13
2.3 - Detecção e sinalização de erros.....	17
3 - Opções de camadas físicas para o protocolo CAN.....	21
3.1 - Camada física de alta velocidade	21
3.2 - Camada física de baixa velocidade tolerante a falhas	23
4 - Protocolos de camada elevada	24
5 - Síntese	25
Capítulo 3 - Hardware para as ligações à rede.....	29
1 - Controladores para rede CAN.....	29
1.1 - Funções do controlador CAN.....	30
1.2 - Pedidos remotos de mensagem.....	34
1.3 - Áreas de aplicação dos controladores BasicCAN e FullCAN.....	34
1.4 - Controladores CAN autónomos	34
1.5 - Microcontroladores com controladores CAN integrados.....	43
2 - As interfaces.....	48
3 - Os módulos.....	51
3.1 - KEIL Software.....	52
3.2 - Os Módulos SBC da Phyttec	54
4 - Síntese	57

Capítulo 4 - Análise teórica de comunicações de tempo-real em redes CAN	61
1 - Introdução.....	61
2 - Características de carga na rede.....	61
2.1 - Mensagens periódicas.....	62
2.2 - Mensagens esporádicas	62
2.3 - Mensagens não tempo-real	63
2.4 - Mensagens de tempo-real: alta vs baixa velocidade.....	63
3 - Notação e modelo de comunicação.....	63
4 - Teoria de escalonamento básico da rede CAN num único processador	64
5 - Análise do CAN ideal	65
6 - Comportamento real do SJA1000	67
7 - Síntese	69
Capítulo 5 - Algoritmos de escalonamento utilizados na análise	73
1 - Introdução.....	73
2 - Escalonamentos analisados	73
2.1 - Escalonamento por prioridades fixas – baixa utilização	74
2.2 - Escalonamento por prioridades dinâmicas – Earliest Deadline First e os problemas de codificação das metas temporais.....	74
2.3 - MTS – Escalonamento por Tráfego Misto	77
2.4 - Condições de escalonamento	79
3 - MTS em CAN.....	83
3.1 - Sobrecargas.....	84
4 - Síntese	85
Capítulo 6 - Análise experimental do escalonamento de tempo-real de mensagens em CAN	89
1 - Introdução.....	89
2 - A escolha das cargas e das placas.....	89
3 - Descrição das diferentes estratégias de escalonamentos independentes implementados	93
3.1 - Algoritmo 1 – Sem escalonamento, filas FIFO em cada nó (CAN convencional)	93
3.2 - Algoritmo 2 – Escalonamento por prioridades local	99
3.3 - Algoritmo 3 – Escalonamento por prioridades local e épocas temporais.....	103
4 - Síntese	106
Capítulo 7 - Conclusões.....	111
Bibliografia e referências	115

Anexo	119
1 - Introdução.....	119
2 - Funções desenvolvidas para o software.....	119
2.1 - Geração das mensagens	119
2.2 - Controlo dos fluxos	122
2.3 - Contabilização das metas temporais perdidas	123
2.4 - Inicializações	124
2.5 - Sem escalonamento, filas FIFO em cada nó (CAN convencional)	127
2.6 - Escalonamento por prioridades local.....	130
2.7 - Escalonamento por prioridades local e épocas temporais	135
3 - Listagens completas do software desenvolvido.....	139
3.1 - Sem escalonamento, filas FIFO em cada nó (CAN convencional)	139
3.2 - Escalonamento por prioridades local.....	149
3.3 - Escalonamento por prioridades local e cálculo de épocas.....	159

Lista de Figuras

Figura 1 – Exemplo de equipamentos electrónicos num veículo	9
Figura 2 – Modo como se ligavam os diferentes equipamentos.....	9
Figura 3 – Rede CAN aplicada num veículo	10
Figura 4 – Evolução do protocolo CAN ao longo dos tempos.....	10
Figura 5 – Difusão numa rede CAN.....	11
Figura 6 – Aplicação do princípio wired-AND e resolução de conflitos por “ <i>bit-wise</i> ”	12
Figura 7 – Formato de uma trama de mensagem CAN	13
Figura 8 – Comparação entre a trama de dados e a trama de pedido remoto de dados..	15
Figura 9 – Trama de erro	16
Figura 10 – Trama de sobrecarga	16
Figura 11 – Espaço entre tramas.....	17
Figura 12 – Detecção de erros CRC	18
Figura 13 – Detecção de erros ACK.....	18
Figura 14 – Campos da trama com formatos fixos.....	19
Figura 15 – Aplicação do “ <i>bit stuffing</i> ” no protocolo CAN	19
Figura 16 – Gestão de erros no protocolo CAN	20
Figura 17 – Exemplo de uma ligação física de uma rede CAN	22
Figura 18 – Estados no barramento do protocolo CAN	22
Figura 19 – Estado “recessivo” no barramento	23
Figura 20 – Estado “dominante” no barramento	23
Figura 21 – Modelo ISO/OSI para o protocolo CAN.....	24
Figura 22 – Princípio de funcionamento do BasicCAN	32
Figura 23 – Princípio de funcionamento do FullCAN	32
Figura 24 – Pior caso na condição de tempo-real de um controlador BasicCAN.....	33
Figura 25 – Nó CAN baseado num controlador autónomo	35
Figura 26 – Diagrama de blocos do funcionamento do PHILIPS SJA1000	37
Figura 27 – Adaptador de barramento integrado no PHILIPS SJA1000	38
Figura 28 – Exemplo do armazenamento de uma mensagem no RxFIFO.....	39
Figura 29 – Diagrama de blocos do funcionamento do INTEL 82527	41
Figura 30 – Nó CAN baseado num microcontrolador com controlador CAN integrado	44
Figura 31 – Diagrama de blocos de funcionamento do MICROCHIP PIC18F258	46
Figura 32 – Diagrama de blocos do funcionamento do PHILIPS XA-C3	47
Figura 33 – IXXAT Automation iPC-I165/PCI	50
Figura 34 – IXXAT Automation iPC-I320/PCI	51
Figura 35 – JANZ CAN-PCI.....	51
Figura 36 – KEIL Software MCB167-NET Evaluation Board	53
Figura 37 – Diagrama de blocos da placa MCB167-NET.....	53
Figura 38 – PHYTEC phyCORE ADuC812	55

Figura 39 – Diagrama de blocos da phyCORE ADuC812	56
Figura 40 – PHYTEC phyCORE XA-C3	56
Figura 41 – Diagrama de blocos do phyCORE XA-C3	57
Figura 42 – Modelo das temporizações para mensagens na rede CAN	64
Figura 43 – Gráfico relativo à utilização do cenário-exemplo de Tindell [TH, 94].....	69
Figura 44 – Estrutura do identificador para EDF	74
Figura 45 – Maior valor possível para o campo da meta temporal	76
Figura 46 – Estrutura do identificador para MTS de fluxos de mensagens	77
Figura 47 – Quantificação das metas temporais relativamente ao início de época	78
Figura 48 – Estrutura dos identificadores para MTS de fluxos de mensagens.....	79
Figura 49 – Exemplo da pior codificação de meta temporal possível.....	82
Figura 50 – Tramas de mensagens CAN	90
Figura 51 – Trama de mensagem CAN observada num osciloscópio.....	90
Figura 52 – Princípio da fila circular.....	94
Figura 53 – Excerto do fluxograma para o escalonamento em fila FIFO	94
Figura 54 – Princípio da fila circular para o escalonamento em fila FIFO	95
Figura 55 – Comportamento da rede com o escalonamento em fila FIFO.....	96
Figura 56 – Gráfico dos resultados da experimentação do algoritmo em fila FIFO	97
Figura 57 – Relação entre as mensagens geradas pelo fluxo que tem mais prioridade e o que tem menos prioridade	98
Figura 58 – Princípio da fila circular para o escalonamento local	100
Figura 59 - Excerto do fluxograma para o escalonamento local da fila.....	100
Figura 60 – Comportamento da rede com o escalonamento local da fila.....	102
Figura 61 – Gráfico dos resultados da experimentação do algoritmo de escalonamento por prioridades local	103
Figura 62 – Excerto do fluxograma para o escalonamento local e cálculo de épocas .	104
Figura 63 – Gráfico dos resultados da experimentação do algoritmo de escalonamento por prioridades local com cálculo de épocas	106
Figura 64 – Fluxograma da interrupção periódica por timer.....	120
Figura 65 – Fluxograma da função transCAN() base.....	121
Figura 66 – Fluxograma da inicialização do microcontrolador.....	126
Figura 67 – Fluxograma da inicialização do controlador CAN SJA1000.....	127
Figura 68 – Fluxograma da função de base para enviar para o controlador.....	128
Figura 69 – Fluxograma da função main() para o primeiro algoritmo	130
Figura 70 – Fluxograma da função transCAN(), que coloca na fila circular	131
Figura 71 – Fluxograma da interrupção periódica com o valor de timeout	132
Figura 72 – Fluxograma da função abortar(), que aborta mensagens no controlador..	133
Figura 73 – Fluxograma da função main() usada para segundo algoritmo	135
Figura 74 – Configuração do identificador de 11 bits duma mensagem em CAN.....	136
Figura 75 – Fluxograma da função enviar() com o cálculo por épocas	138

Lista de Tabelas

Tabela 1 – Tabela comparativa dos controladores de protocolo CAN autónomos	36
Tabela 2 – Formato da mensagem no buffer em modo BasicCAN.....	37
Tabela 3 – Mapa dos registos de estado e controlo no modo BasicCAN.....	38
Tabela 4 – Mapa de registos do buffer de mensagens do INTEL 82527	43
Tabela 5 – Tabela comparativa de microcontroladores com controladores de protocolo CAN integrados	45
Tabela 6 – Tabela comparativa das interfaces CAN para PC.....	49
Tabela 7 – Tabela comparativa dos módulos CAN para nós	52
Tabela 8 – Resultados obtidos do cenário-exemplo de Tindell [TH, 94].....	69
Tabela 9 – Valores obtidos para C_i e T_i , conforme a carga total pretendida na rede.....	92
Tabela 10 – Tabela de quantidade de mensagens de cada fluxo, conforme a carga total aplicada na rede	93
Tabela 11 – Resultados da experimentação do algoritmo em fila FIFO	97
Tabela 12 – Resultados da experimentação do algoritmo de escalonamento por prioridades local	102
Tabela 13 – Resultados da experimentação do algoritmo de escalonamento por prioridades local com cálculo de épocas	105

Capítulo 1

Introdução

Capítulo 1 - Introdução

1 - Problema

Actualmente para efectuar a interligação de sistemas de controlo, sensores electrónicos e entre outros sistemas de automação ou sistemas inteligentes, quer em instalações móveis quer em instalações fixas, é o “estado da arte” usar barramentos de comunicação especificamente desenvolvidos para esta área de aplicação, tal como o barramento de campo (ou “*fieldbus*”).

Muitas destas soluções de barramento começaram por ser especificadas para aplicações específicas mas depressa foram usadas noutras aplicações para as quais não estavam inicialmente destinadas.

Em particular falámos da Controller Area Network ([Law, 97] e [Ets, 2001]), barramento do tipo série, o qual inicialmente foi especificado para o uso na indústria automóvel, mas depressa passou a ser utilizado noutras indústrias (medicina, naval, fabril), onde tem apresentado bons resultados.

Contudo, este tipo de barramento, quando usado na sua forma convencional possui alguns problemas no que diz respeito à perda de metas temporais (“*deadlines*”) no envio de fluxos de mensagens, como poderá ser verificado mais tarde durante a análise experimental efectuada no âmbito deste trabalho.

2 - Objectivos

Assim, propor uma solução simples e eficaz que contribua para a resolução de alguns dos problemas detectados é a proposta fundamental desta tese. A nossa proposta é baseada na implementação eficaz de estratégias de escalonamento locais e globais, para melhorar as características de tempo-real do barramento CAN. Verificou-se que, através de dois tipos de escalonamento diferentes, um local e um global, se consegue melhorar estas características, e diminuir fortemente a perda de metas temporais de mensagens em sistemas de comunicação de tempo-real.

Assim, é o objectivo principal desta dissertação demonstrar que, utilizando apenas hardware e software de desenvolvimento existentes no nosso mercado, sem efectuar alterações no hardware e aplicando os limites de carga máxima possíveis de utilização na rede, é possível melhorar fortemente as características de comunicação de tempo-real do protocolo CAN, através da implementação de um pequeno módulo de software que implemente um escalonamento de tempo-real adequado para transmissão das mensagens. Este módulo de software deverá ser de fácil integração em outros projectos já existentes ou poder-se-á desenvolver todo um projecto à sua volta.

Em vez de ser realizada a análise experimental com tráfego real, vai ser considerada a carga máxima que pode ser colocada na rede, de forma que seja possível verificar os valores máximos de escalonabilidade em cada nó. É de referenciar que este modelo de carga contém bastantes mais mensagens que um modelo de carga real, permitindo obter assim resultados experimentais para situações extremas de utilização, que de outro modo não seriam possíveis.

Por forma a atingir este objectivo foi efectuado um estudo aprofundado do protocolo CAN, da sua implementação prática, dos vários interfaces que existem actualmente para a sua implementação (quer em nós independentes, quer para ligação a computadores) e principalmente qual dos métodos de escalonamento mais usados ou um seu derivado se adequa para este protocolo, tendo em conta quais os tipos de mensagens que possam surgir, como sejam mensagens sem problemas de metas temporais ou mensagens de tempo-real com metas temporais críticas.

Os algoritmos de escalonamento por que se optou como base para a análise experimental foram os algoritmos Earliest Deadline First [ZS, 95] e Mixed Traffic Scheduling [ZS, 95].

3 - Estrutura da dissertação

Esta dissertação encontra-se estruturada em sete capítulos, bibliografia e referência e um anexo, da seguinte forma:

- Capítulo 1 - Apresenta uma introdução à dissertação, descreve quais são os seus objectivos e como esta se encontra estruturada.
- Capítulo 2 - Apresenta uma descrição detalhada de como funciona o protocolo CAN, assim como alguns dos vários chips que utilizam o mesmo. No que se refere a estes últimos será feita apenas a descrição base, remetendo o seu estudo mais aprofundado para a consulta das respectivas folhas de características e de programação.
- Capítulo 3 - Como o protocolo CAN é implementado numa rede, neste capítulo apresentam-se alguns dos interfaces existentes a nível comercial para a mesma, quer do tipo “*embedded*” (módulos), quer do tipo placas de expansão para computador.
- Capítulo 4 - Neste capítulo é efectuada a análise teórica das comunicações de tempo-real em redes CAN, apresentando-se os vários tipos de mensagens que podem surgir, assim como a análise do barramento CAN ideal.
- Capítulo 5 - Descreve os vários algoritmos de escalonamento que foram utilizados no desenvolvimento deste trabalho de dissertação, tais como o Deadline Monotonic (DM), o Earliest Deadline First (EDF) ou o Mixed Traffic Scheduling (MTS), com as respectivas equações matemáticas de estudo.
- Capítulo 6 - Apresenta os testes e o mecanismo usado para o desenvolvimento da componente prática da dissertação, fazendo-se uma breve descrição do software desenvolvido, mas remetendo a explicação do seu desenvolvimento para anexo.
- Capítulo 7 - Apresenta as conclusões a que se chegou com o desenvolvimento deste trabalho e as respectivas justificações. Também apresenta algumas sugestões para trabalho futuro.

- Bibliografia e referências - Apresenta toda a bibliografia consultada e de referência, incluindo alguns artigos e endereços de páginas web, usada para o desenvolvimento desta dissertação.
- Anexo - Para finalizar, são apresentados em anexo os fluxogramas completos dos programas da análise experimental, o modo como foram realizados e as respectivas listagens.

Capítulo 2

O Protocolo CAN

Capítulo 2 - O Protocolo CAN

1 - Introdução

Com a implementação de cada vez mais equipamentos electrónicos nos veículos automóveis modernos, também surgiu o desenvolvimento do CAN. Alguns exemplos destes equipamentos incluem sistemas de gestão de motor, suspensão activa, ABS, *airbags*, entre outros, como se pode ver pela Figura 1.

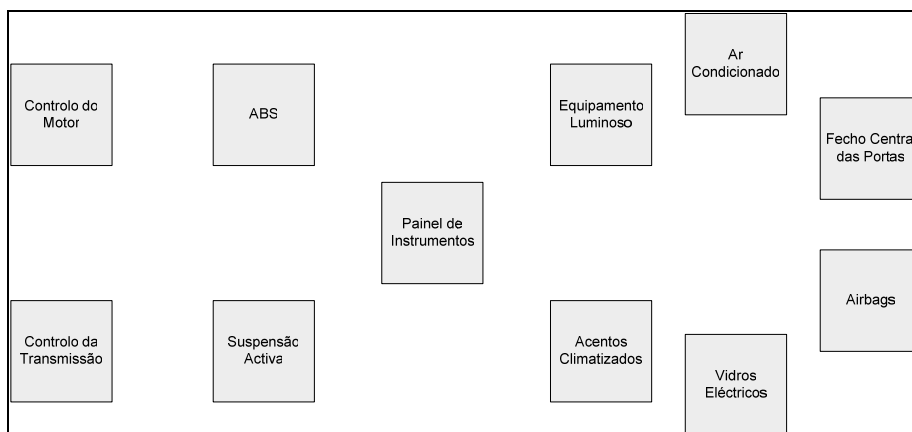


Figura 1 – Exemplo de equipamentos electrónicos num veículo

Isto levou a uma redução de combustível e de poluição e a um maior conforto por parte dos passageiros. Para concretizar estes objectivos e melhorar ainda mais o comportamento do veículo, era necessário que diversos sistemas de controlo pudessem trocar informações entre si. Normalmente isto seria realizado com interligações discretas entre os diversos sistemas, tipo ponto-a-ponto (*“point-to-point”*), como pode ser observado na Figura 2.

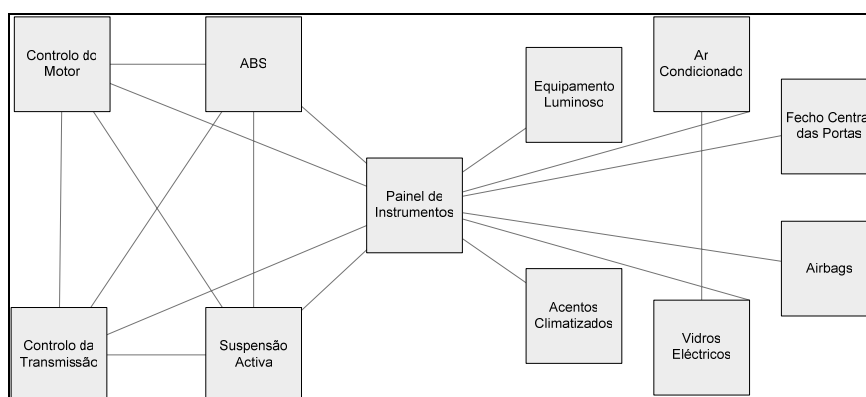


Figura 2 – Modo como se ligavam os diferentes equipamentos

Contudo, o cumprimento deste requisito levou a que o comprimento da rede se tornasse demasiado elevado, assim como a quantidade de fichas necessárias às ligações.

A solução surgiu com a ligação dos sistemas de controlo a um único sistema de barramento série: o barramento CAN.

Em Fevereiro de 1986, Robert Bosch introduziu o sistema de barramento série Controller Area Network – CAN, no congresso da Society Automotive Engineers (SAE) [Ets, 2001]. Esta data é considerada como sendo o nascimento de um dos maiores sucessos em protocolos de rede. Como a implementação principal deste barramento era em veículos, tornou-se claro que teria de cumprir certos requisitos especiais. De modo a fazer a ligação dos sistemas de controlo ao barramento, foi necessário dotar os mesmos de algum hardware específico CAN que fornecesse as regras ou protocolo para transmitir e receber informação pelo barramento, tal como se pode observar pela Figura 3.

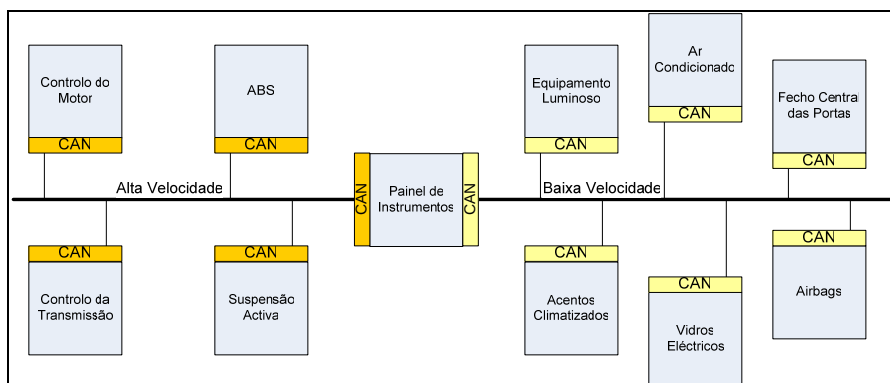


Figura 3 – Rede CAN aplicada num veículo

Actualmente, todos os automóveis de passageiros recentes têm implementada pelo menos uma rede CAN, embora também possa ser implementada em outros tipos de veículos. A Figura 4 apresenta a evolução do protocolo CAN até à actualidade mais próxima.

1983	Start of the Bosch internal project to develop an in-vehicle network
1986	Official introduction of CAN protocol
1987	First CAN controller chips from Intel and Philips Semiconductors
1991	Bosch's CAN specification 2.0 published
1991	CAN Kingdom CAN-based higher-layer protocol introduced by Kvaser
1992	CAN in Automation (CiA) international users and manufacturers group established
1992	CAN Application Layer (CAL) protocol published by CiA
1992	First cars from Mercedes-Benz used CAN network
1993	ISO 11898 standard published
1994	1st international CAN Conference (iCC) organized by CiA
1994	DeviceNet protocol introduction by Allen-Bradley
1995	ISO 11898 amendment (extended frame format) published
1995	CANopen protocol published by CiA
2000	Development of the time-triggered communication protocol for CAN (TTCAN)

Figura 4 – Evolução do protocolo CAN ao longo dos tempos (fonte: <http://www.cia-can.com>)

A especificação de protocolo CAN de Robert Bosch, já na sua versão 2.0, foi submetida para norma internacional no início da década de 90, tendo-se publicado em Novembro de 1993 a norma ISO 11898 referente às especificações do Protocolo CAN, não sem antes haver diversas disputas políticas principalmente com alguns dos maiores fabricantes franceses que haviam desenvolvido a Vehicle Area Network – VAN [Ets, 2001]. Em complemento do protocolo CAN, também foi definida a camada física para velocidades de 1Mbps. A transmissão de dados com tolerância a falhas via protocolo CAN tornou-se padrão pela concretização da norma 11519-2. Em 1995 a norma ISO 11898 foi complementada com uma adenda, a qual descreve o identificador de 29 bits para as mensagens do protocolo CAN.

Contudo, uma vez que as especificações e normas referentes ao protocolo CAN e publicadas até agora possuem erros ou estão de alguma forma incompletos e também por forma a evitar alguma incompatibilidade entre as diversas implementações existentes do protocolo CAN, Robert Bosch tomou algumas precauções para que todos os circuitos integrados para CAN seguissem o seu modelo de referência e princípio do protocolo CAN.

Embora o protocolo CAN tenha quase 20 anos, ainda hoje é largamente usado e melhorado. Como o protocolo em si ainda não foi alterado, é possível transmitir mensagens quer em “*time trigger*” quer em “*event trigger*”, através do mesmo sistema físico do barramento. A extensão TTCAN – Trigger-Timed CAN, vem adicionar mais cinco a dez anos ao tempo de vida do protocolo CAN.

2 - O protocolo

Sendo um protocolo baseado em múltiplos mestres (“*multimaster*”) e difusão (“*broadcast*”), o protocolo CAN é orientado à mensagem, pelo que não há definição dos nós nem dos endereços, apenas à mensagem. Estas mensagens são identificadas pelo uso de um Identificador de Mensagem (ID), o qual será único em toda a rede. Este identificador não define apenas o produtor da mensagem, mas também a prioridade da mensagem, o que é de grande importância em casos que existe competição pelo barramento. O modo como a difusão da mensagem se processa pode ser observado na Figura 5.

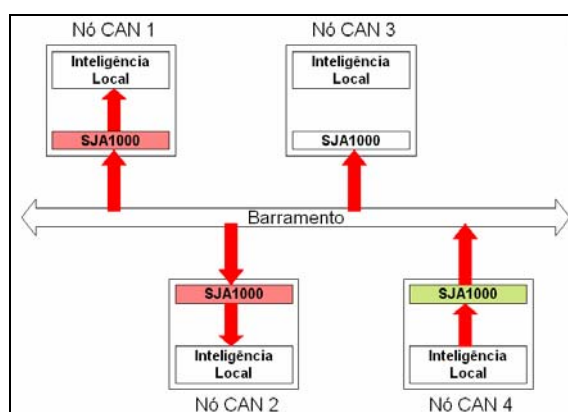


Figura 5 – Difusão numa rede CAN

Com estas características atinge-se um grau de flexibilidade tal que é muito fácil adicionar nós a uma rede com protocolo CAN já existente, sem efectuar modificações ao nível do hardware ou do software nos nós já colocados, desde que os novos nós sejam apenas utilizados para recepção. Isto leva a que se consiga obter uma múltipla recepção e sincronização de processos distribuídos: quaisquer dados que possam ser necessários para vários nós, podem ser enviados através da rede de tal modo que não é necessário saber quem é o transmissor por parte dos nós. Também se pode ter um mais fácil serviço de actualizações de rede, uma vez que nenhuma das transmissões é baseada na disponibilidade de alguns tipos específicos de nós.

2.1 - Transmissão de tempo-real

Dependendo do tipo de aplicação que se tem em uso, existem mensagens que são mais urgentes que outras, obrigando à existência de um esquema de prioridades. Estas prioridades fazem parte do identificador de cada mensagem e são atribuídas no momento da concepção do sistema, não havendo maneira prática de fazer mudanças dinamicamente.

Desta forma, o protocolo CAN gere os acessos ao barramento de acordo com o conceito *Carrier Sense Multiple Access* – CSMA, com arbitragem na prioridade da mensagem. Este conceito de arbitragem evita colisões em mensagens cujos nós iniciaram simultaneamente a transmissão e certifica-se de que a mensagem mais importante é transmitida em primeiro lugar sem perdas significativas de tempo.

Na Figura 6 a) pode-se ver a activação do sinal dos pinos de transmissão (TR activado) em três nós diferentes no barramento, designados nó A, nó B e nó C, assim como o estado do barramento segundo o princípio do “*wired-AND*”, onde o 0 é o estado dominante e o 1 é o estado recessivo (Figura 6 b)).

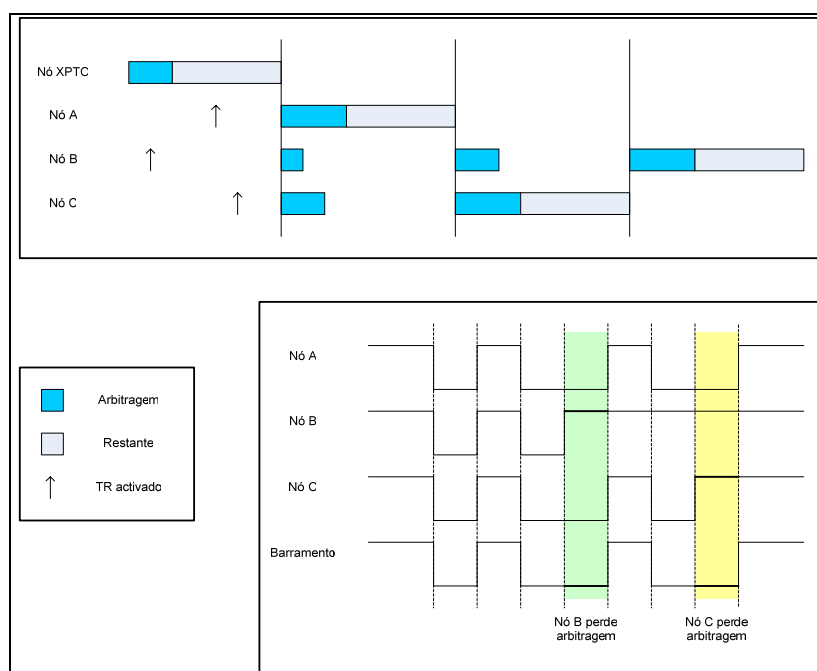


Figura 6 – Aplicação do princípio wired-AND e resolução de conflitos por “bit-wise”

Se dois ou mais nós do barramento iniciarem a transmissão ao mesmo tempo, após terem verificado que o barramento estava em espera, a colisão entre as mensagens é evitada pela arbitragem “*bit-wise*”. Cada nó envia os bits do identificador e fica à escuta dos níveis do barramento. Num determinado instante os nós A e C enviam um bit dominante do identificador. O nó B enviou um bit recessivo mas leu no barramento um bit dominante, pelo que perdeu a arbitragem e o direito ao barramento, passando ao modo de recepção. Alguns bits depois o mesmo sucede ao nó C. Isto significa que o identificador da mensagem do nó A tem um valor binário mais baixo e por isso possui maior prioridade que as mensagens dos nós B e C. Desta forma o nó no barramento que tem maior prioridade ganha a arbitragem sem ter que repetir a mensagem, não perdendo assim tempo. Este é o funcionamento do conceito de arbitragem “*bit-wise*”.

Assim que o barramento voltar a ficar em espera, os nós B e C tentam automaticamente transmitir as suas mensagens. Desta vez o nó B cede perante o nó C, pelo que será a mensagem deste a ser transmitida, seguida pela mensagem do nó B. Assim, a prioridade mais alta é definida como tendo o identificador com o valor binário mais baixo.

Não é permitido a nós diferentes enviarem mensagens com o mesmo identificador, uma vez que podem levar a uma situação de erro.

2.2 - Os formatos das tramas das mensagens

Actualmente existem dois tipos de formato de trama de mensagens suportados pelo protocolo CAN, designados de 2.0A e 2.0B, sendo a única diferença real entre eles o tamanho do identificador. Na versão 2.0A o identificador da mensagem possui 11 bits, enquanto que na versão 2.0B o mesmo identificador possui 29 bits, sendo também conhecido como formato estendido. Na Figura 7 apresentam-se ambos os formatos existentes das mensagens destinadas ao protocolo CAN:

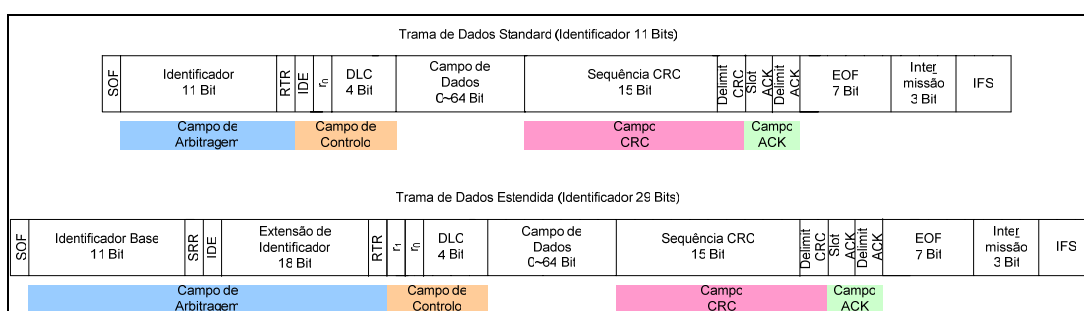


Figura 7 – Formato de uma trama de mensagem CAN: standard (em cima) e estendido (em baixo)

Para além das tramas de dados, que serão especificadas já de seguida, existem mais três:

- Trama de pedido remoto
- Trama de erro
- Trama de sobrecarga

2.2.1 - Trama de dados de formato standard

A trama de dados é gerada pelo nó CAN que pretende transmitir dados. Como se pode observar pela Figura 7, uma mensagem no protocolo CAN é sempre iniciada pelo bit ‘Início de Trama’ (SOF – “*Start of Frame*”), imediatamente seguido pelo ‘Campo de Arbitragem’ (“*Arbitration Field*”). Este campo é constituído pelo próprio ‘Identificador’ (“*Identifier*”) e pelo bit ‘Pedido Remoto de Transmissão’ (RTR – “*Remote Transmission Request*”). Será este bit que distinguirá um pedido remoto de transmissão de um pedido e envio de dados. O ‘Campo de Controlo’ (“*Control Field*”) que se segue contém o bit ‘Extensão do Identificador’ (IDE – “*Identifier Extension*”), o qual serve para efectuar a distinção entre a trama do formato standard e a trama de formato estendido, assim como o ‘Código do Comprimento de Dados’ (DLC – “*Data Length Code*”), o qual indica a quantidade de bytes que o ‘Campo de Dados’ (“*Data Field*”) contém. Nos casos em que a mensagem é um pedido remoto, este campo (DLC) indicará a quantidade de dados pedida. Entre o bit IDE e o DLC encontra-se um bit reservado para uso possível no futuro, caso haja necessidade de se acrescentar algum pormenor em falta a esta trama.

O campo de dados pode conter até 8 bytes de dados. A integridade dos dados é assegurada pela ‘Verificação Cíclica de Redundância’ (CRC – “*Cyclic Redundancy Check*”). O ‘Campo de Recibo’ (ACK – “*ACKnowledge Field*”) é constituído pelo ‘Delimitador ACK’ e pelo bit ‘Slot ACK’. O bit ‘Slot ACK’ é enviado como recessivo, sendo sobreposto pelos receptores como dominante, desde que recebam os dados correctamente. Seja qual for o resultado do filtro de aceitação, desde que a mensagem seja recebida correctamente, os receptores acusam a recepção com este bit.

A mensagem termina com a indicação ‘Fim de Trama’ (EOF – “*End of Frame*”) que corresponde a 7 bits recessivos consecutivos. O último campo corresponde ao ‘Espaço Entre Tramas’ (IFS – “*InterFrame Space*”), o qual indica o espaço entre duas tramas de mensagens consecutivas. Este campo inclui 3 bits obrigatoriamente recessivos designados de ‘Intermissão’ (“*Intermission*”), durante os quais não pode haver nenhuma trama, excepto se estiver sinalizado como sendo de erro.

Se não ocorrer nenhuma mensagem a seguir, o barramento manter-se-á em ‘Estado Livre’ (“*Idle*”).

2.2.2 - Trama dados de formato estendido

A principal diferença entre a trama de formato standard e a trama de formato estendido situa-se precisamente no tamanho do identificador, tal como pode ser observado pela Figura 7, que possui 29 bit distribuídos por 11 bits na mesma posição que a trama do formato standard e 18 bits de extensão situados depois do bit IDE, o qual indica a presença ou não desta extensão. Desta forma, a primeira parte designa-se por ‘Identificador Base’ e a segunda parte por ‘Extensão do Identificador’. Da mesma forma que na trama de dados standard, nesta trama também temos bits reservados para uso futuro, nomeadamente r0 e r1, os quais se encontram entre RTR e DLC. É de referenciar que nesta trama o RTR só surge depois da extensão do identificador, sendo substituído na sua posição normal (a seguir ao identificador base) pelo ‘Pedido Remoto Substituto’ (SRR – “*Substitute Remote Request*”).

Como ambos os formatos tem de coexistir no mesmo barramento, em caso de conflito no acesso ao mesmo está convencionado no próprio protocolo que o formato standard tem sempre prioridade sobre o formato estendido, devido ao bit IDE que também é contado na arbitragem, para além do bit RTR.

2.2.3 - Trama de pedido remoto de dados

Geralmente a transmissão de dados é efectuada de uma forma autónoma com a fonte de dados a enviar uma trama de dados. Mas também é possível um nó de destino requerer os dados do nó fonte, enviando um pedido remoto de dados. Existem duas diferenças essenciais entre a trama de dados e a trama de pedido remoto de dados. A primeira é que na trama de dados o bit RTR é transmitido como dominante e segundo a trama de pedido remoto não possui um campo de dados propriamente dito, isto é, neste campo apenas é especificado a quantidade de dados pedida. Um exemplo desta trama pode ser observada na Figura 8.

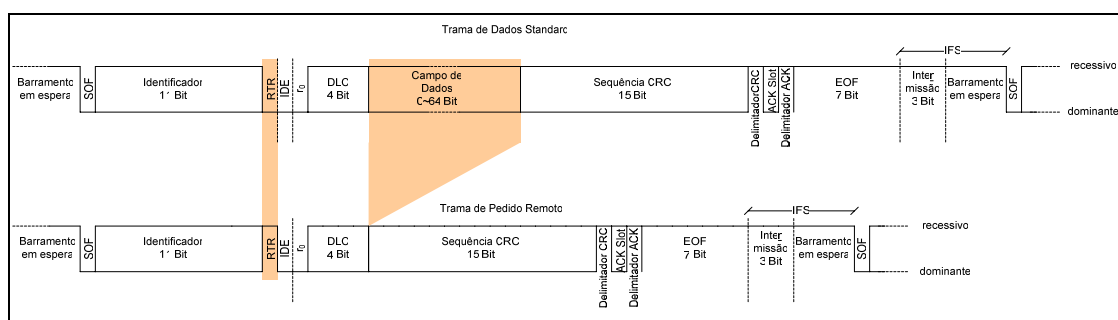


Figura 8 – Comparação entre a trama de dados e a trama de pedido remoto de dados

Na eventualidade improvável da trama de dados e da trama de pedido remoto possuírem o mesmo endereço e serem transmitidas ao mesmo tempo, será a trama de dados a ganhar a arbitragem, uma vez que o bit RTR desta trama é transmitido como dominante. Desta forma o nó que transmitiu o pedido remoto recebe de imediato os dados pretendidos.

2.2.4 - Trama de erro

A trama de erro é gerada por qualquer nó do barramento que detecte um erro no mesmo. Esta trama é composta por dois campos: campo de flag de erro e campo delimitador de erro. Este último consiste em 8 bits recessivos e permite aos outros nós do barramento reiniciar as comunicações no mesmo, limpo após um erro. Contudo existem dois tipos para o campo de flag de erro, dependendo do ‘Estado de Erro’ em que estava o nó que detectou o erro.

Se um nó erro-activo (em transmissão) detecta um erro no barramento, interrompe de imediato a transmissão com uma flag de erro activa, correspondente a 6 bits dominantes consecutivos. Uma vez que esta sequência viola a regra do “*bit stuffing*”, a qual não permite 6 bits consecutivos com o mesmo valor, os outros nós irão detectar o erro e também gerar eles mesmos uma trama de erro. Assim o campo de flag de erro pode conter entre 6 e 12 bits dominantes consecutivos. O campo delimitador de

erro conclui a trama de erro. Após completar esta trama, a actividade no barramento regressa ao normal e o nó que foi interrompido tentará reenviar a mensagem abortada. Na Figura 9 está um exemplo desta trama.

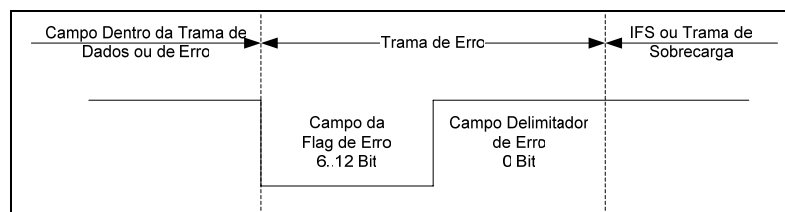


Figura 9 – Trama de erro

Se um nó erro-passivo (em recepção) detecta um erro no barramento, então transmite uma flag de erro passiva, também seguida do já referido delimitador. Esta flag consiste em 6 bits recessivos, pelo que (em referência a um nó erro-passivo) consiste num total de 14 bits recessivos (sem bits dominantes). Isto significa que a menos que o nó transmissor detecte o erro no barramento, a transmissão da trama de erro por um nó passivo não vai ter efeito nos outros nós da rede. Se o nó transmissor gerar uma flag de erro passiva, os outros nós irão então gerar tramas de erro devido à violação da regra de “*bit stuffing*”.

2.2.5 - Trama de sobrecarga

A trama de sobrecarga tem o mesmo formato da trama de erro-activo, mas apenas pode ser gerada durante o espaço entre tramas (IFS). É desta forma que as duas são diferenciadas. A trama de sobrecarga é composta por dois campos: flag de sobrecarga e delimitador de sobrecarga. A flag de sobrecarga consiste em 6 bits dominantes seguidos pelas flags geradas pelos outros nós (um nó erro-activo pode gerar um máximo de 12 bits dominantes). O delimitador de sobrecarga consiste em 8 bits recessivos consecutivos. Esta trama é gerada devido a problemas internos no nó que não permite que este já possa receber a trama que vai ser enviada já de seguida. Um nó pode apenas gerar duas tramas sequenciais de sobrecarga. Na Figura 10 pode ser observada este tipo de trama.

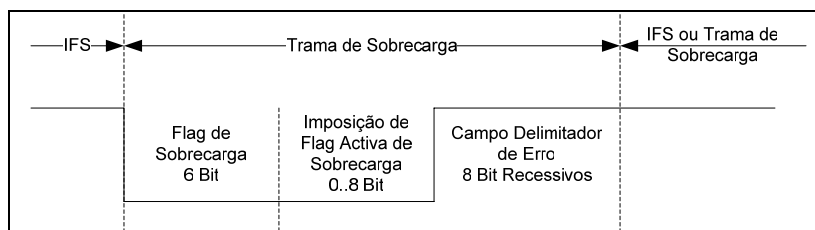


Figura 10 – Trama de sobrecarga

2.2.6 - Espaço entre tramas

O espaço entre tramas (IFS), tal como o nome indica, é o espaço existente entre uma qualquer trama de dados ou de pedido remoto. Este espaço é composto por pelo

menos 3 bits recessivos, designados ‘Intermissão’. Este tempo é necessário para permitir que os nós façam o seu processamento interno antes do início da transmissão de uma nova mensagem. Após o tempo de intermissão, para nós erro-activo, a linha do barramento mantém-se recessiva até que a próxima transmissão seja iniciada, tal como se pode ver pela Figura 11.

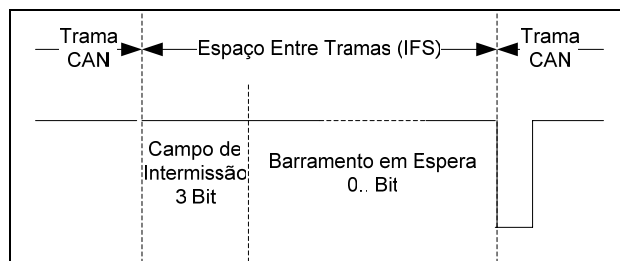


Figura 11 – Espaço entre tramas

O IFS tem um formato ligeiramente diferente para nós erro-passivo que na mensagem anterior foram transmissores. Neste caso, estes nós tem que aguardar mais 8 bits recessivos, designados “Transmissão Suspensa”, antes do barramento entrar em modo de espera para eles depois da intermissão e permitir-lhes enviar outra vez. Devido a este mecanismo, os nós erro-activo têm hipótese de transmitir as suas mensagens antes que os nós erro-passivo tenham permissão para iniciar nova transmissão.

2.3 - Detecção e sinalização de erros

O protocolo CAN possui vários mecanismos sofisticados para detecção de erros, podendo estes ter a seguinte natureza:

- Erros de CRC
- Erros de formato
- Erros de “*bit stuffing*”
- Erros de ACK (Confirmação de Recepção)
- Erros de bit

2.3.1 - Detecção de erros CRC

O transmissor calcula uma soma de verificação para a sequência de bits desde o bit SOF até ao fim do campo de dados. Esta sequência CRC é transmitida no campo CRC da trama. O nó que recebe a trama também faz este cálculo e compara o resultado com a sequência recebida. Este processo pode ser observado na Figura 12.

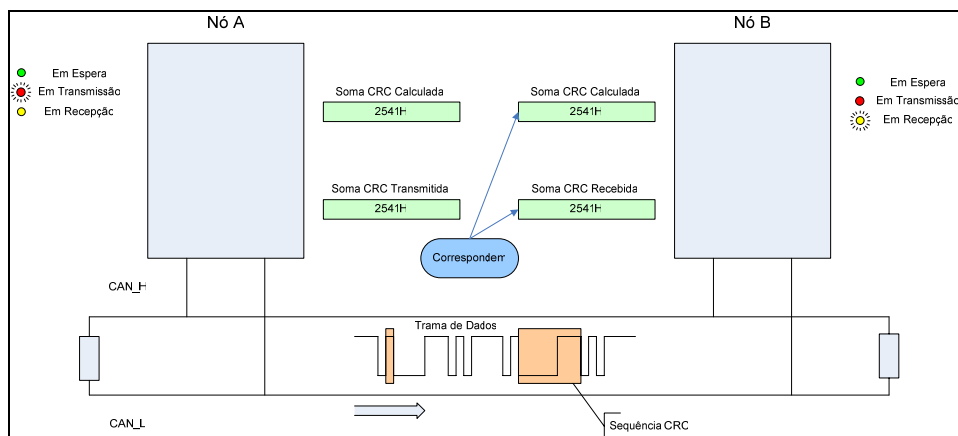


Figura 12 – Detecção de erros CRC

Se o nó de recepção detectar a não correspondência entre a sequência recebida e a sequência calculada, gera um erro CRC. De seguida descarta a mensagem e emite uma trama de erro para pedir a retransmissão da trama errónea.

