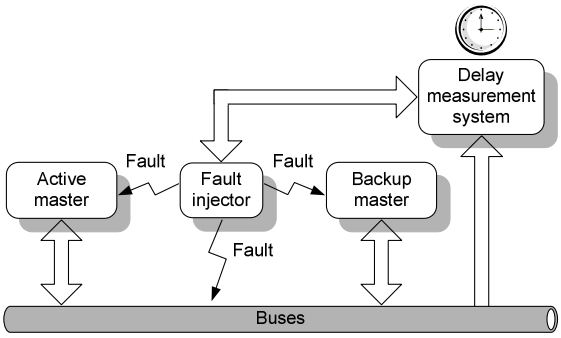
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática
2010

Valter Filipe Miranda Castelão da Silva

Gestão Flexível de Largura de Banda e Redundância em Barramentos de Campo

Flexible Redundancy and Bandwidth Management in Fieldbuses



Universidade de Aveiro
2010

Departamento de Electrónica, Telecomunicações e Informática

Valter Filipe Miranda Castelão da Silva

Gestão Flexível de Largura de Banda e Redundância em Barramentos de Campo

Flexible Management of Bandwidth and Redundancy in Fieldbuses

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Electrotécnica, realizada sob a orientação científica do Doutor José Alberto Gouveia Fonseca, Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Joaquim José de Castro Ferreira, Professor Adjunto da Escola Superior de Tecnologia e Gestão de Águeda da Universidade de Aveiro.

Dissertation submitted to the University of Aveiro in the fulfilment of the requirements for the degree of Doutor em Engenharia Electrotécnica, under the supervision of José Alberto Gouveiva Fonseca, Professor Associado at the Departamento de Electrónica, Telecomunicações e Informática of the University of Aveiro and co-supervision of Joaquim José de Castro Ferreira, Professor Adjunto at Escola Superior de Tecnologia e Gestão de Águeda of the University of Aveiro.

Apoio financeiro da FCT e do FSE no âmbito do III Quadro Comunitário de Apoio.

Apoio da Escola Superior de Tecnologia e Gestão de Águeda da Universidade de Aveiro e do IEETA de Aveiro.

Dedico este trabalho os meus pais, irmão, avós, restantes familiares e amigos.

o júri

Presidente

Prof. Doutor José Joaquim Costa Cruz Pinto
Professor catedrático do Departamento de Química da Universidade de Aveiro

Prof. Doutor Christian Fraboul
Professeur des Universités de l'Institut National Polytechnique de Toulouse, França

Prof. Doutor Adriano da Silva Carvalho
Professor associado da Faculdade de Engenharia da Universidade do Porto

Prof. Doutor Eduardo Manuel de Médicis Tovar
Professor coordenador do Instituto Superior de Engenharia do Instituto Politécnico do Porto

Prof. Doutor João Pedro Estima de Oliveira
Professor associado da Escola Superior de Tecnologia e Gestão de Águeda da Universidade de Aveiro

Prof. Doutor Pedro Nicolau Faria da Fonseca
Professor auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Prof. Doutor José Alberto Gouveia Fonseca
Professor associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Prof. Doutor Joaquim José de Castro Ferreira
Professor adjunto da Escola Superior de Tecnologia e Gestão de Águeda da Universidade de Aveiro

agradecimentos

O longo e árduo trabalho que culmina com a escrita (e defesa) desta tese chegou (finalmente) ao fim!
Neste longo percurso tive a ajuda directa ou indirecta de muitas pessoas, as quais têm o meu sincero agradecimento. Tenho a plena noção de que sem elas este trabalho não teria sucesso.
Neste pequeno texto vou-lhes fazer uma homenagem em poucas palavras que, significam uma profunda gratidão. Assim, e sem que a ordem reflicta qualquer ordenação, o meu muito obrigado:

A José Alberto Fonseca, amigo e (também) orientador científico deste trabalho, pela amizade demonstrada, pela sua orientação perspicaz e pelas horas de discussão sobre "tudo o resto" marginal ao trabalho! A sua ajuda passa muito os limites de um orientador.

A Joaquim Ferreira, amigo e (também) co-orientador científico, pela dedicação, "puxões de orelhas", correcção exemplar desta tese e por todos os momentos de boa disposição que proporciona no laboratório 319!

A Paulo Bartolomeu, pela preciosa ajuda nos testes ao FTT e no aperfeiçoamento do DMS. Também pela amizade e discussões até "altas horas" após os jantares que terminavam tarde (ou cedo, dependendo do ponto de vista).

A Paulo Pedreiras, pelo grande amigo e pessoa que é, pela sua sempre (boa) disposição para "dois dedos de conversa" sobre "tudo e sobre nada".

A Luís Almeida, pelas discussões sobre o andamento do trabalho e pelos empurrões nos momentos mais aborrecidos.

A Ricardo Marau pela amizade demonstrada ao longo destes anos e pelo Verão de 2009 passado frente-a-frente no laboratório 319!

A Mário Calha pelo trabalho que desenvolvemos juntos que em muito ajudou para este trabalho.

A Rui Santos, Alexandre Vieira, Vasco Santos, Arnaldo Oliveira, Frederico Santos, Fernando Dias, Ana Antunes e Zé Vieira pela amizade demonstrada.

A Manuel Barranco, Isidro Calvo e Francisco Carreiro pelos tempos que animaram o laboratório 319!

A Margarida Urbano pela amizade, pelas importantes ajudas na ESTGA e pela discussão de certos temas.

À ESTGA, na pessoa do seu Director, Estima de Oliveira e aos restantes colegas, em especial aos colegas de gabinete. Sem eles este trabalho não teria sido possível.

A todos os meus amigos e amigas por ter dito tantas vezes "não posso, tenho uma tese para fazer", Em particular ao Filipe Oliveira, António Tavares, Rui Soares, Carlos Neves e Marisa Matos.

A Manuela Quintaneiro e Margarida Lima pelas palavras encorajadoras nos momentos difíceis.

A todos os meus professores que me deram competências para chegar a este ponto da minha formação.

A Maria de Lurdes, pelo apoio incondicional, revisão do texto e tempo que não lhe dediquei.

Ao meu irmão, Telmo Silva, pela revisão da tese, pelas ajudas no trabalho e no resto, e por me aturar a má disposição!

Aos meus pais e avós por me aturarem as “taras e manias” e por tudo aquilo que hoje sou.

A toda a restante família (impossível de nomear!) por tudo o que representam e fazem por mim no dia-a-dia.

À Fundação para a Ciência e Tecnologia e ao Fundo Social Europeu pelo apoio financeiro tão importante para a minha dispensa de serviço docente. Adicionalmente a todos os que tornaram possível esse apoio e dispensa.

Apesar das palavras serem poucas comparadas com o que representam, expresso a todos os meus sinceros agradecimentos.

Muito obrigado!

palavras-chave

Redundância, confiança de funcionamento, Comunicações tempo-real, sistemas distribuídos, barramentos de campo, CAN-*Controller Area Network*.

resumo

Os sistemas distribuídos embarcados (*Distributed Embedded Systems* – DES) têm sido usados ao longo dos últimos anos em muitos domínios de aplicação, da robótica, ao controlo de processos industriais passando pela aviónica e pelas aplicações veiculares, esperando-se que esta tendência continue nos próximos anos.

A confiança no funcionamento é uma propriedade importante nestes domínios de aplicação, visto que os serviços têm de ser executados em tempo útil e de forma previsível, caso contrário, podem ocorrer danos económicos ou a vida de seres humanos poderá ser posta em causa.

Na fase de projecto destes sistemas é impossível prever todos os cenários de falhas devido ao não determinismo do ambiente envolvente, sendo necessária a inclusão de mecanismos de tolerância a falhas.

Adicionalmente, algumas destas aplicações requerem muita largura de banda, que também poderá ser usada para a evolução dos sistemas, adicionando-lhes novas funcionalidades.

A flexibilidade de um sistema é uma propriedade importante, pois permite a sua adaptação às condições e requisitos envolventes, contribuindo também para a simplicidade de manutenção e reparação. Adicionalmente, nos sistemas embarcados, a flexibilidade também é importante por potenciar uma melhor utilização dos, muitas vezes escassos, recursos existentes.

Uma forma evidente de aumentar a largura de banda e a tolerância a falhas dos sistemas embarcados distribuídos é a replicação dos barramentos do sistema. Algumas soluções existentes, quer comerciais quer académicas, propõem a replicação dos barramentos para aumento da largura de banda ou para aumento da tolerância a falhas. No entanto e quase invariavelmente, o propósito é apenas um, sendo raras as soluções que disponibilizam uma maior largura de banda e um aumento da tolerância a falhas. Um destes raros exemplos é o FlexRay, com a limitação de apenas ser permitido o uso de dois barramentos.

Esta tese apresentada e discute uma proposta para usar a replicação de barramentos de uma forma flexível com o objectivo duplo de aumentar a largura de banda e a tolerância a falhas. A flexibilidade dos protocolos propostos também permite a gestão dinâmica da topologia da rede, sendo o número de barramentos apenas limitado pelo hardware/software.

As propostas desta tese foram validadas recorrendo ao barramento de campo CAN – *Controller Area Network*, escolhido devido à sua grande implantação no mercado. Mais especificamente, as soluções propostas foram implementadas e validadas usando um paradigma que combina flexibilidade com comunicações *event-triggered* e *time-triggered*: o FTT – *Flexible Time-Triggered*. No entanto, uma generalização para CAN nativo é também apresentada e discutida.

A inclusão de mecanismos de replicação do barramento impõe a alteração dos antigos protocolos de replicação e substituição do nó mestre, bem como a definição de novos protocolos para esta finalidade. Este trabalho tira partido da arquitectura centralizada e da replicação do nó mestre para suportar de forma eficiente e flexível a replicação de barramentos. Em caso de ocorrência de uma falta num barramento (ou barramentos) que poderia provocar uma falha no sistema, os protocolos e componentes propostos nesta tese fazem com que o sistema reaja, mudando para um modo de funcionamento degradado. As mensagens que estavam a ser transmitidas nos barramentos onde ocorreu a falta são reencaminhadas para os outros barramentos.

A replicação do nó mestre baseia-se numa estratégia líder-seguidores (*leader-followers)*, onde o líder (*leader*) controla todo o sistema enquanto os seguidores (*followers*) servem como nós de reserva. Se um erro ocorrer no nó líder, um dos nós seguidores passará a controlar o sistema de uma forma transparente e mantendo as mesmas funcionalidades.

As propostas desta tese foram também generalizadas para CAN nativo, tendo sido para tal propostos dois componentes adicionais. É, desta forma possível ter as mesmas capacidades de tolerância a falhas ao nível dos barramentos juntamente com a gestão dinâmica da topologia de rede.

Todas as propostas desta tese foram implementadas e avaliadas. Uma implementação inicial, apenas com um barramento foi avaliada recorrendo a uma aplicação real, uma equipa de futebol robótico onde o protocolo FTT-CAN foi usado no controlo de movimento e da odometria.

A avaliação do sistema com múltiplos barramentos foi feita numa plataforma de teste em laboratório. Para tal foi desenvolvido um sistema de injecção de faltas que permite impor faltas nos barramentos e nos nós mestre, e um sistema de medida de atrasos destinado a medir o tempo de resposta após a ocorrência de uma falta.

keywords

Redundancy, dependability, real-time communications, distributed systems, fieldbuses, CAN-Controller Area Network.

abstract

Distributed embedded systems (DES) have been widely used in the last few decades in several application domains, from robotics, industrial process control, avionics and automotive. In fact, it is expectable that this trend will continue in the next years.
In some of these application fields the dependability requirements are very important since the fail to provide services in a timely and predictable manner may cause important economic losses or even put humans in risk.
In the design phase it is impossible to predict all the possible scenarios of faults, due to the non deterministic behaviour of the surrounding environment. In that way, the fault tolerance mechanisms must be included in the distributed embedded system to prevent failures occurrence.
Also, many application domains require a high available bandwidth to perform the desired functions, or to turn possible the scaling with the addition of new features.
The flexibility of a system also plays an important role, since it improves the capability to adapt to the surrounding world, and to the simplicity of the repair and maintenance. The flexibility improves the efficiency of all the system by providing a way to efficiently manage the available resources. This is very important in embedded systems due to the limited resources often available.
A natural way to improve the bandwidth and the fault tolerance in distributed systems is to use replicated buses. Commercial and academic solutions propose the use of replicated fieldbuses for a single purpose only, either to improve the fault tolerance or to improve the available bandwidth, being the first the most common. One illustrative exception is FlexRay where the bus replica can be used to improve the bandwidth of the overall system, besides enabling redundant communications. However, only one bus replica can be used.
In this thesis, a flexible bus replication scheme to improve both the dependability and the throughput of fieldbuses is presented and studied. It can be applied to any number of replicated buses, provided the required hardware support is available. The flexible use of the replicated buses can achieve an also flexible management of the network topology.
This claim has been validated using the Controller Area Network (CAN) fieldbus, which has been chosen because it is widely spread in millions of systems. In fact, the proposed solution uses a paradigm that combines flexibility, time and event triggered communication, that is the Flexible Time-Triggered over CAN network (FTT-CAN). However, a generalization to native CAN is also presented and studied.

The inclusion of bus replication in FTT-CAN imposes not only new mechanisms but also changes of the mechanisms associated with the master replication, which has been already studied in previous research work. In this work, these mechanisms were combined and take advantage of the centralized architecture and of the redundant masters to support an efficient and flexible bus replication.

When considering the system operation, if a fault in the bus (or buses) occurs, and the consequent error leads to a system failure, the system reacts, switching to a degraded mode, where the message flows that were transmitted in the faulty bus (or buses) change to the non-faulty ones.

The central node replication uses a leader-follower strategy, where the leader controls the system while the followers serve as backups. If an error occurs in the leader, a backup will take the system control maintaining the system with the same functionalities.

The system has been generalized for native CAN, using two additional components that provide the same fault tolerance capabilities at the bus level, and also enable the dynamic management of the network topology.

All the referred proposals were implemented and assessed in the scope of this work. The single bus version of FTT-CAN was assessed using a real application, a robotic soccer team, which has obtained excellent results in international competitions. There, the FTT-CAN based embedded system has been applied in the low level control, where, mainly it is responsible for the motion control and odometry.

For the case of the multiple buses system, the assessment was performed in a laboratory test bed. For this, a fault injector was developed in order to impose faults in the buses and in the central nodes. To measure the time reaction of the system, a special hardware has been developed: a delay measurement system. It is able to measure delays between two important time marks for posterior offline analysis of the obtained values.

Contents

1 Introduction **1**
- 1.1 The problem . . . 1
- 1.2 The thesis . . . 2
- 1.3 Contributions . . . 2
  - 1.3.1 A proposed solution to use multiple buses in a flexible way . . . 3
  - 1.3.2 Mechanisms to provide fault detection among multiple masters . . . 3
  - 1.3.3 A generalization of the proposed solutions for native CAN . . . 3
  - 1.3.4 A fault injector system for bus based communication systems . . . 4
  - 1.3.5 A delay measurement system . . . 4
  - 1.3.6 An implementation of the FTT-CAN middleware . . . 4
- 1.4 Publications . . . 4
- 1.5 Organization of the dissertation . . . 7

2 Bus media redundancy: a survey **9**
- 2.1 Introduction . . . 9
- 2.2 Fault tolerant communication: a brief introduction . . . 10
  - 2.2.1 Introduction . . . 10
  - 2.2.2 Dependability . . . 10
  - 2.2.3 Fault, error and failure . . . 12
  - 2.2.4 Fieldbuses and dependability . . . 12
  - 2.2.5 Redundancy and fault tolerance . . . 14
  - 2.2.6 Replica consistency . . . 16
- 2.3 Buses for industrial automation . . . 17
  - 2.3.1 Introduction . . . 17
  - 2.3.2 WorldFIP . . . 17
  - 2.3.3 PROFIBUS . . . 20
  - 2.3.4 P-NET . . . 21
- 2.4 Buses for embedded applications . . . 22
  - 2.4.1 Introduction . . . 22
  - 2.4.2 TTP . . . 23
  - 2.4.3 FlexRay . . . 24
  - 2.4.4 MIL-STD-1553B . . . 27
- 2.5 Ethernet based solutions . . . 28
- 2.6 Controller Area Network (CAN) . . . 29
  - 2.6.1 Introduction . . . 29
  - 2.6.2 CAN Basics . . . 30
    - Physical layer . . . 30
    - Data link layer . . . 31
    - CAN fault tolerance . . . 33

2.6.3 CAN protocols with media redundancy . . . 36
TTCAN . . . 36
FlexCAN . . . 38
RedCAN . . . 39
Columbus Egg Idea . . . 40
CANdor . . . 41
Fault-Active Mechanism . . . 42
CANopen . . . 43
MilCAN . . . 45
DeviceNet . . . 45
2.6.4 CAN star topologies . . . 46
2.7 Brief comparison . . . 48
2.8 Conclusions . . . 51

3 A proposal for bus media redundancy in FTT-CAN 53
3.1 Introduction . . . 53
3.2 The FTT-CAN protocol . . . 54
3.2.1 FTT master replication . . . 56
3.2.2 FTT features summary . . . 57
3.3 Limitations of the FTT-CAN . . . 57
3.3.1 The priority inversion and jitter problem . . . 57
3.3.2 Network redundancy support . . . 59
3.3.3 Available bandwidth . . . 59
3.4 FTT-CAN with multiple buses . . . 64
3.4.1 Introduction . . . 64
3.4.2 Multiple buses architecture . . . 65
3.4.3 The trigger message . . . 65
Bandwidth allocation for synchronous and asynchronous messages . . . 69
Strategy and trigger flags example . . . 71
3.4.4 Master replication . . . 72
3.5 Masters and buses: errors and replacement . . . 73
3.5.1 Introduction . . . 73
3.5.2 Fault model and assumptions . . . 74
3.5.3 Detectable faults . . . 74
3.5.4 Master errors: replacement and analysis . . . 75
3.5.5 Bus errors: replacement and analysis . . . 79
Reconfiguring the buses . . . 81
Bus error detection in two or more buses and practical issues . . . 83
3.6 Final remarks . . . 86
3.7 Conclusions . . . 87

4 FTT-CAN Implementation 89
4.1 Introduction . . . 89
4.2 Data structures and API . . . 89
4.2.1 The synchronous requirement database . . . 93
4.2.2 The trigger message . . . 97
4.2.3 Application programming interface . . . 99
4.3 The master implementation . . . 100
4.3.1 Single bus master . . . 101
4.3.2 Multiple buses master . . . 105

4.4 The slave implementation . . . 107
4.5 Evaluation of the computational overhead . . . 107
4.5.1 Experimental results . . . 112
4.6 Conclusions . . . 113

5 Experimental evaluation **115**
5.1 Introduction . . . 115
5.2 Single bus mechanisms' assessment . . . 115
5.2.1 Experimental platform: the CAMBADA soccer robot . . . 116
5.2.2 Communication requirements . . . 117
5.2.3 Communications architecture . . . 119
5.2.4 Synchronizing data flows . . . 121
5.2.5 Experimental validation . . . 123
5.3 Multiple buses experimental platform and results . . . 126
5.3.1 Experiment's rationale . . . 126
5.3.2 Fault injector and delay measurement system . . . 129
Fault injector . . . 129
The delay measurement system (DMS) . . . 130
Using the DMS in related work . . . 131
Joining time measurement and fault injection . . . 133
5.3.3 Experimental setup and results . . . 134
Bus error test results . . . 140
Active master error test results . . . 142
5.3.4 Practical issues . . . 144
Comparing expected and practical delays . . . 147
5.4 Results summary . . . 149
5.5 Conclusions . . . 150

6 Using multiple buses in native CAN - A generalization **153**
6.1 Introduction . . . 153
6.2 Comparing bus and stars architecture in terms of wiring . . . 153
6.3 Providing automatic bus redundancy in legacy CAN . . . 156
6.4 Fault hypothesis . . . 157
6.5 Network switch unit . . . 158
6.6 Topology management unit . . . 159
6.6.1 Topology management unit replication protocol . . . 161
6.7 Operational scenarios . . . 162
6.8 Implementation . . . 164
6.9 Conclusions . . . 165

7 Conclusions and future work **167**
7.1 Thesis validation . . . 169
7.2 Future research . . . 169
7.2.1 The asynchronous messaging system . . . 169
7.2.2 Slave nodes . . . 170
7.2.3 Dependability evaluation . . . 170
7.2.4 Generalization to other protocols . . . 170
7.2.5 Wireless and heterogeneous networks . . . 170

List of Figures

2.1 The dependability tree (from [ALR01]) . . . 11
2.2 Network topologies . . . 16
2.3 Macro cycle example (from [AC98]) . . . 19
2.4 PROFIBUS node architecture with redundant bus(from [LS95]) . . . 21
2.5 FlexRay network topologies . . . 25
2.6 FlexRay communication cycle (from [MT06]) . . . 26
2.7 MIL-STD-1553B system architecture . . . 27
2.8 CAN bit timing (from [RVA99]) . . . 30
2.9 CAN 2.0A frame format . . . 32
2.10 Failures in CAN ISO 11898-2 (from [RVA99]) . . . 34
2.11 Error frame . . . 35
2.12 CAN error states . . . 36
2.13 TTCAN network example (from [MFH+02]) . . . 37
2.14 Fault tolerant TTCAN network example (from [MFH+02]) . . . 37
2.15 FlexCAN node and OSI layers (from [PF04]) . . . 38
2.16 FlexCAN architecture (from [PF04]) . . . 39
2.17 RedCAN module (from [SOJT04]) . . . 39
2.18 Columbus Egg Idea physical layer (from [RVA99]) . . . 40
2.19 CANdor node (from [FNP+98]) . . . 41
2.20 Fault tolerant communication node (from [HKD97]) . . . 42
2.21 CANopen redundant communication . . . 44
2.22 ReCANCentrate architecture (from [BPA09]) . . . 47

3.1 FTT-CAN architecture . . . 54
3.2 The elementary cycle in FTT-CAN . . . 55
3.3 Master-multislave access control and EC schedule coding scheme . . . 55
3.4 FTT-CAN with multiple buses architecture . . . 65
3.5 Scenario (a) . . . 67
3.6 Scenario (b) . . . 68
3.7 Scenario (c) . . . 68
3.8 Example of trigger message with multiple buses . . . 72
3.9 System architecture for multiple masters . . . 73
3.10 Master replacement state diagram . . . 76
3.11 Master replacement protocol . . . 77
3.12 Bus error detection mechanism . . . 79
3.13 Bus error asynchronous message (BEAM) . . . 81
3.14 Bus changing time flow . . . 83
3.15 Multiple errors detection . . . 84
3.16 Delay in transmission of the trigger messages in different buses . . . 85

4.1 FTT stack . . . 92
4.2 SRT line for multiple buses . . . 96
4.3 Synchronous requirement table architecture . . . 97
4.4 Trigger message structure . . . 98
4.5 Development board for PIC18F258 . . . 101
4.6 Development board for dsPIC30F6012A . . . 102
4.7 Master firmware overview . . . 103
4.8 Timer occurrences in masters . . . 104
4.9 Master firmware with multiple buses . . . 106
4.10 Slave firmware overview . . . 108
4.11 Backup master computational overhead . . . 110

5.1 The biomorphic architecture of the CAMBADA robots . . . 117
5.2 Functional robot modules . . . 117
5.3 FTT-CAN low-level control system . . . 122
5.4 Timeline for motion information flow . . . 123
5.5 Master replacement delay . . . 125
5.6 Assessment setup global view . . . 126
5.7 Bus error timeline . . . 128
5.8 Master error timeline . . . 129
5.9 Multiplexer structure . . . 129
5.10 CAN bus multiplexers . . . 131
5.11 Fault injector internal modules . . . 132
5.12 DMS internal structure . . . 133
5.13 xDMS internal modules . . . 134
5.14 Measurement architecture . . . 135
5.15 Developed system . . . 136
5.16 Maximum of t_{phase} . . . 137
5.17 Fault-injection instant histogram . . . 138
5.18 t_{BEAM} histogram (absolute delay) . . . 140
5.19 $Rel(t_{BEAM})$ histogram (relative delay) . . . 141
5.20 $t_{re\_scheduling}$ histogram (absolute delay) . . . 141
5.21 $Rel(t_{re\_scheduling})$ histogram (relative delay) . . . 142
5.22 $Rel(t_{BEAM})$ and $Rel(t_{re\_scheduling})$ injecting a dominant bit . . . 142
5.23 t_{TM_first} histogram (absolute delay) . . . 143
5.24 $Rel(t_{TM\_first})$ histogram (relative delay) . . . 143
5.25 $Rel(t_{TM\_second})$ histogram (relative delay) . . . 144
5.26 Practical analysis for bus error timeline . . . 145
5.27 Practical analysis for master error timeline . . . 146

6.1 Star topology best scenario . . . 154
6.2 Bus topology best scenario . . . 154
6.3 Replicated network with heterogeneous nodes . . . 156
6.4 Proposed architecture . . . 157
6.5 Network switch unit architecture . . . 159
6.6 Topology management unit architecture . . . 160
6.7 Architecture example . . . 162
6.8 Operational scenario example . . . 163
6.9 Operational scenario: star . . . 164
6.10 Switch controller (from [Sil09]) . . . 165

List of Tables

2.1 Type of redundancy . . . 15
2.2 Bus arbitrator table example . . . 18
2.3 Industrial automation protocols comparison . . . 48
2.4 Embedded applications protocols comparison . . . 49
2.5 CAN based protocols comparison . . . 50
2.6 CAN star topologies comparison . . . 51
2.7 Ethernet based protocols comparison . . . 52

3.1 CAN overhead . . . 61
3.2 Trigger message overhead for $5ms$ of elementary cycle . . . 63
3.3 Trigger message overhead for $20ms$ of elementary cycle . . . 63

4.1 Message and task parameters mapping . . . 94
4.2 Trigger message parameters mapping . . . 97

5.1 Low-level control layer communication requirements . . . 118
5.2 Low-level control layer message set and activity tasks . . . 121
5.3 Timeliness of information flow . . . 124
5.4 CAN high and CAN low lines voltages . . . 130
5.5 Multiplexer functions . . . 130
5.6 xDMS parameters for bus faults . . . 139
5.7 xDMS parameters for master faults . . . 139
5.8 Observed, corrected and theoretical delays for bus error . . . 148
5.9 Observed, corrected and theoretical delays for bus master error . . . 148
5.10 Results summary . . . 150

6.1 Cable length comparison . . . 155
6.2 NSU table example . . . 162
6.3 NSU table example after faulty bus . . . 163

Chapter 1

Introduction

1.1 The problem

In the last few decades Distributed Embedded Systems (DES) have been widely used in several application fields, ranging from industrial machinery to avionics, automotive systems and robotics. Most of these applications have strict timeliness requirements. Also, the requirements of these systems that can only be met with deterministic networks, and rely on distributed coordination that require synchronization protocols. Besides in many cases, they require mechanisms for online reconfiguration and fault tolerance. These mechanisms impose an overhead in terms of network bandwidth and computational performance of the nodes.

Fieldbuses are usually adopted for the network infrastructure. One of the most popular fieldbuses is CAN - Controller Area Network [BOS91]. CAN protocol was initially targeted to automotive control systems, as a single digital bus to replace traditional point-to-point cables that were growing in complexity, weight and cost with the introduction of new electrical and electronic systems. The widespread and successful use of CAN in the automotive industry, the low cost associated with high volume production of controllers and its inherent technical merit, have driven to CAN adoption in other application domains. Despite its success story, CAN application designers would be happier if CAN could be made faster, cover longer distances, be more deterministic and more dependable. Some current distributed control applications require higher bandwidth [SFNM05] and the dependability of native CAN is not adequate for some applications, *e.g.*, wheel-chairs robots [BHHP01], autonomous robots for urban transportation vehicles [BNMS05], urban transportation system [MZP$^+$02, WTSW03], x-by-wire system for automotive [Tea98] and home automation systems [BFS$^+$07, FBS$^+$08].

The shortcomings of CAN call for new solutions, either based on CAN extensions/improvements or on the adoption of another fieldbus protocol. It can be said that CAN technology is now mature and engineers are well trained in designing and maintaining CAN based DES, with relative low time to market. So a potential migration to other fieldbus technology is challenging and it could take several years.

In this scenario, and not only to replace CAN, new fieldbuses were proposed, notably FlexRay [Fle02] and TTP/C [KG94] (TTP stands for Time-Triggered Protocol). In parallel with the advent of new fieldbus protocols, extensions/improvements of CAN have been proposed over the years, such as: Time-Triggered CAN (TTCAN) [HMFH00, ISO01], Flexible Time-Triggered over CAN (FTT-CAN) [APF02] or FlexCAN [PF04]. Notice, however, that none of these solutions solve the problems of native CAN, just minimize them. Indeed some solutions address fault tolerance issues, while others increase the available bandwidth.

In this context, it would be desirable to increase the available bandwidth of CAN and to improve its fault tolerance capabilities while maintaining the backward compatibility with legacy systems. A possible solution to this problem is using multiple buses to provide additional bandwidth, which is proportional to the number of used buses. This extra bandwidth can be used either to provide bus media redundancy only, transmitting different data in different buses, or to provide data redundancy, transmitting the same data in different buses. A third alternative would be combining these two approaches in a flexible way, *i.e.*, the network could provide a mix of bus media redundancy and data redundancy. This is a flexible approach in the sense that the bus media redundancy and the data redundancy trade-off can be assigned online according to the application requirements in terms of bandwidth and dependability.

Besides the described features, a multiple buses architecture can also be used to tolerate bus failures, so that the network can be switched to degraded modes using less buses than originally.

This thesis proposes components and protocols to take advantage of multiple CAN buses, and to achieve bus fault tolerance and efficient bandwidth usage. Notice that FTT-CAN is used as a proof of concept, since the proposal can be generalized to other fieldbus protocol, including native CAN.

1.2 The thesis

The thesis supported by the present dissertation argues that:

> *The use of a flexible bus replication scheme could improve both the dependability and the throughput of a network. Furthermore, it is possible to adapt online the network topology to evolving operational scenarios.*

1.3 Contributions

The main contributions presented in this dissertation are:

- A proposed solution to use multiple buses in a flexible way;
- Mechanisms to provide fault detection among multiple masters;

- A generalization of the proposed solutions for native CAN;
- A fault injector system for bus based communication systems;
- A delay measurement system (DMS);
- An implementation of the FTT-CAN middleware.

Despite the system architecture, protocols and components, are targeted to the FTT-CAN protocol [APF02] and native CAN, most of the contributions could equally generalized to other fieldbuses.

Next, the contributions are summarized.

1.3.1 A proposed solution to use multiple buses in a flexible way

The previous versions of FTT-CAN support only one bus making it a single point of failure. One way to avoid this problem is by replicating the bus.

This thesis presents a proposal to replicate the bus to provide flexible bus media redundancy and data redundancy.

The proposal takes advantage of the bus media redundancy to increase the dependability while providing additional bandwidth. In this way, whenever a bus would fail, the affected traffic could be switched to other bus and the network would start operating in a degraded mode. The maximum number of buses the system can accommodate is only limited by the hardware, such as the number of CAN interfaces or by the performance limits of the microcontrollers used.

The management of buses and messages is made by the FTT-CAN central node, the master, that is also replicated. The proposal was implemented and validated using a fault injection system combined with a delay measurement system.

1.3.2 Mechanisms to provide fault detection among multiple masters

The previous FTT-CAN architecture included a protocol to handle master replication [MFA+02]. However, with the introduction of bus media redundancy, the master replication protocol needs to be updated in order to accommodate the management of multiple buses. This thesis proposes a new master replacement protocol, based on the single bus version [FPAF02], where the active master node is responsible to detect and to react to permanent bus faults. This protocol relies on the masters location, at both ends of the buses, to have a global view of the network and to avoid bus partitions.

1.3.3 A generalization of the proposed solutions for native CAN

The proposed multiple buses and multiple master system is targeted to FTT-CAN, however, it can be generalized to any fieldbuses. A proposal to generalize the bus media redun-

dancy to native CAN is presented in this work. The generalization of the concept requires two new components, a Topology Management Unit (TMU) and a Network Switch Unit (NSU), that were designed and partially implemented. The network topology management has a global view of the network and controls all the network switch units. The network topology management is replicated and follows the same strategy as the master replication node in the FTT-CAN bus replication presented before. This is, the network topology management units are located at the end of the CAN bus and exchange messages with the other located in the other end.

1.3.4 A fault injector system for bus based communication systems

A hardware of a fault injector was developed to assess the multiple buses and multiple masters system. This fault injector imposes faults in the CAN bus (or buses) or in the FTT-CAN master node. The fault injector is capable of injecting different kinds of faults at pre-defined instants.

The fault injector has been designed for the FTT-CAN, but it can be used in native CAN without any adaptation. Moreover, it can be easily adapted to any other fieldbus protocol.

1.3.5 A delay measurement system

The fault injector works in close cooperation with a delay measurement system that recognizes the instant of the fault injection and the relevant timing instants of the FTT-CAN, to measure the time elapsed between these different instants. The collected data, a histogram, is stored internally in the Delay Measurement System (DMS) and it can be uploaded to a personal computer.

The DMS is generic, thus it can be used in other contexts, where the events to recognize are not related to FTT-CAN. For example, it has been used in an Ethernet based system [BSF07].

1.3.6 An implementation of the FTT-CAN middleware

A new, designed from scratch, FTT-CAN implementation was made. This implementation became the FTT-CAN reference implementation, used in real world applications.

The FTT-CAN implementation includes the single bus nodes with all the defined features. For the case of the multiple buses, only the master node was implemented. Slave nodes with multiple buses were not implemented because they are out of scope of this thesis.

1.4 Publications

The presented contributions have been published in a book chapter, journals and conference proceedings. They are listed below:

Book chapter

- Luís Almeida, Paulo Pedreiras, Joaquim Ferreira, Mário Calha, José Fonseca, Ricardo Marau, Valter Silva, and Ernesto Martins. *Handbook of Real-Time and Embedded Systems*, chapter 19, pages 19–1 to 19–22. CRC Press, 2007;

Journal papers

- Ricardo Moraes, Francisco Carreiro, Paulo Bartolomeu, Valter Silva, José Fonseca, and Francisco Vasques. Enforcing the timing behavior of real-time stations in legacy bus-based industrial Ethernet networks. *Computer Standards & Interfaces*, In Press, Corrected Proof, 2010;

- Ricardo Marau, Valter Silva, Joaquim Ferreira, Luís Almeida, and José Fonseca. Assessment of FTT-CAN master replication mechanisms for safety-critical applications. *Transactions Journal of Passenger Cars-Electronic and Electrical Systems*, pages 447–455, March 2007♣. Also presented at SAE 2006 World Congress (see paper ♠);

Conference papers

- Valter Silva, Paulo Bartolomeu, Joaquim Ferreira, and José Fonseca. Assessment of multi-bus fault-tolerant communications. In *proceedings of the* 7^{th} *IEEE International Conference on Industrial Informatics (INDIN)*, pages 72 –78, Cardiff, Wales, UK, June 2009;

- José Fonseca, Paulo Bartolomeu, Valter Silva, Vasco Santos, Carlos Abreu, Alexandre Mota, Margarida Cunha, and Arminda Lopes. Using CAN to retrofit houses for quadriplegic people. In 12^{th} *International CAN Conference*, Barcelona, Spain, March 2008;

- Valter Silva, José Fonseca, and Joaquim Ferreira. Adapting the FTT-CAN Master for Multiple-bus Operation. In *proceedings of the* 5^{th} *IEEE International Conference on Industrial Informatics (INDIN)*, Vienna, Austria, July 2007;

- Paulo Bartolomeu, José Fonseca, Vasco Santos, Alexandre Mota, Valter Silva, and Margarida Sizenando. Automating Home Appliances For Elderly and Impaired People: The B-Live Approach. In *Software Development for Enhancing Accessibility and Fighting Info-exclusion*, Vila Real, Portugal, November 2007;

- Valter Silva, Joaquim Ferreira, and José Fonseca. Flexible Bus Media Redundancy. In *proceedings of the* 4^{th} *International Workshop on Dependable Embedded Systems (in conjunction with the* 26^{th} *Symposium on Reliable Distributed Systems)*, Beijing, China, October 2007;

- Valter Silva, Joaquim Ferreira, and José Fonseca. Master Replication and Bus Error Detection in FTT-CAN with Multiple Buses. In *proceedings of the* 12^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Patras, Greece, 2007;

- Paulo Bartolomeu, Valter Silva, and José Fonseca. Delay measurement system for real-time serial data streams. In *proceedings of* 12^{th} *IEEE Conference Emerging Technologies and Factory Automation (ETFA)*, pages 516–523, Patras, Greece, September 2007;

- José Fonseca, Paulo Bartolomeu, Valter Silva, and Francisco Carreiro. On the practical issues of implementing the VTPE-hBEB protocol in small processing power controllers. In *proceedings of the* 12^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 949 – 956, Patras, Greece, September 2007;

- Ricardo Marau, Valter Silva, Joaquim Ferreira, Luís Almeida, and José Fonseca. Assessment of FTT-CAN master replication mechanisms for safety-critical applications. In *proceedings of the 2006 SAE congress & exhibition*, number 06AE-278, 2006♠. published at Transactions Journal of Passenger Cars-Electronic and Electrical Systems (see paper ♣);

- Valter Silva, Joaquim Ferreira, and José Fonseca. Dynamic Topology Management in CAN. In *proceeding of the* 11^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1222 – 1229, Prague, Czech Republic, September 2006;

- Valter Silva, José Fonseca, and Joaquim Ferreira. Using FTT-CAN to the Flexible Control of Bus Redundancy and Bandwidth Usage. In *proceedings of the* 11^{th} *International CAN Conference (iCC)*, pages 5.9 – 5.15, Stockholm, Sweden, September 2006;

- Valter Silva and José Fonseca. Using FTT-CAN to Combine Redundancy with Increased Bandwidth. In *proceedings of the* 6^{th} *IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 54–62, Torino, Italy, June 2006;

- Mário Calha, José Fonseca, Valter Silva, and Ricardo Marau. Kernel design for FTT-CAN systems. In *proceedings of the* 6^{th} *IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 151 – 156, Torino, Italy, June 2006;

- Fernando Ataide, Carlos Pereira, and Valter Silva. A New Approach for Time-Triggered Phase in the FTT-CAN Protocol: A Case Study in an Automotive System. In *Workshop on Models and Analysis Methods for Automotive Systems,* 27^{th} *IEEE Real-Time Systems Symposium*, Brasil, December 2006;

- Valter Silva, José Fonseca, Urbano Nunes, and Rodrigo Maia. Communications Requirements for Autonomous Mobile Robots: Analysis and Examples. In Elsevier, editor, *proceeding of FeT 2005*, pages 91 – 98, Mexico, November 2005;

- Mário Calha, Valter Silva, and José Fonseca. Real-Time Procedures in Distributed Systems. In *proceedings from IFAC International Conference on Fieldbus Systems and their Applications (FeT)*, pages 24 – 31, Mexico, November 2005;

- Valter Silva, Ricardo Marau, Luís Almeida, Joaquim Ferreira, Mário Calha, Paulo Pedreiras, and José Fonseca. Implementing a distributed sensing and actuation system: The CAMBADA robots case study. In *proceedings of the* 10^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 2, pages 781–788, Catania, Italy, September 2005;

- Luís Almeida, Frederico Santos, Tullio Facchinetti, Paulo Pedreiras, Valter Silva, and Luís Lopes. Coordinating distributed autonomous agents with a real-time database: The CAMBADA project. In *proceedings of the* 19^{th} *International Symposium on Computer and Information Sciences (ISCIS)*, 2004;

- Francisco Carreiro, José Fonseca, Valter Silva, and Francisco Vasques. The virtual token-passing Ethernet implementation and experimental results. In *proceedings of the* 3^{rd} *International Workshop on Real-Time Networks*, pages 57 – 60, Catania, Italy, June 2004.

1.5 Organization of the dissertation

In order to support the thesis previously stated, this dissertation is organized as follows:

Chapter 2 – Presents background information concerning dependability, and more specifically redundancy in fieldbuses. This chapter presents a survey of the protocols which support redundant buses or nodes. In particular, the CAN protocol is analyzed, with special attention to its dependability features. This chapter also presents a comparative study regarding the use of redundancy.

Chapter 3 – This chapter begins with the presentation of the FTT-CAN protocol in its single bus, multiple master version, highlighting the limitations in terms of dependability and bandwidth. The FTT-CAN protocol with bus media redundancy is then presented and discussed. This chapter also presents the necessary adaptations of the master node so it could handle multiple buses. Special attention is given to the master replication protocol for the multiple buses case.

Chapter 4 – This chapter describes the implementation of FTT-CAN protocol with special emphasis on the application programming interface available for FTT-CAN applications and presents the relevant data structures. The single bus implementation is a particular instance of the multiple buses one. Despite the slave implementation is only made for the single bus version, it is also discussed in this chapter.

Chapter 5 – The implementation presented in the previous chapter is verified and validated in this chapter. This chapter describes test platforms which are different for the single and multiple buses implementation. The additional hardware needed to perform the verification and validation, the fault injector and delay measurement system, is also described. The chapter is concluded with the experimental results and their discussion.

Chapter 6 – This chapter presents a generalization of the proposed solutions to legacy CAN networks. Two new components are introduced: the network switch unit and the topology management unit. A preliminary implementation of the network switch unit is presented.

Chapter 7 – Sets the conclusion of the dissertation and points out several directions for future work.

Chapter 2

Bus media redundancy: a survey

2.1 Introduction

This chapter presents a survey of the state of the art concerning media redundancy techniques used in communications for safety critical applications. The chapter is divided into five main sections: a brief introduction of fault tolerant communications (section 2.2), buses for industrial automation (section 2.3), buses for embedded applications (section 2.4), Ethernet based solutions (section 2.5) and Controller Area Network (section 2.6).

The first section presents relevant definitions and concepts on the dependability, redundancy and fault tolerance techniques applied to communications.

The second section considers the fieldbuses for interconnecting industrial automation devices, such as the ones used in factory automation, process control and similar applications, *e.g.*, WorldFIP, PROFIBUS and P-NET.

The third section considers fieldbuses that operate embedded in machines, in a general sense. Typical applications include avionics, automotive and robotics. Examples of such fieldbuses are TTP and FlexRay.

The fourth section addresses the protocols based on Ethernet (section 2.5), that are sometimes derived from the mentioned before.

Controller Area Network could obviously be included in one of the previous sections (buses for automation and/or buses for embedded applications), considering either its upper layer protocols such as DeviceNet and CANopen or the native CAN protocol itself. However, since the work presented in this thesis uses CAN, it was decided to describe CAN and its higher layer protocols in a specific section. The CAN section presents the fundamentals of CAN, with a special emphasis on the physical medium and introduces the redundancy techniques used in CAN based communications. Some adaptations of CAN like star topologies are also analyzed.

When considering the communication solutions analyzed in this chapter, it was decided to include a general overview of each and to devote a special attention to the physical layer

redundancy. This was required to enable comparisons with the main results of the work presented in this dissertation.

2.2 Fault tolerant communication: a brief introduction

2.2.1 Introduction

When talking about fault tolerant communication it becomes necessary to introduce a set of definitions and a set of principles. The main works that include definitions concerning fault tolerant systems are [Lap92], [Lap95] and [ALR01]. There, dependability is defined using a set of attributes and means that must be met by the system to resist the threats it is subject to. Fault tolerance is one of the means to achieve dependability.

The definition of a fault tolerant system needs first the definition of a dependable system to be understandable. Both definitions will be presented in this section.

The dependability of a distributed system can be affected by the implicit use of the network. One reason for this is that the use of the communication network needs more electronic components and includes a novel resource, the physical medium. Also, the use of a communication network implies that this network can suffer from external interferences, such as electromagnetic interferences. The availability of the medium is then another concern.

One way to improve the system fault tolerance is to provide components replication, either software or hardware based. In this section the discussion on the replication issues is focused in the network. The component replicas must be synchronized to guarantee a correct operation of the system. Replica consistency is also briefly discussed further in this section.

2.2.2 Dependability

As stated in [ALR01], dependability of a computer system is "*The ability to deliver service that can justifiably be trusted. The **service** delivered by a system is its behaviour as it is perceived by its user(s); a **user** is another system (physical, human) that interacts with the former at the **service interface**. The **function** of a system is what the system is intended for, and is described by the system specification*"

The concept of dependability can be explained with support of three elements: the **threats** to, the **attributes** of, and the **means** by which dependability is attained as shown is figure 2.1.

As it can be seen in figure 2.1, dependability is a complex concept. Mainly, there are six attributes: availability, reliability, safety, confidentiality, integrity and maintainability. More specifically:

- Availability is the capacity of the system to be ready to offer a correct service;
- Reliability is the attribute of the system to offer the continuity of a correct service;

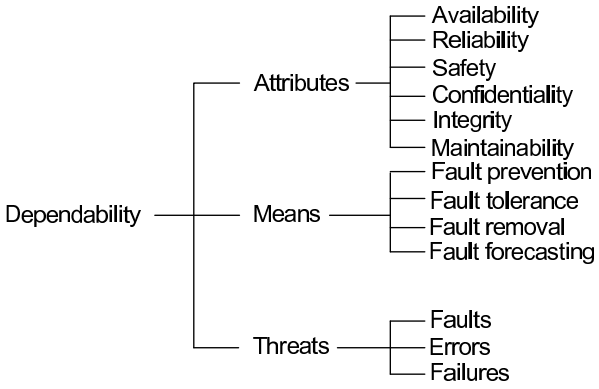


Figure 2.1: The dependability tree (from [ALR01])

- Safety is the attribute of guaranteeing the absence of catastrophic consequences to the user(s) and the environment;
- Confidentiality is the attribute for the absence of unauthorized disclosure of information;
- Integrity is the attribute for the absence of improper system state change;
- Maintainability is the ability to undergo repairs and modifications.

In Portuguese we use the word "segurança" with a broader meaning than in English. In English "segurança" can be mapped either in safety and in security. Security is the absence of unauthorized access to, or handling of, system state [ALR01]. Security is also defined in [ALR04] by "*the concurrent existence of a) availability for authorized users only, b) confidentiality, and c) integrity*", where integrity means the attribute for the absence of unauthorized system state change.

Although security is currently an important issue to attain dependability considering the increasing openness of embedded systems (remote access is currently used), and, as explained before, it deals with several of the six attributes referred previously, it is out of the scope of this work and thus it will not be discussed further in this dissertation.

To develop a dependable system, different means are to be used. They can be grouped in four categories: fault prevention, fault tolerance, fault removal and fault forecasting [ALR04]. More specifically:

- Fault prevention is the means to prevent the occurrence or introduction of faults;
- Fault tolerance is the means to avoid service failure in the presence of faults. Fault tolerance can be accomplished using replication of system components or using specific components to prevent a specific fault (example of such component is a bus guardian);
- Fault removal is the means to reduce the number or severity of the faults;

- Fault forecasting is the means to predict the number, severity and consequence of the faults in the future.

In the scope of our work, and in what concerns dependability, two types of systems are of concern:

- Fail-safe systems: When there are failures in the system, it responds in a way that will cause no harm, or a minimum of harm to the other systems and humans;
- Fail-operational: When there are failures in the system, it responds in a way that will keep the minimum working performance to perform the task continuation.

The concept of fault cannot be dissociated from the concept of error and failure. In the next section these three concepts are explained and discussed.

2.2.3 Fault, error and failure

The concepts of fault (sometimes called defect), error and failure can be found ambiguous in the literature concerning the meaning of fault and failure [Gär99]. These three terms can sometimes be interpreted as the same or can be easily confused. In [Cri91] we can read: "*what one person calls a failure, a second person calls a fault, and a third person might call an error*". This statement demonstrates very well the misconceptions that exist in persons about these three terms used in the fault tolerance scientific community.

Laprie et al. [Lap92] and *Avižienis* [ALR01], define fault, error and failure as threats for dependability. A system failure occurs when the delivered service deviates from the correct service for the system, this is, a failure occurs when the system changes from a correct service delivery to an incorrect service delivery.

Error is what causes the failure [VR01]. This is, an error that reaches the service interface will cause a failure. An activation of a fault results on an error, that reaching the service interface will provoke a failure.

Fault is an abnormal condition that causes a reduction or a loss of the capability of a functional unit to perform a required function.

Even in Portuguese there has been a discussion about the terminology of these three definition. For more information and details about this topic, refer to [VL89] and [Ver96].

2.2.4 Fieldbuses and dependability

The use of fieldbuses combined with the use of sensors and actuators enables the reduction and simplification of the wiring when compared with a centralized topology. Thus, it leads to a system with a lower cost and contributes to the dissemination of the solution. The increasing use of fieldbuses makes the network interfaces and transceivers cheaper and then it is possible to connect more data points in the field. Also, the use of a fieldbus compared

with the use of a centralized topology enables the distribution of the tasks among all the nodes of the network. On the other hand, centralized topologies have a central point of failure. This point of failure is the central node that carries out functions such as the control of the network and control algorithms.

However, fieldbuses introduced new possible failures in components, namely the network and the network interfaces. According to [CCTB03], the error rate of an electrical bus is very low. It can be read in [CCTB03]: "*By its passive electrical nature, a bus has a very low failure rate*". However, there are some faults that can occur and propagate to errors and system failures. The type of errors that can occur in the physical layer are:

1. A continuous transmission to the network due to an internal failure of any node or due to infinite repetitions of the transmissions attemps (babbling idiot behaviour);

2. The non-reception of information by one or by several nodes in the network. This can be caused by:

   - disruption of transmission on the medium causing illegible frames;
   - external aggression such as: cuts, impedance mismatch, loss of line termination, electromagnetic interference.

To make the communication system safe in relation to these faults it is necessary to choose a transmission support adequate to the system and environmental conditions. The dependability of fieldbus systems also depends on the redundancy of the physical layer. There has been some research work analyzing the dependability of fieldbuses such as [KP91], [PC01] and [LY05].

In [KP91] the bus and ring communication topologies for the Delta-4 distributed fault tolerant architecture [Bar93] are evaluated. In this paper, the conclusion is, for the specific case of Delta-4, that the single bus and the single ring topology are equivalent in terms of dependability. In [KP91], for a failure rate of the bus less than $4 \times 10^{-3}/h$, duplication of the medium is more interesting in the case of the bus than in the case of the ring.

In [PC01] the dependability of fieldbus systems in the presence of permanent failures in the bus is analyzed. This analysis includes the study of a redundant mode and also of a degraded mode. Comparing a single bus with a dual bus architecture, the authors conclude that the most important factor is a coverage factor (the coverage factor is modelled as a probability of automatic recovery from a node fault[1]). The coverage factor considers the existence of two types of failures: a failure that does not lead to fieldbus failure, due to a system recovery and a failure that leads to a fieldbus failure. The authors also conclude that just in case of a node coverage factor greater than 0.9 the bus duplication leads to an

[1]Node element fault, not a system failure.

appreciable increase on dependability. This is because the failure rate of the bus is low, the dependability is mainly conditioned by the nodes failure rate and by the coverage factor.

In [LY05] the authors compare a single CAN bus architecture with a dual CAN bus architecture in terms of reliability and stability for a small aircraft. The authors conclude that, "*from the simulations and implementations, the dual bus architecture enhances the fault tolerance of the system and satisfies, reliability and stability*" [LY05].

In [LY05], *Lin* and *Yen* refer stability as an important parameter, not yet referred in this dissertation. Some disciplines, such as control theory, define stability as property which specifies that, for a given parameter with bounded inputs, the parameter is bounded [Sta85]. However, *Stankovic* defines a more specific notion for stability, please refer to [Sta85] for more details. Applying this definition to embedded distributed systems seems to be reasonable with the necessary adaptations.

2.2.5 Redundancy and fault tolerance

In presence of a failure, one way to promote fault tolerance is to use redundant components [Sch90]. The replication of the components can be done in a hardware level or in the application software level. In this dissertation, the redundancy studied is in the hardware, more specifically in the network.

Redundancy is "*The use of more elements than necessary to maintain the performance of a system in the event of failure of one or more of the elements*" [LHB03]. According to [LHB03] there are four types of redundancy: diverse, homogeneous, active and passive [LHB03].

- Diverse redundancy is the use of more than one element of different types to provide redundancy. An example of such redundancy is the use of mechanical and electrical brakes in cars;
- Homogeneous redundancy is the use of more than one element of a single type. One example of such is the use of two wires in a CAN bus, enabling differential operation in the absence of failures and single-ended if one of the wires is, by example, cut;
- Active redundancy is the use of more than one element at all times. Active redundancy distributes the load across all the elements and allows an element failure, repair and substitution with minimal interference in the system performance;
- Passive redundancy is the application of the redundant element only when the active element fails. Example of such redundancy is the use of a spare tire in a car.

In our opinion the first two types of redundancy (diverse and homogeneous) refer to the redundant elements themselves and the last two (active and passive) to the operation of the redundant elements. This is, we are defining two dimensions of redundancy: on the elements (diverse or homogeneous) and regarding type of operation (active or passive).

In the previous paragraphs, the redundancy is seen in a general point of view. However the same definitions can be applied and used in the fieldbuses domain. One example of diverse redundancy is the use of more than one fieldbus type to send the same information.

Concerning the type of operation, other authors [DSS98, WPS+00] define more types besides the two referred before: semi-active and semi-passive. *Défago et al.* [DSS98] stated that a system uses semi-passive redundancy, when there is a primary server and backups. Concerning databases, in semi-passive replication, the selection of the server that processes the request from the client is based on a rotating coordinator paradigm. The same authors [DSS98], define semi-active redundancy for databases. In semi-active replication, all the servers process the request and the primary master applies the changes in all backups. Applying these definitions (semi-passive replication and semi-active replication) to bus redundancy seams to be inadequate, however it can be applied to node redundancy.

Regarding fieldbuses, a passive redundancy is considered when the additional buses are in standby and one of them is activated in the case of a failure in the primary bus. Conversely, when using active redundancy, all the available buses are used to send redundant data.

Each type of redundancy is more adequate for each type of system. In that way, if the cause of the failure cannot be anticipated, the most suitable redundancy type is diverse redundancy. On the other hand, if the probable cause of failure can be anticipated, the most suitable redundancy is the homogeneous redundancy. Active redundancy is used in critical systems that must maintain continuous operation in a case of a system element failure. Passive redundancy is used in elements of a system that are noncritical or in systems where performance interruption is tolerable. In table 2.1 this is systematize.

System/failure	Redundancy
Failure can be antecipated	Homogeneous
Failure cannot be antecipated	Diverse
System must maitain operation	Active
Performance interruption tolerable	Passive

Table 2.1: Type of redundancy

As it can be seen in figure 2.2, for the network there are mainly four topologies (or hybrid solutions among them) that can be used: star, mesh, ring and bus.

Star topology implies that there is a central point that controls all the system network. This central node can be replicated and, also, the links to the nodes can be replicated. Moreover, all the processing must be done by the central point, which must be a system with an appreciable processing power. This can be costly due to the need to replicate this element.

In case of a mesh network topology, all nodes connect to all others. In this network topology, if a link is broken, the communication can be done by other links.

The ring topology connects all the nodes forming a ring (the communication network forms a circular architecture). This kind of topology provides some redundancy, because if

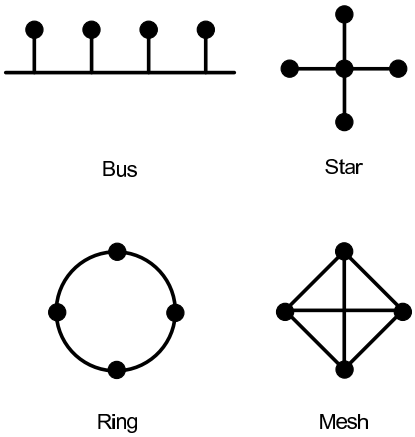


Figure 2.2: Network topologies

a partition is made in the network, the system can work in a degraded mode since all nodes still have a communication path with the others. The degraded mode is achieved because no electrical termination is done in the cut wires. In the case of the partition, the network topology becomes a bus topology without termination.

In the bus topology all the nodes are connected to the same physical wires. Conversely to star topology, where a faulty node can be isolated, in bus topology a node fault can affect all the system. In case of a partition, the bus will be divided into two buses. For the bus topology, the redundancy of the bus can contribute to the system availability and reliability. However it will not contribute to the system safety because it does not avoid catastrophic consequence to persons and environment [CCTB03].

In the case of redundancy, all replicas must be consistent. This topic will be discussed further.

2.2.6 Replica consistency

The issue of replication is of interest in several domains, in particular in databases and in distributed systems. One of the main issues is achieving consistency among replicas which is fundamental to guarantee a correct operation of the system when one of the replicas replaces the unit that failed [PSL80, PLS82]. In the database area, examples of discussion of techniques to guarantee consistency can be found in [Bir92], [PB95] and [Mar03]. However, definitions and solutions are conceptually similar for databases and distributed systems [WPS+00].

The replication of a system component brings new problems in what concerns the synchronization of all the replicas. The component replicas must be synchronized in value and in time domains [HWV03]. The literature defines two main consistency criteria for the distributed systems [WPS+00]: linearisability and sequential consistency.

Linearisability is based in real-time dependencies while sequential consistency is only based on the order of the sequential operations performed in individual processes. Thus, li-

nearisability is a stronger criteria than sequential consistency. A detailed comparison between them can be found in [AW94] and [GS96].

As expected, a full synchronization in every instant is hard to achieve. However, a semi-synchronous approach can be made, where the distributed replicas are not synchronized all the time. There are instants where the replicas are not synchronized. But, after a bounded time interval, all the replicas should maintain a coherent view of the system parameters or data [FNTTJ04].

To enforce replica consistency some authors propose a voting system [Mar03]. When using replication, the use of a voting system determines what are the data that will be chosen. In that case, a certain number of votes is given to each node and an operation can only be made if there is a sufficient quorum [Gif79, GMB85].

To deal with the complexity of synchronization, the notion of group and communication primitives was introduced. The group provides a logical addressing mechanism to join together a set of replicas. Communication primitives provide multicast communication. The two main group communication primitives are atomic broadcast and view synchronous broadcast. Atomic broadcast ensures atomicity and order, that is, the data that will be received by all replicas is the same and in the exact same order. *"View synchronous broadcast is a property that specifies that membership information is ordered relatively to the message flow"* [Mar03]. For more details about atomic broadcast and view synchronous broadcast refer to [CASD85], [HT93] and [SS93].

2.3 Buses for industrial automation

2.3.1 Introduction

The use of fieldbuses in industrial automation begun with ModBus from Modicon [Has80a, Has80b] and the Westinghouse Distributed Processing Family (WDPF) from Westinghouse [Mor82, Sch82] because of their functionality and worldwide acceptance [Dec05, Tho05]. Some other networks were already in use in the time of the birth of ModBus and WDPF. However, they are confined to a small number of applications and companies.

