Bridging the gap between scheduling algorithms and scheduler implementations in time-triggered embedded systems

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Mouaaz Nahas

M.Sc. (Loughborough)

Embedded Systems Laboratory

Department of Engineering

University of Leicester

Leicester, United Kingdom

October 2008

Bridging the gap between scheduling algorithms and scheduler implementations in time-triggered embedded systems

Mouaaz Nahas

Abstract

The scheduling of tasks in real-time, resource-constrained embedded systems is typically performed using a simple scheduler. Scheduling algorithm is the key scheduler component which determines the way in which tasks can be executed to meet their timing constraints. To ensure precise task scheduling, the right decisions about the scheduler implementation have to be made. It has been argued that there is a wide gap between scheduling theory and its practical implementation which must be bridged to achieve a meaningful validation of embedded systems.

The work described in this thesis attempts to address this gap by proposing a simple (generic) technique, called the Scheduler Test Case (STC), which provides the facility to explore how a particular real-time scheduler implementation can be expected to behave under a range of both normal and abnormal operating conditions. The primary focus of this thesis is on single-processor embedded systems employing Time-Triggered Co-operative (TTC) architectures. The technique proposed is a testing method which helps facilitate an empirical "black-box" comparison between the behaviour of a set of representative implementation classes of the TTC scheduling algorithm. The key criterion against which scheduler behaviour is weighed up is the system predictability manifested by predictable task execution sequence, low timing jitter and unplanned-error handling capabilities. The implementation costs (including CPU, memory and power requirements) involved in creating each scheduler are also considered for distinguishing between the different TTC implementations.

The STC technique is then extended to provide a practical means for assessing the behaviour of multi-processor embedded designs employing Shared-Clock (S-C) scheduling architectures and TTC algorithm on the Controller Area Network (CAN) hardware protocol. In this part of the study, the STC technique explores the impact of using particular implementations of the S-C scheduler on the overall timing behaviour of multi-processor embedded systems. In addition to jitter behaviour which is measured empirically, the STC evaluates the communication behaviour by assessing the message latencies between any two communicating nodes in the network and the time frame required by the network to detect a temporary node failure. The results are expressed using mathematical equations. Moreover, the implementation costs (including network utilisation and memory overheads) are also considered to differentiate between the compared S-C schedulers.

The thesis finally concludes by discussing the overall findings of this project and making some proposals for future work in the area concerned with in the project.

I dedicate this thesis to my parents and wife

Professor Mahmoud Nahas, Mrs. Sahar Abousaleh & Mrs. Zilal Aseel

Acknowledgement

First of all, I would like to thank my supervisor *Professor Michael Pont* for providing me with constant guidance and help throughout this research project. Without him, this thesis would not have been possible.

Next, I would like to thank the UK Government for awarding me the EPSRC-DTA scholarship which covered my full-time tuition fees and life expenses in the first three years of this project. Also many thanks to the *University of Leicester's Welfare Office*, *The Worshipful Company of Engineers* and *Sidney Perry Foundation* for funding my project in the last year through paying my part-time tuition fees and some extra fund to compensate for my life expenses. Without their valuable contributions, I would not have been able to complete my course. I would specifically express my thanks to *Mr. Adrian Gascoigne* (a welfare officer at the University of Leicester) who made a substantial effort to help me receive such grants through advising me on the available charitable organisations and the right way to approach them. Of course, without his guidance, I would not have been able to proceed with this process.

I would also like to thank the following people who helped me in one way or another during my PhD period:

- *Zemian Hughes*: for helping me in creating the software codes for the TTC-MTI and TTC-Adaptive schedulers presented in Chapter 5.
- *Dr.Teera Phatrapornnant*: for helping me to carry out the power-consumption measurements, presented in Chapter 7, using a simple and cost-effective method.
- *Dr. Michael Short*: for guiding me through a major part of the work carried out for multi-processor embedded systems. In particular, he provided me with useful comments on the mathematical equations derived for the TTC-SCC scheduling protocols (Chapter 11) and the SBS programs (Appendix H). He also helped me to perform the experiments for the ACC testbed (Appendix G).

- *Dr. Devaraj Ayavoo*: for providing me with useful information about the prices for TTP protocol and supplying me with the software codes for the TTC-SCC1 – TTC-SCC4 schedulers described in Chapter 9.
- *Dr. Fernando Schlindwein*: for providing me with an appreciated help to obtain a three-month (free-of-charge) extension to my PhD period in the final year before I transferred my registration to part-time.
- *Ayman Gendy*: for providing me with useful feedback on the results presented in the thesis, and being a very close and sincere friend of mine during my time in Leicester.
- *Dr. Ricardo Bautista-Quintero*: for providing me with a great deal of support and encouragement, being my best friend in the ESL group.
- *Ziyad Mohammed*, *Saeed Ahmed*, *Yousef Bakkar* and *Dr. Adi Maaita*: for being very close and sincere friends of mine during my time in Leicester.
- *Riyadh Al-Rawi*: for carrying out a proofreading to a major part of my thesis document, and being the closest friend to me during my time in UK.