2.3.2 - Detecção de erros ACK

Tal como já foi afirmado anteriormente, sempre que um nó receptor recebe uma mensagem, envia um sinal de confirmação da recepção (*acknowledge*), isto é, coloca o bit ACK em dominante. Se o transmissor não receber este sinal, gera-se um “Erro ACK”. Este processo pode ser observado Figura 13.

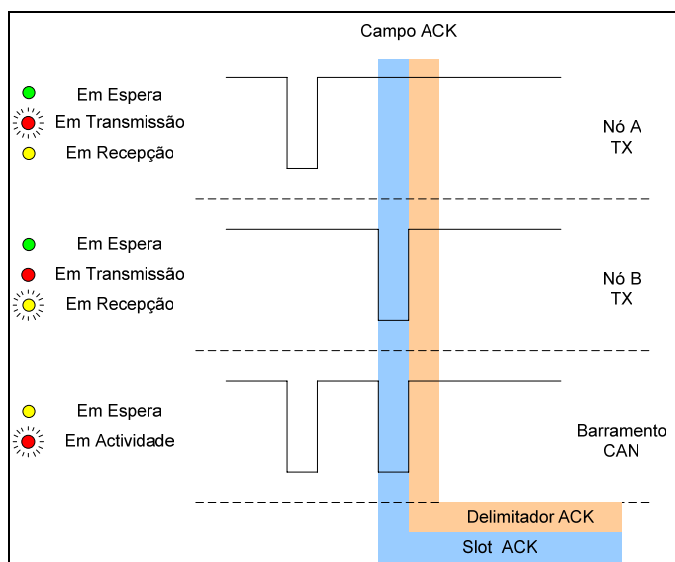


Figura 13 – Detecção de erros ACK

2.3.3 - Detecção de erros de formato

Se o nó transmissor detectar um bit dominante nos seguintes campos: delimitador CRC, delimitador ACK, EOF ou IFS; então ocorre um erro de formato e é

gerada uma trama de erro. A mensagem será depois repetida. Estes campos podem ser observados na Figura 14.

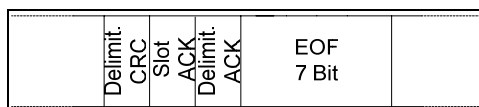


Figura 14 – Campos da trama com formatos fixos

2.3.4 - Detecção de erros por monitorização de bits

Todos os nós efectuam a monitorização dos bits. Um erro acontece se o transmissor envia um bit dominante mas detecta no barramento um bit recessivo ou vice-versa. Desta forma gera-se uma trama de erro e a mensagem é repetida. Este erro não é assinalado durante o campo de arbitragem ou a slot ACK, uma vez que estes campos tem que ter permissão para serem reescritos para atingir o objectivo da arbitragem e reconhecerem funcionalidade.

2.3.5 - Detecção de erro de “bit stuffing”

A codificação usada pelo protocolo CAN para os bits é designada *Non Return to Zero* – NRZ. Mas uma das características desta codificação é que o sinal não possui declives que possam ser usados para a re-sincronização quando é transmitido um grande número de bits consecutivos do mesmo valor. Deste modo, usa-se o “bit stuffing” para assegurar a sincronização para todos os nós do barramento. O “bit stuffing” no protocolo CAN consiste em cada cinco bits consecutivos de um mesmo valor, colocar um bit de sinal oposto. Esta colocação é realizada pelo nó transmissor e incluída na codificação, pelo que os nós receptores podem eliminar estes bits e assim apresentar a mensagem correctamente. Este formato de codificação pode ser observado na Figura 15.

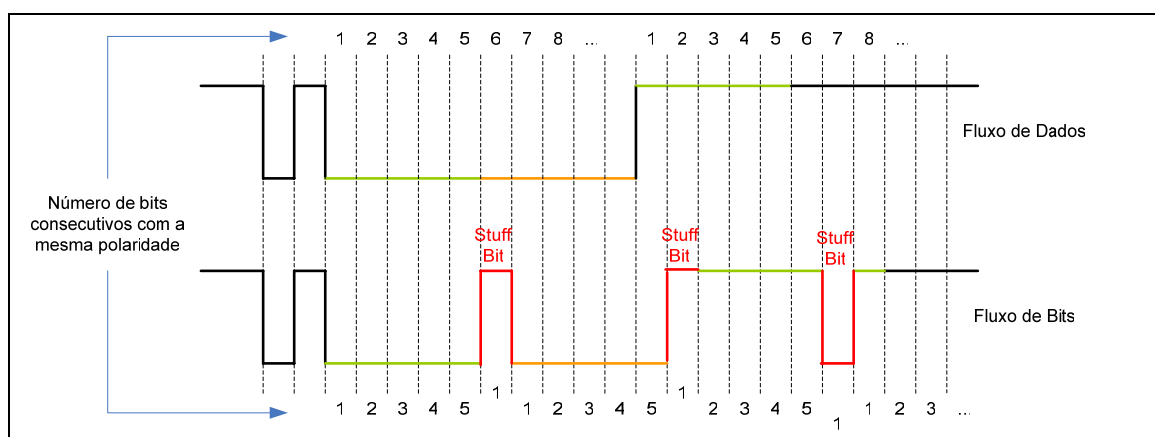


Figura 15 – Aplicação do “bit stuffing” no protocolo CAN

Assim, se forem detectados 6 bits consecutivos do mesmo valor entre SOF e o delimitador CRC, significa que a regra do “bit stuffing” foi violada. Um erro de “bit

stuffing” é assinalado e enviada a trama de erro correspondente, sendo depois a mensagem repetida.

2.3.6 - Gestão dos erros

Tal como pode ser observado na Figura 16, os erros detectados são tornados públicos para todos os nós através de tramas de erro. De seguida a transmissão da mensagem de erro é abortada e a trama repetida assim que seja possível.

De acordo com os contadores internos, cada nó está num de três estados de erro, como pode ser visto na Figura 16:

- Estado erro-activo
- Estado erro-passivo
- Estado barramento desligado

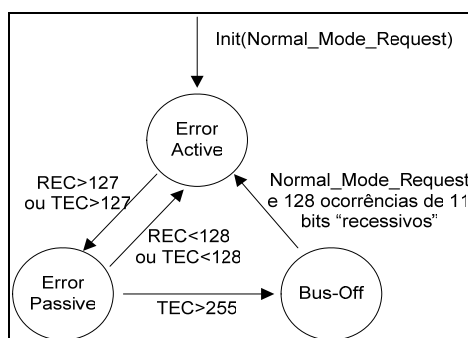


Figura 16 – Gestão de erros no protocolo CAN

O estado erro-activo é o estado normal após a ocorrência de uma reinicialização. Assim, o nó pode receber e transmitir mensagens e tramas de erro-activas (geradas por bits dominantes) sem qualquer restrição. Durante as comunicações do protocolo, os contadores de erro são actualizados de acordo com as regras existentes. Para cada recepção ou transmissão, caso sejam detectados um ou mais erros por pelo menos um nó, a transmissão actual no barramento é interrompida e assinalada com uma “Flag de Erro” e os respectivos contadores de erro são incrementados de um certo valor, sendo os mesmos contadores decrementados sempre que uma transacção se efectua com sucesso.

O estado erro-activo é válido enquanto ambos os contadores têm um valor menor ou igual a 127. Se tanto o contador de erros de recepção como o contador de erros de transmissão chegarem ao valor 128, o nó comuta para o estado erro-passivo. Neste estado, as mensagens ainda podem ser recebidas ou transmitidas, embora após a transmissão das mensagens o nó tenha que suspender a transmissão. Tem que esperar 8 bits a mais de tempo que os nós erro-activo antes de transmitir outra mensagem. Em termos de sinalização de erros, só os nós erro-passivo podem transmitir tramas de erro, constituídas por bits recessivos.

Se ambos os contadores voltarem a ter um valor inferior a 128, devido a comunicações no barramento bem sucedidas, o nó volta para o estado erro activo. Uma das características do protocolo CAN é que os nós defeituosos (com muitos erros) são automaticamente removidos do barramento.

Se estes erros se forem acumulando, o protocolo CAN possui mecanismos próprios para determinar se estes são esporádicos ou permanentes. Com a probabilidade crescente de afectar pelo lado negativo o resto da rede, existem também alguns mecanismos de contenção de falhas, os quais em último caso podem levar ao desligar de determinado nó problemático.

O estado de barramento desligado é atingido se o contador de erros de transmissão chegar ao valor 255. Todas as actividades do barramento são forçadas a parar, o que torna temporariamente impossível esse nó participar na comunicação do barramento. Durante este estado não há transmissão ou recepção de mensagens. Para voltar ao estado erro-activo e colocar a zero os contadores, é necessária a reinicialização do nó CAN.

Esta grande vantagem do CAN na detecção de erros e nos seus mecanismos de gestão, tais como a verificação CRC e a alta imunidade às interferências electromagnéticas, leva a que as mensagens que são detectadas como estando erradas sejam retransmitidas, recuperando-se de erros temporários existentes. Mas se os erros forem permanentes, os nós defeituosos são desactivados. Desta forma é garantida a consistência dos dados a nível global.

3 - Opções de camadas físicas para o protocolo CAN

De acordo com a norma ISO11898-1, o protocolo CAN não possui uma definição própria para uma camada física, mas apenas os valores lógicos que esta terá que representar. Para além disso, esta camada também tem que possuir a capacidade de enviar e receber sinais em simultâneo, o que é necessário para os requisitos temporais ao nível dos bits pelo protocolo CAN.

Actualmente, encontram-se desta forma definidas duas camadas físicas:

- Camada física de alta velocidade, definida pela norma ISO11898-2.
- Camada física de baixa velocidade, tolerante a falhas, definida pela norma ISO11898-3.

3.1 - Camada física de alta velocidade

O CAN é um barramento “*multimaster*” com uma estrutura linear aberta, com uma linha de barramento lógica e nós iguais. A quantidade de nós não é limitada pelo protocolo, suportando tanto “*multicasting*” como o “*broadcasting*”.

Tal como está definido na norma, a camada física é uma linha de barramento com dois fios, onde em cada ponta terminal terá uma resistência a qual representa a impedância característica da rede. De modo algum esta resistência deve ser colocada no interior dos equipamentos que estão ligados à rede. Devido à possível formação de ondas reflectidas, a topologia da rede deve assemelhar-se o mais possível a uma estrutura em linha simples. Na prática, isto significa que as ligações ao barramento devem ser as mais curtas possíveis. Veja-se o exemplo na Figura 17.

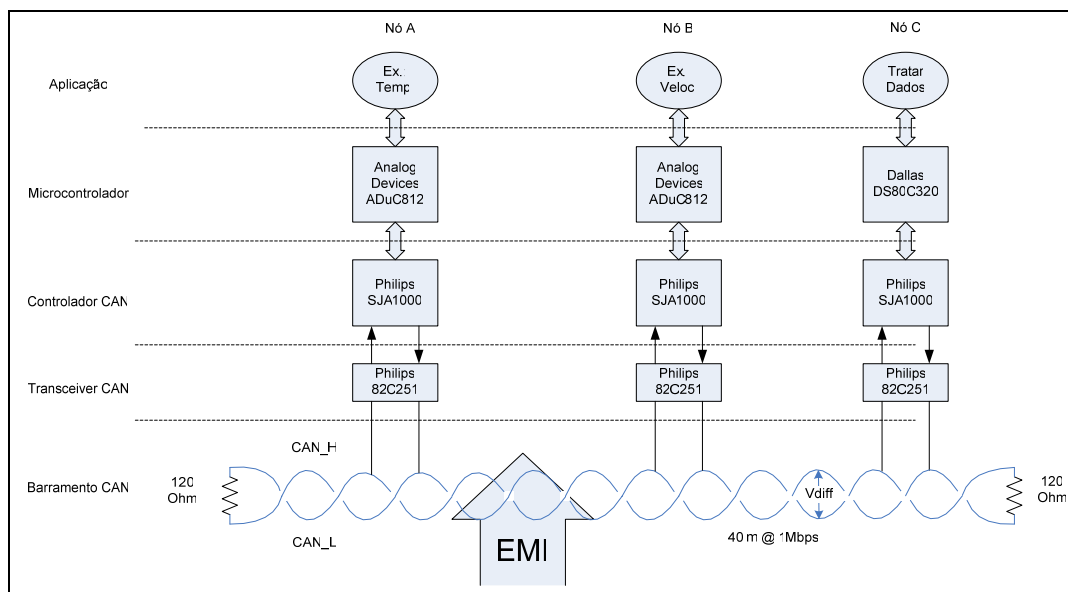


Figura 17 – Exemplo de uma ligação física de uma rede CAN

O comprimento máximo recomendado para a rede usando o par de fios entrelaçado para o barramento e uma velocidade de 1Mbps é de 40 metros, no entanto esta distância pode ser aumentada com recurso a repetidores, isolamento galvânico ou diminuição da velocidade. Como as mensagens são curtas (apenas 8 bytes no máximo) existe um período curto de latência entre o pedido de transmissão e o início da mesma.

O acesso ao barramento é gerido por um protocolo avançado de comunicação série *Carrier Multiple Access/Colision Detection* – CSMA/CD; com prioridade arbitrada na mensagem, não destrutiva. Isto significa que a colisão das mensagens é evitada com recurso ao “*bit-wise*” sem haver perdas para a mensagem mais prioritária.

Existem apenas dois estados lógicos: o dominante e o recessivo. Para obter estes estados, o barramento lógico utiliza o mecanismo “*wired-AND*”, isto é, os bits dominantes equivalentes a um nível lógico ‘0’, sobrepõem-se aos bits recessivos, equivalentes ao nível lógico ‘1’. Isto pode ser observado na Figura 18.

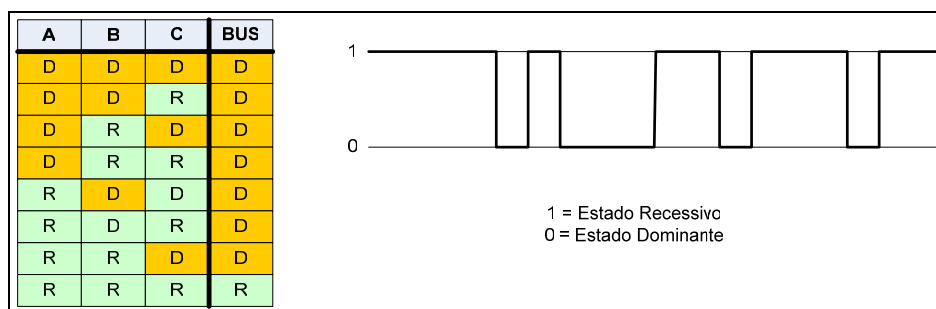


Figura 18 – Estados no barramento do protocolo CAN

Se todos os nós transmitirem bits recessivos e só nesse estado, diz-se que o barramento está no estado recessivo, tal como se pode ver pela Figura 19. Assim que um dos nós transmitir um bit dominante, o barramento passa a considerar-se no estado dominante, como pode ser visto na Figura 20.

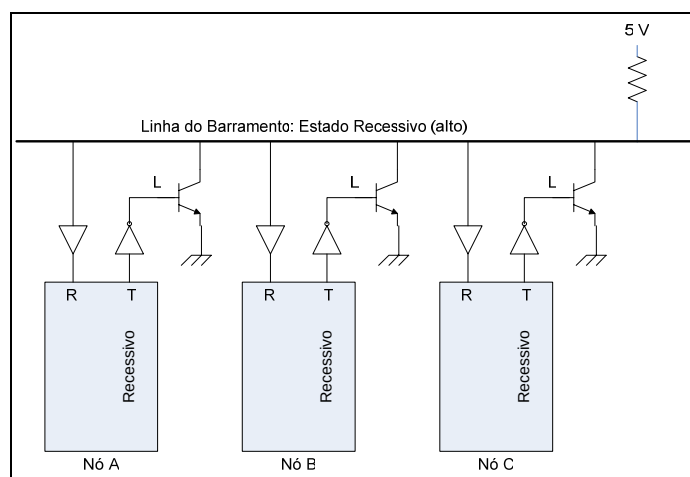


Figura 19 – Estado “recessivo” no barramento

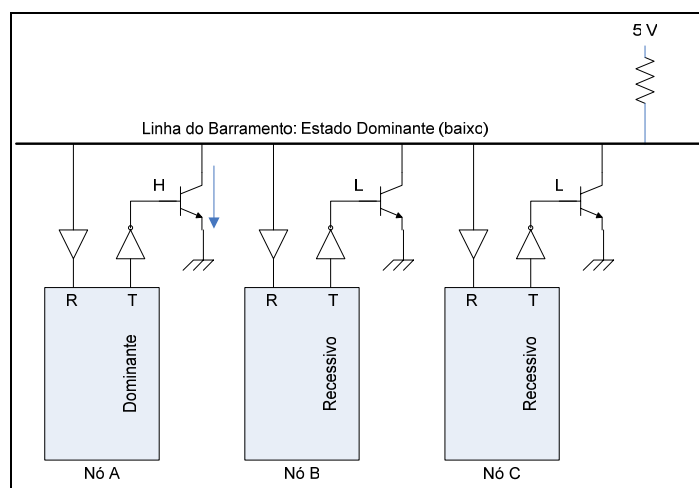


Figura 20 – Estado “dominante” no barramento

3.2 - Camada física de baixa velocidade tolerante a falhas

Para redes baseadas no protocolo CAN, existem várias especificações para camadas tolerantes a falhas. A norma ISO11519-2 efectua a descrição para o circuito do *transceiver*, mas como este tem fraca capacidade de corrente de saída entre outros problemas, esta norma já se encontra em desuso. Para aplicações que exigem tensões diferenciais mais elevadas (é o exemplo das viaturas pesadas) foi especificada a norma 11992. As novas especificações propostas para os transceivers já fazem referência ao baixo consumo dos mesmos, desligar e re-ligação na rede.

Segundo as especificações definidas pela norma ISO11898-3, no barramento CAN apenas podem existir dois estados lógicos: o estado recessivo e o estado dominante; sendo que a distinção entre os dois é realizado mediante uma tensão diferencial. No primeiro estado a linha CAN_L tem uma tensão fixa mais elevada do que a linha CAN_H, sendo transmitido quando o barramento está em espera ou se for um bit recessivo. De modo inverso, tem-se o segundo estado, o qual se sobrepõe ao primeiro e é transmitido durante os bits dominantes.

No entanto, para que estas características tolerantes a falhas resultem não podem existir mais do que 20 nós na rede para uma velocidade máxima de 125kbps com um barramento de 40 metros de comprimento, mas estes valores não são absolutos. A característica principal desta camada física é a sua capacidade de se comportar como transmissão por linha única na eventualidade de ocorrerem curto-circuitos no barramento (linha-negativo, linha-positivo ou linha-linha).

4 - Protocolos de camada elevada

É normal encontrarem-se descrições de comunicação e de objectos de comunicação feitas com modelos, alguns deles em camadas hierarquicamente organizadas, sendo o chamado modelo ISO/OSI um dos mais usados para esse fim. Na Figura 21 pode-se ver que actualmente a implementação por hardware do protocolo CAN cobre as camadas 1 e 2, enquanto que as diversas soluções existentes implementadas por software cobrem as camadas 3 a 7 do modelo ISO/OSI.

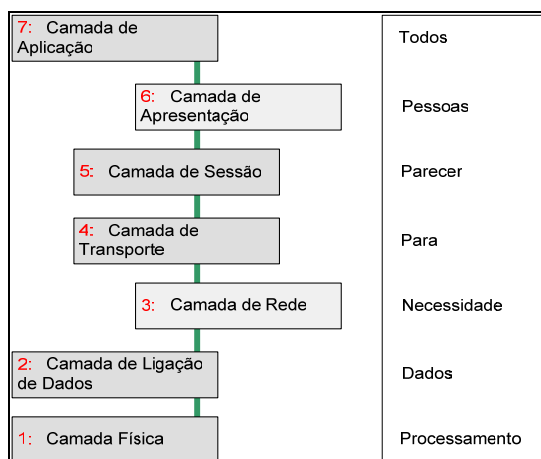


Figura 21 – Modelo ISO/OSI para o protocolo CAN

Fazendo-se uma analogia, as diversas funcionalidades existentes no protocolo CAN são muito semelhantes às letras que os humanos usam para comunicarem. Embora sejam a base da linguagem, não é o suficiente para que a comunicação se processe de maneira eficiente. É também preciso uma espécie de gramática. O utilizador do protocolo CAN pode definir a sua própria linguagem baseada em CAN, uma vez que esta em termos técnicos é da camada de alto nível, o que confirma a camada 7 do modelo OSI de referência.

No entanto, por vezes apenas são necessários procedimentos simples. Tal como na linguagem tem-se aquilo que se designa por “frases feitas”, também no protocolo CAN existem perfis standard com os quais é possível realizar estes procedimentos sem grande esforço.

Assim, a comunicação baseada em CAN pode ser separada em quatro níveis diferentes: a camada física tem a sua implementação realizada por um controlador CAN e um “*transceiver*”, ao passo que a camada da aplicação assim como os perfis são implementados por software, não havendo distinção entre a camada da aplicação e o próprio perfil na maioria das vezes.

5 - Síntese

Após este breve estudo, pode-se concluir que:

- Embora o protocolo CAN seja já relativamente antigo, ainda é um dos mais usados, prevendo-se a sua longa durabilidade do seu uso e desenvolvimento/melhoramento;
- Devido à sua simplicidade, facilidade de implementação e também o facto de estar normalizado segundo as normas ISO, torna-se um protocolo que múltiplas das empresas e indústrias continuarão a implementar num futuro próximo;
- Também se salienta os esforços efectuados por diversas indústrias para melhorar este protocolo de forma que se adapte ainda mais facilmente às necessidades do mundo actual.

Capítulo 3

Hardware para as ligações à rede

Capítulo 3 - Hardware para as ligações à rede

1 - Controladores para rede CAN

Se o objectivo é construir uma rede não muito dispendiosa de sensores e actuadores, esta pode ser projectada usando os chamados controladores CAN SLIO¹. Este tipo de controladores I/O com controlador CAN integrado são configurados e controlados a partir de nós inteligentes (designados de nó anfitrião ou nó “*master*”) através do barramento CAN. Com o recurso a um protocolo fixo, o nó “*master*” tem acesso completo e directo a todas as portas do controlador SLIO, evitando-se assim o recurso a software adicional no nó SLIO. Contudo, com o progresso e o baixo custo actual dos microcontroladores com controlador CAN integrado, não é justificado o uso de circuitos integrados tão especializados como os SLIO. Além disso, se somarmos as limitações a que as redes SLIO estão sujeitas, tais como o número de nós, a taxa de dados e o protocolo específico, torna-se claro porque é que estes circuitos estão quase obsoletos.

Em projectos actuais de circuitos, está em franco crescimento o uso de circuitos integrados de aplicação específica (ASIC) e bibliotecas de célula, em muito devido ao facto de nestes circuitos integrados já estarem integrados controladores de diversos periféricos discretos, o que leva a um sistema mais compacto. Assim, os fabricantes actualmente já oferecem controladores de protocolo CAN em adição às numerosas células periféricas (ex.: controladores com conversores A/D, portas I/O, comparadores e interrupções). Estes controladores CAN geralmente são standards tal como aqueles que se encontram nas soluções por microcontroladores. No entanto, estes também podem sofrer modificações de acordo com as especificações da aplicação.

Devido aos diferentes requisitos de tecnologia, a funcionalidade dos “*transceivers*” (adaptadores) de barramento não está integrada nos controladores de protocolo CAN, embora alguns fabricantes os integrem em algumas das suas soluções com controladores autónomos.

Para uso comercial e em série, os primeiros controladores para o processamento do protocolo CAN a ficarem disponíveis foram o Intel 82526 e o Philips PCA82C200, que foram implementados como soluções do tipo autónomo e comunicavam com os controladores anfitriões através de um simples interface paralelo. Embora ainda existam em larga escala numa grande variedade de aplicações, estes controladores de primeira geração já não se apresentam na sua forma original.

No entanto, a experiência obtida pela implementação destes controladores deu um grande contributo para o desenvolvimento e implementação dos novos controladores. Esta segunda geração de controladores possui um grande conjunto de extensões, dos quais se destacam os controladores “*FullCAN*”, que possuem grande capacidade de buffers de mensagens, ou controladores combinados que possuem características do “*FullCAN*” e do “*BasicCAN*” no mesmo controlador. Também uma

¹ Serial Linked I/O

percentagem significativa destes novos controladores tolera ou suporta o protocolo CAN estendido com um identificador de 29 bits.

Para se poder constituir uma rede CAN completa, não bastará ter apenas os controladores CAN autónomos ou os microcontroladores com controladores CAN integrados. Também é necessário que estes estejam implementados nos respectivos interfaces, sejam placas de nós nos equipamentos ligados numa rede CAN ou placas de interface com computadores (que também podem ser considerados como nós).

Assim, nesta dissertação proceder-se-á também à descrição de algumas interfaces de rede CAN para computadores no seu formato PCI e dos interfaces dos nós que estão colocados ao longo da rede CAN. Em cada uma das descrições, apenas se faz referência ao microcontrolador principal e ao controlador de protocolo CAN e não uma descrição detalhada destes, a qual pode ser consultada nos respectivos manuais de características técnicas.

1.1 - Funções do controlador CAN

Quer seja autónomo, quer seja integrado num microcontrolador, o controlador CAN, tal como foi referido anteriormente, é o responsável pela realização das tarefas relacionadas com a camada 1 e a camada 2 do protocolo CAN, tais como:

- Arbitragem do barramento
- Serialização e de-serIALIZAÇÃO das tramas a enviar ou a receber
- Cálculo e verificação de sequências de controlo (CRC)
- Detecção e sinalização de erros
- Construção do formato de mensagem CAN
- Inserção e apagamento de “*stuff bits*”
- Geração e verificação do “*acknowledge bit*”
- Sincronização ao bit do fluxo recebido

De forma a diminuir a quantidade de tarefas do protocolo no controlador anfitrião, todas estas tarefas são realizadas por um controlador em separado de hardware. Além disso, os controladores CAN realizam outras tarefas, tais como a filtragem de mensagens, armazenamento de mensagens e resposta automática a pedidos de mensagens remotas.

1.1.1 - Filtragem de mensagens

Como já foi referido, neste tipo de redes, a transmissão de dados é efectuada segundo o princípio de difusão de mensagens. No entanto, por vezes um nó apenas está interessado em algumas das mensagens entre as que lhe são dirigidas, pelo que é apropriado possuir um sistema de filtragem de mensagens implementado em conjunto com o controlador CAN, de modo que o controlador anfitrião apenas veja as mensagens que realmente são relevantes. A isto se dá a designação de ‘Filtragem de Aceitação’.

Esta filtragem é efectuada tendo em conta o identificador da mensagem recebida, podendo enviar a mensagem para o buffer de mensagens ou para o buffer do controlador CAN conforme a decisão e também se o controlador anfitrião deve ser avisado ou não da recepção de novas mensagens. De forma a poder efectuar esta filtragem, o controlador CAN deve ser informado de todos os identificadores de mensagens em que o nó está interessado.

Embora os buffers não possam ser configurados para aceitarem apenas determinadas mensagens específicas, ao contrário do que sucede no “*FullCAN*”, podem ser configurados de modo a aceitarem uma determinada gama de identificadores, transferindo assim para o buffer de recepção apenas as mensagens cujos identificadores pertencem a esta gama. Assim, torna-se necessário antes da inicialização do controlador configurar o ‘Registo de Código de Aceitação’ (“*Accept Code*”) e ‘Registo de Máscara de Aceitação’ (“*Accept Mask*”) correctamente. O registo de código de aceitação contém os 8 bits mais significativos do identificador da mensagem (ID10 a ID3), que devem estar presentes obrigatoriamente se a mensagem pretende passar no filtro, quando todos os bits são relevantes.

Com o registo de máscara de aceitação indica-se quais os bits da mensagem que realmente são relevantes para a sua filtragem, isto é, quais os bits que tem que condizer com os valores dos bits do identificador da mensagem recebida. Os bits do identificador relevantes são identificados por um ‘0’ e os irrelevantes por um ‘1’ na máscara de aceitação. A filtragem de uma mensagem apenas ocorre nos seus 8 bits mais significativos, não envolvendo os 3 bits menos significativos (D2 a D0), pelo que o controlador pode aceitar até 8 mensagens com identificadores sucessivos, mesmo quando todos os bits são relevantes.

Contudo, se se pretender receber todas as mensagens, o ‘registo da máscara de aceitação’ terá que ter o valor 255, o que torna todos os identificadores para a filtragem irrelevantes e por isso não efectua a filtragem. Isto torna-se particularmente importante na prática, se for o caso de uma implementação como nó central, visto que este tem que receber um grande fluxo de mensagens.

Ao contrário do “*FullCAN*”, não é possível escolher apenas os identificadores em que o controlador anfitrião está interessado sem haver a preocupação da reserva do identificador. A inclusão ou exclusão é sempre executada sobre um grupo de identificadores, mas ainda assim não pode ser levada a bom termo apenas baseada exclusivamente nas possibilidades dos mecanismos de filtragem de mensagens disponíveis.

Uma vez que a prioridade da mensagem para reserva do acesso ao barramento também é determinada pelo valor do identificador, a reserva dos identificadores das mensagens irá ocorrer numa grande maioria das aplicações, primeiro de acordo com os requisitos específicos da aplicação para prioridades da mensagem. Como a possível utilização de hardware para filtragem de mensagens baseado no “*BasicCAN*” é algo limitado, esta é normalmente efectuada por software.

Os vários mecanismos de filtragem de mensagens existentes nos diversos controladores do mercado são descritos mais adiante com os respectivos controladores.

1.1.2 - Manuseamento de mensagens

Existem dois princípios de manuseamento de mensagens:

- Armazenamento temporário de todas as mensagens num buffer de recepção e num buffer de transmissão.
- Armazenamento de um número limitado de mensagens num buffer específico de transmissão ou de recepção.

A designação mais vulgar para estes dois princípios é “*BasicCAN*” e “*FullCAN*”, respectivamente. Na Figura 22 pode-se ver a estrutura básica de um controlador de protocolo CAN de acordo com o princípio de funcionamento “*BasicCAN*”.

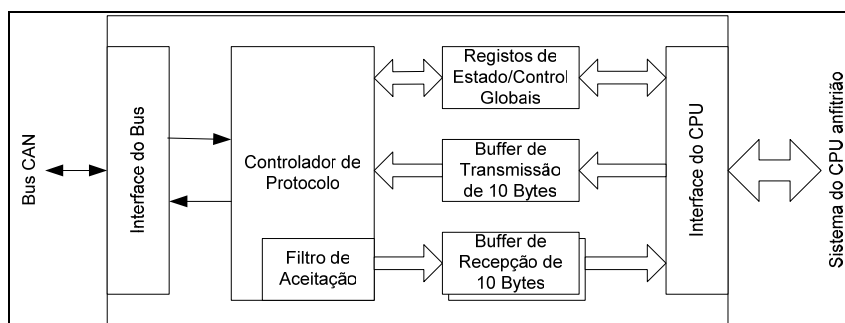


Figura 22 – Princípio de funcionamento do BasicCAN

A Figura 23 apresenta também a estrutura do controlador de protocolo CAN, mas agora segundo o princípio “*FullCAN*”.

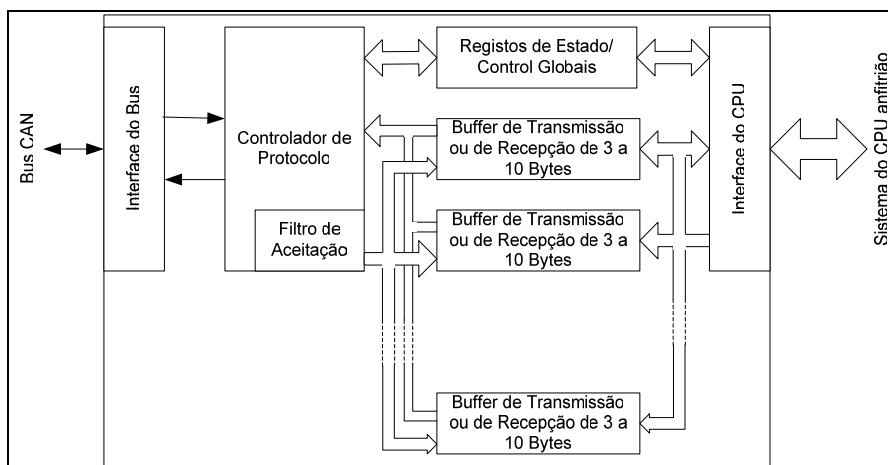


Figura 23 – Princípio de funcionamento do FullCAN

Além disso, estes dois princípios também diferem no que respeita ao pedido remoto de mensagens.

A característica principal do princípio “*BasicCAN*” consiste em possuir dois buffers: uma para a recepção e outro para a transmissão. Aquando da recepção é efectuada a ‘filtragem de aceitação’, antes de fazer o armazenamento no buffer respectivo. No auxílio deste buffer existe um buffer de recepção ‘sombra’, de modo que durante a leitura de uma mensagem recebida por parte do processador, a seguinte possa ser recebida ao mesmo tempo pelo controlador. Contudo como apenas existem dois buffers de recepção, é necessário assegurar que o controlador anfitrião consegue efectuar a leitura da mensagem recebida até imediatamente antes de receber uma terceira mensagem, caso contrário iria criar-se uma sobrecarga de dados e assim perder-se-ão todas as mensagens. Apenas com o cumprimento desta condição se assegura a recepção de todas as mensagens relevantes.

Qualquer nó numa rede CAN pode, por princípio, transmitir as suas mensagens independentemente de os outros nós o fazerem, pelo que assim pode surgir uma sucessão directa de várias mensagens a qualquer momento e que não é previsível. Assim sendo, é muito importante considerar-se a carga média do barramento, mostrando-se na Figura 24 o pior caso para o projecto do sistema.

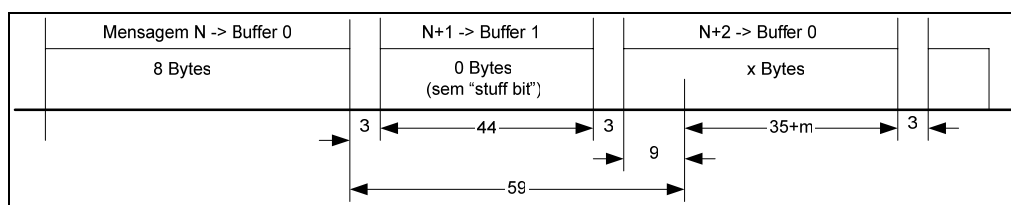


Figura 24 – Pior caso na condição de tempo-real de um controlador BasicCAN

Neste caso tem-se o tempo mínimo para a leitura da mensagem mais longa disponível, pelo que esta situação acontece quando a uma mensagem de 8 byte segue-se uma mensagem de 0 byte imediatamente seguida de outra mensagem. Neste caso, apenas 59 bits de tempo estão disponíveis para a leitura da mensagem de 8 bytes, de acordo com a Figura 24. Com uma taxa de 1 Mbit/s, isto corresponde a um tempo de 59μs, no qual o controlador anfitrião tem que ler 10 bytes do buffer de recepção. Actualmente, com o aumento da capacidade do buffer, estas limitações de tempo-real foram substancialmente reduzidas.

Ao contrário do controlador com princípio “*BasicCAN*”, o controlador com princípio “*FullCAN*” possui um buffer independente para cada mensagem, tendo um número limitado para transmissão e para recepção, tal como pode ser observado na Figura 23. A configuração e reserva dos buffers é feita através do controlador anfitrião, sendo que as mensagens recebidas só irão para os buffers de recepção configurados como tal.

Se num determinado buffer de recepção já estiver uma mensagem e for recebida uma mensagem com o mesmo identificador dessa, a mensagem anterior é sobreposta, garantindo-se assim a existência permanente da mensagem mais actual. Desta forma o controlador anfitrião não tem necessidade de estar sempre a ler as mensagens no buffer. Observe-se, contudo, que como o controlador CAN tem acesso permanente aos buffers, pode ocorrer a situação de o controlador anfitrião estar a ler um buffer ao mesmo tempo que o controlador CAN está a escrever, tendo que o controlador anfitrião verificar a

consistência dos dados e detectar se correspondem a duas mensagens sucessivas mas distintas.

A vantagem deste princípio é que retira parte das tarefas do controlador anfitrião, uma vez que mesmo ao receber várias mensagens com identificadores diferentes em intervalos de tempo muito curtos, o controlador CAN encaminha-os logo para o buffer correspondente, coisa que não acontece no princípio “*BasicCAN*”.

1.2 - Pedidos remotos de mensagem

Como existem vários princípios diferentes de fazer buffer de mensagens, tanto o princípio do “*BasicCAN*” como do “*FullCAN*” também têm diferenças no que respeita ao manuseamento de pedidos de mensagens remotos. Uma vez que o armazenamento de mensagens com recurso ao princípio “*BasicCAN*” é efectuado na memória do controlador anfitrião, este tem que receber o pedido de mensagem remoto e indicar a resposta a dar pelo controlador “*BasicCAN*”. Então depois fornece a mensagem pedida, escrevendo-a no buffer de transmissão do controlador de protocolo.

No que respeita ao princípio “*FullCAN*”, como este já possui um buffer melhor, é possível enviar a mensagem pedida remotamente pelo próprio controlador CAN. Note-se, porém, que para o controlador poder fornecer os dados actualizados dos buffers, a aplicação do controlador anfitrião tem que efectuar uma actualização apropriada de dados.

1.3 - Áreas de aplicação dos controladores BasicCAN e FullCAN

Embora o princípio “*FullCAN*” pareça ser o mais vantajoso, uma vez que retira grande parte da carga de processamento do controlador anfitrião relacionadas com o controlo do protocolo e o buffer de mensagens, o princípio “*BasicCAN*” também implementa algumas funcionalidades básicas do “*FullCAN*”, tornando-o assim eficiente mas de baixo custo. Por observação da Figura 23, pode-se notar que o princípio “*FullCAN*” é algo limitado no que respeita ao número de mensagens que pode receber num único nó, restrição esta que não existe no princípio “*BasicCAN*”.

1.4 - Controladores CAN autónomos

Tal como já foi referenciado, os controladores designados autónomos apenas fornecem capacidades intermediárias de armazenamento de mensagens. O modo de endereçamento dos registos e dos buffers são normalmente endereçados como se de chips de memória RAM se tratassem. Na Figura 25 apresenta-se o diagrama de blocos de um nó CAN baseado num controlador autónomo comum. A interface entre o controlador anfitrião e o controlador CAN suporta vários tipos de barramento, tais como:

- 8 bit multiplexado
- 8 bit não multiplexado (síncrono ou assíncrono)

- 16 bit multiplexado

Alguns controladores também fornecem uma interface série. Um ponto muito importante nas interfaces é a característica temporal requerida pelo barramento CAN, uma vez que os controladores diferem significativamente a esse respeito e podem surgir problemas quando se usam controladores anfitriões demasiado rápidos.

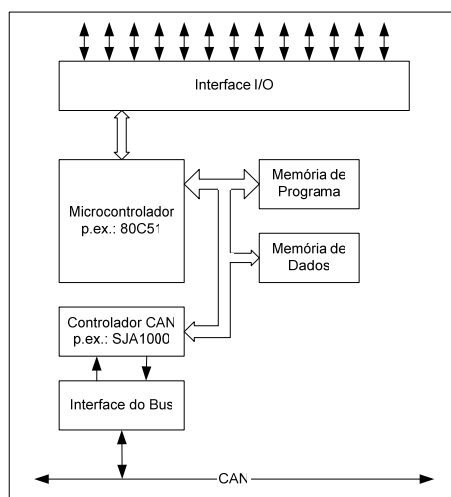


Figura 25 – Nó CAN baseado num controlador autónomo

Em todos os controladores CAN, sejam eles integrados num microcontrolador ou autónomos, existe uma série de registos, para além dos buffers de mensagens, úteis para configuração, monitorização e controlo do controlador CAN. Basicamente estes registos servem para:

- Indicação do estado actual do controlador tal como o estado de erro, reset, transmissão ou recepção;
- Controlo do estado do controlador tal como o “reset” e “set” do controlador após completar a configuração;
- Configuração do modo de operação do controlador tal como activar/desactivar interrupções ou funções similares;
- Configuração dos parâmetros temporais;
- Configuração da secção interna do transceiver.

No entanto, estas não são as únicas características, existindo outras especiais conforme o tipo de controlador CAN.

De seguida apresenta-se na Tabela 1 uma comparação entre alguns controladores CAN autónomos existentes no mercado, fazendo-se depois uma descrição dos mais usados: PHILIPS SJA1000 e INTEL 82527.

Fabricante	Referência:	Tipo de CAN	Buffers de Mensagens	Interfaces e I/O	Particularidades
INFINEON	SAK82C900	2.0 B	32 buffers independentes para Rx ou Tx (configuráveis)	Paralelo 8 bit MUX Série	Controlador CAN duplo; Não possui “transceiver”;
INTEL	82527	2.0 B	15 buffers (8 Bytes): 14 Tx/Rx 1 Rx Programável	2 I/O (8-bit) (QROM); 8/16 bit MUX; 8 bit ÑMUX (sync/assync); Série (SPI)	Várias informações sobre erros obtidos
MICROCHIP	MCP2510	2.0 A 2.0 B	3 buffers Tx (8 bytes) 2 buffers Rx (8 bytes)	Série (SPI)	Pinos próprios para interrupções
PHILIPS	SJA1000	2.0 B	BasicCAN: 1Tx (10 bytes), 1 Rx (10bytes) PeliCAN: Buffer alargado a 13 bytes (FIFO de 4 a 32 msg, total 64 bytes)	Compatível com a grande maioria dos microcontroladores	Aviso de limites de erros programável; Interrupção para erros de barramento

Tabela 1 – Tabela comparativa dos controladores de protocolo CAN autónomos

1.4.1 - Philips SJA1000

O controlador autónomo SJA1000 é a evolução do PCA82C200, sendo totalmente compatível com este, quer em termos de “*pinout*” quer em termos eléctricos. É de realçar que o PCA82C200 apenas funcionava no modo “*BasicCAN*”, tendo sido implementado no SJA1000 não só este modo de funcionamento como também uma versão mais estendida designada “*PeliCAN*”, a qual já suporta a versão 2.0B do protocolo CAN no modo passivo, além de outras características novas.

O diagrama de blocos representativo deste controlador está na Figura 26.

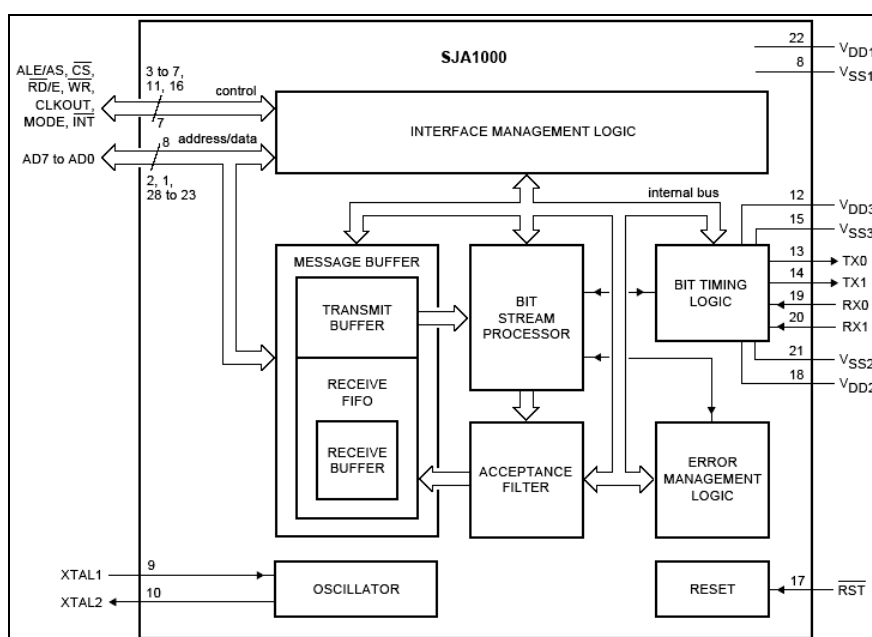


Figura 26 – Diagrama de blocos do funcionamento do PHILIPS SJA1000

De seguida far-se-á uma descrição dos modos de compatibilidade com o PHILIPS PCA82C200, devido ao facto de muitos fabricantes adoptarem muitas características deste controlador como padrão.