After this, several research projects concerning the development of fieldbuses appeared in the 1980s. These projects led to the communication protocols known nowadays. In France the FIP project was started (now known as WorldFIP), in Germany the PROFIBUS and in Denmark the P-NET [Tho05].

2.3.2 WorldFIP

WorldFIP was developed at the early 1980s with the name FIP (Factory Instrumentation Protocol), and later known as WorldFIP and also becoming a French standard. Currently

WorldFIP is a profile of the European fieldbus standard EN-50170 [CEN96]. Other profiles of the EN-50170 are PROFIBUS and P-NET. Also, it is one of the profiles of the IEC International Standard 61158 [IEC00].

WorldFIP is organized according to the Open System Interconnection (OSI) reference model and it only defines the layer 1, layer 2 and layer 7 of the OSI model, *i.e.*, the physical layer, the data link layer and an user interface (application layer).

A WorldFIP network interconnects two types of nodes: Bus Arbitrators (BA) and producer/consumer nodes. The MAC protocol adopts a Producers-Distributor-Consumers (PDC) model [Tho93, ATFV02], where each node can perform these functions simultaneously, but at any given time just one node can perform the bus arbitration [AC98].

A static schedule table (BAT - Bus Arbitrator Table) is present in the Bus Arbitrator (BA) to organize the transmission on the bus. The available bus time is divided into Elementary Cycles (EC) and, at each EC, the BA promotes the corresponding information exchange. Each elementary cycle can have the transmission of more than one producer, and the number of elementary cycles that join together all transmissions is called the macro-cycle.

The bus arbitrator node (BA) broadcasts a question frame with one variable identifier that will be answered by the producer. The answer of the producer will be simultaneously captured by all consuming stations. There will be one producer node and one or more consumer nodes. The consumer(s) of the variable and the bus arbitrator will receive the variable. With this reception, the bus arbitrator can go to the next entry in its internal table and the cycle will begin again.

In table 2.2 is an example of the bus arbitrator table and in figure 2.3 the corresponding macro cycle is presented. Note that each elementary cycle has $5ms$. Note also, the time duration is determined using $2.5Mbps$, a turnaround time of $20\mu s$ and equation 1 of paper [TV01][2].

Variable	Period	Data bytes
A	5	1
B	10	128
C	5	1
D	15	128

Table 2.2: Bus arbitrator table example

In the example of figure 2.3, the remaining available time in each elementary cycle can be used for the aperiodic traffic. This aperiodic traffic is also controlled by the bus arbitrator and also has preliminary stages to perform a variable transmission. The mechanism that supports the aperiodic traffic has three stages, the first where the producer asks the BA to transmit the variable/message (this is done during the periodic window, piggy-backing information in one of its periodic messages identifiers), the second stage, where the bus arbitrator asks the

[2]For more details about the calculation, please refer to [TV01].

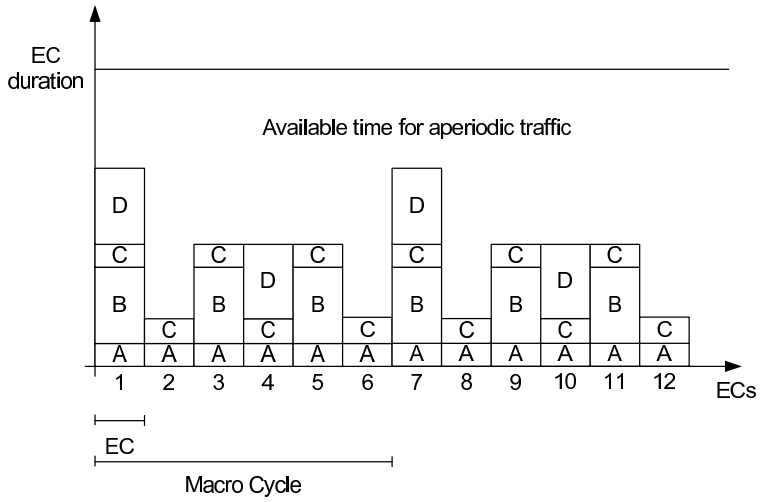


Figure 2.3: Macro cycle example (from [AC98])

producer of the variable/message to transmit the request and the third stage, where the node transmits the variable/message.

In what concerns the physical layer definition, it is compliant with IEC1158-2 and defines two types of physical medium: twisted pair and optical fibber. For these two media, there are three defined bus speeds:

- S1: $31.25kb/s$ (low speed);
- S2: $1Mb/s$ (high speed);
- S3: $2.5Mbp/s$ (high speed).

Speeds S1 and S3 are only used for special purposes and applications, while speed S2 is the standard speed. Additionally, there is an extra speed definition for the optical fibber medium which is capable of $5Mb/s$.

Concerning the bus media redundancy, it is possible in WorldFIP using two buses [AC98]. All the messages are replicated in both media. Each receiver has the capacity of receiving the first message that is detected. Network management can force stations to transmit or listen in a particular channel. This can be done for reasons of maintenance or after a detection of several errors in one channel. To detect this, network management uses a set of error flags and performance counters [AC98].

Since WorldFIP uses a master-slave architecture (the BA is a single point of failure), the master (called bus arbitrator) is also replicated in order to prevent a failure of the BA in charge. Like stated before, there is only one BA active at a time. However, in the global system, several BAs can coexist. Each BA has an identifier that is used to determine the BA that will become in charge if the active BA fails.

2.3.3 PROFIBUS

PROFIBUS (Process Field Bus) is one of the general purpose fieldbuses included in the European Standard, EN 50170 [CEN96] specification. It was proposed in 1989 in Germany by a consortium of Universities and factory automation equipment manufacturers.

PROFIBUS is organized according to the Open System Interconnection (OSI) reference model and it only defines the layer 1, layer 2 and layer 7 of the OSI model, *i.e.*, the physical layer, the data link layer and an user interface (application layer).

There are several PROFIBUS bus protocol definitions: PROFIBUS FMS, PROFIBUS DP, PROFIBUS PA and PROFINET. PROFIBUS DP (DP stands for Decentralized Periphery) is used to connect distributed I/O devices via a fast serial data link with a central controller. PROFIBUS DP supports both analog and digital signals and communicates at speeds from $9.6kbps$ to $12Mbps$ over distances from $100m$ to $1200m$.

PROFIBUS FMS (FMS stands for Fieldbus Message Communications) is a control bus for communications among PLC (Programmable Logic Controller) systems. It is complex, and thus is often replaced by a simpler protocol, the PROFIBUS DP.

PROFIBUS PA (PA stands for Process Automation), is used to monitor measuring equipment via a control system, specially in hazardous environment. It communicates at $31.25kbps$ with a maximum distance of $1900m$ per segment. This protocol is derived from the PROFIBUS DP.

PROFINET is a protocol to allow PROFIBUS communications over Ethernet networks. PROFINET will be discussed further in this chapter.

PROFIBUS DP is the widely used protocol from PROFIBUS family, and thus it will be discussed. The PROFIBUS MAC is based on a token-passing mechanism used by masters stations to grant bus access [CMTV02]. The PROFIBUS token-passing procedure is a simplified version of a Time Token protocol (TT) [TV99]. It is based on a master-slave scheme for the master stations to communicate with the slave stations. These MAC mechanisms are implemented in layer 2 of the OSI model which is called, in the original PROFIBUS definition Fieldbus Data Link (FDL).

PROFIBUS uses a master-slave scheme where several masters can coexist. A PROFIBUS slave could be any peripheral device (*e.g.* measurement sensor device) which processes information and sends output to the master. The slave is a passive station because it does not have autonomous bus access rights, it just answers to the master requests or acknowledges master messages.

On the other hand, a PROFIBUS master is an active node since it has autonomous bus access rights. PROFIBUS defines two types of masters: class 1 and class 2 masters. Class 1 masters handle the normal communications and exchange of data with the slaves assigned to it. A class 2 master is a special purpose master used for slave commissioning, maintenance and diagnosis. Some masters can support class 1 and class 2 functionalities. A master to

master communication can be held between two PROFIBUS systems using a gateway.

PROFIBUS DP operates using a cyclic transfer of data between the master and slaves on a RS485 serial network. Each slave has an assigned master. The master can just write information to its slaves and can read information from every slave. The master grants bus access to its slaves in a cyclic way.

PROFIBUS exhibits a deterministic temporal behaviour due to its cyclical operation mode. Please refer to *Cavalieri et al.* [CMTV02], for a worst case response time analysis.

The PROFIBUS frame is composed by different octets. The octets are transmitted in a asynchronous way with start, stop and even parity bit. According to *Felser* [Fel06], PROFIBUS DP installations show also unexpected transmission errors due to the incorrect cabling, shielding, grounding and termination of the bus (these problems represent more than 80% of the errors).

Concerning dependability, PROFIBUS supports replicated buses with a maximum of two communication channels connected to a separated bus interface in each node [LS95]. The data is transmitted simultaneously in both communication channels and a bus selector switch located in the node selects the channel from which the node will receive the information. There are pre-defined criteria to decide when to switch to an alternative receiver. Figure 2.4 depicts the general architecture of a PROFIBUS node with media redundancy.

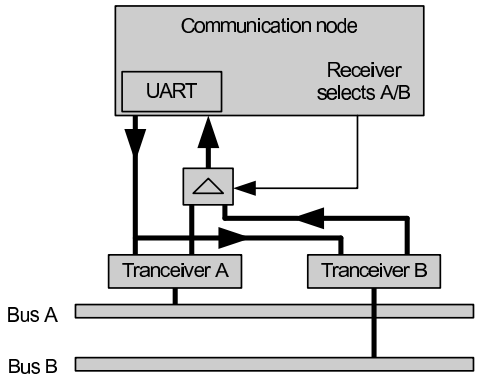


Figure 2.4: PROFIBUS node architecture with redundant bus(from [LS95])

In what concerns the node redundancy, it is possible to implement, as stated in the PROFIBUS web-page [PRO09]. Namely, the master node can be replicated. Example of such replicated nodes can be found in [Com09].

2.3.4 P-NET

P-NET is part of the European Standard, EN 50170 [CEN96] and International Fieldbus Standard IEC 61158 type 4 [Fel02, IEC07].

As stated in the standard, the P-NET fieldbus was designed to connected distributed process components replacing the wires imposed by a centralized control system. Examples

of these components are sensors, actuators and controllers [CEN96].

The electrical specifications are based on the RS-485 standard using a shielded twisted pair cable. This specification allows a cable length of 1200 meters without repeaters. The data is sent in NRZ (Non-Return-to-Zero) in an asynchronous way. The maximum bitrate is 76.8*kbps* using one start bit, one stop bit, eight data bits and one address/data bit.

Each system can have more than one bus segment coupled with a multi-port master (acting as a gateway between the two buses segments), where each bus can handle up to 125 devices. In each segment up to 32 masters can coexist. The communication is based on a master-slave structure where the master sends a request and the addressed slave returns an immediate response.

The system uses a virtual token passing scheme. In this strategy, the master who has the token is able to transmit and the token is passed to the next master without an explicit exchange of a message. This is done using local counters that are incremented in each transmission. When the counter reaches the master address, the master may transmit. This virtual toking passing scheme is an interesting solution to arbitrate the access to the medium and has also been proposed for shared Ethernet [CFP03, Car08].

According to the P-NET specification [CEN96], the ability to use a multi-net system provides "*a natural redundancy which makes the total plant installation very robust with respect to errors*". According to the same specifications [CEN96], the redundancy is provided, because the errors cannot be propagated from one bus segment to the other bus segments.

A company named Proces-Data develops a module to provide network redundancy using P-NET. This module [PD09] connects to two P-NET networks. The data is sent over the two ports (each port connect to one P-NET network) and is received over the two ports. If a short-circuit or a cable break is detected in one network, the corresponding port is automatically switched off and all the communications are done in the other port.

2.4 Buses for embedded applications

2.4.1 Introduction

Nowadays, embedded applications are increasingly using more microcontrollers and fieldbuses to connect their systems. For example modern cars use several microcontrollers, up to 50 in some high-end models, interconnected by fieldbuses. In some cases, more than one type of fieldbus are used. In vehicular applications as well as in other safety critical applications, *e.g.*, avionics, physical redundancy plays an important role. This section presents the main fieldbuses used in embedded applications with a special focus on the bus media redundancy aspect.

2.4.2 TTP

The Time Triggered Protocol (TTP/C)[3] has been developed at the Technical University of Vienna by *Kopetz et al.* [KG94] intended to be used in safety critical real-time systems. The goals of the TTP/C project were to achieve safety, composability and flexibility in the communications [KG94].

In what concerns the network topology, TTP/C does not define any specific topology. *Kopetz* and *Grünsteidl* in [KG94] say "*The communications channel is a passive LAN, e.g., a broadcast bus, that transports one message at a time*". The Time Triggered Architecture (TTA) [KB03] defines the entire communications and computer architectures to be used in TTP/C.

The basic building block of a TTA architecture is the node. A node is a self-contained unit composed by a processor with memory, an input-output subsystem, a communication controller, an operating system and the application software. The communication controller connected to the replicated communication channels forms the so called cluster [KB03]. Clusters can be connected by gateway nodes which restrict the view of one cluster in order to reduce the complexity of the system. In other words, the cluster is a way of creating physical segments in the system network.

The Time Triggered Architecture defines two different topologies for a cluster, the TTA-bus and the TTA-star. In the TTA-bus the physical network consists of two replicated passive buses where the information is sent in both channels at the same time. In the nodes there is one communication controller with two bus guardians (one for each broadcast channel). Ideally, the bus guardians operate in an independent way, each one with its own clock, power supply, distributed clock synchronization and local copy of the message's schedule. Furthermore, the bus guardians should be at a physical distance sufficient to protect the node from spatial proximity faults. This leads to an implementation of the node and the two bus guardians in separated chips. However, the implementation of such system is expensive and the authors in their prototypes put these three systems in the same chip. This only guarantee the enforcement of the fail silent state of the application in case of violation of the fault hypothesis defined.

The TTA-star can tolerate arbitrary node faults, *e.g.* byzantine faults [KB03]. The byzantine fault was first described by *Pease et al.* as a agreement problem in a military environment [PLS82]. Nowadays is considered that a "*byzantine faulty process may behave arbitrarily*" [ZV08].

In the TTA-star network architecture, the nodes are connected to a duplicated star coupler. This means, the nodes have two network interfaces each one connecting to the star coupler, resulting on a duplicated star. The bus guardians have been moved to the star coupler [BKS03] because, according to the authors, there are some advantages, namely: the

[3] C stands for class C applications of Society Automotive Engineers, page 20.272 of [SAE92].

guardians are fully independent and located at a physical distance of the nodes, the algorithms of the guardians can be extended to provide additional monitoring services, if the guardians reshape the physical signals, the architecture becomes resilient to arbitrary node faults and the point-to-point links have better electromagnetic interference characteristics than a bus.

The TTP/C provides the fault tolerant internal synchronization of the local clocks to generate a global clock of known precision. The receiver knows a priori the sending time of a specific frame. Moreover, the receiver knows the time of its reception, thus it knows the difference between these two occurrences [KG93]. Thus, it knows the difference between the sender and receiver clocks.

TTP/C supports redundancy at the node level defining Fault-Tolerant Units (FTU). Fault-tolerant units are composed by two or more computational systems act as redundant nodes [Ins97]. *Pimentel* and *Sacristan* [PS01] propose an adaptation on this mechanism where a node can belong to several FTUs at the same time. Authors claim this allows a higher level of dependability keeping the same number of replicated nodes.

Besides supporting message replication in the space domain (as explained, using two buses), TTP/C also supports the message replication in the time domain, by sending the messages twice on each bus.

2.4.3 FlexRay

FlexRay is a protocol defined by the FlexRay consortium [Fle02]. This consortium started its activities in 2000 with the founding companies BMW, Daimler-Chrysler (now Daimler AG), Philips and Motorola. The consortium has grown and today it includes the largest automotive manufacturers together with the leading microcontrollers manufacturers: Volkswagen, Toyota, General Motors, Ford, Honda, Nissan, Renault, Bosch, Freescale and many others. The target application domains of the FlexRay protocol is the automotive industry where the most used protocol is CAN.

The FlexRay consortium published a set of documents to specify the FlexRay protocol. These documents are the Protocol Specifications [Fle04c], the Physical Layer Specifications [Fle04b] and the Bus Guardian Specifications [Fle04a].

In what concerns fault tolerance, FlexRay supports the concept of scalable fault tolerance, *i.e.*, FlexRay can be used in systems requiring low fault tolerance capabilities and also in systems with strict fault tolerance requirements. The topology flexibility, the fault tolerant clock synchronization and the conceptual separation of the functional and structural domain supports the scalable fault tolerance model of FlexRay [MT06].

The network topology can be a single bus or star, or, for safety critical applications, can be a replicated bus or a replicated star topology. The network can only have three cascade stars between two arbitrary nodes. Systems with hybrid topologies are possible, since FlexRay can

accommodate a mixture of stars with bus. Some results [Fle02] show that the bus running at full bitrate ($10Mbit/s$) is limited to 8 stubs of a maximum of $25cm$ each. In figure 2.5 all type of possible bus topologies are presented.

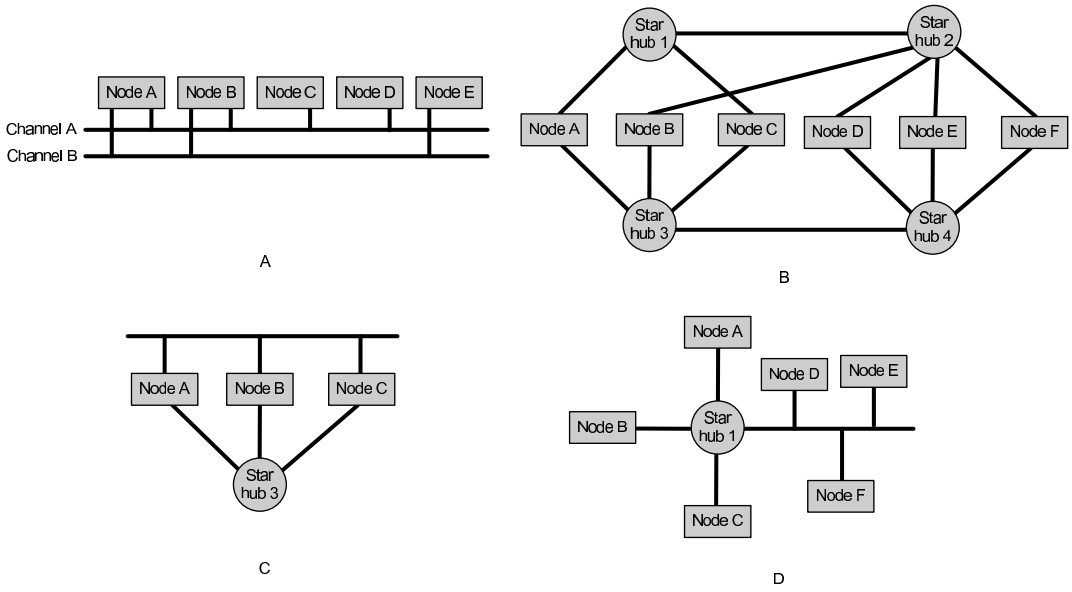


Figure 2.5: FlexRay network topologies

Figure 2.5A depicts the passive bus where some nodes connect to just one channel, while others connect to both channels. Figure 2.5B presents a duplicated active star. In this topology it is also possible to have some nodes which connect just to one channel, however this situation is not represented in the figure. The star can have only one channel instead of the two presented. Figure 2.5C and figure 2.5D present hybrid topologies. Figure 2.5C is a dual channel topology where one channel is a bus and the other is an active star whence figure 2.5D has only one channel having some nodes connected to a star and the others connected to a bus. As it can be seen on the these examples, it is possible to build many network topologies using FlexRay, however all are limited to use two channels.

The medium access control is based on a TDMA (Time Division Multiple Access) scheme where the communication cycle is the fundamental element of the media access scheme. Figure 2.6 depicts the communication cycle.

The communication cycle (see figure 2.6) is composed by the static segment, the dynamic segment, the Symbol Window (SW) and the Network Idle Time (NIT). The static segment is dedicated to the time-triggered data while the dynamic segment is dedicated to the event-triggered data. The symbol window is a communication period in which a symbol can be transmitted in the network. This symbol can be for example, a runtime testing command or a command to wake up an active star. The network idle time is a period free of communication in the channels, and serves to delimit two consecutive communication cycles.

In the static segment a static TDMA scheme is used to arbitrate the transmissions. The

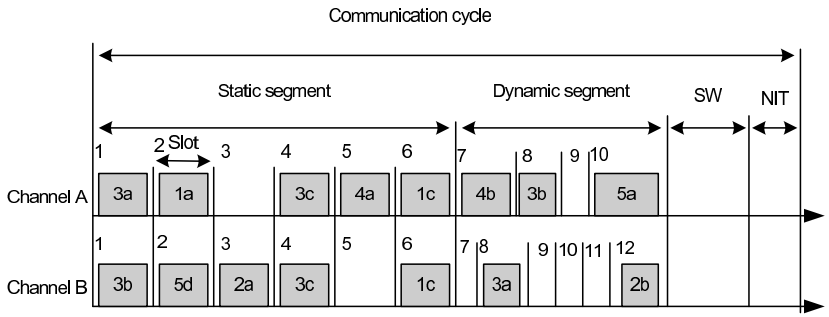


Figure 2.6: FlexRay communication cycle (from [MT06])

number of time slots of the static segment, which have fixed duration, is configured at pre-runtime [Fle04c]. Within the static segment, only one node is allowed to transmit in each communication slot [Fle04c].

In what concerns the dynamic segment, it is used a dynamic mini-slotting scheme to arbitrate transmissions. In the dynamic segment the duration of the slots may vary in order to accommodate the frames of different length (see slot 11 e 12 of channel B in figure 2.6). As it can be seen in figure 2.6, the slot counters in the static part increment at the same time in both channels, while these counters are independent in the dynamic part.

In the static segment, nodes connected to both channels must transmit each frame in both channels at the same time. However, if a node just connects to one channel, the node just transmits information in the channel it is connected to, leading to different data streams in both channels (see figure 2.6, frame 5d and 1a). If two nodes are connected to two different channels they may share the same slot in the static part, leading to a better usage of the available bandwidth.

Moreover, in the dynamic part the traffic in both channels can be different (see figure 2.6). Thus, as stated before, the total available bandwidth of FlexRay corresponds to transmitting different data in both channels.

The FlexRay frame can have a maximum of 262 bytes. Three bytes are dedicated to the CRC (Cyclic Redundancy Check) while five are dedicated to the frame header. Thus, the payload can have at maximum of 254 bytes (the minimum is 0 bytes) leading to a minimum overhead of 3.05% (payload equal to 254 bytes) and a maximum overhead of 88.88% (payload of 1 byte). Note that, although a frame with 0 bytes of payload is possible, it is not commonly used.

A detailed comparison between FlexRay and TTP (presented in section 2.4.2) can be found in [Kop01].

2.4.4 MIL-STD-1553B

The MIL-STD-1553B is an American military standard published by the American Department of Defense [Mil06].

The nodes in a MIL-STD-1553B network can be of one of three types: Bus Controllers (BC), Bus Monitors (BM) or Remote Terminals (RT). The bus controller is responsible for the initialization of all transmissions in the data bus. The bus monitor is responsible for monitoring the bus and collecting information for offline analysis and record. The bus monitor is a passive node, thus it cannot transmit to the bus. The remote terminals are all the other nodes that are neither bus monitor nor bus monitors. Each remote terminal has an unique identifier that identifies it in the network (*e.g.* for the addressing of the bus controller). Each remote terminal can be an interface between the bus and an attached subsystem or can be a bridge between two MIL-STD-1553B buses.

MIL-STD-1553B describes an optional data bus redundancy [DOD78, SL02]. In this standard, the active bus is controlled by the bus controller node (see figure 2.7). Only one bus is active at a time, and the bus controller is responsible for the initialization and management of the communications. Like in the cases of TTP and PROFIBUS, the redundant buses cannot be used to improve the available bandwidth of the system, because only one bus is in use at any given time.

In MIL-STD-1553B there is also the possibility to have three or four data buses, leading to tri or quad redundant buses. These configurations with 3 or 4 buses are normally used in avionics [Con09]. However, as explained before, just one bus is active at any given moment.

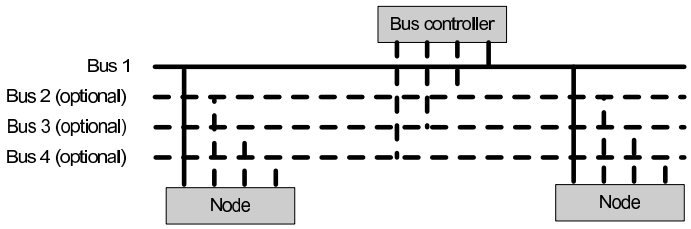


Figure 2.7: MIL-STD-1553B system architecture

The bus controller is a single point of failure of the system. In the literature the information about the replication of this node is rare. In [BPGN05] the authors write: "*Most avionics applications of this databus require a duplicated, redundant bus cable and bus controller to ensure continued system operation in case of a single bus or controller failure*". However, it is not explicit if the standard supports the node replication.

2.5 Ethernet based solutions

Some Ethernet based solutions are currently in use in the automation field. This was possible because the Ethernet is a well established standard with many hardware implementations which are cheap and easy to obtain and install [MFF09]. In addition, the bandwidth provided by Ethernet is suitable for most applications. Ethernet throughput can vary from $10Mbps$ to $1Gbps$ [DHLV96], and more recently $10Gbps$ [MH00].

According [Han05] and [KD06], protocols based on Ethernet targeting industrial automation also require redundancy. In this section these protocols will be briefly discussed with emphasis to the more relevant that uses redundancy in the physical layer.

Ethernet for industrial automation can be based on existing protocols. Examples of such kind of solutions are PROFINET IO (based on PROFIBUS), Time Triggered Ethernet (TTE, based on TTP) and Ethernet Powerlink (based on CANOpen).

PROFINET IO is a standard of PROFIBUS that enables the use of Ethernet in the industrial automation. A typical application for a PROFINET IO system is to allow a PLC (Programmable Logic Controller) to control decentralized field devices [Fel04]. Media redundancy for PROFINET IO is possible and is based on the Spanning-Tree Protocol (the Spanning-Tree Protocol, STP, is a protocol to prevent loops in Ethernet solutions and is also a redundancy protocol, standard 802.1w [IEE09]) and Rapid Spanning-Tree Protocol (RSTP) [Fel08]. As stated in [Hen09], node replication is planned to be included in PROFINET IO.

TTEthernet (TTE) expands the classical Ethernet with services to meet time-trigger constraints. The authors of TTEthernet stand that TTE can be considered the unification of the best properties of standard Ethernet and TTP/C [KAGS05]. A switch with real-time capabilities has been designed to be used in TTEthernet [SGAK06]. With this switch it is also possible to have a redundant medium for safety critical TTEthernet [SGAK06]. The switch is the central point of the network and is replicated, like all the links to the nodes.

Regarding switched Ethernet and targeting FTT-Ethernet (Flexible Time-Trigger over Ethernet [APF02]), *Santos et al.* [SMO+08] are developing a network switch with real-time capabilities.

Ethernet Powerlink is a protocol based on Ethernet and on the CANopen [CAN00] profile [SV07]. Redundancy in Ethernet Powerlink is also possible. *Limal et al.* present a formal verification of media redundancy in Ethernet Powerlink [LPDL07]. Also, the IXXAT company develops a Ethernet Powerlink stack solution for a redundant Ethernet Powerlink controller [CAN09]. In this solution, the Powerlink Managing node (the managing node is a network manager, that checks all the communications to avoid collisions) can be replicated. This solution can be used together with a dual channel communication network resulting on a replicated network with a replicated managing node.

There are other commercial and proprietary solutions for Ethernet. Examples of such type of protocols are EtherChannel (from Cisco Systems, Inc) and DualNet from Nvidia [NVi06]. Another solution, EtherCAT has been originally developed for commercial purposes (developed by the company Beckoff). However, nowadays EtherCAT is an open solution.

Cisco EtherChannel cannot be viewed as a replacement for fieldbuses. It is designed to the Ethernet market, more specifically for campus Ethernet connection. However, it has some similarities with the solution presented in this dissertation. The Cisco EtherChannel solution provides a transparent way of increasing the redundancy and the bandwidth in Ethernet installations. The EtherChannel can aggregate up to eight links providing up to $800Mbps$, $8Gbps$ or $80Gbps$ [Cis03, Huc07] (aggregation of eight $100Mbps$, $1Gbps$ or $10Gbps$ respectively). The supplementary buses can also be used to provide redundancy [Cis03]. EtherChannel does not require any other protocol to maintain the topology state [Cis03] (like Spanning-Tree Protocol, STP, standard 802.1w [IEE09])

Also, there is an IEEE standard that defines link aggregation for the Ethernet: 802.3ad [FDH+07]. This standard defines the aggregation of N Ethernet channels in one link. In the data layer of the OSI reference model, there is a link aggregation sublayer that aggregates all the underlying Ethernet channels.

DualNet from NVidia [NVi06] is a technology for home and small office personal computers (PCs). It provides one chip with two Ethernet controllers with capabilities of teaming and fail-over, among others, not so important for this dissertation: refer to [NVi06] for more details. Teaming allows the two Ethernet ports to be used in parallel to increase the overall link speed of the Ethernet connection. The Fail-over capability allows the chip to switch to the standby port if the active port fails or is disconnected.

EtherCAT (Ethernet for Control Automation Technology) is an open standard based on Ethernet developed by Beckoff to interconnect automation systems. This protocol is based on the principle of the insertion and extraction of the data. The frames are modified on-the-fly by the nodes. The topology can be line, tree, star or ring. The fault tolerance is only obtained in the ring topology [Eth09]. In that way, the system can tolerate the failure of one node or the partition of the ring (if it occurs, the ring becomes a bus).

2.6 Controller Area Network (CAN)

2.6.1 Introduction

Controller Area Network (CAN) is one of the most used fieldbuses for embedded application. It has many application domains ranging from automotive to avionics or industrial machines. The CAN fieldbus has physical redundancy already built-in, since it uses two wires with a differential voltage communication network. Also, some research has been made in the past to add dependability to the CAN fieldbus.

In this subsection the CAN fieldbus will be presented and some important research work about the redundancy of the physical bus will be addressed. There will also be a special focus on the physical layer of the CAN bus, since it is very relevant for this work.

2.6.2 CAN Basics

Physical layer

The CAN ISO standard [ISO93] defines the physical signalling, the synchronization and the bit timing of CAN. Concerning the bit timing of CAN, it is divided in four phases (see figure 2.8 and refer to [HB99]):

- Synchronization Segment (SYNC in figure 2.8), to synchronize the various nodes;
- Propagation Time Segment (PRP in figure 2.8), used to accommodate the signal propagation delay across the bus line and through the bus nodes electronic interface elements. This segment can have different lengths in different nodes, normally configured through the CAN controller;
- Phase Buffer Segment 1 (PH1 in figure 2.8), serving to accommodate the edge phase errors. At the end of this segment, the node will sample the bit. This segment is used in conjunction with the phase buffer segment 2;
- Phase Buffer Segment 2 (PH2 in figure 2.8), also used to compensate the edge phase errors. The length of this segment is programmable, but it has to be at least as long as the information processing time and may not be more than the length of phase buffer segment 1. The information processing time begins at the sampling point and is reserved for the determination of the subsequent bit level.

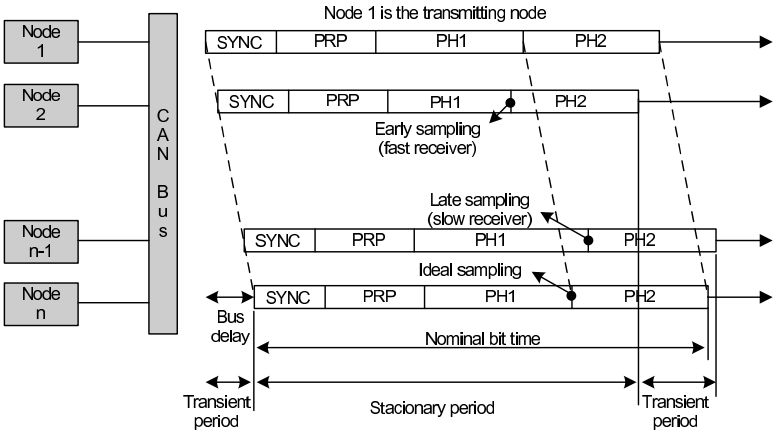


Figure 2.8: CAN bit timing (from [RVA99])

The sum of the four segments presented above is the bit time and, of course, leads to the corresponding bitrate of the bus.

In what concerns the bitrate and the bus length, ISO defines:

- ISO 11898-2 High Speed [ISO93]. This is one of the most used standards for CAN systems. In this physical layer standard the maximum bitrate is $1Mbps$ and the maximum defined bus length is $40m$ for that maximum rate. The bus is composed by two wires with differential voltage levels;

- ISO 11898-3 Fault Tolerant [ISO93]. This standard defines data rates up to $125kbps$ with a maximum of 32 nodes in the network. The transceivers which support this standard will switch automatically to one wire signal transmission in case of a wire cutting or shorted to ground or V_{cc};

- SAE J2411 Single Wire [SAE00]. This standard defines a single-wire standard for application requiring up to $33.3kbps$ with a maximum of 32 nodes. The main application area of this standard is the comfort electronics systems in vehicles;

- ISO 11992 Point-to-Point [ISO03]. This standard defines a point-to-point connection for use mainly in vehicles with trailers. The nominal data rate is $125kbps$ with the maximum bus line length of $40m$ [CAN99].

The most popular CAN physical layer is the ISO 11898-2 high speed standard [ISO93] that is available in most of the CAN transceivers. The mechanisms proposed in this thesis are based on this standard.

This standard defines differential voltage to transmit the information through the bus. This scheme makes the signal transmission more robust and immune to electromagnetic interference. There are two defined states for the medium, the dominant and the recessive, corresponding to a “binary zero” and a “binary one” respectively. If two nodes try to communicate in the bus at the same time, the dominant bit will overwrite the recessive bit. This bitwise arbitration is used for the message prioritization and it will be discussed further in this dissertation.

Concerning the bit coding, CAN bus uses a NRZ line coding where a logic one is represented by a significant condition and a logic zero is represented by other condition. This coding technique has no timing synchronization information because it does not use transictions in all the bits (as the RZ coding scheme).

Data link layer

The CAN data link layer defines the mechanisms to transmit data from one node to another on the CAN network. This layer defines the Medium Access Control (MAC), the

frame formats, the error detection and handling, and the protocol versions (standard or extended).

The bus arbitration is based on a "wired-AND" of the bits. That is, the dominant bits overwrite the recessive bits, meaning that any node which tries to send a recessive bit will be overridden by other nodes which are transmitting a dominant bit.

The data link layer defines four type of messages to be transmitted in the network: error frames, data frames, remote transmission request frames and overload frames. The most used frames are the error frames and data frames.

Error frames are transmitted by any node upon detecting a bus error. This frame can transport an active error flag, or a passive error flag. See further in this dissertation details about these two kinds of error flags.

Concerning the data frame format, the standard defines two kinds of frames (also known as messages), the base frame and the extended frame. The difference between these two types of frames is the length of the identifier, the base frame having an 11 bits identifier, while the extended frame has a 29 bits identifier. These two types are formally known as CAN 2.0A and CAN 2.0B, respectively. In figure 2.9 the general format of the CAN 2.0A frame is shown.

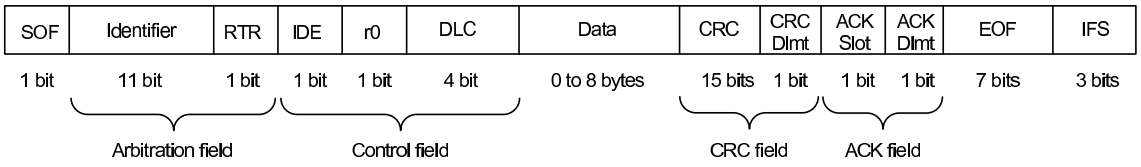


Figure 2.9: CAN 2.0A frame format

The Start of Frame (SOF in figure 2.9) is one dominant bit to signal the beginning of the CAN frame and it is intended to synchronize all the nodes in the network.

The start of frame bit is followed by the arbitration field consisting of 12 bit: the 11 bit identifier (in CAN 2.0A, in CAN 2.0B the identifier has 29 bit), which reflects the contents and the priority of the message, and the Remote Transmission Request bit (RTR in figure 2.9). The remote transmission request bit is used to distinguish a data frame (where RTR is a dominant bit) from a remote frame (where RTR is a recessive bit).

The next field is the control field consisting of six bits. The first bit of this field is called Identifier Extension (IDE) and is a dominant state to specify that the frame is a standard frame. The next bit is reserved and defined as a dominant bit ('r0' in figure 2.9). The remaining four bits of the control field contain the Data Length Code (DLC in figure 2.9) which specifies the number of bytes of data contained in the data field of the message from 0 to 8 bytes. The data being sent follows the DLC Field. The Cyclic Redundancy Field (CRC field in figure 2.9) follows and is used to detect possible transmission errors. The CRC field consists of 15 bits of the redundancy check sequence followed by the CRC delimiter ('CRC

Dlmt' in figure 2.9) bit which is a recessive bit.

The next field is the Acknowledge Field (ACK). During the 'ACK slot' bit, the transmitting node issues a recessive bit. Any node that has received an error free frame acknowledges the correct reception by sending back a dominant bit (regardless the node is configured to accept that specific message or not). The recessive acknowledge delimiter completes the acknowledge slot and cannot be overwritten by a dominant bit ('ACK Dlmt' in figure 2.9).

To complete the frame seven recessive bits follow (EOF in figure 2.9). The Intermission Frame Space (IFS in figure 2.9) is the minimum time in equivalent number of bits separating consecutive messages. Unless another station starts transmitting, the bus remains idle after this.

Like stated before, CAN uses a NRZ coding scheme which could stay for long periods at the same electrical level (e.g. due to a long sequence of the same bit). This can lead to loose of synchronization in nodes. Thus, a bit stuffing technique is used in CAN to prevent nodes from losing synchronization by receiving a long sequence of dominant or recessive bits. The transmitter of a frame adds a stuff bit after five consecutive bits of the same value (the bit added is of opposite value). The receiver of the frame detects the stuff bit and removes it, detecting any possible violation of the stuffing rule. The number of bits introduced by the stuffing mechanism has consequences in determining the worst case transmission time of CAN messages. *Nolte et al.* [NHNP01, NHN02] address this issue and propose a technique to manipulate the message contents in order to reduce the number of stuff bits and thus the uncertainty in the message transmission time.

CAN fault tolerance

The CAN standard defines a fault tolerance scheme for the transmission of data in a two differential wired bus (normally called CAN_H, CAN high and CAN_L, CAN low). In this sense, CAN is able to run in a less robust configuration if a single fault occurs in one wire of the bus. The CAN standard only includes reaction to the following type of failures:

- One wire interruption. An interruption of one of the two wires of the bus is tolerated. This fault is identified in figure 2.10 by 'A' and 'B';
- One wire short-circuit either to power or to ground. The CAN protocol can tolerate one short-circuit to power or to ground in one of the two wires. In figure 2.10 these faults are identified by 'C', 'D' for the short-circuit to the power (V_{cc}) and 'E', 'F' for short-circuit to the ground;
- Two wires short-circuit. The two wires of the circuit are short-circuited one to the other, making the bus to be just one logical wire. Identified by 'G' in figure 2.10.

Upon an electrical fault, the CAN medium interface will switch automatically to single wire operation and switch back to the differential mode when recovered from the fault. The

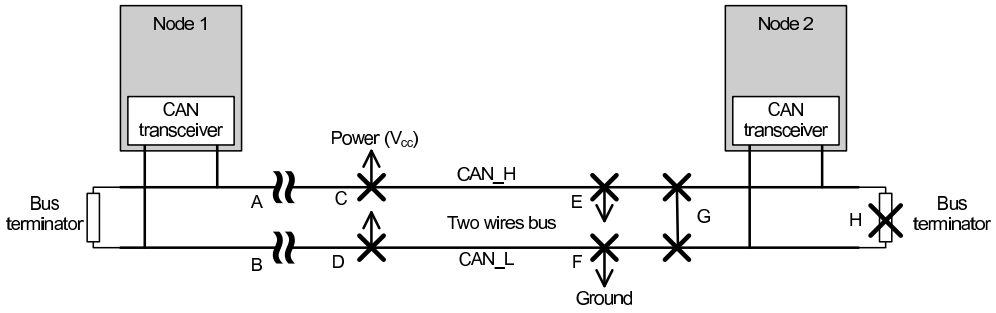


Figure 2.10: Failures in CAN ISO 11898-2 (from [RVA99])

devices which make this commutation are intended for low-speed applications (up to 125$kbps$) with up to 32 nodes [Sem97]. According to [RVA99], there is one exception, a CAN controller from Alcatel [Alc95].

The CAN bus wires are connected together at both ends with a terminator impedance of 120Ω. Resilience to the failure of one terminator (see example 'H' in figure 2.10) can be achieved by taking into account the extra time needed for bus signalling stabilization (this can be done when dimensioning the propagation delay in the bit time calculation [CAN99], see figure 2.8).

In what concerns the detection and signalling of errors, CAN has:

- Cyclic redundancy check to detect corrupted messages. The transmitting node computes the CRC and encapsulates it within the message (see CRC field in figure 2.9). The reception node decodes the message, computes the CRC and compares it with the one conveyed on the message. If they do not match, it means that there has been a CRC error and the reception node will issue an error frame and the erroneous message will be retransmitted;

- Acknowledge errors. The acknowledge bit of the CAN frame is transmitted with a recessive level. If at least one of the receivers correctly decodes the message, it places a dominant level in the ACK field (see figure 2.9). If the transmitter sees a recessive level at the ACK field, it means that the transmitting node is alone in the network or that none of the potential receiving nodes has found the message acknowledgeable. Notice that, any node in the network acknowledge any message in the bus. In case of an acknowledge error no error frames will be generate by the receiver nodes;

- Frame check. This mechanism detects message format violation, *i.e.*, it checks each field of the frame against the fixed format and the frame size correctness. The transmitter also detects a form error if it detects a dominant bit in the fields: CRC delimiter, acknowledge delimiter, end of frame or intermission frame space. If a frame error has occurred, then an error frame is generated and the erroneous frame will be retransmit-

ted;

- Bit monitoring. This mechanism detects bit faults that occur whenever the transmitter sends a dominant bit but senses a recessive bit on the bus line, or vice-versa. If such fault occurs, an error frame is generated and the message will be retransmitted. This mechanism is not applied in the arbitration field and acknowledge field;

- Bit stuffing. A supplementary bit is inserted by the transmitter into the bit stream after five consecutive equal bits between the SOF and CRC delimiter. This leads to a variable frame length even using always the same number of data bytes. As referred, in [NHN03] a study about the bit stuffing insertion is made. If six consecutive bits of the same polarity are detected between start of frame and CRC delimiter, it means that the bit stuffing rule has been violated, an error frame is generated and the message is retransmitted.

If at least one station detects an error it will start the transmission of an error frame as soon as possible aborting the current message transmission. This prevents the other stations from receiving erroneous messages that could lead to inconsistencies. The sender will automatically retransmit the message as soon as possible. However, the message retransmission can be delayed due to the normal arbitration of CAN that favours higher priority messages.

The error frame has a format presented in figure 2.11.

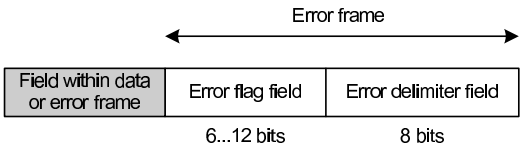


Figure 2.11: Error frame

To prevent a faulty CAN controller to abort all transmissions, including the correct ones, the CAN protocol provides a mechanism to distinguish sporadic errors from permanent errors. This is done locally at each CAN controller by means of error counters and error states. Each CAN controller is in one of three error states, error active, error passive or bus off according to the value of their internal error counters. In figure 2.12 a state diagram of this behaviour is shown.

Nodes have two internal counters: REC (Receive Error Counter) and TEC (Transmit Error Counter). When a node is reset, it goes to the error active mode (most common mode of working). In this mode, the node can send active error frames (composed by dominant bits). If one of these counters reaches 127, the node will switch to error passive mode and the node is only allowed to transmit passive error frames (composed by recessive bits). If the transmit error counter reaches 255 the node will be switched off. The node can go back to the error active state if it monitors 128 occurrences of 11 consecutive recessive bits. In this

case the error counters (TEC and REC) will be reset to zero. Some CAN controllers also issue a warning (hardware interrupt) when the counters go above 96 or the controller goes to bus off.

Notice that the counters, besides increasing, also decrease after a correct transmission (TEC) or a reception (REC). However, the counters are not always increased by 1. For details about the increasing and decreasing of the error counters, please refer to [BOS91].

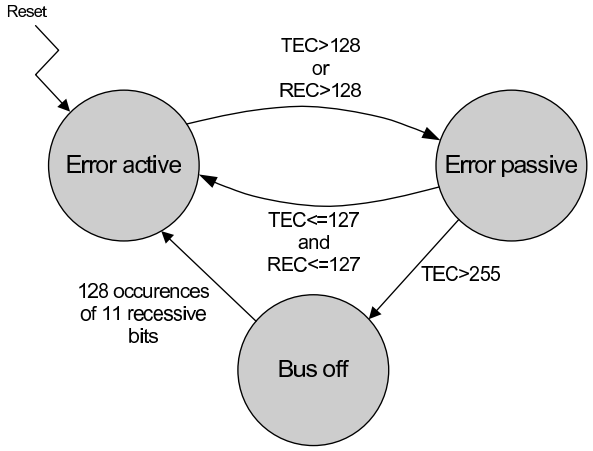


Figure 2.12: CAN error states

2.6.3 CAN protocols with media redundancy

The CAN protocol has a limited fault tolerant scheme in the physical layer. It is based on the dual wires communication scheme as explained before. However, some improvements on this limited fault tolerant solution have been made in the last years. Next, some of these solutions will be presented and briefly discussed.

TTCAN

The ISO organization has standardized an additional layer to the CAN protocol called Time-Triggered Communication on CAN [HMFH00, ISO01].

The TTCAN nodes are fully compatible with the legacy CAN nodes [HMFH00], both in the data link layer and in the physical layer. However, legacy CAN nodes are able to receive TTCAN messages, but not to transmit them. This means that they use the same bus line and transceivers. Existing CAN controllers can receive every TTCAN messages and TTCAN controllers can operate in existing CAN networks, making possible a gradual migration from CAN to TTCAN.

The TTCAN communications is based on one time master that transmits a Reference Message (RM) in a regular basis [HMFH00, MFH+02]. The time between two reference messages is a basic cycle that is dedicated to the transmission of messages [HMFH00].

Muller et al. [MFH+02] proposed a fault tolerant TTCAN architecture based on replicated buses, if there is at least one gateway node that has access to both buses. Authors call the system a coupled TTCAN pair [MFH+02]. Authors consider that a "*fault tolerant TTCAN network is a system of TTCAN buses where each two of them are TTCAN coupled*" [MFH+02]. Thus, figure 2.13 presents one example of a fault tolerant TTCAN network.

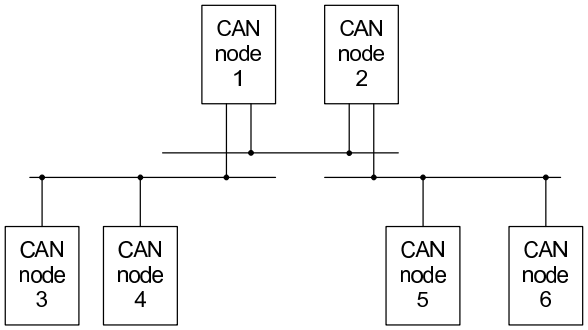


Figure 2.13: TTCAN network example (from [MFH+02])

As it can be seen in figure 2.13 all the three CAN networks depicted are connected via the CAN node 1 and CAN node 2, thus both nodes are considered as TTCAN gateways.

TTCAN network configuration is flexible and other network topologies can be envisaged, as in figure 2.14. In the example depicted in figure 2.14 there is one CAN network (CAN A) acting as gateway between all the CAN nodes. For example, this bus provides communication between node 5 and node 7 (through node 2 and node 3).

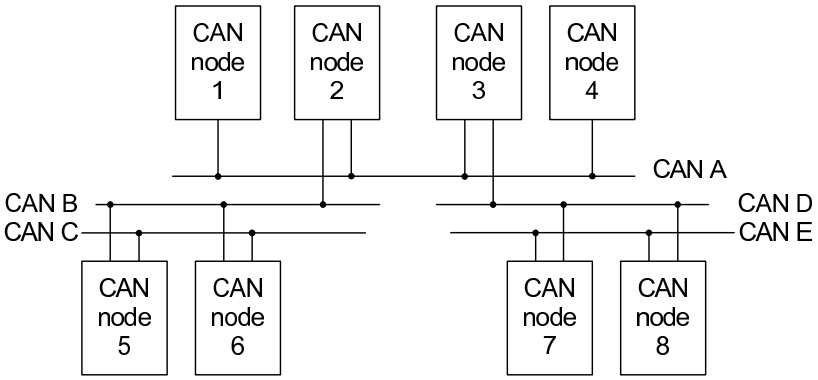


Figure 2.14: Fault tolerant TTCAN network example (from [MFH+02])

The management of the redundant transmission media is assigned to a higher layer protocol, *e.g.*, FTcom or OSEKtime [MFH+02]. In this way, the communication system does not inhibit the transmission of the same message in different buses, or different messages into different buses.

In a TTCAN network the synchronization of the nodes is maintained by the reference messages issued by the time master node. In the TTCAN network up to 8 time masters can coexist at the same time [NNFSL08].

FlexCAN

According to *Pimentel* and *Fonseca* [PF04], FlexCAN is a flexible architecture for highly dependable embedded application. This architecture is associated with the SafeCAN protocol. FlexCAN deals with control systems that use three types of devices: sensors, controllers and actuators. Each of this type of devices corresponds to a *safeware* type. *Safeware* is the application that needs to be defined and designed (*safeware* stands for safe software). These applications are located on the top of a safety layer. Refer to figure 2.15 for the software layers running at each node.

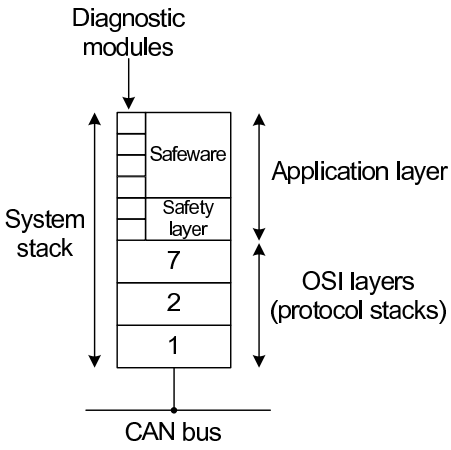


Figure 2.15: FlexCAN node and OSI layers (from [PF04])

FlexCAN is able to support more than one bus. Unlike other protocols (*e.g.* Columbus Egg Idea, presented further on this section), FlexCAN supports more than two buses, more precisely it supports as many buses as the microcontroller can support [PF04]. However, the additional buses cannot be used to improve the total available bandwidth of the system.

According to *Pimentel* and *Fonseca* [PF04], FlexCAN offers four important groups of properties: reliability, availability, safety and security. Reliability and availability are provided by a well known strategy based on the replication of the nodes and also on the replication of the communication channel. The SafeCAN protocol is responsible for the management of the replicated components. Figure 2.16 presents a FlexCAN system with three buses. Notice that it is not required that all the nodes are connected to all the buses.

As it can be seen in figure 2.16, each node can have internal replicas forming a structure called FTU (Fault Tolerant Unit). The network manager does not produce any traffic for the network, being a passive node. However, it can use some messages in order to determine the

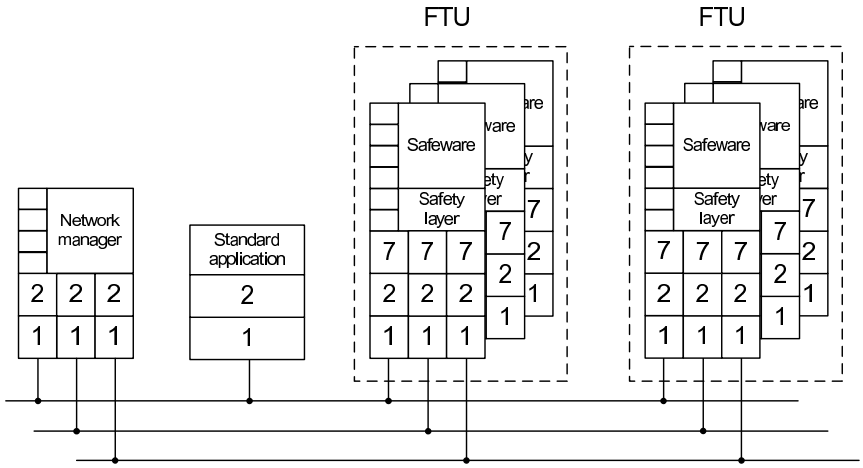


Figure 2.16: FlexCAN architecture (from [PF04])

status of the network components.

Standard off-the-shelf CAN applications running at standard CAN nodes can coexist with nodes running the SafeCAN protocol, as depicted in figure 2.16.

RedCAN

RedCAN is a CAN based protocol that requires specialized hardware. The concept relies on a ring instead of a bus to increase the dependability of the CAN protocol [SOJT04]. The ring is divided into several sections interconnected by the RedCAN module. The RedCAN module has the ability to isolate one section of the CAN ring.

This hardware can connect one node to the bus, disconnect it or do a partition on the bus. This is done using commutation switches that take about 5 to 10ms to switch [SOJT04]. In figure 2.17 the architecture of the RedCAN node is presented.

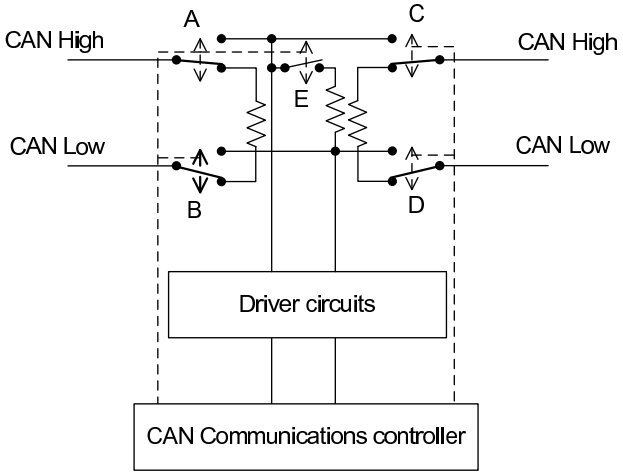


Figure 2.17: RedCAN module (from [SOJT04])

The CAN communications controller connects or disconnects the CAN controller to the network and it can also isolate a bus segment changing the state of the existing switches. As it can be seen in figure 2.17, the switches 'A', 'B', 'C' and 'D' can isolate the node from the bus. These four switches also terminate the bus with the corresponding terminator resistor. Switch 'E', if closed, terminates the bus, however it does not isolate the node from it.

The switches change their state responding to an order from the king of the network. This king is a special node which has the knowledge of the entire network. The nodes that have the RedCAN module are called cities. The communications between the king and the cities are made using CAN Kingdom. For more details about CAN Kingdom, please refer to [Fre95] and [SGN02].

Columbus Egg Idea

The Columbus Egg Idea is a media redundancy solution for CAN [RVA99]. The authors use more than one bus to send the same data over two different physical CAN channels. The information transmitted by the CAN controller is replicated in both CAN channels. At the reception side, the information received in both channels is "ANDed" together (for more details, look at figure 2.18). This strategy is coherent with the strategy used by the CAN arbitration, since the dominant bit (represented by a logic zero) overlaps the recessive bit (represented by a logic one).

All the necessary operations are performed by the hardware, thus the software does not have any knowledge about them. This is an advantage since no software overhead is necessary.

Both buses are wired together via an "AND" gate and thus are both logically and electrically the same bus. In this way, both buses are used to send the same data and, consequently, CAN bandwidth remains unchanged. This means that the total bandwidth available in the system is equal to one system using just one CAN bus. Figure 2.18 presents the structure of the node.

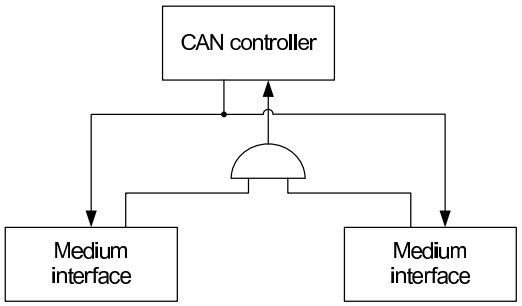


Figure 2.18: Columbus Egg Idea physical layer (from [RVA99])

The medium interfaces are connected to the redundant physical buses. As it can be seen in figure 2.18, the data transmitted by the node is replicated on both buses.

More recently, this system is applied to a more complete system called CANELy (CAN Enhanced Layer). CANELy is a framework able to enhance with some simple mechanisms the off-the-shelf CAN products [RPA08]. More details about this architecture can be found in [Ruf02] and [RPA08].

As an own opinion, the Columbus Egg Idea can be expanded to accommodate three or more buses, leading to a higher dependable system.

CANdor

CANdor is another architecture to increase the dependability in CAN networks. CANdor stands for CAN Duplicated Organization for Reliability [PPMJ99]. Figure 2.19 depicts the architecture of a CANdor node. As it can be seen in the figure 2.19, the CAN network, the controllers and the processor are replicated.

Each CAN channel has a replicated controller, both of them receive the same data from the network transceiver. The outputs of the CAN controllers are compared in a component named CANdor Comparator (CC1 and CC2 in figure 2.19). The data is then "ANDed" together and written to the CAN transceiver.

The data produced by the processors is compared by the Main Comparator (MC) system to check if the data transmitted and received is the same.

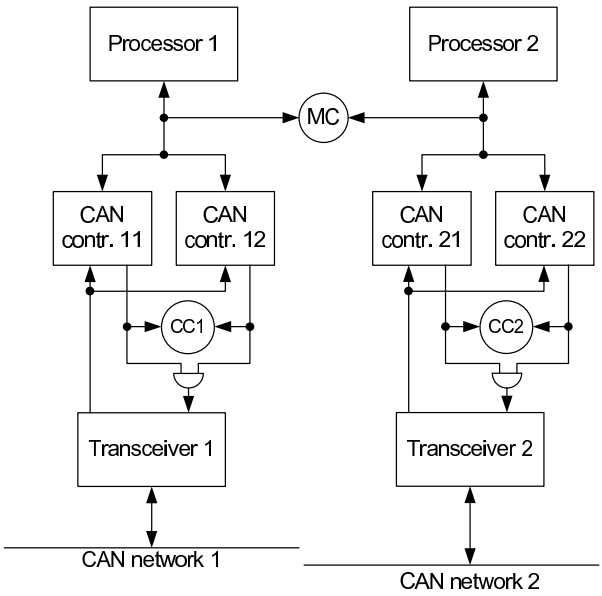


Figure 2.19: CANdor node (from [FNP+98])

A CAN system based on CANdor architecture exhibits three different levels of redundancy [FNP+98]: component level redundancy; channel level redundancy; and system level redundancy.