Also many thanks to all other people in the ESL research group and my friends in Loughborough for their good company.

More importantly, I would like to send special thanks to my honoured parents *Professor Mahmoud Nahas* and *Mrs. Sahar Abousaleh*, parents-in-low *Mr. Zuhair Aseel* and *Mrs. Safaa Labanieh*, brothers *Obeida*, *Mousaab* and *Huzifa Nahas*, and sisters *Arwa*, *Awfa* and *Aya Nahas* for the constant support and encouragement they provided me with during the whole period of my PhD course. I am very grateful to all of you and can never be able to thank you enough for your favour.

Finally, I would like to express my deepest thanks and gratitude to my wife *Zilal Aseel*, for her extensive help, patience, sacrifice, support, encouragement and prayers, without which I would definitely not have been able to complete my PhD work successfully. I am so grateful to you Zilal, and believe that whatever I try to give you in return would never be enough to reward you for what you have done to me.

Table of contents

List of figures

List of tables

List of author's publications

All papers, which have been published and submitted during the course of the work described in this thesis, are listed below. Please note that the contents of some of these papers have been adapted for presentation in this thesis: where applicable, a footnote at the beginning of a chapter indicates that material from one or more papers has been included.

Directly-related publications:

[1] Pont, M.J., Nahas, M., Phatrapornnant, T. and Hughes, Z. (Submitted) "Test cases for single-processor embedded systems which employ a time-triggered system architecture", submitted for a journal.

[2] Nahas, M., Pont, M.J. and Short, M. (In preparation) "Test cases for multi-processor embedded systems which employ a time-triggered and a shared-clock architectures on CAN networks".

[3] Nahas, M., Hughes, Z. and Pont, M.J. (In preparation) "Towards a 'perfect' time-triggered co-operative scheduler implementation for highly-predictable embedded systems"

Indirectly-related publications:

[4] Nahas, M., Pont, M.J. and Jain, A. (2004) "Reducing task jitter in shared-clock embedded systems using CAN". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.184-194. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

[5] Nahas, M., Short, M. and Pont, M. J. (2005) "The impact of bit stuffing on the real-time performance of a distributed control system", Proceeding of the 10th International CAN conference iCC (Rome, Italy, March 2005), pp. 10-1 to10-7.

[6] Nahas, M. and Pont, M.J. (2005) "Maximizing the reliability of CAN-based distributed embedded systems", Poster presentation at 'Festival of Postgraduate Research 2005' at University of Leicester, (Leicester, UK, June 2005).

[7] Nahas, M. and Pont, M.J. (2005) "Using XOR operations to reduce variations in the transmission time of CAN messages: A pilot study". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp.4-17. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

[8] Nahas, M. and Pont, M.J. (2006) "Reducing task jitter in TTC schedulers for resource-constrained embedded systems", Poster presentation at 'Festival of Postgraduate Research 2006' at University of Leicester, (Leicester, UK, June 2006).

[9] Nahas, M., Pont, M. J. and Short, M. (Submitted) "Reducing message-length variations in resource-constrained embedded systems implemented using the CAN protocol", submitted for a journal.

List of abbreviation

General terms:

ACC	Adaptive Cruise Control
CAN	Controller Area Network
CE	Cyclic executive
CM	Calculating Mode
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access / Collision Detection
DVS	Dynamic Voltage Scaling
EEM	Eight-to-Eleven Modulation
EFM	Eight-to-Fourteen Modulation
EMI	Electromagnetic Interface
ESL	Embedded Systems Laboratory
ET	Event-Triggered
GPOS	General Purpose Operating System
HIL	Hardware-in-the-Loop
HW	Hardware
I/O	Input/Output
ISR	Interrupt Service Routine
Kbps	Kilo bits per second
LIN	Local Interconnect Network
Mbps	Mega bits per second
MTI	Multiple Timer Interrupts
NFDT	Node-Failure Detection Time
NU	Network Utilisation
OM	Operating Mode
OS	Operating System
PC	Personal Computer
PID	Proportional, Integral and Derivative
PLL	Phase Looked Loop
RT	Release Time
RTOS	Real-Time Operating System
SBS	Software Bit Stuffing
S-C	Shared-Clock
SCC	Shared-Clock CAN
SD	Sandwich Delay
SL	Super Loop
SW	Software
TDMA	Time Division Multiple Access
TG	Task Guardian
TT	Time-Triggered
TTC	Time-Triggered Co-operative
TTCAN	Time-Triggered Controller Area Network
TTP	Time-Triggered Protocol
TTP/C	Time-Triggered Protocol, Class C