Modo “BasicCAN”

Como pode ser observado na Figura 26, apenas existe um buffer para transmissão e outro para recepção de mensagens. Pode-se olhar para este buffer de recepção como se fosse uma janela de entrada para uma fila FIFO, onde o controlador anfitrião pode observar apenas a primeira mensagem recebida em cada caso. Cada um dos buffers possui um tamanho de 10 bytes, dos quais 2 bytes são reservados para a descrição da mensagem e 8 bytes são reservados para o campo de dados. Em acrescento ao identificador de 11 bits da mensagem, o descritor também contém 4 bits para o tamanho do campo de dados e 1 bit para a especificação do tipo de trama (dados ou pedido remoto).

Na Tabela 2 apresenta-se a estrutura típica de um buffer no modo “BasicCAN”, em que toda a sua informação é transferida entre o controlador CAN e o controlador anfitrião, sendo o controlo do buffer efectuado pela correspondência de bits do estado global/registo de controlo.

Identificador de Mensagem ID 10 - 3		
ID 2 - 0	RTR	Código Comprimento dos Dados
Byte de Dados 1		
Byte de Dados 2		
Byte de Dados 8		

Tabela 2 – Formato da mensagem no buffer em modo BasicCAN

É necessário configurar correctamente vários parâmetros de acordo com o tipo de interface do barramento, tais como parâmetros temporais e “drivers” internos do controlador, parâmetros estes que podem ser observados na Tabela 3, nomeadamente o registo temporal do barramento e o registo do controlo da saída.

7	6	5	4	3	2	1	0	
Test Mode	Sync		Overr. Int. Enable	Error Int. Enable	Tx Int. Enable	Rx Int. Enable	Rst Req	Control
			Goto Sleep	Clear Overr. Status	Rel. Rx Buffer	Abort Tx	Tx Req.	Command
Bus Status	Error Status	Tx Status	Rx Status	Tx Compl. Status	Tx Buffer Access	Data Overr.	Rx Buffer Status	Status
			Wake-up Int.	Overr. Int.	Error Int.	Tx Int.	Rx Int.	Interrupt
AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Acceptance Code
AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0	Acceptance Mask
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	Bus Timing 0
SAM	TSeg2 2	TSeg2 1	TSeg2 0	TSeg1 3	Tseg1 2	Tseg1 1	Tseg1 0	Bus Timing 1
OC TP1	OC TN1	OC POL1	OC TP0	OC TN0	OC POL0	OC Mode 1	OC Mode 0	Output Control

Tabela 3 – Mapa dos registos de estado e controlo no modo BasicCAN

Na Figura 27 apresenta-se o “*driver*” de saída interno do controlador, o qual pode ser configurado de três formas, através dos bits de ‘Modo de Registo’ e de ‘Controlo da Saída’ (M0, M1):

- Modo ‘Saída Normal’, o fluxo transmite-se em ambas as saídas. Para isso, o modo de operação do transistor (“*PullUp*”, “*PullDown*”, “*PushPull*”) e a polaridade das saídas podem ser configuradas.
- Modo ‘Saída por Relógio’, o fluxo é transmitido através de uma saída e o sinal de relógio através da outra. A frequência do sinal de relógio é determinada pelos parâmetros dos registos temporais do barramento.
- Modo ‘Saída Bi-fásica’, os bits dominantes da mensagem são enviados alternadamente numa das saídas, pelo que o fluxo não tem uma componente de corrente directa.

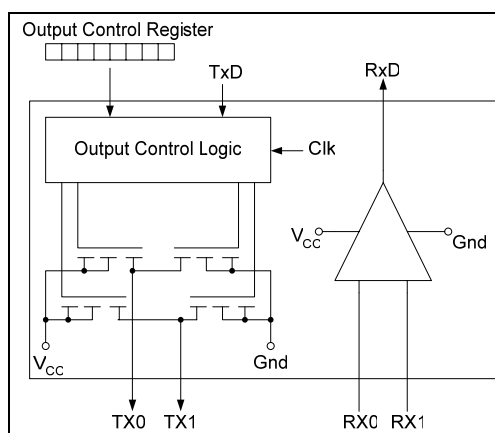


Figura 27 – Adaptador de barramento integrado no PHILIPS SJA1000

De modo comum a outros controladores, mas ao contrário do PHILIPS PCA82C200, o comparador interno da entrada do adaptador de barramento pode ser ultrapassado, transmitindo-se assim os sinais de um adaptador de barramento externo directamente ao controlador do protocolo, o que leva a um encurtamento do sinal de atraso interno do controlador.

Controlo de transmissão

Uma vez configurado, o controlador pode ser iniciado. O set ou o reset pode ser efectuado através do bit “*reset request*” no registo global de controlo (Tabela 3). O reset é efectuado de modo assíncrono, isto é, mesmo que ainda não se tenha completado o envio/recepção de uma mensagem, o controlador cessa de imediato a sua actividade.

Quando o controlador é iniciado, recebe de imediato todas as mensagens que circulam no barramento, armazenando aquelas que passarem pela filtragem e assinala ao controlador anfitrião através do bit “*receive buffer status*” que tem mensagens disponíveis para leitura. O controlador anfitrião pode assim ler o buffer sem ser incomodado com a chegada de novas mensagens.

O FIFO que está por detrás do buffer de recepção separa os dois processos. Quando o controlador anfitrião ler o buffer, vai activar o bit “*release buffer*”, o qual liberta o FIFO. Caso o FIFO contenha outra mensagem, activa-se novamente o bit “*receive buffer status*” e o controlador pode ler a nova mensagem, e assim sucessivamente. No entanto, caso o controlador anfitrião não consiga manter as leituras, acaba por acontecer uma sobrecarga de dados no FIFO, assinalada pelo controlador no bit “*overflow status*”, e a consequente perda das mensagens. Um exemplo deste armazenamento/leitura de mensagens pode ser observado na Figura 28.

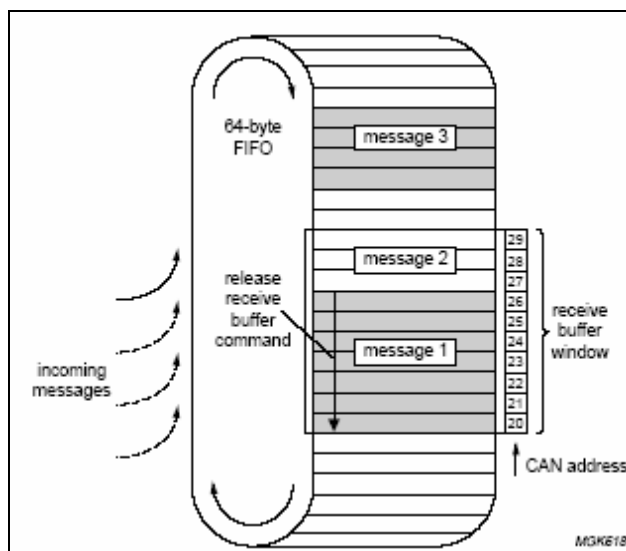


Figura 28 – Exemplo do armazenamento de uma mensagem no RxFIFO

Para enviar uma mensagem, coloca-se esta no buffer de transmissão do controlador e activa-se o bit “*transmission request*”. Se a mensagem puder ser enviada, o controlador activa os bits “*transmission complete status*” e “*transmission buffer status*”. A activação destes dois bits é necessária para o sucesso da transmissão,

tornando possível o abortar de uma ordem de transmissão dada. Por este motivo, o bit “*transmission abort*” está activado. Esta característica de abortagem activa a transmissão das mensagens de acordo com a prioridade.

A recepção de uma mensagem, o evento de sobrecarga de dados, o completar de uma transmissão de uma mensagem, exceder ou baixar o aviso de limites de erros e a transição do estado off do barramento podem ser assinalados ao controlador anfitrião por meio de interrupções. A activação das interrupções individuais é feita pela activação dos bits associados no registo de controlo. Caso ocorra uma interrupção, o controlador anfitrião pode identificar a sua fonte pela leitura do registo de interrupções que desactiva a saída da interrupção do controlador novamente.

Em conjunto com a saída de interrupção e a entrada de reset, também é fornecida uma saída de relógio, através da qual se pode fornecer um sinal de relógio para os outros controladores. O modo do sinal de entrada determina a função das linhas de controlo do barramento (ALE, RD e WR no modo Intel ou AS, E e RD/WR no modo Motorola).

Modo “PeliCAN”

Este modo de funcionamento consegue com que o PHILIPS SJA1000 tenha o suporte para o protocolo CAN especificação 2.0B, o que torna possível o envio e a recepção de mensagens no formato estendido, motivo pelo qual o buffer fica agora alargado a 13 bytes. No entanto, dependendo do tamanho total das mensagens recebidas, o FIFO pode conter de 4 a 32 mensagens.

Note-se, no entanto, que o filtro de aceitação neste modo é mais eficaz, uma vez que tanto o código de aceitação como o registo da máscara de aceitação tem 4 bytes, pelo mesmo as mensagens no formato estendido podem ter considerados todos os seus bits durante a filtragem.

Assim existem dois modos de efectuar a filtragem. No designado modo de filtro simples, os 4 bytes da mensagem passam pelo filtro. No que respeita às mensagens no formato estendido também é considerado o bit RTR, mas se a mensagem tiver o formato standard, além do bit RTR também podem ser incluídos os 2 primeiros bytes do campo de dados. No designado ‘modo de filtro duplo’, o mecanismo de filtragem consiste em 2 filtros independentes de 2 bytes cada, em que ambos são aplicados à mensagem. Basta que a mensagem seja aceite por um dos dois filtros para que seja armazenada no FIFO.

Além disso, o SJA1000 também dispõe de mais dois modos de funcionamento para aplicações para além das comunicações. No modo de apenas escuta elimina-se qualquer influência ao nível do barramento, isto é, mensagens transmitidas correctamente são aceites sem ser enviado o bit de “*acknowledge*”. Normalmente neste modo não é possível enviar mensagens. No designado modo de ‘auto-recepção’, as mensagens transmitidas pelo controlador são também aceites pelo seu FIFO, fornecendo assim uma maneira eficaz de verificar se as mensagens são capazes de passar pelo filtro de aceitação. Desta forma efectua-se também um auto-teste de forma a ver se a mensagem recebida é exactamente a que foi transmitida.

1.4.2 - Intel 82527

O INTEL 82527 é a evolução natural do INTEL 82526, fornecendo uma série de mecanismos adicionais com respeito à gestão de mensagens, não sendo, contudo, totalmente compatível nas funções dos pinos, uma vez que alguns destes eram usados no 82526 como “*chip selects*” e no 82527 são usados para o interface com o microcontrolador. Sendo um componente altamente compacto, ele efectua diversas tarefas de comunicação tais como transmissão e recepção de mensagens, filtragem de mensagens, transmitir buscas e interromper buscas com uma interacção mínima do microcontrolador anfitrião.

Sendo o primeiro da Intel a suportar o protocolo CAN 2.0B, pode transmitir, receber e filtrar mensagens com o formato estendido, sem no entanto deixar de suportar o anterior formato standard do protocolo 2.0A.

Na Figura 29 apresenta-se o diagrama de blocos do INTEL 82527.

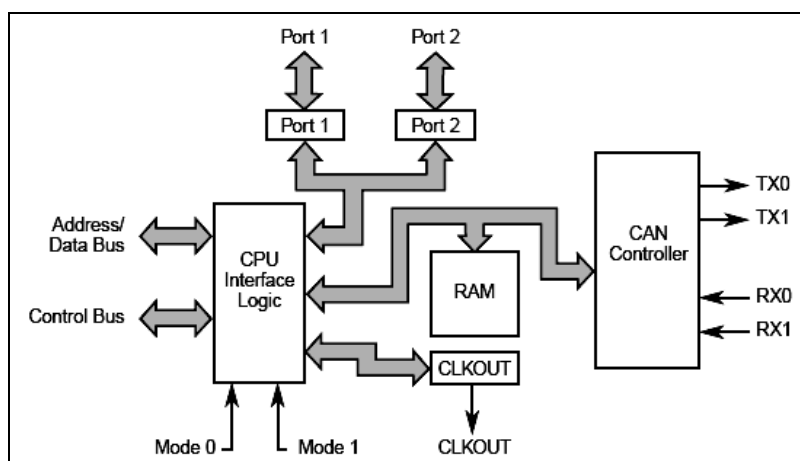


Figura 29 – Diagrama de blocos do funcionamento do INTEL 82527

O 82527 possui uma interface bastante poderosa para o CPU que é bastante flexível, de modo que podem ser ligados diferentes microcontroladores directamente. Esta interface pode ser configurada para microcontroladores com barramento de endereço/dados de 8 bits multiplexados, 16 bit multiplexados ou 8 bits não multiplexados, quer sejam arquitecturas Intel ou não. Também possui uma interface série flexível (SPI) que permite efectuar a interface quando não é pedido um interface paralelo. A selecção do barramento é efectuada mediante a ligação correspondente dos pinos de selecção de modo associados.

Este controlador possui 15 buffers com tamanho fixo de 8 bytes cada, sendo que 14 destes buffers podem ser configurados como buffers de recepção ou de transmissão e seguem o princípio “*FullCAN*”, enquanto o último buffer é apenas de leitura e segue o princípio “*BasicCAN*”, além de também ter uma máscara especial que lhe permite a selecção de diferentes grupos de identificadores de mensagem para receber.

Uma característica melhorada neste controlador é o facto de dar mais informação no que se refere a erros no barramento, em muito devido a um campo de bits existente no registo de estado, diferenciando assim os seguintes erros:

- Erro de “*bit stuffing*”
- Erro de formato
- Erro de “*acknowledge*”
- Erro “bit-1” (nível recessivo mas lido como dominante)
- Erro “bit-0” (nível dominante mas lido como recessivo)
- Erro CRC

Esta característica permite que um nó possa efectuar uma análise de erros do barramento e assim identificar a possível fonte de problemas.

Configuração

Antes de se proceder à transmissão de uma mensagem, é necessário efectuar a configuração dos parâmetros temporais do barramento, assim como o tipo de interface que se vai usar para o barramento, usando para esse efeito o ‘Registo Configuração do Barramento’. Será esta configuração que irá permitir o ultrapassar do comparador da entrada, ligar o pino Rx0 do controlador directamente a níveis TTL de um “*chip transceiver*” externo ou se um bit dominante é esperado com nível alto ou baixo. No normal funcionamento, as duas saídas de driver Tx0 e Tx1 funcionam geralmente no modo “*push-pull*” e complementam-se, isto é, no caso de um bit dominante se Tx0 está com nível baixo então Tx1 está com nível alto.

Filtragem de mensagem

O INTEL 82527 implementa uma máscara global de filtragem, que pode ser usada para mascarar globalmente quaisquer bits do identificador de uma mensagem a caminho, seja esta mensagem no formato standard ou no formato estendido.

Ao receber uma mensagem, esta passa pela máscara de filtragem para verificar se pode ser armazenada num dos 14 buffers existentes. Caso os bits do identificador não correspondam aos da máscara, é efectuada uma ligação à máscara local do buffer n.º 15 para verificar se a mensagem pode ser colocada neste buffer. Se continuar a não haver correspondência de bits, a mensagem é recusada.

Controlo de transmissão

Uma grande inovação que surgiu neste controlador em relação ao seu predecessor foi a possibilidade dos buffers poderem ser reconfigurados mas sem a necessidade de colocar o controlador num estado no qual não lhe é possível receber mensagens, tornando assim, por exemplo, possível transferir todas as mensagens que pretende enviar (mesmo com identificadores diferentes) por um único buffer de transmissão. Mesmo durante a reconfiguração de um buffer de transmissão os buffers mantêm a capacidade de receberem mensagens. Na Tabela 4 apresenta-se o mapa de registos do buffer de mensagens.

7	6	5	4	3	2	1	0	
MsgVal		TxIE		RxIE		IntPnd		Control Register 0
RmtPnd		TxRqst		MsgLst / CPUUpd		NewDat		Control Register 1
ID 28 – 21								Arbitration Register 0
ID 20 – 13								Arbitration Register 1
ID 12 – 5								Arbitration Register 2
ID 4 – 0								Arbitration Register 3
DLC				Dir	Xtd			Configuration Register
								Data Byte 1
								Data Byte 2
								Data Byte 8

Tabela 4 – Mapa de registos do buffer de mensagens do INTEL 82527

Para se efectuar o controlo da transmissão de uma mensagem ou aceder ao seu buffer, usam-se os bits que estão disponíveis de uma forma especial no descritor de registos dos buffers e mensagens (Tabela 4).

Para se poder configurar o buffer, é necessário primeiro desactivá-lo, recorrendo-se ao reset do bit ‘MsgVal’, introduzindo-se de seguida o identificador de mensagem. Depois é preciso introduzir as informações necessárias à transferência, como sejam as direcção dos dados (receber ou enviar), seu tamanho e tipo de protocolo (estendido ou standard), no registo de configuração, após o que se pode então activar (ou não, conforme se pretende) a interrupção de transmissão ou de recepção através dos bits ‘TxIE’ ou ‘RxIE’, respectivamente.

1.5 - Microcontroladores com controladores CAN integrados

Actualmente, com a evolução tecnológica natural, são cada vez em maior número os microcontroladores com controlador CAN já integrado, não havendo praticamente um fabricante que não tenha pelo menos uma versão deste. Devido às grandes vantagens que esta nova tecnologia abarca, das quais se destacam maior rapidez de acesso uma vez que não se usa o barramento externo, menor carga de processamento para o CPU, custo dos chips, requisitos de espaço rígidos, mas suficientemente inteligente para poder efectuar algumas funções locais como seja o pré-processamento de dados, os microcontroladores com controlador CAN integrado são cada vez mais importantes. Na Figura 30 apresenta-se um diagrama de blocos típico de um nó CAN baseado num microcontrolador com controlador CAN integrado.

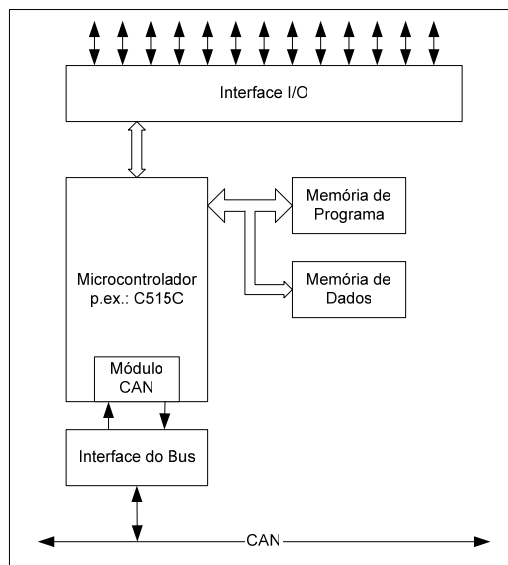


Figura 30 – Nó CAN baseado num microcontrolador com controlador CAN integrado

Para além disso, o número crescente de controladores CAN integrados num único microcontrolador (2 ou mais) está a torna-se cada vez mais importante, visto que se cria assim a possibilidade de interligação entre várias redes CAN num único sistema. O nó de ligação que fornece esta ponte entre redes, normalmente também proporciona a filtragem de mensagens entre essas redes, o que leva a uma menor sobrecarga do barramento.

De seguida, apresenta-se na Tabela 5 a comparação efectuada entre alguns microcontroladores com controladores CAN integrados existentes no mercado, fazendo-se depois uma pequena descrição.

Fabricante	Referência:	Memória	Tipo de CAN	Tipo CPU	nº buffers	Interrupts	Interfaces	Características particulares
ATMEL	T89C51CC01	1.2 kB RAM+XRAM 32 kB Flash Code 2 kB Flash Boot 2 kB EEPROM	2.0A 2.0B	80C51 Core 8 bit	15 buffers	14 com 4 níveis de prioridade	5 portas I/O A/D 10 bit	3 Timer/Counter (16 bit) Sincronização em rede 5 canais PCA
DALLAS	DS80C390	4 kBytes SRAM	2.0B	80C51 Core 8 bit	15 buffers cada	16 (6 por acesso externo)	4 portas I/O (8 bit) 2 RS232 Full Duplex	3 Timer/Counter (16 bit) Coprocessador matemático Duplo controlador CAN
FUJITSU	MB90F394H	384 kB Flash ROM 10 kB RAM	2x 2.0A 2x 2.0B	F ² MC-16LX 16 bit	16 buffers c/ prioridades	8 por acesso externo	Portas I/O 8 canais A/D 10 ou 8 bit 2 RS232 Full Duplex	3 timer (16 bit)
INFINEON	C167CR	128/32 kB ROM 2 kB RAM 2 kB XRAM	2.0B	C166 Core 16 bit	15 buffers	56 c/ 16 níveis de prioridade 8 via periféricos	Até 111 linhas I/O 16 canais A/D (10 bit)	2x 5 timer
INFINEON	C515-8R	64 kB ROM até 64 kB externa 256 B RAM 2 kB XRAM	2.0A 2.0B	8051 compatível 8 bit	15 buffers 256 B na memória ext	7 vectoriais c/ 4 níveis de prioridade	6 portas I/O digitais 8 entradas analógicas 8 canais A/D 10 bit SPI	3 timer/counter (16 bit)
INTEL	87C196CB	56 kB EPROM 1.5 kB register RAM 512 B code RAM	2.0B	MCS 96 Kx/Jx 16-bit	15 buffers	37 c/ prioridades	7 portas 8 canais A/D 10 bit RS232	2 counter/timer (16 bits) barramento externo de 8/16 bit
Microchip	PIC18F248	16 kB Flash 768 RAM 256 EPROM	2.0B		2 buffers Rx 3 buffers Tx c/ prioridade individual	3 por acesso externos	22 portas I/O 5 canais A/D 10 bit	3 timer (16 bit) 1 timer (8 bit) 6 filtros de c/ prioridades
Motorola	MC68HC705X4	4096 EPROM 176 RAM	MCAN 2.0A	MC68HC05 8 bit	Rx 10 B, Tx 10 B	4 por hardware 1 por software	8 portas I/O	Timer 16 bit
PHILIPS	XA-C37	32 kB EPROM 1 kB DATA RAM 512 B XRAM (Até 8kB)	2.0B	XA-C3 Core 16 bit	32 buffers	42 vectoriais c/ 7 por software	4 portas I/O (8 bit) 1 série 1 SPI	3 counter/timer (16 bit)

Tabela 5 – Tabela comparativa de microcontroladores com controladores de protocolo CAN integrados

1.5.1 - Microchip PIC18Fxx8

A Microchip Technology tem uma linha completa de microcontroladores de 8 bits com controlador CAN integrado. Também tem a tecnologia ECAN integrada em muitos microcontroladores PIC. A tecnologia ECAN não é mais que uma interface CAN 2.0B flexível especialmente concebida para aplicações de controlo embebidas.

O diagrama de blocos do microcontrolador PIC18F248 (microcontrolador de gama intermédia da família PIC18Fxx8) pode ser observado na Figura 31.

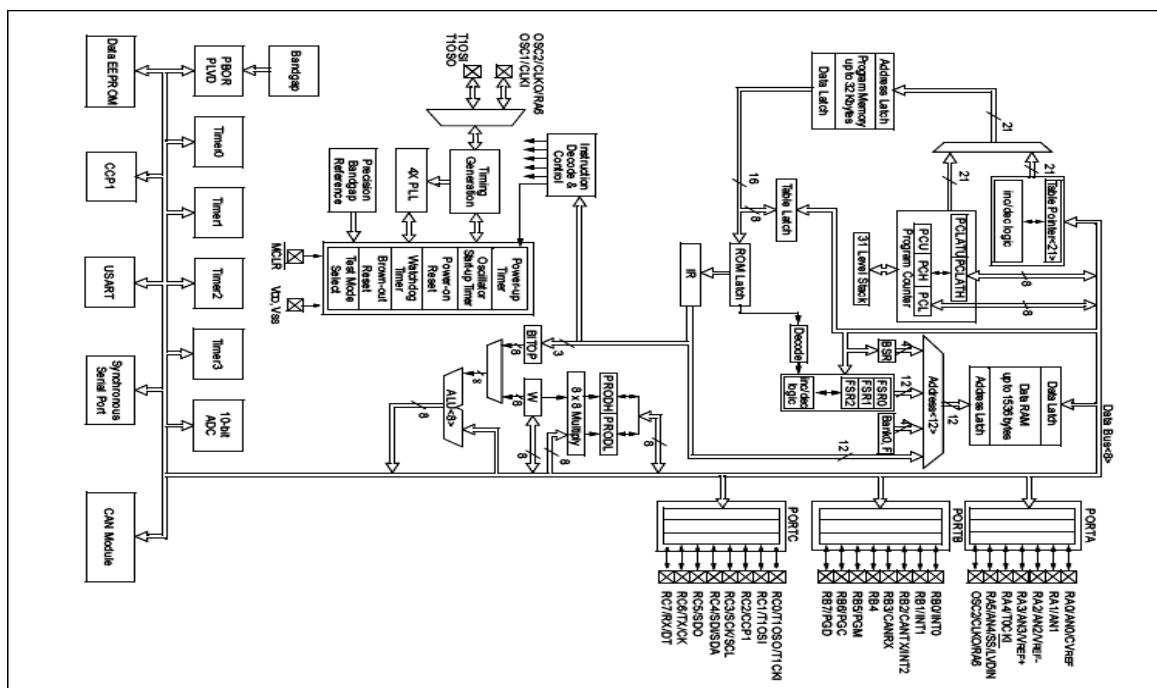


Figura 31 – Diagrama de blocos de funcionamento do MICROCHIP PIC18F258

Outras características destes microcontroladores são:

- Endereçamento de memória de programas até 2 MB
- Endereçamento de memória de dados até 4 kB
- 3 pinos para interrupções externas
- 3 temporizadores (1x 8 bit; 2x 16 bit)
- Conversor A/D de 10 bits, até 8 canais
- Em conformidade com o protocolo CAN 2.0 B activo com:
 - 8 bytes para tamanho da mensagem
 - 2 buffers de recepção
 - 3 buffers de transmissão
 - 6 filtros de aceitação com 29 bits
 - Filtros de aceitação com prioridades
 - Múltiplos buffers de recepção para mensagens de alta prioridade para evitar perdas por sobrecarga

Características específicas do PIC18F248 são as seguintes:

- 16 kB memória “flash”, 768 B SRAM, 256 B EEPROM
- 22 linhas I/O
- 6 canais A/D de 10 bit

1.5.2 - Philips XA-C3

O PHILIPS XA-C3 é um microcontrolador de 16 bits de alta performance pertencente à família 80C51XA (“*eXtendend Architecture*”), sendo composto por um microcontrolador XA-G3 e um controlador CAN PHILIPS SJA1000.

Na Figura 32 apresenta-se o diagrama de blocos do XA-C3.

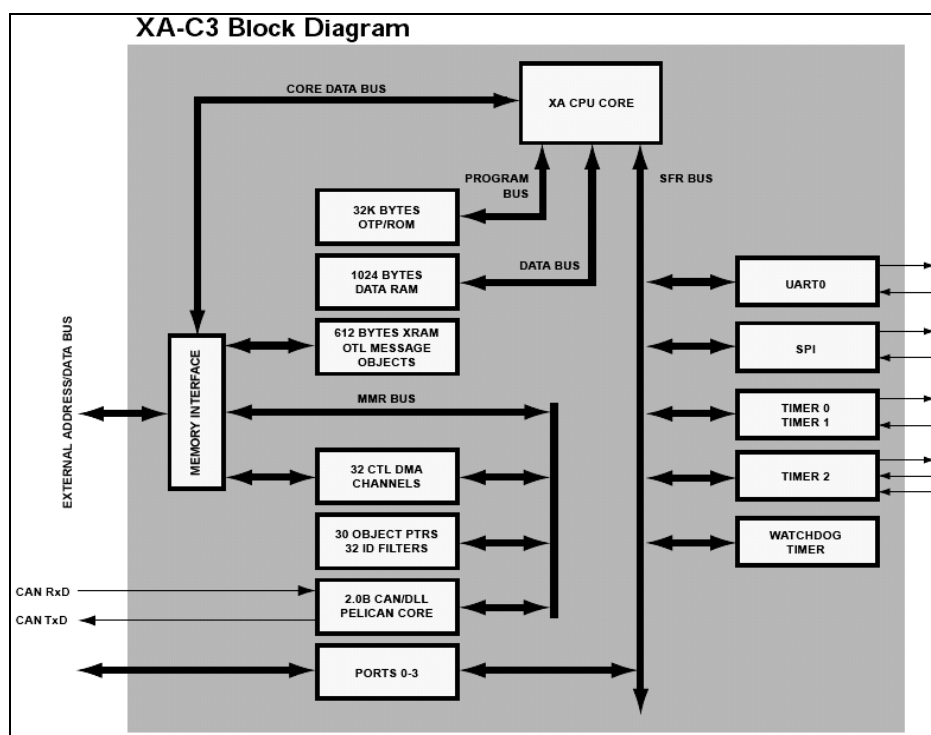


Figura 32 – Diagrama de blocos do funcionamento do PHILIPS XA-C3

Algumas características específicas do XA-C3 são:

- 1 interface série e 1 interface SPI – *Synchronous Peripheral Interface*
- 1024 bytes SRAM de dados
- 42 interrupções vectoriais, incluindo 13 eventos mascaráveis, 7 interrupções por software, 6 exceções entre outros.
- 32 kB de memória EPROM/ROM
- 3 temporizadores/contadores
- 4 portas I/O de 8 bit

O controlador CAN integrado tem 32 buffers de mensagens configuráveis como receptores ou transmissores e 32 filtros de aceitação, consistindo cada um num código de aceitação de 30 bit e uma máscara de 29 bit.

Os 32 buffers estão localizados na memória XRAM e são geridos por 32 canais DMA. O tamanho de cada buffer é configurável desde 2 até 256 bytes. No caso do sistema conter apenas um “*chip*”, existe um buffer com 512 bytes na XRAM (“*on-chip*”), independente do 1 kB de RAM de dados (“*on-chip*”) o qual é extensível até 8kB (“*off-chip*”).

2 - As interfaces

Para um computador não é suficiente existir só a placa de nó CAN para o integrar na rede CAN. Também é necessário uma placa de interface apropriada para o computador para que este possa efectuar as respectivas comunicações e monitorizações com a restante rede.

De entre os vários fabricantes e/ou as necessidades dos equipamentos existem actualmente diversos formatos de placas que podem ser usadas para esta ligação tais como:

- Placas de interface ISA
- Placas de interface de porta LPT (vulgo porta da impressora)
- Placas de interface para portas USB
- Placas de interface para portas PCMCIA
- Placas de interface PC-104
- Placas de interface PCI

Embora as placas de interface do tipo PC-104 sejam as mais vulgarizadas devido aos seu uso a nível industrial, serão utilizadas placas de interface PCI uma vez que são estas as que se usam num computador vulgar tal como o que se vai usar neste trabalho.

Assim, as placas de interface que se determinaram como apropriadas e das quais se irá seleccionar uma para o trabalho foram (Marca: Modelo):

- CNI Informatica: P2CAN
- ESD Electronics: CAN-PCI/331
- I+ME ACTIA: PCI-IntelliCAN
- IXXAT Automation: iPC-I 165/PCI
- IXXAT Automation: iPC-I 320/PCI
- JANZ: CAN-PCI/K2
- JANZ: CAN-PCI2/1
- KVASER Advanced CAN Solutions: PCICan II-S
- Vector Informatik: CAN-AC2-PCI
- Vector Informatik: CANboardXL

Das que mais se adequam à realização deste trabalho será realizada uma descrição detalhada das suas principais características e modo de funcionamento/programação. A seguir apresenta-se a Tabela 6 com o resumo das características destas interfaces (mesmos as que não serão detalhadas) para uma melhor comparação.

Fabricante	Referência	Microcontrolador	Memória	Controlador CAN	Interfaces I/O	Informação Adicional
IXXAT Automation	iPC-I 165/PCI	Infineon C165	512 kB SRAM (até 2 MB) 512 kB Flash (até 1 MB) 8 kB DPRAM	PHILIPS SJA1000 INTEL 82527	RS232 RS485 DeviceNET	-
IXXAT Automation	iPC-I 320/PCI	Dallas DS80C320	2 x 64 kB SRAM 128 kB FLASH 4 kB DPRAM	PHILIPS SJA1000 INTEL 82527	RS232 RS485 DeviceNET	-
JANZ	CAN PCI (MOD-PCI + VMOD-ICAN3)	Motorola MC68332	256 kB SRAM 64 kB DPRAM (16 bit) 128 kB Flash EPROM	PHILIPS SJA1000	RS 232	-
JANZ	CAN-PCI2	Motorola MC68332	256 kB SRAM 64 kB DPRAM (16 bit) 256 kB Flash EPROM	PHILIPS SJA1000	RS232	Upload código no uC
KVASER	PCICan II -S	Mitsubishi/Renesas M16C/62N	10 kB RAM (uC) 256 kB Flash (uC)	on-chip	8 canais A/D 10 bits 2 canais D/A 8 bits 10 portas I/O (uC)	3x UART 5 output / 6 input timers 2 canais DMAC
National Instruments	PCI-CAN SERIES 2	Intel 80386EX	-	PHILIPS SJA1000	-	Alta velocidade
Vector Informatik	CAN-AC2-PCI	Infineon C165	2 kbytes internal RAM	PHILIPS SJA1000	9 portas I/O (uC)	3 canais timer/counter (16 bits)
Vector Informatik	CANboardXL	Atmel AT91	4 kbytes internal RAM	PHILIPS SJA1000	4 portas I/O (uC) 2x USART	-

Tabela 6 – Tabela comparativa das interfaces CAN para PC

2.1.1 - IXXAT Automation: iPC-I165/PCI e iPC-I320/PCI

Sendo de uma marca com bastante prestígio e dedicada ao desenvolvimento de produtos CAN, a placa iPC-I165/PCI é uma solução bastante versátil que permite ligar até duas redes CAN, com o suporte para os protocolos CAN 2.0A e CAN 2.0B activo, permitindo também uma configuração automática do tipo “*Plug and Play*”.

Na Figura 33 apresenta-se o aspecto que esta placa de interface possui.

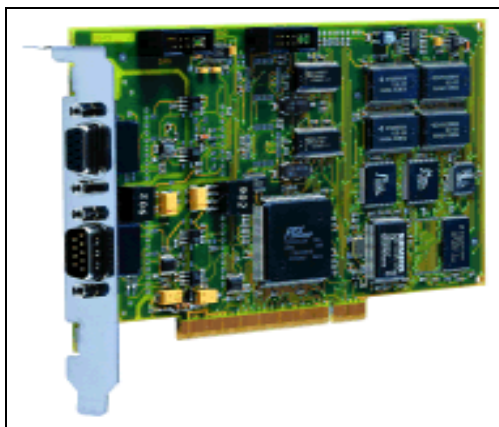


Figura 33 – IXXAT Automation iPC-I165/PCI

O software que o microcontrolador Infineon C165 de 16 bits tem de base permite um pré-processamento, filtragem e armazenamento das mensagens (com marcação temporal) assim como a transmissão de mensagens CAN de tempo-real. Contudo, o catálogo da marca faz referência à possibilidade de se poder efectuar o download de melhoramentos de “*firmware*” ou mesmo “*firmware*” personalizado.

O mesmo microcontrolador tem a capacidade de lidar com duas redes em simultâneo sem perda dados, mesmo em sistemas com elevada carga no barramento.

Outras características deste interface que merecem destaque são:

- 512 kbytes de memória SRAM onboard (base), ampliáveis até 2 Mb
- 512 kbytes de memória Flash onboard (base), ampliáveis até 1 Mb
- 8 kbytes de memória DPRAM onboard
- Até 2 controladores de rede CAN Philips SJA 1000 e/ou Intel 82527
- Opções para isolamento galvânico, interfaces série (RS-232 ou RS-485) ou DeviceNet

As diferenças para o outro modelo, a placa iPC-I320 (apresentada na Figura 34), são em concreto:

- microcontrolador Dallas DS80C320
- 2x 64 kbytes de memória SRAM onboard
- 128 kbytes de memória Flash onboard
- 4 kbytes de memória DPRAM onboard

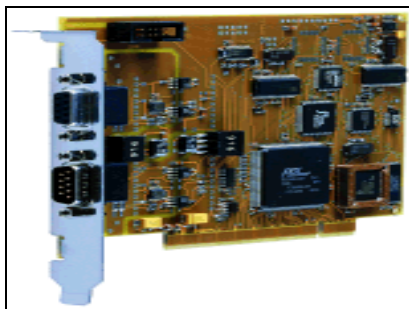


Figura 34 – IXXAT Automation iPC-I320/PCI

Contudo, em ambas as placas as opções mantêm-se iguais.

2.1.2 - JANZ: CAN-PCI

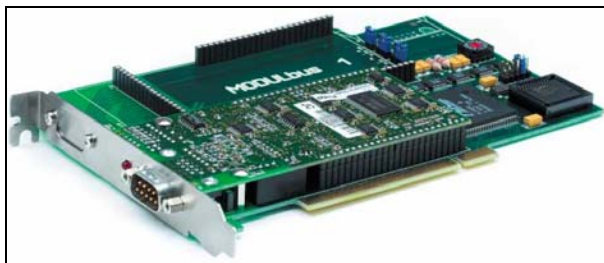


Figura 35 – JANZ CAN-PCI

Esta interface na realidade é composta por duas partes independentes: a placa de interface PCI (MOD-PCI) e a placa para a interface do protocolo CAN (VMOD-ICAN3 activo ou VMOD-FCAN passivo). A placa de interface PCI pode conter até dois módulos para o interface CAN, sendo que o tem interesse para este trabalho o VMOD-ICAN3, o qual possui um microcontrolador MC68332, controlador CAN SJA1000, 256 kbytes de memória SRAM (16 bits), 64 kbytes de memória DPRAM (16 bits) e 128 kbytes memória Flash EPROM (8 bits). Este conjunto toma a referência CAN-PCI/K2. Também existe a possibilidade de se ter a interface opticamente isolada.

3 - Os módulos

Para haver ligação de qualquer equipamento a uma rede CAN é necessário existir a interface apropriada. Conforme os fabricantes e/ou as necessidades dos equipamentos existem diversos formatos de placas que podem ser usadas para esta ligação. De entre os vários formatos destacam-se essencialmente dois tipos:

- SBC (“*Single Board Computer*”) – Computador em Placa Simples
- Placas de Desenvolvimento e Expansão (“*Evaluation Boards*”)

Os SBC caracterizam-se por possuírem um microcontrolador e um pequeno conjunto de componentes com os quais formam um computador simples, não sendo na maioria dos casos muito maior que um vulgar CPU de computador. No caso de o microcontrolador não possuir integrado o controlador CAN, este está colocado externamente na mesma placa (mas sendo por vezes opcional). O uso dos SBC leva ao desenvolvimento de todo o restante circuito do nó (alimentação, sensores, ligações a equipamentos a rede, etc.) e da sua programação. A vantagem destes circuitos é a sua flexibilidade de adaptação, visto que não foram criados para uma aplicação específica.

Com as “*Evaluation Boards*”, tudo o que é necessário ao nó já está montado no circuito, pelo que apenas é necessária alguma adaptação nas ligações aos sensores ou à rede, razão pela qual costumam trazer uma área livre de componentes designada de “*Wrap Wired Area*”. Mas continua a ser necessário desenvolver a programação do software. Contudo, a grande desvantagem (para além do preço) é que estas placas não são muito flexíveis, sendo muitas das vezes desperdiçadas diversas funções existentes nas mesmas.

Também é de realçar que embora a grande maioria dos microcontroladores destas placas sejam programados segundo a linguagem *Assembly* (vulgarizada no Atmel 8051), é necessário hardware apropriado do fabricante para se efectuar a programação dos respectivos microcontroladores.

Na Tabela 7 apresenta-se o resumo das características destes módulos (mesmos os que não serão detalhados) para uma melhor comparação.

Fabricante	Referência	Microcontrolador	Memória (onboard)	Interfaces I/O	Outras Características
KEIL SOFTWARE	MCB167-NET	INFINEON C167 ST ST10F 167/168 16 bit	1 Mbyte high-speed RAM 1 Mbyte Flash ROM (opcional)	CS8900A Ethernet Controller 1 interface série RS232 1 A/D (10 bit) [P5] 5x I/O (16 bits) + 4x I/O (8 bits)	
PHYTEC	phyCORE ADuC8xx	ANALOG DEVICES ADuC812 8 bit	8 kB Flash on-chip 128 a 512 kB onboard 640 bytes Flash EPROM 128 a 1024 kbytes SRAM 2 a 8 kbytes I2C EEPROM (op)	RS232 ou RS485 8 canais A/D (12 bit) 2 D/A (12 bit)	I ² C RTC Fonte Tensão para Referência Sensor de temperatura
PHYTEC	phyCORE XACx	PHILIPS XA-C37 16 bit	256 a 2048 kbytes Flash externa 256 a 1024 kbytes RAM 2 a 8 kbytes SPI EEPROM/FRAM	RS232 ou RS485 Conversor A/D N/A	I ² C RTC 3 timer/counter
RIGEL CORPORATION	R-515JC	INFINEON C515C 8 bit	32 kbytes SRAM (opcional 128 k ~512 k) 32 kbytes monitor EPROM (opcional 64 k)	1 RS232 + 2 RS232/RS485 opcional 12 bits I/O uso geral 8 canais A/D (10 bit) [CPU]	RTC opcional
RIGEL CORPORATION	R-167JIF	ST ST10R167 ST ST10F168 16 bit	256K Flash interno opcional 512K memória RAM 512K memória Flash externa	Interface RS232 111 linhas de I/O uso geral 16 canais A/D (10 bit) [CPU]	5 timer (16 bit)

Tabela 7 – Tabela comparativa dos módulos CAN para nós

3.1 - KEIL Software

Embora esteja mais especializada em software de análise, simulação e programação para redes CAN, a KEIL software possui algumas interfaces para estas redes. Destacam-se assim as “*Evaluation Boards*” MCB517AC e MCB167-NET.

3.1.1 - MCB167-NET “Evaluation Board”

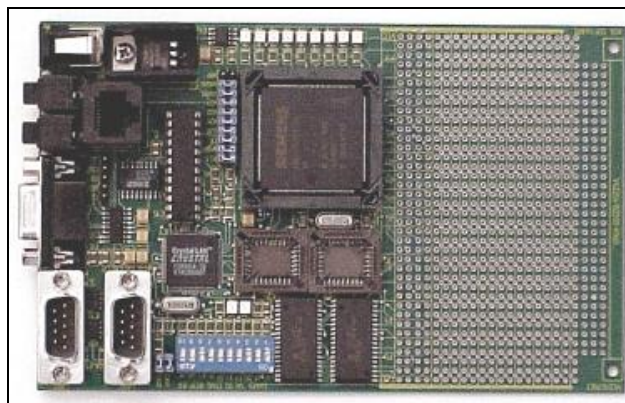


Figura 36 – KEIL Software MCB167-NET Evaluation Board

Embora tenha alguma versatilidade, esta “*Evaluation Board*” é baseada no microcontrolador de 16 bits Infineon C167 ou ST Microelectronics ST10F167 ou ST10F168, os quais já tem o controlador CAN integrado. Sendo uma das placas mais completas, para além da ligação à rede CAN pelo adaptador de barramento Siliconix Si9200EY também possui 1 interface série RS232 e uma interface para Ethernet, controlado pelo CS8900A. Como portas I/O, possui um conjunto de cinco a 16 bits e quatro a 8 bits, assim como 1 conversor A/D de 10 bits.

Na Figura 37 apresenta-se o diagrama de blocos que compõem o circuito.

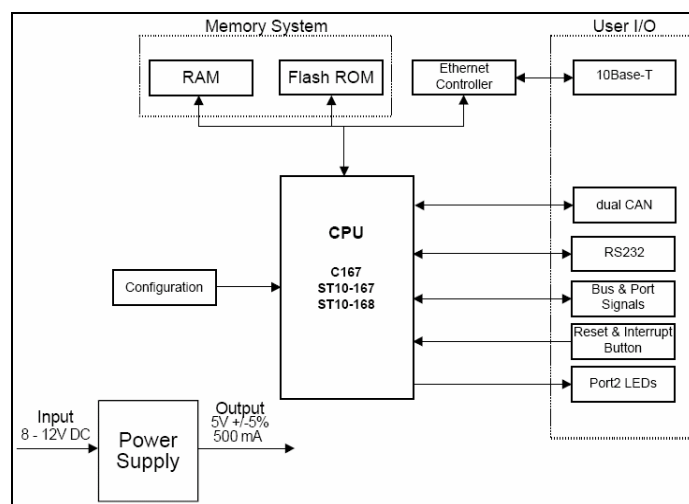


Figura 37 – Diagrama de blocos da placa MCB167-NET

Contendo internamente no microcontrolador 3 temporizadores/contadores, como capacidade de memória possui 1 Mbyte de memória RAM, embora se possa acrescentar mais uma memória Flash ROM de 1 Mbyte.