Ferriol et al. [FNP+98] claim that the component level redundancy is the inherent re-

dundancy of the CAN network and the redundancy introduced in the CAN controllers. The channel level redundancy refers to the duplicated channel to transmit redundant data. And the third level of redundancy (system level redundancy) means the use of a set of communication nodes.

The CANdor architecture is also applicable to industrial networks where each node is a personal computer [PPMJ99]. In [PPMJ99] the authors present a circuit called RCMP (Redundancy and Communication Management Processor) that is a fundamental piece of the strategy for adding fault tolerance to industrial control systems. RCMP is an interface between each computer and the communication channel.

Fault-Active Mechanism

The Fault-Active Mechanism (FAM) is a CAN based fault tolerant system architecture [HKD97] with two CAN networks and a protocol to deal with the redundant buses. According to [HKD97], the transmitter of a CAN message does not know if the receiver receives that message. If no error frames are sent the transmitter assumes a correct reception at the receiver. This will lead to a larger time to detect possible loss of a network node [HKD97].

A fault tolerant communication node for the FAM is presented in figure 2.20. As it can be seen in figure 2.20, there are two redundant buses ('Bus A' and 'Bus B'). Each of them is monitored by the other. In addition, in case of a component fault, a node is able to become active autonomously, *i.e.*, it informs all the network nodes about the failure by transmitting an error notification message through the operational channel. In this way the redundant bus is a kind of a watchdog bus.

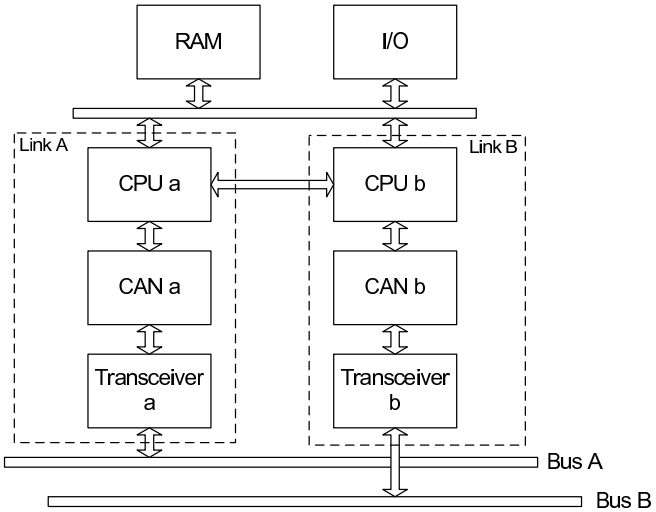


Figure 2.20: Fault tolerant communication node (from [HKD97])

The authors of the FAM argue that the negative confirmation mechanism of the CAN protocol leads to high latencies since the transmitter does not detect the failure of other

network nodes, rather it assumes that, if no error message is received, all of the receivers have correctly received the message.

During the normal operation, data exchange is only done through one communication channel. In the case of a CAN controller fault of the active communication channel, the fault tolerance process takes place as follows:

- The CAN controller fails;
- Reaching 96, the CAN controller sends an interrupt to the corresponding microcontroller;
- The error counter of the CAN controller reaches 128;
- The microcontroller informs the microcontroller of the other link of the loss of the CAN controller;
- The microcontrollers which receive this indication start to transmit the error notification through the other buses;
- All the network nodes receives the error message through non-faulty bus;
- All nodes switch off the faulty bus.

In addition to the watchdog system provided by the use of two parallel links, this architecture also provides reception and transmission monitoring. This mechanisms allow a faster detection of some faults than in a normal CAN controller.

This Fault-Active Mechanism is applicable, as an example, in large-scale manipulators for heavy weights to work in tunnels as described in [KGHL98]. This particular application has to cope with some requirements that cannot be fulfilled with a normal communication system [KGHL98].

CANopen

CANopen is a higher layer protocol based on CAN and on the CAN Application Layer (CAL) [FR97]. CAL is a higher layer protocol developed by Philips Medical Systems [CAN96]. CAL was adopted by the independent CAN group, CAN in Automation (CiA) [Bot00]. It provides all the network management services and all the message protocols. However, as referred by *Boterenbrood* [Bot00], it does not provide any kind of object: "*It defines how, not what*".

CANopen is based on the profile concept. The devices with the same functionality will have the same profile and the same behaviour. For example, two digital I/O modules from different manufacturers will have common functionalities, such as setting the outputs and reading the inputs. This strategy leads to an improved interoperability and also imposes

some standardization. Moreover, the vendors of the devices are not limited to the existing profiles, they can develop their own profiles or functions to a specific device.

The essential part of the device profile is the object dictionary [FRB99]. This consists of a set of data objects, communication objects and commands (or actions).

The communication in CANopen is divided into four classes:

- Service Data Objects (SDO). Mainly used for device parameterization and configuration;
- Process Data Objects (PDO). Are used during the normal operation to transfer real-time data without processing overhead (or, more precisely, with minor processing overhead);
- Network management functions, for coordinating device operations. These are accomplished by a network management facility. This is organized according to a logical master-slave relationship;
- Predefined format messages. These messages target the timing and synchronization.

In what concerns the bus redundancy of CANopen protocol, IXXAT (an industrial and automotive communications supply company [IXX08]) develops a software framework that implements redundant communications for the CANopen protocol targeting maritime applications [EHN+08].

In figure 2.21 the architecture of the CANopen node with the redundant buses is presented. As it can be seen in figure 2.21, there are two independent CAN buses with independent controllers and independent drivers. The software developed by IXXAT provides complete support for the redundancy management according to specification CiA® 307 [CAN07]. The same data is always transmitted in both channels. If a channel fails, the system continues working using the other channel, without interrupt or data loss.

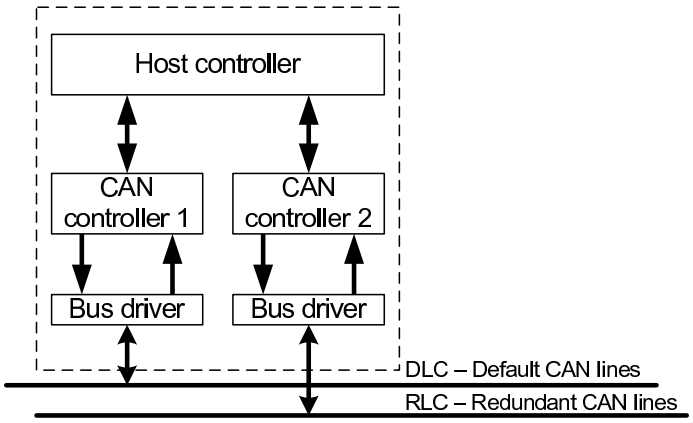


Figure 2.21: CANopen redundant communication

Concerning node redundancy, in [TO08], the authors present a master redundant system for the network management of CANopen. The authors call the system FTNMSCAN (Fault Tolerant Network Management System for CAN).

MilCAN

MilCAN [Mil06] is an open standard interface targeting the military applications. It is based on CAN, achieving deterministic network operation using a prioritized bus access and bounded throughput. The priority of the access to the bus is defined by the criticality of each node function. The maximum transmission latency for the different priorities is guaranteed. The message generation can be limited within their allocated period, and the network traffic can be pre-scheduled to provide deterministic operation.

In the MilCAN specification there are no definitions for the physical layer redundancy [Mil06]. In this specification it can be read: "*N-level media redundancy is not specifically addressed in the MilCAN specification*". However, one can find in the bibliography solutions to introduce redundancy to the MilCAN networks. One example is presented in [Tay06], where a redundant MilCAN bus is applied to a battle tank, called Terrier [VSI04, Mil05]. In this project there are several segments of MilCAN redundant buses. The use of multiple MilCAN dual redundant buses segments serves to minimize data loading in each segment and also to restrict the fault propagation [Mil05].

In [OSCA08], a fault tolerant layer to apply in MilCAN has been presented. This layer is located between the application and the MilCAN layer and is called Fault Tolerant layer (FT). This layer is responsible to manage the physical connections of the node with multiple buses. The authors claim that the FT layer can operate in two or more buses.

DeviceNet

DeviceNet is a network technology based on CAN used in the automation industry to interconnect field devices. In DeviceNet all the system is organized in an object oriented way, where all the operations must be done using the objects definitions.

Regarding the network model, DeviceNet follows the OSI model, where the data link layer is derived from the CAN specifications. For the upper layers, DeviceNet uses the Common Industrial Protocol (CIP) [Ope04]. The common industrial protocol defines an application layer to cover a range of device profiles [Imo02]. Each object has attributes, services and behaviours. For a given device type, a minimum set of common objects will be implemented. The user benefits from interoperability among devices regardless of the manufacturer or the device type [Ope04]. However, vendor specific objects can be defined when there is no suitable object in the definitions.

The common industrial protocol is a definition from ODVA (Open DeviceNet Vendor Association) [Ope10] to unify the communication using media independence. Thus, the

underlying media is masked, and several technologies can be used, such as: Ethernet/IP or CAN (resulting into DeviceNet).

Concerning the physical layer, DeviceNet uses two separated twisted pairs: one for signals and the other for power distribution. The network topology can be bus using stubs (called drop lines) with a maximum of 64 nodes [Ope04]. In the stubs several devices can be connected forming a daisy chain. The possible data rates are $125kbps$, $250kbps$ and $500kbps$, resulting in different bus lengths and drop line lengths. The maximum distance between two nodes cannot exceed the defined length for each speed [Ope04].

Concerning the bus redundancy, a third party manufacturer called Auma [Aum10] sells devices which use bus redundancy. In [Aum03], an actuator to operate industrial valves is presented, and the connection to the DeviceNet network can be redundant. This redundancy uses a dual DeviceNet interface. If an interface fails, the communication can be done using the other interface because the information is always sent in both interfaces. Thus, if the communication is available through both interfaces, the data that arrives first at the destination is used.

Regarding the node redundancy, no information has been found during this work.

2.6.4 CAN star topologies

Star topologies are also often used in communications systems, an example of such being Ethernet. Some years ago, Ethernet migrated from a bus topology to a star topology near the station computers. Currently, Ethernet uses a star topology where the computers connect to a switch (or in rare cases, to a hub). This migration has also been found in adaptations of fieldbuses, particularly in CAN.

Research has been made in the last years concerning star topologies for use in the CAN protocol. Examples of such can be found in [CDV01], [BPRNA06], [SO06] and [BPA09]. According to the star supporters, when compared with the bus topology, the star topology has some advantages, namely: better error containment, the links come to spatial proximity just near the star hub and this hub has a privileged view of the system.

Because of this privileged view of the global system, the hub can isolate any node that may cause problems. In the bus topology this is not true unless bus guardians are used [BPA09]. *Barranco et al.* [BPA08, BPA09] claim that, even using replicated buses, the buses are normally placed near one from the other, leading to common-mode failure, such as a partition in both buses.

On the other hand, the star architecture has some disadvantages like the appearance of a single point of failure (the central hub), the increase of the cabling in the installation and the associated higher cost. The higher cost is derived from the use of a more complex component (the central point) and much more cabling and physical interfaces.

Barranco et al. present an active CAN star hub (CANCentrate [BPRNA06]) that can be

replicated (ReCANCentrate [BPA09]). Other solutions are based on passive stars, such as the one found in [CDV01]. In passive stars there are electrical problems and lower bitrate on the radius of the star. *Saha et al.* [SO06] present an active star that just receives bits from the nodes, does a logical "AND", and transmits the result to all the nodes. However, even this last active star cannot isolate faulty nodes.

In the work presented by *Barranco et al.* [BPA09], each node connects to the hub using one uplink and one downlink. Uplink and downlink are point-to-point unidirectional links providing isolation of the signals from/to the hub. CANCentrate relies on an active star that is able to isolate nodes/links that suffered one of three types of faults: stuck-at faults, medium partition and bit flipping [BPA09].

ReCANCentrate [BAP05] defines a redundant star topology where the hubs are connected together (with a link called interlink). The nodes are connected to both hubs. This architecture is depicted in figure 2.22.

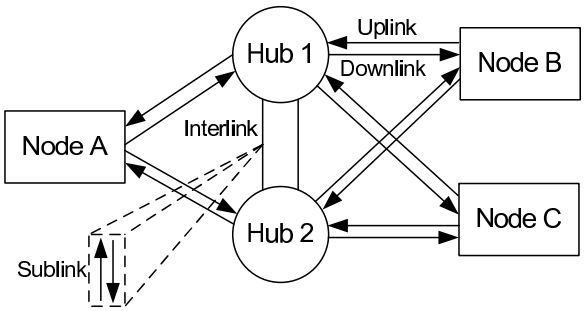


Figure 2.22: ReCANCentrate architecture (from [BPA09])

As it can be seen in figure 2.22, if there is a fault in any downlink or uplink, the system has an alternate path for the data communication. This path uses the interlink connection to the other hub. In case of one of the hubs goes off, the system still works, since there is another hub to maintain the star topology. However, if there is a fault on a link and also in a hub, the system can stop working (depending on the hub and link).

Cena et al. [CDV01] present a star topology intended to enlarge the CAN network and also to increase the bitrate by the same factor, called StarCAN. StarCAN also uses an uplink (called FL - Forward Link) and a downlink (called RL - Reverse Link) to perform the communication between the nodes and the central hub (called LSC - Logic Star Coupler). This LSC carries out all the operations involving the control, the access to the network and is also in charge of resolving possible collisions.

Saha et al. [SO06] present an active high-speed CAN hub. This hub is able to connect nodes that use off-the-shelf CAN transceivers. This connection can be made without any modification to the CAN physical layer. In that way, the authors claim that this hub is compliant with the ISO 11898-2 CAN physical layer standard. In [SO06], the authors present

a developed prototype based on a CPLD (Complex Programmable Logic Device) and three CAN interfaces.

2.7 Brief comparison

In the previous sections, protocols where redundancy in the physical layer can be applied were presented. These protocols were divided into four main groups, where the criteria was the target application. A final comparison among them will be performed in this section. The comparison will also be divided into four tables, respecting the previous separation. However, the same assessment criteria will be used in all of them.

In table 2.3, protocols for industrial automation are compared regarding: Maximum number of buses; the use of the additional buses for bandwidth improvement, the type of redundancy and the possibility of node replication.

Protocol	Maximum number of buses	Bandwidth improvement	Type of redundancy (for bus)	Node redundancy	Comments
WorldFIP	2	No	Active	Yes: bus arbitrator	
PROFIBUS	2	No	Active	Yes	
P-NET	2	No	Passive	No	The redundant buses are developed by a third party company. Information about node redundancy not found.

Table 2.3: Industrial automation protocols comparison

Comparing the protocols presented in table 2.3, they are all similar concerning the assessment criterias. The P-NET has no redundancy at node level while the other two have this feature. Concerning the bus redundancy, specification does not provide it for P-NET. This buses redundancy has been developed by a third party company.

WorldFIP can use bus redundancy and node redundancy. Node redundancy is only applicable to the bus arbitrator node. On the other hand, PROFIBUS can use replicated buses and also redundant nodes.

The three protocols presented in table 2.3 can only have a supplementary bus and none of them can use the additional bus to increment the total available bandwidth of the system.

Regarding the protocols for embedded applications, in table 2.4, the characteristics of them are summarized.

For embedded applications (presented in table 2.4), the analyzed protocols have node replication and bus replication. Regarding the bus replication, MIL-STD-1553B can use up

Protocol	Maximum number of buses	Bandwidth improvement	Type of redundancy (for bus)	Node replication	Comments
TTP	2	No	Active	Yes	
FlexRay	2	Yes	Active	Yes	
MIL-STD-1553B	4	No	Passive	Yes	Few information about node replication.

Table 2.4: Embedded applications protocols comparison

to 4 buses in a passive redundancy scheme. FlexRay and TTP can use up to 2 buses (or stars) using an active redundancy.

FlexRay is able to improve the bandwidth using the additional buses. This means that, in theory, the bandwidth can be doubled using the two buses.

Concerning protocols based on CAN, in table 2.5 they are compared.

TTCAN, FlexCAN and MilCAN can use an unlimited number of buses. In this sense, unlimited means as many buses as the microcontroller can support. This number is limited by the number of CAN controllers, by the processing power and available memory. In the case of TTCAN, the additional buses can be used to increase the system bandwidth. Despite this, the redundant buses are managed by higher layers and thus is not transparent to the application. However, no other analyzed protocol can use the additional buses for this purpose.

Concerning the node redundancy, in RedCAN and Columbus Egg Idea it doesn't make sense to talk about it, because the definitions are only based on the fault tolerance of the bus. In RedCAN the base is a module to isolate nodes or to transform the formed ring CAN in a bus. The Columbus Egg Idea is based on one node, and thus, the replication of a node is not considered. In fact, Columbus Egg Idea is not a protocol definition, but a change to the physical layer.

The Fault Active Mechanism, CANdor and CANopen can use up to 2 buses and replication of the node. In the three protocols, the additional buses cannot be used to send different data in different buses.

DeviceNet is an important standard concerning the industry. However, there is few information about the node and network replication. There is a device developed by a third party company that claims to have DeviceNet network redundancy. Regarding node replication no information has been found.

Except the Fault-Active Mechanism, the others CAN based protocols that provide some kind of network redundancy use an active redundancy.

Regarding the CAN star solutions, the comparison is presented in table 2.6.

In the case of the star topologies, talking about the replication of the network is not the same as talking about the network replication of the bus. In the case of the stars, the link

Protocol	Maximum number of buses	Bandwidth improvement	Type of redundancy (for bus)	Node replication	Comments
TTCAN	Unlimited	Yes	Active	Yes	Node replication is provided for the time master. Bandwidth improvement managed by higher layers.
FlexCAN	Unlimited	No	Active	Yes	
RedCAN	1	No	—	No	
Columbus Egg Idea	2	No	Active	No	Expanding the number of buses can be done.
CANdor	2	No	Active	Yes	
Fault-Active Mechanism	2	No	Passive	Yes	
CANopen	2	No	Active	Yes	
MilCAN	Unlimited	No	Active	No	Information about node replication not found in literature.
DeviceNet	2	No	Active	No	The redundant buses are developed by a third party company. Information about node redundancy not found.

Table 2.5: CAN based protocols comparison

from the node to the star hub is a dedicated link, and can only be used by the node itself.

ReCANCentrate and StarCAN use one link for the uplink and one link for the downlink. Additionally, ReCANCentrate can use a replicated central hub linked together. Moreover, in ReCANCentrate all the nodes are connected to both hubs.

Concerning the Ethernet based solutions, in table 2.7, the protocols presented in section 2.5 are compared.

As it can be seen in table 2.5 only the commercial solution DualNet and EtherChannel can use the additional buses to improve the bandwidth. The other solutions do not have this feature (or no information about it could be found).

EtherCAT, uses a ring topology to provide redundancy. In that way, one partition on the media is supported. There is no bandwidth increasing because of the use of the ring topology.

DualNet is a type of Ethernet controller for personal computers. Thus, there is no definition for the node replication.

Regarding PROFINET, node replication is planned (like stated in [Hen09]). No more information about node replication in PROFINET could be found in the literature.

Protocol	Maximum number of buses	Bandwidth improvement	Type of redundancy (for links)	Node replication	Comments
ReCANCentrate	4	No	Active	Yes	Node replication is valid for the central hub. The nodes are connected to both hubs using uplink and downlink.
StarCAN	2	No	—	No	Use of two unidirectional links (downlink and uplink).
Saha's Hub	1	No	—	No	

Table 2.6: CAN star topologies comparison

2.8 Conclusions

In this chapter definitions concerning dependability and fault tolerant communications have been discussed. Among others, definitions that must be taken into account are the dependability, fault, error and failure. These last three are defined as a sequence of events in a system.

In architectures relying in fieldbuses, the use of a common communication medium can introduce additional dependability problems. These problems arise from the use of more electric/electronic components. In that way, there are more components in the system which can have faults.

The focus of the discussion is fault tolerance and system/architecture to achieve it. In particular, redundancy is discussed since it is a fundamental way to provided dependability. However, the use of redundancy in a specific component can introduce other problem, namely replica consistency problems. The consistency problem can be compared with the consistency problem used in a database system. This problem has been addressed regarding distributed systems in a general way. There are communication primitives to ensure the consistency of the replicas, namely the atomic broadcast.

In this chapter, protocols/communications systems that provide replication in the physical layer have been presented. These protocols have been divided into four main categories regarding their target applications and support on a widely accepted standard: buses for industrial automation, buses for embedded systems, Ethernet based solutions and CAN based solutions.

In section 2.7 a brief comparison among the presented protocols was performed. The used criterias are the bus replication, node replication and if the additional buses serve to increase the bandwidth. The type of redundancy is also assessed. In the presented analysis, only TTCAN can have an unlimited number of active redundant buses with bandwidth

Protocol	Maximum number of buses	Bandwidth improvement	Type of redundancy (for bus)	Node replication	Comments
PROFINET	2	No	Active	Planned	No information found about the bandwidth improvement using the replicated buses.
TTE	2	No	Active	Yes	TTE can use replicated Ethernet channels. The topology is a star with redundant hub.
Powerlink	2	No	Active	Yes	Node replication available for managing node.
EtherChannel	8	Yes	Active	Yes	
DualNet	2	Yes	Active	No	
EtherCAT	1	No	—	No	Ring topology. Information about node replication not found.

Table 2.7: Ethernet based protocols comparison

improvement, and also node replication. However, the bandwidth improvement is managed by the higher layers and thus is not transparent to the application. TTCAN also provides node replication, even if only for the time master node.

The use of the CAN network is still actual since it supports many applications, namely in automotive and avionics. The presented CAN protocols that have redundancy cannot fully satisfy requirements such as redundancy and bandwidth improvement in the lower layers. Moreover, the number of buses that can be used is normally limited (in rare cases there can be used an unlimited number of buses).

In the next chapter, a proposal to achieve active buses redundancy is described. The proposal is targeted to a particular paradigm (FTT-Flexible Time-Triggered), however it can be generalized as it will be presented further on.

Chapter 3

A proposal for bus media redundancy in FTT-CAN

3.1 Introduction

The FTT-CAN protocol (Flexible Time-Triggered communication on CAN) [APF02, APF+07] is a time-triggered protocol based on CAN that provides a high level of operational flexibility. FTT-CAN adopts a master-multislave architecture and in its initial specification it did not consider master replication or other fault tolerance issues. In the last few years, some work was carried out to add fault tolerance properties without compromising dependability [Fer05]. However, the bus media redundancy was not considered, making the non-replicated bus a single point of failure. Also, the available bandwidth of the FTT-CAN system is inherited from the CAN network less the overhead imposed by the FTT-CAN protocol.

The use of replicated buses is the obvious solution, both to remove the single point of failure and, if possible, to gain additional bandwidth. With this solution, the flexibility of FTT-CAN is increased since more traffic can be be transmitted over one or several buses. In case of the unavailability of one bus, the system can switch to a degraded mode where it will operate safely but with some performance penalty, considering that at least one bus is working properly.

In [Fer05], a master replication scheme was been presented to improve the fault tolerance. When applying this replication to a multiple buses system, new problems arise. However, new features can be added using this replication.

In the proposed architecture, the number of replicas (buses and masters) can be unlimited enforcing the dependability of the system as well as improving the bandwidth (in case of bus replication). It should be noticed that the improvement of the bandwidth is a very important feature since the limitations of bandwidth in CAN were considered severe for some applications such as automotive and thus lead to the emergence of new protocols such as

FlexRay [Fle04c].

3.2 The FTT-CAN protocol

The FTT-CAN protocol [APF02] has been developed with the main purpose of combining a high level of operational flexibility with timeliness guarantees. It uses the dual-phase elementary cycle concept for isolated time and event-triggered communication. The time-triggered traffic is scheduled centrally and online in a particular node called a master, facilitating online admission control of requests. The protocol relies on a relaxed master-multislave medium access control in which the same master message triggers the transmission of messages in several slaves (also known as stations) simultaneously (master-multislave). The eventual collisions among slaves' messages are handled by the native distributed arbitration of CAN.

In figure 3.1 the general architecture of the system is depicted.

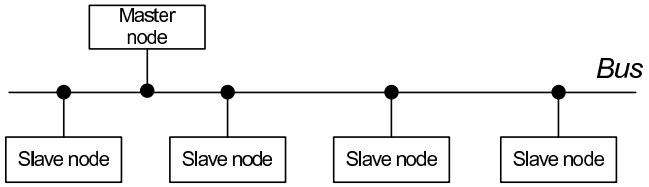


Figure 3.1: FTT-CAN architecture

FTT-CAN bus time is slotted in consecutive Elementary Cycles (EC) with fixed duration: LEC - Length of the Elementary Cycle. All nodes are synchronized at the start of each EC by the reception of a particular message known as an EC Trigger Message (TM). This trigger message is sent by the master node. Within each EC the protocol defines two consecutive windows, asynchronous and synchronous, that correspond to two separate phases (see figure 3.2). The first is used to convey event-triggered traffic, here called asynchronous because the transmission requests can be issued at any instant. The second is used to convey time-triggered traffic, herein called synchronous because its transmission occurs synchronously with the ECs. These two windows are separated by a guard time (α in figuere 3.2). The slaves cannot begin any transmission during this time however, an asynchronous message that began before can occupy this time. Thus, it belongs to the asynchronous window.

The synchronous window of the n^{th} EC has a duration that is set according to the traffic scheduled for it. The schedule for each EC is conveyed in the respective data field of the EC trigger message (see figure 3.3). Since this window is placed at the end of the EC, its starting instant is variable and it is also encoded in the respective EC trigger message.

The communication requirements are held in a database located in the master node [Ped03], the System Requirements Database (SRDB). This database holds several com-

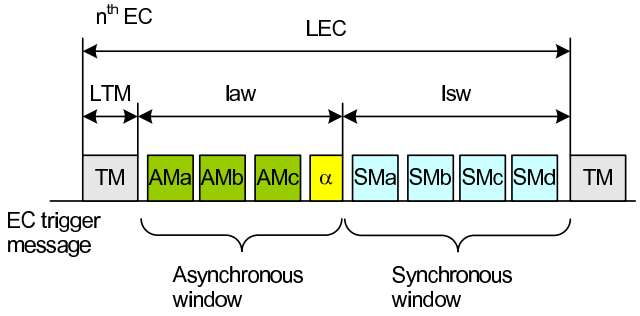


Figure 3.2: The elementary cycle in FTT-CAN

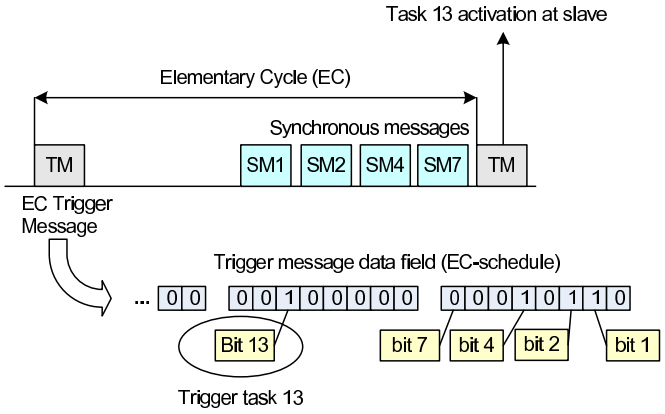


Figure 3.3: Master-multislave access control and EC schedule coding scheme

ponents, one of which is the Synchronous Requirements Table (SRT), that contains the description of the periodic message streams. Based on the SRT, an online scheduler builds the synchronous schedules for each EC (EC - schedule). These schedules are then inserted in the data area (each bit triggers a message, and is called a trigger flag) of the appropriate EC trigger message (see figure 3.3) and broadcasted with it. In the data field of the trigger message, information concerning the control of the system is piggybacked. This information is the length of the synchronous window, a sequence number for control, a master identification, among other. Two data bytes of the data field are reserved for this information.

Moreover, the FTT master can also trigger tasks on the slave nodes [CF04]. The tasks to be triggered are also encoded in the trigger message using the same mechanisms adopted for the synchronous messages. After decoding the TM, the slaves activate the corresponding tasks at the middle of the next TM. Due to the synchronous behaviour of the TM, the task activation has a low jitter. Figure 3.3 shows task 13 activation. This task activation can be used to trigger functions in one or more slave nodes, leading to synchronization among slaves.

Due to the master-multislave architecture, the master has the control of all the communications on the bus and has a centralized view of the traffic, knowing all the messages which must be scheduled at any given time. As a result, these features make possible the use of

any uniprocessor non-preemptive scheduling algorithm by the master node. The slaves send asynchronous messages to the master whenever they need to change the set of messages or tasks to be scheduled. If the new schedule is feasible, the master will reflect this in subsequent trigger messages.

Calha and *Fonseca* defined an architecture to trigger tasks remotely using FTT in a platform independent way [CF03, Cal06]. This architecture uses the native independence of hardware of the Java language to trigger tasks remotely, the so called FTTlet that are scheduled by the FTT master like the servelets used in the Java language [CFSM06]. In the slaves the FTTlets are managed by the Java Virtual Machine (JVM) which must be changed in order to trigger the FTTlets. The authors claim that the system is fast because code is loaded into memory once and runs from memory thereafter; is relatively simple to implement; and it inherits the independence of platform of the Java architecture. However, the use of Java in small devices is not yet very disseminated. Regarding this issue, refer to [Sil02] and [IJCS08] for more details.

Also, the slave nodes can online change the synchronous messages of the system. The slaves ask the master to change the synchronous requirement for a particular message or task (generally called variables). This is done using an asynchronous message sent by the slave to the master.

3.2.1 FTT master replication

Since the master is a single point of failure, it must be replicated. The master replication has already been object of intensive work by *Ferreira* [Fer05]. The master can have one or more replicas to improve the fault tolerance of the system.

The system adopts a leader-follower behaviour, where the leader sends the trigger message and the follower also tries to send it in parallel. If the follower cannot send it (because the bus is busy with the leader trigger message), it remains as the follower. On the other hand, if the follower successfully transmits the trigger message, it becomes the leader, meaning that any problem has occurred with the leader master node. The masters need to be synchronized to perform the same schedule for each EC. This is done using a master replication protocol and synchronization defined in [Fer05]. After receiving the trigger message sent by the leader master, the follower master nodes compare it with the one generated internally. This comparison is made in the entire message, namely, the identifier and data (recall that the data contains the scheduling and control information). Two situations can then occur:

- The trigger messages match. The follower master prepares the next elementary cycle;
- The trigger messages do not match. The follower master enters in the synchronization state.

In the synchronization state, the follower will send an asynchronous message to the master leader. In the subsequent elementary cycles the leader master node will send asynchronous messages with the SRT and with information about the elementary cycle that must be met to successfully re-synchronize the follower master.

Concerning the online changes issued by the slaves, they are received by all the masters which update their synchronous requirement tables.

Regarding the localization of the masters, the active and backup masters can be placed anywhere across the bus.

Next, the FTT features will be summarized.

3.2.2 FTT features summary

Before the start of this work, the main features defined for FTT were:

- Time divided into consecutive elementary cycles;
- Coexistence of synchronous and asynchronous messages;
- Master schedule messages in the bus;
- Master schedule tasks to be triggered in slaves;
- Master replication with synchronization between them;
- Slaves can change online the synchronous' variables properties.

Despite these features, FTT-CAN has limitations that will be detailed in next section.

3.3 Limitations of the FTT-CAN

3.3.1 The priority inversion and jitter problem

Due to the distributed behaviour of FTT-CAN, the software applications run at different nodes, each with different clock skew, even if the hardware is identical. All the nodes are synchronized at the beginning of the elementary cycle using the reception of the trigger message. The time elapsed from the reception of the trigger message until the beginning of the synchronous window is sufficient to cause skew in the clocks of the distributed nodes. These small variations are sufficient to have a priority inversion problem at the beginning of the synchronous window.

All the synchronous messages are triggered by different slaves at the beginning of the synchronous window. If the timer of the slave who needs to send the highest priority message has a small delay compared with the timer of the slave who needs to send a lower priority message, the lower priority message will be transmitted first than the highest priority message [APS06].

Because all the synchronous messages are sent in sequence, this means that the last message in the synchronous window has less priority (if the problem presented in the last paragraph is ignored) than the first ones. The last messages of the synchronous window are affected by the jitter introduced by the bit stuffing imposed by the previous ones on the current elementary cycle. This jitter decreases for higher bit-rates. However, depending on the application and on the bitrate, jitter can affect the performance and the response time of the system [AAP06, APLA06]. In [TBW95], *Tindell et al.* computed the CAN message response time, not considering the bit stuffing, while, in [NHN03] a probabilistic worst-case transmission time based on bit stuffing distributions instead of the worst case values is discussed.

Ataide et al. [APLA06] propose a new scheme to the synchronous part of the elementary cycle of FTT-CAN to deal with jitter and priority inversion problems. In this solution the synchronous part is divided into slots, as in TDMA, where each slot is assigned to a specific message. Each slot allocates room for the largest possible message, *i.e.*, 8 bytes of data and maximum bit stuffing. This implies a bandwidth loss every time the transmitted message is smaller than the largest one. According to the scheme, messages are not sent in burst and between each two messages there is a guard window. The guard window corresponds to the stuff bits that are not inserted (because they were not necessary) plus the time to complete the largest message. This technique solves the priority inversion problem and decreases the jitter. In this guard time the slaves cannot begin any transmission.

In the asynchronous part of the elementary cycle, there is also a priority inversion problem. This problem is the same as the inversion problem of some CAN controllers documented in [THW95]. This problem has been reported in the literature for the Earliest Deadline First (EDF) scheduling policy [MNS96]. This problem is common in CAN controllers with more than one transmission buffer. The relative priority of the transmission buffers is usually assigned by software. However, to do a correct buffer management, the software must inspect all ready messages and assign the correct priority to the buffers just before message transmission. Some CAN controllers will do this by hardware making the correct priority assignment to the buffers according to the identifiers of the message to be transmitted. According to [THW95], an example of such controllers is the Intel 82527 [Int04]. This priority inversion will occur if the CAN controller has to wait for the bus to become idle (because there is messages on bus) while the application tries to send messages, that must wait at buffers.

Regarding the physical layer, FTT-CAN has two major limitations, the redundancy support and the limited available bandwidth. These limitations will be explained in next two sections (sections 3.3.2 and 3.3.3).

3.3.2 Network redundancy support

The CAN physical layer defines support for fault tolerance and error detection, through mechanisms to perform error detection at the message level: cyclic redundancy check, frame check and acknowledge errors.

At the bit level there are two error detection mechanisms, transmission monitoring, where each node transmitting a message also reads that message to ensure its correct transmission and bit stuffing (even if this is not used natively for error detection but for synchronization).

This error detection is effective and can be considered efficient for a large number of applications. On the other hand, the redundancy support in CAN is not so effective as the error detection. The CAN definition determines that the signal is transmitted in a dual wire operating in differential voltage. This kind of redundancy is implemented in a transparent way to the application and no information is passed to the application. The transmission of the electrical signal in a differential manner, just supports the partition of one wire. When this occurs, the noise signal ratio becomes lower and some CAN controllers switch to a degraded mode, configured at $125kbps$ [RVA99]. Thus, the bus redundancy is limited, since the controllers are switched to a degraded mode while operating with just one wire.

FTT-CAN does not define any redundancy at the physical layer. The available redundancy on FTT-CAN is inherited from the redundancy of the CAN underlying network layer. The use of redundant buses enables the transmission of different messages in different buses, leading to an improvement of the available bandwidth. The bandwidth problem will be addressed in the next section.

3.3.3 Available bandwidth

The available bandwidth in FTT-CAN is limited by the available bandwidth of the CAN underlying network. In addition to this limitation, the FTT-CAN uses some extra bandwidth to the trigger message, thus the bandwidth which can be used by the application decreases.

The CAN protocol data link layer imposes significant overhead. This overhead is due to the necessary signalling bits (identifier, CRC, etc) and due to bit stuffing. The CAN frame payload can vary from 0 to 64 bits while the number of signalling bits remains constant.

For CAN 2.0A (11 bits identifier), the minimum number of bits of a CAN frame without stuffing is represented by equation 3.1 [NHNP01]. In the referred equation, the three bits of intermission time are taken into account because these bits contribute for the total overhead. Equation 3.2 represents the maximum number of bits of a CAN frame [NHNP01].

$$\min(N_{bits}) = 8 \times DLC + 13 + g \tag{3.1}$$

$$\max(N_{bits}) = 8 \times DLC + 13 + g + \left\lfloor \frac{g + 8 \times DLC - 1}{4} \right\rfloor \tag{3.2}$$

The variable g in these equations (equations 3.1 and 3.2) is the number of bits exposed to stuffing. For CAN 2.A (11 bits identifier) is 34 and for CAN 2.0B (29 bits identifier) is 54 [NHNP01, NHN03], thus $g \in \{34, 54\}$. The number of bits, N_{bits}, represents the number of bits of a CAN message, which is an integer between $\min(N_{bits})$ and $\max(N_{bits})$ (as expressed in statement 3.3).

$$N_{bits} \in \mathbb{N} \wedge N_{bits} \in [\min(N_{bits}), \max(N_{bits})] \tag{3.3}$$

The overhead of the CAN message is represented by equation 3.4 while in equation 3.5 the CAN data payload throughput is presented.

$$CAN_{ovh} = (1 - \frac{8 \times DLC}{N_{bits}}) \times 100(\%) \tag{3.4}$$

$$CAN_{thp} = \frac{8 \times DLC}{N_{bits}} \times 100(\%) \tag{3.5}$$

The overhead of the CAN message is the percentage of bits which do not transport any useful information for the CAN user. This is the percentage of bits used by the protocol. The maximum and minimum CAN overhead is represented by equation 3.6 and 3.7, respectivelly.

$$\max(CAN_{ovh}) = (1 - \frac{8 \times DLC}{\max(N_{bits})}) \times 100(\%) \tag{3.6}$$

$$\min(CAN_{ovh}) = (1 - \frac{8 \times DLC}{\min(N_{bits})}) \times 100(\%) \tag{3.7}$$

From equation 3.1 to 3.7, DLC is the number of data bytes of the CAN message (please refer to section 2.6.2).

In table 3.1, the CAN message overhead is presented considering the minimum and maximum stuff bits (using equations 3.6 and 3.7) for all possible DLC cases.

Data bytes	CAN Overhead	
	Min S (%)	Max S (%)
0[4]	-	-
1	85.5	87.7
2	74.6	78.7
3	66.2	71.8
4	59.5	66.3
5	54	61.9
6	49.5	58.3
7	45.6	55.2
8	42.3	52.6

Table 3.1: CAN overhead
Min S: Minimum number of stuff bits
Max S: Maximum number of stuff bits

In table 3.1, the case of zero data bytes is a particular situation where no useful data is sent over the CAN network. However, sometimes a message with no data bytes can be useful, *e.g.*, to signal some event or to ask the transmission of a value (remote frame).

As it can be seen in table 3.1 the minimum overhead is achieved with 8 data bytes and without stuff bits. However, even with this scenario, the native CAN protocol overhead is 42.3%. A message without stuff bits has a low probability of occurrence.

Message manipulation can be performed to reduce bit stuffing [NHNP01, NHN02]. The proposed method is to perform a XOR of a bit pattern with the data to be transmitted at the transmission side. At the reception side the same operation must be done in order to restore the original data. The bit pattern must be chosen according to the data to be transmitted, thus adding a significant computing overhead. Even using the method presented by *Nolte et al.* [NHN02], it is not possible to get an overhead less than 42.3%.

As presented in table 3.1, the overhead can be as high as 87.7% (in the worst case). This means that, in a bus running at $1Mbps$, the maximum payload throughput achievable is $577kbps$ in the best case and $123kbps$ in the worst case.

The previous analysis was for the case of an identifier length of 11 bits. The same analysis can be made for CAN 2.0B (29 bits identifier). For this case, the overhead will be higher. As an example, for eight data bytes without stuffing, the overhead is 51.1% and for 1 data bytes with maximum bit stuffing the overhead is 89.3%. The work presented in this dissertation uses CAN 2.0A as reference, thus this issue will not be further discussed.

Other important factor that impacts on the CAN bitrate is the cable length. At the maximum bitrate ($1Mbps$) the cable length can be only $40m$, maximum. This length can be limitative if the bus is to be used in an industrial environment. To use a $1km$ bus the bitrate must be decreased for a value as low as $50kbps$ [Pas04]. This means that, even in the best case, the system will just have a throughput of $28.85kbps$ in CAN 2.0A (assuming the case

[4]Unusable for data transmission.

of no stuffing and 8 data bytes). This value is very low for the actual bandwidth demands of typical applications on the fieldbus domains.

In addition to the limitations of the CAN network physical layer, the FTT-CAN also imposes an overhead due to control messages, mostly by the trigger message. The bandwidth consumed by the trigger message depends on the length of the elementary cycle (in percentage), the bitrate and on the data bytes of the trigger message (recall that the number of data bytes of the trigger message is directly related with the number of synchronous messages in the system). Equation 3.9 represents the trigger message overhead without bit stuffing, while equation 3.10 represents the trigger message overhead with the maximum bit stuffing.

$$TM_{ovh} = \frac{N_{bits} \times \frac{1}{B_r}}{LEC} \times 100(\%) \tag{3.8}$$

$$\min(TM_{ovh}) = \frac{\min(N_{bits}) \times \frac{1}{B_r}}{LEC} \Leftrightarrow \min(TM_{ovh}) = \frac{(13 + g + 8 \times DLC_{TM}) \times \frac{1}{B_r}}{LEC} \times 100(\%) \tag{3.9}$$

$$\max(TM_{ovh}) = \frac{\max(N_{bits}) \times \frac{1}{B_r}}{LEC} \Leftrightarrow$$
$$\Leftrightarrow \max(TM_{ovh}) = \frac{(13 + g + 8 \times DLC + \left\lfloor \frac{g+8\times DLC_{TM}-1}{4} \right\rfloor) \times \frac{1}{B_r}}{LEC} \times 100(\%) \tag{3.10}$$

Where:

- N_{bits} is the number of bits of a CAN message (the trigger message). Refer to equations 3.1, 3.2 and 3.3;
- $g \in \{34, 54\}$, according the CAN standard used (as for equations 3.1 and 3.2);
- DLC_{TM} is the number of data bytes of the trigger message;
- LEC is the length of the elementary cycle;
- B_r is the CAN bitrate.

Table 3.2 presents the overhead of the trigger message (values in percentage) for an elementary cycle of $5ms$ while table 3.3 presents the overhead of the trigger message (values in percentage) using an elementary cycle of $20ms$. For a correct assessment of the tables 3.2 and 3.3 a note must be kept in mind: the trigger message needs at least two data bytes for the master to transmit specific information regarding the control of the network.

For an elementary cycle of $5ms$, the overhead of the trigger message cannot be less than 1.4%. This value is achieved with just one byte for the trigger message flags for a bitrate of $1Mbps$.

DLC_{TM} / B_r (kbps)	3		4		5		6		7		8	
	Min S	Max S	Min S	Max S	Min S	Max S	Min S	Max S	Min S	Max S	Min S	Max S
50	28.4	34	31.6	38	34.8	42	38	46	41.2	50	44.4	54
100	14.2	17	15.8	19	17.4	21	19	23	20.6	25	22.2	27
125	11.4	13.6	12.6	15.2	13.9	16.8	15.2	18.4	16.5	20	17.8	21.6
250	5.7	6.8	6.3	7.6	7	8.4	7.6	9.2	8.2	10	8.9	10.8
500	2.8	3.4	3.2	3.8	3.5	4.2	3.8	4.6	4.1	5	4.4	5.4
1000	1.4	1.7	1.6	1.9	1.7	2.1	1.9	2.3	2.1	2.5	2.2	2.7

Table 3.2: Trigger message overhead for $5ms$ of elementary cycle
Min S: Minimum number of stuff bits
Max S: Maximum number of stuff bits

DLC_{TM} / B_r (kbps)	3		4		5		6		7		8	
	Min S	Max S	Min S	Max S	Min S	Max S	Min S	Max S	Min S	Max S	Min S	Max S
50	7.1	8.5	7.9	9.5	8.7	10.5	9.5	11.5	10.3	12.5	11.1	13.5
100	3.6	4.3	4	4.8	4.4	5.3	4.8	5.8	5.2	6.3	5.6	6.8
125	2.8	3.4	3.2	3.8	3.5	4.2	3.8	4.6	4.1	5	4.4	5.4
250	1.4	1.7	1.6	1.9	1.7	2.1	1.9	2.3	2.1	2.5	2.2	2.7
500	0.7	0.8	0.8	1	0.9	1.1	1	1.2	1	1.3	1.1	1.4
1000	0.4	0.4	0.4	0.5	0.4	0.5	0.5	0.6	0.5	0.6	0.6	0.7

Table 3.3: Trigger message overhead for $20ms$ of elementary cycle
Min S: Minimum number of stuff bits
Max S: Maximum number of stuff bits

With a larger elementary cycle the trigger message overhead decreases. However, with a larger elementary cycle, the time resolution of the system would also decrease. In table 3.3 the minimum overhead of the trigger message is 0.4%. This is achieved with 1 data byte for trigger flags and without stuff bits. For 8 data bytes, with a bitrate of $1Mbps$, the overhead of the trigger message varies from 0.6% (without stuffing) to 0.7% (with maximum stuffing).

As an example, in the CAMBADA (Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture) soccer robot with distributed control, where the FTT-CAN is applied to connect sensors, controllers and actuators, the trigger message has 6 data bytes, the elementary cycle was set to $5ms$ and the bitrate was set to $250kbps$ [ASF+04], leading to an overhead of the trigger message of 7.6%. Also note that this value assumes zero bit stuffing (unrealistic in a real environment). If the worst case of stuff bits is used, the trigger message overhead will be 9.2%.

In that way, the total overhead regarding the FTT-CAN is the combined overhead of the CAN overhead and the trigger message overhead. Thus, the maximum available bandwidth for the application is:

$$BW = B_r \times (100 - CAN_{ovh}) \times (100 - TM_{ovh})(kbps) \tag{3.11}$$

Using the example explained before (the CAMBADA soccer robot), with:

- Elementary cycle of $5ms$;
- Trigger message with 6 data bytes;

- Bitrate of $250kbps$;
- Synchronous and asynchronous messages of 8 data bytes (FTT-CAN messages for data exchange);
- Assuming no stuff bits in all the messages (trigger, synchronous and asynchronous messages).

In this case, the available bandwidth for the application is:

$$BW = 250 \times (100 - 7.6) \times (100 - 42.3) = 133.28kbps \tag{3.12}$$

Notice that the presented analysis used a real-world example for the system parameters, but with no bit stuffing. This means that the throughput can be even worse.

To overcome the bandwidth limitation and the lack of bus media redundancy in CAN and FTT-CAN, the bus replication is the main focus of this dissertation and several proposals will be detailed next.

3.4 FTT-CAN with multiple buses

3.4.1 Introduction

The use of more than one communication channel is a paradigm already used in some systems, *e.g.* FlexRay, TTP/C and Ethernet. However, all of them have limitations, as presented in section 2.7. FlexRay, for example, only allows the use of two redundant buses that can also be used to send different data, improving the available bandwidth of the system. On the other hand, TTP only allows the transmission of the same message over all the buses.

Concerning CAN, the available protocols also exhibit limitations regarding the bus redundancy support. More specifically, TTCAN and FlexCAN are the more featured solutions regarding the bus redundancy. However, FlexCAN cannot use the additional buses for improving the available bandwidth while TTCAN has this feature managed by higher layers protocols.

FTT-CAN can achieve high efficiency and flexibility. To improve this flexibility and overcome the limitations presented in section 3.3, an architecture with support for two or more buses is proposed in this dissertation. The data can be the same in all the buses, increasing the redundancy of a message, or can be different, increasing the available bandwidth of the system [SF06, SFF07a]. This flexibility is an easy way to overcome the redundancy and bandwidth limitation, without adding significant modifications to the nodes in the system. Even the slave nodes can be exactly the same as for the case of a single bus.

The following sections present the architecture and required modifications to FTT-CAN necessary to provide bus media redundancy and/or increase its payload.

3.4.2 Multiple buses architecture

The natural solution to overcome the presented limitations is to use more than one bus in FTT-CAN, since, with multiple buses, both the bandwidth and the dependability can be increased [SF06, SFF06b]. In addition, in the last years, the microcontrollers manufacturers added extra CAN controllers to their microcontrollers stimulating the use of multiple buses and facilitates any possible implementation. In figure 3.4 an example of the planned system architecture with four buses is presented.

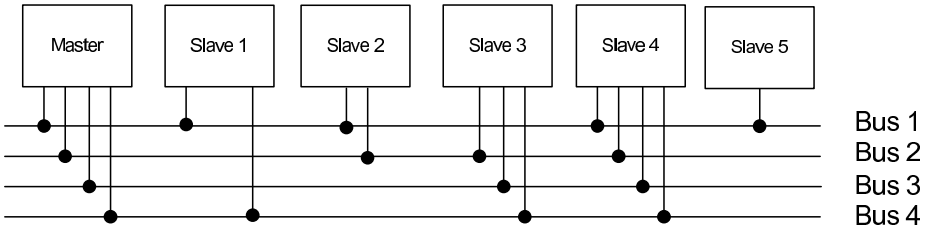


Figure 3.4: FTT-CAN with multiple buses architecture

FTT-CAN is a master-multislave architecture where the master has a complete view of the system. As it can be seen in figure 3.4, the master node connects to all the available buses while the slaves can connect to just one, to a subset or to all buses. As in the case of FTT-CAN with one bus, the master node performs the scheduling for all the buses. The slave nodes can connect to just one bus (slave 5 in figure 3.4) or to a set of buses (slaves 1, 2 and 3 in figure 3.4). Thus, the slaves used by the FTT-CAN with just one CAN bus, can be used in this new architecture.

The additional buses improve the overall bandwidth of the system while increasing the fault tolerance. The additional buses can be used to transmit the same message on all the buses (or a subset of them) or to transmit different messages in different buses. Moreover, a hybrid scheduling of the messages is possible, *e.g.*, a message could be transmitted in two buses and other message in other buses at the same time. This also improves the flexibility of FTT-CAN, a major design goal since its initial design stages [APF02].

3.4.3 The trigger message

The trigger message (TM) needs to be adapted to the new multiple buses architecture. Several approaches can be devised to trigger message transmission in the multiple buses system.

The trigger message can be transmitted in all the buses or it could also be transmitted in a subset of the buses. If transmitted in all the buses it can be synchronized with the other trigger messages in the other buses, or it could be unsynchronized. However, to take advantage of the possibility of synchronizing the slaves with the trigger message, it was decided to synchronize

the TM transmission in all the buses, *i.e.*, in all the buses the elementary cycle begins and ends at the same time. This design decision has some advantages comparing with having different elementary cycle lengths in different buses:

- All the buses have the same timing mark. In that way, slaves that are connected to just one bus are synchronized in time with all the other slaves in the system. If the trigger messages were not transmitted at the same instant in all the buses, an important property of the system would be lost: the synchronization of all the slaves;
- It is easy to send redundant messages in different buses. If all the buses were not synchronized, it would be almost impossible to send redundant messages since it would be necessary to send information concerning message scheduling and time of production/consumption.

In the case of just one bus, the trigger message is transmitted, containing the scheduling for the present elementary cycle. However, for the case of more than one bus, several approaches can be foreseen for the behaviour of the trigger message [SF06]. Three approaches to transmit the trigger messages have been identified:

(a) Only one trigger message is transmitted in one bus per each elementary cycle;

(b) N identical trigger messages are transmitted in all the buses per each elementary cycle;

(c) Different trigger messages are transmitted in different buses per each elementary cycle.

The transmission of the trigger message in just one bus or a subset of buses (scenario (a) above) can be considered with low interest, since some of the slaves may only be connected to just one bus. This would imply that all the slaves should be connected to the bus where the trigger message is transmitted. Notice that this scheme leads to a better bandwidth usage since it requires less trigger messages, but it decreases the flexibility of the entire system.

In what concerns the trigger flags (bits of the trigger message data field, see figure 3.3) also three scenarios are possible (or a hybrid solution between them):

1. A trigger flag indicating the transmission of a certain message in every bus to which the producer is connected;
2. The message should be transmitted only in the bus where the respective trigger flag was set active in the trigger message;
3. The trigger flag indicates the transmission of the message in a set of buses previously assigned to it and known by the producer.

Scenario (a) requires that the timings of all buses should be derived from the reception of the trigger message in the specific bus where it was transmitted. This scenario for the trigger

message can only be combined with scenarios 1 and 3 for the trigger flags. Its advantages are essentially the similarity of the master with the current FTT master and the additional bandwidth derived from the absence of redundant trigger messages. This additional bandwidth can be used by additional synchronous messages. The bandwidth is lost if the master is not able to schedule the maximum number of messages in every bus. As this is almost impossible to happen, this scenario becomes of reduced interest. Since the length of the synchronous window (lsw) is coded in the trigger message, slaves must compute two different values of lsw: with and without TM. Figure 3.5 shows a simplified example of this combination of scenarios for a reduced number of synchronous messages (the meanings of the additional parameters in the figure are: LEC - Length of the Elementary Cycle; ltm - Length of the Trigger Message; law - Length of the Asynchronous Window; uppercase means fixed sized windows, lower case means variable).

For now on, the subscript element indicates the bus while the superscript element indicates the elementary cycle of a specific parameter. Example: law_1^0 means the length of asynchronous window of elementary cycle zero of bus one.

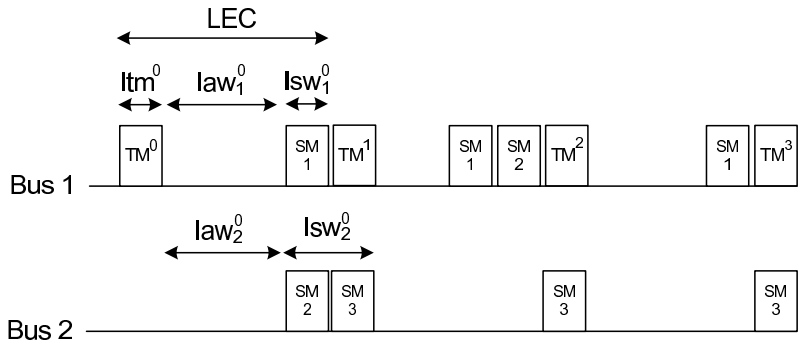


Figure 3.5: Scenario (a)

Scenario (b) presented in figure 3.6, describes the transmission of redundant synchronous messages in every bus. In this case the length of the synchronous window, lsw, is identical in every bus. It should be associated with scenario 1 or 2 otherwise bandwidth would be lost (recall that the lsw is equal in all the buses).

This scenario leads to a larger bandwidth usage than the first scenario. Also, the slave nodes have the elementary cycle well defined in all buses (comparing with scenario (a)), because all of them are separated by the trigger messages. Thus, the computational overhead for the slave's middleware is smaller than in the first scenario where the slaves must compute different elementary cycles in different buses. In addition, in scenario (b), there is not a well defined separation between elementary cycles in $N_{buses} - 1$ buses.

Finally, scenario (c) combined with strategy 2, is quite useful and flexible (see figure 3.7). In fact, if N_{buses} redundancy is not required (N_{buses} represents the number of buses of the system), bandwidth can be spared by just using the adequate number of redundant messages.

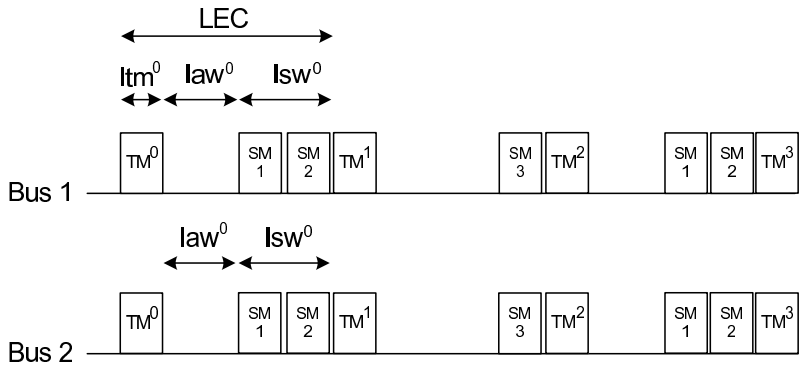


Figure 3.6: Scenario (b)

For example, in a system with 3 buses, safety critical messages can be transmitted in bus 1 and bus 2, while bus 3 is used to transmit non-critical messages. If bus 1 fails, the master can reschedule the replicas that use bus 1 to bus 3, keeping the level of redundancy at the cost of delaying or discarding non-critical messages. In what concerns the computational overhead in the slaves, this scenario leads to more overhead than scenario (b), but less than scenario (a). In this case the slaves must decode N_{buses} trigger messages in each EC while in scenario (b) it can be enough to decode just one trigger message per EC, because all are supposed to be identical.

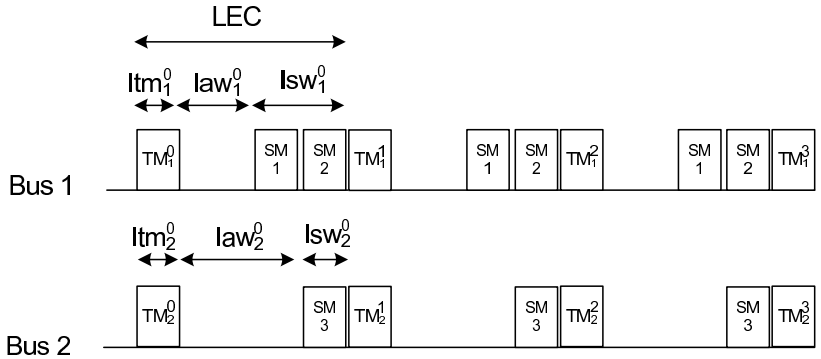


Figure 3.7: Scenario (c)

A remark must be made here. With a maximum payload of 8 bytes, *i.e.*, 64 bits in the CAN messages, the trigger message can only have 48 flags (2 bytes are used for coding additional information). If a larger number of synchronous messages is to be used, then the trigger message must be extended with at least one more CAN message. So, at least two CAN messages will be used as triggers in the beginning of the elementary cycle. This can have a significant impact in the bandwidth usage.