3.2 - Os Módulos SBC da Phyttec

Possuidora de uma larga gama de placas para nós de rede CAN, a PHYTEC destaca-se pelos seus módulos SBC muito versáteis e de baixo custo, os quais depois de programados via um kit de desenvolvimento, podem ser aplicados aos nós da rede CAN como se fossem um único circuito integrado. Desenvolvidos segundo as arquitecturas mais vulgares (8, 16 e 32 bits) estes módulos podem ter dimensões físicas muito variadas, existindo também as versões KitCON, as quais englobam numa única peça o módulo (na maioria dos casos phyCORE) e a “*Evaluation Board*”, sendo assim necessária apenas a programação.

Como módulos de 8 bits temos:

- miniMODUL 515C
- microMODUL 8051 w/ CAN
- phyCORE 591
- phyCORE ADuC8xx
- phyCORE DS80C390
- phyCORE T89C51CC01

Como módulos de 16 bits temos:

- DIPmodul 164
- miniMODUL 167CR
- nanoMODUL 164
- phyCORE 161
- phyCORE 167
- phyCORE ST10F168
- phyCORE 167CS HS/E
- phyCORE ST10F269 HS/E
- phyCORE XACx
- phyCORE XC16x com Ethernet/CAN

3.2.1 - phyCORE ADuC8xx



Figura 38 – PHYTEC phyCORE ADuC812

Tendo a possibilidade incorporar um de 2 microcontroladores da Analog Devices (o ADuC812 ou o ADuC824), este SBC é o um dos poucos que contém em simultâneo conversores A/D e D/A (derivados dos microcontroladores) sendo o primeiro de 12 bits com 8 canais e o segundo duplo de 12 bits (se for o microcontrolador ADuC812, o ADuC824 tem 1 conversor A/D de 24 bits e outro de 16 bits para um total de 5 canais, para além de 1 conversor D/A de 12 bits).

Este módulo não possui controlador CAN integrado, mas possui espaço para se interligar um controlador CAN Philips SJA 1000. Também o relógio I²C de tempo-real é opcional, assim como a bateria para backup. Para além dos conversores, o SBC também possui uma interface série que pode ser configurável como RS-232 ou RS-485.

A memória “*on-chip*” do módulo consiste em 8 kbytes de memória CODE Flash e 640 bytes de memória DATA Flash. Mas esta pode ser ampliada desde 128 kbytes até 512 kbytes com uma memória Flash externa (“*onboard*” na SBC) ou desde 128 kbytes até 1 Mbyte em memória SRAM (também “*onboard*” na SBC). Opcionalmente, ainda se pode ter desde 2 kbytes até 8 kbytes de memória EEPROM I²C.

Como características adicionais este SBC possui um sensor de temperatura, assim como uma fonte de tensão para referência. Note-se que tal como os restantes módulos da família phyCORE, o phyCORE ADuC8xx também tem as funções do microcontrolador estendidas para os pinos dos topos do módulo.

Na Figura 39 apresenta-se o diagrama de blocos que constituem este SBC.

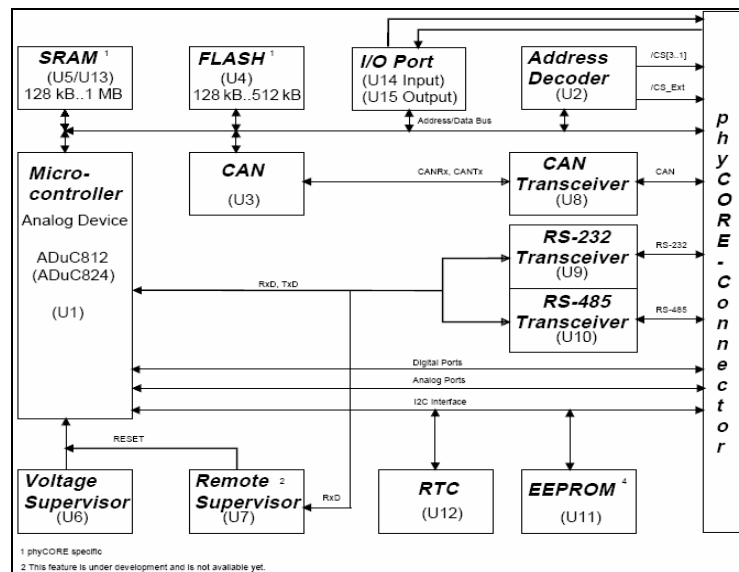


Figura 39 – Diagrama de blocos da phyCORE ADuC812

3.2.2 - phyCORE XACx

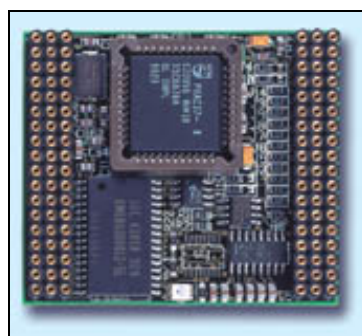


Figura 40 – PHYTEC phyCORE XA-C3

O phyCORE XAC3 tem como base o microcontrolador XA-C37 de 16 bits da Philips, o qual já integra um controlador CAN 2.0B on-chip baseado no SJA1000.

Para aplicações que exigem quantidades altas de memória, este microcontrolador pode ser expandido em memória CODE até 1 Mbyte e em memória XDATA também até 1 MB, totalizando assim 2 MB de expansão.

Para além das características inerentes do microcontrolador, o módulo completo tem adicionalmente as seguintes:

- 256 kbytes de memória Flash externa (expansíveis a 2 MB)
- 256 kbytes de memória SRAM externa (expansíveis a 1 MB)
- 2 kbytes de SPI-EPROM (expansíveis a 8 kbytes)
- Interface RS232 e/ou RS485

Na Figura 41 pode-se observar o diagrama de blocos deste módulo.

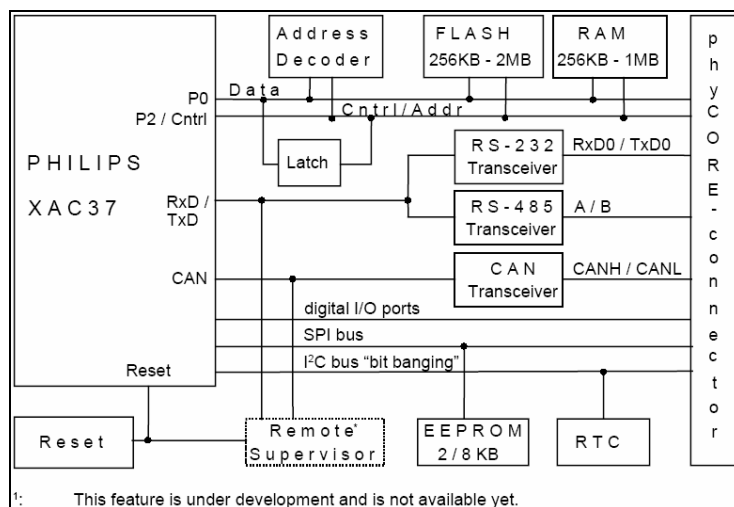


Figura 41 – Diagrama de blocos do phyCORE XA-C3

4 - Síntese

É necessário analisar cuidadosamente o controlador CAN e as suas especificações no momento de desenvolver os nós de uma rede CAN. Embora existam diversos componentes no mercado de diversos fabricantes, com preços e métodos de entrega diferentes, apenas dois conceitos básicos são considerados: o “*BasicCAN*” e o “*FullCAN*”.

O princípio “*FullCAN*” é mais apropriado (e por vezes necessário) para altas taxas de transmissão, um número pequeno de mensagens a receber por cada nó e uma baixa carga de permissibilidade por parte do controlador anfitrião ou controladores anfitriões de menor capacidade [Ets, 2001]. No entanto, se tiver que receber grande quantidade de mensagens ou é tolerável uma pequena sobrecarga no controlador anfitrião, o uso do conceito “*BasicCAN*” será menos dispendioso [Ets, 2001].

É por estes motivos que estes dois princípios se encontram implementados nos controladores de segunda geração. Mas as diferenças entre os dois princípios não se referem ao modo como processam o protocolo, pelo que ambas as implementações dos princípios podem ser usadas numa mesma rede [Ets, 2001].

Para se efectuar a construção de uma infra-estrutura do tipo CAN é preciso analisar todas as placas que se terão que integrar no sistema, principalmente no que respeita ao tipo e capacidade de memória, quantidade de entradas e saídas analógicas e digitais, como efectuar a programação dos microcontroladores e das memórias e também o seu custo.

Após a análise de todas estas características, chegou-se à conclusão que para a placa que deverá integrar os nós da rede as melhores soluções são os módulos phyCORE da Phytex, mais concretamente o módulo microMODUL 8051 com CAN, o módulo phyCORE ADuC8xx e o módulo phyCORE T89C51CC01 para 8 bits, sendo para 16 bits as soluções mais aconselhadas os módulos DIPmodul 164, nanoMODUL 164 e phyCORE 167. Note-se que é necessário adquirir uma placa do tipo “*evaluation*”

board”, uma vez que a programação dos microcontroladores é feita através desta. A escolha destes módulos deve-se principalmente à sua compacta dimensão e à possibilidade de desenvolver todo o restante hardware à volta destes, considerando-os como se fossem circuitos integrados de grande dimensão.

Contudo, as placas da Rigel Corporation oferecem também uma boa alternativa, uma vez que estas possuem características muito próximas da Phytex e já possuem um espaço para “*wire-wrap*” para desenvolvimento de hardware adicional. No entanto têm como desvantagem o seu tamanho.

No que respeita às placas de interface PCI para CAN, a escolha mais apropriada recai sobre as soluções apresentadas pela IXXAT Automation no seu modelo iPC-165/PCI, JANZ no seu modelo CAN-PCI2. É de salientar que o microcontrolador desta última é o modelo Motorola MC68332 de 32 bits.

As ferramentas usadas na programação são quase todas derivadas da Keil Software ou da Tasking Software, tendo um princípio de funcionamento idêntico.

Assim, a solução final escolhida foi:

- Módulo phyCORE ADuC812 da Phytex, para os nós independentes.
- Placa de interface iPC-I320 da IXXAT Automation, para recepção dos dados via rede CAN no computador.

Esta escolha que recaiu sobre os módulos da Phytex, tem a ver com o facto de estes serem os mais completos mas manterem a simplicidade de forma a serem facilmente integrados em projectos já existentes. Também o facto de permitir ser programa recorrendo à linguagem C é uma grande vantagem. O módulo específico ADuC812 é o mais completo de todos, com as seguintes características: canais analógico-digitais e digita-analógicos, real-time clock, expansão de memória e controlador CAN externo ao microcontrolador.

Capítulo 4

Análise teórica de comunicações de tempo-real em redes CAN

Capítulo 4 - Análise teórica de comunicações de tempo-real em redes CAN

1 - Introdução

Uma das maiores dificuldades em engenharia de sistemas de tempo-real é o desenvolvimento de um mecanismo para a análise do comportamento temporal destes sistemas. Neste tipo de aplicações normalmente existem sensores inteligentes, que necessitam do cumprimento de dois requisitos para o correcto funcionamento em rede: 1) o protocolo deve dar acesso rapidamente ao barramento às mensagens prioritárias e 2) permitir também o suporte a mensagens não prioritárias, periódicas ou não.

Muito do trabalho realizado nesta área tem considerado como algoritmo de base o escalonamento por prioridades fixas, tendo sido dirigido ao escalonamento de mensagens em barramentos de partilha por difusão, particularmente do tipo “prioridades”. Mas estes trabalhos assumem como certeza o comportamento ideal da interface entre o processador anfitrião e o adaptador de comunicações do barramento. Contudo, uma implementação específica pode não condizer com esta certeza, pelo que investigações mais recentes tem trabalhado com um protocolo de barramento em particular e diversas implementações de diferentes origens. Neste trabalho também iremos verificar como pequenas alterações numa mesma implementação de uma interface podem ter grandes consequências no desempenho temporal das mensagens.

O protocolo usado para ser efectuada a análise foi o Controller Area Network, vulgarmente designado pelas suas iniciais, CAN. Devido ao módulo escolhido para a componente experimental deste trabalho (Capítulo 3, ponto 3.2), o controlador de interface que será sujeito de estudo é o Philips SJA1000, o qual corresponde à evolução natural do controlador Philips 82C200 (Capítulo 3, ponto 1.4). Sendo o CAN um barramento de difusão por detecção de colisão, a componente mais importante no que respeita à performance de tempo-real é o uso do identificador como prioridade. O modo como a detecção e transmissão na rede é efectuada pode ser verificada no Capítulo 2, ponto 2.1. O formato de mensagem CAN contém 47 bits de informação de controlo do protocolo (identificador, dados CRC, bits de sincronismo e acknowledge, etc.) e, dependendo do padrão de bits da mensagem, pode ter de 0 a 19 stuff bits para 64 bits de dados.

2 - Características de carga na rede

Na indústria actual, os principais equipamentos de comunicação são os controladores (CPUs e/ou PLCs), actuadores e sensores. Embora alguns destes equipamentos efectuem periodicamente comunicações entre si (como será o caso dos controladores), outros apenas efectuam a comunicação na ocorrência de um evento específico (por exemplo, o funcionamento de sensores inteligentes). Para além destes casos, também existem situações em que operadores necessitam de informação de vários equipamentos, informação essa que não está sujeita a restrições temporais. Desta forma, classificam-se assim as mensagens em três grandes grupos ou categorias:

1. Mensagens periódicas com metas temporais apertadas/rígidas
2. Mensagens esporádicas com metas temporais apertadas/rígidas
3. Mensagens aperiódicas não de tempo-real

2.1 - Mensagens periódicas

Como exemplo comum e típico deste tipo de mensagens considera-se o controlo de motores de posicionamento. O controlador tem que periodicamente realizar uma amostragem da posição/velocidade do motor e efectuar as correcções necessárias. Este tipo de mensagens possuem metas temporais do tipo crítico bastante apertado, uma vez que se uma mensagem sofrer qualquer tipo de atraso para lá da sua meta temporal, poderá provocar um desvio dos servos com graves consequências.

De um modo geral, qualquer sistema que tenha que efectuar um processo predeterminado terá mensagens periódicas com metas temporais críticas, em grande parte devido ao ciclo “sensor – controlador – actuador” periódico.

Note-se que uma mensagem periódica simples tem múltiplas invocações, cada uma espaçada de um período da seguinte, pelo que ao se usar o termo “mensagem” para fazer referência à periodicidade, é referente a todas as invocações da mesma.

2.2 - Mensagens esporádicas

Pode ser observado na natureza que todos os eventos que ocorrem são esporádicos. Contudo, se estes tiverem uma ocorrência frequente, então é possível aplicar uma monitorização periódica para os detectar e assim efectuar a acção apropriada, como é o caso exemplificativo dos servo-motores. Também existem outros eventos que embora não tenham uma ocorrência tão frequente, possuem intervalos longos de tal forma que entre dois eventos consecutivos pode não existir nenhuma relação ou até o evento pode não voltar-se repetir.

Nestas situações, o uso de mensagens periódicas é um erro que provoca desperdício de processamento e largura de banda, uma vez que grande parte do tempo não há nada a transmitir. Nesse caso o mais apropriado é recorrer ao uso de sensores inteligentes, pela sua capacidade de autonomia de processamento que lhes permite enviar as mensagens apenas quando é necessário, devido a algum evento específico.

Se estas mensagens forem tratadas como puramente aperiódicas, assume-se que podem ser geradas em qualquer instante, podendo até surgirem em sucessões rápidas. Dessa forma não é possível haver um cumprimento das metas temporais. Mas em situações reais exige-se um intervalo mínimo entre eventos aperiódicos consecutivos, designado por ‘Tempo Mínimo Entre Chegadas’ (MIT – “*Minimum Interarrival Time*”). Será este tipo de mensagens que se designam por “Mensagens Esporádicas”. A partir do momento que se sabe o MIT de uma série de mensagens é sempre possível cumprir a sua meta temporal, mesmo na pior situação possível.

2.3 - Mensagens não tempo-real

Num ambiente industrial existe, por exemplo, um operador que faz a monitorização dos equipamentos do sistema, especialmente os que tem necessidade de apresentar dados e informações operacionais do seu funcionamento tais como limites e diagnósticos. Qualquer protocolo de comunicação que seja usado tem que suportar este tipo de mensagens, garantido que as suas metas temporais são cumpridas, embora não sejam possuidoras de requisitos para tempo-real.

2.4 - Mensagens de tempo-real: alta vs baixa velocidade

As mensagens que podem existir num sistema contém diversas metas temporais, as quais podem ir desde alguns microssegundos até a algumas unidades de segundo. Devido a este facto, as mensagens de tempo-real são assim divididas em duas classes: alta velocidade e baixa velocidade; conforme o estado crítico da sua meta temporal.

O termo alta velocidade aqui usado é relativo à meta temporal mais crítica existente na carga (d_0). Desta forma, todas as mensagens que fazem parte da mesma ordem de grandeza de metas temporais (ou dentro da diferença de magnitude estipulada aceitável) que d_0 podem ser consideradas mensagens de alta velocidade. Todas as outras são consideradas de baixa velocidade. [ZS, 95]

3 - Notação e modelo de comunicação

Uma mensagem CAN para transmissão apenas pode ter um de dois formatos: mensagem de dados ou mensagem de pedido remoto de transmissão; tendo um comprimento entre 0 bytes e 8 bytes (relativamente aos dados) e um identificador (descrito anteriormente). O conjunto de todas as mensagens no sistema designa-se por fluxos de mensagens.

Como num sistema típico uma mensagem é enfileirada por uma tarefa da aplicação, foi assumido que cada tarefa é invocada de forma repetida, considerando que a mesma chegou sempre que é efectuada uma acção de invocação, possuindo um período, que é o MIT definido anteriormente e não um intervalo de tempo fixo. Se uma mensagem enfileirada por uma determinada tarefa é potencialmente enviada no instante em que se realiza a invocação, diz-se que a mensagem possui um período igual ao período da tarefa. Desta forma, o período da mensagem m é representado por T_m . O pior tempo de resposta de uma tarefa i , designada por R_i , é definido como o maior intervalo de tempo decorrido entre a chegada da mensagem e o tempo necessário para completar o processamento computacional.

Mas, de um modo geral, o enfileiramento das mensagens ocorre com “*jitter*”, isto é, variação nos tempos de enfileiramento, pelo que para ser efectuada a análise de um modo mais correcto é necessário ter em conta este factor. O enfileirar do jitter é definido como a diferença entre o tempo mais cedo e o tempo mais tarde com que uma mensagem pode ser enfileirada [TB, 94].

Tal como o período, o jitter de uma mensagem m também está relacionado com o início de uma tarefa de transmissão. Para uma qualquer tarefa i , cuja pior resposta temporal seja R_i , a enviar uma mensagem m , a janela de oportunidade para o enfileiramento não é maior do que R_i , isto é, a diferença entre os tempo mais cedo e o

tempo mais tarde para o enfileiramento. O “*jitter*” de uma mensagem m é então designado por J_m . Note-se que num sistema realístico todas as mensagens possuem um qualquer tipo de “*jitter*” no enfileiramento.

O pior caso de resposta temporal de uma mensagem m designa-se por R_m e corresponde ao tempo mais longo que é necessário decorrer para a mensagem chegar aos nós de destino, relativamente ao tempo de chegada da tarefa de envio.

O tempo mais longo que leva transmitir uma mensagem m é designado por C_m . Se a mensagem contiver 8 bytes de dados (a maior mensagem que é permitida numa rede CAN) e for transmitida a 1MBit/s, C será de 130 μ s (64 bits de dados, 47 bits no controlo – identificador, CRC, etc. – e até 19 stuff bits). A Figura 42 demonstra este sincronismo numa mensagem CAN.

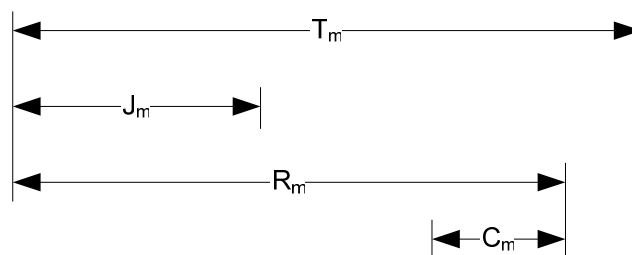


Figura 42 – Modelo das temporizações para mensagens na rede CAN

4 - Teoria de escalonamento básico da rede CAN num único processador

A base que está por detrás do escalonamento natural das mensagens num barramento do tipo CAN (prioridades fixas, não preemptivo) é análoga ao escalonamento de tarefas por prioridades fixas (por exemplo, através do algoritmo Rate Monotonic Não Preemptivo). Desta forma é possível efectuar a análise por mecanismos já existentes, pelo que de seguida faz-se uma breve síntese da teoria de escalonamento de tarefas por prioridades fixas num único processador.

Audsley [AB, 93] e Burns [BN, 93] mostram como a análise de Joseph e Pandya [JP, 86] pode ser melhorada de forma a ser possível incluir factores de bloqueio introduzidos por períodos de não-preempção, libertar o tempo de “*jitter*” e ter o valor exacto em conta para uma tarefa ser não-preemptiva durante um certo intervalo de tempo antes de terminar. As equações seguintes representam esta análise.

$$R_i = J_i + w_i + C_i \quad \text{eq. 1}$$

Onde w_i é dado por:

$$w_i^{n+1} = B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n + J_j + \tau_{res}}{T_j} \right\rceil C_j \quad \text{eq. 2}$$

Onde $hp(i)$ é o conjunto de tarefas com maior prioridade relativamente à tarefa i , C_i é a pior situação de tempo de processamento requerido pela tarefa i e T_j é o período da tarefa j . B_i é o factor de bloqueio da tarefa i (uma ligação temporal que embora seja de uma tarefa menos prioritária impede a execução da tarefa i ; o protocolo de tecto de prioridades [SL, 90] controla esta inversão de prioridades e define como B_i pode ser processada. τ_{res} é a resolução com que foi medido o tempo. Quando se trabalha com barramento CAN, considera-se as unidades temporais como múltiplos do tempo de bit, designados por t_{bit} , pelo que num barramento de 1 Mbit/s temos o corresponde a 1 μ s.

J_i é o jitter que foi libertado pela tarefa i , análogo ao enfileirar do jitter de uma mensagem. É de notar que se uma tarefa é invocada por uma mensagem que está a entrar, esta tarefa irá herdar o jitter (assim como o período) que foi libertado da mensagem, da mesma forma que uma mensagem herda o “jitter” enfileirado de uma tarefa de envio; isto é conhecido como herança de atributos, a qual leva a uma aproximação designada de escalonamento holístico. [TC, 94].

Assim, a escalonabilidade de uma determinada tarefa pode ser determinada de um modo trivial, pela comparação da pior situação de resposta temporal de uma tarefa contra a sua meta temporal. Note-se que a meta temporal de uma determinada tarefa, designada por D_i , é assumida como sendo de valor igual ou inferior a T_i . Também é assumido que uma tarefa por si só não tem capacidade de se interromper voluntariamente, pelo que o processador não entra em modo *idle* enquanto houver trabalho para realizar. A equação eq. 2 faz a descrição de uma relação recorrente, onde a $(n+1)$ -ésima aproximação ao valor de w_i é encontrada em termos da n -ésima aproximação, sendo a primeira aproximação igualada a 0, pelo que desta forma a solução encontra-se quando a $(n+1)$ -ésima aproximação é igual à n -ésima.

Neste momento, já é possível aplicar esta análise introduzida ao escalonamento do barramento CAN. Primeiro será feita uma análise como se o barramento tivesse um comportamento ideal, discutindo-se depois como a análise pode ser afectada pelo comportamento do hardware implementado. Para esta análise com hardware considerou-se o controlador CAN Philips SJA1000. A análise aqui apresentada é idêntica à análise apresentada por Tindell [TB, 94].

5 - Análise do CAN ideal

Fazendo uma analogia entre o escalonamento de mensagens e o escalonamento de tarefas, a análise feita no ponto anterior pode ser aplicada a uma rede CAN ideal: uma tarefa é libertada num determinado instante de tempo, isto é, colocada numa fila ordenada por prioridades e compete com outras tarefas até que se torne na mais prioritária.

A tarefa escalonada tem que competir com todas as outras tarefas de prioridade superior até que estas tenham sido todas transmitidas e o processador esteja livre. Segundo o modelo de Burns *et al* [AB, 93], se uma tarefa já está a ser executada, esta tem que terminar a execução até ao fim, só depois é que se volta à fila a procurar a próxima tarefa a ser executada.

Este princípio é o mesmo para as mensagens numa rede CAN: uma mensagem compete com as outras de maior prioridade, mas quando ganha vez pela sua prioridade

mais elevada começa a transmitir sem interrupções até terminar (escalonamento por prioridades não preemptivo).

A situação de pior resposta temporal da mensagem em fila que pode surgir, medida desde que a mensagem enfileirada chega até ao momento da sua completa transmissão, é obtida da analogia com a equação eq. 1 por:

$$R_m = J_m + w_m + C_m \quad \text{eq. 3}$$

onde J_m é o jitter enfileirado da mensagem m , herdado da pior resposta temporal $R_{sender(m)}$, onde $sender(m)$ é referente à mensagem m enfileirada. O termo w_m representa a pior situação de atraso na fila, isto é, o tempo mais longo entre a colocação da mensagem na fila ordenada por prioridades e começar a sua transmissão. Fazendo a analogia à equação eq. 2, temos que:

$$w_m = B_m + \sum_{\forall j \in hp(m)} \left\lceil \frac{w_m + J_j + \tau_{bit}}{T_j} \right\rceil C_j \quad \text{eq. 4}$$

onde o termo B_m é a pior situação de bloqueio temporal da mensagem m , sendo análogo ao factor de bloqueio da análise do protocolo de prioridade de topo. B_m é igual ao maior tempo que levou a transmitir a mensagem de menor prioridade, sendo dada por:

$$B_m = \max_{\forall k \in lp(m)} (C_k) \quad \text{eq. 5}$$

onde $lp(m)$ será o conjunto de fluxos de mensagens de menor prioridade que a mensagem m no sistema. Mas se m já for a mensagem de mais baixa prioridade, B_m será zero. τ_{bit} é o tempo necessário para transmitir um bit no barramento e $hp(m)$ é o conjunto de fluxos de mensagens com prioridade superior à mensagem m . C_m é o maior período possível para transmitir a mensagem m .

Como em CAN temos 47 bits por mensagem e 5 bits de stuff, mas apenas 34 bits dos 47 podem levar os bits de stuff, C_m pode ser definido como:

$$C_m = \left(\left\lceil \frac{34 + 8s_m}{5} \right\rceil + 47 + 8s_m \right) \tau_{bit} \quad \text{eq. 6}$$

onde s_m é o número de bytes de dados na mensagem. A equação eq. 4 pode então ser desenvolvida da mesma forma que a equação eq. 2.

6 - Comportamento real do SJA1000

Antes de se poder fazer uma análise do comportamento para tempo-real, é necessário ter em conta o comportamento do controlador de rede CAN de tempo-real, mais precisamente o Philips SJA1000, o qual está incorporado nos módulos escolhidos para a experimentação prática da análise.

O controlador CAN da Philips é bastante simples mas completo, possuindo dois buffers de mensagens, um para transmissão e outro para recepção, ambos com 10 bytes de espaço em modo “*BasicCAN*” e com 13 bytes em modo “*PeliCAN*”, sendo que o buffer de recepção pode conter de 4 a 32 mensagens, dependendo do tamanho destas. O modo de funcionamento “*PeliCAN*” permite o uso da especificação CAN 2.0B, com um formato de identificador de mensagem estendido a 29 bits.

Normalmente o microcontrolador acede a este controlador como um periférico mapeado em memória mapeada. Este controlador permite a sinalização por interrupção de mensagem recebida e mensagem enviada para além das interrupções de erro. O microcontrolador aceita os três principais sinais que nos interessam para as simulações: envia mensagem; aborta mensagem e remove mensagem.

Para a recepção de mensagens é necessário que o microcontrolador as retire o mais depressa possível do buffer do controlador e as guarde o quanto antes na sua própria memória.

Para enviar uma mensagem, o microcontrolador coloca os dados no buffer de transmissão do controlador e envia um pedido de transmissão para o controlador, passando a este o controlo da mensagem. Consideremos o tempo mais longo para esta operação como sendo τ_{copy} . Actuando de acordo com o protocolo CAN, o controlador tenta efectuar a transmissão, assinalando com uma interrupção assim que esta for efectuada. Da mesma forma, quando é recebida uma mensagem sem erros, é activada a interrupção de recepção e pedido ao microcontrolador que copie os dados do buffer de recepção do controlador para a sua memória para posterior processamento.

Existe um comando de abortar mensagem, essencial na escrita do software quando se pretende enfileiramento por prioridades (apenas para fila em software, na rede o enfileiramento é do tipo FIFO), caso contrário a performance de trabalho de tempo-real seria quase inexistente: considere-se a seguinte situação em que existe uma mensagem no buffer de transmissão do controlador mas o microcontrolador acabou de colocar uma mensagem com prioridade superior na fila de espera. Se não for possível abortar a mensagem que está no buffer do controlador, a mensagem de maior prioridade fica presa na fila até que a de menor prioridade seja transmitida. Mas esta só conseguirá iniciar a sua transmissão quando as outras mensagens de maior prioridade da rede tiverem sido todas completamente transmitidas, o que pode levar demasiado tempo. Este problema é conhecido como “**Inversão de Prioridades**”.

A solução aplicada para esta questão é haver a possibilidade do microcontrolador, mandar abortar a mensagem que está no buffer e colocar lá a mensagem com prioridade superior. Mas isto não funcionará se a mensagem que está no buffer já tiver começado a sua transmissão para rede, caso em que haverá um pequeno atraso igual, no máximo, ao tempo de transmissão da mais longa mensagem antes que o barramento seja libertado.

Outro problema que ainda persiste é a questão do tempo entre mensagem enviada e o microcontrolador copiar a próxima mensagem para o buffer não ser nulo, embora seja extremamente curto. Durante este tempo, outro nó com uma mensagem de

menor prioridade pode reclamar o barramento e assim indeferir a transmissão da nova mensagem copiada, embora esta seja de prioridade superior (outro exemplo de inversão de prioridades). Esta situação também pode surgir no caso em que a mensagem é removida do buffer, já que o curto espaço de tempo que demora a abortar a mensagem e copiar a nova mensagem (mais prioritária) para o buffer liberta o barramento para o tráfego de baixa prioridade.

Este último caso (preempção) deixa na verdade a possibilidade em aberto do barramento ser reclamado duas vezes pelo tráfego de menor prioridade: quando existe remoção da fila por uma mensagem de alta prioridade e quando uma mensagem de baixa prioridade foi transmitida.

Considere-se um caso mais ilustrativo em que temos uma mensagem M para ser enviada a partir de determinado nó, uma mensagem H de alta prioridade e uma mensagem de baixa prioridade $L1$ no mesmo nó. De outros nós apenas temos as mensagens de baixa prioridade $L2$, $L3$ e $L4$. Com este cenário, a transmissão da mensagem M pode ser atrasada até 4 vezes, embora uma seja pelo mecanismo de gestão do buffer do controlador, que impõe uma ordem FIFO na transmissão das mensagens.

O primeiro atraso ocorre quando a mensagem M é colocada na fila, porque tal como já vimos antes, o controlador SJA1000 não tem a capacidade de abortar uma mensagem se esta já tiver começado a sua transmissão. Logo que a mensagem seja enviada, o microcontrolador copia a mensagem M para o buffer de transmissão, o que leva um tempo máximo de τ_{copy} . Durante este período de tempo, o barramento tanto pode ficar em pausa como ser reclamado por outro nó, mesmo com mensagens de prioridades mais baixas ($L1$).

Depois que a mensagem M terminou a sua cópia para o controlador e está pronta para transmissão, pode ser atrasada por uma mensagem de menor prioridade ($L2$) que acabou de iniciar a sua transmissão a partir de outro nó. Imediatamente antes de M ser transmitida, surge a mensagem H que vai preceder M , pelo que o controlador aborta a mensagem M e coloca no seu lugar a mensagem H . Durante este espaço de tempo, mais uma vez o barramento ficou livre e houve uma mensagem de menor prioridade ($L3$) que começou a ser transmitida, atrasando deste modo ambas as mensagens H e M .

Assim que a mensagem H é transmitida, o microcontrolador volta a colocar a mensagem M no buffer de transmissão, o que volta a originar a libertação do barramento e a sua ocupação por uma mensagem de menor prioridade ($L4$).

Desta forma podemos obter uma relação directa dos atrasos com a inversão de prioridades e com o copiar das mensagens para o buffer de transmissão do controlador. A inversão de prioridades pode ser de tal forma elevada que proporciona a pior situação de performance do controlador. A tabela seguinte, em conjunto com o gráfico, demonstra este cenário, tendo como base o modelo de escalonamento ‘Rate Monotonic’ em que as metas temporais são iguais aos períodos.

<i>Mensagem</i>	<i>T</i>	<i>D</i>	<i>C</i>	<i>U</i>
<i>H</i>	605	605	47	7,8%
<i>M</i>	610	610	47	7,8%
<i>L1</i>	100000	100000	130	0,13%
<i>L2</i>	100000	100000	130	0,13%
<i>L3</i>	100000	100000	130	0,13%
<i>L4</i>	100000	100000	130	0,13%

Tabela 8 – Resultados obtidos do cenário-exemplo de Tindell [TH, 94]

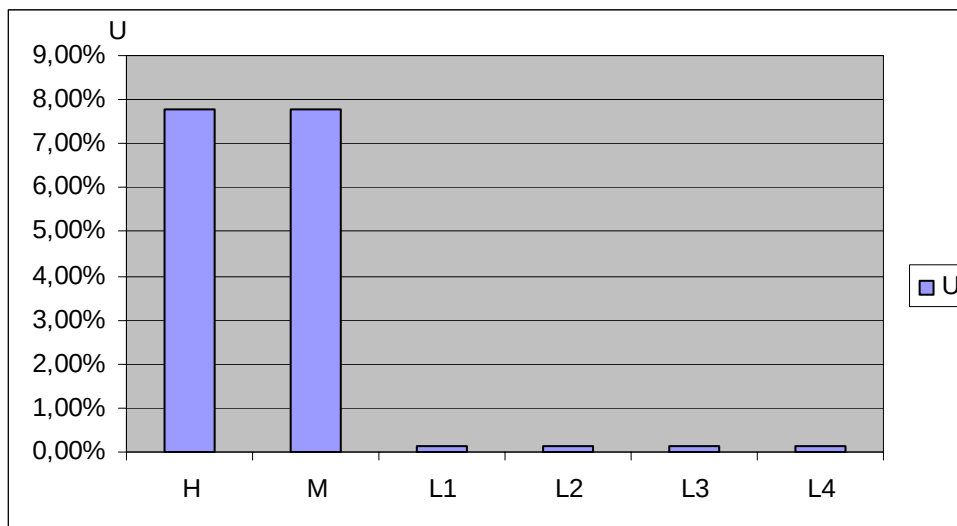


Figura 43 – Gráfico relativo à utilização do cenário-exemplo de Tindell [TH, 94]

É de realçar as semelhanças deste cenário com o caso do escalonamento 2 realizado nas simulações, das quais se faz a análise detalhada no Capítulo 6.

Na Tabela 8, todos os valores temporais são em μs , é assumido que não existe jitter e o tempo de cópia de uma mensagem τ_{copy} é muito curto, no entanto suficientemente longo para permitir o acesso de mensagens de menor prioridade ao barramento enquanto coloca uma mensagem no buffer de transmissão, mas não tão longo que forme uma parte significativa do tempo de resposta de uma mensagem, contudo este valor é inatingível.

Este exemplo de mensagem que se demonstrou se verificarmos o cenário, vemos que o tempo de resposta da mensagem *M* é de cerca de 614 μs , a que corresponderá uma utilização do barramento ligeiramente abaixo de 16%.

7 - Síntese

Antes de se passar ao desenvolvimento e à experimentação dos escalonamentos, efectuou-se um estudo prévio do protocolo CAN convencional. De facto existem três tipos de fluxos de mensagens a considerar em constante circulação numa rede CAN: fluxo de mensagens periódicas (com metas temporais apertadas/rígidas), fluxo de mensagens esporádicas (com metas temporais apertadas/rígidas) e fluxo de mensagens aperiódicas não de tempo-real. Por isso é necessário que tanto o controlador de rede

CAN, como o microcontrolador anfitrião seja capaz de lidar com tal tráfego. Contudo, o verdadeiro problema surge na situação em que apenas se possui um microcontrolador e o controlador de rede (neste caso o SJA1000), não possui vários buffers internos para guardar as mensagens a receber ou a enviar, pelo que se torna necessário desenvolver um buffer por software para o microcontrolador, tendo como princípio uma fila circular.

Aliás, este problema da fila circular acentua o problema da inversão de prioridades, como foi possível observar pelo exemplo de Tindell [TH, 94] e como vai ser possível verificar durante a análise experimental dos escalonamentos, nomeadamente na experimentação do escalonamento com CAN convencional.

Capítulo 5

Algoritmos de escalonamento utilizados na análise

Capítulo 5 - Algoritmos de escalonamento utilizados na análise

1 - Introdução

Tal como acontece em qualquer protocolo de comunicação que pretenda suportar comunicação de tempo-real, foi necessário escolher alguns algoritmos de escalonamento a aplicar no envio de fluxos de mensagens para a rede. Neste capítulo serão descritos três desses algoritmos, nomeadamente o Deadline Monotonic [ZS, 95], o Earliest Deadline First [ZS, 95] e o Mixed Traffic Scheduling [ZS, 95]. A razão que levou à escolha destes três algoritmos, para além de serem os mais usados, é o facto de também serem de fácil adaptação para o software de envio para a rede CAN.

Os pontos base desta adaptação foram baseados em diversos estudos de escalonamentos não preemptivos para aplicações de controlo de tempo-real [ZS, 95].

2 - Escalonamentos analisados

Tal como já foi demonstrado anteriormente, os acessos ao barramento CAN são controlados pelos identificadores dos fluxos de mensagens em competição. Então, é necessário colocar a questão “Como atribuir os identificadores aos fluxos de mensagens de forma a se obter o máximo rendimento do escalonamento?” [ZS, 95].

Antes de se observar as dificuldades de efectuar escalonamento de mensagens em CAN, vamos resumir a constituição de um chip de interface para barramento CAN. Normalmente estes chips têm um espaço da memória reservado para uma ou mais mensagens, de forma que quando o processador termina de calcular o identificador para a mensagem, coloca-a nesse espaço. Nesse momento, o chip passa a trabalhar de forma autónoma, iniciando a competição pelo barramento para iniciar a transmissão. Por vezes, existe uma opção de notificar o processador de que a mensagem foi transmitida.

Assim que a mensagem tenha sido transferida na sua totalidade para o chip CAN, o seu identificador manter-se-á fixo, a menos que o processador cancele a mensagem e altere o identificador. Se o identificador for derivado da prioridade da mensagem, esta prioridade tem que se manter fixa, pelo menos por um período de tempo razoavelmente longo. Assim, o escalonamento por prioridades fixas (por exemplo, Rate Monotonic [ZS, 95]) é a forma natural dos escalonamentos nos chips CAN, pelo que cada mensagem terá que ter uma prioridade única que fará parte do seu identificador, não permitindo apenas a identificação específica da mensagem para fins de recepção mas também o seu escalonamento de uma forma previsível.

Contudo, o escalonamento por prioridades fixas tem menor utilização (ou rendimento) do que outros escalonamentos como o não-preemptivo Earliest Deadline First (EDF) [ZS, 95]. Isto leva a que se considere uma possibilidade o uso do escalonamento EDF no CAN. Mas como este escalonamento tem prioridades dinâmicas e o barramento CAN unicamente tem prioridades fixas, não é prático estar continuamente a actualizar as mensagens para reflectir as novas prioridades, pelo que torna-se impraticável o seu uso em CAN.

2.1 - Escalonamento por prioridades fixas – baixa utilização

Tal como já foi afirmado, o escalonamento por prioridades fixas é a solução aplicada de um modo natural nos chips CAN actuais. A forma mais corrente deste tipo de escalonamento para tempo-real é o Rate Monotonic (RM) [ZS, 95], em que os fluxos de mensagens com o período mais curto tem prioridade sobre os fluxos de mensagens que tem o período mais longo, isto é, o RM assume que a prioridade é igual ao inverso do período, o que nem sempre corresponde à verdade. Desta forma, usou-se um escalonamento relativamente próximo do RM, o Deadline Monotonic (DM) [ZS, 95]. Neste escalonamento, aos fluxos de mensagens com metas temporais mais apertadas são atribuídas prioridades mais elevadas, as quais farão parte do identificador de cada mensagem. Uma vez iniciada a transmissão, esta é completada mesmo que surjam mensagens com prioridade mais elevada, isto é, usa-se a versão não preemptiva do algoritmo DM [ZS, 95].

Já foi visto no Capítulo 4, ponto 2, que a rede tem que suportar tanto os fluxos de mensagens periódicas críticas como os fluxos de mensagens esporádicas, para além de tráfego não tempo-real. O algoritmo DM consegue este objectivo ao efectuar o tratamento das mensagens esporádicas como se tratassem de mensagens periódicas com período igual ao MIT, atribuindo depois as prioridades às mensagens de tempo-real conforme as metas temporais pretendidas/necessárias, quebrando os laços da arbitrariedade, deixando assim os restantes níveis de prioridade para mensagens não tempo-real. Desta forma é assegurado o tráfego de tempo-real existente em relação ao tráfego não tempo-real.

Assim, podem-se usar condições de escalonamento bem conhecidas para verificar se todas as mensagens de tempo-real são escalonáveis ou não. O algoritmo DM é um esquema simples e de fácil implementação em CAN. Contudo, de forma a aumentar a utilização de escalonamento tem-se como hipótese o uso do algoritmo EDF para o escalonamento de mensagens em CAN [ZS, 95].

2.2 - Escalonamento por prioridades dinâmicas – Earliest Deadline First e os problemas de codificação das metas temporais

O principal modo de funcionamento do EDF é dar a prioridade mais elevada às mensagens que terão o seu início de transmissão mais próximo, no momento do escalonamento, de modo que o objectivo pretendido será reflectir as metas temporais nos identificadores das mensagens. Mas isto não significa que as mensagens deixem de ter um identificador único, antes pelo contrário, uma vez que este é um requisito obrigatório em CAN. O modo como se pode fazer isto é dividir o identificador de mensagens em três campos diferentes, como pode ser visto na Figura 44.

Prioridade 1 Bit	Meta Temporal 5 Bits	Unicidade 5 Bits
---------------------	-------------------------	---------------------

Figura 44 – Estrutura do identificador para EDF

O campo ‘meta temporal’ é derivado da própria meta temporal, mais precisamente do seu complemento lógico (é necessária a meta temporal ser a mais crítica para ter a prioridade mais alta). Na eventualidade de surgirem duas mensagens

com a mesma meta temporal, existe o campo da unicidade que fará a distinção. Esta unicidade é um código único para cada fluxo de mensagens, o qual também poderá servir para fazer a distinção dos fluxos de mensagens na recepção para um posterior processamento. No caso do escalonamento EDF, estes códigos únicos podem ser atribuídos de uma forma arbitrária, desde que para cada fluxo sejam únicos. Contudo, esta atribuição pode ser algo complexa e revelar-se mesmo crítica no algoritmo MTS, como será visto mais tarde nas experiências realizadas.

Na Figura 44 pode-se observar no início um campo que terá somente 1 bit. Será este campo que irá definir a prioridade da mensagem: ‘1’ para fluxos de mensagens de tempo-real e ‘0’ para outros casos. Desta forma é assegurada a distinção entre as prioridades.

À medida da passagem do tempo, os valores das metas temporais vão aumentar, havendo assim a necessidade de cada vez mais bits para o identificador da mensagem CAN. Uma maneira de se ultrapassar este problema será não considerar o tempo próprio da meta temporal mas sim o tempo que falta para que essa mesma meta temporal seja atingida. Mas esta solução vem introduzir dois novos problemas:

- Uma vez que o tempo que falta para o cumprimento da meta temporal é alterado a cada ciclo do programa, é necessário que os identificadores dos fluxos de mensagens sejam continuamente actualizados, o que leva a um excesso de carga no processador.
- Como já foi afirmado anteriormente, uma carga típica de comunicação contém diversas metas temporais. Isto origina uma codificação de uma grande variedade de valores para a qual pode não haver bits suficientes no identificador CAN.