Also, it can suffer from the limited number of trigger flags if the system wide message identification is used, *i.e.*, if a trigger flag in position K always triggers the same message in every bus. An alternative is to use also bus aware message identification. In that case a

trigger flag in position K will trigger a different message in each bus. This approach explores well the possibility to connect some slaves to a reduced set (or just one) of the available buses.

If N_{sw} is the number of system wide flags and N_{ba} the number of bus aware flags, it is possible then to trigger the following number of synchronous messages (or tasks):

$$N_{sm} = N_{sw} + N_{buses} \times N_{ba} \tag{3.13}$$

With one trigger message per elementary cycle, N_{sw} and N_{ba} are related by:

$$N_{sw} + N_{ba} = 8 \times (DLC_{TM} - N_{reserved}) \tag{3.14}$$

Where DLC_{TM} is the DLC CAN message field of the trigger message, and $N_{reserved}$ is the number of data bytes of the trigger message reserved to other information (not the trigger flags). With one trigger message per elementary cycle, using 2 data bytes for reserved information,N_{sw} and N_{ba} are related by:

$$N_{sw} + N_{ba} = 48 \tag{3.15}$$

Using any of the strategies presented before, the remote triggering of tasks defined by *Calha et al.* [CF04, CSF05] is still valid for the case of multiple buses. This is, using any of the strategies for the trigger flags, it is possible to trigger tasks in the slaves. The trigger flags indicate the number of the task to be triggered in the current elementary cycle as for the single bus FTT-CAN version.

Asynchronous messages can fully use the N_{buses} asynchronous windows.

Bandwidth allocation for synchronous and asynchronous messages

In what concerns the bandwidth available for the synchronous and asynchronous messages the three scenarios presented before are different. For characterization of the different available bandwidth using more than one bus, first it is necessary to recall the quantification of the available bandwidth using just one CAN bus [APF02].

When using just one trigger message, the number of flags and thus, the number of data bytes of the trigger messages, depends on the number of synchronous messages to trigger. However, if the maximum is used, then a CAN message with 8 bytes in the payload will be used. The number of bits of a CAN frame depends on the stuff bits which are variable depending on the contents of the frame. According to [NHN03], the maximum CAN frame has 135 bits (including the 3 bits of the inter frame space, 'IFS' in figure 2.9). The frame duration is:

$$t_{fd} = N_{bits} \times \frac{1}{B_r} \tag{3.16}$$

And the maximum frame duration, $\max(t_{fd})$, *i.e.*, the maximum transmission time for a given bitrate by:

$$\max(t_{fd}) = 135 \times \frac{1}{B_r} \tag{3.17}$$

Thus, the maximum length of the trigger message for the single bus version, LTM, is:

$$LTM = \max(ltm^i) = \max(t_{fd}), \forall i = 0, ...\infty \tag{3.18}$$

Where i represents the number of the elementary cycle.

In FTT-CAN systems, the system designer imposes the EC duration: LEC, and a maximum length for the synchronous messages window, lsw_{max}. This last value guarantees some room for the asynchronous messages (and corresponds to a law_{min}) which then have a minimum window per EC. Thus,

$$law_{min} = LEC - LTM - lsw_{max} \tag{3.19}$$

Using a pessimistic analysis where all the messages have the maximum size, the maximum number of synchronous messages, when using the maximum length for the synchronous window in a single bus system, is:

$$\max(N_{sm}) = \left\lfloor \frac{lsw_{max}}{\max(t_{fd})} \right\rfloor \tag{3.20}$$

On the other hand, the minimum number of asynchronous messages with the maximum number of bits that can be sent in the system with one CAN bus is:

$$\min(N_{am}) = \left\lfloor \frac{law_{min}}{\max(t_{fd})} \right\rfloor \tag{3.21}$$

However, if more than one CAN bus is used, these equations will change. In case of scenario (a) the maximum number of synchronous messages in the system is:

$$\max(N_{sm}) = \left\lfloor \frac{lsw_{max}}{\max(t_{fd})} \right\rfloor + (N_{buses} - 1) \times \left\lfloor \frac{lsw_{max} + LTM}{\max(t_{fd})} \right\rfloor \tag{3.22}$$

Note that, in this scenario, the synchronous window is increased in all the buses except in the one where the trigger message is transmitted. Thus, the maximum number of synchronous messages results from the adding of the the possible number of messages in one bus with trigger message (represented by $\left\lfloor \frac{lsw_{max}}{\max(t_{fd})} \right\rfloor$ in equation 3.22) with the number of possible messages in the other buses (represented by $(N_{buses} - 1) \times \left\lfloor \frac{lsw_{max}+LTM}{\max(t_{fd})} \right\rfloor$ in equation 3.22).

For scenario (b), the number of synchronous messages which can be scheduled in all the buses is:

$$\max(N_{sm}) = N_{buses} \times \left\lfloor \frac{lsw_{max}}{\max(t_{fd})} \right\rfloor \tag{3.23}$$

This equation is the same for the case with just one bus multiplied by the number of existing buses in the system. This is because the trigger message is equal in all the buses, the maximum length of the synchronous window is equal in all the buses.

For scenario (c), the trigger messages can be different in all the buses leading to a different length of the synchronous window in each bus. Thus, the maximum number of synchronous messages is:

$$\max(N_{sm}) = \sum_{i=1}^{N_{buses}} \left\lfloor \frac{lsw_{max_i}}{\max(t_{fd})} \right\rfloor \tag{3.24}$$

Where lsw_{max_i} represents the maximum of the synchronous window of bus i.

In what concerns the minimum number of asynchronous messages in the system with more than one bus, for scenarios (a) and (b):

$$\min(N_{am}) = N_{buses} \times \left\lfloor \frac{law_{min}}{\max(t_{fd})} \right\rfloor \tag{3.25}$$

For the scenario (c) the minimum number of asynchronous messages is:

$$\min(N_{am}) = \sum_{i=1}^{N_{buses}} \left\lfloor \frac{LEC - ltm_i - lsw_{max_i}}{\max(t_{fd})} \right\rfloor \tag{3.26}$$

It should be noticed that, in this scenario, there can be a variation in the length of the trigger messages. This length becomes then bus dependent and was named ltm_i.

Strategy and trigger flags example

For an effective combination of redundancy and increased bandwidth the scenario (c) must be used, because it is the one that permits different trigger messages in different buses, enabling more flexibility. Using this scenario, and taking advantage of having a central node (master), the scheduling of the traffic can be changed online from a bus to the other, for example if one has a failure.

Also, if the system, in a moment, needs to transmit a large amount of synchronous information through the buses, the master can provide the available bandwidth using the entire synchronous window in all the buses. On the other hand, if a particular message needs redundancy, the master can schedule the same message in more than one bus.

In figure 3.8 an example of such is shown.

In figure 3.8, synchronous message 0 is transmitted in all the buses, while synchronous message 1 is transmitted in bus 1 and bus 2. The remaining bandwidth is used to send different information in all the available buses. The trigger message is different in all the

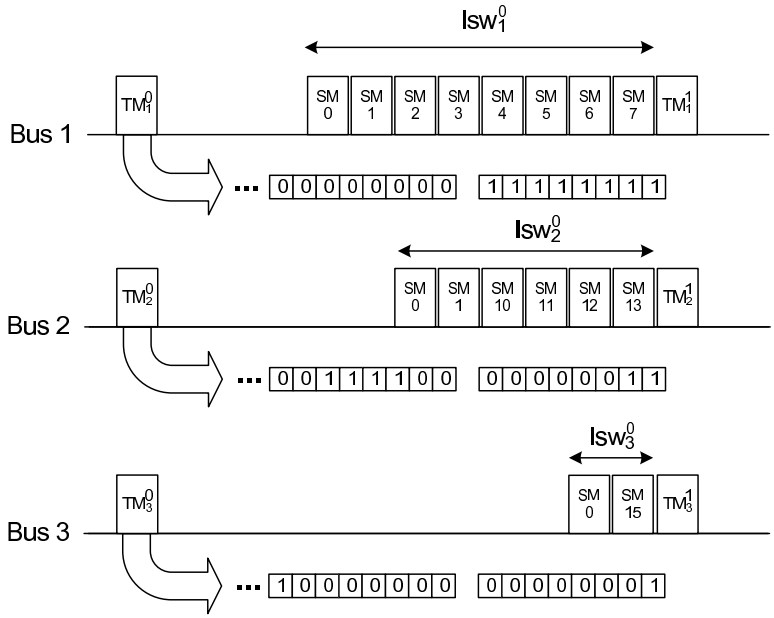


Figure 3.8: Example of trigger message with multiple buses

buses and the trigger flags have a global meaning. This is, one trigger flag has the same meaning in all the buses. For example of the trigger flag 0 triggers the synchronous message 0 in the bus 1, bus 2 and bus 3. Each bus has its own trigger message with different trigger flags. The size of the synchronous window is different in all the buses and it is encoded in the respective trigger message.

All the trigger messages are issued by the master node, that needs to be replicated to avoid the single point of failure inherited by its central nature. This master replication is addressed in next section.

3.4.4 Master replication

The master replication requirement has been defined for the FTT-CAN using just one bus by *Pedreiras* [Ped03]. Recent work by *Rodríguez-Navas et al.* [RNPR+04] defines protocols to enforce synchronization among masters. *Ferreira et al.* [FAF+03] also define a protocol to enforce consistency while a request for online changes of the SRT is performed by the slaves.

The master replication for multiple buses will be discussed here. However, the enforcement of the consistency among masters during a request for online changes of the synchronous requirement database by the slaves is out of scope of this work.

One of the advantages that could be taken from the existence of multiple buses in FTT-CAN, is the fault tolerance in the physical layer of the communication network. However, the use of a single master remains a single point of failure that must be replicated.

In figure 3.9, the architecture of the system using replicated masters is presented.

The master nodes are located in both ends of the buses and are connected to all the buses.

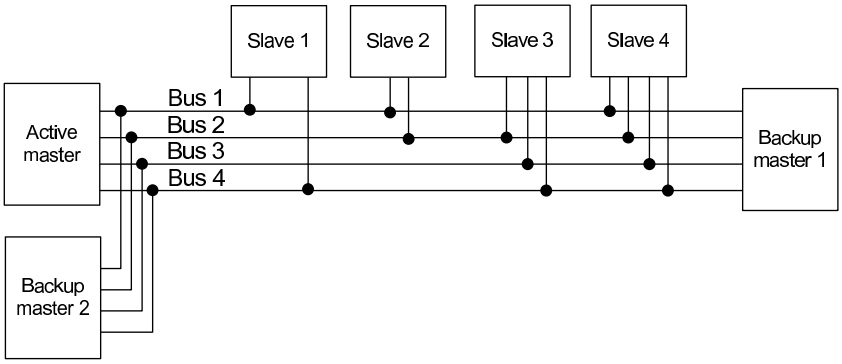


Figure 3.9: System architecture for multiple masters

They must be connected to all the buses to have a complete view of the system. The location in the buses ends aims the detection of errors that will be explained further. However, the location of the master can be elsewhere, but the error detection capability is lost (or reduced).

An important remark must be made concerning the number of active masters and their replicas relation. The number of backup masters in one end must be equal to the number of backup masters in the other end. This will maintain the system working after several masters failures. An exception is made, if there is a even number of masters (odd number of backup masters).

The master replication is considered to be a semi-passive replication as the backup masters process all requests of the slaves and works at the same time as the active master. The backup masters replace automatically the active master upon its failure. This can be assumed as an active redundancy, but, on the other hand, the active and backup masters do not transmit trigger messages at the same time.

Notice that, if the system has just one master or the backups are located at the same end as the active, the system works in a degraded mode regarding the bus error detection capabilities. Thus, the location of the masters (backups divided among both ends) is important to maintain the bus error detection capabilities in case of a failure of the active master.

Thus, master replication is essential to the error detection and replacement. In the next section the master replacement and the bus replacement mechanisms, in case of an error, are presented.

3.5 Masters and buses: errors and replacement

3.5.1 Introduction

Concerning the masters node and the buses of the system, it is possible to take advantage of their multiplicity in order to increase the dependability of the system. The use of a replicated master scheme improves the dependability of the system. Besides that, it also

enables detection of bus failures and master failures.

The use of more than one bus in the system enables the replacement of a faulty bus. In that way, if a bus has a problem the system can replace this bus. The messages that are normally sent on the faulty bus must be switched to a non-faulty bus. This implies a mechanism of faulty bus detection and also a mechanism to switch messages from the faulty bus to a non-faulty one.

3.5.2 Fault model and assumptions

The following fault model and assumptions were considered in this work:

- The multiple buses architecture of FTT-CAN is able to detect partitions in one or more buses, but not in all buses simultaneously;
- Only transient errors that affect the bus in more than one elementary cycle plus a guard time (further denoted as $TMTW$, Trigger Message Transmission Window) can be detected and treated by the system;
- The system must have at least one bus free of errors at any time. This ensure that the masters have a bus to communicate between them the control data and recovery procedures. It is also assumed that the bus stubs which connect to masters are free of errors;
- Possible electromagnetic interference does not occur in all the buses at the same time. This means that no simultaneous problems will occur in all the buses. If the buses follow different paths this assumption is quite reasonable;
- There is only one active master at any given moment;
- If a message is transmitted from one master and is received by a backup master on the other end of the bus, then the bus is not partitioned;
- The master nodes have a fail silent behaviour both in time and value domains;
- Besides detecting bus partitions it is assumed that the masters also detect stuck-at faults.

3.5.3 Detectable faults

The system can detect the following faults:

- Medium partition fault. The medium partition occurs in such a way that the bus is broken in two or more sub networks. As a result, two nodes that are in different sub networks are not able to communicate. The detection of the bus partition is based on

the systolic nature of the trigger message. In section 3.5.5, the bus partition detection will be explained. This fault is detected due to the error it produces: the missing of the trigger message in the bus;

- Stuck-at fault. Stuck-at faults can be dominant or recessive. Since the CAN bus is an "AND" of all the contributions of all the nodes, then the stuck-at recessive fault does not cause any error in the system. If there is a stuck-at dominant fault, the error counters of the master node will increase and the fault can be detected. The number of consecutive dominant bits necessary to detect an error is presented in [RVA99] and [BPRNA06]. One way to detect the stuck-at recessive bit is to check the 'ACK' bit sent by individual nodes. However, this is impossible in a bus architecture. In a star architecture [BPRNA06] the authors propose this method for detection the stuck-at recessive fault;

- Bit-flipping fault. This fault occurs whenever a system component (node or medium) starts behaving in an uncontrolled way in the value domain by sending random bits. The dominant bits of the uncontrolled stream will overcome the correct recessive bits of a node that tries to send a correct frame;

- Master fault. As explained before, the master has a fail-safe behaviour, in that case the system can detect a master fault and replace it.

The system cannot detect:

- Babbling-idiot fault. It occurs whenever a node sends incorrect messages in the time domain [BB03]. It can be caused by hardware or software. An example of such fault appear when the software enters in an infinite loop sending constantly messages to the bus. The babbling-idiot fault can be minimized using bus guardians such as the ones presented in [BB03] and [FAM+03]. *Ferreira et al.* [FAM+03, Fer05] define bus guardians to prevent the babbling-idiot fault in FTT-CAN systems. These bus guardians are still valid in case of the multiple buses system with the necessary adaptations. The proposed architecture does not consider the use of bus guardians, thus it is not able to detect babbling-idiot faults.

3.5.4 Master errors: replacement and analysis

To deal with the trigger messages omissions, *Ferreira* [Fer05] defines the Trigger Message Transmission Window ($TMTW$). The trigger message can be retransmitted during this window if a transient fault occurs during its transmission [Fer05]. In case of a delay in a trigger message (within $TMTW$), the next trigger message will not be delayed. Thus, the elementary cycle will be reduced. This is an important property because all the trigger messages must be issued at the same time in all the buses, even with a retransmission in one

of them. In that way just one trigger message is not synchronized with the others, but in the next activation it will be synchronized again.

As it was stated on the fault hypothesis, the system has always a non-faulty bus working. This means that all trigger messages received by the backup masters were correctly transmitted by the active master (at least in one bus). Thus, if no trigger message is received in any bus during the trigger message transmission window, that means the active master has failed and it needs to be replaced by a backup master. Figure 3.10 depicts the state diagram of the master replacement protocol. Also, the bus error detection is included in figure 3.10, since it cannot be dissociated. The bus error will be discussed later.

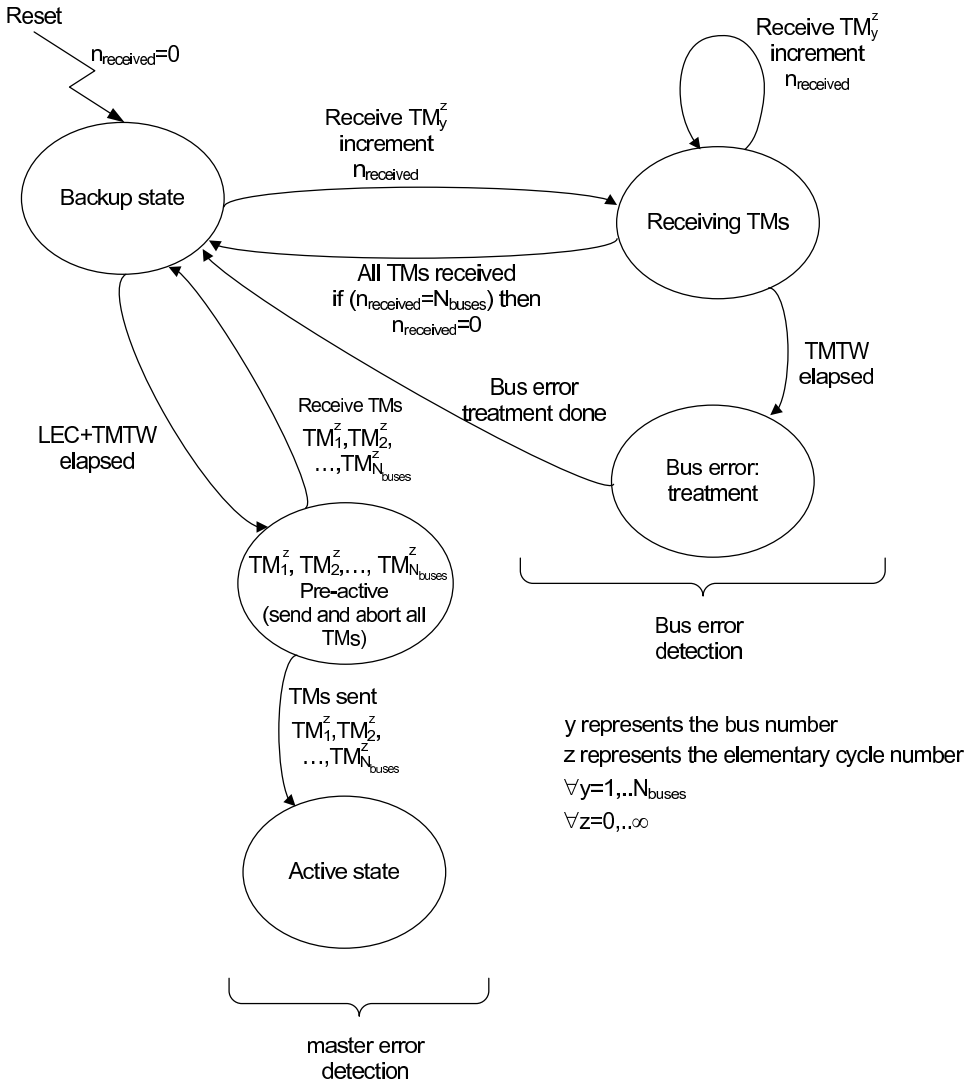


Figure 3.10: Master replacement state diagram

If the backup masters do not receive any trigger message, in any bus, during the $TMTW$, then all backup masters will try to become active by sending a trigger message in all buses, aborting the transmission in the next instruction cycle. The first backup master that suc-

ceeds transmitting the trigger messages will become the active master. The master knows that it now the active master because it senses a transmission interrupt prior to the receive interrupt. The others masters will stay in backup mode because they will succeed to abort the transmission of the trigger messages.

Summarizing, master replacement occurs whenever no trigger messages are received within $TMTW$. Notice that, if only one trigger message is missing, then a backup master assumes that the particular bus is faulty. Figure 3.11 depicts the bus master replacement protocol.

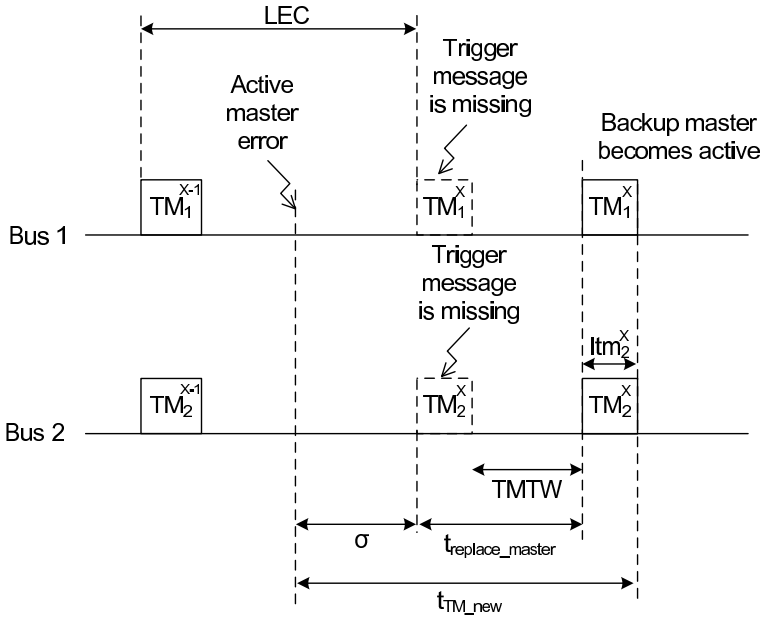


Figure 3.11: Master replacement protocol

As it can be seen in figure 3.11, after the error of the active master, the trigger messages TM_i^X that should be issued by the active master are missing (where i is the bus number, thus an integer and $i \in [1, N_{buses}]$ and X is the number of the missing trigger messages). After elapsing the $TMTW$, the backup master will transmit its own trigger message in all the buses, becoming the active.

The time required for the replacement of the master is given by (refer to figure 3.11):

$$t_{replace\_master} = TMTW + \max(ltm_1^X, ltm_2^X, \ldots, ltm_{N_{buses}}^X) \tag{3.27}$$

The worst case corresponds to the duration of the trigger message transmission window plus the maximum duration of the trigger message, which is:

$$\max(t_{replace\_master}) = TMTW + LTM \tag{3.28}$$

The time since the active master error to the new trigger messages are send by the backup

master is:

$$t_{TM\_new} = t_{replace\_master} + \max(ltm_1^X, ltm_2^X, \ldots, ltm_{N_{buses}}^X) + \sigma \tag{3.29}$$

This time corresponds to the time the system will be without a master. The worst case situation, that occurs when the error happens just after the trigger message $X-1$, which correspondent to:

$$\max(t_{TM\_new}) = LEC + TMTW + LTM \tag{3.30}$$

Taking into account the stuff bits difference between trigger messages $X-1$ and X, this equation can be re-written to:

$$\max(t_{TM\_new}) = LEC + TMTW + LTM + (\max(N_{bits}) - \max(N_{bits})) \times \frac{1}{B_r} \tag{3.31}$$

Note that, the value t_{TM_new} can be different for different buses, because different buses imply different trigger messages and thus, different number of bit stuffing. Further in this dissertation this issue will be discussed in detail.

If there are more than one backup masters in the system, the one that will become active will be the one that first starts to transmit its TM and not the one with the highest priority. However, if two (or more) backup masters start transmitting at the same time, the one with the highest priority TM will win the CAN arbitration and will become active. The priority of the trigger message is dependent of the identifier (ID) of the master, that is coded in the CAN message identifier.

Regarding the masters synchronization, the same method defined by *Ferreira et al.* [FAF+03] for the single bus system is still valid. However, modifications must be done to accommodate the multiple buses. This issue is considered to be out of scope of this work.

In the previous version of FTT-CAN (with just one bus), the backup master will always try to transmit its own trigger message in the middle of the trigger message sent by the active master, aborting in the next instruction. In case of the active master sends the trigger message, the backup master will always succeed in aborting the transmission of its own TM.

For the multiple buses case the backup master cannot follow the same strategy because it will lead to an inconsistent state. This state is reached when one master becomes the active master for one bus and other master becomes the active master for other buses. This can occur in case of an error in one bus, since just one trigger message is missing.

Comparing with the strategy adopted for the case of FTT-CAN with just one bus [FAF+06], where the replacement time is about one half of the trigger message [MSF+06], the master replacement time is considerably higher in case of a multiple buses FTT-CAN

network due to the used strategy that must wait the $TMTW$ to issue the new trigger message.

3.5.5 Bus errors: replacement and analysis

The use of more than one master with multiple buses permits the detection of errors [SFF07b], such as the master error (explained before) and bus errors. As in the case of the master replacement protocol, the bus partition error detection mechanism also takes advantage of the systolic nature of the trigger message. A bus fault is detected whenever a backup master receives, in a given elementary cycle, less trigger messages than the number of buses. The bus where a trigger message was not received is considered faulty. As depicted in figure 3.12, trigger message TM_2^X cannot be delivered to one backup master due to an error at bus 2. Recall that at least one backup master is at the other end of the bus. This means that if an error in the bus occurs, at least one backup master will detect it.

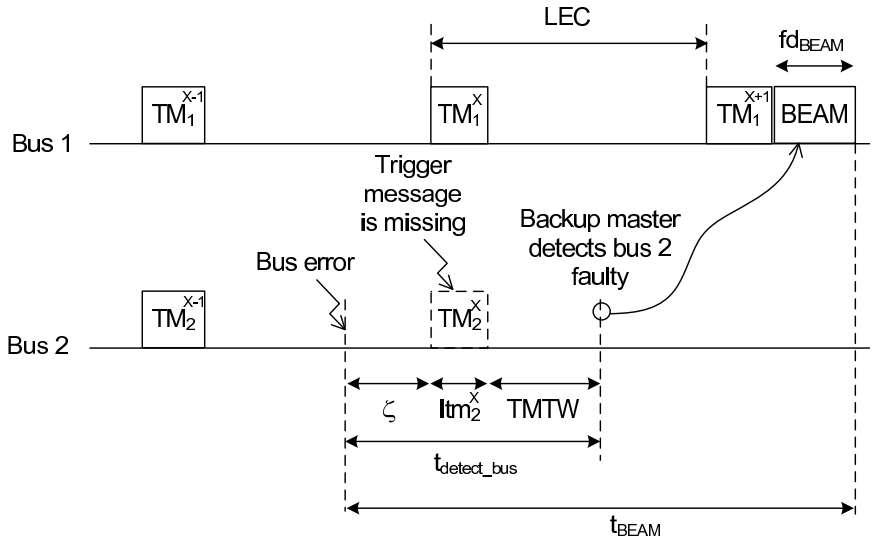


Figure 3.12: Bus error detection mechanism

The backup master will detect the bus error after the trigger message transmission window elapses. Meaning that:

$$t_{detect\_bus} = TMTW + \zeta + ltm_I^X \tag{3.32}$$

where:

- I corresponds to the number of the erroneous bus (in case of figure 3.12, $I = 2$);
- X corresponds to the number of the missing trigger messages;
- ζ represents the time from the bus error until the missing trigger message.

The worst case detection time occurs when the bus becomes faulty right after the reception of the previous trigger message, thus:

$$\max(t_{detected\_bus}) = TMTW + LEC \tag{3.33}$$

Where LEC is the length of the elementary cycle. Note that, this equation do not take into account the bit stuffing.

Notice that is considered that a permanent fault last for longer than $LEC - LTM$ (where LTM is the maximum length of the trigger message).

Note that the previous protocol and analysis assumes that the microcontroller acting as an active master can send the trigger messages at exactly the same time in both buses. However, in practice, this is not true and, using two or more buses, the time taken by the active master to send the trigger messages in all buses is not equal. This will be discussed further.

After the detection of a bus error, the backup master informs the occurrence to the active master, transmitting a high priority asynchronous message (named BEAM - Bus Error Asynchronous Message) to the active master using all the non-faulty buses (refer to figure 3.12).

In figure 3.12, the time t_{BEAM} represents the delay from the the bus error until the reception of the BEAM. Thus:

$$t_{BEAM} = LEC + \zeta + fd_{BEAM} + \max_{i \neq I}(ltm_i^{X+1}) \tag{3.34}$$

where fd_{BEAM} represents the BEAM frame duration and I the number of the erroneous bus. The worst case value for t_{BEAM} happens if the bus error occurs just after TM_i^{X-1}. Meaning that:

$$\max(t_{BEAM}) = 2 \times LEC + fd_{BEAM} \tag{3.35}$$

If the stuff bits are take into account, the maximum value for t_{BEAM} are:

$$\max(t_{BEAM}) = 2 \times LEC + \max(fd_{BEAM}) + (\max(N_{bits}) - \min(N_{bits})) \times \frac{1}{B_r} \tag{3.36}$$

where $\max(fd_{BEAM})$ is the maximum duration of the BEAM message. According to figure 3.13 it is a CAN frame with $DLC = 2$ thus, using equation 3.2 $\max(fd_{BEAM})$ value can be found.

If the system has more than one backup master, any one of them can consider that a particular bus is faulty. Thus, several BEAM messages could be transmitted in the same elementary cycle. However, if a a backup master receives a BEAM message it can compare this BEAM with the one it will send. If both are equal it can abort the transmission, sending it, if both are different. The structure of a BEAM asynchronous message is presented in

figure 3.13.

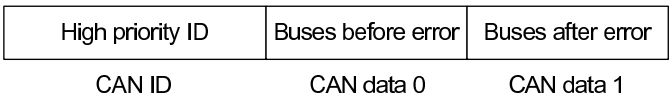


Figure 3.13: Bus error asynchronous message (BEAM)

A high priority identifier is assigned to this BEAM asynchronous message. Thus it will be transmitted before any regular asynchronous messages issued by slave nodes. Notice that the message sent by the backup masters encodes the state of the buses before the error and right after the error in a bitwise manner, *i.e.*, the non-faulty buses are coded as '1' while the faulty ones are coded as '0'. When compared with the possibility of encoding only the faulty buses, this approach has two advantages:

- After receiving these messages, the active master can easily determine which bus (or buses) is (are) faulty, by executing a logic "AND" between the buses state before and after the error (CAN data 0 and CAN data 1 in figure 3.13);
- The active master obtains a global view of the buses that is consistent with the view of the backup masters.

After receiving this message the active master and the backups master should change the buses where the synchronous messages were issued and recompute the bus scheduling. This reconfiguration will be explained next.

Reconfiguring the buses

The bus reconfiguration is done at the active master side just after the reception of the BEAM message and at the backup master side right after the transmit interrupt of this message. This operation needs to be done in all masters, both active and backup, to maintain the consistency among masters [SFF07c]. However, if this is not done in a backup master due to some unforeseen reason, this is not so critical because such a backup master will declare itself as unsynchronized and will ask for a re-synchronization using the protocol defined in [FAF+06].

For the bus reconfiguration, the synchronous requirement table in the masters must be changed. This change will be made in the fields of each synchronous message that support the multiple buses (explained more detailed further in this dissertation). Each synchronous message has three characterization fields for bus management, which are:

- Initial Bus Allocation (IBA). This byte indicates the buses where the message is initially transmitted;

- Physically Connected Bus (PCB). This byte indicates the buses where the message can be issued;
- Current Bus Allocation (CBA). This byte indicates which are the current buses (or bus) where the message must be issued.

To perform the bus change for specific synchronous messages some operations must be performed by the masters (refer to figure 3.14). The operational sequence is:

1. Reception of the BEAM message from the bus (causing an interrupt). Denoted as 'INT' in figure 3.14;
2. After the scheduling (denoted as 'SCH' in figure 3.14, that begins during the TM_i^{X+1}) for the next EC (for the EC $(X+2)^{th}$) has finished, the SRT must be copied to a new table. This ensures the mutual exclusion during the process. Note that this copy must be made after the scheduler finishes since when the BEAM message arrives it is possible that the scheduler is already running (during the time of BEAM) and making operations on the synchronous requirements table. This copy operation is denoted as 'CPY' in figure 3.14;
3. Perform a logic "AND" between the data byte 1 of BEAM message, the CBA byte (see figure 3.13) and the PCB byte (refer to section 4.2.1) of each synchronous message stored in the copy of the SRT. This is a simple operation that just takes few instructions. However, it must be performed for all synchronous messages of the system. Denoted as 'Logical "ANDs"' in figure 3.14;
4. After the scheduler action for the $(X+K+1)^{th}$ EC, the changed SRT is copied back (K is the number of elementary cycles the bus changing operation takes to complete and $K \in \{\mathbb{N} - \{0\}\}$). This ensure that, in the next EC, the schedule will take into account the new bus configuration (denoted as 'CPB' in figure 3.14);
5. In TM_1^{X+K+2} the scheduling with the messages from bus 2 will be present.

Since the bus changing operation (denoted as 'Bus changing' in figure 3.14) will use K elementary cycles, the messages that are issued in bus 2 will be masked for $K+2$ elementary cycles. This is the time taken for the bus changes of all messages upon a bus failure. As a result, the time elapsed from the error to the new scheduling is:

$$t_{re\_scheduling} = LEC \times (K+2) + \zeta + \max_{i \neq I}(ltm_i^{X+K+2}) \tag{3.37}$$

Where I is the number of the erroneous bus.

The worst case occurs when the bus error occurs after the trigger message $X-1$, thus:

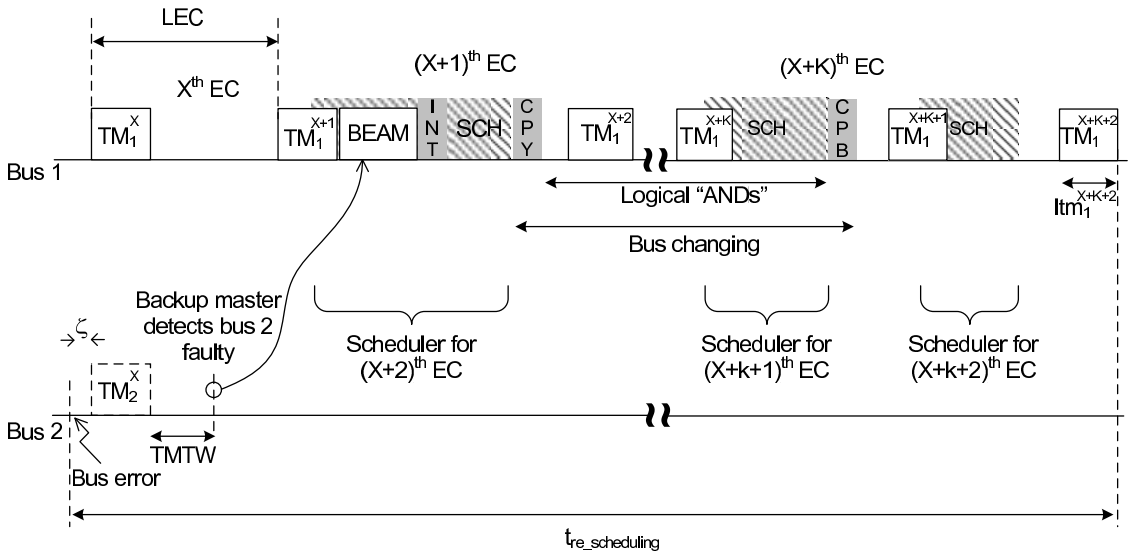


Figure 3.14: Bus changing time flow

$$\max(t_{re\_scheduling}) = LEC \times (K+2) + LEC \tag{3.38}$$

However, if the CAN bit stuffing is take into account, this equation can be re-written as:

$$\max(t_{re\_scheduling}) = LEC \times (K+2) + LEC + (\max(N_{bits}) - \min(N_{bits})) \times \frac{1}{B_r} \tag{3.39}$$

Note that, synchronous messages can be lost. These messages are the ones that will be transmitted in the faulty bus during the ζ interval.

Bus error detection in two or more buses and practical issues

In the previous sections the error detection for one bus in a multiple buses system was presented. However, if more than one bus fails at the same time (or, more precisely, at the same elementary cycle) some minor differences must be introduced to the previous protocol.

The active master cannot trigger at the exact same time two or more messages in different buses, since, in the current solution, it is just one microprocessor controlling more than one CAN controller (this can be changed). Also, the size of the trigger messages are different for each bus, since the scheduling embedded in the data field are different and the number of the stuff bits may also be different (briefly discussed in previous section). These two differences must be accommodated by the detection protocol. Figure 3.15 depicts this case.

For the case of more than one erroneous buses in the same EC, equation 3.32 can be re-written to:

$$t_{detect\_bus_I} = TMTW + \zeta_I + ltm_I^X + \delta \tag{3.40}$$

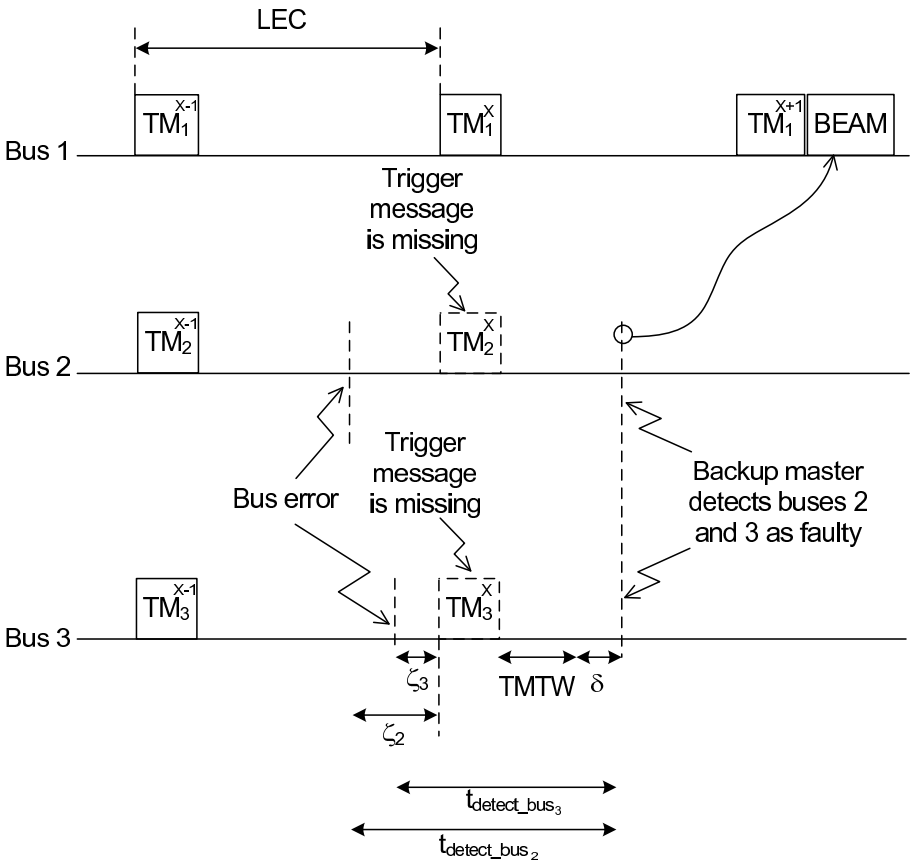


Figure 3.15: Multiple errors detection

where:

- I represents the number of erroneous bus. Thus, in case of figure 3.15, $I \in \{2, 3\}$;
- $t_{detect\_bus_I}$ is the time to detect the bus error in bus number I;
- ζ_I is the time from the bus fault until the expected trigger message in bus number I.

Note that, all the equations presented before for a single error in one bus, still valid for the case of more than one erroneous buses in the same EC. However they need to be adapt.

In the example of figure 3.15 bus 2 and 3 become faulty at the same elementary cycle. The backup master waits the trigger message transmission window ($TMTW$) and must also wait δ, *i.e.*, δ represents the necessary time for the active master to send all the trigger messages in all buses due to the incapacity of a node to send CAN messages simultaneously in several buses. δ will increase with the increasing of the number of buses of the system. In that way:

$$\delta = (N_{buses} - 1) \times \varepsilon \tag{3.41}$$

where:

- N_{buses} is the number of buses in the system;

- ε is the maximum time taken by the processor between the transmission of two consecutive trigger messages in two different buses.

Assuming that the messages are triggered in sequential instructions in all the buses by the microprocessor and there is no delay between the trigger of the message and the beginning of its transmission on the bus, ε is the instruction cycle time of the processor. In fact, δ is the delay imposed by the processor to trigger all the messages in all the buses.

This is presented in the figure 3.16.

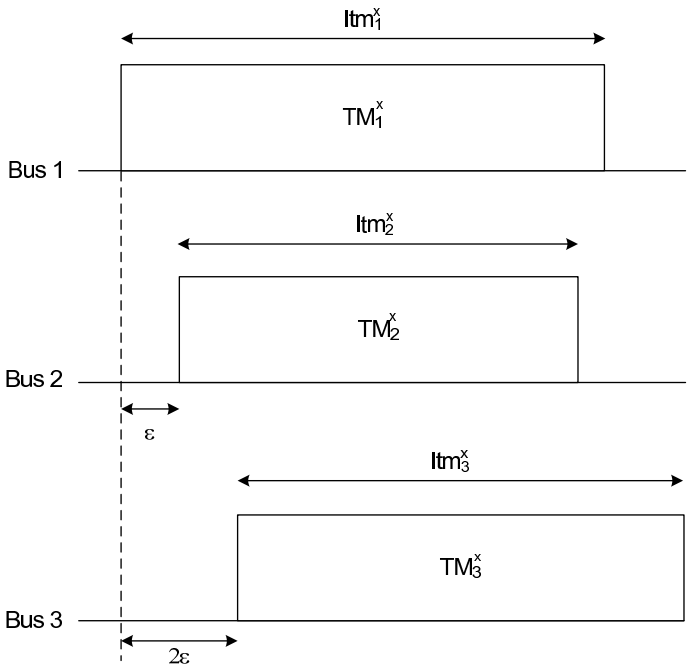


Figure 3.16: Delay in transmission of the trigger messages in different buses

The delay between the sending of several trigger messages among all the buses is represented by $\delta = (N_{buses} - 1) \times \varepsilon$. This means that the partial delays are equal for all the buses (as it is represented in figure 3.16). If the partial delays, ε, are not equal and the bit stuffing are taken into account (because the trigger messages can occupy different time among different buses), the equation becomes:

$$\delta = (\sum_{i=2}^{N_{buses}} \varepsilon_i) + (\max(N_{bits}) - \min(N_{bits})) \times \frac{1}{B_r} \tag{3.42}$$

Where ε_i is the delay between the beginning of the trigger message in bus i and the beginning of the last trigger message of the current EC in the other buses.

Additionally, within a trigger message, there could be clock skew among different clocks in the several masters. In practical implementations this time must also be taken into account by adding a delay in equation 3.42 that represents the maximum skew that can occur among all the masters.

3.6 Final remarks

To detect the bus errors and also to permit the master replacement, the system cannot act similarity to the solution proposed for a single bus. This is, the backup master will transmit and abort, in all elementary cycles and in all buses, its own trigger message. This transmission is done at the middle of the time reserved for the trigger message transmitted by the active master. If the backup master succeeds the transmission in all the buses, it becomes the active. However, if it just succeeds in one bus or in a set of buses it declares the buses as faulty.

In fact, this strategy will lead to a system where two active masters coexists. And, because the masters are located at the bus ends, a set of slaves will have a master, while the other slaves will have other master. The slaves cannot communicate with each others in case of a bus partition.

It is also true that the design assumption of only one active master per elementary cycle (section 3.5.2) is allowed. With this strategy explained before, there could be an active master in one bus and other active master in other bus(es). On the other hand, the replacement time with this strategy will be smaller. In the worst case it will be one half of the maximum time of a trigger message (as for the case of FTT-CAN with one bus).

Comparing the proposal with other protocols presented in chapter 2, there are several advantages in this proposal, namely:

- Number of buses only dependent on the number of the CAN controllers the microprocessor can support. In other solutions, this is limited to two buses. Example of such are Columbus Egg Idea and CANdor. Other solutions can use more buses but do not use them to provide bandwidth improvement. If so, this feature is manage by a higher layer;

- One can consider active redundancy in buses. This proposal uses active redundancy while other similar proposals use passive redundancy. There are many application domains that could not cope with the delays resulting from passive redundancy. Example of such applications domains is avionics;

- Semi-passive redundancy in the master replication. All the backup masters are able to replace the active master (leader-follower strategy). All of them have an updated copy of the synchronous messages requirement table. In addition, the master replication is proposed with an unlimited number of masters. Node replication is not considered in some of the presented protocols;

- It can be generalized to other protocols different from FTT-CAN. The proposed protocol can be generalized to master-slave architectures. If there is not a periodic message (like the trigger message in FTT-CAN) it can be introduced;

- Fail-safe system. If there is a failure, it responds in a way that will cause no harm to other systems and humans. In case of a failure, the system replaces the master or the bus without notice. The system still work, however it can switch to a degraded mode.

Further in this dissertation, in chapter 6, a proposal to improve dependability and bandwidth in CAN will be presented. It can be seen as a generalization of the presented work for non master-slave systems using additional hardware components.

In the presented proposal, there are some issues that were not considered because they are out of the scope of this work, namely: the asynchronous messaging system and the multiple buses slave nodes. They are considered to be future work, and will be presented with more detail in the conclusions, chapter 7.

3.7 Conclusions

As it was pointed out at the beginning of this chapter the single bus FTT-CAN architecture has some limitations related with redundancy support and limited bandwidth. These two limitations arise from the CAN network and, in the case of the bandwidth, it has an extra overhead penalty due to the FTT-CAN protocol.

This chapter has presented some possible improvements that can be made to the FTT-CAN system to deal with more than one bus. These improvements do not affect the essentials of the protocol and existing slave nodes are still usable for the system with more than one CAN bus. However, they cannot take advantage of all the features provided by the new architecture. The FTT-CAN master cannot be the same, because it must control more than one bus to have a global view of the system. The additional buses can be used to improve both the fault tolerance and the available bandwidth.

This chapter has also addressed the issue of having multiple trigger messages in multiple buses and several scenarios have been proposed and discussed, resulting in different degrees of flexibility and available bandwidth both for the synchronous and asynchronous messages.

The FTT-CAN master node replication was considered, and a new master replication scheme has been proposed. The master replicas are located at both ends of the buses and a leader-follower behaviour is proposed for master replication where the active master is the leader and the other replicas are the followers. The location of the masters in conjunction with the systolic nature of the trigger message permits the detection of errors in the buses. When a bus error is detected, a backup master informs the active master of this event, and it will switch the messages to non-faulty buses. The backup masters are also able to substitute the active master whenever it fails to transmit trigger messages in all buses. In our case, the master replication is considered to be a semi-passive replication because only one master is active at a time and there is a active master and backups.

For the case of the bus replication it is considered an active redundancy, since, when a

message is replicated is sent in the respective buses at the same time (in fact, at the same elementary cycle).

Chapter 4

FTT-CAN Implementation

4.1 Introduction

This chapter describes the implementation of FTT-CAN from scratch. There were some previous implementations of FTT-CAN [APF02], however, they targeted old microcontrollers and the porting effort to more recent microcontrollers posed some challenges. Thus, it was decided to design and implement a newer version of FTT-CAN optimizing the software architecture to handle multiple buses. The design and implementation effort started with the single bus, single master FTT-CAN. Afterwards, the master node was adapted to handle master replication and, finally a master node version to handle multiple buses has been designed. A single bus slave node was also implemented.

This chapter presents the data structures and the Application Programming Interface (API) based on the FTT-CAN requirements along with the internal modules of the master node.

Since the target environment of FTT-CAN is the embedded applications, where the low computing power microcontrollers are common, the evaluation of the computational overhead of the implementation is an important issue that is also addressed in this chapter.

4.2 Data structures and API

A distributed system based on the FTT paradigm can handle periodic messages and tasks and aperiodic messages. The periodic messages and tasks are schedule by the master node according to their parameters. In general, a periodic message, σ_j, can be characterized by its period, worst-case transmission time, and relative phase [Cal06]. In that way, the set of messages in the system is:

$$\psi = \{\sigma_j(C_j, T_j, Ph_j), \forall j = 1, ..., n\} \tag{4.1}$$

Where:

- T_j is the message period;
- C_j is the worst-case transmission time;
- Ph_j is the relative phase.

According to *Calha* [Cal06], from these parameters an extended set of parameters can be derived to define the set of instances of one message:

$$\psi = \{\sigma_{j,k}(C_j, T_j, Ph_j, D_j, PT_j, CTL_{j,i}, d_{j,k}, r_{j,k}), \\ \forall k = 1, ..., nInst_j, \forall j = 1, ..., n, \forall i = 1, ..., m\} \tag{4.2}$$

where the additional parameters are:

- D_j is the deadline measured relatively to the release instant;
- PT_j is the producer task;
- $CTL_{j,i}$ is the consumer task list;
- $d_{j,k}$ is the absolute deadline;
- $r_{j,k}$ is the release instant;
- $nInst_j$ is the number of instances of a message σ_j;
- n is the number of messages;
- m is the number of tasks (as will be defined bellow for the task set).

However, this set of message parameters is defined to support a global synchronization in the holistic schedule defined by *Calha* [Cal06]. In the present work, the consumer and producer task list are not important, since the master does not need to know this information to implement the FTT-CAN protocols. In this way, the set of messages is defined by:

$$\psi = \{\sigma_{j,k}(C_j, T_j, Ph_j, D_j, d_{j,k}, r_{j,k}), \forall k = 1, ..., nInst_j, \forall j = 1, ..., n\} \tag{4.3}$$

The presented work is focused on the FTT-CAN message system. However, the implementation also includes the task triggering system. The tasks are scheduled by the master node but, they run in the slaves. The master do not know the state of the tasks or they dynamic parameters, such as the computational overhead or deadline.

For the case of interactive tasks, the set of tasks $\tau_{i,k}$, is defined in equation 4.4. Interactive tasks perform a closed-loop control and communicate with other tasks (producing or consuming one message, as defined in [Cal06]). Conversely, stand-alone tasks do not communicate with other tasks.

$$\Gamma = \{\tau_{i,k}(C_i, T_i, Ph_i, D_i, N_i, MP_i, MC_i, d_{i,k}, r_{i,k}), \forall k = 1, ..., nInst_i, \forall i = 1, ..., m\} \quad (4.4)$$

With the necessary adaptations, the parameters defined for the messages are identical to the parameters for the tasks (*e.g.* C_j is the worst-case transmission time of a message, while C_i is the worst-case computational time of a task).

Where the additional parameters are:

- N_i represents the node where the task runs;
- MP_i is the message produced;
- MC_i is the message consumed;
- $nInst_i$ is the number of instances of a task τ_i;
- m is the number of tasks.

For the present work, the master does not need to know in which node the task will run (the master node will just trigger the task), nor the messages it produces or consumes. Thus, the tasks are stand-alone where N_i parameter is not considered. Also, the worst-case computational time is not required, because it is assumed that all the tasks are limited to the time available for it execution. This is, all tasks triggered in the slaves will finish their execution prior to the beginning of the execution of the next task (same or other task). The absolute deadline and release instants are constants at the master, thus can be omitted.

In that way, the set of tasks are defined as:

$$\Gamma = \{\tau_{i,k}(T_i, Ph_i, D_i), \forall k = 1, ..., nInst_i, \forall i = 1, ..., m\} \quad (4.5)$$

The master and bus redundancy protocols are based on the periodic nature of the trigger message, enabling the detection of master and bus faults. The trigger message also conveys the information about the messages and tasks that should be produced during the next elementary cycle. The trigger message is a set of trigger flags for tasks ($\beta^l_{q,o}$ in equation 4.6) and messages ($\xi^l_{q,p}$ in equation 4.6). The trigger message can be expressed as:

$$\Upsilon^l_q = \{q, S^l_q, \Theta^l_q(\beta^l_{q,o}, \forall o = 1, ..., n\_tasks^l_q), \Lambda^l_q(\xi^l_{q,p}, \forall p = 1, ..., n\_mess^l_q)\},$$
$$\forall l = 0, ..., \infty, \forall q = 1, ..., N_{buses} \quad (4.6)$$

Where:

- q is the bus number where the trigger message is transmitted;

- S_q^l is the size in bits occupied by the trigger message in the bus;
- Θ_q^l is the set of the tasks scheduled for the present elementary cycle, whose number is $n\_tasks_q^l$;
- Λ_q^l is the set of the messages scheduled for the present elementary cycle, whose number is $n\_mess_q^l$;
- l represents the elementary cycle number.

The application programming interface provides the necessary interface for the developers to deploy their applications and the functions to manipulate the middleware of the masters and the slaves.

Figure 4.1 presents the implemented communication stack. The applications access to the API which interacts with the FTT middleware (see figure 4.1). The FTT stack also controls, at the data link layer, features of some of the others layers proposed in the OSI model [Zim80, ISO94]. Example of such feature is the message filtering implemented in software. In that way there is a upper data link layer (see figure 4.1) that is part of the FTT stack. This upper data layer communicates with the lower data link layer (see figure 4.1).

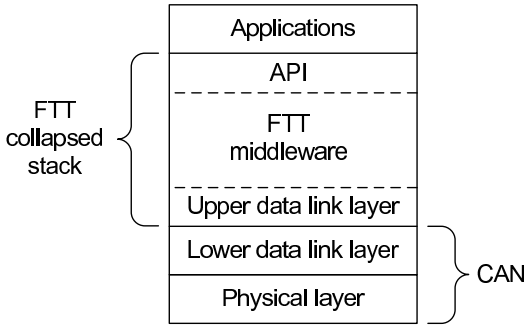


Figure 4.1: FTT stack

The master node does not need any application to work. The FTT system will provide all the functionalities that the master needs. Eventually, the master can run other applications, however, this could be undesirable in case of the use of a low processing power microcontroller.

Concerning the initial state, there are three different types of FTT-CAN master nodes:

- Master with Initial Configuration (MIC). The master parametrize the synchronous messages flows and boots the FTT protocol. There are no other applications running on the FTT master;
- Stand-Alone Master (SAM). The master just boots the FTT protocol. The synchronous messages parametrization is done online upon slave requests. This kind of master can be used in any system;

- Master with Online Reconfiguration (MOR). The master may perform the parametrization of the synchronous messages and boot the FTT protocol. The application can change the synchronous flows online, for example to respond to an external event.

Thus, the following operations can be done by the master:

- Initialization and parametrization;
- Add, remove or change variables (messages or tasks).

Applications running on the slave nodes can perform the following operations:

- Initialization and parametrization;
- Add variables, messages or tasks, upon master approval;
- Request parameter changes (for messages or tasks) to the masters;
- Send/produce messages and receive/consume variables (messages or tasks).

Further in this chapter the application programming interface to perform the presented operations will be described in detail.

4.2.1 The synchronous requirement database

The main memory area managed by the FTT middleware is the Synchronous Requirement Database (SRDB) which includes several tables and which stores:

- The synchronous messages requirements and properties stored in the Synchronous Requirement Table (SRT);
- The tasks requirements and properties are also stored in the Synchronous Requirement Table (SRT). In this table each line corresponds to a message or task (will be explained further);
- Asynchronous requirements. The asynchronous requirements are stored at the Asynchronous Requirements Table (ART) and at the Non-Real-Time Requirements Table (NRT). The ART stores the properties of asynchronous messages that may have timeliness requirements (such as alarms). The NRT stores the size of the longest non-real-time message produced by each node (this can be useful to determine the maximum size of the synchronous and asynchronous window). However, in this implementation, the asynchronous requirements are not taken into account, and thus, this will not be discussed further in the implementation.
- Configuration and Status Record (CSR). This data structure stores parameters related with the physical layer and with the maximum dimension of the system:

- CAN bitrate;
- Maximum length of a CAN message, in number of bits, including the stuff bits (this is also the maximum number of bits of the trigger message, $\max(N_{bits})$);
- Maximum number of synchronous messages. This value is defined by the system developer and depends on the maximum number of trigger flags the system can have and the tasks the master can trigger;
- Maximum number of system tasks. This value is calculated by the master and is dependent of the maximum number of trigger flags the system can have and the maximum number of messages defined by the developer;
- Length of the elementary cycle (LEC);
- Maximum size of synchronous window (lsw_{max_i}), where i is the bus number;
- Implementation dependent auxiliary parameters, such as timers pre-scalers and other low-level parameters.

The defined message and task parameters must be mapped in the SRT. Some of the task and message parameters are identical (such as the period, the first release instant and the deadline measured relatively to the release instant) and can be mapped into the same field in the SRT. However, the messages have parameters that the tasks do not and thus, need additional fields in SRT.

In table 4.1, the mapping of the messages and tasks parameters to the SRT fields is presented. Each variable corresponds to a SRT entry (line) that is composed by a set of fields presented in table 4.1 (see also figures 4.2 and 4.3).

Message parameter	Task parameter	SRT field (units)
Worst-case transmission time (C_j)	NA	Bit size (number of bits)
Period (T_j)	Period (T_i)	Period (number of ECs)
First release instant (ph_j)	First release instant (ph_i)	Init (number of ECs)
Deadline measured relatively to the release instant (D_j)	Deadline measured relatively to the release instant (D_i)	Deadline (number of ECs)
Absolute deadline ($d_{j,k}$)	NA	Absolute deadline (number of ECs)
Release instant ($r_{j,k}$)	NA	Init (number of ECs)

Table 4.1: Message and task parameters mapping
NA: Not Available

Note that, in table 4.1 the bit size is not consider for the tasks because the tasks do not occupied time at the bus and, as referred, finish their execution before the beginning of the next instance of other task running in the respective slave. For the point of view of the

master node, the release instant is constant, thus the absolute deadline is also constant (and derived from the release instant and from the relative deadline).

In the SRT the fields are stored according the FTT-CAN rules, *i.e.*, they are multiples of the elementary cycle (except the number of bits). Notice that the memory footprint is an important limitation on embedded systems, that has been carried out in this SRT implementation. There are some additional fields that are needed to support and speed up the master operation, these are:

- Size, in bytes, of the message. If the line of the SRT belongs to a task, this value is zero because a task does not occupy any bandwidth. The size of the message occupies one byte in each database entry. However, it can just be greater or equal to zero and less than nine;

- Size in bits. This is the total size of the message in bits. If the variable is a task this value is zero, otherwise it is different from zero. The bit size value is calculated according the rules of the CAN protocol message and it is equal to the sum of all CAN message fields with the worst case bit stuffing that a message can have (according to equation 3.2). This value is calculated when the message is added to the system;

- Byte index. The byte index field corresponds to the byte in which the data field of the trigger message flags the variable (refer to figure 3.3). This value is calculated when a variable is added to the system according to its identifier. The identifier of the message imposes a specific byte on the trigger flags of the trigger message. All the variables with an identifier within zero and seven have byte index zero, all the variables with identifier within eight and fifteen have byte index one, and so on. This means that this value can be calculated by the integer of the division between the identifier value and 8 ("ID mod 8");

- Bit mask. The bit of the trigger flags that marks the position of the flag inside the byte index. This value is calculated when a variable is added to the system and depends on the identifier of the variable. For example the variable with identifier zero has byte index zero and bit mask 0x01 and, for example, the variable with identifier eleven has byte index one and bit mask 0x08, *i.e.*, the bit mask is the necessary "AND mask" to get the trigger flag for the variable (see figure 3.3)[5];

- Begin mask. This field represents the initial instant the variable must be mask. Masking a message means that the message is scheduled for a specific EC but the system will not dispatch it in order to perform any housekeeping of the communications. The begin field of the SRT line is a dynamic field, *i.e.*, it may have different values in different

[5]This may look a bit complex but it is important to reduce the overhead in processing the trigger message at the nodes.

elementary cycles. Beginning means the startup of the system or the time after one variable has been added upon a successful request from a slave. When masked, the variable is still scheduled. The variable is unmasked when the number of elementary cycles indicated by this field has elapsed. This value is an add-on to provide more flexibility to the system designer;

- Physically Connected Bus (PCB in figure 4.2). This byte indicates the buses where the message can be transmitted. Recall that the slaves can connect to a bus, a set of buses or to all buses. This field is not necessary in case of a single bus system. This has been explained in section 3.5.5;

- Initial Bus Allocation (IBA in figure 4.2). This byte indicates the buses where the message is initially transmitted. This is useful in case of reverting to the initial settings. This field is not necessary in case of a single bus system. This has been explained in section 3.5.5;

- Current Bus Allocation (CBA in figure 4.2). This byte indicates which are the current buses where the message must be transmitted. This field is not necessary in case of a single bus system. This has been explained in section 3.5.5.

The set of fields from table 4.1 and these additional ones compose a SRT line, corresponding to a variable: message or task. The set of lines compose the SRT (the SRT can be seen as a table, see figure 4.3). Figure 4.2 depicts a SRT line.

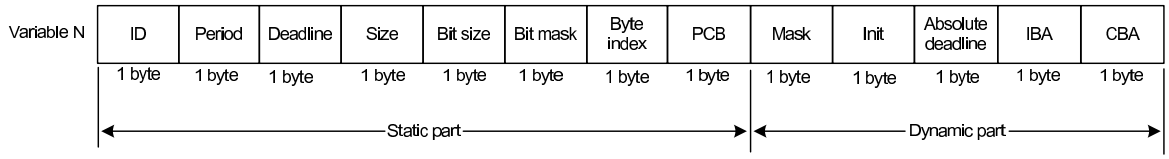


Figure 4.2: SRT line for multiple buses

The SRT is divided into two sections: the message section and the task section (see figure 4.3). The messages and the tasks are stored in the same table to speedup the scheduler algorithm, since it just needs to consult one table instead of two to get all the information regarding all the variables (messages and tasks).

Each message or task has the static part and the dynamic part stored in the table (as presented in figures 4.2 and 4.3). The static part corresponds to the information that is constant in time, like the period and the identifier, and the dynamic part is the information the master changes while it is scheduling the variables. Examples of dynamic information are the number of elementary cycles to the next activation and the mask field.

The table is sorted in an ascending order regarding to a specific field of the variable. Again this is done to speedup the scheduler algorithm, because it is more efficient to search

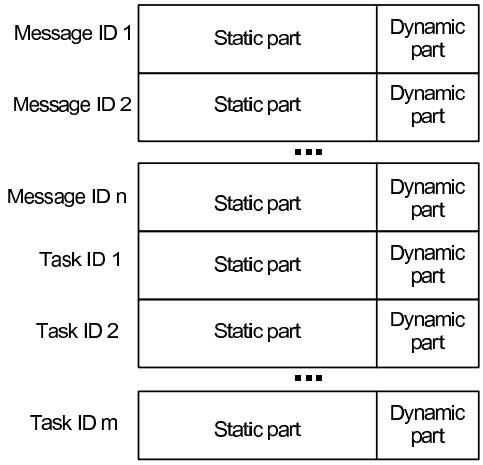


Figure 4.3: Synchronous requirement table architecture

an ordered table.

4.2.2 The trigger message

The size of the synchronous window (expressed in terms of the total number of bits of the respective synchronous messages), is transmitted to the slaves using the trigger message. Thus, the size of the asynchronous window is also transmitted in a indirect way, because LEC is a global parameter. However, to preserve the number of bits of the trigger message, this value is encoded in another value, the FTT Time Quantum (FTTTQ). The length of the elementary cycle is divided into time ticks, each representing a FTTTQ. This value is calculated in a way to have sufficient resolution and, at the same time, to be just 1 byte long (The minimum FTTTQ becomes then the maximum length of the synchronous window divided by 250).

Table 4.2 presents the way how trigger message parameters are mapped into the fields of the trigger message.

Parameter	TM field	CAN message field (figure 4.4)
Size in bits (S_l)	FTTTQ	data byte 0
Tasks trigger flags ($\beta^l_{q,o}$)	Trigger flags	data byte 2 to 7
Messages trigger flags ($\xi^l_{q,p}$)	Trigger flags	data byte 2 to 7

Table 4.2: Trigger message parameters mapping

In addition to the CAN fields presented in table 4.2, the trigger message has some additional fields to exchange other information among slaves and masters or for functionality reasons. These additional fields are:

- Signature. The most significant 4 bits of the CAN message identifier are dedicated to identify the CAN message as a trigger message. In this case, this 4 bits are ”0001”;

- Master identification. The next four bits of the CAN identifier of the trigger message are dedicated to the number of the master that is currently transmitting the trigger message. Each master has a unique identifier in the system that can be used by the slaves to detect what is the actual active master. Identified as 'Master ID' in figure 4.4;
- Sequence number. The three less significant bits of the CAN identifier of the CAN message encapsulates a sequence number so that the slaves can verify if some trigger message was missed or received out of order. Identified as 'Sequence number' in figure 4.4;
- Answers to slave requests (denoted as 'Request answer' in figure 4.4). The second byte of the data field of the trigger message is the answer from the active master to the online requests made by a slave to change the SRT.

In figure 4.4 a general structure of the trigger message is shown.

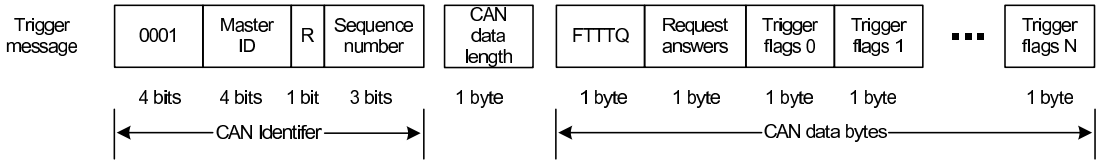


Figure 4.4: Trigger message structure

The FTTTQ and the request answers are encoded into the CAN data field (as depicted in figure 4.4). Thus, from equation 3.14, the length of the CAN message data field of the trigger message is:

$$DLC_{TM} = N_{reserved} + \frac{N_{sw}}{8} \tag{4.7}$$

Because N_{sw} is the number of system wide flags (for messages and tasks), DLC_{TM} is an integer number and $N_{reserved} = 2$, the equation can be re-written as:

$$DLC_{TM} = 2 + \left\lceil \frac{n+m}{8} \right\rceil \tag{4.8}$$

Where:

- n is the number of triggers flags for messages (as in equation 4.3), *i.e.*, the maximum number of messages the master can trigger;
- m is the number of the trigger flags dedicated to trigger tasks in the slaves (as in equation 4.5), *i.e.*, the maximum number of tasks the master can trigger.

The maximum number of CAN data bytes is 8, that lead to a maximum number of variables of 48. This is a reasonable value for the target applications of the FTT-CAN protocol.

4.2.3 Application programming interface

The application programming interface allows an application using the FTT-CAN protocol to interact with the underlying network. In the master node, the available functions are:

- `MstTab_AddVar`. This function adds variables (messages or tasks) to the synchronous requirement table. In fact this function adds a line to the SRT, using parameters and fields passed to it. It is called one time for each variable (message or task);
- `MstTab_DelVar`. This function deletes variables (messages or tasks) from the synchronous requirement table;
- `MstTab_ChangeVar`. This function changes the variables (messages or tasks) parameters on the synchronous requirement table;
- `FTT_MstInit`. This function is called at the startup of the application to initialize all the system. These initializations include the global parameters and the initializations of the microcontroller peripherals;
- `FTT_MstStart`. This function is called to startup the master, initializing the interrupts, turning on the timers and starting the scheduling for the first elementary cycle.

The available functions in the slave nodes are:

- `AM_SlvTab_AddVar`. This function adds an asynchronous message to the asynchronous table;
- `SlvTab_AddVar`. This function is called to add a variable (message or task) to the synchronous table[6] in the slaves. The message parameters (for example length and if the slave is producer or consumer) are passed to the function as input parameters;
- `produce`. This function is called when the application needs to produce a synchronous message. The FTT slave middleware verifies if the current slave is a producer of the message and updates the data accordingly. Note the synchronous message is sent to the CAN bus when the master schedules it;
- `consume`. This function is called when the application needs to consume a synchronous variable (task or message). In case of a message the FTT verifies if the slave is a consumer of the message and returns the synchronous message data. In case of a task, the FTT system, after the verifications, triggers the task;
- `VarStatus`. This function verifies the status of a variable and returns it;

[6]This synchronous table is similar to the one existing in the masters, however, having just the necessary parameters for the slaves to work.

- `AM_send`. This function sends an asynchronous message to the CAN bus. In fact, the message is not immediately sent, it is placed into a buffer and will be transmitted in the asynchronous part of the elementary cycle;

- `AM_receive`. This function receives an asynchronous message. In fact it reads a message from a buffer where the FTT system puts all the asynchronous messages received;

- `AddVariable`. This function adds a variable (message or task) to the synchronous requirement table of the master. It sends an asynchronous message to the master with parameters (passed as input parameters to the function) of the synchronous variable to be added. It will wait for the answer of the master to check if the synchronous variable has been accepted and added correctly. The master answers to the slave in the second byte of the data field of the trigger message (as explained before);

- `ChangeVariable`. This function provides a way to change parameters or fields of a variable on the synchronous requirements table. The slave sends an asynchronous message with the parameters (passed as input parameters to the function) to the master, which processes the change request and anwsers in second byte of the data field of the trigger message. Only dynamic fields of a variable can be changed;

- `DelVariable`. With this function a slave can request the master to delete a variable from the synchronous requirement table and waits for the answer which will come piggybacked in the trigger message (as for the `AddVariable` and `ChangeVariable`);

- `FTT_SlvIni`. This function is called at the beginning of the application to initialize the slave FTT-CAN system, including the timer generations values, the hardware peripherals and the interrupt system;

- `FTT_SlvStart`. This function starts the slave FTT system.

4.3 The master implementation

As it was discussed at the beginning of this chapter, previous implementations of FTT-CAN were targeted to old microprocessors with low processing power and small memory. For this reason, features such as the tasks triggering in the slaves and master replication were not implemented then. Notice that early implementations of FTT-CAN were made prior to the definition of remote task triggering and master replication.

The new implementation of the FTT-CAN protocol started with the single bus, single master FTT-CAN. Afterwards, the master node was adapted to handle master replication and, finally, a master node version to handle multiple buses has been implemented. Both implementations (single and multiple buses) use the same microcontroller brand, however

from different families with different internal architecture. The single bus master version has been implemented on a Microchip PIC18F258 [Mic06], while the master node for multiple buses has been implemented in a Microchip dsPIC30F6012A [Mic08].

The master firmware takes care of the main functionalities of the FTT-CAN protocol. These functionalities are:

- Scheduling, based on the communications requirements;
- Coordination of the network, includes the management of the bus replicas and the transmission of the trigger message using the scheduling data;
- Communication requirements management;
- Master replication.

Figure 4.5 depicts the FTT-CAN master board developed for the single bus version while figure 4.6 shows the developed multiple buses FTT-CAN master node version.

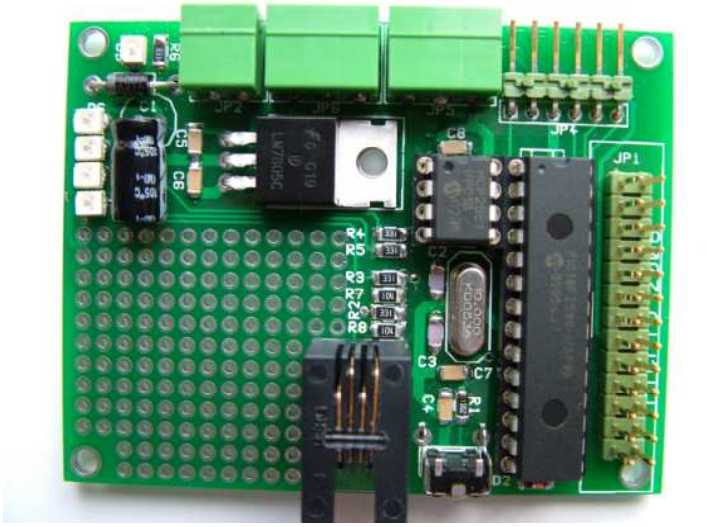


Figure 4.5: Development board for PIC18F258

4.3.1 Single bus master

Figure 4.7 depicts the general overview of the single bus master firmware. The operation of the master is carried out by the scheduler module which is responsible for defining which messages should be transmitted in the next elementary cycle. The output of the scheduler is sent to the dispatcher module which prepares and controls the transmission of the trigger message, accessing the hardware drivers to send it. The systolic nature of the trigger message and other timeliness functions are controlled by the timer handler module which relies on a time mark generated at hardware level.

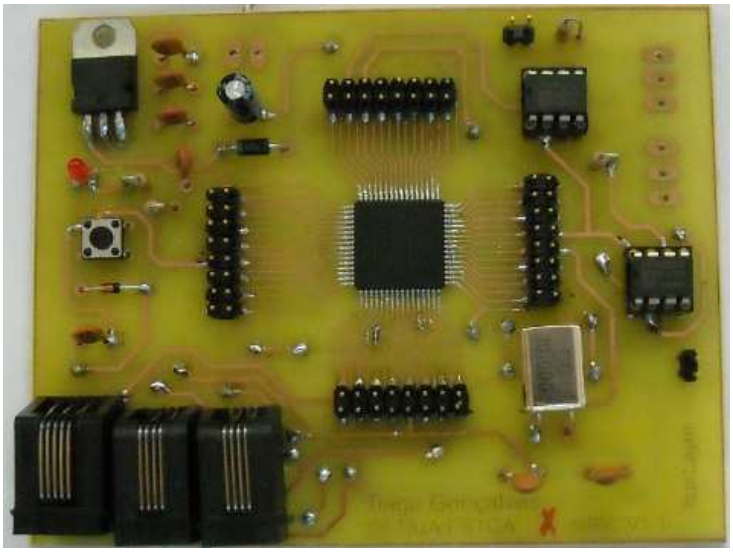


Figure 4.6: Development board for dsPIC30F6012A

Each event of the timer handler is signalled with a hardware interrupt. In figure 4.8 the timing events occurrences during one elementary cycle are presented. These events can be generated by a timer event (by the timer handler) or generated asynchronously (such as generated by the completion of a function). Notice that the timing events are different depending if the master is active or backup (as depicted in figure 4.8). This events are valid for the single bus master and for the multiple buses master.

In the active master there are four events triggered by the timer handler:

- Timing event 0. This event occurs when the timer handler triggers the dispatching of the trigger message. A trigger message is scheduled in the previous elementary cycle, or in case of being the first trigger message, it is scheduled at startup time;

- Timing event 1. This event triggers the scheduling for the next elementary cycle. The trigger of the scheduler is done after the completion of the dispatcher, *i.e.*, the scheduler is called after the dispatcher finishes transmitting the previous EC trigger message. Thus, this event is an asynchronous event;

- Timing event 2. This event marks the end of the trigger message transmission window ($TMTW$). In the active master is the time available for the retransmission of the trigger message;

- Timing event 3. This event corresponds to the beginning of the synchronous window of the elementary cycle. The dispatcher handler informs the timer handler about the size of the synchronous window. Thus, the timer handler can then compute the length of the asynchronous window knowing the length of the elementary cycle. Recall that the master node needs to know the asynchronous window duration to be able to receive

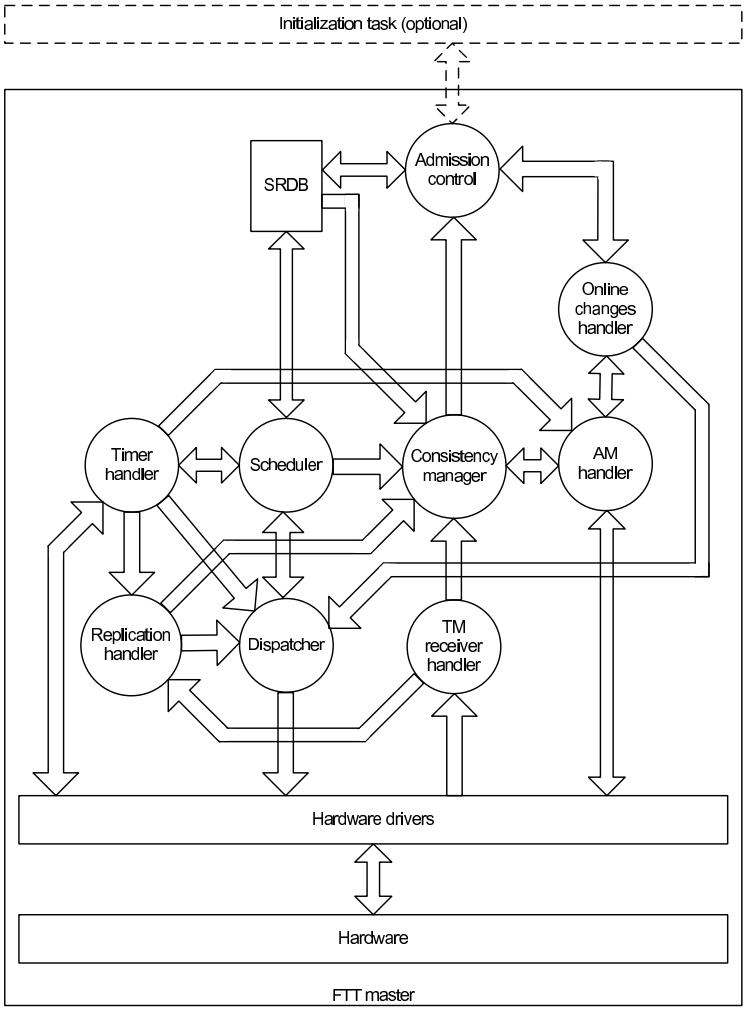


Figure 4.7: Master firmware overview

online changes request send by the slaves and also to perform the synchronization algorithm among all the masters presented in [FAF+03].

The timer handler of the backup master generates timer events in other moments of the elementary cycle. They are:

- Timing event 0. This event is a timer mark that occurs at the middle of the trigger message and is intended to trigger the dispatching of the trigger message on the backup master. In the case of figure 4.8 the trigger message X was dispatched by the active master. The backup master tries to send its own trigger message at this timer event and tries to abort the TM transmission at next processor instruction cycle. If the active master has a problem (it will not send the trigger message due to its fail-silent behaviour), the backup master will succeed transmitting its own trigger message and become active. On the other hand, if the backup master aborts its own trigger message, means the active master still healthy. This event is only valid for the single bus master;

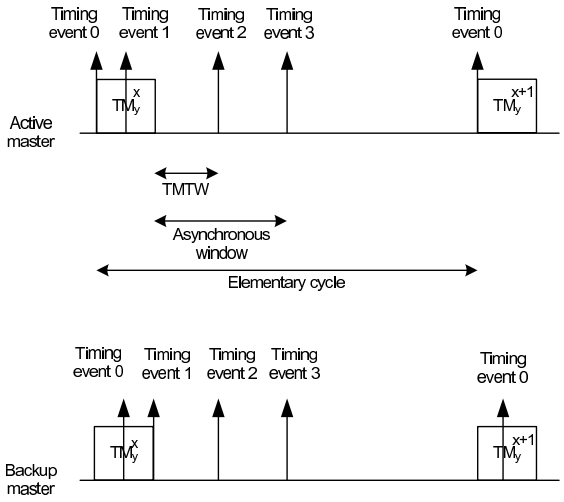


Figure 4.8: Timer occurrences in masters

- Timing event 1. In this event, the scheduler for the next trigger message starts working. In figure 4.8 it is marked at the end of the trigger message, but, in fact the scheduling for the next elementary cycle begin as soon as the dispatcher finish her job. Thus, this event is an asynchronous event;

- Timing event 2. This event marks the end of the trigger message transmission window. In the backup master signals the time it must wait for the trigger message. In addition, in the multiple buses system marks the occurrence of an error (in active master or in the bus);

- Timing event 3. This timer handler signals the end of the asynchronous window. This mark is synchronized with the active master. As it happens in the active master, the dispatcher informs the timer handler about the duration of the synchronous window. The timer handler then computes the asynchronous window size knowing the CAN bitrate and the elementary cycle size. The backup masters need to know the length of the asynchronous window to receive the requests from the slaves for changing the SRT and, as in the active master, to perform the synchronization algorithm.

In addition to the scheduler, dispatcher and timer handler, the other modules of the FTT-CAN master node depicted in figure 4.7, are:

- The asynchronous messages handler (denoted as 'AM handler' in figure 4.7) receives the online requests sent in the asynchronous window. Notice that this handler is also used by other tasks concerning the master replication and synchronization and it needs the information of the elementary cycle temporization to define the synchronous and asynchronous window boundaries;

- The 'online changes handler' processes the online requests made by the slaves, which were received by the 'AM handler', and sends them to the admission control for processing. The answer of the request is then sent to the dispatcher and is piggybacked in the trigger message;

- The 'admission control' is responsible for analyzing and changing the SRT. Thus, it checks if a request from the online changes handler is admissible;

- The 'TM receiver handler' is responsible for the trigger message reception. This handler is only active if the master is a backup master, because in this case it must receive the trigger message transmitted by the active master to compare it with its own, so it can detect possible inconsistencies;

- The 'replication handler' manages all the issues related with the master replication. Depending on the reception of the correct trigger message in the correct time, this handler could detect if a backup master must become active master or must remain in the same state;

- The 'consistency manager' is responsible for the management of the consistency protocol. It can declare the backup master[7] inconsistent whenever the local trigger message does not match with the received trigger message issued by the active master. If they differ, the backup master sends an asynchronous message to the active master so it can start to upload the correct SRT.

Finally, it should be referred that the access to the hardware is made through the hardware drivers. In figure 4.7 these hardware drivers include the CAN controllers access for configuration, reception and transmission of the messages.

4.3.2 Multiple buses master

The firmware overview of the master with multiple buses is depicted in figure 4.9. The master node firmware requires some adaptations to cope with multiple buses [SFF07c]. Two new handlers (shadowed in figure 4.9) were added to the single bus FTT-CAN master node:

- The 'bus error detector' handler is responsible for the detection of faulty buses. After the detection of a faulty bus, this handler informs the multiple buses handler about the error;

- The 'multiple buses handler' manages the bus replication. It receives the information about the faulty buses from the bus error detector and changes the SRT accordingly. This handler also processes requests from the slaves to change the bus where a specific

[7] The active master is always considered consistent by the fail silence assumption.

message is transmitted. Notice that the reception of these requests is made by the online changes handler. Also notice, in the current implementation, it is not possible to the change bus configuration online upon a slave request.

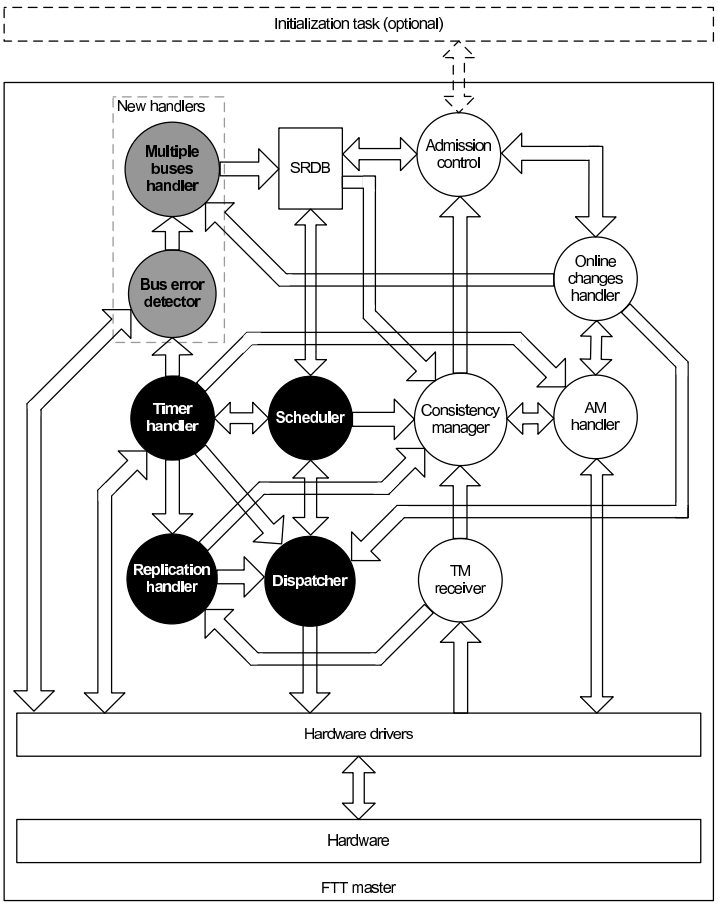


Figure 4.9: Master firmware with multiple buses

Four handlers have to be modified (reverse coloured in figure 4.9) to deal with the multiple buses system:

- 'Scheduler'. The scheduler was adapted to do the scheduling for all the buses;
- 'Dispatcher'. The dispatcher was adapted so it can transmit trigger messages in all the available buses;
- 'Replication handler'. The master replication protocol is different for the multiple buses version. In the single bus version, the master replication is based on a transmit and abort technique, while in the multiple buses version, the master replication protocol is based on trigger messages reception (as explained in chapter 3);
- 'Timer handler'. For the case of a backup master, the timer event 0 is not generated (see figure 4.8), but the generation of a timer event marking the end of the trigger message

transmission window is kept (see figure 3.11 and 3.12). This event is also used for the detection of a faulty bus.

4.4 The slave implementation

This section describes the implementation of the slave node for the single bus version. The multiple buses slave node was not implemented because this thesis is focused on the FTT-CAN master node with multiple buses and, for a proof of concept implementation, a single bus slave node is enough. In that way, figure 4.10 depicts the single bus slave implementation.

Typically, the slave nodes perform data acquisition, data treatment and send actuation information to the actuators in case of a control system. The slaves exchange data among them, either synchronously or asynchronously, to execute a distributed algorithm. Synchronous messages are scheduled by the master and the slaves are informed of their dispatching instants via the trigger message. The asynchronous messages processing is a responsibility of the slave nodes. The synchronous message handler ('SM handler' in figure 4.10) takes care of the time-triggered messages while the asynchronous message handler ('AM handler' in 4.10) handles the sporadic event-triggered messages.

The online changes handler (figure 4.10) manages slave requests to the master to change synchronous messages flows.

The slave node can run in a microprocessor with no other applications or it can run in a microprocessor or computer with multiprocessing. An application running in a slave node can have several tasks that interact with the slave middleware via the API. The initialization tasks initialize all the messages (synchronous and asynchronous) and the tasks to be triggered in the slave. Note that the tasks can interact with other tasks, depending on the particular operation being executed.

The hardware access is made through the hardware specific device drivers. For example, the hardware timers are managed by the timer handler (see figure 4.10) that controls the synchronous and asynchronous window and the tasks execution.

4.5 Evaluation of the computational overhead

The aim of computing the master operational overhead is to assess the possibility of executing all master operations in less than an elementary cycle, because otherwise it would not be possible to implement the protocol. The FTT-CAN master middleware was implemented in a Microchip PIC18F258 [Mic06] microcontroller and it takes advantage of its interrupt system. Since the scheduler (the most time consuming task of the FTT-CAN master) and the dispatcher are preemptable, to compute the operational overhead, one has to know which are

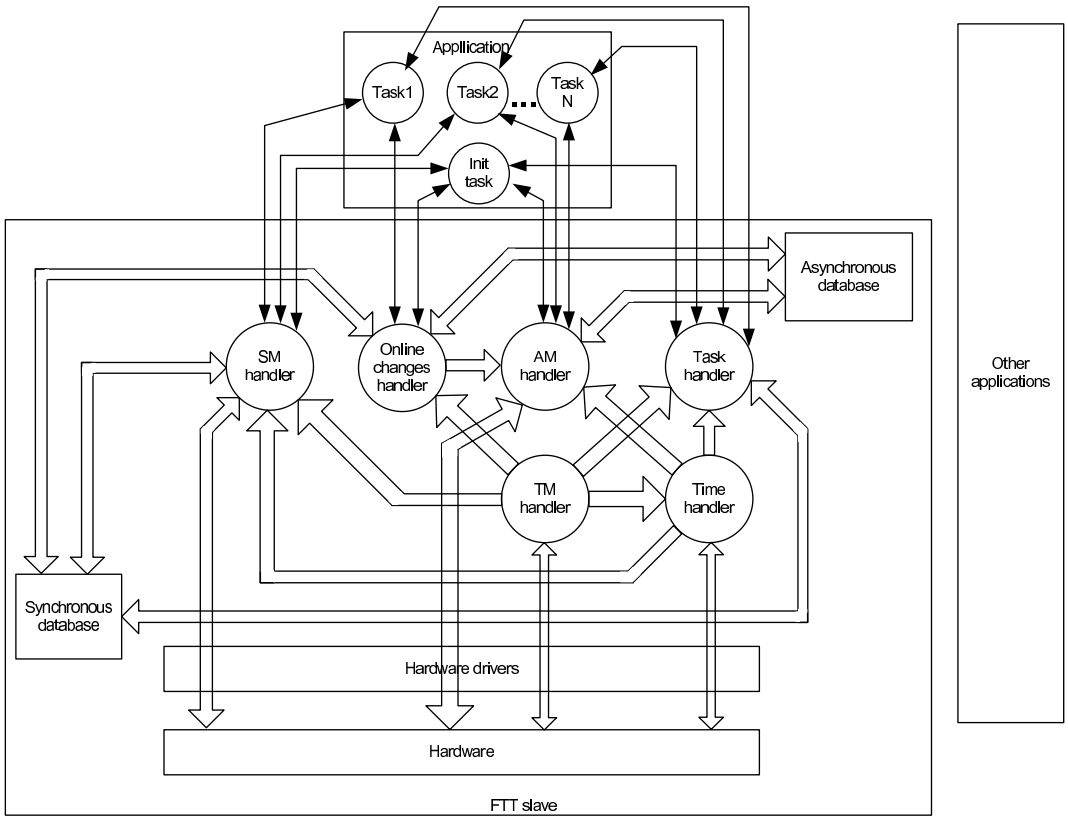


Figure 4.10: Slave firmware overview

the interrupt service routines (ISRs) that can occur while the scheduler and the dispatcher are running. These ISRs are the following:

- Trigger message reception ISR. If the master is a backup master, it must receive the trigger message for comparison with the one internally generated for inconsistency verification;
- Asynchronous message ISR:
  - Asynchronous data messages among slaves. These messages do not cause any ISR, since they are filtered in the CAN controller hardware of the master node;
  - Online change requests to the SRT. Each of these messages generates a reception ISR (at active and backup masters). The minimum size of these messages is 2 data bytes [Fer05];
  - Request from a backup master for sending the SRT for re-synchronization purposes. This messages cause a reception ISR at the active master and a transmission ISR at the backup master. The minimum size of these messages is 0 data bytes [Fer05];

– SRT transmission for synchronization purposes. The SRT is split and transmitted in multiple asynchronous messages. These messages cause transmission ISRs at the active master and reception ISRs in the backup master which issued the request. These messages can have three different sizes. Eight data bytes to transfer the static part of the SRT line, messages with three data bytes to transfer the dynamic part of the SRT line and delimiters with one data byte to distinguish between the static and dynamic part [Fer05].

To assess the computational overhead it is necessary to compute the time taken to process asynchronous messages that are interrupt sources, plus the time taken to run the scheduler, the dispatcher and TM reception ISR (at a backup master). Note that the worst case only noticeable difference in the computation overhead from the active master to the backup master is the trigger message reception at the backup master. The trigger message reception is an exclusive overhead of the backup master and does not occur in the active master. All the other overheads are similar in the active and backup masters.

This computational overhead assessment is hard to perform, since the length of the messages varies and the execution time of the ISRs depends on each type of message. In this way, the computational overhead of the master node using just one bus is (in figure 4.11 there is the graphical representation of this):

$$C_{Mst} = C_{Sch} + C_{Disp} + C_{ISR} \tag{4.9}$$

$$C_{ISR} = \sum_{i=1}^{N_{ISR_R}} C_{ISR_R(i)} + \sum_{i=1}^{N_{ISR_T}} C_{ISR_T(i)} \tag{4.10}$$

$$C_{Mst} = C_{Sch} + C_{Disp} + \sum_{i=1}^{N_{ISR_R}} C_{ISR_R(i)} + \sum_{i=1}^{N_{ISR_T}} C_{ISR_T(i)} \tag{4.11}$$

Where:

- C_{Mst} is the computational overhead of the master node;
- C_{Sch} is the computational overhead of the scheduler;
- C_{Disp} is the computational overhead of the dispatcher;
- C_{ISR} is the computational overhead of all interrupt service routines;
- C_{ISR_R} is the computational overhead of the reception interrupt service routine. It includes the trigger message reception, in case of a backup master (denoted as 'TMR' in figure 4.11);
- N_{ISR_R} is the number of reception ISRs;

- C_{ISR_T} is the computational overhead of transmission interrupt service routine and N_{ISR_T} is the number of transmission ISRs.

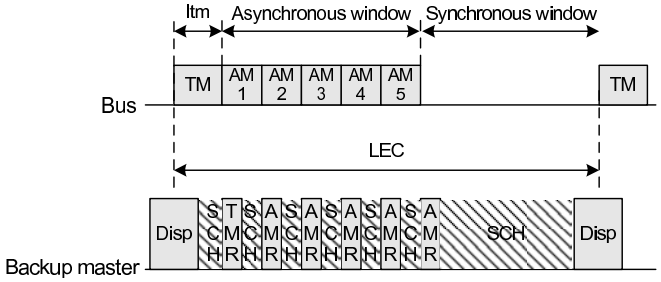


Figure 4.11: Backup master computational overhead

In figure 4.11, the time available for the dispatcher is represented as 'Disp' while the time available for the scheduler is represented as 'SCH'. In this figure, the worst case is represented, because the asynchronous window is filled with asynchronous messages received by the backup master. There is no transmission ISR represented, but the system will have a similar behaviour and the computational overhead assessment is still valid, since in equation 4.12 the maximum of an ISR overhead is used (reception or transmission).

A possible simplification of this procedure is to consider the maximum number of asynchronous messages that fit in a maximum sized asynchronous window and that each message causes an interrupt that takes the highest processing time among all of them. In this way, the computational overhead of the interrupt service routines can be approximated to:

$$C_{ISR} = \max(N_{am}) \times \max(C_{ISR_{R(i)}}, C_{ISR_{T(j)}}), \forall i = 1, ..., N_{ISR_R}, \forall j = 1, ..., N_{ISR_T} \quad (4.12)$$

Where $\max(C_{ISR_{R(i)}}, C_{ISR_{T(j)}})$ is the maximum computational overhead of an interrupt service routine, and $\max(N_{am})$ is the maximum number of asynchronous messages on the bus. Thus:

$$\max(N_{am}) = \left\lfloor \frac{law_{max}}{\min(t_{fd})} \right\rfloor \quad (4.13)$$

Where:

- law_{max} is the maximum length of the asynchronous window;
- $\min(t_{fd})$ is the minimum frame duration of a CAN message. This is a frame with zero data bytes an no stuff bits. In fact this is a very particular frame, with very low usability, but for a pessimistic analysis it can be used.

The maximum length of the asynchronous window occurs considering the synchronous window do not exists. Thus, law_{max} is:

$$law_{max} = LEC - \min(ltm^k), \forall k = 0, ..., \infty \tag{4.14}$$

Where:

- LEC is the length of the elementary cycle;
- k is the number of the elementary cycle;

Thus, the maximum computational overhead is:

$$C_{ISR} = \left\lfloor \frac{LEC - \min(ltm^k)}{min(t_{fd})} \right\rfloor \times \max(C_{ISR_{R(i)}}, C_{ISR_{T(j)}}),$$
$$\forall i = 1, ..., N_{ISR_R}, \forall j = 1, ..., N_{ISR_T}, \forall k = 0, ..., \infty \tag{4.15}$$

Making a pessimistic analysis, the maximum computational overhead occurs when the trigger message has two data bytes and the asynchronous messages have no data bytes. In fact, this is an unrealistic scenario (and very pessimistic), since a trigger message with two data bytes cannot trigger any messages or tasks. It is also true that asynchronous messages with zero data bytes are useless, but this scenario will be considered for a worst case analysis.

The minimum CAN message length, without stuff bits, is 47 bits (please refer to equation 3.1 using $g = 34$ and $DLC = 0$), thus, the time taken by such message is:

$$\min(t_{fd}) = 47 \times \frac{1}{B_r} \tag{4.16}$$

where B_r is the bitrate of the CAN bus.

The minimum length for the trigger message (using 2 data bytes, $DLC = 2$, and $g = 34$ in equation 3.1), is:

$$\min(ltm^k) = 63 \times \frac{1}{B_r} \tag{4.17}$$

For the case of a multiple buses master, the computational overhead of a FTT-CAN master with multiple buses can be determined using the same approach as before, since the new modules required by the multiple buses implementation are quite simple. However, an additional assumption must be made in order to calculate the computational overhead of the master:

- All FTT-CAN protocol related messages are conveyed on the same bus, *i.e.*, update request messages and master synchronization.

Based on this assumption, the computational overhead imposed by the asynchronous interrupt service routines is the same as for a single bus. So, re-writing equation 4.9 for the case

of a multiple buses system, the computational overhead for the active master and for the backup master is:

$$C_{Mst\_N\_Bus} = N_{buses} \times (C_{sch} + C_{disp}) + C_{ISR} \tag{4.18}$$

Where N_{buses} is the number of existing buses in the system.

Equation 4.18 considers the execution of the scheduler and dispatcher module in a round robin scheme.

4.5.1 Experimental results

A single bus system with six slaves, one active and a backup master was implemented in an autonomous robot [MSF+06, MSF+07]. The elementary cycle was set to $5ms$ which is the minimum period of tasks and messages in the system (see section 5.2.2 and table 5.1). The set of messages added to the system is composed of 9 messages with periods from 4 to 200 elementary cycles (thus, $20ms$ to $1s$). This set of synchronous messages results in a communication load close to 27% (using the maximum number of stuff bits) or 22,5% (using the minimum number of stuff bits), with FTT-CAN running at $250kbps$ (these loads have been calculated using table 5.1). The maximum duration of the asynchronous window is the duration of the EC minus the duration of the trigger message. Concerning the hardware, it was used the PIC18F258 running at 40Mhz which corresponds to an instruction cycle of $100ns$ (there is an internal divisor of the external crystal oscillator).

In these conditions, the worst execution times measured in a long run were:

- Scheduler module, $C_{sch} = 628\mu s$;
- Dispatcher module, $C_{Disp} = 82\mu s$;
- The maximum execution time observed for the interrupt service routine, $\max(C_{ISR_{R(i)}}, C_{ISR_{T(j)}}) = 105\mu s$.

The minimum size of the trigger message in the presented test platform is 6 data bytes. Thus, considering no stuff bits in the trigger message (see equation 3.1):

$$\min(ltm^k) = (47 + 6 \times 8) \times \frac{1}{250 \times 10^3} = 380\mu s \tag{4.19}$$

$$\min(t_{fd}) = 47 \times \frac{1}{250 \times 10^3} = 188\mu s \tag{4.20}$$

$$law_{max} = 5m - 380\mu = 4.62ms \tag{4.21}$$

$$\max(N_{am}) = \left\lfloor \frac{4.62m}{188\mu} \right\rfloor = 24 messages \tag{4.22}$$

$$C_{Mst} = 628\mu + 82\mu + 24 \times 105\mu = 3.23ms \tag{4.23}$$

Considering the elementary cycle duration of $5ms$, the results show that the master running at this microcontroller is able to do all the necessary tasks in less than an EC time. Note that the analysis performed is pessimist because it has been assumed that the CAN messages (synchronous or asynchronous messages) always have no stuff bits. In this case, the maximum number of messages occurs, thus the maximum number of ISRs.

For a master with three CAN buses and using the same processor, the overhead can be extrapolated to:

$$C_{Mst\_N\_Bus} = 3 \times (628\mu + 82\mu) + 24 \times 105\mu = 4.65ms \tag{4.24}$$

Thus, using three buses this microcontroller can still perform all the necessary tasks within one elementary cycle.

4.6 Conclusions

This chapter presented the implementation of FTT-CAN in small embedded systems, describing the data structures and the application program interface. The asynchronous messaging system was not considered because it does not impact the master node. Asynchronous messages are handled by the slave nodes only by themselves.

The FTT-CAN protocol was developed to support applications that require flexibility and real-time and synchronization capabilities. An application programming interface was implemented to support those requirements and a detailed description of each API function was presented in this chapter. The data structures to support the execution of the middleware in the master have been presented. Special focus has been given to the synchronous requirement database and to the trigger message structure. The mapping of these structures into the developed software has been addressed because it is an important development issue.

The software modules (handlers) of the master node with one bus and with multiple buses were also presented. In fact, the multiple buses version is an adaptation of the single bus version, where some of the internal modules were adapted and a few new modules were added.

Experimental results have shown that the computational overhead of running the FTT-CAN protocol on low computational power microcontrollers is small, enabling the implementation of the protocol with single or multiple buses in one microcontroller.

Chapter 5

Experimental evaluation

5.1 Introduction

The FTT-CAN implementation described in the previous chapter is assessed and validated in this chapter. Two different evaluation approaches were adopted, one for the single bus case and other for the multiple buses implementation.

The single bus implementation has been extensively tested and it is running on soccer robots since 2005 [SMA+05]. Its most critical issue, the master replication protocol, was previously validated using the model checking formal verification tool SPIN (Simple PROMELA Interpreter) [RNPR+04]. Formal validation results have shown that the master replication protocol works correctly for the considered fault hypothesis. However, it was not demonstrated then that the system could be implemented on low processing power microcontrollers. Thus, the main purpose of designing from scratch the single bus FTT-CAN implementation, described in the previous section, was to validate the conjecture that microcontrollers with low processing power and memory may, in fact, implement the single bus FTT-CAN with master replication with little overhead.

The introduction of the multiple buses to FTT-CAN system brings new potential problems that require preliminary extensive laboratory validation prior to deploying them in real world applications (such as the soccer robots). The approach adopted to validate the multiple buses FTT-CAN architecture was based on fault injection. With the custom fault injector hardware, one can generate very specific fault scenarios that contribute to access the dependability of the system while proving that it can be deployed, with small overhead, in low memory and low processing power microcontrollers.

5.2 Single bus mechanisms' assessment

The single bus FTT-CAN architecture was evaluated using a soccer robot platform that was developed at our research lab. The soccer robot operates in a highly dynamic environment

where it interacts with other robots from the same team, with external devices and with robots of the opposite team.

The robot platform is being developed for several years and the initial distributed control system was based on native CAN bus connecting all nodes without any upper level synchronization of the messages and the tasks running in different nodes.

The deployment of FTT-CAN as the backbone communication system of the soccer robots favours an explicit synchronization among all the nodes, a tight control of the traffic flows transmission, thus reducing message's jitter. Adding to this, it enables also the simultaneous triggering of tasks in different nodes.

This section addresses the deployment and assessment of single bus FTT-CAN and compares the experimental results, obtained on the soccer robots, with the initial implementation, in order to highlight the benefits of using this protocol.

5.2.1 Experimental platform: the CAMBADA soccer robot

The robotic platform CAMBADA [ASF+04] was developed to participate in the RoboCup soccer competition [KAK+97]. Currently, the requirements posed on such teams of autonomous robots have evolved in two directions. On one hand, robots must move faster and with accurate trajectories to close the gap with the dynamics of the processes that they interact with, *e.g.*, a ball can move very fast. On the other hand, robots must interact with each other in order to develop coordinated actions more efficiently, *e.g.*, only the robot closer to the ball should try to get it while others should move to appropriate positions. The former requirement calls for tight closed-loop motion control while the latter demands for an appropriate communication system that allows building a global information base to support cooperation. Both cases are subject to time constraints that must be met for adequate performance.

The robot architecture for such application is based on a biomorphic (see figure 5.1) paradigm [SMA+05] being centred in the main processing unit, the brain, which is responsible for the higher level coordination behaviour. This main processing unit handles external communication with other robots and has high bandwidth sensors, typically vision, directly attached to it. This unit receives low bandwidth sensing information and sends actuation commands to control the robot behaviour by means of a distributed low-level sensing/actuation system, the nervous system (see figure 5.1).

The low-level sensing/actuating system follows the fine-grain distributed model [Kop97] where most of the elementary functions, *e.g.* basic reactive behaviours and closed-loop control of complex actuators, are encapsulated in small microcontroller-based nodes interconnected by a network. The nodes are based on the PIC18Fx58 microcontroller [Mic06] operating at 40MHz. At this level there are 3 DC motors with their respective controllers plus an extra controller that, altogether, provide holonomic motion. Each motor has an incremental encoder that is used to obtain speed and displacement information. Another node is responsible

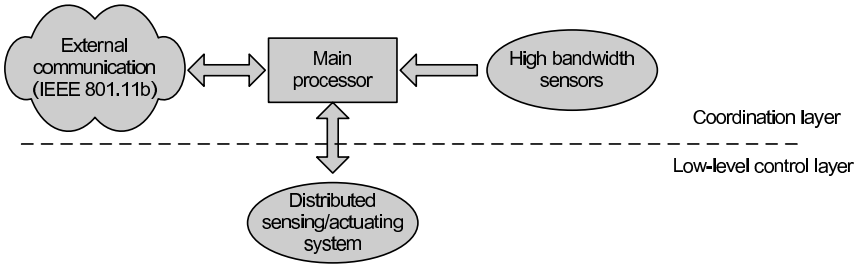


Figure 5.1: The biomorphic architecture of the CAMBADA robots

for combining the encoder readings from the 3 motors and building a coherent displacement information that is then sent to the coordination layer.

Also, there is a node responsible for the control of the kicking electro-mechanical subsystem. It consists of a couple of sensors to detect the ball in position to be kicked and to trigger the kicker. This node also carries out the battery voltage monitoring. In figure 5.2 the functional modules of the robot are presented.

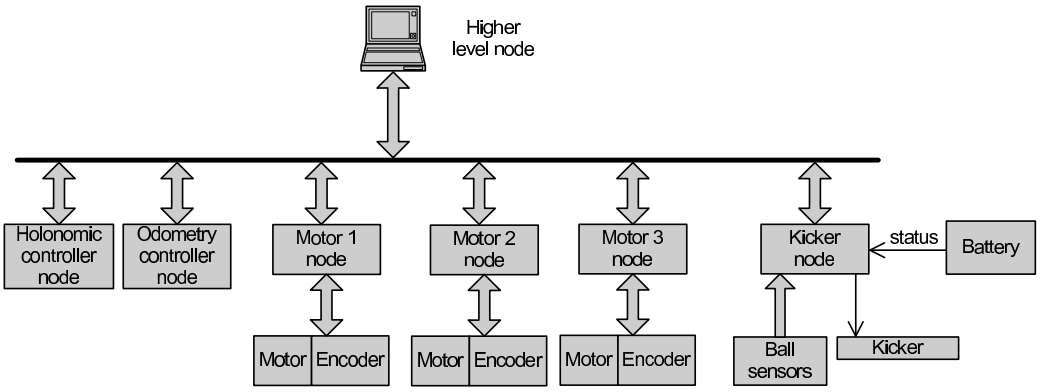


Figure 5.2: Functional robot modules

5.2.2 Communication requirements

The specific mapping of the functional architecture over the hardware platform leads to an operational architecture which presents requirements concerning both the tasks that need being executed on each node as well as the messages that must be exchanged by the nodes. The communication requirements are shown in table 5.1.

The motion function spans across 4 nodes, the 3 motor controllers plus the holonomic controller. This controller that translates the robot velocity vector set-point received from the upper layer into individual speed set-points for each of the motors. Both the motor controllers and the holonomic controller execute in a periodic fashion but with different periods. The former ones execute a PI-type closed-loop motor speed control once every $5ms$. This value has been deduced from the dynamics of the robot. Moreover, these tasks are relatively light,

ID	Source	Target	Type	Period/mit (ms)	Size (B)	Short Description
M1	Holonomic ctrl	Motor node [1:3]	Periodic	30	6	Aggregate motor speeds set points
M2	Kicker	Higher level node	Periodic	1000	2	Battery status
M3.1-M3.3	Motor node [1:3]	Odometry node	Periodic	5 to 20	3*3	Wheels encoder values
M4.1-M4.2	Odometry node	Higher level node	Periodic	50	7+4	Robot position + orientation
M5.1-M5.2	Higher level node	Odometry node	Sporadic	500	7+4	Set/reset robot position + orientation
M6.1-M6.2	Higher level node	Holonomic ctrl	Periodic	30	7+4	Velocity vector (linear+angular)
M7	Higher level node	Kicker	Sporadic	1000	1	Kicker actuation
M8-M12	Every node	Higher level node	Sporadic	1000	5*2	Node hard reset

Table 5.1: Low-level control layer communication requirements

taking less than $1ms$ to accomplish. On the other hand, the holonomic controller executes a cyclic conversion of the higher layer set-points once every $30ms$. This node is relatively loaded as each conversion takes about $16ms$ to carry out. The chosen period is, nevertheless, sufficiently small to support a smooth robot motion.

In terms of communication, the motion function requires the periodic transfer of the robot velocity vector set-point from the higher level node to the holonomic controller and then the periodic transfer of the motor speed set-points from the holonomic controller to the individual motor controllers. Both transfers are carried out once every $30ms$.

The former transfer requires two messages (M6.1 and M6.2 in table 5.1) to convey the linear and angular information, respectively. Concerning the latter transfer, the motor speed set-points generated for the motor controllers should be applied to each motor approximately at the same time thus they are piggybacked on the same message and transferred as a broadcast (M1 in table 5.1). Finally, the control loops of the 3 motor controllers should also be synchronized among themselves so that they generate motor actuation signals at approximately the same time.

Another important subsystem is the one corresponding to the odometry function. This function also spans across 4 nodes, the 3 motor controllers plus a fourth node that combines the individual encoder readings into a coherent displacement information sent up to the higher layer. The encoder readings are the same as used by the closed-loop motor speed control and thus they are sampled every $5ms$, and this should be carried out synchronously in all three motors. However, depending on the desired precision in constructing the robot displacement information, these readings can be sent with a periodicity that varies from $5ms$

to $20ms$ (higher to lower precision).

During the execution of certain high level behaviours the odometry information is not needed, *e.g.* when tracking the ball, and thus it can also be temporarily switched off.

Three messages are used to convey the encoder readings (M3.1 to M3.3 in table 5.1). Upon reception of these messages, the odometry node calculates the robot position and orientation, taking approximately $4ms$, and sends it to the higher layer, every $50ms$, using 2 messages (M4.1 and M4.2 in table 5.1). This period is compatible with the cycles used by the processes running within the higher layer. The odometry function also includes a pair of sporadic messages (M5.1 and M5.2 in table 5.1) received from the higher layer to set or reset the current robot position and orientation information within the odometry node. These messages are not expected to be generated within less than $500ms$ intervals (minimum inter-arrival time: column 'Period/mit' in table 5.1).

Finally, the kick and system monitor functions are integrated in the same node, the kicker controller, which is lightly loaded. The former corresponds to execution the kicking commands received from the higher layer. These are conveyed within one sporadic message (M7 in table 5.1) which is not expected to be transmitted more often than once every second. In fact, the kicker is electromagnetic and takes about this time to recharge between consecutive kicks. On the other hand, the latter function currently encompasses the batteries level sampling which is sent up to the higher layer using a periodic message (M2 from table 5.1) with a period of $1s$, as well as a set of five sporadic messages (M8 to M12 in table 5.1) that inform the higher layer whenever a hard reset occurs in the respective node.

5.2.3 Communications architecture

The first version of the communication architecture of the robot was based on a native CAN network. New master nodes (active and backup) and a new software layer were required to deploy the FTT-CAN protocol (components shadowed in figure 5.3).

In order to effectively use FTT-CAN it is necessary to identify the information flows related with cyclic activities. Knowing the respective transmission and execution times (presented in table 5.1), one determine which activity triggers each flow and also which should be the appropriate offset of each transmission or activity.

The first aspect is to separate the periodic from the sporadic traffic, as presented in table 5.1. The latter is handled by the asynchronous subsystem similarly to a non-synchronized framework. The periodic traffic is then named using FTT-CAN synchronous identifiers.

The EC duration was set to $5ms$ which is the shortest period among all periodic activities and messages, *i.e.*, the closed-loop motor speed control period. For a trigger message with 5 bytes, the communication overhead is lower than 8.4% ($420\mu s/5ms$) (refer to table 3.2) while the communication load according to table 5.1 is close to 27% of the bus bandwidth at $250kbps$. This value is obtained adding the individual contribution of each information flow.

This is encapsulating each one in a CAN message with DLC equal to the respective number of bytes and using the maximum bit stuffing.

The next step is to analyze the synchronization requirements to identify the set of activities that need synchronization and the respective set of synchronous triggers. Examples of such are the encoders reading or a production of the message with the encoders data. The former results in FTT-CAN identifier 8 (in table 5.2) and the later results in FTT-CAN identifier 9 (in table 5.2).

Finally, the offsets of all messages and synchronous triggers are established so that transmissions are carried out soon after the respective data becomes available and, conversely, activities are triggered with enough slack, so data can be generated as close as possible to the respective transmission instant. Moreover, the synchronous triggers allow triggering of several remote activities at approximately the same time (within a few micro-seconds), as it is required by the odometry function. These concerns lead to increase the freshness of the data in the information flows, reducing the respective end-to-end latency and jitter, with a positive impact in the performance of the respective global control loops associated to the high level behaviours.

Table 5.2 shows the system synchronous requirements, including both synchronous messages and tasks. For the case of the tasks, the 'Destination' column represents the node where the task runs. The offsets extracted from the system requirements are expressed in the column 'Init time' and they are also expressed in number of elementary cycles. Notice that, all periods and offsets ('Init time') are expressed in multiples of the elementary cycle duration.

The table 5.2 was mapped in the hardware and software of the low-level control system.In figure 5.3, the robot low-level communication architecture is depicted (shadowed parts correspond to FTT elements).

It was necessary to add two extra nodes to control the synchronous communications in the CAN bus. These nodes are the master and its replica. Also, a gateway is used to interface the higher level unit with the low-level control system.

Applications running on both the master node and in its replica, add new synchronous messages to the FTT-CAN protocol using the API presented in section 4.2. After that, these masters are started.

At the slave nodes, the application should configure the synchronous variables (messages and tasks) and the asynchronous messages. The function provided by the FTT-CAN slave API that adds a synchronous message (or task), requires the identification of the synchronous message and the nature of the node (producer or consumer). For the asynchronous messages, the application running at the slave must indicate its role (sender or receiver) and the identifier of the message. After this two initial configuration steps, the slaves are ready to start to send/receive messages and trigger tasks.

FTT-CAN ID	Source	Destination	Period (#ECs)	Init time (#ECs)	Short Description
0	Holonomic ctrl	Motor node [1:3]	6	5	Motor speed setpoints
1	Motor 1 node	Odometry node	(1:4)	2	Encoder count in motor 1
2	Motor 2 node	Odometry node	(1:4)	2	Encoder count in motor 2
3	Motor 3 node	Odometry node	(1:4)	2	Encoder count in motor 3
4	Odometry node	Gateway	10	4	Current position
5	Odometry node	Gateway	10	4	Current orientation
6	Gateway	Holonomic ctrl	6	0	Velocity vector (linear)
7	Gateway	Holonomic ctrl	6	0	Velocity vector (angular)
8	—	Motor node [1:3]	1	0	Triggers the encoder readings
9	—	Motor node [1:3]	2	1	Triggers production of messages 1,2,3 at the motor nodes (encoder readings)
10	—	Odometry node	2	3	Triggers the consumption of encoder messages 1,2,3 at the odometry node
11	—	Odometry node	10	3	Event to produce messages 4,5
12	—	Holonomic ctrl	6	1	Triggers the consumption of messages 6,7 in the holonomic controller
13	—	motor node [1:3]	6	6	Triggers the consumption of message 0 in the motor nodes
14	Kicker	Gateway	200	200	Battery status

Table 5.2: Low-level control layer message set and activity tasks

5.2.4 Synchronizing data flows

One of the key features of FTT-CAN is the synchronization among applications. Figure 5.4 shows the timeline of the two main synchronous information flows, related with the motion function (top in figure 5.4) and the odometry function (bottom in figure 5.4).