2.2.1 - Épocas temporais

A solução por épocas temporais pode ser aplicada tanto no algoritmo EDF como no algoritmo MTS. Uma solução possível para resolver o primeiro problema exposto é alterar o barramento dos chips de interface para integrarem contadores programáveis nas posições adequadas dos identificadores. Desta forma, o cálculo do tempo que faltava para o cumprimento das metas temporais é efectuado automaticamente em cada ciclo. No entanto, este tipo de chips não se encontra disponível comercialmente. Assim, a melhor solução encontrada passa por um compromisso custo/performance, aplicando-se uma solução por software [ZS, 95].

Neste caso, é ao processador que cabe a tarefa de efectuar as actualizações, devendo fazê-lo com uma frequência reduzida, de forma a gastar o menor tempo possível do processador na actualização. Desta forma a solução passa por usar os valores actuais das metas temporais expressas relativamente a uma referência incrementada periodicamente, designada de ‘Início de Época’ (SOE – “*Start of Epoch*”). O tempo decorrido entre dois SOE consecutivos designa-se por ‘Duração da Época’ [ZS, 95].

Desta forma, o campo da meta temporal para a mensagem i será o inverso lógico de:

$$d_i - SOE = d_i - \left\lfloor \frac{t}{l} \right\rfloor l \quad \text{eq. 7}$$

onde d_i é a meta temporal da mensagem i e t é o tempo actual (assume-se que todos os relógios estão sincronizados entre si). O valor de l depende da fracção temporal do processador que foi dispensada para as actualizações, designada por x . Seja M o desempenho (em MIPS) do processador e n o número de instruções necessárias para cada actualização. Como cada actualização deve estar distanciada l segundos da seguinte, então:

$$l = \frac{n}{xM \times 10^6} \quad \text{eq. 8}$$

Desta forma irá existir um processo periódico em cada nó que a cada l segundos actualiza os identificadores dos fluxos de mensagens prontas, de acordo com a equação anterior. Fazendo a comparação com a aproximação tempo para a meta temporal, observa-se que se l for muito maior que a duração média de uma mensagem, é visível que a aproximação inicial usa menos tempo de processador, o processo de actualização é periódico e previsível, enquanto que na última aproximação o processador é interrompido aperiodicamente no início de cada ronda de arbitragem. No entanto esta aproximação tem como custo a maior quantidade de bits necessários no identificador. Seja D o maior valor de $(d - r)$ de uma qualquer mensagem, em que r é o tempo de saída. Então, para a aproximação “tempo para a meta temporal” o comprimento do campo da meta temporal é $m = \log_2 D$, mas pelo último método é $m = \log_2 (l + D)$. Este cálculo é de forma a acomodar o pior caso que possa surgir, o qual está representado na Figura 45.

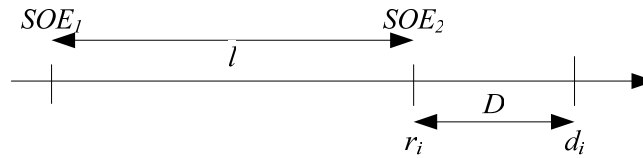


Figura 45 – Maior valor possível para o campo da meta temporal

Quando uma mensagem i que possui o maior valor $(d - r)$ é iniciada imediatamente antes de um novo SOE, então o campo da meta temporal dessa mesma mensagem i terá o valor

$$d_i - SOE_1 = l + D \quad \text{eq. 9}$$

Comprimento do campo do identificador

A razão pela qual não é prático aplicar o escalonamento EDF ao barramento CAN é a necessidade de demasiados bits para o campo da meta temporal no identificador, visto existir a necessidade de codificar a meta temporal no identificador. Por exemplo, numa carga típica, existem fluxos de mensagens associados a controladores de alta velocidade, cujas metas temporais podem ser na ordem dos microssegundos, mas existem outros fluxos de mensagens relacionados com sensores de

temperatura, por exemplo, cujas metas temporais são da ordem de vários segundos. Se as metas temporais forem representadas com uma resolução de μs são necessários mais de 20 bits para representar as metas temporais na ordem dos segundos.

De forma a poder acomodar estes valores, pode-se usar o identificador de 29 bits do formato de mensagem CAN estendido, contudo também aumenta o tamanho total da mensagem em 20 bits em comparação com o formato standard do CAN, o que implica um aumento de cerca de 20% a 30% da largura de banda desperdiçada.

Desta forma surge um novo dilema: o algoritmo DM é fácil de implementar mas possui fraca utilização; por outro lado o algoritmo EDF aumenta o tamanho das mensagens a tal ponto que não é obtido ganho na rede. Assim a solução passa por criar uma solução de compromisso entre os dois escalonamentos, surgindo assim o ‘Escalaonamento por Tráfego Misto’ (MTS – Mix Traffic Scheduler) [ZS, 95].

2.3 - MTS – Escalonamento por Tráfego Misto

Este escalonamento tenta aplicar o melhor dos algoritmos de escalonamento EDF e DM, respectivamente a alta utilização e o identificador de 11 bits [ZS, 95].

No DM um identificador de 11 bits permite a representação de até 2048 fluxos de mensagens diferentes, mas num sistema realista não é necessário o uso de tantos fluxos de mensagens, pelo que alguns bits do identificador não são usados, podendo ser aplicados para melhorar o escalonamento. Como os fluxos de mensagens que consomem maior largura de banda são os de alta velocidade, a melhoria no escalonamento pode ser significativa se incidir apenas sobre este tipo de fluxos de mensagens.

A ideia subjacente a este escalonamento é o uso do princípio do EDF para fluxos de mensagens de alta velocidade enquanto é aplicado o DM para fluxos de mensagens de baixa velocidade. A primeira coisa a fazer é atribuir as prioridades de forma a dar maior prioridade aos fluxos de mensagens de alta velocidade em relação aos fluxos de mensagens de baixa velocidade e não tempo-real, tal como é representado na Figura 46.

1	Meta Temporal	Prioridade DM
---	---------------	---------------

Figura 46 – Estrutura do identificador para MTS de fluxos de mensagens de alta velocidade

Esta atribuição é necessária para protecção dos próprios fluxos de mensagens de alta velocidade em relação ao restante tráfego. Se o campo da unicidade for constituído, por exemplo, por 5 bits (o que permite 32 mensagens de alta velocidade) e o campo de prioridade por 1 bit, os restantes 5 bits ainda não são suficientes para codificar as metas temporais, relativamente ao último SOE. Uma solução possível será efectuar a quantificação do tempo em regiões e codificar as metas temporais de acordo com as regiões em que se encontrarem. No caso de haver duas mensagens que coincidam na mesma região, é usada a prioridade DM para fazer a sua distinção, como se do código de unicidade se tratasse.

No nível mais alto está o algoritmo EDF: se as metas temporais de duas mensagens podem ser distinguidos após a quantificação, então a meta temporal mais

próxima terá maior prioridade. No nível mais baixo está o algoritmo DM: se os fluxos de mensagens tiverem as respectivas metas temporais na mesma região, estes serão escalonados segundo sua prioridade DM [ZS, 95].

O tamanho de uma região pode ser calculado como sendo:

$$l_r = \frac{l + D_{\max}}{2^m} \quad \text{eq. 10}$$

Onde $D_{\max}$ é a meta temporal relativa mais longa de qualquer mensagem de alta velocidade e m é o tamanho do campo da meta temporal (5 bits neste exemplo). Isto fica mais perceptível se observar a Figura 47, onde $m = 3$.

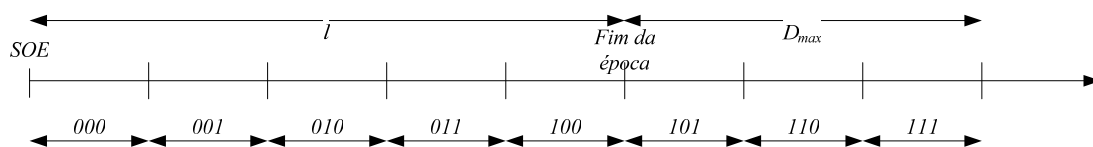


Figura 47 – Quantificação das metas temporais relativamente ao início de época

A situação do pior caso ocorre quando uma mensagem com meta temporal $D_{\max}$ é enviada mesmo antes do fim da época, pelo que a sua meta temporal absoluta recai $l + D_{\max}$ para lá do SOE actual. O campo da meta temporal terá que também codificar esta diferença temporal usando m bits, o que origina a expressão anterior de l_r [ZS, 2000].

Outra opção é usar logaritmos para quantificar as metas temporais, obtendo-se dessa forma grande resolução para metas temporais próximas e menor resolução para metas temporais mais afastadas. No entanto este esquema só é viável se os campos das metas temporais de todos os fluxos de mensagens são actualizados no SOE e mantêm-se fixos até ao próximo SOE. Desta forma, o que no início da época estava longe, eventualmente ficará próximo com o avançar do tempo, contudo mantendo-se a codificação da meta temporal basicamente na mesma, apesar de haver uma degradação do MTS para o DM.

Assim será usado escalonamento DM para fluxos de mensagens de baixa velocidade e escalonamento por prioridades para os fluxos de mensagens não tempo-real, as quais serão atribuídas mais tarde de modo arbitrário. Os identificadores para estes fluxos de mensagens são mostrados na Figura 48.

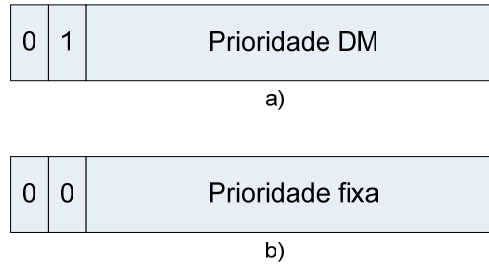


Figura 48 – Estrutura dos identificadores para MTS de fluxos de mensagens a) baixa velocidade e b) não tempo-real

O segundo bit mais significativo dá aos fluxos de mensagens de baixa velocidade maior prioridade sobre os fluxos de mensagens não tempo-real. Este esquema permite até 32 fluxos de mensagens diferentes de alta velocidade (periódicos ou esporádicos), 512 de baixa velocidade (periódicos ou esporádicos) e 480 fluxos de mensagens não tempo-real, o que deve ser suficiente para a maioria das aplicações.

2.4 - Condições de escalonamento

Para a aplicação do algoritmo MTS são necessárias algumas condições de escalonamento “*offline*”, as quais quando cumpridas garantem que todos os fluxos de mensagens cumprem as metas temporais. Antes disso serão revistas essas condições para a versão não preemptiva dos escalonamentos EDF e DM, as quais serão então evoluídas para o escalonamento MTS.

Earliest Deadline First

As condições de escalonabilidade para o algoritmo EDF não preemptivo são descritas nos trabalhos de Zheng e Shin [ZS, 94]. Para a pior situação possível, os fluxos de mensagens esporádicos são tratados como se tratassem de fluxos de mensagens periódicos, em que o período é igual ao MIT. Desta forma, se todas as mensagens forem enviadas ao mesmo tempo, quando $t = 0$ em que se cria a pior situação de congestionamento, estas ainda podem ser escalonáveis se cumprirem as seguintes condições [ZS, 95]:

$$1. \sum_{j=1}^n \frac{C_j}{T_j} \leq 1 \quad \text{eq. 11}$$

$$2. \forall_{t \in S}, \sum_{i=1}^n \left\lceil \frac{t - d_i}{T_i} \right\rceil^+ C_i + C_p \leq t, \quad \text{eq. 12}$$

onde:

$$S = \bigcup_{i=1}^n S_i; \quad \text{eq. 13}$$

$$S_i = \left\{ d_i + nT_i : n = 0, 1, \dots, \left\lfloor \frac{t_{\max} - d_i}{T_i} \right\rfloor \right\}; \quad \text{eq. 14}$$

$$t_{\max} = \max \left\{ d_1, \dots, d_n, C_p + \frac{\sum_{i=1}^n \left(1 - \frac{d_i}{T_i} \right) C_i}{1 - \sum_{i=1}^n \frac{C_i}{T_i}} \right\}. \quad \text{eq. 15}$$

T_i , C_i e d_i representam respectivamente o período, a duração de transmissão e a meta temporal da mensagem i , C_p é o pacote mais longo possível e $\lceil x \rceil^+ = n$ se $n - 1 \leq x < n$, $n = 1, 2, 3, \dots$; e $\lceil x \rceil^+ = 0$ se $x < 0$.

A primeira condição assegura que a utilização máxima não ultrapassa a capacidade e a segunda assegura que qualquer mensagem cuja meta temporal seja inferior ou igual a t pode terminar até t .

Se estas condições forem estendidas de forma a se aplicarem ao caso das mensagens não serem todas lançadas ao mesmo tempo, o pior caso que pode surgir é o de uma mensagem ser lançada no instante $t = 0$ e a mensagem seguinte ser lançada no instante temporal igual ao desfasamento relativo à anterior. Nesse caso, é possível modelar cada par de mensagens, considerando que a primeira mensagem tem $\varphi_1 = 0$ e a segunda tem φ_2 igual ao desfasamento relativamente à anterior. Assim, estas condições passam a ser [ZS, 95]:

$$1. \sum_{j=1}^n \frac{C_j}{T_j} \leq 1 \quad \text{eq. 17}$$

$$2. \forall_{t \in S}, \sum_{i=1}^n \left\lceil \frac{t - d_i - \varphi_i}{T_i} \right\rceil^+ C_i + C_p \leq t, \quad \text{eq. 18}$$

onde:

$$S = \bigcup_{i=1}^n S_i; \quad \text{eq. 19}$$

$$S_i = \left\{ d_i + nT_i : n = 0, 1, \dots, \left\lfloor \frac{t_{\max} - d_i - \varphi_i}{T_i} \right\rfloor \right\}; \quad \text{eq. 20}$$

$$t_{\max} = \max \left\{ d_1, \dots, d_n, C_p + \frac{\sum_{i=1}^n \left(1 - \frac{d_i}{T_i} \right) C_i}{1 - \sum_{i=1}^n \frac{C_i}{T_i}} \right\}. \quad \text{Eq. 21}$$

Deadline Monotonic

Para a situação não preemptiva, é possível efectuar o escalonamento de uma mensagem i , se todas as mensagens prioritárias forem escalonáveis e a mensagem i encontrar uma oportunidade para ser transmitida em qualquer altura entre $[0, d_i - C_i]$. Desta forma, tudo o que é necessário fazer é analisar as mensagens com prioridade mais elevada que a mensagem i e assumir que são as únicas mensagens no sistema [ZS, 95].

Se o barramento alguma vez ficar disponível durante $[0, d_i - C_i]$, então i é escalonável. Se as mensagens são enumeradas de acordo com a sua prioridade, sendo $j = 0$ a mensagem de mais alta prioridade, então i é escalonável se:

$$\exists_{t \in S}, \sum_{j=1}^{i-1} \left\lceil \frac{t - \varphi_j}{T_j} \right\rceil C_j + C_p \leq t \quad \text{eq. 22}$$

onde:

$S = \{\text{conjunto das metas temporais de saída das mensagens } 0, 1, \dots, i - 1 \text{ até ao tempo } d_i - C_i\} \cup \{d_i - C_i\}$

e φ_j são os desfasamentos relativos.

Para verificar o escalonamento de um conjunto de fluxos de mensagens, tem que se repetir este processo para todos os fluxos de mensagens.

Escalonamento por Tráfego Misto

Fazendo-se a distinção entre mensagens de alta velocidade e de baixa velocidade e começando pela primeira, a pior situação que pode surgir é quando ocorrem as seguinte condições:

- 1 – O pior congestionamento de tráfego possível
- 2 – A pior codificação da meta temporal possível

A primeira situação surge quando são lançadas todas as mensagens no mesmo instante t_0 (as mensagens que tiverem $\varphi_i \neq 0$ são enviadas mais tarde). Para o segundo caso, considere-se a Figura 49:

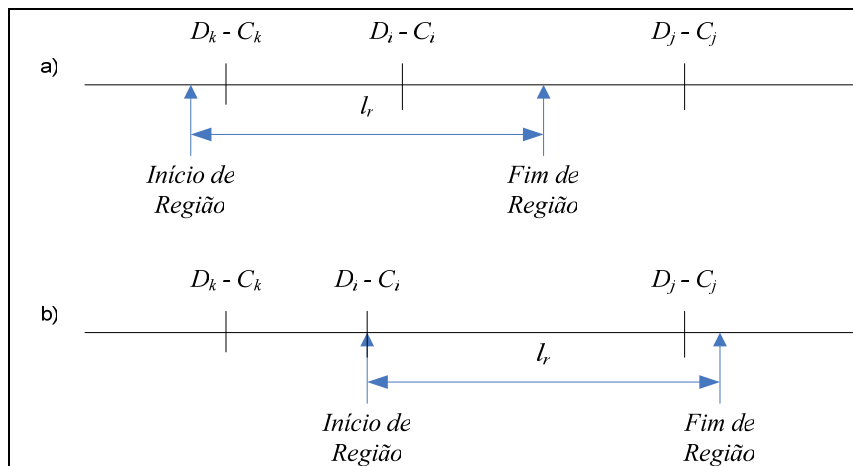


Figura 49 – Exemplo da pior codificação de meta temporal possível

Suponha-se que j tem maior prioridade DM do que i mas não k . Então, na Figura 49 a) i tem a mais alta prioridade e em b) tem a mais baixa, sendo esta a pior situação.

Suponha que se pretende determinar a pior situação de escalonamento para a mensagem i . O objectivo é maximizar o número de mensagens que obtêm prioridade acima de i . Se a meta temporal para o início de i cair sobre o início da região, então todas as mensagens com a meta temporal para o início mais cedo irão cair nas regiões mais cedo e obter a prioridade mais alta [ZS, 95].

Além disso, permite que um maior número de mensagens que possuam uma meta temporal para o início maior do que a de i caia na mesma região que i . Se tiverem prioridade DM mais alta, também saem antes de i .

Efectuando agora um paralelo entre as condições de escalonabilidade do algoritmo MTS e do algoritmo DM, para determinar a escalonabilidade de um fluxo de mensagens considera-se a primeira invocação, isto é, todos os j com prioridade maior que i na situação do pior caso (para o algoritmo MTS isto significa que a meta temporal para o início coincide com o início da região). Então, a partir do que já foi afirmado anteriormente, invocações da mensagem j terão prioridade sobre i se:

1. $(d_i - C_i) > (d_j - C_j)$ ou
2. a) $(d_i - C_i) < (d_j - C_j) \leq (d_i - C_i + l_r)$, e
- b) prioridade DM de j maior que a de i , e
- c) j é libertado antes de $d_i - C_i$

Note-se que j representa invocações individuais e não fluxos de mensagens inteiras. Se considerar apenas as invocações de j que satisfaçam estas condições e escaloná-las de acordo com o algoritmo MTS e o barramento não paralisar durante o intervalo $[t_0, t_0 + (d_i - C_i)]$, então a mensagem i terá uma hipótese de ser enviada e é escalonável. Esta verificação tem que ser feita para cada mensagem de alta velocidade.

Formalmente, uma mensagem é escalonável se a sua primeira invocação i satisfizer a condição:

$$\exists_{t \in S}, \sum_{\forall_j} \left\lceil \frac{t - (t_0 + \varphi_j)}{T_j} \right\rceil C_j + C_p \leq t \quad \text{eq. 23}$$

onde:

- cada j satisfaz as condições anteriores
- $S = \{\text{conjunto de tempos de libertação de cada } j\} \cup \{d_i - C_i\}$
- C_p é o tamanho do pacote mais longo possível
- φ_j é o deslocamento de fase relativo

Para efectuar a verificação da escalonabilidade de mensagens de baixa velocidade apenas é preciso verificar a escalonabilidade DM para cada uma das mensagens de baixa velocidade. Como as mensagens de alta velocidade tem metas temporais mais curtas que as de baixa velocidade, estas automaticamente tem a prioridade DM mais elevada [ZS, 95].

3 - MTS em CAN

Na implementação do algoritmo MTS em CAN, o objectivo principal é minimizar a sobrecarga suportada pelo nó anfitrião na transmissão de uma mensagem. Esta tem os seguintes componentes:

1. Colocar/armazenar mensagens em software se não existirem buffers nos adaptadores de rede (chips CAN).
2. Transferir mensagens para o adaptador de rede.
3. Lidar com interrupções relacionadas com transmissão de mensagens.

Em CAN, a inversão de prioridades pode não ser limitada. Se os buffers do adaptador contiverem mensagens de baixa prioridade, estas mensagens não serão enviadas enquanto existirem mensagens de prioridade intermédia algures na rede. Consequentemente, uma mensagem de alta prioridade pode ficar bloqueada por tempo indeterminado, causando a perda da sua meta temporal. Devido a este problema de inversão de prioridades, qualquer implementação de escalonamentos em CAN (sem definir qual o algoritmo que vai ser implementado – algoritmo DM ou algoritmo MTS) tem que assegurar que os buffers contém sempre as mensagens de maior prioridade e apenas as mensagens de baixa prioridade são colocadas na fila do software [ZS, 2000].

Suponha-se que são reservados B buffers para transmissão de mensagens. Se o total de fluxos de mensagens a sair é igual a B ou inferior, então a implementação do algoritmo MTS é directa, atribuindo-se um buffer a cada fluxo. Sempre que o chip CAN recebe uma mensagem para transmitir, simplesmente copia essa mensagem para o buffer reservado para esse fluxo. Neste caso não existe a necessidade do adaptador CAN gerar interrupções indicando o fim da transmissão, o que origina a sobrecarga mais baixa possível no processador.

Se o número de fluxos de mensagens exceder o valor de B , algumas mensagens terão que ser armazenadas em software. De forma a reduzir a sobrecarga ao processador, deve-se armazenar o menor número possível de mensagens enquanto se evita a inversão de prioridades. Como o algoritmo MTS trata de modo diferente mensagens de alta velocidade e de baixa velocidade para fins de escalonamento, para fins de implementação também serão diferenciadas. O objectivo será manter a sobrecarga para mensagens frequentes (as que pertencem a fluxos periódicos de alta velocidade) o mais baixa possível, de forma a manter uma sobrecarga média baixa por mensagem [ZS, 2000].

Nesta implementação se o número de fluxos periódicos de mensagens de alta velocidade N_{Hp} for menor que B , então são reservados N_{Hp} buffers para fluxos de alta velocidade, tratando-os da mesma forma que anteriormente (sem armazenamento em software). Os restantes $L = B - N_{Hp}$ buffers são usados para mensagens de alta velocidade esporádicas, de baixa velocidade e não tempo-real. Assim que estas mensagens chegam ao chip são inseridas numa fila ordenada por prioridades. Para evitar a inversão de prioridade, o chip tem que se assegurar que os L buffers contêm sempre as L mensagens da cabeça da fila. Assim, se chegar uma mensagem de maior prioridade do que a de menor prioridade que estiver no buffer, a mensagem é retirada e colocada na fila, escrevendo a nova mensagem no buffer. Isto aumenta a sobrecarga no processador mas é necessário para evitar a inversão de prioridade. A mensagem na fila será transmitida de acordo com a sua prioridade.

Entre estes buffers, o buffer que contiver a $I+1$ -ésima mensagem de menor prioridade está configurado para activar uma interrupção na transmissão da mensagem (I será definido mais tarde). Esta interrupção é usada para recarregar os buffers com as mensagens da fila. I tem que ter um valor suficientemente elevado para assegurar que o barramento não fica parado enquanto se processa a interrupção e os buffers são recarregados. Note-se que isto impõe uma restrição a L que tem que ser maior que I . Fazer L igual ou menor a I leva a que o barramento CAN fique parado enquanto se processa a interrupção, mas também disponibiliza mais buffers para mensagens periódicas de alta velocidade. Isto só é útil se as mensagens de baixa velocidade forem apenas uma pequena parte da carga e as mensagens de alta velocidade esporádicas são poucas ou quase inexistentes.

Se $N_{Hp} > B$, então todas as mensagens tem que ser armazenadas numa fila em software. Desta forma obtém-se apenas uma fila ordenada por prioridades para todas as mensagens de saída e todos os B buffers são preenchidos a partir desta fila. É de referenciar que esta situação foi a que ocorreu na experiências efectuadas.

3.1 - Sobrecargas

Para fluxos com buffers dedicados, a sobrecarga no processador é referente apenas ao cálculo do identificador da mensagem e à transferência dessa mesma mensagem e do seu identificador para o controlador do barramento. Note-se que os dados da mensagem podem ser copiados directamente do espaço de utilizador para o controlador do barramento de forma a manter a sobrecarga no mínimo.

Para mensagens armazenadas em fila de software, existe uma sobrecarga adicional da inserção da mensagem na fila (incluindo a cópia dos 8 ou menos bytes de dados da mensagem do espaço de utilizador para o espaço do chip antes de inserir na fila), mais a sobrecarga para o processamento das interrupções geradas aquando da

transmissão da mensagem. Esta sobrecarga pela interrupção ocorre uma vez em cada $Q - I$ transmissões de mensagens, onde Q é o número de buffers a serem preenchidos a partir da fila (Q pode ser B ou L , dependendo se as mensagens periódicas de alta velocidade são armazenadas ou não). Também cada mensagem irá potencialmente ser ultrapassada por outra. A mensagem ultrapassada que já tinha sido copiada para o controlador do barramento uma vez irá ser copiada de novo, pelo que a sobrecarga pela preemptividade é equivalente à sobrecarga da transferência da mensagem para o adaptador de rede [ZS, 2000].

Note-se que no algoritmo DM as sobrecargas são idênticas. A diferença reside em que os identificadores dos fluxos das mensagens em escalonamento DM são fixos, pelo que não é necessário recalculá-los de cada vez. Para além desta diferença, a implementação do algoritmo DM não é diferente da implementação do algoritmo MTS.

4 - Síntese

O algoritmo MTS [ZS, 95] foi desenvolvido para oferecer um melhor desempenho no escalonamento em relação ao algoritmo DM. Uma vez que as metas temporais no algoritmo MTS são quantificadas, é de esperar que o seu desempenho seja muito próximo do algoritmo EDF, desde que a duração das mensagens se mantenha constante. Considere-se algoritmo EDF* como um escalonamento ideal derivado do algoritmo EDF mas que apenas necessita de um identificador de 11 bits. Então, a performance de algoritmo EDF* seria o limite superior do algoritmo MTS.

Visto que ainda não existem resultados analíticos que comprovem qual o melhor escalonamento, apenas se pode recorrer a experimentações práticas para efectuar as avaliações, sendo necessário um modelo de carga a aplicar.

Como a análise que se vai realizar irá incidir mais nos limites da rede e do processador, optou-se por gerar mensagens de uma forma sequencial, sempre de forma a colocar o maior número possível de mensagens na rede com as metas temporais mais críticas possíveis, mas tendo sempre em linha de conta a capacidade de processamento do processador dos módulos das placas de simulação.

No capítulo seguinte é apresentado em detalhe o modelo de carga idealizado, assim como a avaliação das implementações dos vários escalonamentos utilizados.

Capítulo 6

Análise experimental do escalonamento de tempo-real de mensagens CAN

Capítulo 6 - Análise experimental do escalonamento de tempo-real de mensagens em CAN

1 - Introdução

Após a realização das análises teóricas nos capítulos anteriores, é altura de tentar fazer a implementação destes algoritmos propostos, de modo a aferir os resultados que verifiquem ou ponham em causa as análises efectuadas por Tindell [TH, 94] e Zuberi [ZS, 95].

A forma adoptada para efectuar os testes foi a utilização de cartas ou módulos com nós CAN, sendo constituída cada uma por um microcontrolador e um controlador de rede CAN (módulo Phytex phyCore ADuC812). Desta forma, utilizou-se a carga máxima possível nestas cartas, recebendo os dados respectivos no computador, a partir de uma carta CAN da IXXAT aí instalada.

Cada um dos dois nós foi programado com os escalonamentos a testar, tendo-se obtido os resultados apresentados no final deste capítulo.

2 - A escolha das cargas e das placas

De forma a efectuar a experimentação, foi necessário escolher as placas de entre as que foram estudadas no Capítulo 3. Tal como foi indicado, as que melhor se adequavam pelas suas características para criar os nós da rede CAN pretendidos foram as phyCORE ADuC812 da Phytex, as quais possuem o microcontrolador ADuC812 da Analog Devices, que já inclui conversores A/D e D/A, e o controlador para rede CAN SJA1000 da Philips Semiconductors, o mais usado actualmente.

De seguida foi realizado um estudo aprofundado da programação destas placas (daqui em diante designadas por nós), de forma a poder efectuar a implementação dos escalonamentos propostos.

Contudo, antes de implementar os escalonamentos, é necessário escolher as cargas das mensagens, isto é, quais as metas temporais e qual o número de mensagens que cada nó deverá conter. Desta forma, e após vários testes, foi decidido que em cada nó seriam colocados 4 fluxos de mensagens, cada um com uma determinada meta temporal.

Após alguns testes de envio e análises efectuadas com osciloscópios, conseguiu-se determinar que embora cada trama de mensagem CAN deverá ter teoricamente 55 bits de comprimento, o envio de uma trama com identificador 0 e um dado no campo de dados de valor 0 na prática produziu uma trama com 4 “*stuff bits*”, obtendo-se uma trama com um comprimento total de 59 bits. Na Figura 50 pode ser observada a trama teórica assim como a trama obtida. Na Figura 51 podem ser observadas as fotografias tiradas durante os testes.

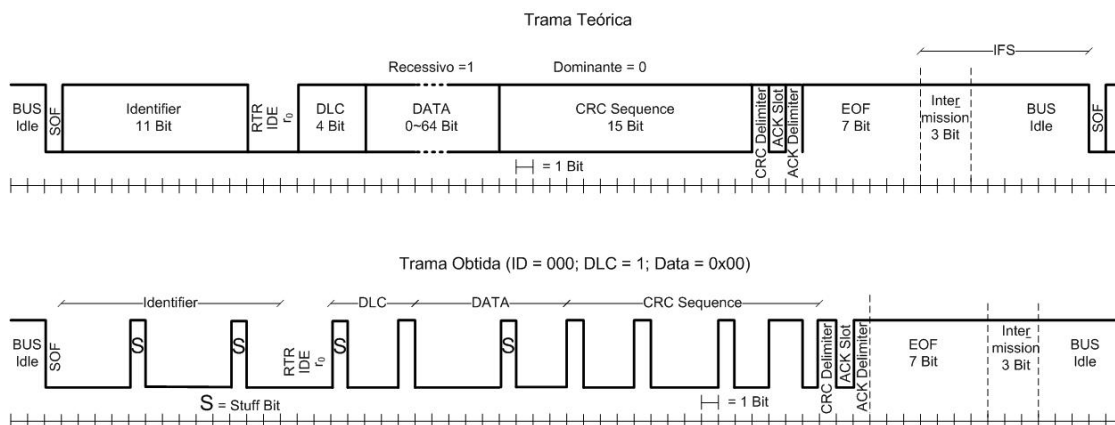


Figura 50 – Tramas de mensagens CAN: teórica (em cima) e obtida por testes (em baixo)



Figura 51 – Trama de mensagem CAN observada num: osciloscópio semi-digital (à esquerda); osciloscópio digital (à direita).

Por forma a simplificar os cálculos, foram ignorados os “*stuff bits*” e considerou-se que cada mensagem tem uma duração de 5,5ms. Para além disso, atribuiu-se uma periodicidade aos fluxos de mensagens mais prioritários (fluxo 1 e fluxo 2, nó 2 e nó 1 respectivamente) de 30ms. Este valor mantém-se constante independentemente das cargas de utilização aplicadas à rede. Desta forma obtém-se um ponto comum às várias cargas para comparação.

O modo como foram calculadas as metas temporais para os restantes fluxos (do fluxo 3 ao fluxo 8), conforme as necessidades de carga de utilização para teste, foi o seguinte:

- Tal como foi afirmado, atribuiu-se a periodicidade de 30 ms aos fluxos de mensagens mais prioritários. Decidiu-se também atribuir uma periodicidade aos outros fluxos de mensagens que permitisse a existência de no máximo 4 ou 5 mensagens consecutivas dentro do mesmo período, considerando a existência de uma carga total de rede de 90 %.
- De seguida atribuiu-se um factor multiplicativo aos restantes fluxos, de forma que existisse uma relação entre a periodicidade do fluxo de mensagens mais prioritário e a dos outros fluxos de mensagens. Este factor de relação é o incremental 2 de T_1 , referente ao fluxo mais prioritário, isto é:

$$\begin{aligned}
 T_1 &= \text{valor atribuido} \\
 T_2 &= 2a \times T_1 \\
 T_3 &= 4a \times T_1 \\
 T_4 &= 6a \times T_1
 \end{aligned}
 \tag{eq. 24}$$

- Mas o ideal é que exista também uma relação entre estas periodicidades e a utilização total da rede. Esta relação é conseguida através da multiplicação de um factor pelas periodicidades. O princípio a partir do qual este factor foi obtido é o seguinte:

$$U_{total} = n^o \text{ nós} \left(\frac{C}{T_1} + \frac{C}{T_2 \times a} + \frac{C}{T_3 \times a} + \frac{C}{T_4 \times a} \right)
 \tag{eq. 25}$$

onde a é o factor de multiplicação entre a utilização total da rede e as periodicidades de cada fluxo de mensagens. Assim, para cada carga de utilização pretendida é necessário calcular o respectivo factor pela equação seguinte:

$$a = \frac{30,25}{90 \times \frac{U}{100} - 33}; U = [\%]
 \tag{eq. 26}$$

Assim, aplicando as equações eq. 24 e eq.26 para uma carga, por exemplo, de 75% obtemos os seguintes factores e metas temporais:

$$\begin{aligned}
 a &= 0,8768 \\
 T_1 = 30\text{ms} &\Rightarrow U_1 = 18,3\% \\
 T_2 = 53\text{ms} &\Rightarrow U_2 = 10,4\% \\
 T_3 = 105\text{ms} &\Rightarrow U_3 = 5,2\% \\
 T_4 = 158\text{ms} &\Rightarrow U_4 = 3,5\%
 \end{aligned}$$

É de referir que os valores dos cálculos são arredondados às unidades, devido às características do microcontrolador usado nos nós. Desta forma obtemos assim vários quadros com cargas desde até 70% até 100%, os quais vão ser usados na análise experimental das várias implementações de escalonamentos por software.

Não nos podemos esquecer que os fluxos são iguais em termos de periodicidade para as duas placas, pelo que de seguida são apresentadas as várias tabelas (Tabela 9) com os valores para T_i e C_i respectivos mas apenas para uma das placas. O valor para a utilização na rede vem multiplicado por 2.

Fluxo	C(ms)	T(ms)	U(%)
1	5,5	30	18,3%
3	5,5	60	9,2%
5	5,5	121	4,5%
7	5,5	181	3,0%

Total= 35,1%

a) 70%

Fluxo	C(ms)	T(ms)	U(%)
1	5,5	30	18,3%
3	5,5	53	10,4%
5	5,5	105	5,2%
7	5,5	158	3,5%

Total= 37,4%

b) 75%

Fluxo	C(ms)	T(ms)	U(%)
1	5,5	30	18,3%
3	5,5	47	11,7%
5	5,5	93	5,9%
7	5,5	140	3,9%

Total= 39,9%

c) 80%

Fluxo	C(ms)	T(ms)	U(%)
1	5,5	30	18,3%
3	5,5	42	13,1%
5	5,5	83	6,6%
7	5,5	125	4,4%

Total= 42,5%

d) 85%

Fluxo	C(ms)	T(ms)	U(%)
1	5,5	30	18,3%
3	5,5	38	14,5%
5	5,5	76	7,2%
7	5,5	113	4,9%

Total= 44,9%

e) 90%

Fluxo	C(ms)	T(ms)	U(%)
1	5,5	30	18,3%
3	5,5	35	15,7%
5	5,5	69	8,0%
7	5,5	104	5,3%

47,3%

f) 95%

Tabela 9 – Valores obtidos para C_i e T_i , conforme a carga total pretendida na rede.

Nestas tabelas, temos os seguintes componentes: *Fluxo*, identificador do fluxo de mensagens; *C*, correspondente à duração (média) da trama que irá circular na rede; *T*, correspondente à periodicidade do fluxo de mensagens respectivo; *U*, correspondente à utilização da rede pelo fluxo de mensagens respectivo; *Total*, refere-se à soma das utilizações individuais dos fluxos de mensagens para a rede, devendo-se multiplicar pelo número de nós existentes (no caso das experimentações efectuadas, foram utilizados dois nós).

Na Tabela 10 é apresentada a quantidade de mensagens de cada um dos fluxos dos dois nós, que é enviada para a rede, conforme a carga de utilização. Estes valores foram obtidos de forma experimental para cada uma das cargas, obrigando os fluxos mais rápidos (fluxo 1 e fluxo 2) a que quando atingissem o valor 2000 fizessem uma paragem total do programa e enviassem os dados estatísticos pela porta série do nó.

Estes dados estatísticos, que serão apresentados em tabelas mais à frente durante a explicação dos algoritmos práticos usados, são referentes ao total de mensagens enviado por cada fluxo do nó e respectiva quantidade de metas temporais perdidas.

Msg/Fluxo	Cargas						
	70%	75%	80%	85%	90%	95%	100%
F1	2000	2000	2000	2000	2000	2000	2000
F2	2000	2000	2000	2000	2000	2000	2000
F3	1000	1132	1276	1428	1579	1714	1875
F4	1000	1132	1276	1428	1579	1714	1875
F5	496	572	645	723	790	870	938
F6	496	572	645	723	790	870	938
F7	332	380	429	480	531	577	625
F8	332	380	429	480	531	577	625

Tabela 10 – Tabela de quantidade de mensagens de cada fluxo, conforme a carga total aplicada na rede

3 - Descrição das diferentes estratégias de escalonamentos independentes implementados

De forma a verificar estes dados teóricos, foi desenvolvido algum software para efectuar testes experimentais, dos quais resultaram três modelos de escalonamento:

- **Algoritmo 1** – Sem escalonamento, filas FIFO em cada nó (CAN convencional).
- **Algoritmo 2** – Escalonamento por prioridades local.
- **Algoritmo 3** – Escalonamento por prioridades local e épocas temporais.

3.1 - Algoritmo 1 – Sem escalonamento, filas FIFO em cada nó (CAN convencional)

Este é o algoritmo natural das redes de barramento CAN, no qual a primeira mensagem a ser gerada e a pedir transmissão é a primeira a ser enviada pelo nó, mas não necessariamente a primeira a ser aceite pelo barramento, isto é, faz parte da natureza do barramento CAN dar a prioridade a mensagens que possuam o identificador mais baixo.

Optou-se por efectuar um conjunto de experiências com um barramento CAN convencional, ou seja sem escalonamento e com filas FIFO, por forma a podermos avaliar o grau de melhoria introduzido pelas propostas de escalonamento efectuadas nesta tese.

Para ser efectuada a implementação dos algoritmos foi aplicado o conceito de ‘fila circular’, o qual pode ser observado na Figura 52:

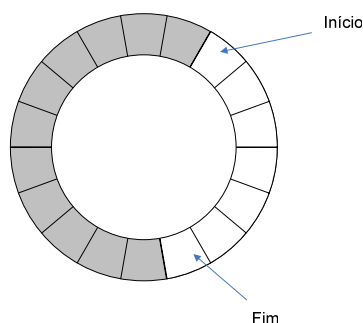


Figura 52 – Princípio da fila circular

A forma como este algoritmo foi implementado no nó ao nível do software pode ser observado no fluxograma, do qual é apresentado um excerto na Figura 53.

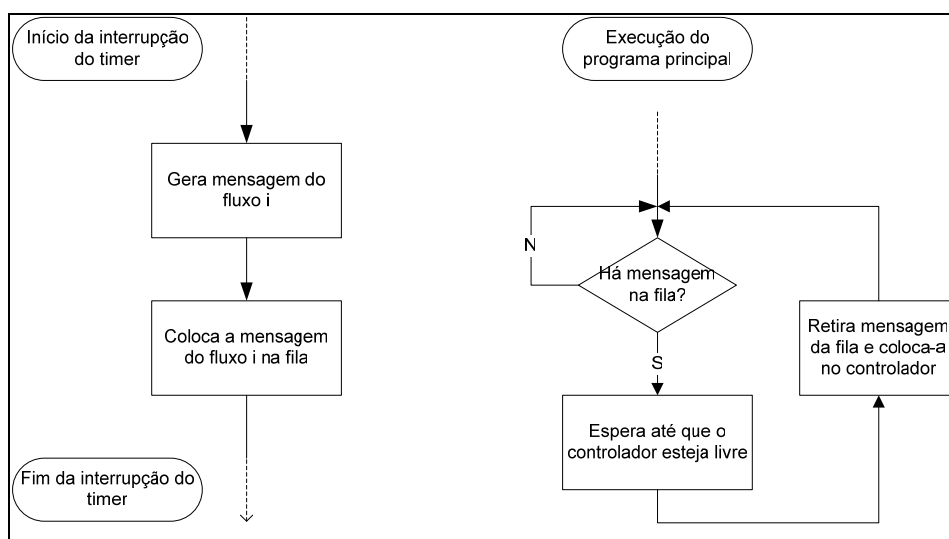


Figura 53 – Excerto do fluxograma para o escalonamento em fila FIFO

A implementação deste escalonamento é simples. A geração dos fluxos de mensagens é desencadeada pela activação da interrupção do microcontrolador a cada 500µs. Mas só é realmente gerada uma mensagem quando o contador que simula a meta temporal desse fluxo atingir o valor zero. Depois da mensagem ser gerada, a mesma é colocada numa fila por software, do tipo FIFO (Figura 53, à esquerda).

No decorrer normal do programa, caso haja a detecção de uma mensagem na fila pronta para envio, a mesma é então retirada da fila e colocada no buffer do controlador da rede, até que possa ser enviada para a rede (Figura 53, à direita).

Aplicando-se este funcionamento com o conceito da fila circular, obtemos o gráfico da figura seguinte:

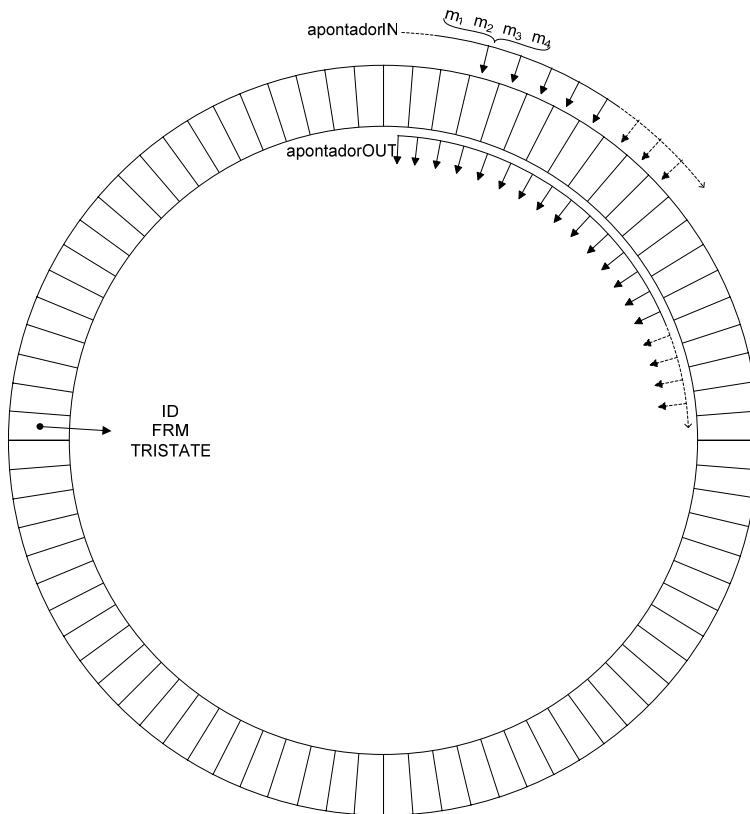


Figura 54 – Princípio da fila circular para o escalonamento em fila FIFO

A forma como este funcionamento foi implementado ao nível do software pode ser observado no excerto de código seguinte:

```
...
aux = apontadorIN + 3;
apontadorIN = aux & MASKEND;
while (*apontadorIN != 0){
    aux = apontadorIN + 4;
    apontadorIN = aux & MASKEND;
}
apontadorIN -= 3;
*apontadorIN++ = mi.ID >> 3;
*apontadorIN++ = mi.ID;
*apontadorIN++ = mi.FRM;
*apontadorIN++ = mi.TRISTATE;
...
```

onde `apontadorIN` é referente à posição na fila FIFO em memória externa; `mi.ID` é referente à componente do identificador da mensagem `i`; `mi.FRM` é referente aos dados

no campo de dados da mensagem i (neste caso, será um inteiro incremental entre 0 e 255); $mi.TRISTATE$ é referente ao estado da mensagem i , com os seguintes significados: 0 – não existe nada; 1 – mensagem na fila, pronta para ser colocada no buffer do controlador de rede; 2 – mensagem em processamento no controlador de rede.

Devido ao comportamento previsível deste algoritmo, recorrendo-se a um intervalo de tempo equivalente a dois períodos do fluxo de mensagens mais lento, é possível obter-se um gráfico de comportamento da rede (exemplo realizado para uma carga global de 90%), tal como se mostra na Figura 55.

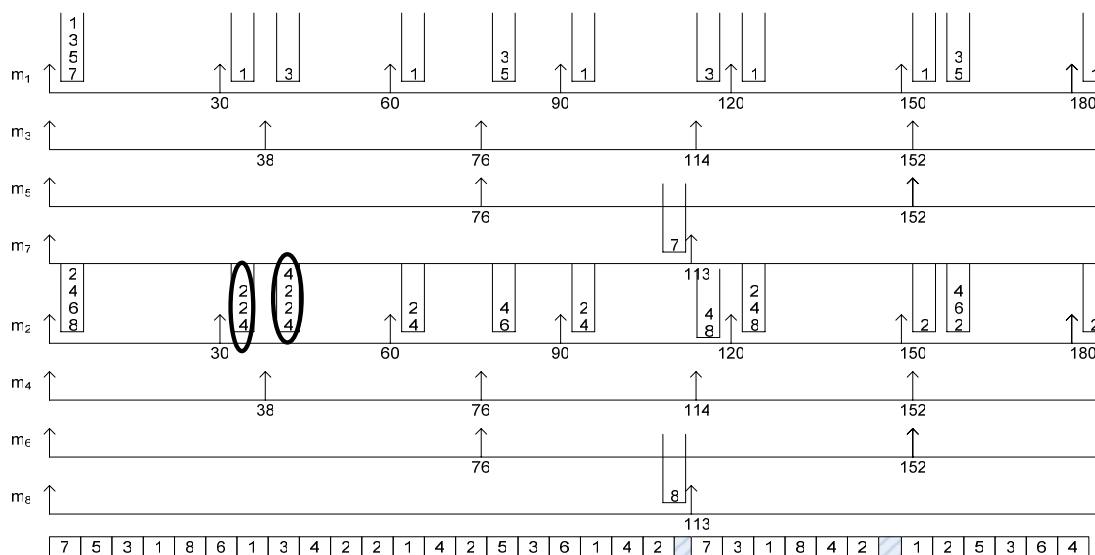


Figura 55 – Comportamento da rede com o escalonamento em fila FIFO

Neste gráfico fica evidente que o protocolo CAN, quando implementado no seu modo convencional, possui diversas falhas, tais como:

- Perda de metas temporais, sinalizadas pelas elipses representadas na Figura 55;
- Não só os fluxos de mensagens mais prioritárias perdem metas temporais, ou seja, utilizando filas FIFO teremos um barramento CAN com um comportamento imprevisível, visto que tanto pode ser uma mensagem de um fluxo prioritário a perder a sua meta temporal, como uma mensagem de um fluxo menos prioritário;
- O instante em que um fluxo sofre perdas de metas temporais é também imprevisível;
- Existe um tempo durante o qual o barramento está livre, em que não é enviada qualquer mensagem dos controladores, apesar de anteriormente terem sido perdidas metas temporais devido às inversões de prioridade.

Após a realização das experiências com a implementação deste algoritmo nos nós, chegou-se aos valores de metas temporais ultrapassadas que se representam na Tabela 11:

Algoritmo 1 – Sem escalonamento, filas FIFO em cada nó (CAN convencional)						
Fluxos	Cargas					
	70%	75%	80%	85%	90%	95%
m1	0	0	1	4	7	10
m2	0	7	15	29	65	72
m3	0	0	0	0	0	0
m4	0	0	0	0	0	0
m5	0	0	0	0	0	0
m6	0	0	0	0	0	0
m7	0	0	0	0	0	0
m8	0	0	0	0	0	0

Tabela 11 – Resultados da experimentação do algoritmo em fila FIFO

Tendo presente o funcionamento do barramento CAN, as mensagens que possuem menor identificador são as que ganham vez no barramento. Mas antes de ser colocada uma mensagem no barramento, a mesma teve que ser colocada numa fila de software do tipo FIFO. O gráfico da Figura 56 mostra a relação entre as cargas e as metas temporais perdidas, onde é evidente a perda das metas temporais nos fluxos de maior prioridade.

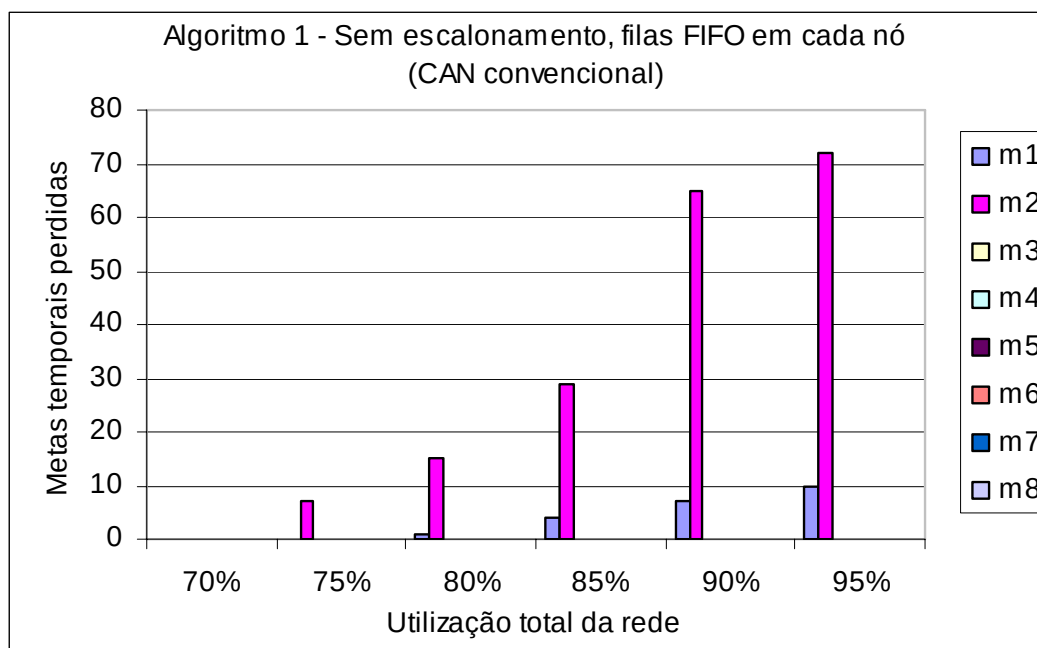


Figura 56 – Gráfico dos resultados da experimentação do algoritmo em fila FIFO

De forma a detectar se uma mensagem perdeu a meta temporal, adoptou-se o pressuposto de que se perde uma meta temporal se uma mensagem é colocada na fila sem que a anterior tenha conseguido ser retirada da fila e colocada no controlador de barramento. Isto é feito ao nível do nó, como pode ser observado pelo extracto de código seguinte:

```

...
nrMSG1++; // Total das mensagens do fluxo
msgCount1++; // Controlo de deadlines
if (msgCount1 > 1) lostDeadl1++; // Contagem de metas temporais
// perdas
...
    
```

Sempre que entra na função para criar uma nova mensagem de um fluxo e colocá-la na fila, é incrementado o contador $nrMSG_i$, o qual conta o total de mensagens do fluxo i . Em simultâneo é incrementada uma variável de controlo, $msgCount_i$ que indica se já existe uma mensagem do mesmo fluxo na fila de espera. De seguida faz o teste desta mesma variável. Em caso positivo, existe uma perda de meta temporal, pelo que é incrementado o contador que contabiliza as metas temporais perdidas, $lostDeadl_i$. Esta mesma variável também será enviada como dado estatístico. Repare-se que isto é feito com referência no nó.

Como as mensagens dos fluxos menos prioritários têm uma muito maior periodicidade, verifica-se que no instante de geração de nova mensagem, já não existe nenhuma mensagem desse fluxo na fila, ao contrário das mensagens dos fluxos mais prioritários que são gerados mais frequentemente, obtendo-se então os resultados da Tabela 11. Este facto pode ser observado no exemplo da Figura 57:

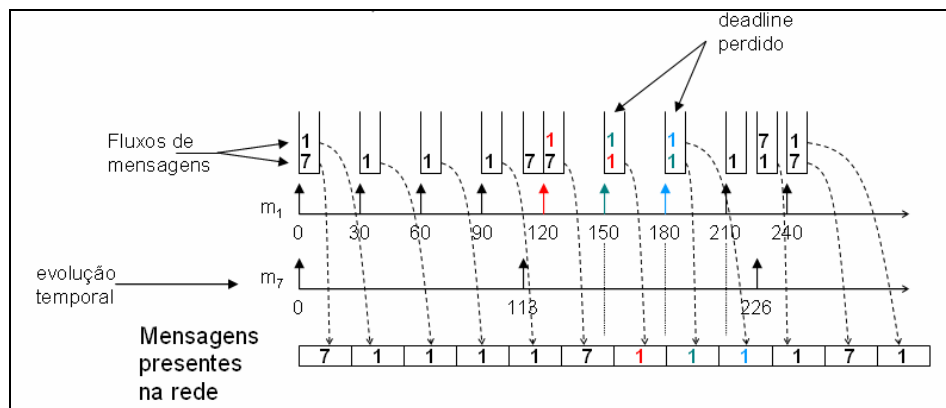


Figura 57 – Relação entre as mensagens geradas pelo fluxo que tem mais prioridade e o que tem menos prioridade

Na Figura 57 note-se que no instante $t = 150$ em que é gerada uma mensagem do fluxo 1, ainda existia uma mensagem desse mesmo fluxo gerada no instante $t = 120$, originando uma situação de perda de meta temporal do fluxo 1 (assinalada com a elipse), ao passo que não existem perdas de metas temporais do fluxo 7. A mesma situação volta a surgir no instante $t = 180$.

Ou seja, não efectuando escalonamento e utilizando uma fila FIFO (método tradicional), temos uma situação muito indesejável: os fluxos de mensagens para os quais são perdidas metas temporais, são os fluxos mais prioritários. Este modo de funcionamento é absolutamente indesejável para um sistema de comunicação que se pretende ser de tempo-real.

3.2 - Algoritmo 2 – Escalonamento por prioridades local

O segundo algoritmo de escalonamento analisado tem como base o algoritmo não-preemptivo *Earliest Deadline First* – EDF, também usando o princípio da fila circular para o armazenamento das mensagens a enviar para a rede.

A fila circular usada neste algoritmo difere da anterior, pela forma como os dados são nela inseridos. Neste algoritmo, em vez da mensagem gerada na interrupção ser colocada de imediato no topo da fila, esta será colocada num local adequado, em termos das suas características temporais. O cálculo da posição de inserção é efectuado em função do período do fluxo de mensagens, por forma que quando se geram mensagens mais prioritárias, estas tenham lugar no topo da fila, sem a necessidade de pesquisar a fila toda e reordená-la, de cada vez que se faz nova inserção.

A fila circular contém um apontador, o qual a cada 5 ms será incrementado. Pretende-se com este apontador simular a passagem do tempo (*timeout*)

Cada fluxo de mensagens tem uma determinada periodicidade de ocorrência. Ao inserir uma nova mensagem na fila, verifica-se previamente qual o valor dessa periodicidade. Este valor é então somado ao valor de *timeout*. Será o resultado desta soma que indicará qual a posição na fila onde a mensagem irá ser inserida.

No espaço de tempo que demora a gerar duas mensagens consecutivas de um fluxo menos prioritário, serão geradas múltiplas mensagens de um fluxo mais prioritário (Figura 57) e também o valor de *timeout* aumenta (passagem do tempo). No momento de serem gerados os fluxos de mensagens mais prioritários, como a soma da sua periodicidade com o valor de *timeout* é inferior à soma da periodicidade dos fluxos de mensagens menos prioritários com o valor de *timeout*, as mensagens mais prioritárias serão colocadas mais próximas do topo da fila, pelo que serão também essas a ser enviadas primeiro para o controlador.

Poderia, no entanto, surgir o problema de essa posição já estar ocupada. Nessa situação, mesmo que fosse uma mensagem originalmente com menor prioridade (vamos supor que será uma mensagem do fluxo de mensagens 8), com o passar do tempo e o incremento do *timeout* esta mensagem ficou com maior prioridade, o que significa que pode ter uma prioridade superior ao fluxo de mensagens 1, pelo que a posição de inserção vai novamente ser incrementada de uma unidade.

Desta forma é garantido que as mensagens com uma sua meta temporal mais próxima serão enviadas em primeiro lugar, tal como é definido no algoritmo EDF. Isto mesmo pode ser observado na Figura 58.

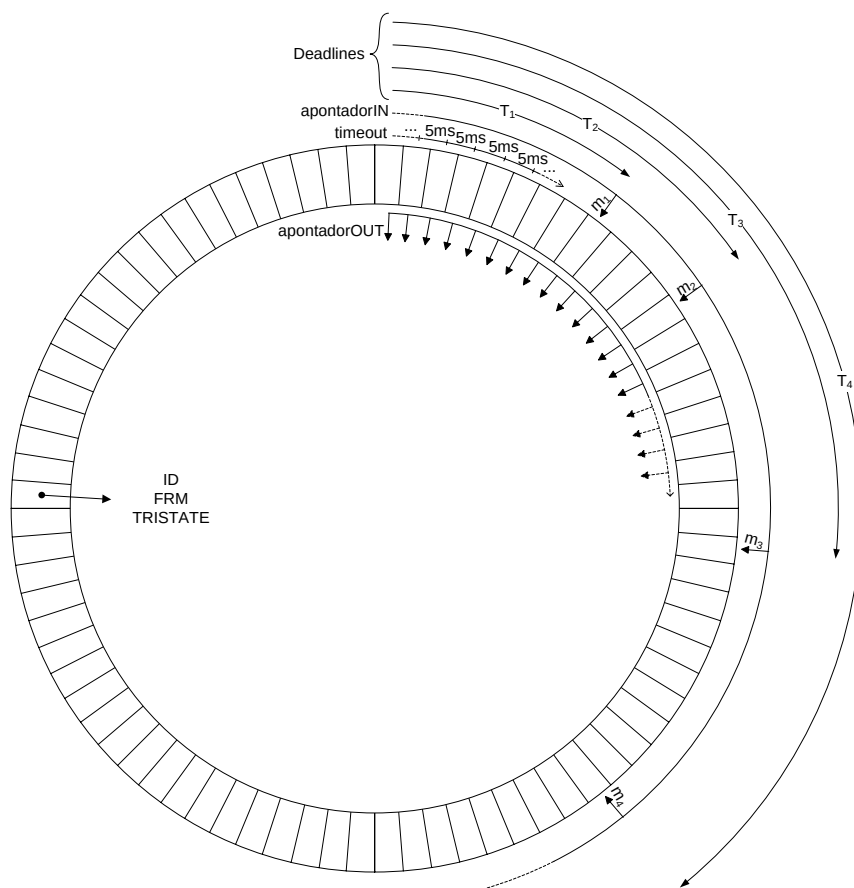


Figura 58 – Princípio da fila circular para o escalonamento local

A forma como este algoritmo foi implementado no nó ao nível do software pode ser observado no fluxograma, do qual é apresentado um excerto na Figura 59.

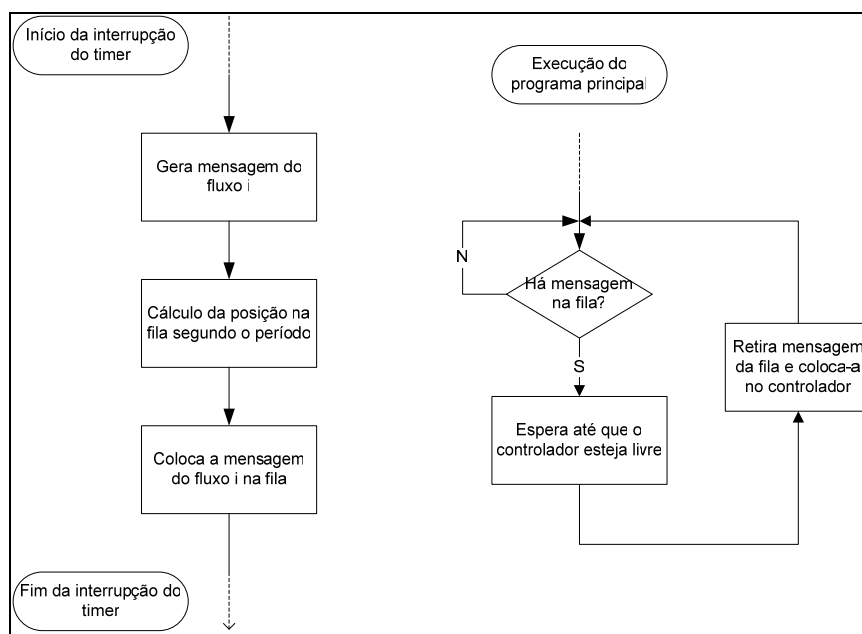


Figura 59 - Excerto do fluxograma para o escalonamento local da fila

Em termos de software, esta implementação significa que é necessário um cálculo prévio para colocar as mensagens na fila, da seguinte forma:

```
...
    lsbAptIn += APi;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;

    apontadorIN += 3;
    while (*apontadorIN != 0x00){
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;

    *apontadorIN++ = mi.ID >> 3;
    *apontadorIN++ = mi.ID;
    *apontadorIN++ = mi.FRM;
    *apontadorIN++ = mi.TRISTATE;
...
```

onde `lsbAptIn` é um apontador auxiliar, o qual ao ser incrementado pela quantidade `APi` (valor do período do fluxo), irá indicar a que distância do valor de timeout (`endTimeout`) irá ser colocada a mensagem na fila. A segunda parte do código é para garantir que os dados não se sobrepõem aos que ainda estiverem na fila. A contagem das mensagens que perderam a sua meta temporal é idêntica à do escalonamento anterior.

O modo como as mensagens são retiradas da fila é idêntico ao escalonamento por fila FIFO, mas em vez de depois de retirada a mensagem da fila, o apontador de saída apontar para a próxima posição, é forçado a reiniciar no valor timeout.

Devido ao comportamento previsível deste algoritmo, recorrendo-se a um espaço temporal equivalente a dois períodos do fluxo de mensagens mais lento, é possível obter-se um gráfico de comportamento da rede (exemplo realizado para uma carga global de 90%), tal como se mostra na Figura 60.

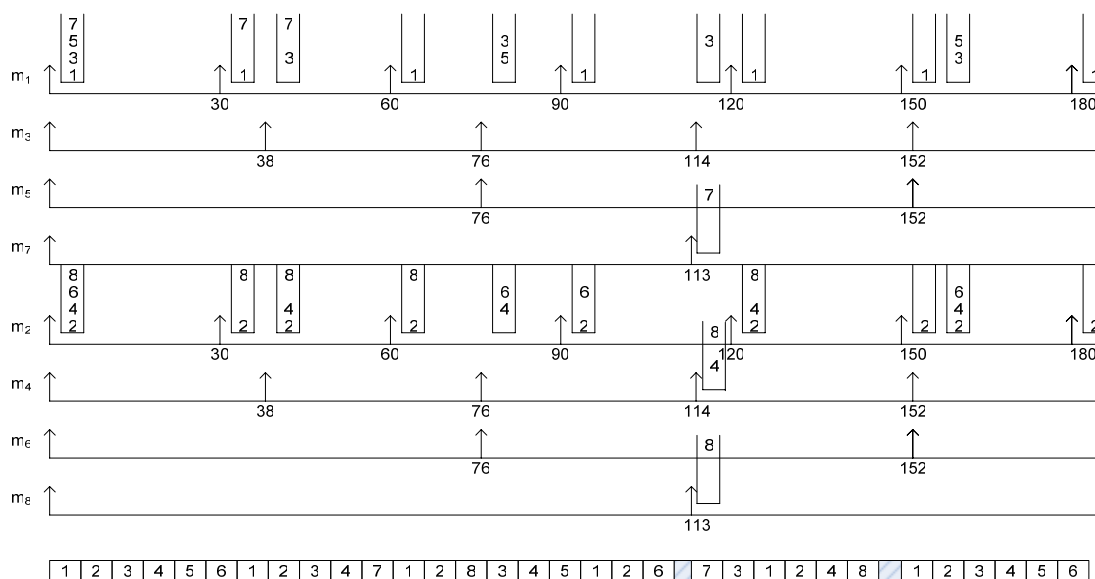


Figura 60 – Comportamento da rede com o escalonamento local da fila

Neste gráfico fica evidente que o protocolo CAN, quando em conjunto com uma implementação de um escalonamento por prioridades local tal como o que foi proposto, elimina algumas das falhas existentes na sua implementação convencional, tais como:

- Reordenação das mensagens na fila, de forma a garantir que a mensagem mais prioritária está na cabeça da fila, isto é, evita a inversão de prioridades;
- Diminui a quantidade de metas temporais perdidas para os fluxos de mensagens mais prioritários.

Após a realização das experiências com a implementação deste algoritmo nos nós, chegou-se aos valores de metas temporais perdidas da tabela seguinte:

		Algoritmo 2 – Escalonamento por prioridades local					
		Cargas					
Fluxos		70%	75%	80%	85%	90%	95%
m1		0	0	0	0	0	0
m2		0	0	0	0	1	1
m3		0	0	0	0	0	0
m4		0	0	0	0	0	1
m5		0	0	0	0	0	1
m6		0	0	0	0	1	49
m7		0	0	0	0	0	5
m8		0	0	0	0	6	159

Tabela 12 – Resultados da experimentação do algoritmo de escalonamento por prioridades local

Como pode ser confirmado pela Tabela 12, as mensagens mais prioritárias, que são colocadas na cabeça da fila, quase não sofrem perda de metas temporais. Já as mensagens menos prioritárias, que são colocadas na cauda da fila, demoram mais a sair para a rede e por isso perdem mais metas temporais. No gráfico da Figura 61 pode ser melhor observada esta relação.

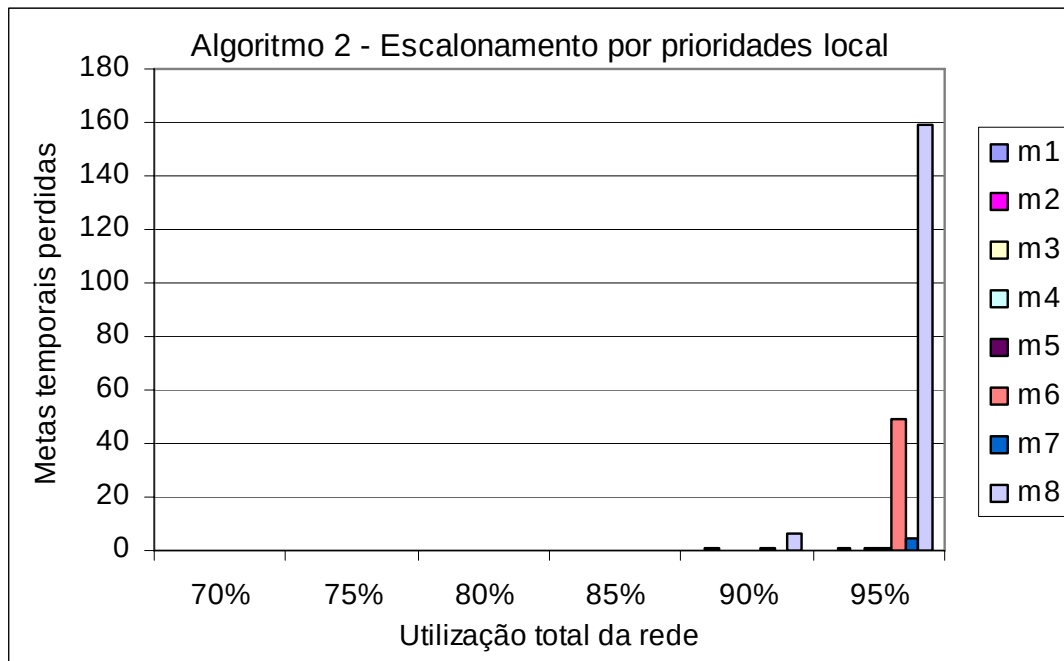


Figura 61 – Gráfico dos resultados da experimentação do algoritmo de escalonamento por prioridades local

Estes resultados obtidos foram os esperados para este escalonamento.

3.3 - Algoritmo 3 – Escalonamento por prioridades local e épocas temporais

Este algoritmo tem como base o algoritmo anterior de escalonamento local, mas acrescenta-lhe um cálculo para incorporar no identificador a enviar para a rede, designado de “cálculo de épocas”. Desta forma, tenta-se aproveitar a melhoria introduzida pelo escalonamento local referente às mensagens prioritárias, mas mantendo o princípio convencional de funcionamento do barramento CAN no caso de existirem mensagens com o mesmo identificador. Este modo de implementação é idêntico ao algoritmo MTS apresentado por Zuberi [ZS, 95], tendo como base para o cálculo do identificador das mensagens a equação eq. 27:

$$meta\ temporal = \begin{cases} \frac{x - timeout}{16}, & \text{se } \frac{x - timeout}{16} < 15 \\ 255, & \text{se } \frac{x - timeout}{16} \geq 15 \end{cases} \quad \text{eq. 27}$$

Esta implementação pode ser observada no extracto do fluxograma na Figura 62.

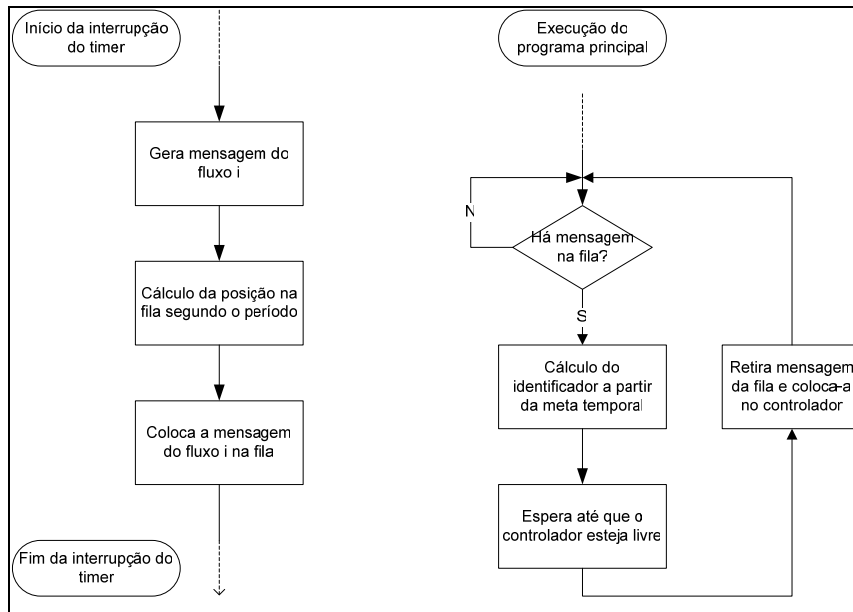


Figura 62 – Excerto do fluxograma para o escalonamento local e cálculo de épocas

Em termos de software, a implementação foi realizada da seguinte forma:

```

...
lsbAptIn += APi;
aux = endTimeout + lsbAptIn;
apontadorIN = aux & MASKEND;

apontadorIN += 2;
while (*apontadorIN != 0x00) {
    aux = apontadorIN + 4;
    apontadorIN = aux & MASKEND;
}
apontadorIN -= 2;

*apontadorIN++ = mi.ID;
*apontadorIN++ = mi.FRM;
*apontadorIN++ = mi.TRISTATE;
apontadorIN++;
...

```

A diferença entre este algoritmo e o anterior pode ser resumida nos dois pontos seguintes: o primeiro é que a inserção dos dados na fila não contempla a primeira parte do identificador do fluxo; a segunda é correspondente ao cálculo da época, que agora é incluída no identificador do fluxo a enviar para a rede. Este cálculo apenas é realizado quando é encontrada uma mensagem na fila para ser enviada para a rede, implementando-se em código C da seguinte forma:

```

...
fluxo = *apontadorOUT >> 3;
deadline = (apontadorOUT - endTimeout)/16;
if(deadline > 14) deadline = 255;
else deadline = deadline & DEADLIN;
...

```

onde *deadline* corresponde ao valor a incluir na construção do identificador.

Também neste algoritmo é usado o princípio da fila circular, mas como o cálculo necessário para o identificador da mensagem a colocar na rede só é necessário quando se encontra a mensagem a enviar e não quando esta é inserida na fila, o princípio da fila circular usado é o mesmo da fila circular para o escalonamento local, com a diferença indicada anteriormente.

Devido à complexidade existente neste algoritmo, não foi possível obter o gráfico de comportamento das mensagens na rede como nos anteriores. Após a implementação prática deste algoritmo, obteve-se a seguinte tabela de resultados:

Algoritmo 3 – Escalonamento por prioridades local e épocas temporais						
Fluxos	Cargas					
	70%	75%	80%	85%	90%	95%
m1	0	0	0	0	0	0
m2	0	0	0	0	0	0
m3	0	0	0	0	0	0
m4	0	0	0	0	0	0
m5	0	0	0	0	0	0
m6	0	0	0	0	0	5
m7	0	0	0	0	0	0
m8	0	0	0	0	0	22

Tabela 13 – Resultados da experimentação do algoritmo de escalonamento por prioridades local com cálculo de épocas

O gráfico da Figura 63 mostra a relação entre as cargas e as metas temporais perdidas, onde é evidente a perda das metas temporais nos fluxos de menor prioridade.

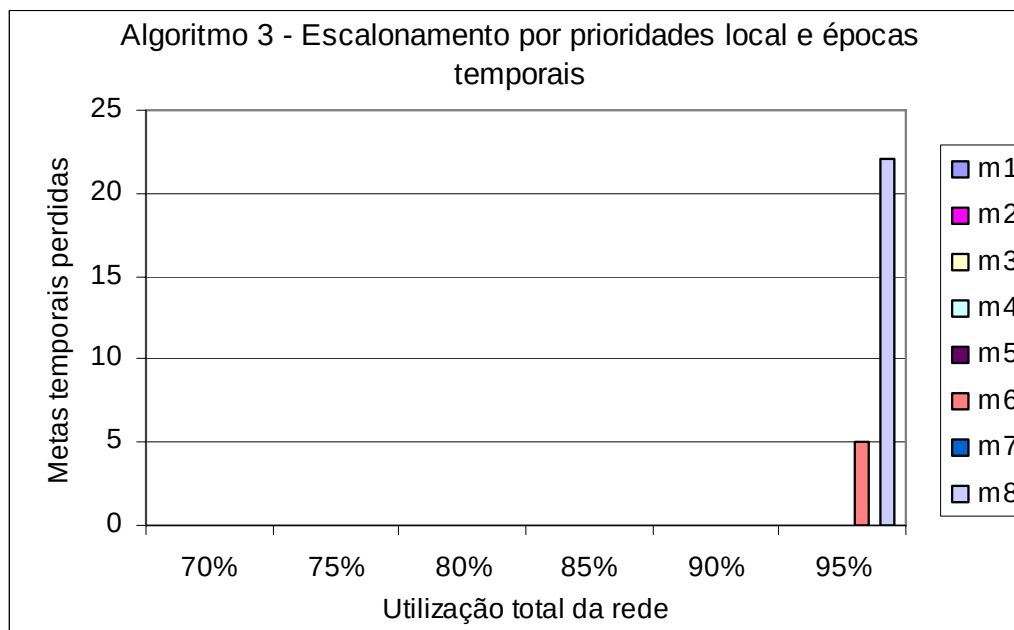


Figura 63 – Gráfico dos resultados da experimentação do algoritmo de escalonamento por prioridades local com cálculo de épocas

Neste gráfico fica evidente a grande melhoria em relação à implementação CAN convencional, assim como a ligeira melhoria em relação à implementação do algoritmo por prioridades local:

- Inexistência de metas temporais perdidas nos fluxos de maior prioridade;
- Diminuição da quantidade de metas temporais perdidas para os fluxos de mensagens de menor prioridade, comparativamente ao algoritmo anterior implementado.

4 - Síntese

A análise experimental efectuada vem comprovar que, a tentativa de captar o melhor de dois escalonamentos, neste caso baseado no Deadline Monotonic [ZS, 95] e Earliest Deadline First [ZS, 95], é possível, obtendo-se um terceiro escalonamento baseado no Mixed Traffic Scheduling [ZS, 95], designado por “escalonamento por prioridades local e épocas temporais”, o qual tem melhor rendimento e eficácia que os anteriores, no que se refere ao escalonamento de mensagens prioritárias.

O CAN convencional, que é baseado numa fila FIFO, não permite, no caso de cargas elevadas, o cumprimento das metas temporais dos fluxos de mensagens mais prioritários. A razão principal que leva à perda das metas temporais é o facto de existir “inversão de prioridades”, isto é, quando uma mensagem de um fluxo de mensagens menos prioritário é gerada e colocada na fila para enviar, não é removida ou reordenada na própria fila, até que o barramento esteja livre e seja enviada para a rede, o que vai bloquear uma mensagem de um fluxo de mensagens mais prioritário por tempo indeterminado.

Outra razão que também leva à perda das metas temporais de fluxos de mensagens prioritários é o facto de estes serem gerados mais rapidamente que os menos prioritários, pelo que quando uma nova mensagem mais prioritária é colocada na fila, com a ocorrência da inversão de prioridades em alguns dos casos, ainda existirá uma mensagem anterior do fluxo de mensagens mais prioritário na fila.

O software que foi desenvolvido inicialmente, aplicando um escalonamento por prioridades local na fila baseado no escalonamento *Earliest Deadline First*, permitiu observar experimentalmente que a simples ordenação na fila dos fluxos de mensagens, conforme a sua periodicidade, permite melhorar em muito os resultados no envio de fluxos de mensagens prioritários para a rede, evitando perdas de metas temporais. Contudo, este escalonamento levou a um aumento considerável de perdas de metas temporais de fluxos de mensagens menos prioritários.

Fazendo uma pequena alteração no software implementado, de forma a poder incluir o cálculo das metas temporais de Zuberi [ZS, 95], obteve-se um escalonamento por prioridades local e cálculo de épocas. Este escalonamento permitiu baixar a quantidade de metas temporais perdidas de fluxos de mensagens menos prioritárias, mas evitando a perda de metas temporais (quase na totalidade) de fluxos de mensagens mais prioritários. Note-se que, apesar da alteração da implementação do software de escalonamento por prioridades local para este último ser bastante simples, os resultados obtidos são significativamente superiores.

Desta forma fica demonstrado o principal objectivo desta dissertação: “é possível melhorar fortemente as características de comunicação de tempo-real do protocolo CAN, através da implementação de um pequeno módulo de software que implemente um escalonamento de tempo-real adequado para transmissão das mensagens”.

Capítulo 7

Conclusões

Capítulo 7 - Conclusões

O protocolo CAN de comunicação em barramento já tem alguns anos. Contudo, ainda é bastante utilizado pela sua simplicidade de implementação, fácil manuseamento e principalmente pela capacidade de lidar com fluxos de mensagens com prioridades. Devido a isto, o protocolo CAN foi proposto para norma, constituindo actualmente as normas ISO 11898-1, ISO 11898-2 e ISO 11898-3. De facto, este protocolo é bastante usado na indústria automóvel e existe bastante desenvolvimento de melhorias de forma a tornar este protocolo mais eficaz, para além de o adaptar para a realidade actual de outras indústrias.

Para ser realizada a análise experimental, procedeu-se a uma análise do tipo de mensagens que podem surgir numa rede CAN convencional, assim como de alguns modelos de escalonamentos mais usados para essas situações. De facto existem três tipos de fluxos de mensagens a considerar em constante circulação numa rede CAN: fluxo de mensagens periódicas (com metas temporais apertadas/rígidas), fluxo de mensagens esporádicas (com metas temporais apertadas/rígidas) e fluxo de mensagens aperiódicas não de tempo-real.

Assim, partiu-se de algumas análises e propostas de novos escalonamentos para redes CAN, de forma a verificar qual o que melhor se adequava e qual a sua facilidade de implementação ao nível dos nós da rede. Tendo como base principal as análises efectuadas por Tindell [TH, 94] e Zuberi [ZS, 95], adoptou-se como base para os escalonamentos a desenvolver os algoritmos DM, EDF e MTS.

O algoritmo MTS foi desenvolvido para oferecer melhor performance e escalonamento em relação ao algoritmo DM. Uma vez que as metas temporais no algoritmo MTS são quantificadas, é de esperar que a sua performance seja muito próxima do algoritmo EDF, desde que a duração das mensagens se mantenha constante. Considerando a existência de um algoritmo EDF*, o é visto como um escalonamento ideal derivado do algoritmo EDF mas que apenas necessita de um identificador de 11 bits, podemos então considerar que a performance do algoritmo EDF* é o limite superior do algoritmo MTS.

Como a análise experimental realizada incide mais nos limites da rede e do processador, optou-se por gerar mensagens de uma forma sequencial, sempre de forma a colocar o maior número possível na rede com as metas temporais mais críticas possíveis, mas tendo sempre em linha de conta a capacidade de processamento do processador dos módulos das placas de experimentação.

A análise experimental efectuada vem comprovar que realmente é possível obter elevadas melhorias no envio de fluxos de mensagens prioritárias na rede CAN. Através da criação de um pequeno software de teste tendo como base uma fila circular, foi possível verificar as perdas existentes nos fluxos de mensagens mais prioritários. Assim, foram efectuadas mais tentativas de implementação, de forma a tentar captar o melhor de dois escalonamentos, neste caso baseado no DM e EDF, verificando-se que é possível, obter-se um terceiro escalonamento baseado no MTS, designado por

“escalonamento por prioridades local e épocas temporais”, o qual tem melhor rendimento e eficácia que os anteriores, no que se refere ao escalonamento de mensagens prioritárias.

O CAN convencional, que é baseado em uma fila FIFO, não permite o cumprimento das metas temporais dos fluxos de mensagens mais prioritários, o que foi confirmado experimentalmente. A razão principal que leva à perda das metas temporais é o facto de existir “inversão de prioridades”, isto é, quando uma mensagem de um fluxo de mensagens menos prioritário é gerada e colocada na fila para enviar, não é removida ou reordenada na própria fila, até que o barramento esteja livre e seja enviada para a rede, o que vai bloquear uma mensagem de um fluxo de mensagens mais prioritárias por tempo indeterminado.

Outra razão que também leva à perda das metas temporais de fluxos de mensagens prioritárias é o facto de estas serem geradas mais rapidamente que as menos prioritárias, pelo que quando uma nova mensagem mais prioritária é colocada na fila, com a ocorrência da inversão de prioridades em alguns dos casos, ainda existirá uma mensagem anterior do fluxo de mensagens mais prioritário na fila.

O software que foi desenvolvido inicialmente, aplicando um escalonamento por prioridades local na fila, permitiu observar experimentalmente que a simples ordenação na fila dos fluxos de mensagens, conforme a sua periodicidade, permite melhorar em muito os resultados no envio de fluxos de mensagens prioritários para a rede, evitando perdas de metas temporais. Contudo, este escalonamento levou a um aumento considerável de perdas de metas temporais de fluxos de mensagens menos prioritários, que também foi verificado de forma experimental.

Fazendo uma pequena alteração no software implementado, de forma a poder incluir o cálculo das metas temporais de Zuberi [ZS, 95], obteve-se um escalonamento por prioridades local e cálculo de épocas. Este escalonamento permitiu baixar a quantidade de metas temporais perdidas de fluxos de mensagens menos prioritárias, evitando também a perda de metas temporais de fluxos de mensagens mais prioritários.

Verificou-se experimentalmente que esta alteração da implementação do software de escalonamento por prioridades local para o cálculo com épocas temporais é bastante simples, contudo os resultados obtidos são significativamente superiores. O software implementado foi concebido de forma a poder ser integrado num software mais complexo de comunicação em protocolo CAN.

Desta forma, o objectivo a que tinha sido proposto atingir esta dissertação, o qual era fazer o desenvolvimento de um pequeno software que melhorasse a eficácia do envio de fluxos de mensagens de tempo-real através de um barramento com o uso do protocolo CAN, foi alcançado, demonstrando-se assim que o barramento CAN ainda pode ter muitas mais potencialidades do que as que já são usadas, e pode ser bastante melhorado.

Bibliografia e referências

Bibliografia e referências

- [AB, 93] N. Audsley, A. Bums, M. Richardson, K. Tindell, A. Wellings; “Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling”; *Software Engineering Journal* 8(5), pág. 285-292 (Setembro 1993)
- [BN, 93] A. Bums, M. Nicholson, K. Tindell, N. Zhang; “Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System”; *Proc. Workshop on Parallel and Dist. Real-Time Syst.*, pág. 11-20 (Abril 1993)
- [Car, 2004] Mauricio Cardoso de Sá; “Programação C para microcontroladores 8051”; Editora Érica (2004)
- [Ets, 2001] Prof. Dr. Konrad Etschberger; "Controller Area Network: Basics, Protocols, Chips and Applications"; IXXAT Automation (2001)
- [JP, 86] M. Joseph and P. Pandya; “Finding Response Times in a Real-Time System”; *Computer Journal* 29(5), pág. 390-395 (Outubro 1986)
- [Law, 97] Prof. Wolfhard Lawrenz; "CAN System Engineering: From Theory to Practical Applications"; Springer (1997)
- [LK, 98] Mohammad Ali Livani and Jörg Kaiser; "EDF Consensus on CAN Bus Access for Dynamic Real-Time Applications"; *Lecture Notes in Computer Science*, Volume 1388, pág. 1088-1097 (1998)
- [Kei, 2001a] Keil Software; “Cx51 Compiler: Optimizing C Compiler and Library Reference for Classic and Extended 8051 Microcontrollers” (2001)
- [Kei, 2001b] Keil Software; “Getting Started with μ Vision2 and the C51 Microcontroller Development Tools” (2001)
- [KR, 88] Brian W. Kernighan, Dennis M. Ritchie; “C: A linguagem de programação padrão ANSI”; Editora Campus (1988)
- [KT, 97] Robert Kruse, C.L. Tondo, Bruce Leung; "Data Structures & Program Design in C - 2nd Ed."; Prentice-Hall (1997)
- [Nat, 2000] Marco Di Natale; "Scheduling the CAN Bus with Earliest Deadline Techniques"; *Real-Time Systems Symposium*, pág. 259-268 (Novembro 2000)
- [Phi, 98] Philips; "16-Bit 80C51XA Microcontrollers (eXtended Architecture) - Data Handbook IC25" (1998)

- [Phy, 2003] Phytex Technology Holding; “phyCORE – AduC812 Hardware Manual” (Fevereiro 2003)
- [Sch, 98] Thomas W. Schultz; “C and the 8051: Hardware, Modular Programming and Multitasking”; Prentice-Hall (1998)
- [SK, 91] David B. Stewart and Pradeep K. Khosla; "Real-Time Scheduling of Dynamically Reconfigurable Systems"; Systems Engineering, IEEE International Conference, pág. 139-142 (Agosto 1991)
- [SL, 90] L. Sha, J. P. Lehoczky, R. Rajkumar; “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”; IEEE Trans. on Computers, 39(9), pág. 1175-1185 (Setembro 1990)
- [TB, 94] Ken Tindell and Alan Burns; "Guaranteed Message Latencies for Distributed Safety-Critical Hard Real-Time Control Networks"; Documento nº YCS-94-229, Department of Computer Science, University of York (1994)
- [TC, 94] K. Tindell and J. Clark; “Holistic schedulability analysis for distributed hard real-time systems”; Microprocessing and Microprogramming, 40(2-3), pág. 117-134 (Abril 1994)
- [TH, 94] K. W. Tindell and H. Hansson; "Analysing Real-Time Communications: Controller Area Network (CAN)"; Real-Time Systems Symposium, pág. 259-263 (Dezembro 1994)
- [ZS, 94] Q. Zheng and K. G. Shin, “On the ability of establishing real-time channels in point-to-point packet-switched networks”; IEEE Trans. Communications, pág. 1096-1105 (Fevereiro/Março/Abril 1994)
- [ZS, 95] Khawar M. Zuberi and Kang G. Shin; "Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications"; Real-Time Technology and Applications Symposium, pág. 240-249 (Maio 1995)
- [ZS, 2000] Khawar M. Zuberi and Kang G. Shin; "Design and Implementation of Efficient Message Sheduling for Controller Area Network"; Computers, IEEE Transactions on Volume 49, Nº 2, pág.182-188 (Fevereiro 2000)

Anexo

Anexo

1 - Introdução

Aqui vão ser descritos em pormenor, os módulos de software que foram criados para as análises experimentais. Todos os programas foram desenvolvidos em linguagem ANSI C, uma vez que esta é mais fácil de trabalhar para os objectivos pretendidos e para além disso, acompanha as placas dos módulos para os nós da Phytex, um cdrom com software de desenvolvimento funcional de evolução, que permite fazer a compilação dos programas. Este software de desenvolvimento que acompanha os módulos dos nós é o μ Vision 2 da Keil Software, o qual permite desenvolver os módulos de software de um modo agradável devido à sua interface com o utilizador e a possibilidade de simulação interna que possui.