In what concerns the motion function, the flow is triggered by a pair of messages (6 and 7 in table 5.2 and figure 5.4) sent by the gateway with offset 0 and arriving from the higher layer. These messages contain a velocity vector. These values are received by the holonomic controller that is synchronized in trigger 12 (see table 5.2 and figure 5.4), which is produced right after the transmission of the messages, with offset of 1 EC. This trigger starts the execution of the holonomic controller to process the new velocity vector. The resulting motor speed set-points will be available after $16ms$, which rounds up to 4 ECs [SMA$^+$05, MSF$^+$06, MSF$^+$07]. Thus the respective message (0 in table 5.2 and in figure 5.4) is transmitted to the motor nodes in the following cycle, *i.e.* with an offset of 5 ECs. Trigger 13 (see table 5.2 and figure 5.4) is used to synchronize the closed-loop speed control of each motor with the arriving set-point. The offset is 6 ECs to enforce a reduced latency between reception and

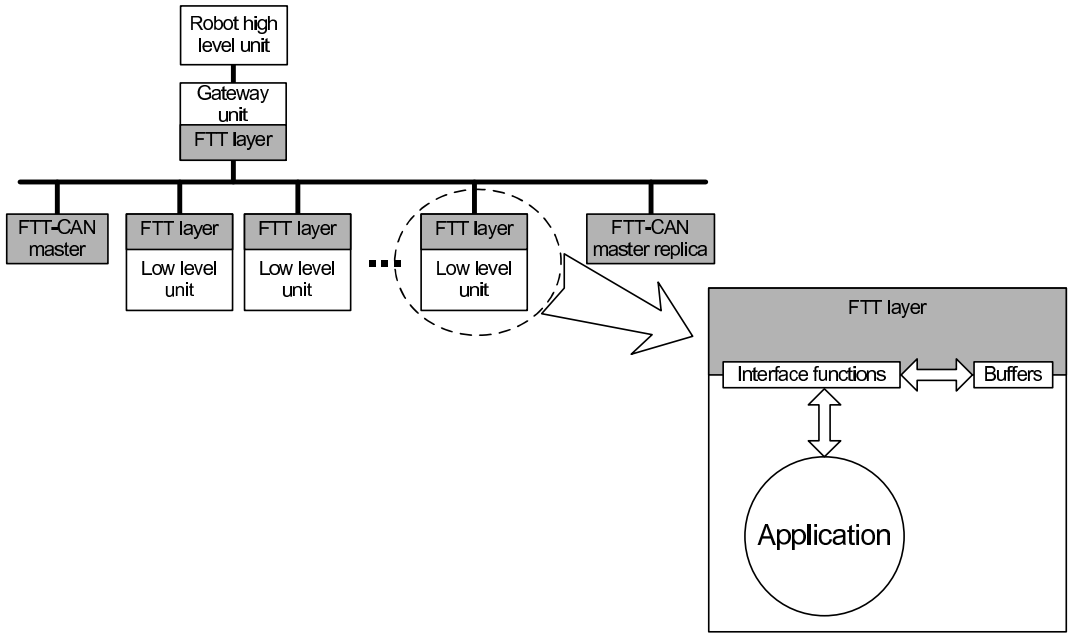


Figure 5.3: FTT-CAN low-level control system

use of the set-points.

The transmission of the next velocity vector, and thus the start of the next cycle, is carried out in the following EC.

In what concerns the odometry function, the respective information flow starts with trigger 8 (see table 5.2 and figure 5.4), with offset 0, which causes the synchronous sampling of the encoders in the 3 motors. These values are locally accumulated until they are transmitted. In the example, the transmission of the encoder readings is set to have a period of 2 ECs (messages 1-3) and the respective values are produced with trigger 9 (see table 5.2 and figure 5.4), in the EC before their transmission. Thus the offset of messages 1-3 is 2 ECs while the offset of trigger 9 is 1 EC. The periods of these entities can vary depending on the desired odometry precision from 1 EC (highest) to 4 ECs (lowest) as expressed in tables 5.1 and 5.2. They can also be suspended (period set to 0) when the odometry function is not needed.

The odometry node work is triggered right after the transmission of the messages 1-3 (see table 5.2 and figure 5.4) carrying the encoder readings, using trigger 10 (see table 5.2 and figure 5.4) with an offset of 3 ECs. Since it executes in less than one EC, the production of the current position and orientation (trigger 11 in table 5.2 and in figure 5.4) is carried out in that EC (same offset of 3 ECs) while the message transmissions (messages 4 and 5 in table 5.2 and in figure 5.4) are assigned to the following EC thus with an offset of 4 ECs.

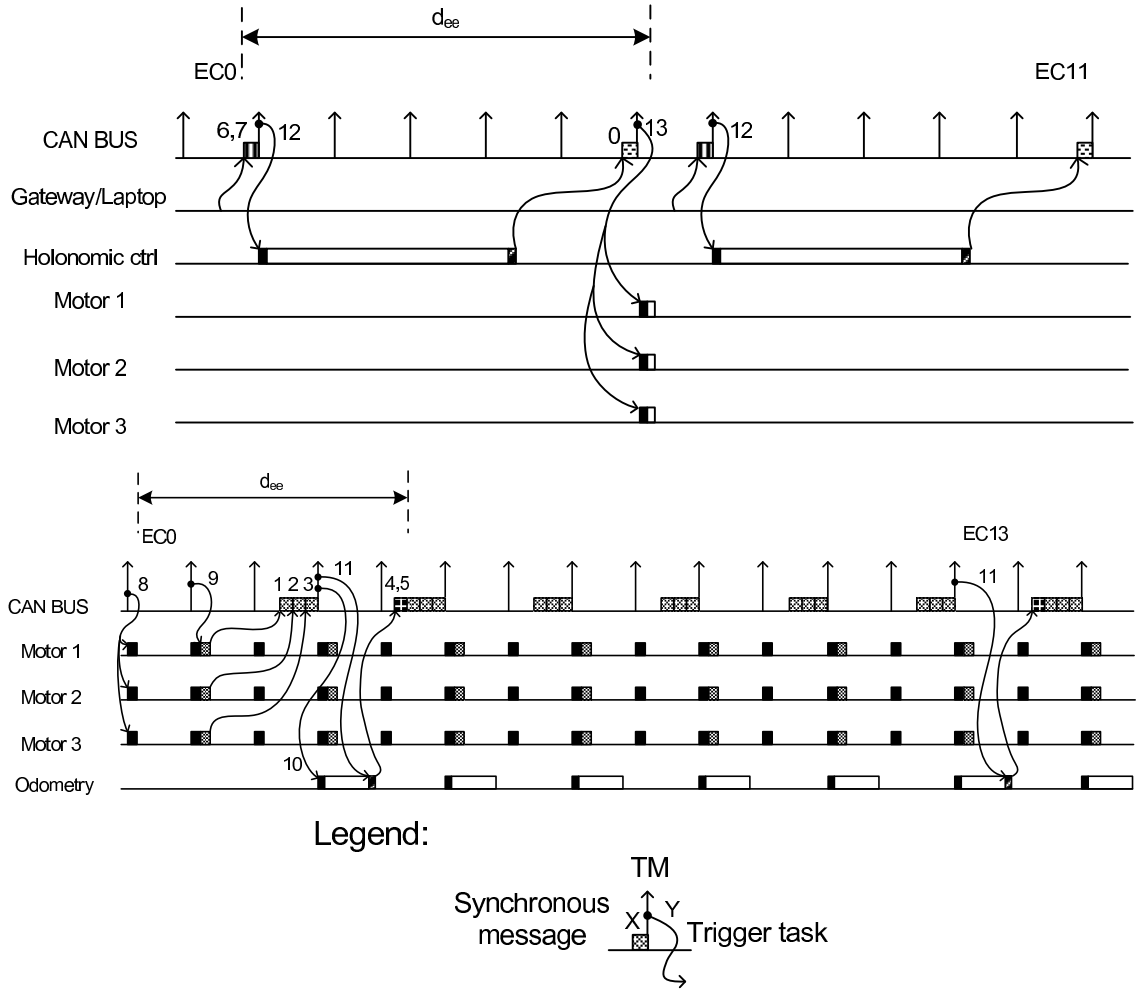


Figure 5.4: Timeline for motion information flow
Top: motion. Bottom: odometry.

5.2.5 Experimental validation

Soccer robots with the FTT-CAN network have been extensively tested in competitions. The low-level control system based on FTT-CAN with one bus and a replicated master have met all the requirements both in terms of performance of the communications and in terms of reliability. The CAMBADA soccer team with five robots have participated in over 50 matches of 30 minutes each (clock-time) [ABB+09, Uni10], besides other robot demonstrations and festivals where the team has played demonstration games. Among others good classifications, the CAMBADA soccer team has won the world cup in 2008 and had the third place in 2009 in world tournament [Uni10]. During all this time, the low-level distributed control system based on FTT-CAN did not have any malfunction (as for the case of standard CAN). This has been considered a sufficient test to validate the noticed operation of a FTT based system in real environment with stringent requirements. In order to further assess the benefits of

using FTT-CAN, some experiments were carried out, aiming to compare timeliness properties with the case in which CAN was used without support for synchronization among remote tasks. For that comparison, the end-to-end delay associated with the two information flows, were measured [MSF+06, MSF+07]:

- Motion control information flow. Measured from the instant in which the gateway starts transmitting a velocity vector to the instant when one of the motors receives the corresponding speed set-point (see top of figure 5.4);
- Odometry information flow. Measured from the instant in which the encoder of one motor is read to the instant when the respective new position is received by the gateway (see bottom of figure 5.4).

The results are presented in table 5.3, concerning the maximum and minimum values observed for the end-to-end delays (d_{ee}) of both information flows in the two approaches referred before, *i.e.* unsynchronized using CAN and globally synchronized using FTT-CAN.

Information flow	CAN		FTT-CAN	
	Max d_{ee} (ms)	Min $d_{ee}(ms)$	Max $d_{ee}(ms)$	Min $d_{ee}(ms)$
Motion	64.4	38.8	27.7	26.7
Odometry	21	12	21.7	21.6

Table 5.3: Timeliness of information flow

It can be seen that the absence of synchronization among multiple chained cycles creates large delays and, mainly, large delay variations (jitter). On the other hand, the synchronization capabilities of FTT-CAN allow establishing adequate offsets that can be used to reduce end-to-end delays and, more important, the associated jitter. The reduction of end-to-end delays is only noticeable when the cycle durations are large enough, at least 3 ECs long. For shorter cycles, as it is the case with the odometry information flow, the temporal resolution of FTT-CAN limits the achievable reduction of the end-to-end delay. However, there is still a high jitter reduction, nearly elimination, which is probably more beneficial for control purposes than the reduction on the end-to-end delay.

Another advantage of FTT-CAN is the simple and efficient process of triggering tasks synchronously. In fact, this is done without extra messages, just using the trigger messages. These triggers allow synchronizing tasks in remote nodes with relatively high precision. In the specific case of the closed-loop speed control in the 3 motors, the use of triggers allowed to synchronize all the loops within +/- 130μs (this value has been obtained using an oscilloscope).

To assess the feasibility and correctness of the master replacement, some experiments were carried out in the network. Apart from the synchronous messages, asynchronous ones (M5.1, M5.2, M7, M8, M9, M10, M11 and M12 in table 5.1) were also injected onto the bus but with lower priority than those involved in the master replication mechanisms.

In order to assess both the accuracy of the worst-case synchronization time estimation and the reliability of the master replica re-synchronization scheme, a software routine injects an inconsistency in the backup's master requirement table and measured the elapsed time until the backup master became synchronized with the active master. Measured values ranged from $40.2ms$ to $45.4ms$ (this value has been obtained using an oscilloscope). This variation can be explained with the different synchronous scheduling in each EC which directly interferes with the FTT-CAN asynchronous window size, that is often larger than $2.1ms$ (minimum asynchronous window size). Thus, if more room is available in the asynchronous windows, the SRT is transferred faster.

As referred in previous chapter (see figure 4.2), the static data of the SRT are the message identifier, data size (the bit size is derived from the data size), period and deadline, while the scheduling state data consists of the absolute deadline, the 'Mask' field (see figure 4.2) and the relative phasing of the messages at the beginning of the next plan. Each of these properties are encoded in one byte, resulting in a synchronous requirement table occupying 126 bytes (there are 14 variables, see table 5.2, occupying 9 bytes each).

To test the mechanism of master replacement upon a failure of the active master, the active master was unplugged from the network several times. The backup master took an average time of $310\mu s$, measured with an oscilloscope, to replace the active master (see figure 5.5, where 'TMA' is a trigger message sent by the active master and 'TMB' is a trigger message sent by the backup master). Notice that this value depends on the bit stuffing of the trigger messages. In fact, the replacement timer of the backup masters is triggered when the previous TM is received, which is affected by bit stuffing. When trigger messages have 5 data bytes, the number of stuff bits ranges from 0 to 15 [NHN03]. This corresponds to a maximum variation of $60\mu s$ at the current bitrate. The time taken by the backup master to replace the active is not noticed in the robot behaviour and thus, there is no effect on its movements or tasks.

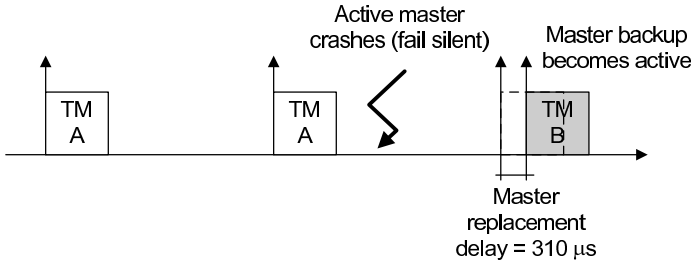


Figure 5.5: Master replacement delay

In order to assess the synchronous requirement table update protocol, the high level robot control application issued several update requests to the master node in order to change the sampling period of the wheel encoder values. This is something to be done automatically in the future because it enables an adjustment to the robot dynamics. That is, the sampling

rate is increased when the robot speed is high and decreased when the robot speed is low. The response time of each synchronous requirement table update request ranged between 15.8ms and 19ms (between 3 and 4 ECs), as expected by *Ferreira* [Fer05].

5.3 Multiple buses experimental platform and results

In the previous section, the single bus assessment made in a robotic soccer team has been presented.

In this section, the test platform and results for the multiple buses are presented. For this test platform, a fault injection system and a delay measurement system was developed. Next, this test platform and their elements will be explained. The results follow the presentation of the test platform, with special focus to the practical and expected values.

5.3.1 Experiment's rationale

The validation of the multiple buses FTT-CAN network is based on a fault injector capable causing faults both on the buses and at the master nodes. A fault injector requires a measurement equipment to register the impact of the fault in the network, and in the master nodes. A Delay Measurement System (DMS) was developed for this purpose. It is capable of monitor the buses to identify the relevant events, mainly trigger messages, so it can measure the delays between the instant a fault is injected and the instant the corresponding error is visible on the buses.

Only the replacement of buses and masters were assessed, thus, inclusion of slaves in the test platform are not so important.

Figure 5.6 depicts a global view of the experimental validation setup.

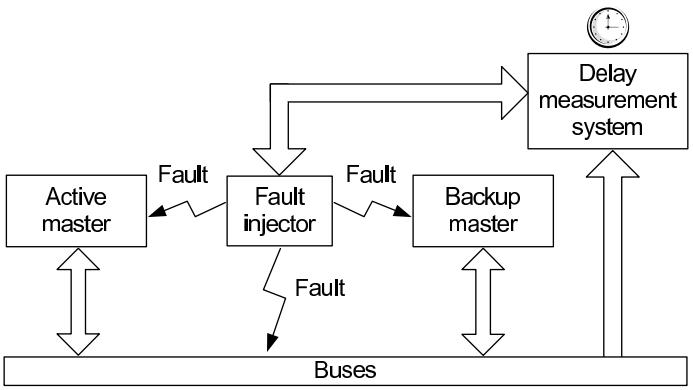


Figure 5.6: Assessment setup global view

This fault injector is able to cause entire fault scenarios presented in section 3.5.3 considering on the fault hypothesis defined in section 3.5.2.

For the case of the master nodes, the fault injector imposes a fail silent behaviour, by turning a master node off.

For the case of the bus faults, the fault injector can cause the following faults:

- Force lines at logic '1'. This corresponds to a stuck-at recessive bits fault;
- Force lines at a logic '0'. This corresponds to a stuck-at dominant bits fault;
- Open the bus. This corresponds to a situation of cutting the bus wires (partitioning).

To understand the time marks the DMS must sense when a bus or master fault occurs, the bus fault timeline and the master fault timeline will be explained next.

Bus fault timeline

The fault measuring equipment takes advantage of the systolic nature of the trigger message and the position of the masters (at the end of the buses) in order to detect bus and master errors [SBFF09]. Notice that the masters are placed at both ends of the buses to ensure that bus partitions are always detected.

A bus is assumed to be faulty whenever the trigger message issued by the active master is not received by the backup master within a given time window ($TMTW$). If a backup master detects a faulty bus, a high priority asynchronous message is transmitted in all buses, indicating the active master that a reconfiguration is required [SBFF09]. The reconfiguration corresponds to reallocate and reschedule all the messages from the faulty bus to the remaining ones according to a best effort policy. Figure 3.12 presents the timeline of a bus error detection while figure 5.7 depicts the timeline of an error recovery.

In figure 5.7, the asynchronous message issued by the backup master to indicate an error in bus 2 is identified as 'BEAM' (Bus Error Asynchronous Message). The time elapsed since the transmission of the last trigger message (TM_y^{X-1} in figure 5.7, where 'y' represents the bus number) and the instant of the error is presented as t_{phase}. After missing the TM_2^X in bus 2, the backup master waits the transmission window ($TMTW$) to issue the BEAM message. The delay from the fault to the reception of this message is identified as t_{BEAM}. After its reception, the active master changes the bus 2 messages to bus 1 (operation indicated as 'Bus changing' in figure 5.7) and their schedule will appear in TM_1^{X+3} (elapsing $t_{re\_schedule}$ time). The time elapsed from the last correct trigger message (TM_y^{X-1}) to the respective occurrence is called a relative delay for that occurrence ($Rel(t_{BEAM})$ for the BEAM message and $Rel(t_{re\_schedule})$ for the new schedule).

Note that the new schedule cannot be present in TM_1^{X+2} since the scheduling for this EC is initiated during the TM_1^{X+1} and eventually terminates after the reception of the request message. To guarantee the mutual exclusion of the internal tables, as presented in [SFF07c] and discussed before (at section 3.5.5), the bus changing operation is performed in a copy instance of the SRT.

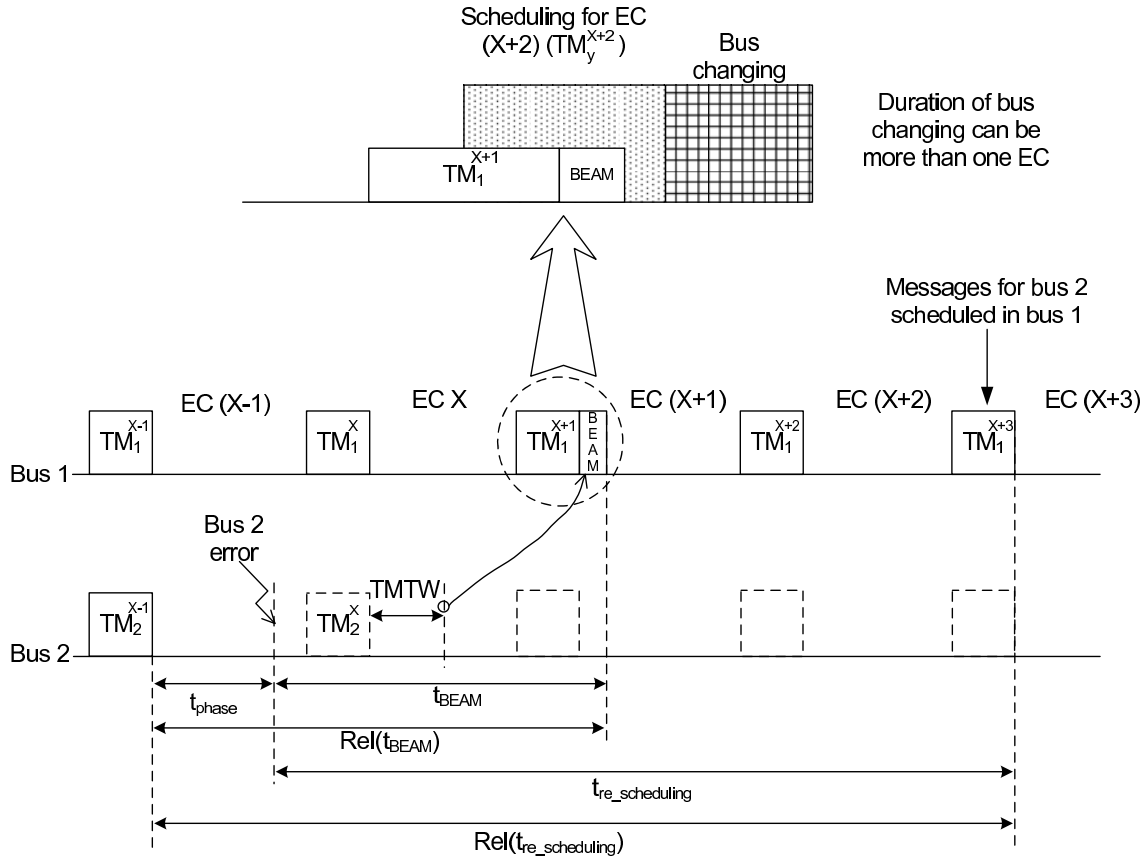


Figure 5.7: Bus error timeline

Master fault timeline

Concerning master errors, as defined in the fault hypothesis, at least one bus at a time will be working correctly. As a result, if the backup master does not receive the trigger messages in any bus, this can only mean that the active master is faulty [SBFF09]. Whenever this happens, the backup master waits the transmission window ($TMTW$) before gaining control of the buses and starting the trigger message transmission. In that way if the backup master is the only working master, bus error detection is unsupported [SFF07c], thus the system enters in a degraded mode. Figure 5.8 depicts the timeline of an error in the active master.

In figure 5.8, t_{phase} indicates the time elapsed from the last trigger message (TM_y^{X-1}, where 'y' is the bus number) to the error occurrence in the FTT-CAN master. Since the trigger messages can have a different number of bits (due to bit stuffing), two different delays were defined: t_{TM_first} and t_{TM_second} (in chapter 3 only t_{TM_new} was defined because was assumed that the master sends both trigger messages at same time with the same number of bit stuffing). The former is the time elapsed from the error in the active master and the first trigger message reception instant. The later is similar, but regarding the second trigger message reception. The time elapsed from the last correct trigger message (TM_y^{X-1}) to the

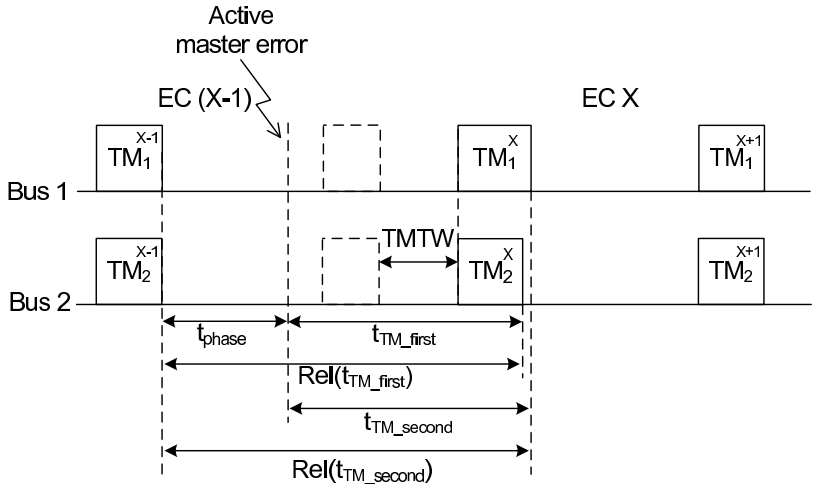


Figure 5.8: Master error timeline

respective occurrence is called a relative delay for that occurrence ($Rel(t_{TM\_first})$ for the t_{TM_first} and $Rel(t_{TM\_second})$ for the t_{TM_second}).

5.3.2 Fault injector and delay measurement system

Fault injector

To switch off a FTT-CAN master (simulating a fail silent behaviour), the fault injector acts on the reset pin of the microcontroller (holding it at reset state).

The fault injection hardware, the way how to cause faults on the buses, is more complex than the one required to turning off the master node, since it must be able to cut the buses or to put them in a recessive or dominant state. To meet these goals, some specialized hardware was developed. The specialized hardware is based on a multiplexer made of three 4066 analog switches [Mot95]. Figure 5.9 presents the structure and connections of the multiplexer.

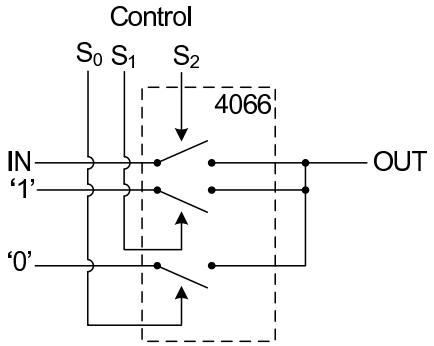


Figure 5.9: Multiplexer structure

The multiplexer is suitable for the CAN low and CAN high bus lines, thus, two multiplexers are needed for each bus, as depicted in figure 5.10. In table 5.4, the voltages for CAN

high and CAN low according the ISO11898 signalling [ISO93] are presented.

Parameter \ CAN bit	Dominant	Recessive
Logical value	'0'	'1'
CAN_H voltage	$3.5V$	$2.5V$
CAN_L voltage	$1.5V$	$2.5V$

Table 5.4: CAN high and CAN low lines voltages

In figure 5.9, the signals S_0, S_2 and S_3 are the control switches signals. According to the combination the control signals S_0, S_2 and S_3, the output takes a value presented in table 5.5, where:

- "0" means the respective switch is open and "1" means the respective switch is closed;
- ND stands for "not defined" as it is an inconsistent combination from the point of view of the CAN bus.

S_0	S_1	S_2	OUT
0	0	0	ND
0	0	1	IN
0	1	0	Recessive
0	1	1	Recessive
1	0	0	Dominant
1	0	1	Dominant
1	1	0	ND
1	1	1	ND

Table 5.5: Multiplexer functions

As referred, each bus needs a set of two multiplexers. The combination of two multiplexers for using in one bus is presented in figure 5.10.

The internal structure of the fault injector is presented in figure 5.11.

A personal computer is used to configure the fault injector through its parameters configurator (see figure 5.11). Examples of parameters to configure are the type of fault to inject and the instant of the fault occurrence.

The multiplexers are controlled by the 'fault handler', a module of the fault injector (see figure 5.11), through the hardware drivers. The 'fault handler' also has digital outputs that are used to reset the master nodes (see figure 5.11). The 'timer handler' controls the timers and imposes the moment to inject the fault.

The delay measurement system (DMS)

The Delay Measurement System (DMS) measures the time elapsed between the injected fault and the respective error, so it needs information from the fault injector and also from

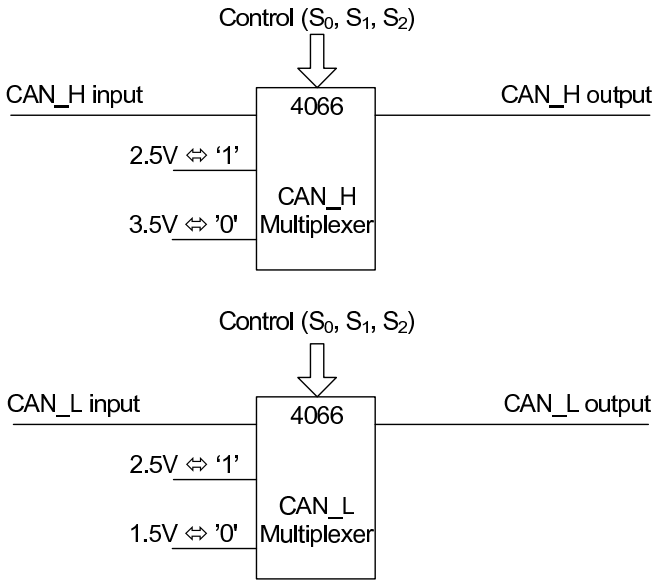


Figure 5.10: CAN bus multiplexers

the buses.

The fault injector and the DMS are able to repeat an experiment several times. The delay measurements are sent to a personal computer for offline processing and analysis.

The internal structure of the DMS is presented in figure 5.12.

The delay measurement system is configured through the parameters configurator that, in turn, programs the event recognizer. The event recognizer can distinguish events through the hardware drivers that arrive from a computational or communication system. The parameters configurator can also program the time stamper, for example to define the base time or the time range of the delays.

One of the outputs of the DMS is a histogram representing the collected delays. The histogram parameters that can be configured are resolution, the number of bars and the value of the first bar of the histogram.

The DMS also computes the average and standard deviation of the measured delays.

The delay measurement system is built on top of a dsPIC30F6012A microcontroller [BSF07].

Using the DMS in related work

The DMS is a low cost, flexible, customizable and easy to operate device allowing the performance assessment of several communication systems through the transmission of a serial data stream. Its architecture makes it suitable to be used in any communication technology supporting the transmission of a serial data stream or, with changes, any communication protocol. Thus, the DMS is not a system specific for the purpose of this work and can be

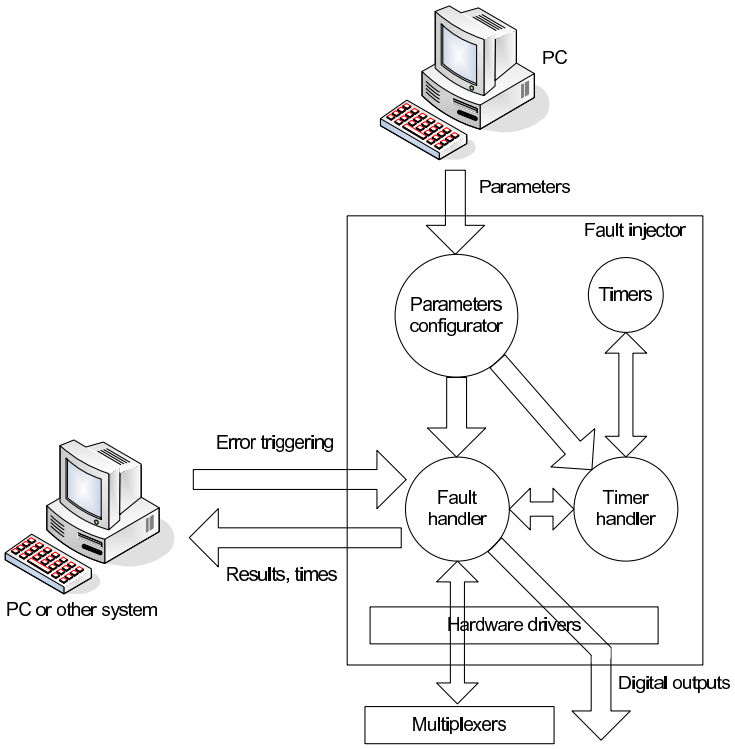


Figure 5.11: Fault injector internal modules

applied to other test platforms.

The delay measurement system was used during the validation process of the Virtual Token Passing Ethernet h-BEB (VTPE-hBEB) protocol proposed by *Carreiro et al.* [CFSV04, Car08, MCB+10].

The use of a serial stream in VTPE-hBEB is justified by the option, in an early development stage, to perform two assessments on the communication system: subjective and objective. The subjective assessment consisted in transmitting music through the communication system and evaluating its quality at the receiver end. Although this approach can be interesting for demonstration purposes, it is not suitable for assessing a real-time communication system, since it does not provide quantitative figures [BSF07]. In the context of VTPE-hBEB protocol objective validation, DMS was able to measure the Ethernet frame transmission delays and the corresponding processing delays on the microprocessors [FBSC07] and compute the respective delays histograms.

Recently, the DMS was used for measuring delays in location systems based on wireless personal area networks with Zigbee. Also, it was used to delay measurements in a system used to capture channels in noisy wireless communications environments [BF10].

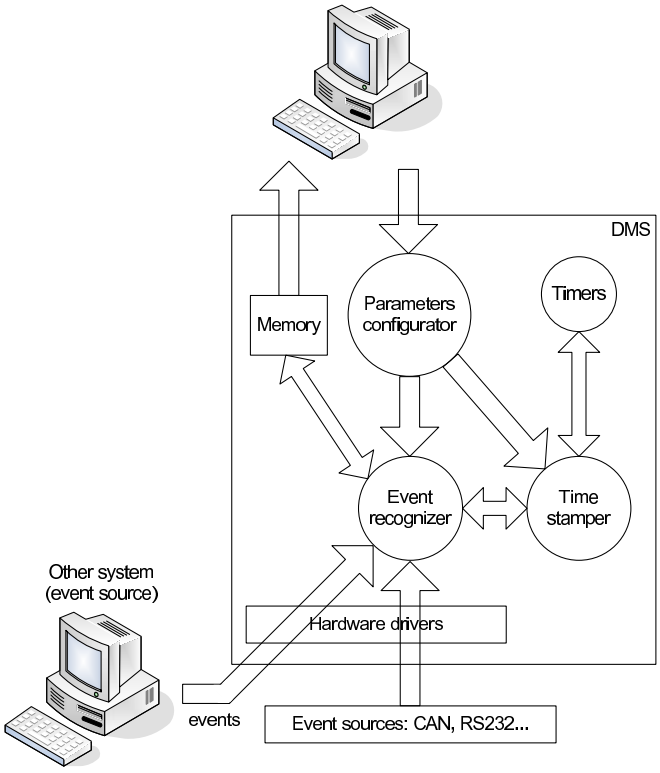


Figure 5.12: DMS internal structure

Joining time measurement and fault injection

As it was obvious from previous sections, there is a logical dependency between fault injection and delay measurement. So, both modules were physically implemented together, because:

- The delay measurement system must communicate with the fault injector;
- Since both modules must communicate, the communication delays are smaller and the resulting precision is higher if both modules share the the same processor;
- As both systems must be configured, it is easier to configure them if they are located in the same microcontroller;
- The implementation of both modules in a single microprocessor is easier because the communication between them is via internal memory.

Figure 5.13 depicts the new device that joins the DMS and the fault injector, the eXtended Delay Measurement System (xDMS).

The xDMS has been developed also in a dsPIC30F6012A [Mic08] as the initial DMS (the hardware is shown in figure 4.6). The communication with the personal computer for configuration and data upload purposes is done through a RS232 serial line.

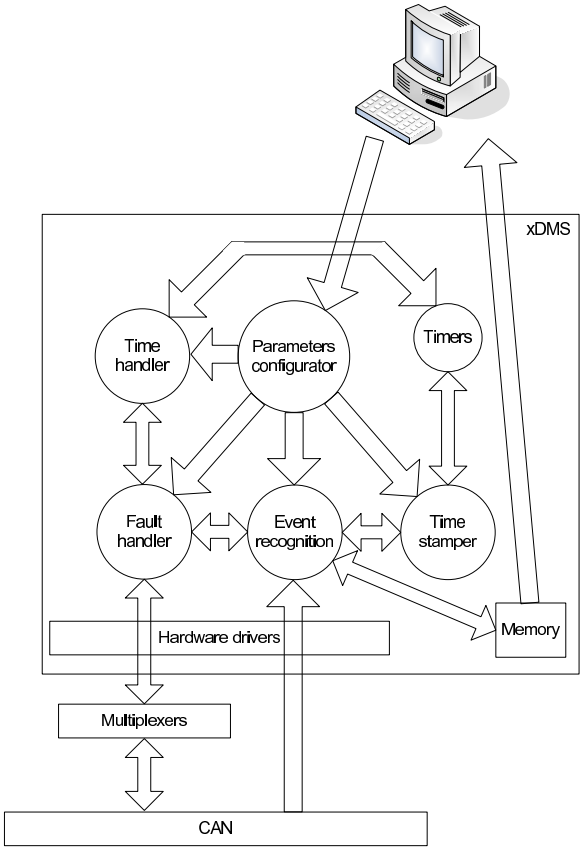


Figure 5.13: xDMS internal modules

5.3.3 Experimental setup and results

A measurement setup was developed to validate the multiple buses multiple masters FTT-CAN communication system. This setup aims at providing the basic features required for the assessment of bus and master redundancy and, as such, it includes an active master and a backup master connected by two CAN buses (bus 1 and bus 2).

The evaluation of the delays associated with reconfiguration as result of bus or master errors is conducted using the extended delay measurement system (xDMS) [BSF07] presented before. This evaluation aims the assessment of the delays presented in figure 5.7 for the case of a bus error (t_{BEAM}, $Rel(t_{BEAM})$, $t_{re\_scheduling}$ and $Rel(t_{re\_scheduling})$) and in figure 5.8 for the case of a master error (t_{TM_first}, $Rel(t_{TM\_first})$, t_{TM_second} and $Rel(t_{TM\_second})$).

For the present assessment, a bus error is either a bus segmentation or setting it in a dominant state. For proof of concept, the recessive state was also evaluated, however, no results will be presented.

Figure 5.14 presents a general architecture of the measurement setup including the xDMS. Figure 5.15 displays a picture of the measurement setup, excluding the PC presented in figure 5.14.

For each measured delay, a set of trials will be performed in an automatic way by the

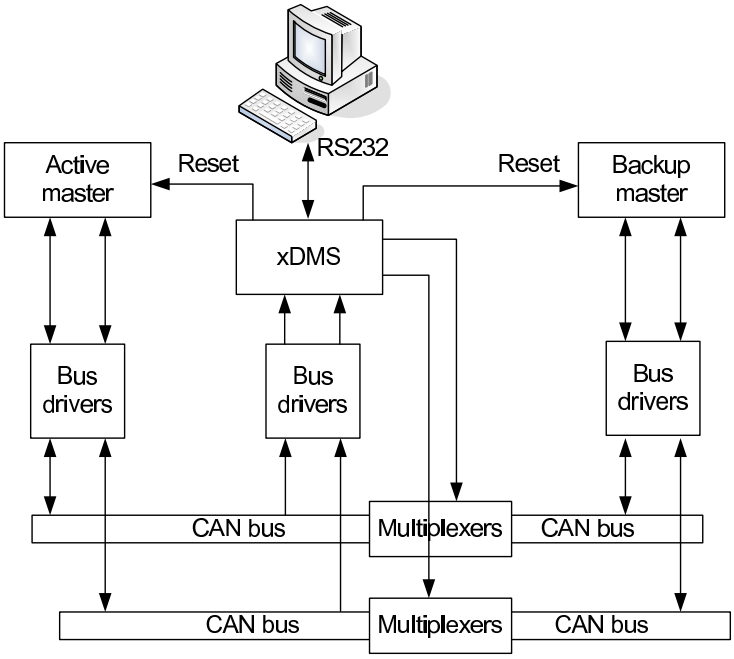


Figure 5.14: Measurement architecture

xDMS (configured by the personal computer). This set of trials is called a batch.

A trial is conducted using the following sequence of operations:

1. Reset the masters (active and backup) begins the trial;
2. Wait for a random number of trigger messages (the xDMS has the possibility to bound this random number). This random number is $X - 1$ figures 5.7 and 5.8;
3. Wait for a random amount of time (less than an elementary cycle). This value is represented as t_{phase} in figures 5.7 and 5.8;
4. Apply the specified error (the error is specified as parameter using the personal computer);
5. Measure the delays: t_{BEAM} and $t_{re\_scheduling}$ (or $Rel(t_{BEAM})$ and $Rel(t_{re\_scheduling})$) in case of bus bus error and t_{TM_first} and t_{TM_second} (or $Rel(t_{TM\_first})$ and $Rel(t_{TM\_second})$) for the case of master error.

The random amount of time elapsed since the last trigger message (t_{phase} in figure 5.7 and 5.8) is limited by the first next received trigger message. Figure 5.16 describes the delay components of the maximum permitted phase for error injection.

The delay between the dispatching of two trigger messages in two buses results from the processing overhead on the master, represented in figure 5.16 by t_1. The time duration of a trigger message (TM) is not constant, *i.e.* it depends on the payload and on the associated

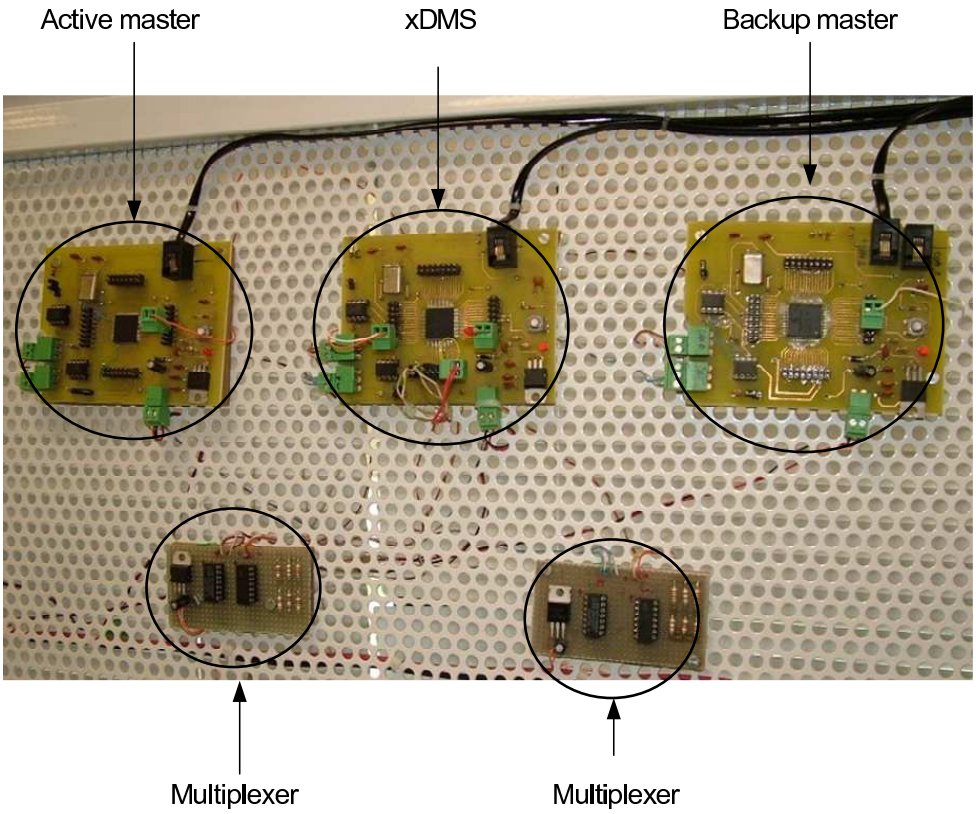


Figure 5.15: Developed system

stuffing bits. The worst case scenario occurs when one TM has the maximum length (TM_2^X) and the other has the minimum length (TM_1^{X+1}). In figure 5.16, the size of the trigger message and of the elementary cycle is represented by ltm_y^x (where y is the number of bus and x is the elementary cycle number) and LEC, respectively. The extended delay measurement system handles trigger messages with interrupt service routines. The time elapsing from the end of a TM to the instant of its processing in the interrupt service routine is represented by t_2.

Thus:

$$\max(t_{phase}) = LEC - t_1 - t_2 - (\max(ltm_y^x) - \min(ltm_y^x)), \forall y = 1, ..., N_{buses}, \forall x = 0, ..., \infty \tag{5.1}$$

Where $\max(ltm_y^x)$ and $\min(ltm_y^x)$ are the maximum and minimum length of a trigger message, respectively.

Thus, $\max(ltm_y^x) - \min(ltm_y^x)$ is the maximum delay associated with bit stuffing in a TM. $\max(t_{phase})$ is the value of the maximum phase that can be used (t_{phase}). This value ensures that the error is never triggered in the time window elapsing between the end of TM transmissions, which could result in an error injection on the next elementary cycle, thus invalidating the trial due to error triggering in the wrong elementary cycle.

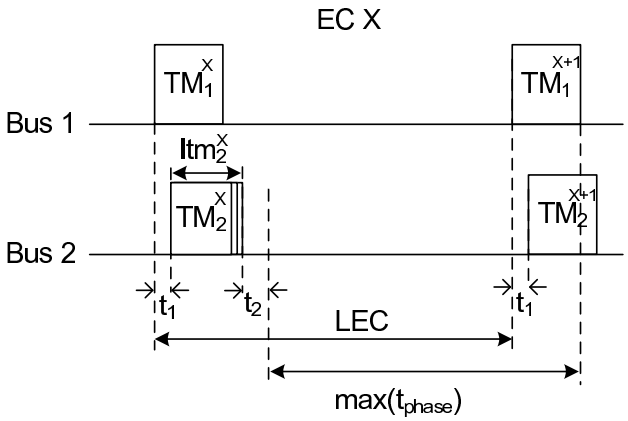


Figure 5.16: Maximum of t_{phase}

The main parameters for the $\max(t_{phase})$ calculation and for the setup previous presented are:

- Number of buses: $N_{buses} = 2$;
- Elementary cycle time duration: $LEC = 5ms$;
- CAN bitrate: $250kbps$;
- Trigger message transmission window: $TMTW = \frac{EC}{4} + \frac{LTM}{2} = 1.514ms$;
- Time between the sending of trigger messages in different buses (t_1 in figure 5.16): $t_1 = 28\mu s$ (this value has been measured with an oscilloscope);
- Time for the processing of an interrupt reception of the trigger message in the xDMS, and prepare the fault injection (t_2 in figure 5.16): $t_2 = 89\mu s$ (this value has been measured with an oscilloscope);
- $\min(ltm_y^x) = 444\mu s$, corresponding to 8 data bytes without bit stuffing, refer to equations 3.1 and 3.16;
- $\max(ltm_y^x) = 540\mu s$, corresponding to 8 data bytes, maximum number of stuff bits (refer to equation 3.17).

Using these parameters and equation 5.1, the phase of an error (t_{phase}) can take the maximum value of $\max(t_{phase}) = 4.787ms$. Thus, faults are injected in the bus or in the active master at random instants within an elementary cycle (in fact within 0 and $4.787ms$). In figure 5.17, a histogram of the fault injection time is presented.

Note that the instant of the fault injection is randomly distributed across an elementary cycle. The histogram presented in figure 5.17 has been acquired during one experience batch to demonstrate that the instant of the injection is randomly distributed. It is assumed that,

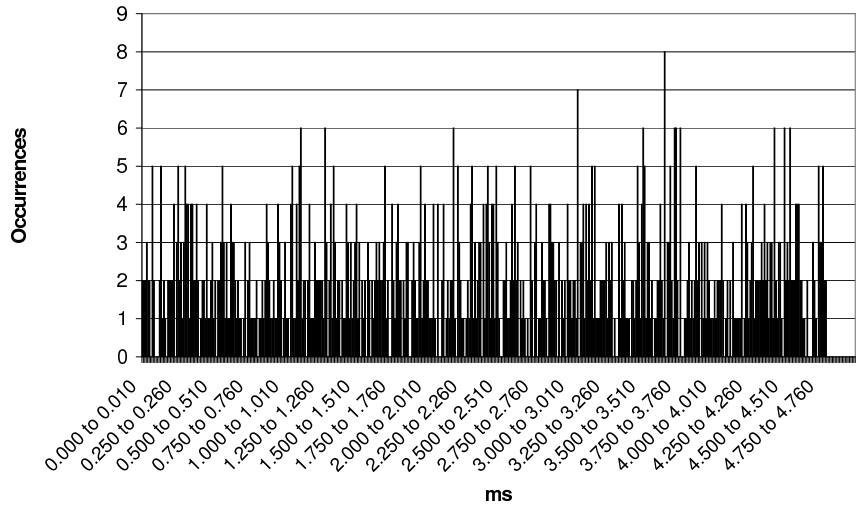


Figure 5.17: Fault-injection instant histogram

the random distribution is sufficient to perform the measurements and it is not necessary to have a uniform distribution of this instant. Also note, this random distribution of the error injection delay is valid for the bus error and master error measurements.

Adding to the main parameters of the system presented before, the xDMS has the possibility to configure the trial parameters, that are:

- Number of trials per batch. Represented by N_{trials} in tables 5.6 and 5.7;
- Maximum number of EC elapsed before the error occurrence (is the number of elementary cycles before 'EC X' in figure 5.7 and in figure 5.8). This value is randomly distributed. Represented by $\max(N_{EC\_b\_f})$ in tables 5.6 and 5.7;
- Maximum time within an elementary cycle elapsed before an error (t_{phase} in figure 5.7 and in figure 5.8. Explained in equation 5.1);
- Type of fault to inject. This can be a bus fault (cut off or dominant or recessive bit) or a master fault (acting on master reset pin to shut it down);
- Histogram parameters. The xDMS will produce histogram data that can be plotted in a PC software like Microsoft Excel®. These parameters are:
  - Number of bars. Represented as N_{bars_hist} in tables 5.6 and 5.7;
  - Value for the first bar. Represented as $first_{bar}$ in tables 5.6 and 5.7;
  - Resolution for each bar. Represented as Res_{hist} in tables 5.6 and 5.7.

Experimental results for each type of fault and delay to measure were obtained with the xDMS configured with the parameters presented in tables 5.6 (for bus faults) and 5.7 (for master faults).

Parameter \ Delay	Bus fault			
	t_{BEAM}	$Rel(t_{BEAM})$	$t_{re\_scheduling}$	$Rel(t_{re\_scheduling})$
N_{trials}	1000			
$\max(N_{EC\_b\_f})$	1000			
$\max(t_{phase})\,(ms)$	4.787			
N_{bars_hist}	500			
Fault description	Bus partition recurring to multiplexers			
$first_{bar}\,(ms)$	5.2	10	15	19.9
$Res_{hist}\,(\mu s)$	10	0.5	10	0.5

Table 5.6: xDMS parameters for bus faults

Parameter \ Delay	Master fault			
	t_{TM_first}	$Rel(t_{TM\_first})$	t_{TM_second}	$Rel(t_{TM\_second})$
N_{trials}	1000			
$\max(N_{EC\_b\_f})$	1000			
$\max(t_{phase})\,(ms)$	4.787			
N_{bars_hist}	500			
Fault description	Hold reset pin of active master at reset state			
$first_{bar}\,(ms)$	2	6.6	2	6.6
$Res_{hist}\,(\mu s)$	10	1	10	1

Table 5.7: xDMS parameters for master faults

Using the parameters presented in tables 5.6 and 5.7, several batches were run in order to validate the system, namely:

- Just for proof of concept, one of the two wires of the CAN bus was cut off. It was verified that the bus continued to work correctly just with the other wire. This property is a built-in CAN specification property;

- Bus error: recessive. This was also just a proof of concept. Since, with recessive bits, by definition, no interference in the bus is made. Thus, forcing the bus to the recessive state does not affect the communications of the CAN physical layer. The system behaves as expected, this is, no interference in the communication system has been noticed with the injection of recessive bits;

- Bus error: cut off both wires of the bus;

- Bus error: dominant bits. In this test, the bus was injected with dominant bits (the obtained results are similar as the ones obtained for the bus cut off);

- Active master error: the active master was reset to simulate a fail silent master behaviour.

Results presented further are divided into two delay categories: absolute and relative. Absolute delays are measured from the fault instant itself while relative delays are measured from the last trigger message before the occurrence of the fault. In that way a relative delay corresponds to the absolute delay plus the respective t_{phase} (as presented in figure 5.7 and 5.8).

Bus error test results

Figure 5.18 presents an histogram of the bus error asynchronous message delay (t_{BEAM} in figure 5.7) measured from the instant of the fault injection (in presented case is a bus partition). As shown in figure 5.7, this delay is random (depends on the instant where the error was injected thus, depends on t_{phase}) and it ranges between $5ms$ and $10ms$ (1 and 2 ECs, respectively). However, if t_{BEAM} is measured from the last TM, it is reasonably constant as shown in figure 5.19.

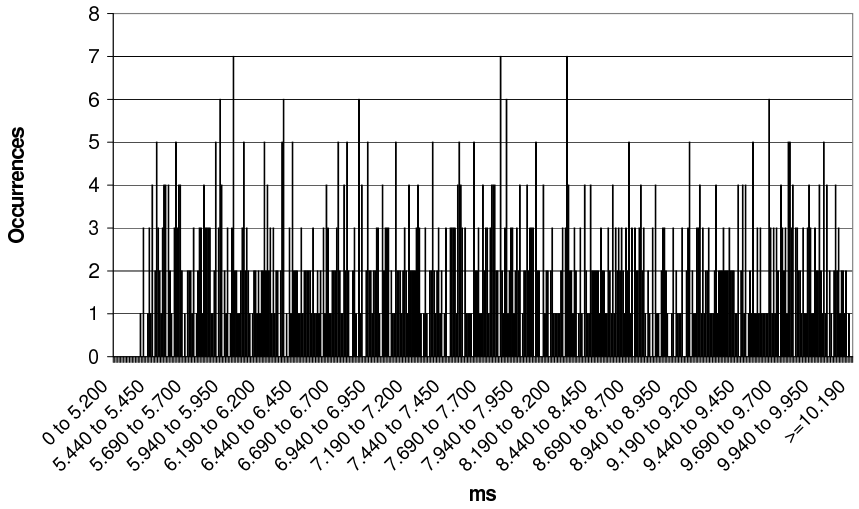


Figure 5.18: t_{BEAM} histogram (absolute delay)

As it can be seen in figure 5.19, the time elapsing from the last TM to the transmission of the re-scheduling request message is almost constant in all trials, which suggests that the backup master recognizes the bus error and requests a bus re-scheduling in a consistent and timely fashion. The small difference between the maximum and minimum value observed (difference of $17\mu s$ with minimum $10,173\mu s$ and maximum $10,190\mu s$) results from the bit stuffing of the trigger message. The average of t_{BEAM} is $10,182\mu s$ with a standard deviation of $7\mu s$. Because t_{BEAM} includes the duration of the associated BEAM message it can take values higher than $10ms$ (2 ECs) (refer to figure 5.7).

After the reception of the request message, the active master reschedules the faulty bus traffic in the remaining (operational) buses. The new scheduling will appear two trigger messages ahead (TM_1^{X+3} in figure 5.7). This time is denoted by $t_{re\_schedulling}$ and was measured

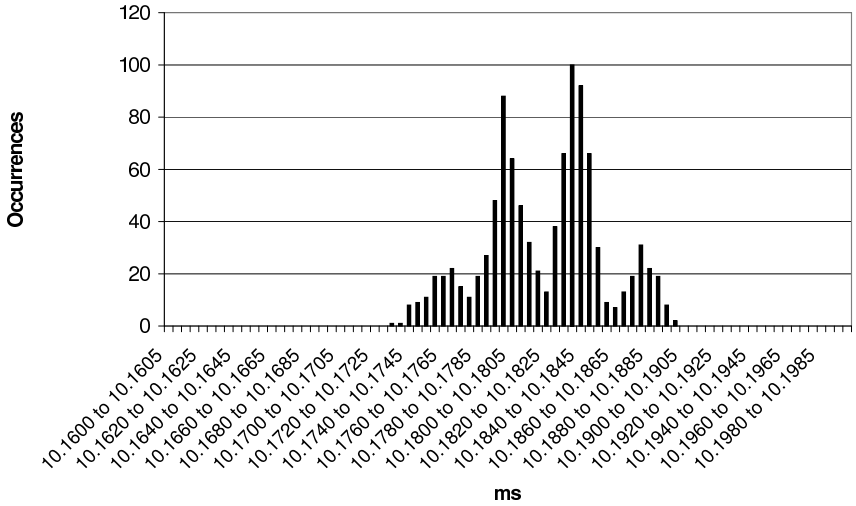


Figure 5.19: $Rel(t_{BEAM})$ histogram (relative delay)

using the xDMS with the set of parameters specified in table 5.6. The corresponding results are shown in absolute and relative terms in figure 5.20 and 5.21, respectively. Absolute delays are measured from the occurrence of the error while absolute delays are measured from the TM before the error.

The histogram of figure 5.21 shows that the time elapsed from the last trigger message before an error to the re-scheduling of the synchronous messages has a maximum value of $19,920\mu s$, *i.e.* the system takes a maximum of 4 elementary cycles to recover from the error. The minimum value obtained in the batch were $19,820\mu s$. Thus, the difference between the minimum and maximum is $100\mu s$. These values demonstrate the system reacts to the bus error in a very narrow time interval. Notice also that the average of $t_{re\_scheduling}$ is $19,910\mu s$ with a standard deviation of $4\mu s$.

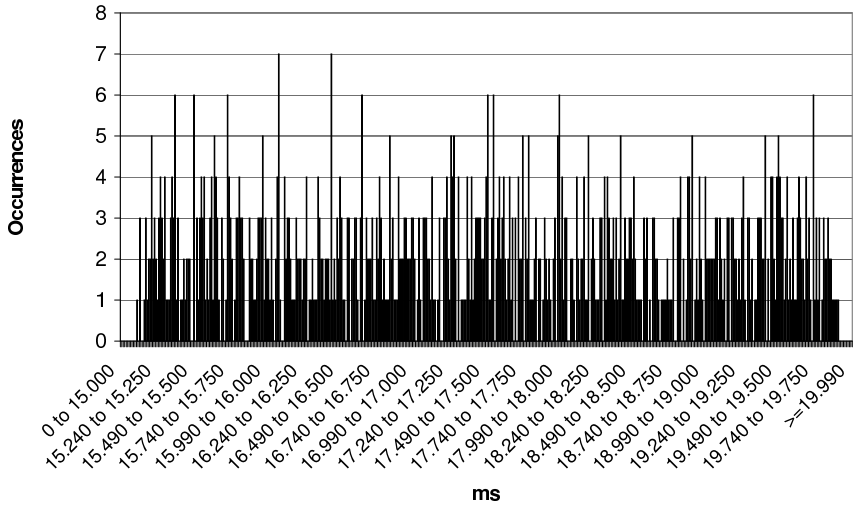


Figure 5.20: $t_{re\_scheduling}$ histogram (absolute delay)

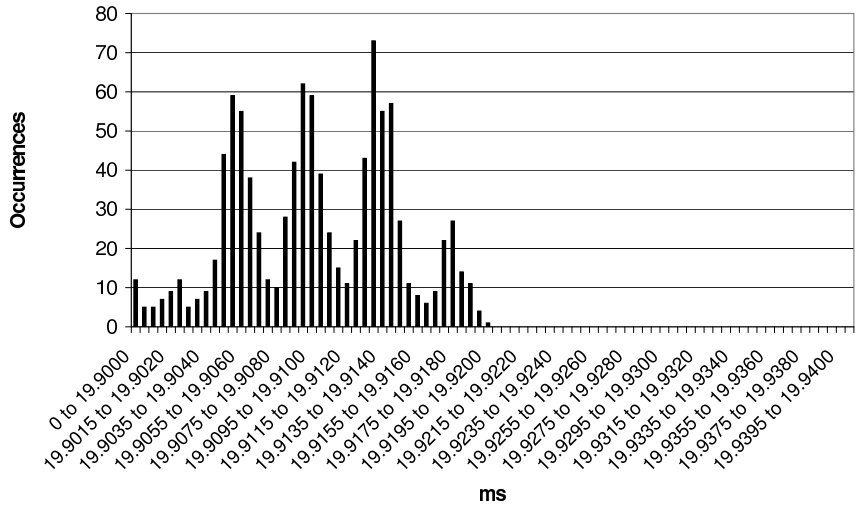


Figure 5.21: $Rel(t_{re\_scheduling})$ histogram (relative delay)

Note that the presented values are for the case of bus partition. For the case of injecting a dominant bit the histograms are similar to the ones presented in figures 5.18, 5.19, 5.20 and 5.21. Just for the comparison with the bus partition, in figure 5.22 there are presented the delays $Rel(t_{BEAM})$ and $Rel(t_{re\_scheduling})$ when dominant bits are injected in the bus (left of figure 5.22 and right of figure 5.22, respectively).

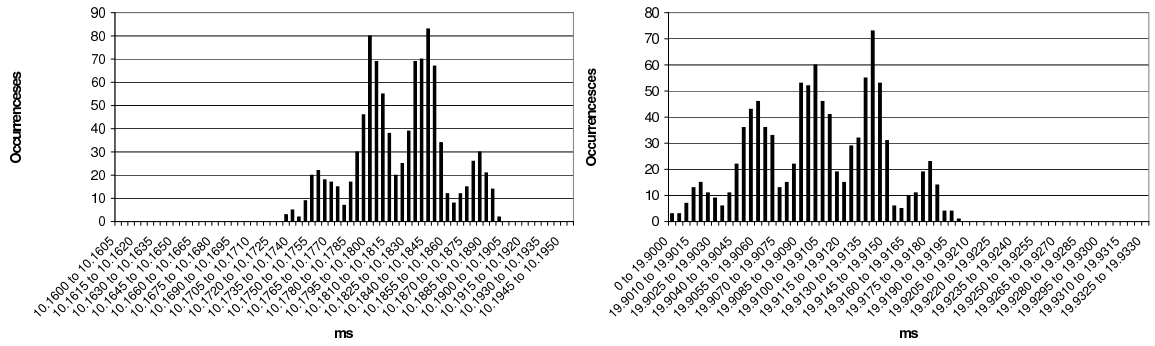


Figure 5.22: $Rel(t_{BEAM})$ and $Rel(t_{re\_scheduling})$ injecting a dominant bit

Active master error test results

As in bus errors, the xDMS waits a random number of ECs (randomly distributed) before injecting a fault in the active master. As a consequence, the active master fails by crashing (fail silent) and the backup master waits a period of time denoted by $TMTW$ before gain control of the buses and transmitting the next trigger message. As such, the first elementary cycle transmitted by the new active master lasts longer than usual, (see figure 5.8) depending on the defined $TMTW$ (in the presented case $TMTW = 1.514ms$).

As illustrated in figure 5.8, there is a small difference in the reception instants of the trigger message in both buses (TM_1^X and TM_2^X). This occurs because the active master

has a single processor architecture that hinders simultaneous transmissions. Therefore, TM transmissions are dephased by a small amount of time that, together with the TM length variation introduced by stuffing bits, results in different reception instants. The histogram of the absolute delays associated with the first trigger message reception, after the active master fault injection (t_{TM_first}), is presented in figure 5.23. Figure 5.24 shows the relative delays ($Rel(t_{TM\_first})$) corresponding to the sum of t_{phase} and absolute t_{TM_first} delays.

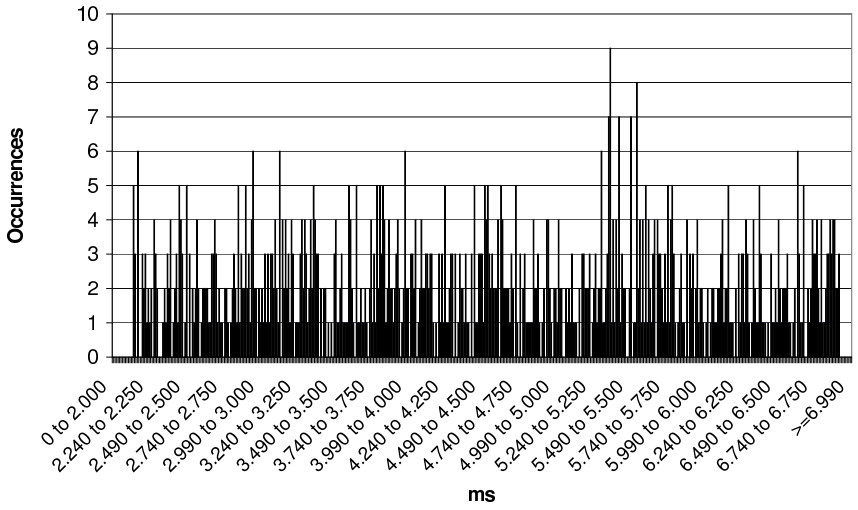


Figure 5.23: t_{TM_first} histogram (absolute delay)

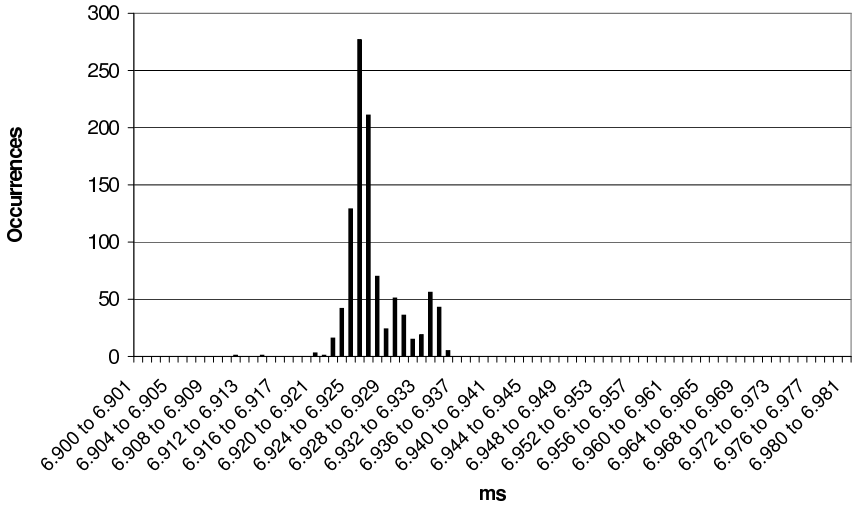


Figure 5.24: $Rel(t_{TM\_first})$ histogram (relative delay)

The first trigger message is received after an average delay of $6,927\mu s$ with a standard deviation of $3\mu s$. The maximum value obtained was $6,935\mu s$, while the minimum value was $6,912\mu s$. Thus, the difference between the maximum and the minimum is $23\mu s$. This difference between the maximum and the minimum of 4 bits time can be can arise bit stuffing of the CAN messages. These value demonstrate the system reacts to a master error with a

very predictable behaviour.

The absolute delay for the reception of the second trigger message (t_{TM_second}) is shown in figure 5.25. It can be seen that t_{TM_second} and t_{TM_first} histograms are similar although t_{TM_second} is delayed relatively to t_{TM_first} due to the different dispatch instants and stuff bits.

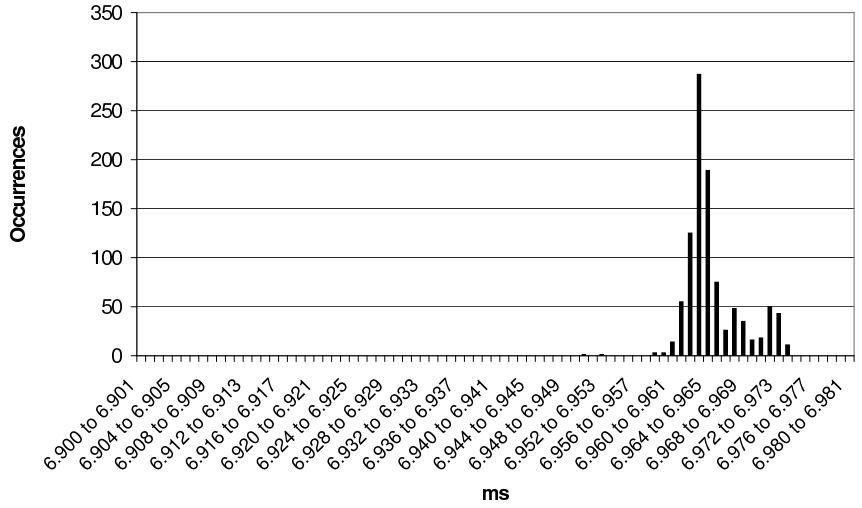


Figure 5.25: $Rel(t_{TM\_second})$ histogram (relative delay)

For the set of measures presented in figure 5.25, the average delay is $6,965\mu s$ with a standard deviation of $3\mu s$. The minimum value obtained was $6,950\mu s$, while the maximum value obtained is $6,974\mu s$. Thus, the difference between the maximum and the minimum is $24\mu s$.

Note that the difference of the average relative delays, $Rel(t_{TM\_first})$ and $Rel(t_{TM\_second})$ is $38\mu s$ (with very similar standard deviation). As stated before, this difference results from the delayed transmission of the TM ($t_1 = 28\mu s$) plus the delays associated with stuffing bits.

Also note that, all the values presented where are the values measured by the xDMS. In fact they are affected by practical issues that will be explained in next section.

5.3.4 Practical issues

The experimental results presented in previous section are similar with the theoretical results presented in chapter 3 with some minor differences arising from some practical issues not considered in the theoretical analysis. In this section these practical issues are presented. Also, the obtained results are compared with the theoretical analysis.

All experimental results (delays) are measured with the timers of the xDMS device, which does not starts and stops at the exact moment of the event occurrence due to the overheads resulting from the interrupt processing and event recognition. In a similar work regarding this issue [FBSC07], a detailed study of the overheads in the DMS is presented when it is

used for measuring delays in a Ethernet token passing system.

In this section the measure delays performed by the xDMS are compared with the corrected delays and, these last ones with the expected delays. For this, three definitions are needed:

- Observed delays are the measured delays presented in previous section. These delays measured by the xDMS are affected by practical issues not considered before;
- The corrected delays are the delay measured by the xDMS but applying the practical issues;
- The Theoretical delays (or expected delays). Are the delays expected by the theoretical analysis.

Figure 5.26 represents the overheads in the xDMS for the case of measuring the t_{BEAM} and $t_{re\_scheduling}$ delay, without taking into account the bit stuffing.

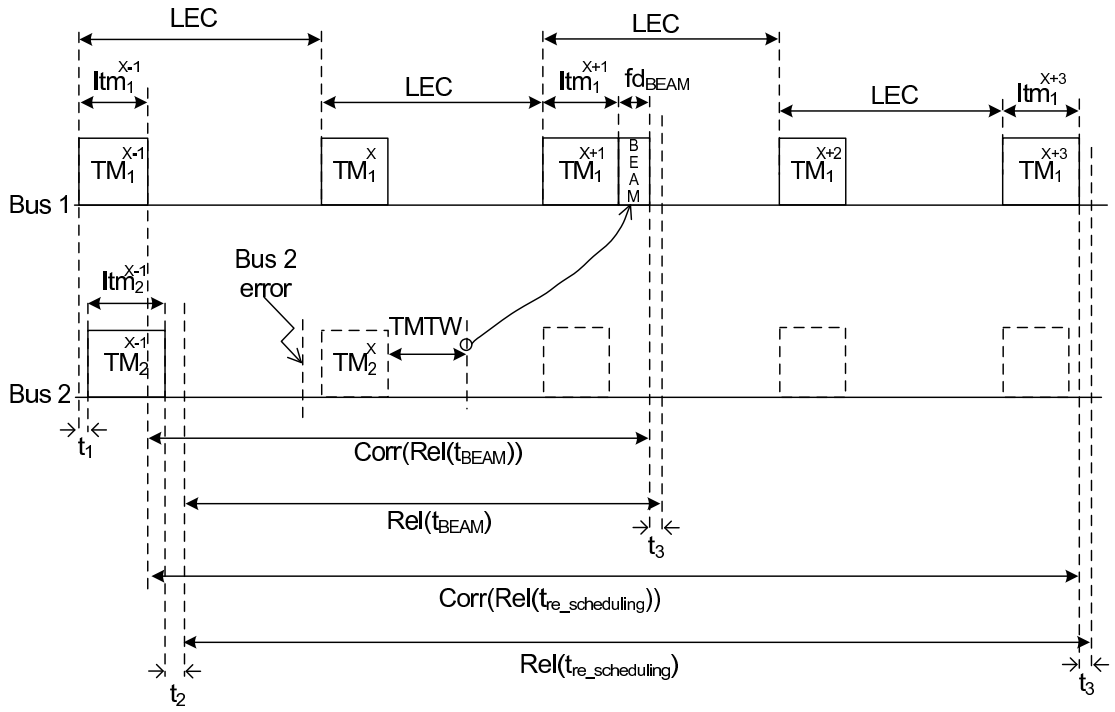


Figure 5.26: Practical analysis for bus error timeline

For the case of the master error, the practical overheads are presented in figure 5.27.

Figures 5.26 and 5.27 assume all the timings explained in figure 5.7 and in 5.8 respectively. Additionally:

- t_1 represents the time between the sending of trigger messages in different buses: t_1=28μs (this value has been measured with an oscilloscope). This value is also valid for figure 5.27 and is the same as in section 5.3.3;

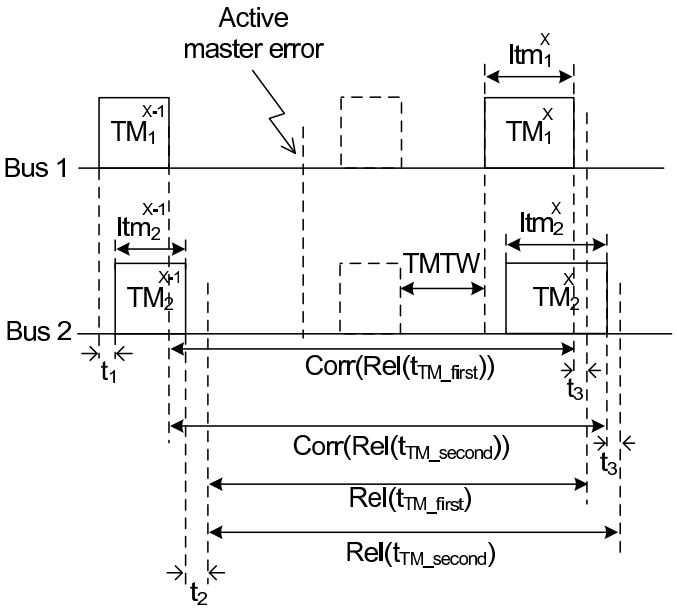


Figure 5.27: Practical analysis for master error timeline

- t_2 represents the time to the processing of an interrupt reception of the trigger message in the xDMS, and prepare the fault injection: $t_2 = 89\mu s$. This value has been measured with an oscilloscope. This value is also valid for figure 5.27 and is the same as in section 5.3.3;

- t_3 represents the time to process the interrupt of the event reception: $t_3 = 6\mu s$ (this value has been measured with an oscilloscope). In case of bus error (figure 5.26) is the time to process the BEAM message or the TM_1^{X+3} trigger message. In case of master error (figure 5.27) is the time to process TM_1^X or TM_2^X;

- As an example, the delay $Rel(t_{BEAM})$ is the delay measured by the xDMS, however the corrected delay for $Rel(t_{BEAM})$ is $Corr(Rel(t_{BEAM}))$. For all delays of bus and master errors timeline there are the measure delay and the corrected delay.

As it can been seen from figures 5.26 and 5.27 all the delays are affected in the same manner. This is, all of them are affected by the t_1, t_2 and t_3. Thus:

$$Corr(F) = F - t_3 + t_1 + t_2 \tag{5.2}$$

Tables 5.8 and 5.9 (for bus error and master error respectively) compare the delays observed and presented in previous section with the corrected delays according to equation 5.2 and with the expected theoretical delays.

Comparing expected and practical delays

In tables 5.8 and 5.9, the observed delays with the xDMS and the corrected delays are presented. The corrected delays can now be compared with the expected theoretical ones obtained in chapter 3. In chapter 3 only absolute delays are presented, however, the relative delays can be extrapolated from the absolute ones. The maximum and minimum values result from a combination of bit stuffing where some reference messages have the maximum bit stuffing, while others have zero bit stuffing. As an example, for the case of $Rel(t_{BEAM})$, the maximum occurs when $\mathrm{TM}_1^{\mathrm{X}+1}$ and BEAM has the maximum stuff bits and $\mathrm{TM}_1^{\mathrm{X}-1}$ has zero stuff bits. In that way, the maximum and minimum values for the four studied delays are presented next. For the case of $Rel(t_{BEAM})$, the theoretical maximum is:

$$\max(Rel(t_{BEAM})) = 2 \times LEC + \max(ltm_y^x) - \min(ltm_y^x) + \max(fd_{BEAM}) \tag{5.3}$$

And, the minimum value for $Rel(t_{BEAM})$ is:

$$\min(Rel(t_{BEAM})) = 2 \times LEC + \min(ltm_y^x) - \max(ltm_y^x) + \min(fd_{BEAM}) \tag{5.4}$$

For the $Rel(t_{re\_scheduling})$ the maximum and minimum values expected are (using equation 3.39 with $K = 1$):

$$\max(Rel(t_{re\_scheduling})) = 4 \times LEC + \max(ltm_y^x) - \min(ltm_y^x) \tag{5.5}$$

$$\min(Rel(t_{re\_scheduling})) = 4 \times LEC + \min(ltm_y^x) - \max(ltm_y^x) \tag{5.6}$$

For the case of the master error, from a theoretical point of view, only t_{TM_new} is specified (see equation 3.31). Thus, for $Rel(t_{TM\_first})$ and $Rel(t_{TM\_first})$ the maximum and minimum values expected are:

$$\begin{aligned}\max(Rel(t_{TM\_second})) = \max(Rel(t_{TM\_first})) = \\ LEC + TMTW + 2 \times \max(ltm_y^x) - \min(ltm_y^x)\end{aligned} \tag{5.7}$$

$$\begin{aligned}\min(Rel(t_{TM\_second})) = \min(Rel(t_{TM\_first})) = \\ LEC + TMTW + 2 \times \min(ltm_y^x) - \max(ltm_y^x)\end{aligned} \tag{5.8}$$

From equation 5.3 to equation 5.8:

- y is the number of the trigger message and $y \in \mathbb{N}$;
- x is the number of the bus, for the present case: $x \in \{1, 2\}$.

In tables 5.8 and 5.9, compare the observed delays (measured by the xDMS) with the corrected delays and with the expected theoretical delays. For the computation of the expected theoretical delays, the used maximums and minimums for the trigger message and BEAM message are (assuming the parameters of the test platform):

- $\min(ltm_y^x) = 444\mu s$, that corresponds to a trigger message with 8 data bytes with no stuff bits;
- $\max(ltm_y^x) = 528\mu s$, that corresponds to a trigger message with 8 data bytes maximum bit stuffing;
- $\min(fd_{BEAM}) = 252\mu s$, that corresponds to the BEAM message with no bit stuffing (refer to figure 3.13);
- $\max(fd_{BEAM}) = 300\mu s$, that corresponds to the BEAM message with the maximum of bit stuffing (refer to figure 3.13).

	$Rel(t_{BEAM})$			$Rel(t_{re\_scheduling})$		
delay (μs) / Value	Observed	$Corr()$	Expected	Observed	$Corr()$	Expected
Average	10,182	10,293	10,276	19,910	20,021	20,000
Minimum	10,173	10,284	10,168	19,820	19,931	19,916
Maximum	10,190	10,301	10,384	19,920	20,031	20,084

Table 5.8: Observed, corrected and theoretical delays for bus error

	$Rel(t_{TM\_first})$			$Rel(t_{TM\_second})$		
delay (μs) / Value	Observed	$Corr()$	Expected	Observed	$Corr()$	Expected
Average	6,927	7,038	7,000	6,965	7,076	7,000
Minimum	6,912	7,023	6,874	6,950	7,061	6,874
Maximum	6,935	7,046	7,126	6,974	7,085	7,126

Table 5.9: Observed, corrected and theoretical delays for bus master error

In tables 5.8 and 5.9, the average of the expected delays is an arithmetic average between the respective maximum and the minimum.

The values presented in table 5.8 and 5.9 demonstrate the correctness of the analysis. All the corrected delays are within the maximum and minimum expected theoretical values. As

an example, the theoretical (expected) minimum and maximum values for the $Rel(t_{BEAM})$ are $10,168\mu s$ and $10,384\mu s$, respectively. The corrected maximum and minimum values (corrected from the measured) are $10,284\mu s$ and $10,301\mu s$, meaning the system behaves as expected.

The corrected average values (derived from the measured) are very close to the expected ones, validating the assessment made.

5.4 Results summary

For the case of the single bus assessment, the use of FTT-CAN in the low-level control system of a soccer robot permits a synchronization among tasks, reducing the jitter associated with the end-to-end delay of the information flows. More particularly, the jitter of the end-to-end delay of the motion flow reduces from $25.6ms$ to $1ms$ while the jitter for the end-to-end delay of the odometry flow reduces from $9ms$ to $100\mu s$. Adding to this, the system is able to replace the master node in a very narrow time interval, $310\mu s$. During the tests, this value is unnoticeable for the robot motion and behaviour. Besides the timeliness results, one could refer a subjective practical evaluation, resulting from the use of the robots where the system is running playing several tournaments and demonstration games. In fact, the robot team has achieved the following results:

- 1^{st} place in RoboCup World Championship 2008;
- 2^{nd} place in RoboCup German Open 2010;
- 3^{rd} place in RoboCup World Championship 2009;
- 1^{st} place in Portuguese Robotic Open 2010, 2009, 2008 and 2007.

For the case of the multiple buses, the system was tested using a laboratory test bed. The tests are divided into bus error and active master error. The delays associated with the bus error were presented in figure 5.7 while the delays associated with the active master error were presented in figure 5.8. The obtained results for these delays are summarized in table 5.10. In table 5.10 only the relative delays (the corrected delays) are presented because they are easier to interpret due to the strong marks associated (recall that absolute delays are measured from the randomly instant of the error injection). Also, from the relative results is possible to obtain the absolute results.

As presented in table 5.10, for the bus error, the trigger message and the BEAM message are extremely precise. The standard deviation is $7\mu s$, corresponding to one bit and a half at the current bitrate. The difference between the maximum and minimum value obtained is $17\mu s$, corresponding to four bits. The message with the new scheduling is transmitted after an average of $20,021\mu s$ with a standard deviation of $4\mu s$. This means that the system

delay (μs) / Value	Bus error (dominant or cut off)		Active master error	
	$Rel(t_{BEAM})$	$Rel(t_{re\_scheduling})$	$Rel(t_{TM\_first})$	$Rel(t_{TM\_second})$
Average	10,293	20,021	7,038	7,076
Std. deviation	7	4	3	3
Maximum	10,301	20,031	7,046	7,085
Minimum	10,284	19,931	7,023	7,061
Max-Min	17	100	23	24

Table 5.10: Results summary

recovery from a bus fault is performed after 4 elementary cycles ($5ms$ each elementary cycle). This value can be considered a very low value and, we foresee that it will be unnoticeable for most applications. The standard deviation for this delay is $4\mu s$, which corresponds to a bit time. These values prove that the system reacts to a bus failure in a very timely and predictable way.

In what concerns the active master error, the values obtained for $Rel(t_{TM\_first})$ and for $Rel(t_{TM\_second})$ are very similar, but $Rel(t_{TM\_second})$ is delayed relatively to $Rel(t_{TM\_first})$ due to the difference of the triggering of messages in different buses by the master and due to the stuff bits. The standard deviation obtained is very low (less than a bit time), and similar for the two trigger messages send in the two different buses. Note also that the difference between the maximum and minimum is also similar for $Rel(t_{TM\_first})$ and $Rel(t_{TM\_secod})$ and is of the order of four bit times. In the presented tests, for the case of master replacement, the system will be without a master for 40% of an elementary cycle ($2ms$). The values obtained (namely the standard deviation) mean that the system reacts in a very predictable way, resulting in the same behaviour in every trial of the batch test (recall the batch test is 1000 trials). Moreover, the time the system will be without a master node is very low, making the active master replacement almost or practically unnoticeable for the system.

The values obtained seem to be very promising, because the worst recovery time, which happens when a bus fault occur (partition or bus dominant bits), is less than $20ms$. Comparing this value with other recovery intervals found in the literature, *e.g.* CAN Kingdom, this result is several orders of magnitude better. In fact, considering CAN Kingdom, authors claim in [SGN02] that "*The time for recovery is estimated to be at maximum 1s*".

5.5 Conclusions

This chapter has presented the experimental validation of FTT-CAN with single and multiple buses.

The single bus FTT-CAN implementation was deployed in a robotic soccer team where it was used in the low-level control to coordinate the motion and the ball kicking. The single bus FTT-CAN architecture with redundant master nodes was previously validated using model checking, but it was not clear then if it was suitable for implementation in low power microcontrollers. This work has shown that the overhead of the protocol when implemented in microcontrollers is low and it can be used in real competitions with a timely and dependable behaviour. It was verified that the addition of the FTT-CAN protocol to the control of the robots contributed to the synchronization of all the data streams and tasks running in the nodes. Furthermore, the jitter associated with the communication flows has been reduced from $25.6ms$ and $9ms$ to $1ms$ and $0.1ms$ (for motion and odometry respectively).

In what concerns the multiple buses experimental results, a preliminary validation was presented using fault injection and an error measurement equipment. The extended delay measurement system (xDMS), that incorporates a fault injector and a delay measurement system, was developed for that purpose and showed to be useful for other research projects. The fault injector is able to inject faults in the CAN buses and in the master nodes, according to the considered fault hypothesis.

Experimental results have shown that the FTT-CAN architecture with multiple masters and multiple buses is feasible and it can be implemented in a real applications using low processing power processors.

The obtained delays for the multiple buses architecture are very narrow with very low standard deviation, indicating the system behaves has expected in a timeliness and predicable way.

Chapter 6

Using multiple buses in native CAN - A generalization

6.1 Introduction

The previous chapters have addressed bus media redundancy for FTT-CAN based systems both to improve their dependability and to increase the available bandwidth.

On the scope of this work it was realized that a solution could also be envisaged for legacy CAN systems. This chapter contains the description of the solution, some preliminary steps concerning its validation and some proposals for the implementation of the system elements used in the solution. With the proposed solution it is possible to improve the dependability and bandwidth of legacy CAN networks, without modifying existing nodes' hardware or software.

The chapter starts, with a short comparison between star and bus technologies. This comparison is required due to the recent and, in our opinion, unsustained interest in star topologies. Indeed, it is easy to see that these are a particular case of bus topologies, with many drawbacks and with a unsatisfactory use of cabling. In fact, alternatives among the pure star and the pure bus as discussed in this work seem much more promising than pure star topologies.

The system components and behaviour have been proposed by the author while the preliminary implementation and validation have been developed by *Sher* [She09] and *Silva* [Sil09] under the author' supervision.

6.2 Comparing bus and stars architecture in terms of wiring

Star topologies are becoming increasingly popular in fieldbus networks due to their fault isolation capabilities. This trend partially contradicts one of the first arguments used back in last century' eighties: the reduced cabling that favoured distributed computer controlled

systems based in fieldbuses over their centralized counterparts. The cabling harness of a fully star based network, with only a node per branch and with the nodes distributed over a circle, is comparable with the one of a centralized system and it is proportional to the number of nodes, while the cabling of the corresponding bus based network is constant. If one considers the opposite scenario, *i.e.*, nodes equally spaced over an imaginary line, the cabling required for a solution based in a central star is also much higher than the one of a bus based topology. These scenarios are depicted in figure 6.1 and figure 6.2.

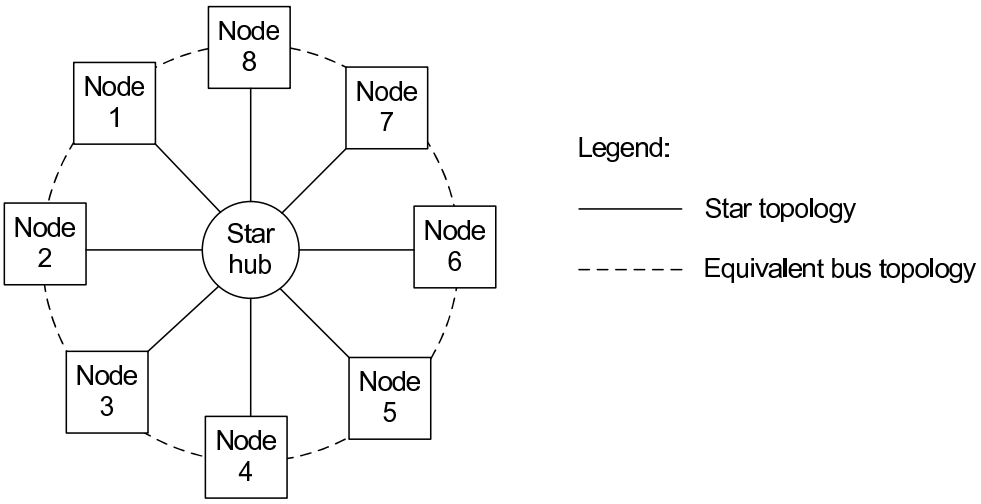


Figure 6.1: Star topology best scenario

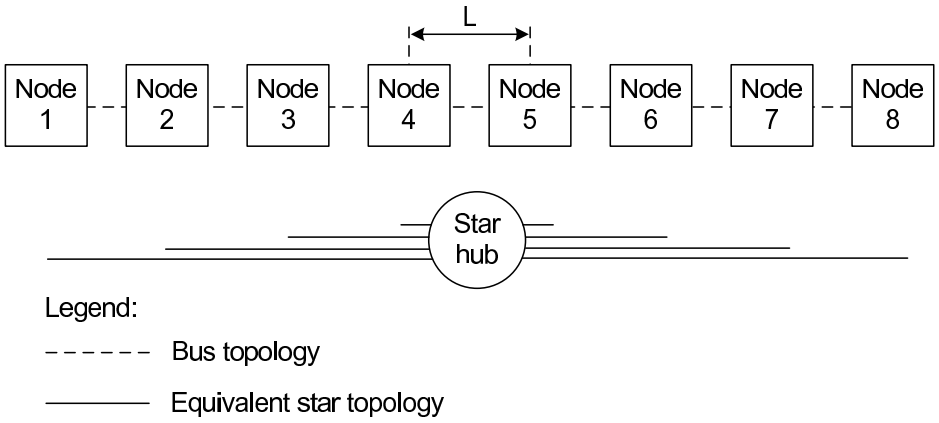


Figure 6.2: Bus topology best scenario
Top: Bus. Bottom: Star with the same spatial node distribution.

Figure 6.1 depicts the case where nodes are aligned over a circumference (the most favourable case for the star wiring length). Using a star interconnection network requires a total cable length of $N \times R$, where N is the number of nodes of the system and R is the distance from the central point. If replicated connections are used, the total cable length is $N \times R \times M$, where M is the number of connections from the central point to each node.

On the other hand, if the nodes are interconnected by a bus and aligned over a circumference, , the total cable needed is constant and equal to $2\pi RM$ where M is the number of buses to interconnect the nodes. In fact this is the total length for a ring topology. However, this value can be used as a worst case value.

Figure 6.2 depicts the case where nodes are equally spaced by an L distance and aligned over an imaginary line (the most favourable case for the bus wiring length). The total cable length for a bus interconnection network is $L \times M \times (N\text{-}1)$, while the total cable needed for a star interconnection network is given by:

$$\begin{cases} \{\sum_{i=1}^{\frac{N}{2}} L(2i-1)\} \times M & \text{if } N \text{ is even} \\ \{(\sum_{i=1}^{\lfloor \frac{N}{2} \rfloor} L(2i-1)) + \left\lfloor \frac{N}{2} \right\rfloor \times L + \frac{L}{2}\} \times M & \text{if } N \text{ is odd} \end{cases} \tag{6.1}$$

Knowing that $\left\lfloor \frac{N}{2} \right\rfloor = \frac{N-1}{2}$, the equation 6.1 can be simplified:

$$\begin{cases} L \times \frac{N^2}{4} \times M & \text{if } N \text{ is even} \\ L \times (\frac{N^2+1}{4}) \times M & \text{if } N \text{ is odd} \end{cases} \tag{6.2}$$

A comparison summary of the cable length (in m) is presented in table 6.1. This table compares a system where the nodes are aligned over a circumference with a system where the nodes are aligned over a line. There are values for 2 to 10 nodes with 1 to 3 communications channels. In table 6.1, R and L are assumed to be $1m$.

	Nodes in circumference						Nodes in line					
	Star			Bus			Star			Bus		
N \ M	1	2	3	1	2	3	1	2	3	1	2	3
2	2	4	6	6.28	12.56	18.85	1	2	3	1	2	3
3	3	6	9	6.28	12.56	18.85	2.5	5	7.5	2	4	6
4	4	8	12	6.28	12.56	18.85	4	8	12	3	6	9
5	5	10	15	6.28	12.56	18.85	6.5	13	19.5	4	8	12
6	6	12	18	6.28	12.56	18.85	9	18	27	5	10	15
7	7	14	21	6.28	12.56	18.85	12.5	25	37.5	6	12	18
8	8	16	24	6.28	12.56	18.85	16	32	48	7	14	21
9	9	18	27	6.28	12.56	18.85	20.5	41	61.5	8	16	24
10	10	20	30	6.28	12.56	18.85	25	50	75	9	18	27

Table 6.1: Cable length comparison

From this two opposite situations it becomes clear from table 6.1 that bus interconnection

networks require less cabling. This is true for the case depicted in figure 6.1 when the number of nodes is greater than 2π:

$$2\pi R \leq N \times R \Leftrightarrow N \geq 2\pi \tag{6.3}$$

Thus, N must be greater than 6 (as shown in table 6.1).

In the case of figure 6.2 when the number of nodes is greater than 2 (thus, in every situations) the bus interconnection requires less cabling. Note that, when two nodes are aligned over a circumference, in fact they are aligned in a line, and thus, the cable length is equal in both network topologies. This can be extrapolated to the case of more nodes and thus, the cable length presented in table 6.1 for a circumference alignment with bus interconnection can be reduced. However, in table 6.1 the worst case scenario is presented.

The reduced cabling required by a bus interconnection network becomes even more important when network replication is considered to provide additional fault tolerance capabilities.

6.3 Providing automatic bus redundancy in legacy CAN

Bus replication alone, in some cases, is not enough to enforce the full connectivity of the network. Consider, for example, the case presented in figure 6.3, where node 2 and 3 have only one communication interface. These two nodes are connected to bus 1. In case of this bus fails, node 2 and 3 cannot communicate anymore.

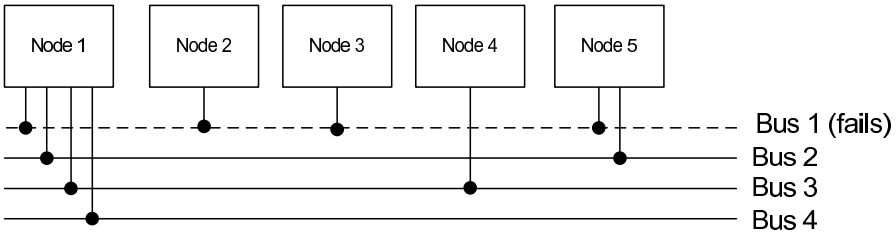


Figure 6.3: Replicated network with heterogeneous nodes

It would be desirable to have a CAN based network infrastructure capable of preserving the proved merits of CAN while providing fault confinement, via online network reconfiguration using replicated buses and additional bandwidth in a transparent way. The proposed architecture, depicted in figure 6.4, uses two components, the Network Switch Unit (NSU) and a Topology Management Unit (TMU) to handle these tasks.

The TMU is responsible for the definition of the topology used at each instant. It issues commands to the NSUs to indicate which bus should be used to connect the node. This can be seen as a Dynamic Redundancy Management (DRM) as the buses can be switched online upon failure detection [SFF06a]. Moreover, the DRM operation is transparent to the node. Additionally and whenever required, the NSU can act as a bus guardian [FAF+06]. Notice that the topology management unit needs to be replicated, since otherwise it would

be a single point of failure. This replication requires a protocol to enforce determinism and synchronism in case of failure of the active TMU.

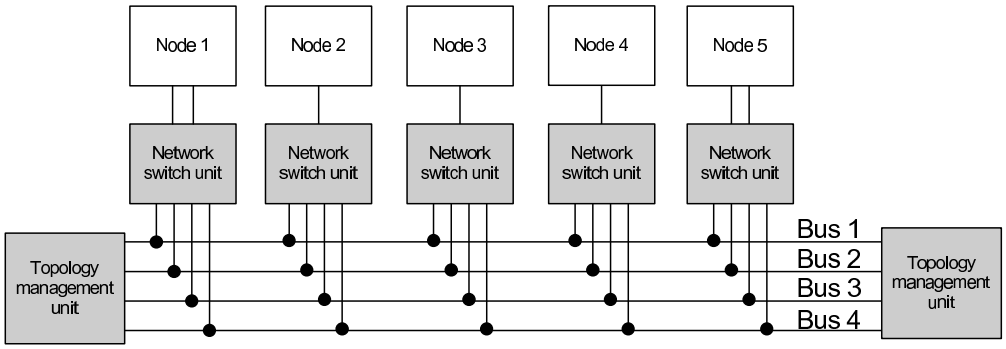


Figure 6.4: Proposed architecture

With this architecture it is possible to react to permanent bus faults by dynamically redefining system topology, replacing the faulty bus and keeping the system running, possibly with a degraded quality of service.

Notice that, in a limit situation, the proposed network architecture can be used for online change the network topology, for example, configuring a star network [SFF06a] (this topic will be discussed further in this chapter).

6.4 Fault hypothesis

A fault hypothesis specific for the FTT-CAN network has been presented in section 3.5.2. The proposals presented in this chapter are targeted to native CAN, thus the fault hypothesis is adapted accordingly:

- Node faults - the node and the corresponding network switch unit are a fault tolerant unit (FTU). This FTU is fail-silent both in time and value domains. The topology manager unit is also considered to be fail silent both in time and value domain. To enforce of this behaviour some of the techniques adopted in [FAF+06] could be used;

- Channel transient faults - only transient faults that change the value of, at least, one bit are considered;

- Message atomicity - when a message is transmitted in parallel in several replicated buses and received by nodes with at least two bus interfaces, its transmission is considered atomic. This assumption is based in experimental data of CAN bit error rate [FOFF04] showing that CAN bit error rate in an aggressive environment can be as low as 2.6×10^{-7} with a corresponding inconsistent message omission rate below 10^{-9} per hour;

- Channel permanent faults - the transmission medium is a single point of failure of CAN and permanent faults of the transmission medium, such as bus partition or stuck-at faults could occur. In order to prevent the occurrence of inconsistent message omissions, the bus is also considered permanently faulty whenever the respective CAN controller located in the Topology Manager Unit reaches the error passive state;

- Bus partitions - bus partition detection takes advantage of the topology manager unit replication, with both replicas located at each end of the bus. Therefore, if the active replica fails, the system begins to operate in a degraded mode without bus partition detection capabilities;

- Is assumed that at least one bus is free of erros;

- There is only one topology management unit in the system at a time.

6.5 Network switch unit

The Network Switch Unit (NSU), connects the N interfaces of each node to the M available CAN buses as an $N \times M$ switch matrix that can be implemented in specialized hardware such as a FPGA (Field Programmable Gate Array). The internal architecture of the network switch unit is depicted in figure 6.5.

Notice that this architecture represents a fully connected NSU, however, a reduced version could be envisaged connecting each node to only a subset of the M available CAN buses.

The NSU is composed of a switch matrix, a switch controller and a switch table. The switch matrix is responsible for the physical connection between the CAN interfaces of the nodes and the CAN buses, implementing the connection rules stored in the switch table. This table is a flag matrix with M columns and N rows, where each column represents the connection of each CAN bus and each row represents the connection of each device bus.

The switch table can be changed dynamically so that the network can be adapted to evolving operational scenarios, *e.g.*, a bus failure or the need to provide additional bandwidth. The switch controller receives special identified messages sent by the topology management unit and changes the switching matrix accordingly. The switch controller has one CAN controller for each bus and is also responsible for replying to special messages issued by the topology management unit. These messages aim the verification of the status of the NSU and the status of the buses. The address and status register stores the address identification of the NSU and the flags to indicate its actual operation mode, such as receiving new table or normal operation.

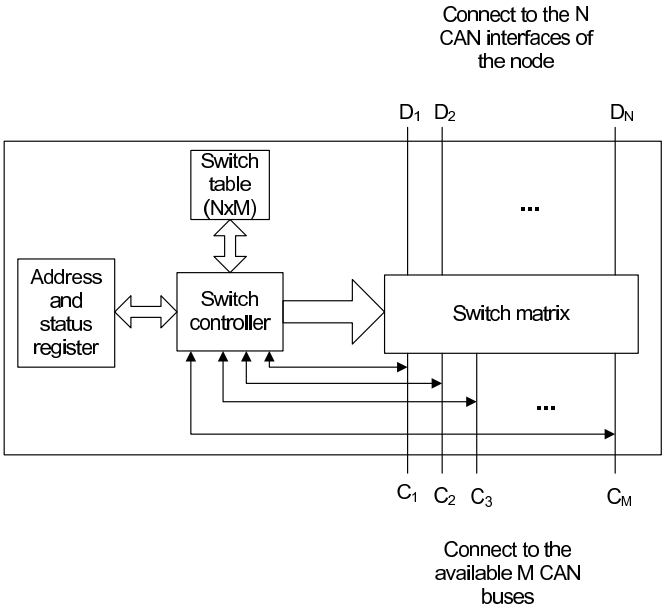


Figure 6.5: Network switch unit architecture

6.6 Topology management unit

The Topology Management Unit (TMU) is responsible for the dynamic online configuration of the network topology. It operates in a master-slave mode, where the network switch units (NSU) assume the slave role. Figure 6.6 depicts the architecture of the topology management unit (TMU). The TMU can be fully implemented in hardware, using an approach similar to the one adopted for the scheduling co-processor of the FTT-CAN master [MNF02] or using a processor with enough processing power.

The topology management unit is made of several modules which process the data stored in a set of tables. The Topology Reconfigurator (TR, see figure 6.5) is the most important module of the TMU as it takes the decisions concerning the topology management. The TR starts working upon information about bus failure coming from the Bus Fault Detector (BFD, see figure 6.5). This information can result from a spontaneous event, *e.g.* the detection of a stuck-at fault in one of the buses, or after a periodic verification of the bus status issued by the BFD, *e.g.* detecting a bus partition. As stated in the fault hypothesis, and in order to prevent the occurrence of inconsistent message omissions, the bus is also considered permanently faulty whenever the respective CAN controller located in the topology management unit reaches the error passive state. The TR reacts to a bus failure according to the procedures stored in the Reconfiguration Table (RT, see figure 6.5) that specify what should be done in case of a failure of a specific bus.

The RT includes a procedure for every possible bus failure situation. Considering the most simple operation scenario, any fault in the bus leads to a bus failure (more complex scenarios could eventually allow using part of the bus, *e.g.*, a bus partition). This means

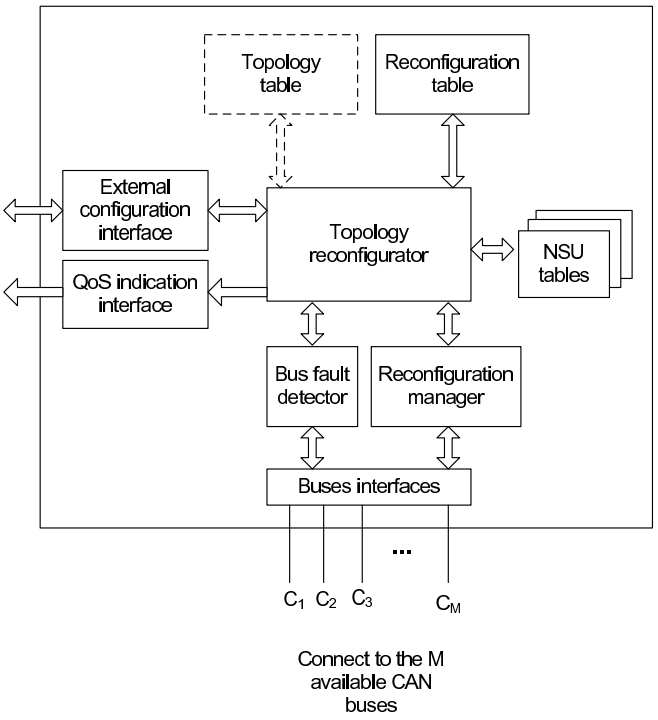


Figure 6.6: Topology management unit architecture

that the RT has an entry for each failure scenario that still enables the system to operate, even if in a degraded mode. As the fully operational situation is when all the buses are available and the total system failure is when all buses fail, then failure situations happen when 1, 2, ... M-1 buses fail. The number of bus failures and, in consequence, the number of RT entries, $DIM(RT)$, is then given by the possible combinations of those failures, that is:

$$DIM(RT) = \sum_{i=1}^{M-1} C_{M,i} = \sum_{i=1}^{M-1} \frac{M!}{i!(M-i)!} \tag{6.4}$$

It should be noticed that the system designer should consider that some of these combinations lead to situations in which the operation of the system is impossible. In those cases the RT must just contain an indication to stop the system. Besides these data, the RT also holds the location where the default configuration is stored so, in case of temporary faults, the system can recover back to the fully operational mode.

Although the topology management unit is autonomous in configuring the physical connections and, thus, there is no mandatory need of any other system element, it may be of interest, either for high level decisions or for statistical operation data recording, to have information about the system operation status. For this purpose, the TMU provides an interface, the QoS Indication Interface, which indicates the operation status by means of a code that can be related to the reconfiguration table in use.

It should be reinforced that this dynamic topology management could be enhanced by

integrating a complex scheduler into the TMU. However, this commutation between configurations ensures a simple and effective operation. As it was implicit in the simple example discussed above, the TR stores the actual state of the system in the NSU tables. Whenever these are changed, the TR activates the reconfiguration manager which sends the information of the new system topology to the NSUs. The NSU tables contain the current state of all the network switches present in the network. The NSU identification is also stored there. To enable external configuration, the TMU includes an external configuration interface used to program the reconfiguration table. The last component of the TMU is an optional topology table that contains information on the physical location of the NSUs along the buses to enable diagnosing possible bus partitions.

6.6.1 Topology management unit replication protocol

The topology management unit is a single point of failure that needs to be replicated in order to increase the dependability of the network. The approach adopted for this purpose is similar to the one used in the FTT-CAN master to enforce replica determinism and synchronization [FAF+06]. The TMUs follow a leader-follower behaviour where the leader issues periodic commands with the actual state of the buses (comparable to trigger messages in FTT-CAN) and the follower also tries to do it at the same time. If the leader successfully transmits the command message, the follower aborts its transmission, receiving the command message from the leader. If the leader fails to transmit the TMU synchronization message, the follower becomes the leader, upon successful TMU synchronization message transmission. TMUs could be internally duplicated, to enforce fail-silent behaviour, using a dual-processor CAN controller interface as described in [FAM+03]. An advantage of using a replicated TMU, at both ends of the network, is a simplified protocol to detect bus partitions using ping like commands only. In this way, the leader and the follower periodically exchange messages with their view of the buses state and look for possible mismatches indicating bus partitions or other fault scenarios. According to this protocol, the worst case bus partition detection time equals the period of the ping commands plus a blocking factor equal to the size of a maximum sized CAN message. Bus faults are always detected only by the leader. The follower sends its view to the leader but never decides that a particular bus is faulty. Upon bus fault detection, the leader reconfigures the network topology and the follower is implicitly notified through the reconfiguration commands issued by the leader. Notice that all commands are issued in parallel to all the buses, using high priority messages and, according to the fault hypothesis, these messages are assumed to be atomic.

6.7 Operational scenarios

Figure 6.7 presents an example of application of the proposed system. Node 1, node 4 and node 5 use more than one CAN bus to communicate, while the other nodes have just one CAN interface.

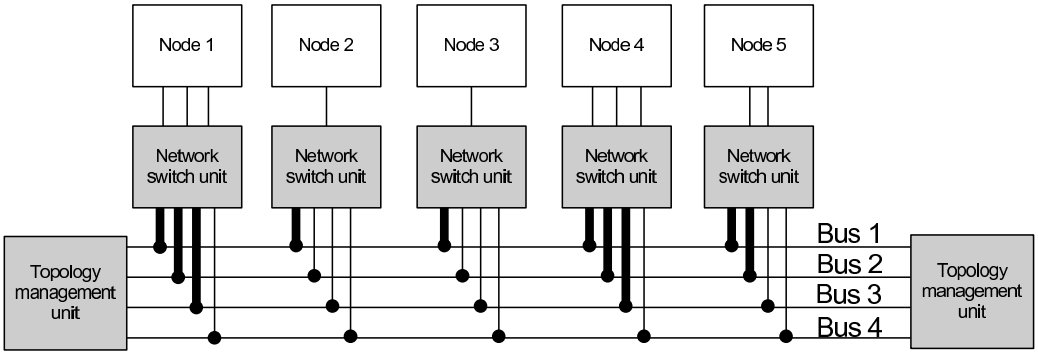


Figure 6.7: Architecture example

In figure 6.7 node 1 and node 4 communicate using three CAN buses. Note that the network itself has four buses. Thus, the additional bus is used for redundancy purposes.

In the example depicted in figure 6.7, the system designer could choose to use a default connection in which node 1 connects to buses 1, 2 and 3; node 5 connects to buses 1 and 2; node 2 and 3 connects to bus 1; and node 4 is connected to buses 1, 2 and 3. In figure 6.7 this default configuration is marked with ticker lines. This default configuration would lead to the NSU switch table presented in table 6.2.

Node switch	Switches	Bus in use
Node 1	1,1,1,0	Bus 1,2,3
Node 2	1,0,0,0	Bus 1
Node 3	1,0,0,0	Bus 1
Node 4	1,1,1,0	Bus 1,2,3
Node 5	1,0,1,0	Bus 1,3

Table 6.2: NSU table example

If the bus 1 becomes faulty, the topology manager unit detects it and reconfigures the bus in order to maintain the system in operation. In that case, the available bus will be used to enable the substitution of the bus 1. In that case, the network switch unit table will show the configuration presented in table 6.3. The NSU tabe is, in fact, a connection matrix where the row corresponds to the node and the column to the bus. As so, rebuilding the network transferring the connections from bus 1 to bus 4 corresponds to moving the data from column 1 to column 4.

The reconfiguration table (see figure 6.6) for this specific case has 14 reconfiguration scenarios:

Node switch	Switches	Bus in use
Node 1	0,1,1,1	Bus 2,3,4
Node 2	0,0,0,1	Bus 4
Node 3	0,0,0,1	Bus 4
Node 4	0,1,1,1	Bus 2,3,4
Node 5	0,0,1,1	Bus 3,4

Table 6.3: NSU table example after faulty bus

$$DIM(RT) = \sum_{i=1}^{M-1} C_{M,i} = \sum_{i=1}^{M-1} \frac{M!}{i!(M-i)!} = \frac{4!}{1 \times 3!} + \frac{4!}{1 \times 2} + \frac{4!}{1 \times 3!} = 14 \qquad (6.5)$$

Notice there are some specific scenarios that are not taken into account since they will generate an incorrect configuration, for example the use of only bus 1 in all node [SFF06a].

The proposed architecture may also be used to allocate a dedicated CAN bus linking two nodes. This feature is useful to provide permanent or temporary extra bandwidth to more demanding applications without interfering with other traffic. Other possible feature provided by the proposed architecture is the use of virtual paths, as depicted in figure 6.8, where node 1 communicates with node 4, node 2 communicates with node 3 and node 5 communicates with node 6 (thicker lines). Notice that this can be a temporary topology configuration. For example, in a normal behaviour this topology can use only one bus, and, due to a temporary bandwidth increase request for a particular traffic flow, the topology is modified to accommodate it. Moreover, using this strategy, dynamic stars can be implemented. To do that, the central node must have as many CAN interfaces as the number of nodes it needs to communicate to. This scenario is depicted in figure 6.9, where dynamic star paths are represented by thicker lines. Notice that, in examples of figures 6.8 and 6.9, bus 4 can be used for redundancy, thus, if some of the others fail, the traffic is re-routed for bus 4.

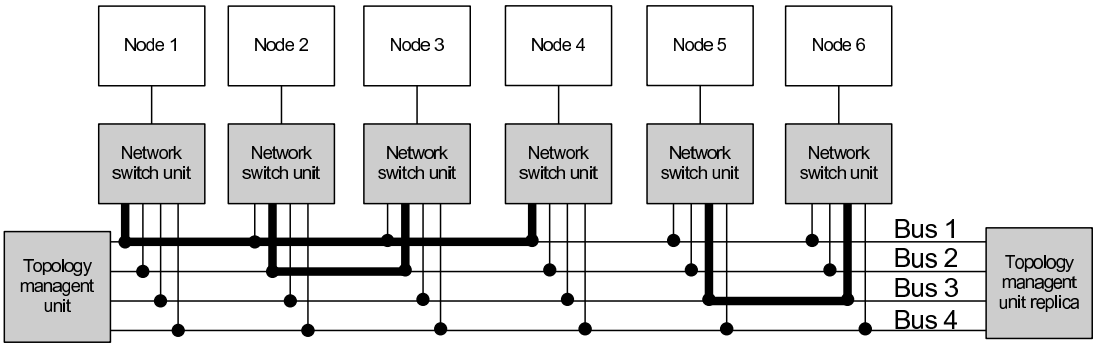


Figure 6.8: Operational scenario example

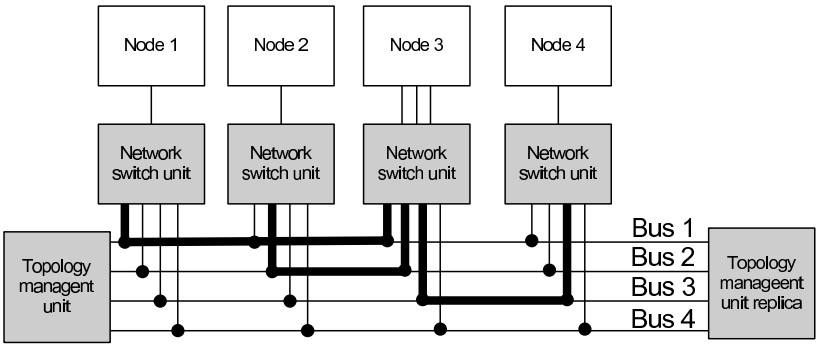


Figure 6.9: Operational scenario: star

6.8 Implementation

A prototype implementation of both the network switch unit and the topology management unit has been carried on [Sil09]. Both units were implemented in a FPGA (Field Programmable Gate Array) and the switch matrix was made using an analog switch. It was used a RC10 development board from Celoxica [Cel05] with a Xilinx Spartan-3 FPGA [Xil09] where the CAN controllers were based on a IP core developed in our laboratory [Oli07].

Some design space exploration made during the design process concluded that a good approach would be hardware/software co-design. *Silva* [Sil09] defends that the TMU is difficult to implement in hardware and thus, he has implemented it partially in hardware on the FPGA and partially in software using the PicoBlaze from Xilinx [Xil10]. PicoBlaze is a 8 bit microprocessor implemented in the FPGA.

Figure 6.10 presents the internal architecture of the NSU module that controls the switch matrix (see figure 6.5).

Sher [She09], has made a detailed study on the topology management unit replication protocol and has proposed a new solution based on the cyclic redundancy check code (called Distributable Table Content Consistency Checker - DT3C) to maintain the consistency among the topology management units. *Sher* defends that his network switch unit replication protocol is more efficient to maintain the consistency because network switch units only exchange the information about the CRC, and just the parts of the TMU table that are inconsistent are transmitted in the buses.

The presented implementations have been carried out in the scope of Master thesis and are merely exploratory. However they already enable us to verify that the FPGA based implementation is feasible.

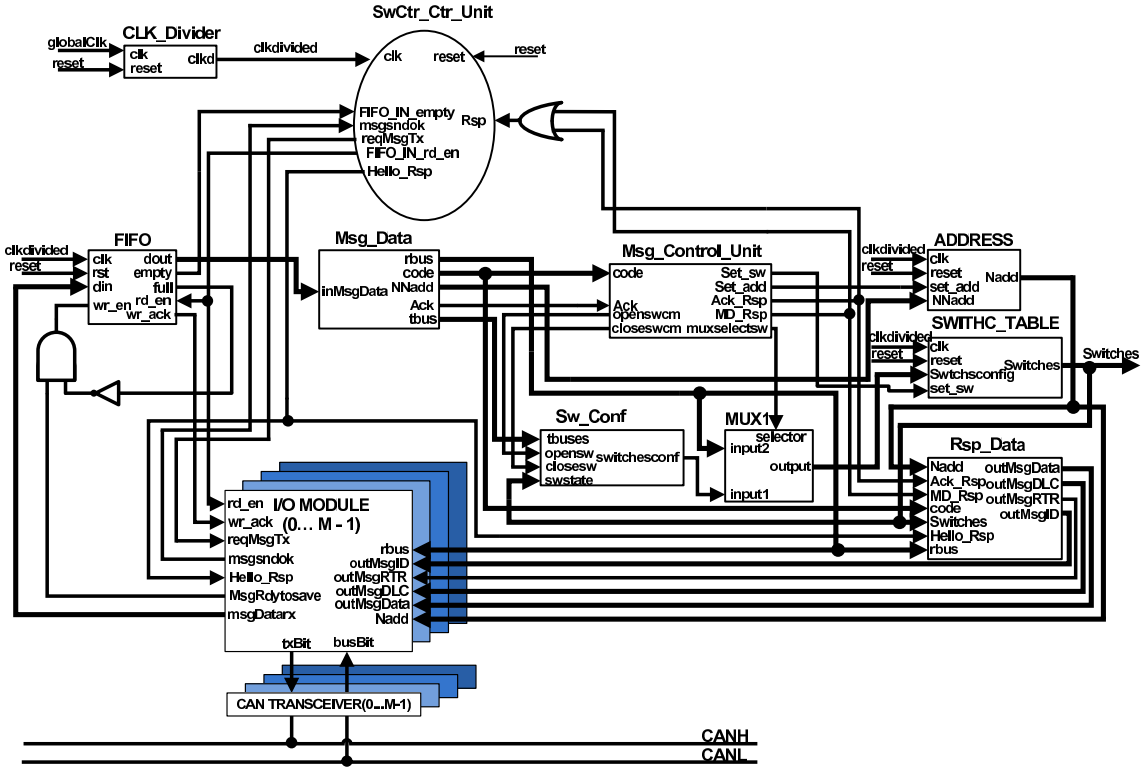


Figure 6.10: Switch controller (from [Sil09])

6.9 Conclusions

In this chapter a generalization to support redundancy and bandwidth improvement in native CAN has been presented. The proposed solution uses two additional components: the network topology management and the network switch unit.

The network switch unit connects the node to the CAN network using a switch matrix while the network topology management unit has a global view of the network and controls all the network switch units. The topology management unit is replicated and follows the same strategy already adopted by the FTT-CAN master node, *i.e.*, one network topology management unit is located at one of the ends of the CAN bus and exchanges messages with the other located at the other end.

A preliminary hardware implementation of the network switch, based on a FPGA, was assessed together with a first draft implementation of the topology management unit using a hardware/software co-design approach. Preliminary results indicate that the network switch unit can be implemented in the FPGA and the topology management unit can be implemented in hardware and software using a PicoBlaze microprocessor.

Chapter 7

Conclusions and future work

The central proposition of the thesis supported by the present dissertation, claims that the use of a flexible bus replication scheme could improve both the dependability and the throughput of a network. Furthermore, it is possible to adapt online the network topology to evolving operational scenarios.