Os programas a seguir descritos, vão o ser a partir dos fluxogramas, apresentando-se a listagem do software logo a seguir. Muitos pontos dos vários módulos de software são comuns aos três algoritmos de escalonamento, visto que a implementação de um algoritmo seguinte é a evolução do anterior.

Antes de se passar à implementação do software propriamente dito, foi necessário identificar até que ponto era possível usar a linguagem C de programação. Embora existam algumas diferenças entre a linguagem C padrão ANSI [KR, 88] e a linguagem C aplicada na programação dos microcontroladores do tipo 8051 [Sch, 98] [Kei, 2001a] [Kei, 2001b], estas não foram significativas para o desenvolvimento do trabalho.

2 - Funções desenvolvidas para o software

Os três algoritmos implementados por software foram: sem escalonamento, fila FIFO; escalonamento por prioridades local; e escalonamento por prioridades local com cálculo de épocas temporais. Em cada um destes algoritmos foi utilizada uma fila circular, onde são guardadas as mensagens geradas pelos fluxos e que posteriormente serão enviadas para o barramento CAN através do controlador SJA1000. A descrição do software será realizada segundo os vários fluxogramas desenvolvidos, deixando a descrição relativa aos algoritmos específicos para o final.

2.1 - Geração das mensagens

Por forma a ser possível gerar os fluxos de mensagens necessários para as experimentações, foi implementada uma função de interrupção por *timer*, a qual é activada a cada 500 μ s. O fluxograma da Figura 64 representa esta função, usada como base em todos os algoritmos implementados.

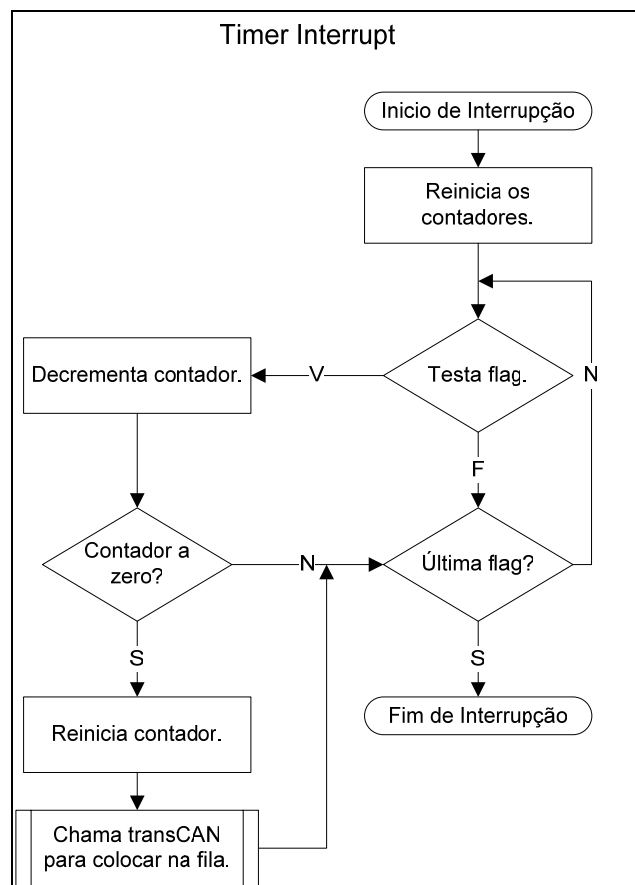


Figura 64 – Fluxograma da interrupção periódica por timer

Assim, a cada 500 μ s, a partir de um conjunto de flags de estado activadas pelo utilizador via porta série, é verificado quais os fluxos de mensagens que se pretendem activos ou não para gerar as mensagens. No caso de as flags estarem activadas, existe um contador (*timer*), o qual pretende simular o valor da periodicidade de cada fluxo, e que vai ser decrementado de cada vez que é activada a interrupção. Quando este atingir o valor zero, é efectuada a chamada à função `transCAN()` por forma a ser colocada a mensagem respectiva na fila circular. Isto mesmo pode ser observado no extracto da listagem seguinte (exemplo para dois fluxos de mensagens).

```

...
247 // Interrupção provocada pelo timer
248 void intTimer(void) interrupt 1 {
249
250     TR0 = 0;
251     TH0 = TC0INIC/256;
252     TL0 = TC0INIC;
253     TR0 = 1;
254     if (flag4) {
255         timer4--;
256         if (timer4 == 0) {
257             timer4 = TINIC4;

```

```
258         transCAN4();
259
260
261     }
262 }
263 if (flag3) {
264     timer3--;
265     if (timer3 == 0) {
266         timer3 = TINIC3;
267         transCAN3();
268
269
270     }
271 }
...
```

A função `transCAN()` começa por actualizar as variáveis estatísticas que serão enviadas mais tarde pela porta série para processamento. De seguida, é mascarado o endereço da fila (`apontadorIN`) onde vai ser colocada a mensagem (necessário para a fila ter um comportamento circular) e coloca-a nessa posição, actualizando por fim o valor da variável de estado `TRISTATE`. Isto mesmo pode ser observado pelo fluxograma da Figura 65 e pela listagem seguinte.

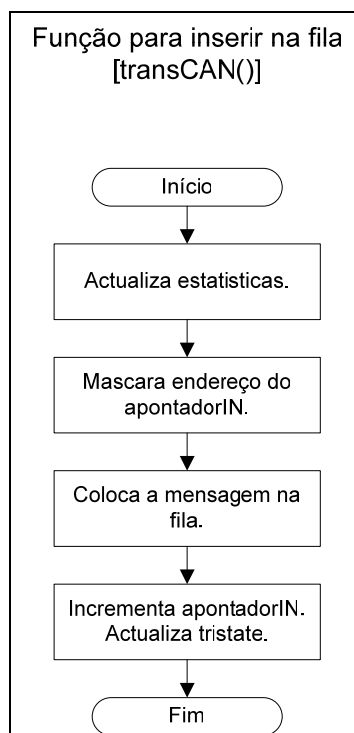


Figura 65 – Fluxograma da função `transCAN()` base

```
...
226 void transCAN4(void){
227 unsigned int aux;
228 nrMSG4++;          // Total das mensagens do fluxo
229 msgCount4++;       // Controlo de deadlines
230 if (msgCount4 > 1) lostDeadl1++; // Contagem das metas
231 // de deadlines perdidos
232 aux = apontadorIN + 3;
233 apontadorIN = aux & MASKEND;
234 while (*apontadorIN != 0){
235     aux = apontadorIN + 4;
236     apontadorIN = aux & MASKEND;
237 }
238 apontadorIN -=3 ;
239 *apontadorIN++ = m4.ID >> 3;
240 *apontadorIN++ = m4.ID;
241 *apontadorIN++ = m4.FRM;
242 *apontadorIN++ = m4.TRISTATE;
243 m4.FRM++;
244 }
...
```

2.2 - Controlo dos fluxos

Por forma a se poder ter um controlo efectivo de quais os fluxos que estão ou não activados, foi implementado na função `main()` uma condição que analisa alguns comandos enviados pela porta série, que vão activar/desactivar flags de controlo, as quais serão usadas na interrupção para identificar quais os fluxos que vão gerar as mensagens. O modo como isto foi implementado pode ser observado na listagem seguinte.

```
...
438 if (RI){
439     RI=0;
440     switch (SBUF){
441         case '1': flag1 = !flag1; break;
442         case '2': flag2 = !flag2; break;
443         case '3': flag3 = !flag3; break;
444         case '4': flag4 = !flag4; break;
445         case '5': flag1 = flag2 = flag3 = flag4 = 1; break;
446         case 'q': flag1 = flag2 = 1; break;
447         case 'w': flag1 = flag3 = 1; break;
448         case 'e': flag2 = flag3 = 1; break;
449         case 'r': flag1 = flag2 = flag3 = 1; break;
```

```

450         case 't': flag1 = flag2 = flag4 = 1; break;
451         case 'y': flag1 = flag3 = flag4 = 1; break;
452         case 'u': flag2 = flag3 = flag4 = 1; break;
453         case '0': {
454             flag1 = flag2 = flag3 = flag4 = 0;
455             timer1 = TINIC1;
456             timer2 = TINIC2;
457             timer3 = TINIC3;
458             timer4 = TINIC4;
459         } break;
...

```

2.3 - Contabilização das metas temporais perdidas

A contabilização das metas temporais perdidas é realizada com o recurso a contadores internos, designados: `msgCount` – indica a quantidade de mensagens presentes na fila de determinado fluxo de mensagens, que se for superior a uma mensagem é indicação de perda da meta temporal do fluxo de mensagens a que se refere; `lostDead1` – indica a quantidade de metas temporais perdidas (“*lost deadlines*”) de determinado fluxo de mensagens; e `nrMSG` – indica a quantidade total de mensagens geradas para determinado fluxo de mensagens pela interrupção.

Sempre que há uma chamada à função `transCAN()` (independentemente de qual o algoritmo implementado), estes valores referidos (também designados por variáveis estatísticas) vão ser actualizados da seguinte forma: incremento do valor `nrMSG`, por geração de uma nova mensagem para a fila; incremento do valor `msgCount`, por forma a verificar se já existem mais mensagens deste fluxo na fila; e incremento do valor `lostDead1`, se o valor de `msgCount` for superior a um, isto é, se na fila existirem pelo menos duas mensagens do mesmo fluxo de mensagens, é indicação de perda de meta temporal.

Isto pode ser observado na listagem seguinte.

```

...
228     nrMSG4++;          // Total das mensagens do fluxo
229     msgCount4++;       // Controlo de deadlines
230     if (msgCount4 > 1) lostDead1++; // Contagem das metas
231     // de deadlines perdidos
...

```

Contudo, para serem processados, estes dados tem que ser transmitidos para o computador. Isto é realizado a partir da função `posProces()`, a qual envia os valores dos contadores internos para a porta série. Por uma questão de facilidade da leitura dos dados, foi criada uma variável `dlCount` do tipo *string*, onde é colocada uma frase identificativa do fluxo a que se refere os dados, assim como os respectivos dados estatísticos. De seguida, esta variável é enviada ao computador via porta série, com

recurso à função `portaSerie()`. Isto pode ser observado no extracto da listagem seguinte.

```
...
370          // Passagem das estatísticas à posta série
371 void posProces(void) {
372
373     sprintf(dlCount, "ESCALONAMENTO 1 \n\r");
374     portaSerie(dlCount);
375     sprintf(dlCount, "maxFila = %u \n\r", maxFila);
376     portaSerie(dlCount);
377     sprintf(dlCount, "Fluxo                               %u:
378     portaSerie(dlCount);
379     sprintf(dlCount, "Fluxo                               %u:
380     portaSerie(dlCount);
381     sprintf(dlCount, "Fluxo                               %u:
382     portaSerie(dlCount);
383     sprintf(dlCount, "Fluxo                               %u:
384     portaSerie(dlCount);
385     portaSerie("\n\r");
386     }
...
```

É de referir que o modo como são contabilizadas as perdas das metas temporais é comum a todos os algoritmos implementados.

2.4 - Inicializações

De forma a facilitar tarefas de programação e tornar o software mais legível e transparente para qualquer programador, criaram-se diversas inicializações para valores constantes e que derivam de um mesmo valor, no início do software. Assim foi criado o endereço `BASE`, o qual tem como valor inicial `FC80h`, que corresponde à posição em que o microcontrolador `ADuC812` começa a escrever nos endereços do controlador `CAN SJA1000` [Phy, 2003]. Assim, a partir da tabela de registos do `SJA1000` [Phi, 98], basta somar ao valor `BASE` o valor corresponde ao endereço do registo do controlador que é necessário configurar. Por exemplo, para se configurar o registo de comandos (“*command register*”) faz-se `BASE+1`, tal como se pode ver na linha 20 no extracto seguinte da listagem.

```
...
19  #define BASE 0xFC80// SJA base address
20  #define CMR BASE+1// Command Register
21  #define STATUS BASE+2// Status Register
22  #define BUSTIM BASE+6// Bus Timing
23  #define TXBUF BASE+16// TxBuffer do SJA
24  #define CODMAS BASE+16// Code Mask/Acceptance Mask
25  #define CLKDIV BASE+31// Clock Divider
...
```

Também alguns valores correspondentes a bits específicos do controlador foram definidos no início do software como constante, assim como os valores das periodicidades de cada fluxo de mensagens como se pode ver no extracto da listagem seguinte.

```
...
26  #define TBS 0x04// Transmit buffer status bit
27  #define TCS 0x08// Transmit complete status bit
28  #define RM 0x01// Request Mode/Reset Request
29  #define TR 0x01// Transmission request bit
30  #define TS 0x20// Transmit status bit
31  #define AT 0x02// Abort transmission bit
...
```

Ainda enquadrado nas inicializações, temos duas funções essenciais e sem as quais nenhum dos algoritmos funciona. São elas `inituC()`, que configura e inicia o microcontrolador; e `initSJA()`, que configura e inicializa o controlador CAN SJA1000. Assim, quando arranca a função principal do programa, `main()`, antes de iniciar a parte correspondente ao próprio algoritmo, chama-se a função `inituC()` para, configurar os vários parâmetros de funcionamento do microcontrolador, tais como os *timers/counters* e respectivas interrupções para a contagem da periodicidade dos fluxos das mensagens, as prioridades de interrupção, a porta série por onde serão enviados os dados estatísticos. Este início pode ser observado no fluxograma da Figura 66 e listagem seguinte. Também se aproveitou esta função para limpar todos os dados existentes na fila circular criada.

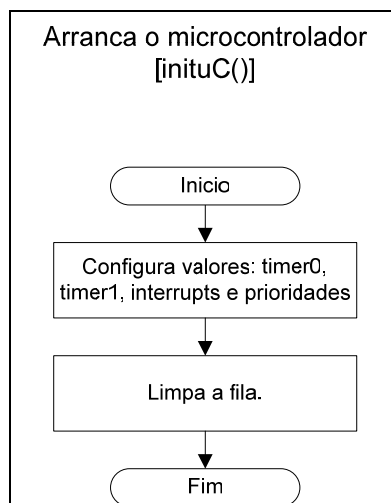


Figura 66 – Fluxograma da inicialização do microcontrolador

```
...
293 // Inicialização do uC
294 void inituC(void) {
295     unsigned int i;
296     TH0 = TC0INIC/256;
297     TL0 = TC0INIC;
298     TH1 = 255;
299     IE = 0x83;
300     IP = 0x11;
301     TCON = 0x00;
302     for (i=0;i<=LENGHT;i++) fila[i]=0x00;
303     P3_5 = 1;
304 }
...
```

Logo após se arrancar o microcontrolador, dá-se o arranque do controlador de barramento CAN. O fluxograma da mesma pode ser vista na Figura 67.

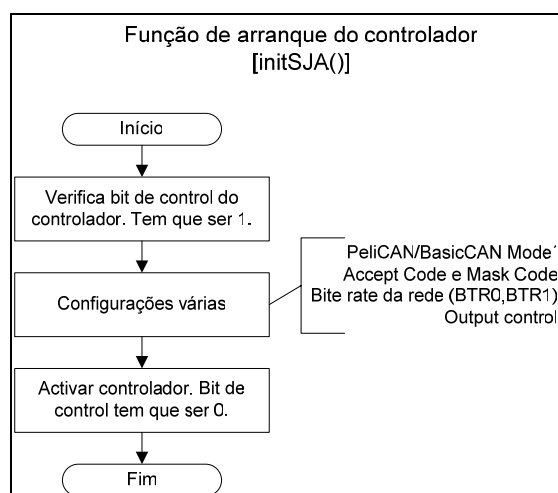


Figura 67 – Fluxograma da inicialização do controlador CAN SJA1000

Para o controlador SJA1000 “saber” que vai entrar em modo de configuração, obriga-se o software a entrar num ciclo infinito até o valor do bit de controlo ter o valor lógico 1. Assim que o controlador entrar no modo de configuração, pode-se configurar os vários parâmetros, tal como se indica na listagem do software. Após a configuração, no mesmo endereço do bit de controlo é colocado o valor lógico 0 e espera-se até que se leia esse mesmo valor.

Logo a seguir ao arranque do microcontrolador e do controlador CAN são efectuadas algumas configurações extra, como os valores de TCON e SCON de modo a poder usar a porta série do microcontrolador, os valores iniciais dos apontadores e das flags de controlo da geração de fluxos de mensagens. O facto do valor apontadorOUT ter um valor de arranque inicial inferior ao apontadorIN (apontadores de retirada e inserção na fila circular, respectivamente) deve-se a um atraso provocado pela configuração durante o programa, em que é iniciada logo a pesquisa da fila por mensagens para enviar para o controlador, mas a geração de fluxos de mensagens para colocar na fila surge mais tarde. Se não houver este ajuste de desfaseamento, a quantidade de metas temporais perdidas é superior.

2.5 - Sem escalonamento, filas FIFO em cada nó (CAN convencional)

Tal como o seu nome indica, neste software não foi considerado qualquer tipo de escalonamento, apenas se criou a fila circular e aplicou-se o princípio do CAN convencional, em que logo que surja uma mensagem de algum sensor (aqui simulado pela interrupção periódica) esta é logo colocada no controlador. Para o caso de este estar ocupado com alguma mensagem no seu buffer, estas são previamente colocadas na fila circular, de onde saem para o controlador apenas quando este está disponível.

Após serem efectuadas todas as inicializações, a função `main()` entra num ciclo infinito que efectuará logo de início uma verificação do valor `MSGLIM`, o qual representa a quantidade máxima de mensagens que são geradas com maior frequência e não pretendemos ultrapassar.

Quando o valor pré-definido por `MSGLIM` é atingido, o software dá a ordem de paragem à interrupção por timer, o que permite terminar a geração de fluxos de

mensagens. Após esta paragem, é chamada a função `posProces()` para enviar os dados estatísticos para a porta série.

Após a verificação da quantidade total de mensagens geradas, é realizada a pesquisa das mensagens na fila para serem colocadas no controlador CAN. Sempre que a pesquisa encontrar uma mensagem para colocar no controlador, efectua a chamada da função `enviar()`, a qual vai esperar que o controlador fique livre para lá colocar a mensagem. O modo como a mensagem é introduzida no controlador é idêntico ao modo como este é configurado, tendo em conta a ordem de entrada dos dados, a qual tem que ser a seguinte: DLC, primeira parte do identificador, segunda parte do identificador, dados a enviar no campo de dados. Isto é de acordo com o estipulado na folha de características do controlador [Phi, 98].

Antes de colocar a mensagem no controlador, o software actualiza as estatísticas, isto é, actualiza os valores das variáveis usadas para determinar se houve perda de metas temporais ou não. Isto pode ser observado no fluxograma da Figura 68.

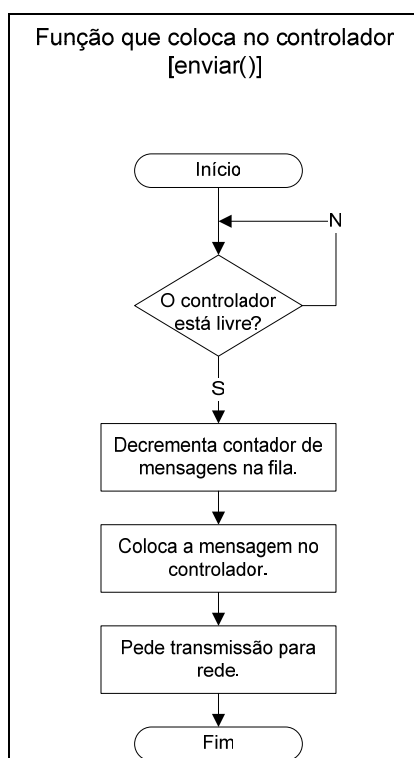


Figura 68 – Fluxograma da função de base para enviar para o controlador

No extracto de listagem seguinte apresenta-se como isto foi realizado.

```

...
68 // Função que coloca no buffer do SJA para enviar para a
rede
69 void enviar (void){
70     enderec = STATUS;
71     #ifdef placa
72     while(!(*enderec & TBS));

```

```
73     #endif
74     apontadorOUT -=3;
75     switch(*(apontadorOUT+1)){
76     #ifdef placa1
77         case 2: msgCount1--; break;
78         case 4: msgCount2--; break;
79         case 6: msgCount3--; break;
80         case 8: msgCount4--; break;
81     #endif
82     #ifdef placa2
83         case 1: msgCount1--; break;
84         case 3: msgCount2--; break;
85         case 5: msgCount3--; break;
86         case 7: msgCount4--; break;
87     #endif
88     }
89     enderec = TXBUF;
90     *enderec++ = DLC;
91     *enderec++ = *apontadorOUT++;
92     *enderec++ = *apontadorOUT++ << 5;
93     *enderec++ = *apontadorOUT++;
94     *apontadorOUT++ = 0x00; // variável TRISTATE
95     enderec = CMR;
96     *enderec++ = TR;
97
98     enderec = STATUS;
99     while ((*enderec & TBS)==0 | (*enderec & TCS)==0);
100 }
...
```

No fluxograma da Figura 69 podemos observar a função `main()` para este escalonamento.

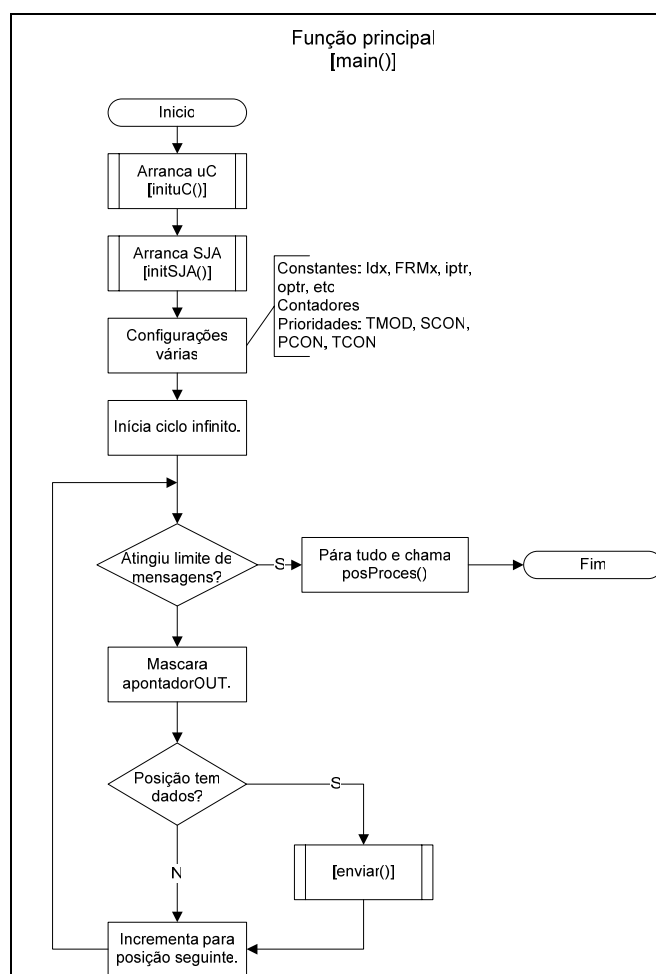


Figura 69 – Fluxograma da função main() para o primeiro algoritmo

2.6 - Escalonamento por prioridades local

Este algoritmo teve como base o algoritmo anterior. As principais diferenças situam-se nas funções main() e transCAN() e na interrupção por *timer*, para além da introdução da função abortar().

A função transCAN() tem como principal diferença o facto de após actualizar as estatísticas, acrescentar a um valor de *timeout* (endTimeout), que vai incrementando a cada 5 ms, o valor da periodicidade do fluxo e ser esse o valor da posição na fila onde vai ser colocada a mensagem. De modo a evitar sobreposições, é feita uma verificação para ver se nessa posição já existe uma mensagem. Caso exista, o apontador de inserção na fila é incrementado na distância correspondente a um bloco inteiro de 4 mensagens. Este funcionamento pode ser observado no fluxograma da Figura 70.

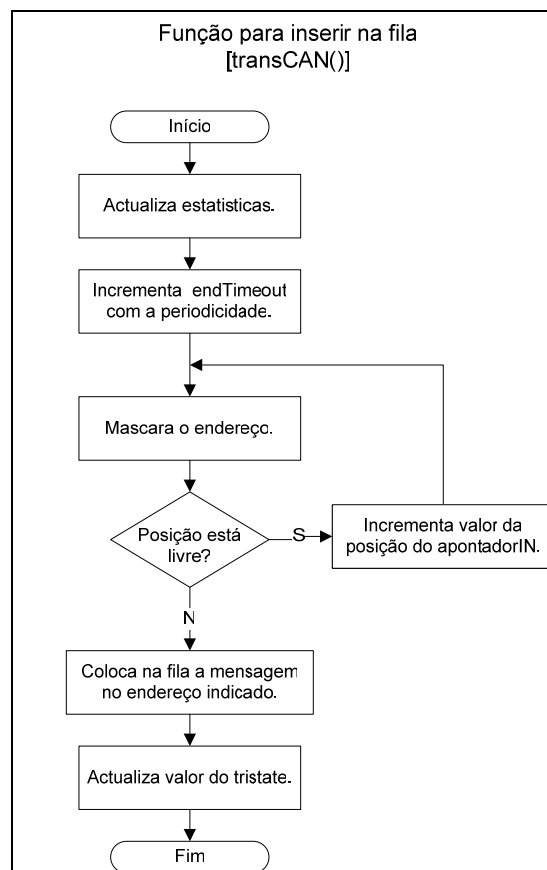


Figura 70 – Fluxograma da função transCAN(), que coloca na fila circular

Tal como no algoritmo anterior, é a interrupção por *timer* que faz a chamada à função transCAN(). A diferença existe no ponto em que agora há um apontador endTimeout, que vai ser incrementado de cada vez que o contador timeout atinge o valor 0, o que se sucede a cada 5 ms. O fluxograma representativo desta nova interrupção pode ser observado na Figura 71.

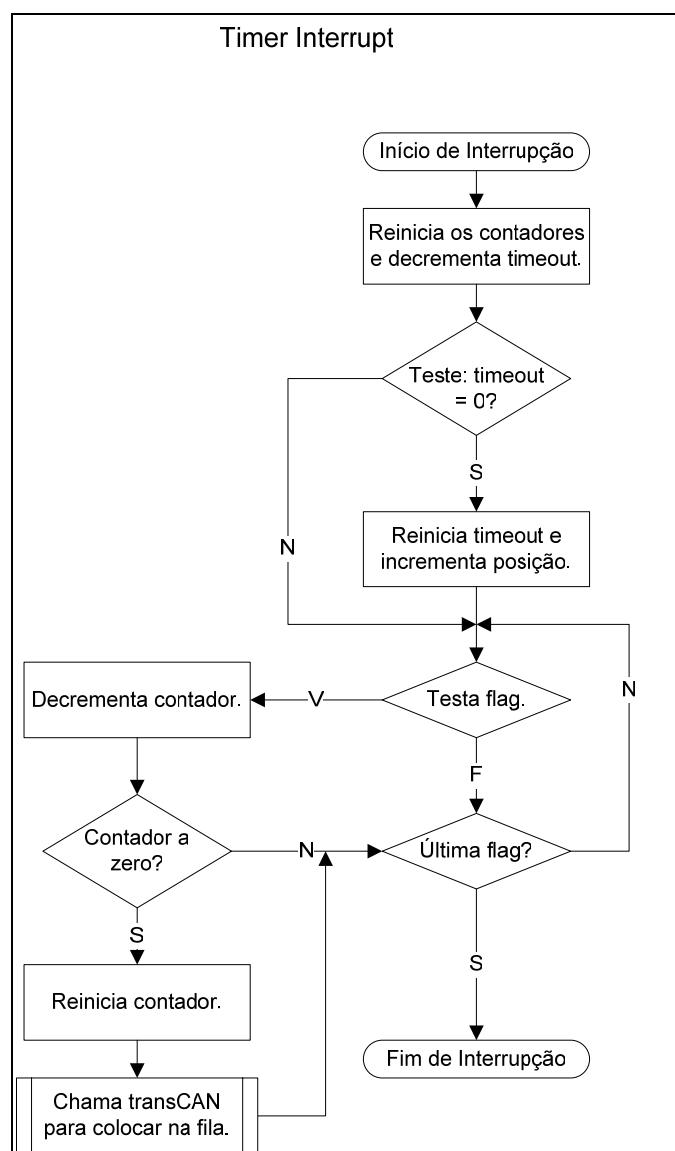


Figura 71 – Fluxograma da interrupção periódica com o valor de timeout

No que respeita à função `abortar()`, esta é chamada pela função principal (`main()`), sempre que na pesquisa por nova mensagem a variável de estado `TRISTATE` contenha o valor 2, o que significa que possivelmente o controlador ficou com aquela mensagem presa. Esta função verifica se realmente o controlador está ocupado (TBS bit) mas não está a transmitir para o barramento (TS Bit). Em caso positivo, é dado o comando ao controlador de abortar a mensagem presente. Após este comando, o programa espera que o controlador fique livre e verifica se entretanto a mensagem conseguiu vez no barramento e acabou por ser transmitida (TCS Bit). Nesse caso, considera-se falso alarme e é colocado um 0 na variável de estado `TRISTATE`. Na eventualidade da mensagem não ter sido transmitida, é colocado um 1 na variável de estado `TRISTATE` (equivale a voltar para a fila) e actualizadas as informações referentes às metas temporais perdidas. Este funcionamento pode ser verificado no fluxograma da Figura 72.

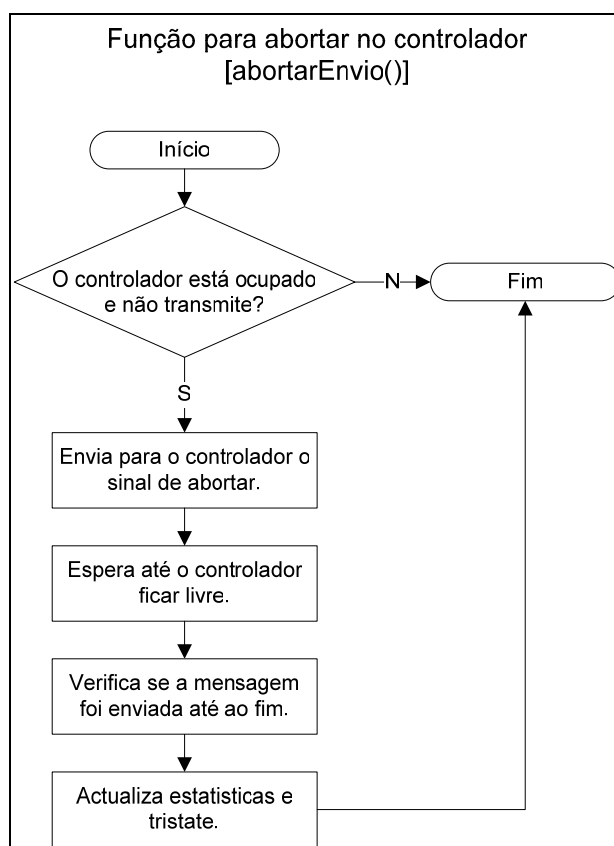


Figura 72 – Fluxograma da função abortar(), que aborta mensagens no controlador

A listagem seguinte mostra como foi implementada esta função:

```

...
147 // Função para abortar mensagens em processamento no SJA
148 void abortarEnvio(void){
149
150     enderec = STATUS;
151
152     #ifdef placa
153
154     if ((*enderec & TBS)==0x04 & (*enderec & TS)==0x00) {
155         enderec = CMR;
156         *enderec = AT; // abortar
157         enderec = STATUS;
158
159         while (!(*enderec & TBS)); // Para verificar se o buffer
160             // está livre. Pode não ter abortado.
161         if (*enderec & TCS){
162             *apontadorOUT = 0x00;
163         } else{
164             *apontadorOUT = 0x01;
165

```

```
166
167     switch(*apontadorOUT) {
168 #ifdef placa1
169     case 2: {if (msgCount1>0) lostDeadl1++; msgCount1++;}
break;
170     case 4: {if (msgCount2>0) lostDeadl2++; msgCount2++;}
break;
171     case 6: {if (msgCount3>0) lostDeadl3++; msgCount3++;}
break;
172     case 8: {if (msgCount4>0) lostDeadl4++; msgCount4++;}
break;
173 #endif
174 #ifdef placa2
175     case 1: {if (msgCount1>0) lostDeadl1++; msgCount1++;}
break;
176     case 3: {if (msgCount2>0) lostDeadl2++; msgCount2++;}
break;
177     case 5: {if (msgCount3>0) lostDeadl3++; msgCount3++;}
break;
178     case 7: {if (msgCount4>0) lostDeadl4++; msgCount4++;}
break;
179 #endif
180     }
181 }
182 }
183 #else
184     *apontadorOUT = 0x00;
185 #endif
186 }
...
```

Na função `main()`, repare-se que com a introdução da função `abortar()` e da variável de estado `TRISTATE`, o software passou a efectuar três testes, resultando numa pesquisa mais longa para determinar a existência das mensagens. Isto pode ser observado no fluxograma da Figura 73.

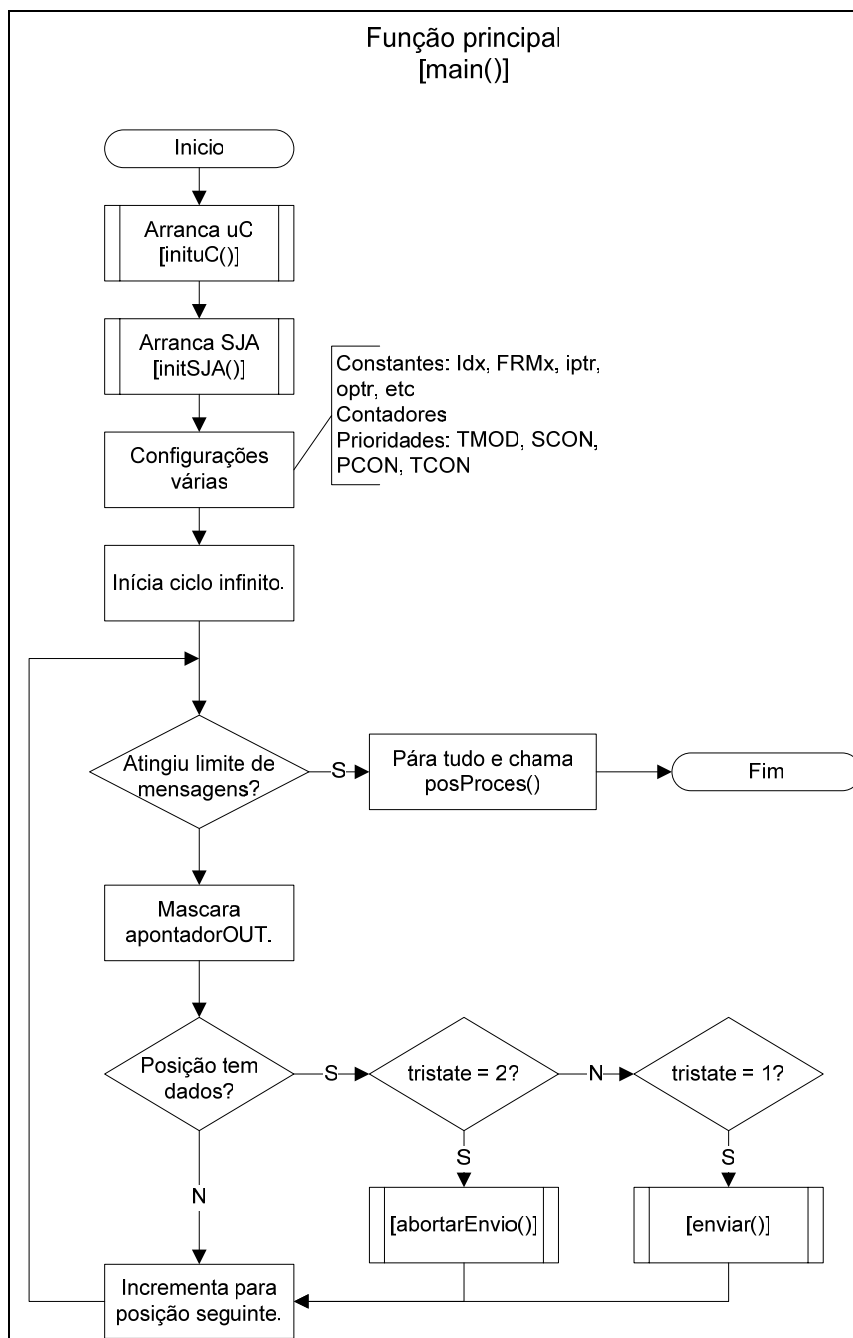


Figura 73 – Fluxograma da função main() usada para segundo algoritmo

2.7 - Escalonamento por prioridades local e épocas temporais

Para ser realizada a implementação deste algoritmo, tomou-se como base o algoritmo anterior (escalonamento local por prioridades), visto que a ideia de haver um escalonamento local por prioridades continua a imperar. As alterações maiores referem-se à inclusão de um cálculo extra, a partir da equação eq. 27 [ZS, 95]. Esta equação permite calcular épocas temporais, de forma a aumentar a eficiência no que se refere à transmissão de fluxos de mensagens menos prioritários, mas sem perder a eficiência de transmissão dos fluxos de mensagens mais prioritários.

Desta forma alteraram-se as funções `transCAN()` e `enviar()`, principalmente esta última que sofreu grandes modificações.

A função `transCAN()` sofreu alterações no instante em que coloca a mensagem na fila, em que anteriormente o identificador de cada mensagem era dividido em duas partes, isto é, como o identificador usado para a análise experimental é de 11 bits com uma determinada configuração, como pode ser visto na Figura 74, foi necessário guardar o valor do identificador de cada fluxo em duas posições da fila.

Endereço 17 do controlador SJA1000: Identificador 1 TX							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3
Endereço 18 do controlador SJA1000: Identificador 2 TX							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ID2	ID1	ID0	X	X	X	X	X

Figura 74 – Configuração do identificador de 11 bits numa mensagem no barramento CAN

Assim, eram guardados os últimos três bits do identificador de fluxo numa posição da fila, de forma que no momento de enviar para o controlador era só colocar na posição pretendida deste, sem modificações. No algoritmo do CAN convencional e no algoritmo do escalonamento por prioridades, este modo de actuar era bastante útil.

Mas com a introdução do cálculo por épocas, deixou de ser necessário fazer esta operação, pelo que a função `transCAN()` possui assim menos uma linha, a que era usada para guardar a segunda parte do identificador de fluxo. No entanto, o fluxograma da função mantém-se o mesmo do algoritmo de escalonamento local por prioridades.

A modificação principal que surge neste algoritmo está presente na função `enviar()`, onde são feitas as modificações necessárias de modo a introduzir o resultado do cálculo da época temporal no identificador do fluxo a enviar para o controlador. No seguinte excerto do software pode-se ver melhor esse aspecto.

```

...
rede 103 // Função que coloca no buffer do SJA para enviar para a
104 void enviar (void){
105     unsigned char fluxo; // Auxiliar para cálculos dos
endereços
106
107     enderec = STATUS;
108     #ifdef placa
109         while(!(*enderec & TBS));
110     #endif
111
112     apontadorOUT -=2;

```

```
113     switch(*(apontadorOUT)){
114 #ifdef placa1
115     case 2: msgCount1--; break;
116     case 4: msgCount2--; break;
117     case 6: msgCount3--; break;
118     case 8: msgCount4--; break;
119 #endif
120 #ifdef placa2
121     case 1: msgCount1--; break;
122     case 3: msgCount2--; break;
123     case 5: msgCount3--; break;
124     case 7: msgCount4--; break;
125 #endif
126     }
127     fluxo = *apontadorOUT >> 3; // Construção do ID
128     // Construção do ID (também pode ser >>4)
129     deadline = (apontadorOUT - endTimeout)/16;
130     if(deadline > 14) deadline = 255;
131     else deadline = deadline & DEADLIN;
132
133     enderec = TXBUF;
134     *enderec++ = DLC;
135     *enderec++ = (deadline << 2) + fluxo;
136
137     *enderec++ = *apontadorOUT++ << 5;
138     *enderec++ = *apontadorOUT++;
139     *apontadorOUT++ = 0x02; // variável de estado TRISSTATE
140     apontadorOUT++; // Tenho um bit a menos!
141
142     enderec = CMR;
143     *enderec++ = TR;
144     }
...
```

Esta mesma função está apresentada no fluxograma da Figura 75.

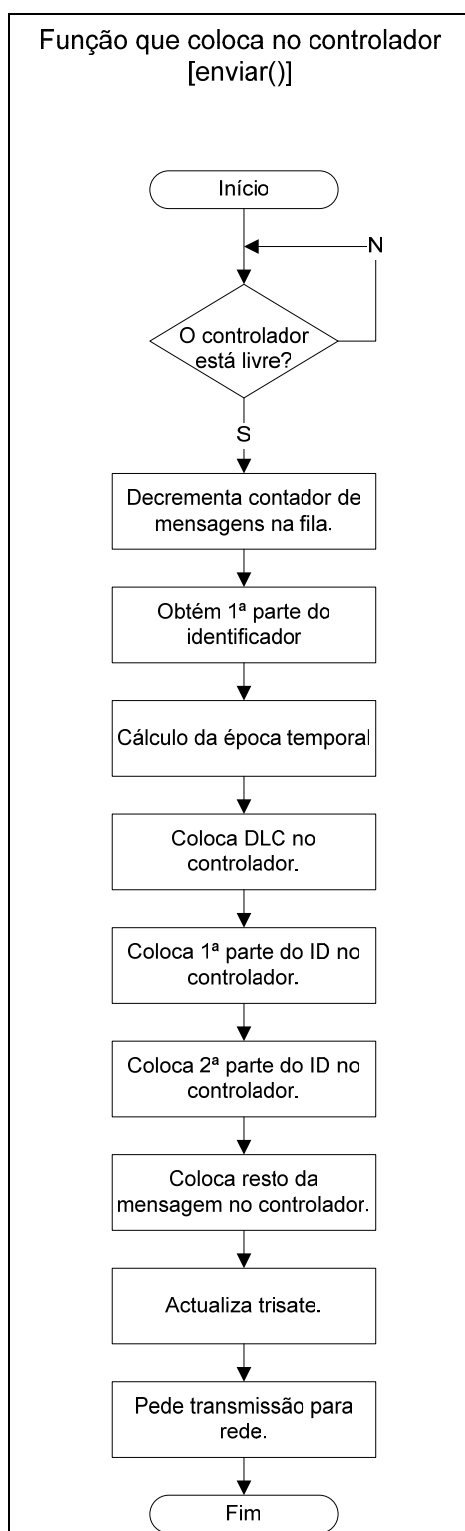


Figura 75 – Fluxograma da função enviar() com o cálculo por épocas

Devido a estas inclusões no software, optou-se por alterar o modo de pesquisa na fila, diminuindo em um elemento a incrementação para verificar a variável de estado TRISTATE, mas a incrementação relativa à quantidade de elementos da mensagem manteve-se o mesmo, de forma a simplificar o processo e manter os módulos de software o mais parecidos possível, verificando-se as alterações apenas no essencial.

3 - Listagens completas do software desenvolvido

De seguida, serão apresentadas as listagens completas do software desenvolvido para a experimentação, tal como este foi usado.