The Flexible Time-Trigger (FTT) paradigm over the CAN network (FTT-CAN) was used, as a case study, to prove this claim. FTT-CAN is a protocol that combines the predictability of time-triggered systems, favouring the design of fault tolerance mechanisms with the flexibility of CAN, increasing the adaptation to evolving conditions. For the case of native CAN, there is also a proposal for using redundancy at the bus level leading to a flexible bus topology with redundancy and bandwidth management. All these three properties can be joined together making a flexible CAN network with possible online modifications of the topology.

The FTT-CAN protocol has been designed to provide flexibility, timeliness and efficiency for supporting event-triggered and time-triggered message transmission. Opposite to other protocols, such as TTCAN and FlexRay, FTT-CAN allows time-triggered messages to be scheduled online and dynamically, supporting modification of the message flows while the system is running. On the one hand, protocols such as FlexRay support the use of replicated bus both to improve fault tolerance and to increase the available bandwidth. Notice, however, that FlexRay cannot use more than one replicated bus and it does not have the flexibility of FTT-CAN. On the other hand, FTT-CAN has limited fault tolerance capabilities. Recent work has proposed the replication of the master node and has defined a scheme to maintain consistency among masters. This scheme is based on a leader-follower behaviour, where the active master is the leader and the followers are the backup masters.

In what concerns the bus replication, some fieldbuses or fieldbus-based adapted solutions address this issue, *e.g.*, FlexRay and CAN Columbus Egg Idea. However, and for these two particular cases, just one bus replica can be used. Other protocols have other limitations, *e.g.*, only replicated messages are allowed in the replicated buses. To cope with these limitations,

this work proposes the use of multiple buses, both to increase the bandwidth and to improve the dependability, with no limits on the number of redundant buses.

The main function of the FTT-CAN master node is to schedule the synchronous messages and to dispatch the trigger message to the buses. For the case of multiple buses, a trigger message for each elementary cycle is transmitted in each bus, thus the message scheduler must work in both time and spatial (bus) dimensions.

The single bus master replication protocol was adapted in order to support multiple buses. The basic idea is similar to the single bus version, but replicated masters rely on the trigger message transmission window ($TMTW$) to replace the active master if no trigger message is received within the $TMTW$. The master error detection and replacement is slower than for the single bus version. More specifically, in the multiple buses case, this procedure takes the $TMTW$ window length, while for the single bus version it takes half of a trigger message duration. Other difference to the single bus version is the location of the replicated masters, that are now placed at the end of the buses. The physical location of the replicated masters is crucial both to the bus error detection algorithm and to the master replication protocol. If no trigger message is received from all the buses, this means, according to the fault hypothesis, that there is an active master failure. If there is a missing of one (or a set) of trigger message, then the bus where the trigger message is missing is considered faulty.

All the protocols and components proposed in this document were implemented and experimentally validated. The validation process was carried out in two phases: for the single bus version and for the multiple buses version. For the single bus version, the validation was made on soccer robotic platforms during many soccer games. The multiple buses version was validated using fault injection and an experimental setup including two master nodes with two CAN buses each, a fault injector and a delay measurement system. The fault injector and the delay measurement system are located in the same hardware platform, the extended delay measurement system. The fault injector is able to impose faults in the buses (stuck-at and partition) and also at the master nodes (resetting them). Experimental results have shown that the system behaves as expected, with measured average delays according to the theoretical expectations and with very narrow standard deviation.

Partial results of this thesis were used in other contexts, notably the delay measurement system that has performed the assessment of real-time information flows of an Ethernet based network.

This thesis also proposes a generalization of its claims to native CAN networks. The solution relies on multiple CAN buses where the nodes can connect to one, to a set of all buses or to all buses. This system needs two additional components, the topology management unit and the network switch unit. A preliminary work concerning the development of the required components has shown that it is possible to implement them with current technology and that they are not too complex.

Summarizing, it could be said that the proposals presented on this thesis contributed to the increase of dependability and bandwidth of networks in a flexible and timely way.

7.1 Thesis validation

The thesis stated in chapter 1, arguing that it is possible to improve both the dependability and the throughput of a network using a flexible bus replication scheme. This solution was validated for the specific case of FTT-CAN and for native CAN. Furthermore, for the case of native CAN, it was demonstrated that it is possible to change online the network topology.

In fact, it has been shown, mainly with the work presented in chapters 3, 4 and 5, that it is possible to use, in a flexible way, more than one bus to improve the dependability and the available bandwidth of the system. Moreover, in chapter 6, it has been shown a way to flexibly change the topology of the network without compromising the overall performance of a legacy CAN network.

7.2 Future research

The work conducted for this thesis unveiled some interesting research ideas that indicate future research lines, which are are summarized and briefly discussed next.

7.2.1 The asynchronous messaging system

FTT-CAN has an asynchronous messaging system where the messages are sent by the slaves without the coordination of the master node. In this way, the asynchronous messaging system is equivalent to legacy CAN with inaccessibility periods corresponding to the trigger message and to the synchronous window. It is possible to bound the worst case transmission time of an asynchronous message for the case of the single bus version [AFF99]. However, the addition of bus media redundancy introduces some new problems arising from the possibility of having different asynchronous traffic in each replicated bus.

The main foreseen problem is the possible priority inversion whenever a specific bus has more asynchronous traffic than the others. A message with lower priority can be transmitted first in a bus with less load than a higher priority message transmitted in the overloaded bus. Note that, this problem only arises if the set of buses is seen as one (as it is viewed by the application layer).

Possible solutions to this problem include the allocation of the asynchronous traffic to just one bus.

7.2.2 Slave nodes

This thesis has not addressed the necessary changes of the slaves to accommodate more than one bus. In fact, slave nodes for the single bus FTT-CAN can be used without modification in the multiple buses version, but they are limited to just one bus.

The synchronous messaging subsystem for the multiple buses slave nodes is mostly a replication of the existent synchronous message subsystem defined in section 4.4. It is necessary to add extra interfaces to the slave nodes and to adapt their synchronous and asynchronous messaging subsystems.

So, this future line of work is more targeted to fully demonstrate the operation than to a challenging research topic.

7.2.3 Dependability evaluation

The dependability evaluation is associated with the development of a model that describes the system behaviour in presence of faults and their impact on the dependability.

The proposals presented on this thesis increase the wiring and the hardware/software complexity. Although the increased complexity and the extra hardware components were part of the fault tolerance mechanisms, it is not clear the real impact on the overall system dependability. The idea is to assess the dependability of the single bus single master FTT-CAN network and compare it with the full featured multiple buses multiple masters FTT-CAN implementation.

There are several methodologies and tools that can be adopted to investigate this issue, notably, Stochastic Preti Nets that have become a widely used framework to perform dependability evaluation of fieldbuses [PC01, MDM07, KLMB08].

7.2.4 Generalization to other protocols

The work presented in this dissertation is mainly intended for CAN. In fact most of the work is dedicated to FTT-CAN as it was the target protocol of the case study. However, the proposals can be generalized to other fieldbuses. At a first view, the FTT-CAN redundancy system presented in this work cannot be generalized as is, since it requires a master-slave scheme.

On the other hand, one could foresee generic topology management unit and generic network switch units with a common core and specialized interfaces for each fieldbus.

7.2.5 Wireless and heterogeneous networks

The presented proposal is intend to wired networks. Currently, however, wireless networks are widely adopted in many application domains and in millions of consumer electronics systems. Thus, migrating some of the proposed solutions to the wireless world is a natural

evolution path. It seems reasonable to define a equivalent system using wireless communications such as WiFi, Bluetooth or ZibBee. Instead of using multiple buses, multiple channels can be used in the same way as the multiple buses, with the necessary adaptions.

Also, it is possible to apply most of the proposals of this thesis to hybrid networks, *e.g.*, different channels can use diverse network media, for example one channel could use WiFi while other could use CAN. This research line requires an extensive work because of the differences among technologies.

References

[AAP06] Fernando Ataide, Alan Assis, and Carlos Pereira. Automotive X-by-Wire System Based on Linux - An Open Source Project. In *proceeding of the* 8^{th} *Real-Time Linux Workshop*, pages 61–67, China, October 2006. {58}

[ABB+09] Minoru Asada, Tucker Balch, Andrea Bonarini, Ansgar Bredenfeld, Steffen Gutmann, Berhard Kraetzschmar, Pedro Lima, Emanuele Menegatti, Pieter Jonker, Alireza Fadaei Tehrani, Takayuki Nakamura, Gerald Steinbauer, Martin Lauer, Yasunori Takemura, Huimin Lu, Enrico Pagello, Fernando Ribeiro, Thorsten Schmitt, Wei-Min Shen, Hans Sprong, Shoji Suzuki, Yasutake Takahashi, Paul G. Ploeger, Frank Schreiber, Jurge van Eijck, Akihiro Matsumoto, Saeed Shiry Ghidary, Roel Merry, Bernardo Cunha, Darwin Lau, Saeed Ebrahimijam, and Oliver Zweigle. Middle size robot league rules and regulations for 2010. Available: `robocupmsl.googlegroups.com/web/msl-rules-2009-18-12.pdf`. Last accessed 20 March 2010, 2009. {123}

[AC98] J. Azevedo and N. Cravoisy. *WorldFIP protocol*, 1998. {v,18,19}

[AFF99] Luís Almeida, José Fonseca, and Pedro Fonseca. A Flexible Time-Triggered Communication System Based on the Controller Area Network: Experimental Results. In *proceedings of the* 3^{rd} *International Conference on Fieldbus Technology (FeT)*, Magdeburg, Germany, September 1999. {169}

[Alc95] Alcatel. *MTC-3054 CAN Interface Data Sheet*, December 1995. {34}

[ALR01] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. Technical Report 739, University of Newcastle upon Tyne, School of Computing Science, 2001. {v,10,11,12}

[ALR04] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004. {11}

[APF02] Luís Almeida, Paulo Pedreiras, and José Fonseca. The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics*, 49(6):1189–1201, December 2002. {2,3,28,53,54,65,69,89}

[APF+07] Luís Almeida, Paulo Pedreiras, Joaquim Ferreira, Mário Calha, José Fonseca, Ricardo Marau, Valter Silva, and Ernesto Martins. *Handbook of Real-Time and Embedded Systems*, chapter 19, pages 19–1 to 19–22. CRC Press, 2007. {53}

[APLA06] Fernando Ataide, Carlos Pereira, Walter Lages, and Alan Assis. On the design of an embedded FTT-CAN platform with improvement of its inherent jitter. In *proceedings of the* 4^{th} *International IEEE Conference on Industrial Informatics (INDIN)*, Singapore, August 2006. {58}

[APS06] Fernando Ataide, Carlos Pereira, and Valter Silva. A New Approach for Time-Triggered Phase in the FTT-CAN Protocol: A Case Study in an Automotive System. In *Workshop on Models and Analysis Methods for Automotive Systems,* 27^{th} *IEEE Real-Time Systems Symposium*, Brasil, December 2006. {57}

[ASF+04] Luís Almeida, Frederico Santos, Tullio Facchinetti, Paulo Pedreiras, Valter Silva, and Luís Lopes. Coordinating distributed autonomous agents with a real-time database: The CAMBADA project. In *proceedings of the* 19^{th} *International Symposium on Computer and Information Sciences (ISCIS)*, 2004. {63,116}

[ATFV02] Luís Almeida, Eduardo Tovar, José Fonseca, and Francisco Vasques. Schedulability Analysis of Real-Time Traffic in WorldFIP Networks: an Integrated Approach. *IEEE Transactions on Industrial Electronics*, 49(5):1165–1174, October 2002. {18}

[Aum03] Auma. Actuator controls: Short instructions bus connections. Available: `www.auma.com/uploads/media/sp_import2/betriebsanleitungen/steuerungen/aumatic/ka_ac1_cdnet_en.pdf`. Last accessed 3 June 2010, 2003. {46}

[Aum10] Auma. Auma WebPage. Available: `www.auma.com`. Last accessed 3 June 2010, 2010. {46}

[AW94] Hagit Attiya and Jennifer Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994. {17}

[BAP05] Manuel Barranco, Luís Almeida, and Julián Proenza. ReCANcentrate: a replicated star topology for CAN networks. In *proceedings of the* 10^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 2, pages 8 pp.–476, September 2005. {47}

[Bar93] P.A. Barrett. Delta-4: an open architecture for dependable systems. In *proceedings of the IEE Colloquium on Safety Critical Distributed Systems*, pages 2/1–2/7, London, UK, October 1993. {13}

[BB03] I. Broster and A. Burns. An analysable bus-guardian for event-triggered communication. In *proceedings of the* 24^{th} *Real-time Systems Symposium*, pages 410–419, Cancun, Mexico, December 2003. IEEE. {75}

[BF10] Paulo Bartolomeu and José Fonseca. Channel Capture in Noisy Wireless Contention-Based Communication Environments. In *proceedings of the* 8^{th} *IEEE International Workshop on Factory Communication Systems (WFCS)*, Nancy, France, May 2010. {132}

[BFS+07] Paulo Bartolomeu, José Fonseca, Vasco Santos, Alexandre Mota, Valter Silva, and Margarida Sizenando. Automating Home Appliances For Elderly and Impaired People: The B-Live Approach. In *Software Development for Enhancing Accessibility and Fighting Info-exclusion*, Vila Real, Portugal, November 2007. {1}

[BHHP01] G. Bourhis, O. Horn, O. Habert, and A. Pruski. An autonomous vehicle for people with motor disabilities. *IEEE Robotics & Automation Magazine*, 8(1):20–28, March 2001. {1}

[Bir92] Kenneth Birman. Maintaining consistency in distributed systems. In *proceedings of the* 5^{th} *workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pages 1–6, France, 1992. ACM. {16}

[BKS03] Günther Bauer, Hermann Kopetz, and Wilfried Steiner. The central guardian approach to enforce fault isolation in the time-triggered architecture. In *proceedings of the* 6^{th} *International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 37–44, Pisa, Italy, April 2003. {23}

[BNMS05] Luís Bento, Urbano Nunes, Fernando Moita, and António Surrecio. Sensor fusion for precise autonomous vehicle navigation in outdoor semi-structured environments. In *proceedings of the* 8^{th} *International IEEE Conference on Intelligent Transportation Systems*, pages 245–250, Vienna, Austria, September 2005. IEEE Computer Society. {1}

[BOS91] Robert BOSCH. *CAN Specification Version 2.0.* Postfach 300240, D-7000 Stuttgart 30, 1991. {1,36}

[Bot00] Boterenbrood. CANopen: high-level protocol for CAN-bus. Technical report, NIKHEF, 2000. {43}

[BPA08] Manuel Barranco, Julián Proenza, and Luís Almeida. Management of Media Replication in ReCANcentrate. In *proceedings of the* 12^{th} *international CAN Conference*, pages 05.8 – 05–15, Barcelona, Spain, March 2008. {46}

[BPA09] Manuel Barranco, Julián Proenza, and Luís Almeida. Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate. *Computer*, 42(5):66–73, May 2009. {v,46,47}

[BPGN05] Héctor Benítez-Pérez and Fabián García-Nocetti. *Reconfigurable Distributed Control*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. {27}

[BPRNA06] Manuel Barranco, Julián Proenza, Guillermo Rodriguez-Navas, and Luís Almeida. An active star topology for improving fault confinement in CAN networks. *IEEE Transactions on Industrial Informatics*, 2:78–85, May 2006. {46,75}

[BSF07] Paulo Bartolomeu, Valter Silva, and José Fonseca. Delay measurement system for real-time serial data streams. In *proceedings of* 12^{th} *IEEE Conference Emerging Technologies and Factory Automation (ETFA)*, pages 516–523, Patras, Greece, September 2007. {4,131,132,134}

[Cal06] Mário João Barata Calha. *A Holistic Approach Towards Flexible Distributed Systems*. PhD thesis, University of Aveiro, 2006. {56,89,90}

[CAN96] CAN in Automation. CAL, CAN Application Layer for Industrial Applications, February 1996. CiA Draft Standard DS-201 to DS-207. {43}

[CAN99] CAN in Automation. CAN physical layer. Available: `http://www.can-cia.org/index.php?id=88`. Last accessed 8 June 8 2010, 1999. {31,34}

[CAN00] CAN in Automation. CANOpen application layer an communication profile, June 2000. {28}

[CAN07] CAN in Automation. CAN dictionary. Available: `www.can-cia.de/fileadmin/cia/pdfs/CANdictionary_v4.pdf`. Last accessed 31 August 2010, May 2007. {44}

[CAN09] CAN in Automation. Supplement of CAN newsletter. Page 6, June 2009. {28}

[Car08] Francisco Borges Carreiro. *Using the Ethernet Protocol for Real-Time Communications in Embedded Systems*. PhD thesis, University of Aveiro, 2008. {22,132}

[CASD85] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *proceedings of the* 15^{th} *International Symposium on Fault-Tolerant Computing (FTCS)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press. {17}

[CCTB03] Laurent Cauffriez, Blaise Conrard, Jean-Marc Thiriet, and Mireille Bayart. Fieldbuses and their influence on dependability. In *proceedings of the* 20^{th} *IEEE Instrumentation and Measurement Technology Conference (IMTC)*, volume 1, pages 83–88, VAIL, CO, USA, May 2003. {13,16}

[CDV01] Gianluca Cena, Luca Durante, and Adriano Valenzano. A new CAN-like field network based on a star topology. *Computer Standards & Interfaces*, 23(3):209–222, 2001. {46,47}

[Cel05] Celoxica. Platform Developer's Kit: RC10 Manual. Available: `teal.gmu.edu/courses/ECE448/documentation/RC10%20Manual.pdf`. Last accessed 7 June 2010, 2005. {164}

[CEN96] CENELEC. General Purpose Field Communication System - European Standard EN-50170, July 1996. {18,20,21,22}

[CF03] Mário Calha and José Fonseca. FTTlet based distributed system architecture. In *proceedings of the* 2^{nd} *International Workshop on Real-Time LANs in the Internet Age (RTLIA)*, Porto, Portugal, July 2003. {56}

[CF04] Mário Calha and José A. Fonseca. Approaches to the FTT-based scheduling of tasks and messages. In *proceedings of the* 5^{th} *IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 3–11, Vienna, Austria, 2004. IEEE Computer Society. {55,69}

[CFP03] Francisco Carreiro, José Fonseca, and Paulo Pedreiras. Virtual Token-Passing Ethernet - VTPE. In *proceedings of* 5^{th} *IFAC Conference on Fieldbus Technology (FeT)*, Aveiro, Portugal, July 2003. {22}

[CFSM06] Mário Calha, José Fonseca, Valter Silva, and Ricardo Marau. Kernel design for FTT-CAN systems. In *proceedings of the* 6^{th} *IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 151 – 156, Torino, Italy, June 2006. {56}

[CFSV04] Francisco Carreiro, José Fonseca, Valter Silva, and Francisco Vasques. The virtual token-passing Ethernet implementation and experimental results. In *proceedings of the* 3^{rd} *International Workshop on Real-Time Networks*, pages 57 – 60, Catania, Italy, June 2004. {132}

[Cis03] Cisco Systems, Inc. *Cisco EtherChannel Technology.* Cisco Systems, Inc, 2003. {29}

[CMTV02] Salvatore Cavalieri, Salvatore Monforte, Eduardo Tovar, and Francisco Vasques. Evaluating worst case response time in mono and multi-master profibus DP. In *procedings of the* 4^{th} *IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 233–240, Västerås, Sweden, 2002. {20,21}

[Com09] Comsoft. PRS-Profibus DP redundancy Switch. Available: `www.comsoft.de/download/icpe/product/prs_e.pdf`. Last accessed 8 June 2009, 2009. {21}

[Con09] Condor Engineering, Inc. Mil-std-1552 tutorial. Available: `microsat.sm.bmstu.ru/e-library/military`. Last accessed 9 July 2009, 2009. {27}

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications ACM*, 2:56–78, 1991. {12}

[CSF05] Mário Calha, Valter Silva, and José Fonseca. Real-Time Procedures in Distributed Systems. In *proceedings from IFAC International Conference on Fieldbus Systems and their Applications (FeT)*, pages 24 – 31, Mexico, November 2005. {69}

[Dec05] Jean-Dominique Decotignie. Ethernet-Based Real-Time and Industrial Communications. *Proceedings of the IEEE*, 93(6):1102–1117, June 2005. {17}

[DHLV96] Steve Dabecki, Barry Hagglund, Vern Little, and Iain Verigin. IEEE802.3z Gigabit Ethernet. Available: `grouper.ieee.org/groups/802/3/z/public/presentations/sep1996/BHgutp_b.pdf`. Last accessed 9 January 2009, September 1996. {28}

[DOD78] DOD - Department of Defense. MIL-STD-1553B, Aircraft Internal Time Division Command/Response Multiplex Data Bus. Technical report, Department of Defense, September 1978. {27}

[DSS98] Xavier Défago, André Schiper, and Nicole Sergent. Semi-Passive Replication. In *proceedings of Symposium on Reliable Distributed Systems*, pages 43–50, 1998. {15}

[EHN+08] K. Etschberger, R. Hofmann, A. Neuner, U. Weissenrieder, and B. Wiulsroed. A Failure-Tolerant CANopen System for Marine Automation Systems. Available: `www.ixxat.com/download/artikel_norcontrol.pdf`. Last accessed 27 March 2008, 2008. {44}

[Eth09] EtherCAT Technology Group. EtherCAT - The Ethernet fieldbus. Available: `www.ethercat.org/pdf/ethercat_e.pdf`. Last accessed 21 July 2009, 2009. {29}

[FAF+03] Joaquim Ferreira, Luís Almeida, José Fonseca, Guillermo Rodriguez-Navas, and Julián Proenza. Enforcing Consistency of Communication Requirements Updates in FTT-CAN. In *proceedings of the Workshop on Dependable Embedded Systems, held in conjunction with the* 22^{nd} *Symposium on Reliable Distributed Systems (SRDS)*, pages 7–12, Florence, Italy, October 2003. {72,78,103}

[FAF+06] Joaquim Ferreira, Luís Almeida, José Fonseca, Paulo Pedreiras, Ernesto Martins, Guillermo Rodriguez-Navas, Joan Rigo, and Julián Proenza. Combining operational flexibility and dependability in FTT-CAN. *IEEE Transactions on Industrial Informatics*, 2(2):95–102, May 2006. {78,81,156,157,161}

[FAM+03] Joaquim Ferreira, Luís Almeida, Ernesto Martins, Paulo Pedreiras, and José Fonseca. Components to Enforce Fail-Silent Behavior in Dynamic Master-Slave Systems. In *proceedings of the* 5^{th} *IFAC International Symposium on Intelligent Components and Instruments for Control Applications (SICICA)*, Aveiro, Portugal, July 2003. {75,161}

[FBS+08] José Fonseca, Paulo Bartolomeu, Valter Silva, Vasco Santos, Carlos Abreu, Alexandre Mota, Margarida Cunha, and Arminda Lopes. Using CAN to retrofit houses for quadriplegic people. In 12^{th} *International CAN Conference*, Barcelona, Spain, March 2008. {1}

[FBSC07] José Fonseca, Paulo Bartolomeu, Valter Silva, and Francisco Carreiro. On the practical issues of implementing the VTPE-hBEB protocol in small processing power controllers. In *proceedings of the* 12^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 949 – 956, Patras, Greece, September 2007. {132,144}

[FDH+07] Howard Frazier, Schelto Van Doorn, Robert Hays, Shimon Mulle, Bruce Tolley, Paul Kolesar, Geoff Thompson, and Brad Turner. IEEE 802.3ad Link Aggregation. Available: `www.ieee802.org/3/hssg/public/apr07/frazier_01_0407.pdf`. Last accessed 8 January 2009, April 2007. {29}

[Fel02] Max Felser. The fieldbus standards: History and structures. Available: `felser.ch/download/FE-TR-0205.pdf`. Last accessed 3 August 2010, 2002. {21}

[Fel04] J. Feld. PROFINET - scalable factory communication for all applications. In *proceedings of the* 5^{th} *IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 33–38, Vienna, Austria, September 2004. {28}

[Fel06] Max Felser. Quality of PROFIBUS installations. In *proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 113–118, Torino, Italy, June 2006. {21}

[Fel08] M. Felser. Media Redundancy for PROFINET IO. In *proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 325–330, Desden, germany, May 2008. {28}

[Fer05] Joaquim José Castro Ferreira. *Faul-Tolerance in Flexible Real-Time Communication Systems.* PhD thesis, University of Aveiro, 2005. {53,56,75,108,109,126}

[Fle02] FlexRay group. Handouts of the International FlexRay Workshop. *FlexRay Consortium*, April 2002. {2,24,25}

[Fle04a] FlexRay Consortium. FlexRay Communications System - Bus Guardian Specification, v2.0. Technical report, FlexRay Consortium, 2004. {24}

[Fle04b] FlexRay Consortium. FlexRay Communications System - Electrical Physical Layer Specification, v2.0. Technical report, FlexRay Consortium, 2004. {24}

[Fle04c] FlexRay Consortium. FlexRay Communications System - Protocol Specification, v2.0. Technical report, FlexRay Consortium, 2004. {24,26,54}

[FNP+98] P. Ferriol, F. Navio, J. Pons, Julián Proenza, and José Miro-Julia. A double CAN architecture for fault-tolerant control systems. In *proceedings of the* 5^{th} *International CAN Conference, (iCC)*, San Jose, USA, November 1998. {v,41}

[FNTTJ04] K. Forsberg, S. Nadjm-Tehrani, J. Torin, and R. Johansson. Maintaining consistency among distributed control nodes. In *proceedings of the* 23^{rd} *Digital Avionics Systems Conference (DASC)*, volume 2, pages 6.D.2–61–12, October 2004. {17}

[FOFF04] Joaquim Ferreira, Arnaldo Oliveira, Pedro Fonseca, and José Fonseca. An Experiment to Assess Bit Error Rate in CAN. In *proceedings of the* 3^{rd} *International Workshop on Real-Time Networks (RTN) satellite held in conjunction with the* 16^{th} *Euromicro International Conference on Real-Time Systems*, Rennes, France, June 2004. {157}

[FPAF02] Joaquim Ferreira, Paulo Pedreiras, Luís Almeida, and José Fonseca. The FTT-CAN protocol: improving flexibility in safety-critical systems. *IEEE Micro (special issue on Critical Embedded Automotive Networks)*, 22(4):46–55, July, August 2002. {3}

[FR97] M. Farsi and K. Ratcliff. CANopen: the open communications solution. In *proceedings of the IEEE International Symposium on Industrial Electronics (ISIE)*, volume 1, pages 112–116, Guimarães, Portugal, July 1997. {43}

[FRB99] M. Farsi, K. Ratcliff, and Manuel Barbosa. An Introduction to CANopen. *Computing & Control Engineering Journal*, 1:161–168, 1999. {44}

[Fre95] L. B. Fredriksson. CAN Kingdom Rev. 3.01. Technical report, KVASER AB, Sweden, 1995. {40}

[Gär99] Felix Gärtner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM computing surveys*, 31(1):1–26, March 1999. {12}

[Gif79] David Gifford. Weighted voting for replicated data. In *proceedings of the seventh ACM symposium on Operating systems principles (SOSP)*, pages 150–162, New York, NY, USA, 1979. ACM. {17}

[GMB85] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985. {17}

[GS96] Rachid Guerraoui and André Schiper. Fault-Tolerance by Replication in Distributed Systems. In *proceedings of Conference on Reliable Software Technologies (invited paper)*, pages 38–57, Montreux, Switzerland, 1996. Springer Verlag. {17}

[Han05] K. Hansen. Redundancy Ethernet in industrial automation. In *proceedings of the* 10^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 2, pages 7 pp.–947, Catania, Italy, September 2005. {28}

[Has80a] E. Hassler. An industrial master-slave system: Organization for communications for decentralized freely-programmable controls. *Technische Rundschau*, 72(17):25–26, April 1980. {17}

[Has80b] E. Hassler. Industrial master/slave system. *Technische Rundschau*, 72(14):17–18, April 1980. {17}

[HB99] Florian Hartwich and Armin Bassemir. The configuration of the CAN Bit Timing. In *proceedings of the* 6^{th} *International CAN Conference (iCC)*, Turin, Italy, November 1999. {30}

[Hen09] Carl Henning. PROFIBUS and PROFINET fieldbuses. Available: `www2.sea.siemens.com/NR/rdonlyres/0377ADFD-67DA-4770-8513-1A6CE01BCE3E/0/1574.pdf`. Last acessed 29 August 2009, 2009. {28,50}

[HKD97] H. Hilmer, H.-D. Kochs, and E. Dittmar. A fault-tolerant communication architecture for real-time control systems. In *proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS)*, Barcelona, Spain, October 1997. {v,42}

[HMFH00] F. Hartwich, B. Müller, T. Führer, and R. Hugel. CAN Network with Time-triggered Communication. In *proceedings of* 7^{th} *International CAN Conference (iCC)*, Amsterdam, The Netherlands, October 2000. CAN in Automation GmbH. {2,36}

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, 2^{nd} edition, 1993. {17}

[Huc07] David Hucaby. *CCNP BCMSN Oficial Exam Certification Guide*. Cisco Press, 2007. {29}

[HWV03] B.S. Heck, L.M. Wills, and G.J. Vachtsevanos. Software technology for implementing reusable, distributed control systems. *Control Systems Magazine, IEEE*, 23(1):21–35, Feb 2003. {16}

[IEC00] IEC. IEC International Standard 61158: Fieldbus standard for use in industrial control systems - Type 1: Foundation Fieldbus H1; Type 3: PROFIBUS; Type 7: WorldFIP. *International Electrotechnical Committee*, 2000. {18}

[IEC07] IEC. Industrial communication networks - Fieldbus specifications - Part 4-1: Data-link layer protocol specification - Type 1 elements. Available: `webstore.iec.ch/webstore/webstore.nsf/artnum/038760`. Last accessed 28 January 2009, December 2007. {21}

[IEE09] IEEE. 802.1w. Available: `www.ieee802.org/1/pages/802.1w.html`. Last accessed 3 August 2010, 2009. {28,29}

[IJCS08] C. Isen, L. John, Jung Pil Choi, and Hyo Jung Song. On the representativeness of embedded java benchmarks. In *proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 153 –162, Seattle, WA, USA, september 2008. {56}

[Imo02] M. Imoto. Global standardization activities of DeviceNet. In *proceedings of the* 41^{st} *SICE Annual Conference*, volume 2, pages 913–916, Osaka, Japan, August 2002. {45}

[Ins97] Institut für Technische Informatik. TTP/C. Available: `www.vmars.tuwien.ac.at/projects/ttp/ttpc.html`. Last accessed 8 August 2010, September 1997. {24}

[Int04] Intel Corporation. Intel 82527 datasheet, 2004. {58}

[ISO93] International Standards Organisation. *ISO 11898. Road Vehicles–Interchange of digital information–Controller Area Network (CAN) for high speed communication*, 1993. {30,31,130}

[ISO94] ISO. *Information technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*. ISO/IEC, 7498-1 edition, 1994. {92}

[ISO01] ISO. Road vehicles - controller area network (CAN) - part 4: Time triggered communication, 2001. {2,36}

[ISO03] ISO. ISO 11992: Road vehicles – interchange of digital information on electrical connections between towing and towed vehicles, 2003. {31}

[IXX08] IXXAT. IXXAT web page. Available: `www.ixxat.de/fo-star-coupler_en,7460,5873.html`. Last accessed 24 February 2008, 2008. {44}

[KAGS05] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The Time-Triggered Ethernet (TTE) Design. In *proceeding of the* 8^{th} *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 22–33, Orlando, Florida, USA, May 2005. {28}

[KAK+97] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The Robot World Cup Initiative. In *proceedings of the first international conference on Autonomous agents: AGENTS '97*, pages 340–347, New York, USA, 1997. ACM Press. {116}

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. In *proceedings of the IEEE*, volume 91, pages 112–126, 2003. {23}

[KD06] H. Kirrmann and D. Dzung. Selecting a standard redundancy method for highly available industrial networks. In *proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 386–390, Torino, Italy, 2006. {28}

[KG93] Hermann Kopetz and Günter Grünsteidl. TTP - A time-triggered protocol for fault-tolerant real-time systems. In *proceedings of the* 23^{rd} *International Symposium on Fault-Tolerant Computing (Digest of Papers)*, pages 524–533, Toulouse, France, June 1993. {24}

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP - A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, January 1994. {2,23}

[KGHL98] Hans-Dieter Kochs, Walter Geisselhardt, Holger Hilmer, and M. Lenord. Fault-tolerant communication in large-scale manipulators. In *proceedings of the* 17^{th} *International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 254–266, Heidelberg, Germany, 1998. Springer-Verlag. {43}

[KLMB08] Máté Kovács, Paolo Lollini, István Majzik, and Andrea Bondavalli. An integrated framework for the dependability evaluation of distributed mobile applications. In *proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems (SERENE)*, pages 29–38, New York, NY, USA, 2008. ACM. {170}

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Press, 1997. {116}

[Kop01] Hermann Kopetz. A comparison of TTP/C and FlexRay. Technical report, Technishe Universitat Wien, Austria, May 2001. {26}

[KP91] K. Kanoun and D. Powell. Dependability evaluation of bus and ring communication topologies for the Delta-4 distributed fault-tolerant architecture. In *proceedings of the* 10^{th} *Symposium on Reliable Distributed Systems*, pages 130–141, Pisa, Italy, September, October 1991. {13}

[Lap92] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-tolerant Systems*. Springer-Verlag, 1992. IFIP WG 10.4. {10,12}

[Lap95] Jean-Claude Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *proceedings of the* 25^{th} *International Symposium on Fault-Tolerant Computing 'Highlights from Twenty-Five Years'*, pages 2–11, June 1995. {10}

[LHB03] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design (sections)*. Rockport Publishers, October 2003. {14}

[LPDL07] S. Limal, S. Potier, B. Denis, and J.-J. Lesage. Formal verification of redundant media extension of ethernet powerlink. In *proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1045–1052, Patras, Greece, September 2007. {28}

[LS95] D. Loy and R. Schmalek. Thoughts about redundancy in fieldbus systems anchored in OSI Layer-4 and applied to the Lontalk Protocol on neuron based network nodes. In *proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 21–26, Leysin, Switzerland, October 1995. {*v*,21}

[LY05] C.E. Lin and H.M. Yen. Reliability and stability survey on CAN-based avionics network for small aircraft. In *proceedings of the* 24^{th} *Digital Avionics Systems Conference (DASC)*, volume 2, October 2005. {13,14}

[Mar03] João Martins. A Replica Consistency Algorithm for GlobData. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, March 2003. {16,17}

[MCB+10] Ricardo Moraes, Francisco Carreiro, Paulo Bartolomeu, Valter Silva, José Fonseca, and Francisco Vasques. Enforcing the timing behavior of real-time stations in legacy bus-based industrial Ethernet networks. *Computer Standards & Interfaces*, In Press, Corrected Proof, 2010. {132}

[MDM07] I. Majzik, P. Domokos, and M. Magyar. Tool-supported Dependability Evaluation of Redundant Architectures in Computer Based Control Systems. In E. Schnieder and G. Tarnai, editors, *proceedings of the* 6^{th} *Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT)*, pages 342–352, Braunschweig, Germany, January 2007. {170}

[MFA+02] Ernesto Martins, Joaquim Ferreira, Luís Almeida, Paulo Pedreiras, and José Fonseca. An Approach to the Synchronization of Backup Masters in Dynamic Master-Slave Systems. *proceedings of the WiP Session of* 23^{rd} *IEEE International Real-Time Systems Symposium (RTSS)*, 2002. {3}

[MFF09] Ahlem Mifdaoui, Fabrice Frances, and Christian Fraboul. Centralized vs distributed communication scheme on Switched Ethernet for embedded military applications. In *proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 201–210, Lausanne, Switzerland, July 2009. {28}

[MFH+02] Bernd Müller, Thomas Führer, F. Hartwich, R. Hugel, and H. Weiler. Fault Tolerant TTCAN Networks. In *proceedings of* 8^{th} *International CAN Conference (iCC)*, Las Vegas, NV, USA, October 2002. CAN in Automation GmbH. {*v*,36,37}

[MH00] Shimon Muller and Ariel Hendel. IEEE P802.3ae 10Gb/s Ethernet. Available: `grouper.ieee.org/groups/802/3/ae/public/blue_book.pdf`. Last accessed 17 January 2009, July 2000. {28}

[Mic06] Microchip. PIC18FXX8 Data Sheet, 2006. 41159e version. {101,107,116}

[Mic08] Microchip. DsPIC30F6011A/6012A/6013A/6014A Data Sheet, 2008. DS70143D version. {101,133}

[Mil05] MilCAN working Group. MilCAN on Terrier. Available: `www.milcan.org/Newsletters/MilCAN_matters_Iss3_.pdf`. Last accessed 3 August 2010, December 2005. {45}

[Mil06] MilCAN Working Group. MilCAN Specification, March 2006. {27,45}

[MNF02] Ernesto Martins, Paulo Neves, and José Fonseca. Architecture of a fieldbus message scheduler coprocessor based on the planning paradigm. *Microprocessors and Microsystems*, 26(3):97–106, 2002. {159}

[MNS96] A. Meschi, M. Di Natale, and M. Spuri. Priority inversion at the network adapter when scheduling messages with earliest deadline techniques. In *proceedings of the* 8^{th} *Euromicro Workshop on Real-Time Systems*, pages 243–248, L'Aquila, Italy, December 1996. {58}

[Mor82] H. M. Morris. Distributed system makes wide use of bubble memories. *Control Engineering*, 29(1):68–70, January 1982. {17}

[Mot95] Motorola. MC54/74HC4066 Datasheet. Available: `www.datasheetcatalog.org/datasheet/motorola/4066.pdf`. Last accessed 4 March 2010, 1995. {129}

[MSF+06] Ricardo Marau, Valter Silva, Joaquim Ferreira, Luís Almeida, and José Fonseca. Assessment of FTT-CAN master replication mechanisms for safety-critical applications. In *proceedings of the 2006 SAE congress & exhibition*, number 06AE-278, 2006. {78,112,121,124}

[MSF+07] Ricardo Marau, Valter Silva, Joaquim Ferreira, Luís Almeida, and José Fonseca. Assessment of FTT-CAN master replication mechanisms for safety-critical applications. *Transactions Journal of Passenger Cars-Electronic and Electrical Systems*, pages 447–455, March 2007. {112,121,124}

[MT06] Rainer Makowitz and Cristopher Temple. FlexRay - A Communication Network for Automotive Control Systems. In *proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 113–118, Torino, Italy, June 2006. {v,24,26}

[MZP+02] A. Martinoli, Y. Zhang, P. Prakash, E. K. Antonsson, and R. D. Olney. Towards evolutionary design of intelligent transportation systems. In *proceedings of the*

11^{th} *International Symposium of the Associazione Tecnica dell'Automobile on Advanced Technologies for ADAS Systems*, Siena, Italy, 2002. {1}

[NHN02] Thomas Nolte, Hans Hansson, and Christer Norström. Minimizing can response-time jitter by message manipulation. In *proceedings of the* 8^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 197–206, San Jose, California, USA, September 2002. IEEE Computer Society. {33,61}

[NHN03] Thomas Nolte, Hans Hansson, and Christer Norström. Probabilistic Worst-Case Response-Time Analysis for the Controller Area Network. In *proceedings of the The* 9^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 200–207, Washington, DC, USA, 2003. IEEE Computer Society. {35,58,60,69,125}

[NHNP01] Thomas Nolte, Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Using bit-stuffing distributions in CAN analysis. In Steve Liu Iain Bate, editor, *proceedings of the IEEE/IEE Real-Time Embedded Systems Workshop in conjunction with the* 22^{nd} *IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001. Department of Computer Science, University of York. {33,59,60,61}

[NNFSL08] Nicolas Navet and Françoise Simonot Lion. *The Automotive Embedded Systems Handbook*. Industrial Information Technology Series. Taylor & Francis / CRC Press, 2008. {38}

[NVi06] NVidia. NVIDIA DualNet With Teaming Advanced Networking. Available: `www.nvidia.com/content/nforce5/TB-02499-001_v01_DualNet.pdf`. Last accessed 6 January 2009, May 2006. {29}

[Oli07] Arnaldo Silva Rodrigues Oliveira. *Especialização e Síntese de Processadores para Aplicação em Sistemas Tempo-real*. PhD thesis, University of Aveiro, 2007. {164}

[Ope04] Open DeviceNet Vendor Association (ODVA). DeviceNet Technical Overview. Available: `www.odva.org/Portals/0/Library/Publications_Numbered/PUB00026R1.pdf`. Last accessed 7 June 2010, 2004. {45,46}

[Ope10] Open DeviceNet Vendor Association (ODVA). ODVA WebPage. `www.odva.org`, accessed June 2010, 2010. {45}

[OSCA08] Panagiotis Oikonomidis, Elias Stipidis, Periklis Charchalakis, and Falah Ali. MilCAN Fault Tolerance Layer. In *proceedings of the* 12^{th} *international CAN Conference*, pages 05.1 – 05.7, Barcelona, March 2008. {45}

[Pas04] Michel Passemard. Atmel Microcontrollers for Controller Area Network (CAN). Technical report, Atmel Corporation, 2004. {61}

[PB95] Evaggelia Pitoura and Bharat Bhargava. Maintaining consistency of data in mobile distributed environments. In *proceedings of the* 15^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pages 404–413, Vancouver, British Columbia, Canada, June 1995. {16}

[PC01] Paulo Portugal and Adriano Carvalho. On dependability evaluation of field-bus networks: a permanent fault analysis. In *proceedings of the* 27^{th} *Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pages 299–304, Denver, USA, November, December 2001. {13,170}

[PD09] Proces-Data. Intelligent Redundancy for P-NET. Available: `www.proces-data.com/?/53/00S001P1D5/LHAFH-01/Description_ENG.htm`. Last accessed 4 July 2009, 2009. {22}

[Ped03] Paulo Bacelar Reis Pedreiras. *Supporting Flexible Real-Time Communication on Distributed Systems*. PhD thesis, University of Aveiro, Portugal, July 2003. {54,72}

[PF04] Juan Pimentel and José Fonseca. FlexCAN: A Flexible Architecture for Highly Dependable Embedded Applications. In *proceedings of the* 3^{rd} *International Workshop on Real-Time Networks sattelite held in conjunction with the* 16^{th} *Euromicro International Conference on Real-Time Systems*, Rennes, France, June 2004. {v,2,38,39}

[PLS82] Marshall Pease, Leslie Lamport, , and Robert Shostak. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. {16,23}

[PPMJ99] Julián Proenza, J. Pons, and J. Miro-Julia. A low-cost fail-safe circuit for fault-tolerant control systems. In *proceedings of The* 6^{th} *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, volume 2, pages 887–890, Pafos, Cyprus, September 1999. {41,42}

[PRO09] PROFIBUS. PROFIBUS Webpage. Available: `www.pofibus.com`. Last accessed 5 August 2009, 2009. {21}

[PS01] Juan Pimentel and T. Sacristan. A fault management protocol for TTP/C. In *proceedings of the* 27^{th} *Annual Conference of the IEEE Industrial Electronics Society (IECON)*, volume 3, pages 1800–1805, Denver, CO, USA, 2001. {24}

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. {16}

[RNPR+04] Guillermo Rodríguez-Navas, Julián Proenza, J. Rigo, Joaquim Ferreira, Luís Almeida, and José Fonseca. Design and Modeling of a Protocol to Enforce Consistency among Replicated Masters in FTT-CAN. In *proceedings of the* 5^{th} *Workshop on Factory Communication Systems (WFCS)*, pages 229–238, Vienna, Austria, September 2004. {72,115}

[RPA08] José Rufino, Ricardo Pinto, and Carlos Almeida. FPGA-based Engineering of Bus Media Redundancy in CAN. In *proceeding of the* 12^{th} *internacional CAN Conference*, pages 4.23 – 7.29, Barcelona, March 2008. {41}

[Ruf02] José Rufino. *Computational System for Real-Time Distributed Control.* PhD thesis, Technical University of Lisbon, 2002. {41}

[RVA99] José Rufino, Paulo Veríssimo, and Guilherme Arroz. A Columbus' Egg Idea for CAN Media Redundancy. In *proceedings of the* 29^{th} *Annual International Symposium on Fault-Tolerant Computing (Digest of Papers)*, volume 0, pages 286–293, Madison, WI , USA, June 1999. IEEE Computer Society. {*v*,30,34,40,59,75}

[SAE92] SAE. *SAE Handboolk*, volume 2. SAE, 1992. {23}

[SAE00] SAE International. Single Wire CAN Network for Vehicle Applications, February 2000. {31}

[SBFF09] Valter Silva, Paulo Bartolomeu, Joaquim Ferreira, and José Fonseca. Assessment of multi-bus fault-tolerant communications. In *proceedings of the* 7^{th} *IEEE International Conference on Industrial Informatics (INDIN)*, pages 72 –78, Cardiff, Wales, UK, June 2009. {127,128}

[Sch82] T. H. Schwalenstocker. A process control system using multibus. In *proceedings of the* 1^{st} *Annual Control Engineering Conference*, pages 133–137, 1982. {17}

[Sch90] A. Schill. Dependability in distributed applications-approaches and issues. In *proceedings of the Euromicro Workshop on Real Time*, pages 170–177, Horsholm, Sweden, June 1990. {14}

[Sem97] Philips Semiconductors. *TJA1053 - Fault-tolerant CAN transceiver*, October 1997. {34}

[SF06] Valter Silva and José Fonseca. Using FTT-CAN to Combine Redundancy with Increased Bandwidth. In *proceedings of the* 6^{th} *IEEE International Workshop*

on Factory Communication Systems (WFCS), pages 54–62, Torino, Italy, June 2006. {64,65,66}

[SFF06a] Valter Silva, Joaquim Ferreira, and José Fonseca. Dynamic Topology Management in CAN. In *proceeding of the* 11^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1222 – 1229, Prague, Czech Republic, September 2006. {156,157,163}

[SFF06b] Valter Silva, José Fonseca, and Joaquim Ferreira. Using FTT-CAN to the Flexible Control of Bus Redundancy and Bandwidth Usage. In *proceedings of the* 11^{th} *International CAN Conference (iCC)*, pages 5.9 – 5.15, Stockholm, Sweden, September 2006. {65}

[SFF07a] Valter Silva, Joaquim Ferreira, and José Fonseca. Flexible Bus Media Redundancy. In *proceedings of the* 4^{th} *International Workshop on Dependable Embedded Systems (in conjunction with the* 26^{th} *Symposium on Reliable Distributed Systems)*, Beijing, China, October 2007. {64}

[SFF07b] Valter Silva, Joaquim Ferreira, and José Fonseca. Master Replication and Bus Error Detection in FTT-CAN with Multiple Buses. In *proceedings of the* 12^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Patras, Greece, 2007. {79}

[SFF07c] Valter Silva, José Fonseca, and Joaquim Ferreira. Adapting the FTT-CAN Master for Multiple-bus Operation. In *proceedings of the* 5^{th} *IEEE International Conference on Industrial Informatics (INDIN)*, Vienna, Austria, July 2007. {81,105,127,128}

[SFNM05] Valter Silva, José Fonseca, Urbano Nunes, and Rodrigo Maia. Communications Requirements for Autonomous Mobile Robots: Analysis and Examples. In Elsevier, editor, *proceeding of FeT 2005*, pages 91 – 98, Mexico, November 2005. {1}

[SGAK06] Klaus Steinhammer, Petr Grillinger, Astrit Ademaj, and Hermann Kopetz. A Time-Triggered Ethernet (TTE) switch. In *proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 794–799, 3001 Leuven, Belgium, 2006. European Design and Automation Association. {28}

[SGN02] Håkan Sivencrona, F. Björn G., and Nilsson. A Novel Distributed Add-on Concept to Detect and Recover from Bus Failures on Controller Area Network using RedCAN. In *proceeedings of the* 8^{th} *International CAN Conference*, pages 05(9–15), Las Vegas - USA, February 2002. {40,150}

[She09] Michael Philip Sousa Sher. Dynamic topology management unit for fieldbus. Master's thesis, University of Aveiro, 2009. {153,164}

[Sil02] Valter Filipe Miranda Castelão Silva. Ambiente *Java* para Sistemas *Embedded*. Master's thesis, University of Aveiro, 2002. {56}

[Sil09] Vítor Hugo Martins Silva. Desenvolvimento de componentes para sistemas de segurança crítica. Master's thesis, University of Aveiro, 2009. {*vi*,153,164,165}

[SL02] J. Srinivasan and K. Lundqvist. Real-time architecture analysis: a COTS perspective. In *proceedings of the* 21^{st} *Digital Avionics Systems Conference*, volume 1, pages 5D4–1 – 5D4–9, Irvine, CA, October 2002. {27}

[SMA+05] Valter Silva, Ricardo Marau, Luís Almeida, Joaquim Ferreira, Mário Calha, Paulo Pedreiras, and José Fonseca. Implementing a distributed sensing and actuation system: The CAMBADA robots case study. In *proceedings of the* 10^{th} *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 2, pages 781–788, Catania, Italy, September 2005. {115,116,121}

[SMO+08] Rui Santos, Ricardo Marau, Arnaldo Oliveira, Paulo Pedreiras, and Luís Almeida. Designing a costumized Ethernet switch for safe hard real-time communication. In *proceeding of the IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 169–177, Dresden, Germany, May 2008. {28}

[SO06] Heikki Saha and Sandvik Tamrock Oy. Active high-speed CAN hub. In *proceedings of the* 11^{th} *international CAN Conference*, pages 2–8 to 2–14, Stockholm, Sweden, September 2006. {46,47}

[SOJT04] Håkan Sivencrona, Torbjörn Olsson, Roger Johansson, and Jan Torin. RedCAN: Simulations of Two Fault Recovery Algorithms for CAN. In *proceedings of the* 10^{th} *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 302–311, Washington, DC, USA, 2004. IEEE Computer Society. {*v*,39}

[SS93] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *proceedings of the* 13^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pages 561–568, Pittsburgh, Pennsylvania, USA, May 1993. IEEE Computer Society Press. {17}

[Sta85] J.A. Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions on Software Engineering*, SE-11(10):1141–1152, OctŽober 1985. {14}

[SV07] L. Seno and S. Vitturi. A simulation study of ethernet powerlink networks. In *proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 740–743, Patras, Greece, September 2007. {28}

[Tay06] Graham Taylor. CAN implementation on Terrier. In *proceedings of* 11^{th} *international CAN Conference (iCC)*, pages 08.8 – 08.12, Stockholm, Sweden, September 2006. {45}

[TBW95] Ken Tindell, A. Burns, and A. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995. {58}

[Tea98] X-By-Wire Team. X-By-Wire - Safety Related Fault Tolerant Systems in Vehicles. Technical report, X-By-Wire Consortium, 1998. {1}

[Tho93] Jean-Pierre Thomesse. Time and Industrial Local Area Networks. In *proceeding of 'Computers in Design, Manufacturing, and Production', (CompEuro)*, pages 365–374, May 1993. {18}

[Tho05] Jean-Pierre Thomesse. Fieldbus Technology in Industrial Automation. *Proceedings of the IEEE*, 93(6):1073–1101, June 2005. {17}

[THW95] Ken Tindell, Hans Hansson, and Andy Wellings. Analysing Real-Time Communications: Controller Area Network (CAN). In *proceedings of the* 15^{th} *IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1995. IEEE Society Press. {58}

[TO08] Vardhaman Tiwatane and Deelip Omana. Implementation of Fault Tolerant Network Management System for CAN Bus using CANopen. In *proceedings of the* 12^{th} *International CAN Conference*, pages 05.16 – 05–21, Barcelona, Spain, March 2008. {45}

[TV99] Eduardo Tovar and Francisco Vasques. Real-time fieldbus communications using profibus networks. *IEEE Transactions on Industrial Electronics*, 46(6):1241–1251, December 1999. {20}

[TV01] Eduardo Tovar and Francisco Vasques. Distributed computing for the factory-floor: a real-time approach using WorldFIP networks. *Computers in Industry*, 44(1):11 – 31, January 2001. {18}

[Uni10] University of Aveiro. CAMBADA Webpage. Available: `www.ieeta.pt/atri/cambada/index.htm`. Last accessed 2 August 2010, 2010. {123}

[Ver96] Paulo Veríssimo. Segurança e Confiabilidade: na Ordem do Dia dos Sistemas Distribuídos. Jornadas do Colégio de Engenharia Electrotécnica, Ordem dos Engenheiros, 1996. {12}

[VL89] Paulo Veríssimo and Rogério Lemos. Confiança no funcionamento: Proposta para uma terminologia em português. Technical report, INESC and LCMI/UFSC, 1989. {12}

[VR01] Paulo Veríssimo and Luís Rodrigues. *Distributed Systems For System Architects*. Kluwer Academic Press, 2001. {12}

[VSI04] VSI - Vehicle Systems Integration. Terrier: Vehicle Systems Integration in a new platform. Available: `www.vsi.org.uk/VSI_Briefing_Day_RO_Defence.pdf`. Last accessed 18 January 2009, September 2004. {45}

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *proceedings of* 20^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pages 264–274, Taipei, Taiwan, April 2000. IEEE Computer Society Technical Commitee on Distributed Processing. {15,16}

[WTSW03] Fei-Yue Wang, Shuming Tang, Yagang Sui, and Xiaojing Wang. Toward intelligent transportation systems for the 2008 olympics. *IEEE Intelligent Systems*, 18(6):8–11, 2003. {1}

[Xil09] Xilinx. Xilinx Spartan-3 FPGA datasheet. Available: `www.xilinx.com/support/documentation/data_sheets/ds099.pdf`. Last accessed 1 June 2010, December 2009. {164}

[Xil10] Xilinx. PicoBlaze 8-bit Embedded Microcontroller User Guide. Available: `www.xilinx.com/support/documentation/ip_documentation/ug129.pdf`. Last accessed 1 June 2010, January 2010. {164}

[Zim80] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425 – 432, April 1980. {92}

[ZV08] Wenbing Zhao and F. Villaseca. Byzantine fault tolerance for electric power grid monitoring and control. In *proceedings of the International Conference on Embedded Software and Systems (ICESS)*, pages 129 – 135, Chengdu, Sichuan, China, july 2008. {23}

Acronyms

ACK	Acknowledge, page 32
AM	Asynchronous Message, page 104
API	Application Programming Interface, page 89
ART	Asynchronous Requirements Table, page 93
BA	Bus Arbitrator, page 18
BAT	Bus Arbitration Table, page 18
BEAM	Bus Error Assynchronous Message, page 80
BFD	Bus Failure Detector, page 159
CAL	CAN Application Layer, page 43
CAMBADA	Cooperative Autonomous Mobile Robots with Advanced Distributed Architecture, page 63
CAN	Controller Area Network, page 1
CANdor	CAN Duplicated Organization for Reliability, page 41
CANELy	CAN Enhanced Layer, page 40
CBA	Current Bus Allocation, page 81
CiA	CAN in Automation, page 43
CIP	Common Industrial Protocol, page 45
CPLD	Complex Programmable Logic Device, page 47
CRC	Cyclic Redundancy Check, page 26
CSR	Configuration and Status Record, page 93
DES	Distributed Embedded Systems, page 1
DLC	Data Length Code, page 32
DMS	Delay Measurement System, page 4
DP	Decentralized Periphery, page 20
DRM	Dynamic Redundancy Management, page 156
DT3C	Distributable Table Content Consistency Checker, page 164
EC	Elementary Cycle, page 18
EDF	Earliest Deadline First, page 58
EOF	End-of-Frame, page 32

EtherCAT	Ethernet for Control Automation Technology, page 29
FAM	Fault-Active Mechanism, page 42
FDL	Fieldbus Data Link, page 20
FIP	Factory Instrumentation Protocol, page 17
FMS	Fieldbus Message Communications, page 20
FPGA	Field Programmable Gate Array, page 158
FT	Fault Tolerant, page 45
FTNMSCAN	Fault Tolerant Network Management System for CAN, page 44
FTT-CAN	Flexible Time-Triggered communication on Controller Area Network, page 2
FTTTQ	FTT Time Quantum, page 97
FTU	Fault-Tolerant Unit, page 24
h-BEB, hBEB	High Priority Binary Exponential Backoff, page 131
I/O	Input/Output, page 20
IBA	Initial Bus Allocation, page 81
ID	Identifier, page 78
IDE	Identifier Extension, page 32
IFS	Intermission Frame Space, page 32
ISR	Interrupt Service Routine, page 108
JVM	Java Virtual Machine, page 56
LAW	Lenght of the Assynchronous Window, page 66
LEC	Lenght of the Elementary Cycle, page 54
LSC	Logic Star Coupler, page 47
LSW	Length of the Synchronous Window, page 66
LTM	Lenght of the Trigger Message, page 66
MAC	Media Access Control, page 18
MC	Main Comparator, page 41
MIC	Master with Initial Configuration, page 92
MOR	Master with Online Reconfiguration, page 93
NIT	Network Idle Time, page 26
NRT	Non-Real-Time Requirements Table, page 93
NRZ	Non-Return-to-Zero, page 22
NSU	Network Switch Unit, page 4
ODVA	Open DeviceNet Vendor Association, page 45
PA	Process Automation, page 20
PC	Personal Computer, page 29

PCB Physically Connected Bus, page 81
PDC Producers-Distributor-Comsumers, page 18
PDO Process Data Objects in CANopen, page 44
PH1 Phase Buffer Segment 1, page 30
PH2 Phase Buffer Segment 2, page 30
PI Proportional-Integral, page 117
PLC Programmable Logic Controller, page 20
PROFIBUS Process Field Bus, page 9
PRP Propagation Time Segment, page 30
RCMP Redundancy and Communication Management Processor, page 41
REC Receive Error Counter, page 35
RL Reverse Link, page 47
RM Reference Message, page 36
RSTP Rapid Spanning-Tree Protocol, page 28
RT Reconfiguration Table, page 159
RT Remote Terminal, page 27
RTR Remote Transmission Request, page 32
RZ Return-to-Rero, page 31
SAM Stand-Alone Master, page 92
SDO Service Data Objects in CANopen, page 43
SM Synchronous Message, page 107
SOF Start of Frame, page 32
SRDB Synchronous Requirements Database, page 54
SRT Synchronous Requirements Table, page 54
STP Spanning-Tree Protocol, page 28
SW Symbol Window, page 26
SYNC Synchronization Segment, page 30
TDMA Time Division Multiple Access, page 25
TEC Transmit Error Counter, page 35
TM Trigger Message, page 54
TMTW Trigger Message Transmission Window, page 74
TMU Topology Management Unit, page 4
TR Topology Reconfigurator, page 159
TT Time Token, page 20
TTA Time Triggered Architecture, page 23

TTCAN	Time-Triggered Controller Area Network, page 2
TTE	Time-Triggered Ethernet, page 28
TTP	Time Triggered Protocol, page 2
VTPE	Virtual Token Passing Ethernet, page 131
WDPF	Westinghouse Distributed Processing Family, page 17
xDMS	Extended Delay Measurement System, page 133