3.1 - Sem escalonamento, filas FIFO em cada nó (CAN convencional)

```
/* **** */
/*      Algoritmo 1 - Sem escalonamento, filas FIFO      */
/* **** */

#include <aduc812a.h>
#include <stdio.h>

#define placa // para testes somente de software, comentar esta linha
#define placa1 // endereços de fluxo da placa 1
// #define placa2 // endereços de fluxo da placa 2
#define TC0INIC 65536-(477*11.0592/12) // inicialização do timer T0
// para 500us

#define BUFINIC 0x5000 // posição inicial da fila na memória
#define NrElem 4 // Número de elementos das mensagens
#define NrFlux 4 // Número de fluxos
#define NrBloc 16 // Número de blocos totais pretendidos
#define LENGHT NrElem*NrFlux*NrBloc // tamanho da fila
#define MSGLIM 2000-1 // número limite de mensagens
#define DLC 0x01 // valor do Data Length Code
#define BASE 0xFC80 // SJA base address
#define CMR BASE+1 // Command Register
#define STATUS BASE+2 // Status Register
#define BUSTIM BASE+6 // Bus Timing
#define TXBUF BASE+16 // TxBuffer do SJA
#define CODMAS BASE+16 // Code Mask/Acceptance Mask
#define CLKDIV BASE+31 // Clock Divider
#define TBS 0x04 // Transmit buffer status bit
#define TCS 0x08 // Transmit complete status bit
#define RM 0x01 // Request Mode/Reset Request
#define TR 0x01 // Transmission request bit
#define TS 0x20 // Transmit status bit
#define AT 0x02 // Abort transmission bit
```

```

#define FILA LENGHT/4-1    // Máx de mensagens na fila em simultâneo

#define PERIOD1 30         // Periodicidade do fluxo 1 e 2
#define PERIOD2 35         // Periodicidade do fluxo 3 e 4
#define PERIOD3 69         // Periodicidade do fluxo 5 e 6
#define PERIOD4 104        // Periodicidade do fluxo 7 e 8
#define TINIC1 PERIOD1*2   // tempo inicial para transCAN1
#define TINIC2 PERIOD2*2   // tempo inicial para transCAN2
#define TINIC3 PERIOD3*2   // tempo inicial para transCAN3
#define TINIC4 PERIOD4*2   // tempo inicial para transCAN4
#define MASKEND 0xF0FF     // Máscara para os endereços

bit flag1, flag2, flag3, flag4;

// contadores para calculos de inserção na fila
unsigned int timer1, timer2, timer3, timer4;
// Quantidade de mensagens em cada fluxo
unsigned int nrMSG1, nrMSG2, nrMSG3, nrMSG4;
// contagem de lost deadlines
unsigned int lostDeadl1, lostDeadl2, lostDeadl3, lostDeadl4;
// contagem mensagens de entrada
unsigned char msgCount1, msgCount2, msgCount3, msgCount4;
// O que realmente coloca na porta série
unsigned char xdata dlCount[20];

unsigned char xdata fila[LENGHT] _at_ BUFINIC;

unsigned char xdata *enderec;           // endereço externo para onde
                                         // são enviados os dados
unsigned char xdata *apontadorIN;       // apontador de entrada na fila
unsigned char xdata *apontadorOUT;      // apontador de saída da fila

/* combinações de BTR0 e BTR1

    BTR0      BTR1      Result
    -----
    0x00      0x14      1 Mbit
    0xC4      0x3A      100 kbit
    0x27      0xDC      10 kbit      */

//bits de configuração pela seguinte ordem:
//4x AcceptCode, 4x MaskCode, BTR0, BTR1, OutputControl

```

```
unsigned char code init[]={0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF,
0xFF, 0x27, 0xDC, 0x4B};
```

```
// estrutura de um frame CAN, sem incluir DLC e FIM
struct MSG{
unsigned int ID;           // Identificador da mensagem
unsigned char FRM;        // Dados da mensagem
unsigned char TRISTATE;    // variável tristate do estado da mensagem
// (0=vazio;1=standby;2=proc)
} m1, m2, m3, m4;
```

```
// Função que coloca no buffer do SJA para enviar para a rede
```

```
void enviar (void){
    enderec = STATUS;
#ifdef placa
    while(!(*enderec & TBS));
#endif
    apontadorOUT -=3;
    switch(*(apontadorOUT+1)){
#ifdef placal
        case 2: msgCount1--; break;
        case 4: msgCount2--; break;
        case 6: msgCount3--; break;
        case 8: msgCount4--; break;
#endif
#ifdef placal2
        case 1: msgCount1--; break;
        case 3: msgCount2--; break;
        case 5: msgCount3--; break;
        case 7: msgCount4--; break;
#endif
    }
    enderec = TXBUF;
    *enderec++ = DLC;
    *enderec++ = *apontadorOUT++;
    *enderec++ = *apontadorOUT++ << 5;
    *enderec++ = *apontadorOUT++;
    *apontadorOUT++ = 0x00; // Bit tristate
    enderec = CMR;
    *enderec++ = TR;
    enderec = STATUS;
    while ((*enderec & TBS)==0 | (*enderec & TCS)==0);
```

```
}

// Função para abortar mensagens em processamento no SJA
void abortarEnvio(void){
    enderec = STATUS;
#ifdef placa
    if ((*enderec & TBS)==0x04 & (*enderec & TS)==0x00) {
        enderec = CMR;
        *enderec = AT; // abortar
        enderec = STATUS;
        while (!(*enderec & TBS)); // Para verificar se o buffer está
                                   // livre. Pode não ter abortado.

        if (*enderec & TCS){
            *apontadorOUT++ = 0x00;
        } else{
            *apontadorOUT++ = 0x01;
            switch(*(apontadorOUT-2)){
#ifdef placal
                case 2: {if (msgCount1>0) lostDeadl1++;msgCount1++;}
                        break;
                case 4: {if (msgCount2>0) lostDeadl2++;msgCount2++;}
                        break;
                case 6: {if (msgCount3>0) lostDeadl3++;msgCount3++;}
                        break;
                case 8: {if (msgCount4>0) lostDeadl4++;msgCount4++;}
                        break;
#ifdef placal2
                case 1: {if (msgCount1>0) lostDeadl1++;msgCount1++;}
                        break;
                case 3: {if (msgCount2>0) lostDeadl2++;msgCount2++;}
                        break;
                case 5: {if (msgCount3>0) lostDeadl3++;msgCount3++;}
                        break;
                case 7: {if (msgCount4>0) lostDeadl4++;msgCount4++;}
                        break;
#endif
            }
        }
    }
#endif
    *apontadorOUT++ = 0x00;
}
#endif
}
```

```
void transCAN1(void){

    unsigned int aux;
    nrMSG1++;          // Total das mensagens do fluxo
    msgCount1++;       // Controlo de deadlines
    if (msgCount1 > 1) lostDeadl1++; // Contagem de deadlines perdidos
    aux = apontadorIN + 3;
    apontadorIN = aux & MASKEND;
    while (*apontadorIN != 0){
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;
    *apontadorIN++ = m1.ID >> 3;
    *apontadorIN++ = m1.ID;
    *apontadorIN++ = m1.FRM;
    *apontadorIN++ = m1.TRISTATE;
    m1.FRM++;
}
```

```
void transCAN2(void){

    unsigned int aux;
    nrMSG2++;          // Total das mensagens do fluxo
    msgCount2++;       // Controlo de deadlines
    if (msgCount2 > 1) lostDeadl2++; // Contagem de deadlines perdidos
    aux = apontadorIN + 3;
    apontadorIN = aux & MASKEND;
    while (*apontadorIN != 0){
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;
    *apontadorIN++ = m2.ID >> 3;
    *apontadorIN++ = m2.ID;
    *apontadorIN++ = m2.FRM;
    *apontadorIN++ = m2.TRISTATE;
    m2.FRM++;
}
```

```
void transCAN3(void){
```

```

    unsigned int aux;
    nrMSG3++;           // Total das mensagens do fluxo
    msgCount3++;        // Controlo de deadlines
    if (msgCount3 > 1) lostDeadl3++; // Contagem de deadlines perdidos
    aux = apontadorIN + 3;
    apontadorIN = aux & MASKEND;
    while (*apontadorIN != 0){
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;
    *apontadorIN++ = m3.ID >> 3;
    *apontadorIN++ = m3.ID;
    *apontadorIN++ = m3.FRM;
    *apontadorIN++ = m3.TRISTATE;
    m3.FRM++;
}

```

```

void transCAN4(void){

```

```

    unsigned int aux;
    nrMSG4++;           // Total das mensagens do fluxo
    msgCount4++;        // Controlo de deadlines
    if (msgCount4 > 1) lostDeadl4++; // Contagem de deadlines perdidos
    aux = apontadorIN + 3;
    apontadorIN = aux & MASKEND;
    while (*apontadorIN != 0){
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;
    *apontadorIN++ = m4.ID >> 3;
    *apontadorIN++ = m4.ID;
    *apontadorIN++ = m4.FRM;
    *apontadorIN++ = m4.TRISTATE;
    m4.FRM++;
}

```

```

// Interrupção provocada pelo timer
void intTimer(void) interrupt 1 {

```

```

    TR0 = 0;

```

```
TH0 = TC0INIC/256;
TL0 = TC0INIC;
TR0 = 1;
if (flag4) {
    timer4--;
    if (timer4 == 0) {
        timer4 = TINIC4;
        transCAN4();
    }
}
if (flag3) {
    timer3--;
    if (timer3 == 0) {
        timer3 = TINIC3;
        transCAN3();
    }
}
if (flag2) {
    timer2--;
    if (timer2 == 0) {
        timer2 = TINIC2;
        transCAN2();
    }
}
if (flag1) {
    timer1--;
    if (timer1 == 0) {
        timer1 = TINIC1;
        transCAN1();
    }
}
}
```

```
// Inicialização do uC
void inituC(void) {
```

```
    unsigned int i;
    TH0 = TC0INIC/256;
    TL0 = TC0INIC;
    TH1 = 255;
    IE = 0x83;
    IP = 0x11;
    TCON = 0x00;
```

```
    for (i=0;i<=LENGHT;i++) fila[i]=0x00;
    P3_5 = 1;
}

// Inicialização do SJA
void initSJA(void){

    #ifdef placa
    unsigned int i;
    unsigned char ClockDivideReg;    // Clock Divider Register:
    // configuração PeliCAN/BasicCAN
    // Mode e bypass CAN input comp.
    enderec = BASE;
    do{                                // RM = Reset Mode Request Bit
        *enderec = RM;                // enquanto não for zero, não é possível
    }while(!(*enderec & RM)); // configurar o SJA
    ClockDivideReg=0xe8;    // PeliCAN Mode e bypass do CAN input comp.
                           // porque usa transceiver externo

    enderec = CLKDIV;;
    *enderec=ClockDivideReg;
    enderec = CODMAS;    // ver descrição do vector de configuração
    for(i=0;i<8;i++) *enderec++ = init[i];    // ver configuração do SJA
    enderec=BUSTIM;
    for(i=8;i<11;i++) *enderec++ = init[i];
    enderec = BASE;
    do{
        *enderec = 0x00;
    }while(*enderec);
    #endif
}

// Funções para pós-processamento
// envio para porta série
void portaSerie(char *ptr){
    while(*ptr){
        SBUF=*ptr++;
        while (!TI);
        TI = 0;
    }
}
```

```
// Passagem dos valores dos deadlines e quantidade de mensagens
void posProces(void){

    sprintf(dlCount,"ESCALONAMENTO 1 \n\r");
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",m1.ID,lostDeadl1,nrMSG1);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",m2.ID,lostDeadl2,nrMSG2);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",m3.ID,lostDeadl3,nrMSG3);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",m4.ID,lostDeadl4,nrMSG4);
    portaSerie(dlCount);
    portaSerie("\n\r");
}

// Função principal
void main(void){

    unsigned int aux;
    inituC();
    initSJA();
    flag1=flag2=flag3=flag4=1;
    timer1 = timer2 = timer3 = timer4 = 1;
    TMOD = 0x21;
    SCON = 0x50;
    PCON = PCON | 0x80;
    TCON = 0x50;
    apontadorIN = BUFINIC + 60;
    apontadorOUT = BUFINIC;
    m1.FRM = m2.FRM = m3.FRM = m4.FRM = 0;
    m1.TRISTATE = m2.TRISTATE = m3.TRISTATE = m4.TRISTATE = 1;
    //placa2
#ifdef placa2
    m1.ID=1; m2.ID=3; m3.ID=5; m4.ID=7;
#endif
    //placa1
#ifdef placa1
    m1.ID=2; m2.ID=4; m3.ID=6; m4.ID=8;
    P3_5 = 0;
#endif
    TR0 = 0;
    while(P3_5);
```

```
TR0 = 1;
while(1){
    if (nrMSG1>MSGLIM | nrMSG2>MSGLIM){
        flag1=flag2=flag3=flag4=0;
        TR0 = 0; P3_5 = 1;
        posProces();
        nrMSG1=nrMSG2=nrMSG3=nrMSG4=0;
    }
    aux = apontadorOUT;    // Necessário porque se está
                           // a trabalhar com apontadores
    apontadorOUT = aux & MASKEND;    // Máscara da fila (só
                                     // dá para 256 posições)

    if (*(apontadorOUT+1) != 0x00){
        apontadorOUT += 3;    // Incrementa para testar TRISTATE
        if (*apontadorOUT == 2) {
            abortarEnvio();
        }else if (*apontadorOUT == 1) {
            enviar();
        }else apontadorOUT++;    // Se entrar no if e incrementar 3,
                                // só falta 1 p/ msg seguinte
    } else apontadorOUT += 4;    // incrementa para verificar
                                // mensagem seguinte

    if (RI){
        RI=0;
        switch (SBUF){
            case '1': flag1 = !flag1; break;
            case '2': flag2 = !flag2; break;
            case '3': flag3 = !flag3; break;
            case '4': flag4 = !flag4; break;
            case '5': flag1 = flag2 = flag3 = flag4 = 1; break;
            case '0': {
                flag1 = flag2 = flag3 = flag4 = 0;
                timer1 = TINIC1;
                timer2 = TINIC2;
                timer3 = TINIC3;
                timer4 = TINIC4;
            } break;
            case 'p': {
                TR0 = 0;
                flag1=flag2=flag3=flag4=0;
                posProces();
                TR0 = 1;
            } break;
        }
    }
}
```

```
    }  
  }  
}
```

3.2 - Escalonamento por prioridades local

```
/* **** */  
/*      Algoritmo 2 - Escalonamento por prioridades local      */  
/* **** */  
  
#pragma db oe  
#include <aduc812a.h>  
#include <stdio.h>  
  
#define placa // para testes somente de software, comentar esta linha  
#define placal// endereços dos fluxos na placal  
//#define placa2 // endereços dos fluxos na placa2  
#define TC0INIC 65536-(477*11.0592/12) // inicialização do timer  
// T0 para 500us  
  
#define BUFINIC 0x5000// posição inicial da fila na memória  
#define NrFluxos 4 // quantidade de fluxos existente  
#define NrBlocos 16 // quantidade de blocos de mensagens pretendidos  
#define NrElemMSG 4 // quantidade de elementos definidos na estrutura  
#define LENGHT NrBlocos*NrElemMSG*NrFluxos // tamanho efectivo da fila  
#define MSGLIM 2000-1 // número limite de mensagens  
#define MASKEND 0xF0FF // Máscara para os endereços da fila  
#define DLC 0x01 // valor do Data Lenght Code  
#define BASE 0xFC80 // SJA base address  
#define CMR BASE+1 // Command Register  
#define STATUS BASE+2 // Status Register  
#define BUSTIM BASE+6 // Bus Timing  
#define TXBUF BASE+16 // TxBuffer do SJA  
#define CODMAS BASE+16 // Acceptance Code and Acceptance Mask  
#define CLKDIV BASE+31 // Clock Divider  
#define RXERRBASE+14 // Contador de erros de recepção  
#define TXERR BASE+15 // Contador de erros de transmissão  
#define TBS 0x04 // Transmit Buffer Status bit  
#define TCS 0x08 // Transmit Completed Status bit  
#define RM 0x01 // Reset Mode / Reset Resquest bit  
#define TR 0x01 // Transmission Request bit  
#define TS 0x20 // Transmit Status bit  
#define AT 0x02 // Abort Transmition bit
```

```

#define PERIOD1 30          // Periodicidade do fluxo 1 e 2
#define PERIOD2 35          // Periodicidade do fluxo 3 e 4
#define PERIOD3 69          // Periodicidade do fluxo 5 e 6
#define PERIOD4 104         // Periodicidade do fluxo 7 e 8
#define TINIC1 PERIOD1*2    // tempo inicial para transCAN1
#define TINIC2 PERIOD2*2    // tempo inicial para transCAN2
#define TINIC3 PERIOD3*2    // tempo inicial para transCAN3
#define TINIC4 PERIOD4*2    // tempo inicial para transCAN4
#define TIMEOUTINIC 10      // tempo para o timeout
#define TIMEOUTINCREM 4     // incremento de posições do timeout
#define AP1 PERIOD1/5*8-28;  // valor de di para o fluxo1
#define AP2 PERIOD2/5*8-28;  // valor de di para o fluxo2
#define AP3 PERIOD3/5*8-28;  // valor de di para o fluxo3
#define AP4 PERIOD4/5*8-28;  // valor de di para o fluxo4

bit flag1, flag2, flag3, flag4;

// contadores para calculos de inserção na fila
unsigned int timer1, timer2, timer3, timer4, timeout;
// Quantidade de mensagens em cada fluxo
unsigned int nrMSG1, nrMSG2, nrMSG3, nrMSG4;
// contagem de lost deadlines
unsigned int lostDeadl1, lostDeadl2, lostDeadl3, lostDeadl4;
// contagem mensagens de entrada
unsigned char msgCount1, msgCount2, msgCount3, msgCount4;
// O que realmente coloca na porta série
unsigned char xdata dlCount[20];
// Apontadores auxiliares para o cálculo das posições na fila
unsigned char data lsbTimeout, lsbAptIn, lsbAptOut;
unsigned char xdata fila [LENGHT] _at_ BUFINIC;
// endereço externo para onde são enviados os dados
unsigned char xdata *enderec;
unsigned char xdata *endTimeout; // endereço do apontador de Timeout
unsigned char xdata *apontadorIN; // apontador para colocação na fila
unsigned char xdata *apontadorOUT; // apontador para retirada da fila

/* combinações de BTR0 e BTR1

    BTR0      BTR1      Result
    -----
    0x00      0x14      1 Mbit
    0xC4      0x3A      100 kbit
    0x27      0xDC      10 kbit      */

```

```
//bits de configuração pela seguinte ordem:
//4x AcceptCode, 4x MaskCode, BTR0, BTR1, OutputControl
unsigned char code init[]={0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF,
0xFF, 0x27, 0xDC, 0x4B};

// estrutura de um frame CAN, sem incluir DLC e FIM
typedef struct MSG{
unsigned int ID;           // Identificador da mensagem
unsigned char FRM;        // Dados da mensagem
unsigned char TRISTATE;    // Bit tristate da mensagem
// (0=vazio;1=standby;2=proc)
} MENSAGEM;

// estrutura de 1 bloco da fila de espera com as 4 mensagens
struct bloco{
    MENSAGEM m1;
    MENSAGEM m2;
    MENSAGEM m3;
    MENSAGEM m4;
}bloco;

// Função que coloca no buffer do SJA para enviar para a rede
void enviar (void){

    enderec = STATUS;
#ifdef placa
    while(!(*enderec & TBS));
#endif
    apontadorOUT -=3;
    switch(*(apontadorOUT+1)){
#ifdef placal
        case 2: msgCount1--; break;
        case 4: msgCount2--; break;
        case 6: msgCount3--; break;
        case 8: msgCount4--; break;
#endif
#ifdef placa2
        case 1: msgCount1--; break;
        case 3: msgCount2--; break;
        case 5: msgCount3--; break;

```

```
        case 7: msgCount4--; break;
#endif
    }
    enderec = TXBUF;
    *enderec++ = DLC;
    *enderec++ = *apontadorOUT++;
    *enderec++ = *apontadorOUT++ << 5;
    *enderec++ = *apontadorOUT++;
    *apontadorOUT++ = 0x02;          // variável de estado tristate
    enderec = CMR;
    *enderec++ = TR;
}

// Função para abortar mensagens em processamento no SJA
void abortarEnvio(void){

    enderec = STATUS;
#ifdef placa
    if ((*enderec & TBS)==0x04 & (*enderec & TS)==0x00) {
        enderec = CMR;
        *enderec = AT;          // abortar
        enderec = STATUS;
        while (!(*enderec & TBS));    // Para verificar se o buffer está
                                    // livre. Pode não ter abortado.

        if (*enderec & TCS){
            *apontadorOUT = 0x00;
        } else{
            *apontadorOUT = 0x01;
            switch(*apontadorOUT-2){
#ifdef placa1
                case 2: {if (msgCount1>0) lostDeadl1++; msgCount1++;} break;
                case 4: {if (msgCount2>0) lostDeadl2++; msgCount2++;} break;
                case 6: {if (msgCount3>0) lostDeadl3++; msgCount3++;} break;
                case 8: {if (msgCount4>0) lostDeadl4++; msgCount4++;} break;
            #endif
#ifdef placa2
                case 1: {if (msgCount1>0) lostDeadl1++; msgCount1++;} break;
                case 3: {if (msgCount2>0) lostDeadl2++; msgCount2++;} break;
                case 5: {if (msgCount3>0) lostDeadl3++; msgCount3++;} break;
                case 7: {if (msgCount4>0) lostDeadl4++; msgCount4++;} break;
            #endif
            }
        }
    }
}
```

```

    }
#else
    *apontadorOUT = 0x00;
#endif
}

// Funções que colocam na fila os dados
void transCAN1(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG1++;          // conta mensagens na entrada da fila
    msgCount1++;       // controlo de deadlines das mensagens
    if (msgCount1 > 1) lostDeadl1++;    // Lost Deadlines
    lsbAptIn += AP1;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;
    apontadorIN += 3;
    while (*apontadorIN != 0x00){      // Assim garanto que não sobrepõe
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;
    *apontadorIN++ = bloco.m1.ID >> 3;
    *apontadorIN++ = bloco.m1.ID;
    *apontadorIN++ = bloco.m1.FRM;
    *apontadorIN++ = bloco.m1.TRISTATE;
    bloco.m1.FRM++;
}

void transCAN2(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG2++;          // conta mensagens na entrada da fila
    msgCount2++;       // controlo de deadlines das mensagens
    if (msgCount2 > 1) lostDeadl1++;    // Lost Deadlines
    lsbAptIn += AP2;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;
    apontadorIN += 3;
    while (*apontadorIN != 0x00){      // Assim garanto que não sobrepõe

```

```
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;
    *apontadorIN++ = bloco.m2.ID >> 3;
    *apontadorIN++ = bloco.m2.ID;
    *apontadorIN++ = bloco.m2.FRM;
    *apontadorIN++ = bloco.m2.TRISTATE;
    bloco.m2.FRM++;
}

void transCAN3(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG3++;          // conta mensagens na entrada da fila
    msgCount3++;       // controlo de deadlines das mensagens
    if (msgCount3 > 1) lostDeadll++;    // Lost Deadlines
    lsbAptIn += AP3;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;
    apontadorIN += 3;
    while (*apontadorIN != 0x00){      // Assim garanto que não sobrepõe
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 3;
    *apontadorIN++ = bloco.m3.ID >> 3;
    *apontadorIN++ = bloco.m3.ID;
    *apontadorIN++ = bloco.m3.FRM;
    *apontadorIN++ = bloco.m3.TRISTATE;
    bloco.m3.FRM++;
}

void transCAN4(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG4++;          // conta mensagens na entrada da fila
    msgCount4++;       // controlo de deadlines das mensagens
    if (msgCount4 > 1) lostDeadll++;    // Lost Deadlines
    lsbAptIn += AP4;
    aux = endTimeout + lsbAptIn;
```

```
apontadorIN = aux & MASKEND;
apontadorIN += 3;
while (*apontadorIN != 0x00){    // Assim garanto que não sobrepe
    aux = apontadorIN + 4;
    apontadorIN = aux & MASKEND;
}
apontadorIN -= 3;
*apontadorIN++ = bloco.m4.ID >> 3;
*apontadorIN++ = bloco.m4.ID;
*apontadorIN++ = bloco.m4.FRM;
*apontadorIN++ = bloco.m4.TRISTATE;
bloco.m4.FRM++;
}
```

```
// Interrupção provocada pelo timer
void intTimer(void) interrupt 1 {

    TR0 = 0;
    TH0 = TC0INIC/256;
    TL0 = TC0INIC;
    TR0 = 1;
    timeout--;
    if (0==timeout) {
        timeout = 10;
        lsbTimeout += TIMEOUTINCREM; // teste de fim da fila
        endTimeout = BUFINIC + lsbTimeout;
    }
    if (flag4) {
        timer4--;
        if (timer4 == 0) {
            timer4 = TINIC4;
            transCAN4();
        }
    }
    if (flag3) {
        timer3--;
        if (timer3 == 0) {
            timer3 = TINIC3;
            transCAN3();
        }
    }
    if (flag2) {
        timer2--;
```

```
        if (timer2 == 0) {
            timer2 = TINIC2;
            transCAN2();
        }
    }
    if (flag1) {
        timer1--;
        if (timer1 == 0) {
            timer1 = TINIC1;
            transCAN1();
        }
    }
}

// Inicialização do uC
void inituC(void) {

    unsigned int i;
    TH0 = TC0INIC/256;
    TL0 = TC0INIC;
    TH1 = 255;
    IE = 0x83;
    IP = 0x11;
    TCON = 0x00;
    for (i=0;i<=LENGHT;i++) fila[i]=0x00;
    P3_5 = 1;
}

// Inicialização do SJA
void initSJA(void){

#ifdef placa
    unsigned int i;
    unsigned char ClockDivideReg;// Clock Divider Register:
                                // configuração PeliCAN/BasicCAN
                                // Mode e bypass CAN input comp.

    enderec = BASE;
    do{        // RM = Reset Mode Request Bit
        *enderec = RM;// enquanto não for zero, não é
        // possível configurar o SJA
    }while(!(*enderec & RM));
    ClockDivideReg=0xe8;// PeliCAN Mode e bypass do CAN input comp.
```

```
// porque usa transceiver externo
enderec = CLKDIV;;
*enderec=ClockDivideReg;
enderec = CODMAS;          // ver descrição do vector de configuração
for(i=0;i<8;i++) *enderec++ = init[i];  // ver configuração do SJA
enderec=BUSTIM;
for(i=8;i<11;i++) *enderec++ = init[i];
enderec = BASE;
do{
    *enderec = 0x00;
}while(*enderec);
#endif
}

// Funções para pós-processamento
// envio para porta série, idêntico à mensagem do LCD
void portaSerie(char *ptr){
    while(*ptr){
        SBUF=*ptr++;
        while (!TI);
        TI = 0;
    }
}

// Passagem dos valores dos deadlines e quantidade de mensagens
void posProces(void){

    sprintf(dlCount,"ESCALONAMENTO 2 \n\r");
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo                                     %u:
%u/%u\n\r",bloco.m1.ID,lostDeadl1,nrMSG1);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo                                     %u:
%u/%u\n\r",bloco.m2.ID,lostDeadl2,nrMSG2);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo                                     %u:
%u/%u\n\r",bloco.m3.ID,lostDeadl3,nrMSG3);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo                                     %u:
%u/%u\n\r",bloco.m4.ID,lostDeadl4,nrMSG4);
    portaSerie(dlCount);
    portaSerie("\n\r");
}
```

```
// Função principal
void main(void){

    unsigned int aux;
    inituC();
    initSJA();
    flag1 = flag2 = flag3 = flag4 = 1;
    timer1 = timer2 = timer3 = timer4 = 1;
    endTimeout = BUFINIC;lsbTimeout=0;
    timeout = TIMEOUTINIC;
    apontadorOUT = endTimeout;
    TMOD = 0x21;
    SCON = 0x50;
    PCON = PCON | 0x80;
    TCON = 0x50;
    bloco.m1.FRM=bloco.m2.FRM=bloco.m3.FRM=bloco.m4.FRM=0x00;
    bloco.m1.TRISTATE=bloco.m2.TRISTATE=0x01;
    bloco.m3.TRISTATE=bloco.m4.TRISTATE=0x01;
//placa2
#ifdef placa2
    bloco.m1.ID=1; bloco.m2.ID=3; bloco.m3.ID=5; bloco.m4.ID=7;
#endif
//placa1
#ifdef placa1
    bloco.m1.ID=2; bloco.m2.ID=4; bloco.m3.ID=6; bloco.m4.ID=8;
    P3_5 = 0;
#endif
    TR0 = 0;
    while(P3_5);
    TR0 = 1;
    while(1){
        if ((nrMSG1>MSGLIM) | (nrMSG2>MSGLIM)){
            flag1=flag2=flag3=flag4=0;
            TR0 = 0; P3_5 = 1;
            posProces();
            nrMSG1=nrMSG2=nrMSG3=nrMSG4=0;
        }
        aux = endTimeout + lsbAptOut;
        apontadorOUT = aux & MASKEND;
        if (*(apontadorOUT+1) != 0x00){
            apontadorOUT += 3;
            if (*apontadorOUT == 2) {
```

```
        abortarEnvio();
        lsbAptOut = 0;
    }else if (*apontadorOUT == 1) {
        enviar();
        lsbAptOut = 0;
    }else lsbAptOut += 4;
} else lsbAptOut += 4;
if (RI){
    RI=0;
    switch (SBUF){
        case '1': flag1 = !flag1; break;
        case '2': flag2 = !flag2; break;
        case '3': flag3 = !flag3; break;
        case '4': flag4 = !flag4; break;
        case '5': flag1 = flag2 = flag3 = flag4 = 1; break;
        case '0': {
            flag1 = flag2 = flag3 = flag4 = 0;
            timer1 = TINIC1;
            timer2 = TINIC2;
            timer3 = TINIC3;
            timer4 = TINIC4;
        } break;
        case 'p': {
            TR0 = 0;
            flag1=flag2=flag3=flag4=0;
            posProces();
            TR0 = 1;
        } break;
    }
}
}
```

3.3 - Escalonamento por prioridades local e cálculo de épocas

```
/* **** */
/*      Algoritmo 3 - Escalonamento por prioridades      */
/*      local e épocas temporais                          */
/* **** */

#pragma db oe
#include <aduc812a.h>
```

```
#include <stdio.h>

#define placa // para testes somente de software, comentar esta linha
#define placa1 // endereços dos fluxos na placa1
// #define placa2 // endereços dos fluxos na placa2
#define TC0INIC 65536-(477*11.0592/12) // inicialização do timer T0
// para 500us

#define BUFINIC 0x5000 // posição inicial da fila na memória
#define NrFluxos 4 // quantidade de fluxos existente
#define NrBlocos 16 // quantidade de blocos de mensagens pretendidos
#define NrElemMSG 4 // quantidade de elementos definidos na estrutura
#define LENGHT NrBlocos*NrElemMSG*NrFluxos // tamanho efectivo da fila
#define MSGLIM 2000-1 // número limite de mensagens
#define MASKEND 0xF0FF // Máscara para dar a volta na fila
#define DEADLIN 0x1F // Máscara para cálculo do deadline
#define DLC 0x01 // valor do Data Lenght Code
#define BASE 0xFC80 // SJA base address
#define CMR BASE+1 // Command Register
#define STATUS BASE+2 // Status Register
#define BUSTIM BASE+6 // Bus Timing
#define TXBUF BASE+16 // TxBuffer do SJA
#define CODMAS BASE+16 // Acceptance Code and Acceptance Mask
#define CLKDIV BASE+31 // Clock Divider
#define TBS 0x04 // Transmit Buffer Status bit
#define TCS 0x08 // Transmit Completed Status bit
#define RM 0x01 // Reset Mode / Reset Resquest bit
#define TR 0x01 // Transmission Request bit
#define TS 0x20 // Transmit Status bit
#define AT 0x02 // Abort Transmition bit

#define PERIOD1 30 // Periodicidade do fluxo 1 e 2
#define PERIOD2 35 // Periodicidade do fluxo 3 e 4
#define PERIOD3 69 // Periodicidade do fluxo 5 e 6
#define PERIOD4 104 // Periodicidade do fluxo 7 e 8
#define TINIC1 PERIOD1*2 // tempo inicial para transCAN1
#define TINIC2 PERIOD2*2 // tempo inicial para transCAN2
#define TINIC3 PERIOD3*2 // tempo inicial para transCAN3
#define TINIC4 PERIOD4*2 // tempo inicial para transCAN4
#define TIMEOUTINIC 10 // tempo para o timeout
#define TIMEOUTINCREM 4 // incremento de posições do timeout
#define AP1 PERIOD1/5*8-28; // valor de di para o fluxo1
#define AP2 PERIOD2/5*8-28; // valor de di para o fluxo2
#define AP3 PERIOD3/5*8-28; // valor de di para o fluxo3
#define AP4 PERIOD4/5*8-28; // valor de di para o fluxo4
```

```
bit flag1, flag2, flag3, flag4;

unsigned int timer1, timer2, timer3, timer4, timeout;
// Quantidade de mensagens em cada fluxo
unsigned int nrMSG1, nrMSG2, nrMSG3, nrMSG4;
// contagem de lost deadlines
unsigned int lostDeadl1, lostDeadl2, lostDeadl3, lostDeadl4;
// contagem mensagens de entrada
unsigned char msgCount1, msgCount2, msgCount3, msgCount4;
// O que realmente coloca na porta série
unsigned char xdata dlCount[20];
unsigned char data lsbTimeout, lsbAptIn, lsbAptOut, deadline;
unsigned char xdata fila [LENGHT] _at_ BUFINIC;
// endereço externo para onde são enviados os dados
unsigned char xdata *enderec;
unsigned char xdata *endTimeout; // endereço do apontador de Timeout
unsigned char xdata *apontadorIN; // apontador para colocação na fila
unsigned char xdata *apontadorOUT; // apontador para retirada da fila

/* combinações de BTR0 e BTR1

    BTR0      BTR1      Result
    -----
    0x00      0x14      1 Mbit
    0xC4      0x3A      100 kbit
    0x27      0xDC      10 kbit      */

//bits de configuração pela seguinte ordem:
//4x AcceptCode, 4x MaskCode, BTR0, BTR1, OutputControl
unsigned char code init[]={0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF,
0xFF, 0x27, 0xDC, 0x4B};

// estrutura de um frame CAN, sem incluir DLC e FIM
typedef struct MSG{
unsigned int ID;           // Identificador da mensagem
unsigned char FRM;        // Dados da mensagem
unsigned char TRISTATE;    // Bit tristate da mensagem
// (0=vazio;1=standby;2=proc)
} MENSAGEM;

// estrutura de 1 bloco da fila de espera com as 4 mensagens
```

```
struct bloco{
    MENSAGEM m1;
    MENSAGEM m2;
    MENSAGEM m3;
    MENSAGEM m4;
}bloco;

// Função que coloca no buffer do SJA para enviar para a rede
void enviar (void){

    unsigned char fluxo;// Auxiliar para cálculos dos endereços
    enderec = STATUS;
#ifdef placa
    while(!(*enderec & TBS));
#endif
    apontadorOUT -= 2;
    switch(*apontadorOUT){
#ifdef placal
        case 2: msgCount1--; break;
        case 4: msgCount2--; break;
        case 6: msgCount3--; break;
        case 8: msgCount4--; break;
#endif
#ifdef placa2
        case 1: msgCount1--; break;
        case 3: msgCount2--; break;
        case 5: msgCount3--; break;
        case 7: msgCount4--; break;
#endif
    }

    fluxo = *apontadorOUT >> 3;          // Obtenção da 1ª parte do ID
    // Cálculo da época temporal, segundo a equação eq. 27 [ZS, 95]
    deadline = (apontadorOUT - endTimeout)/16;
    if(deadline > 14) deadline = 255;
    else deadline = deadline & DEADLIN;

    enderec = TXBUF;
    *enderec++ = DLC;
    *enderec++ = (deadline << 2) + fluxo;
    *enderec++ = *apontadorOUT++ << 5;
    *enderec++ = *apontadorOUT++;
    *apontadorOUT++ = 0x02; // Bit TRISTATE
```

```

    apontadorOUT++;
    enderec = CMR;
    *enderec++ = TR;
}

// Função para abortar mensagens em processamento no SJA
void abortarEnvio(void){

    enderec = STATUS;
#ifdef placa
    if ((*enderec & TBS)==0x04 & (*enderec & TS)==0x00) {
        enderec = CMR;
        *enderec = AT;          // abortar
        enderec = STATUS;
        while (!(*enderec & TBS));    // Para verificar se o buffer está
        // livre. Pode não ter abortado.
        if (*enderec & TCS){
            *apontadorOUT = 0x00;
        } else{
            *apontadorOUT = 0x01;
            switch(*(apontadorOUT-2)){
#ifdef placal
                case 2: {if (msgCount1>0) lostDeadl1++; msgCount1++;} break;
                case 4: {if (msgCount2>0) lostDeadl2++; msgCount2++;} break;
                case 6: {if (msgCount3>0) lostDeadl3++; msgCount3++;} break;
                case 8: {if (msgCount4>0) lostDeadl4++; msgCount4++;} break;
            #endif
#ifdef placa2
                case 1: {if (msgCount1>0) lostDeadl1++; msgCount1++;} break;
                case 3: {if (msgCount2>0) lostDeadl2++; msgCount2++;} break;
                case 5: {if (msgCount3>0) lostDeadl3++; msgCount3++;} break;
                case 7: {if (msgCount4>0) lostDeadl4++; msgCount4++;} break;
            #endif
            }
        }
    }
#else
    *apontadorOUT = 0x00;
#endif
}

// Funções que colocam na fila os dados

```

```
void transCAN1(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG1++;          // conta mensagens na entrada da fila
    msgCount1++;       // controlo de deadlines das mensagens
    if (msgCount1 > 1) lostDeadl1++;    // Lost Deadlines
    lsbAptIn += AP1;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;
    apontadorIN += 2;
    while (*apontadorIN != 0x00){      // Assim garanto que não sobrepõe
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 2;
    *apontadorIN++ = bloco.m1.ID;
    *apontadorIN++ = bloco.m1.FRM;
    *apontadorIN++ = bloco.m1.TRISTATE;
    apontadorIN++;
    bloco.m1.FRM++;
}
```

```
void transCAN2(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG2++;          // conta mensagens na entrada da fila
    msgCount2++;       // controlo de deadlines das mensagens
    if (msgCount2 > 1) lostDeadl1++;    // Lost Deadlines
    lsbAptIn += AP2;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;
    apontadorIN += 2;
    while (*apontadorIN != 0x00){      // Assim garanto que não sobrepõe
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 2;
    *apontadorIN++ = bloco.m2.ID;
    *apontadorIN++ = bloco.m2.FRM;
    *apontadorIN++ = bloco.m2.TRISTATE;
    apontadorIN++;
}
```

```

        bloco.m2.FRM++;
    }

void transCAN3(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG3++;          // conta mensagens na entrada da fila
    msgCount3++;       // controlo de deadlines das mensagens
    if (msgCount3 > 1) lostDeadl1++;    // Lost Deadlines
    lsbAptIn += AP3;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;
    apontadorIN += 2;
    while (*apontadorIN != 0x00){        // Assim garanto que não sobrepe
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 2;
    *apontadorIN++ = bloco.m3.ID;
    *apontadorIN++ = bloco.m3.FRM;
    *apontadorIN++ = bloco.m3.TRISTATE;
    apontadorIN++;
    bloco.m3.FRM++;
}

void transCAN4(void) {

    unsigned int aux;
    lsbAptIn = 0;
    nrMSG4++;          // conta mensagens na entrada da fila
    msgCount4++;       // controlo de deadlines das mensagens
    if (msgCount4 > 1) lostDeadl1++;    // Lost Deadlines
    lsbAptIn += AP4;
    aux = endTimeout + lsbAptIn;
    apontadorIN = aux & MASKEND;
    apontadorIN += 2;
    while (*apontadorIN != 0x00){        // Assim garanto que não sobrepe
        aux = apontadorIN + 4;
        apontadorIN = aux & MASKEND;
    }
    apontadorIN -= 2;

```

```
*apontadorIN++ = bloco.m4.ID;
*apontadorIN++ = bloco.m4.FRM;
*apontadorIN++ = bloco.m4.TRISTATE;
apontadorIN++;
bloco.m4.FRM++;
}

// Interrupção provocada pelo timer
void intTimer(void) interrupt 1 {

    TR0 = 0;
    TH0 = TC0INIC/256;
    TL0 = TC0INIC;
    TR0 = 1;
    timeout--;
    if (0==timeout) {
        timeout = 10;
        lsbTimeout += TIMEOUTINCREM; // teste de fim da fila
        endTimeout = BUFINIC + lsbTimeout;
    }
    if (flag4) {
        timer4--;
        if (timer4 == 0) {
            timer4 = TINIC4;
            transCAN4();
        }
    }
    if (flag3) {
        timer3--;
        if (timer3 == 0) {
            timer3 = TINIC3;
            transCAN3();
        }
    }
    if (flag2) {
        timer2--;
        if (timer2 == 0) {
            timer2 = TINIC2;
            transCAN2();
        }
    }
    if (flag1) {
        timer1--;
```

```

        if (timer1 == 0) {
            timer1 = TINIC1;
            transCAN1();
        }
    }
}

// Inicialização do uC
void inituC(void) {

    unsigned int i;
    TH0 = TC0INIC/256;
    TL0 = TC0INIC;
    TH1 = 255;
    IE = 0x83;
    IP = 0x11;
    TCON = 0x00;
    for (i=0;i<=LENGHT;i++) fila[i]=0x00;
    P3_5 = 1;
}

// Inicialização do SJA
void initSJA(void){

#ifdef placa
    unsigned int i;
    unsigned char ClockDivideReg;    // Clock Divider Register:
    // configuração PeliCAN/BasicCAN
    // Mode e bypass CAN input comp.
    enderec = BASE;
    do{    // RM = Reset Mode Request Bit
        *enderec = RM;// enquanto não for zero, não é
        // possível configurar o SJA
    }while(!(*enderec & RM));
    ClockDivideReg=0xe8;    // PeliCAN Mode e bypass do CAN input comp.
                           // porque usa transceiver externo

    enderec = CLKDIV;;
    *enderec=ClockDivideReg;
    enderec = CODMAS;    // ver descrição do vector de configuração
    for(i=0;i<8;i++) *enderec++ = init[i];    // ver configuração do SJA
    enderec=BUSTIM;
    for(i=8;i<11;i++) *enderec++ = init[i];

```

```

    enderec = BASE;
    do{
        *enderec = 0x00;
    }while(*enderec);
#endif
}

// Funções para pós-processamento
// envio para porta série, idêntico à mensagem do LCD
void portaSerie(char *ptr){
    while(*ptr){
        SBUF=*ptr++;
        while (!TI);
        TI = 0;
    }
}

// Passagem dos valores dos deadlines e quantidade de mensagens
void posProces(void){

    sprintf(dlCount,"ESCALONAMENTO 2 \n\r");
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",bloco.m1.ID,lostDeadl1,nrMSG1);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",bloco.m2.ID,lostDeadl2,nrMSG2);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",bloco.m3.ID,lostDeadl3,nrMSG3);
    portaSerie(dlCount);
    sprintf(dlCount,"Fluxo %u: %u/%u\n\r",bloco.m4.ID,lostDeadl4,nrMSG4);
    portaSerie(dlCount);
    portaSerie("\n\r");
}

// Função principal
void main(void){

    unsigned int aux;
    inituC();
    initSJA();
    flag1=flag2=flag3=flag4=1;
    timer1 = timer2 = timer3 = timer4 = 1;

```

```
    endTimeout = BUFINIC;lsbTimeout=0;
    timeout = TIMEOUTINIC;
    apontadorOUT = endTimeout;
    TMOD = 0x21;
    SCON = 0x50;
    PCON = PCON | 0x80;
    TCON = 0x50;
    bloco.m1.FRM=bloco.m2.FRM=bloco.m3.FRM=bloco.m4.FRM=0x00;
    bloco.m1.TRISTATE=bloco.m2.TRISTATE=0x01;
    bloco.m3.TRISTATE=bloco.m4.TRISTATE=0x01;
//placa2
#ifdef placa2
    bloco.m1.ID=1; bloco.m2.ID=3; bloco.m3.ID=5; bloco.m4.ID=7;
#endif
//placa1
#ifdef placa1
    bloco.m1.ID=2; bloco.m2.ID=4; bloco.m3.ID=6; bloco.m4.ID=8;
    P3_5 = 0;
#endif
    TR0 = 0;
    while(P3_5);
    TR0 = 1;
    while(1){
        if ((nrMSG1>MSGLIM) | (nrMSG2>MSGLIM)){
            flag1=flag2=flag3=flag4=0;
            TR0 = 0; P3_5 = 1;
            posProces();
            nrMSG1=nrMSG2=nrMSG3=nrMSG4=0;
        }
        aux = endTimeout + lsbAptOut;
        apontadorOUT = aux & MASKEND;
        if (*apontadorOUT != 0x00){
            apontadorOUT += 2;
            if (*apontadorOUT == 2) {
                abortarEnvio();
                lsbAptOut = 0;
            }else if (*apontadorOUT == 1) {
                enviar();
                lsbAptOut = 0;
            }else lsbAptOut += 4;
        }else lsbAptOut += 4;
        if (RI){
            RI=0;
            switch (SBUF){
```

```
case '1': flag1 = !flag1; break;
case '2': flag2 = !flag2; break;
case '3': flag3 = !flag3; break;
case '4': flag4 = !flag4; break;
case '5': flag1 = flag2 = flag3 = flag4 = 1; break;
case '0': {
    flag1 = flag2 = flag3 = flag4 = 0;
    timer1 = TINIC1;
    timer2 = TINIC2;
    timer3 = TINIC3;
    timer4 = TINIC4;
} break;
case 'p': {
    TR0 = 0;
    flag1=flag2=flag3=flag4=0;
    posProces();
    TR0 = 1;
} break;
}
}
}
}
```