UART	Universal Asynchronous Receiver/Transmitter
WCET	Worst Case Execution Time

Scheduler architectures:

TTC-SL	Time-Triggered, Co-operative, Super Loop
TTC-ISR	Time-Triggered, Co-operative, Interrupt Service Routine
TTC-Dispatch	Time-Triggered, Co-operative, Dispatch
TTC-SD	Time-Triggered, Co-operative, Sandwich Delays
TTC-MTI	Time-Triggered, Co-operative, Multiple Timer Interrupts
TTC-TG	Time-Triggered, Co-operative, Task Guardians
TTC-DVS	Time-Triggered, Co-operative, Dynamic Voltage Scaling
TTC-SC	Time-Triggered, Co-operative, Shared-Clock
TTC-SCC	Time-Triggered, Co-operative, Shared-Clock CAN

PART A:

INTRODUCTION

Chapter 1
Introduction

1.1 Introduction

This introductory chapter provides a general overview of the work carried out during the course of this PhD project. It discusses the scope of this research and explains the aim of the studies detailed in the remainder of this thesis.

1.2 What is an embedded system?

Unlike general-purpose desktop computers, an embedded system is a special-purpose computer system which is designed to perform a small number of dedicated functions for a specific application (Sachitanand, 2002; Ali, 2004; Kamal, 2003). An embedded system might contain one or more programmable chips such as a microcontroller, microprocessor or digital signal processor (Pont, 2001). The word "embedded" indicates that the computer unit (e.g. microprocessor) is fully surrounded by the device it controls, and is invisible to the user of the device. An embedded system usually consists of hardware, software and perhaps mechanical or other components, and can be a small part of a larger system or machine (Barr, 1999; Kamal, 2003). In desktop computer systems, the user usually interacts with the software application through a set of highly-capable input / output devices such as keyboard, mouse, and coloured screen. In contrast, embedded systems have no such sophisticated interface devices: instead, they interact with the surrounding environment through a set of simple components such as switches, small keypads, light-emitting diodes (LEDs) and so on.

Historically, the first computer system, recognised as an embedded system, was the Apollo Guidance Computer developed in 1959 to control the Apollo spacecraft (Hall, 2000). The first successful commercial minicomputer was the PDP-8 produced by Digital Equipment Corporation in 1965 (Bell and Newell, 1971). In 1971, Intel released the first commercial single-chip microprocessor, the Intel 4004, which was primarily used in calculators and small systems (Bellis, 2007). Although this single-chip had replaced hardwired circuitry, external memory and support chips were still required

with the microprocessor unit until 1980s, when microcontrollers were developed to integrate all components of a microprocessor system into a single chip (Axelson, 1994; Bolton, 2000). Since then, many commercial companies have become involved in the development of embedded microcontrollers to meet the increasingly growing demand of modern technology, e.g. Atmel, Philips, Intel, Infineon, Texas Instruments, Microchip and Motorola. Examples of different processor platforms used nowadays in the design of embedded systems are: 8051 (Pont, 2001), ARM (ARM, 2001), PIC (Huang, 2004), MIPS (Chow, 1989), PowerPC (Chakravarty and Cannon, 1994), Atmel AVR (Kühnel, 2006), MPC555 (Bannatyne, 2004) and C16x (Siemens, 1996).

People in the 21st century may not realise the fact that without the emergence of embedded technology their life would have become harder. This is because most electrical devices people use nowadays are utilising embedded processors. Examples of such devices are: microwave ovens, TVs, VCRs, DVDs, mobile phones, MP3 players, washing machines, air conditions, handheld calculators, printers, digital watches, digital cameras, automatic teller machines (ATMs) and medical equipments (Barr, 1999; Bolton, 2000; Fisher *et al.*, 2004; Pop *et al.*, 2004; Kamal, 2003). Figure 1-1 shows examples of the wide ranging use of embedded systems in modern applications.

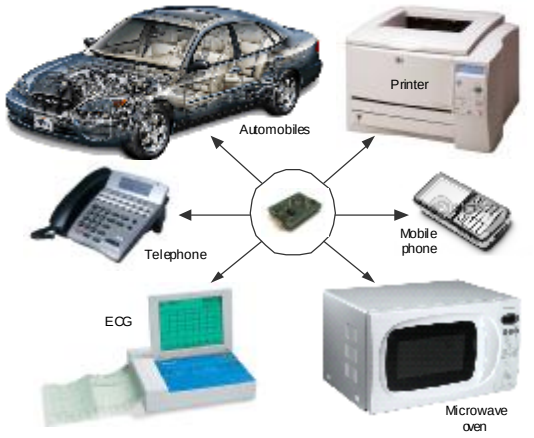


Figure 1-1: Examples of applications using embedded systems.

In a recent publication by ABB Corporate Research (2006), Christoffer Apneseth has reported that the need for embedded microprocessors arises mainly because general-purpose computers, like PCs, generally exceed the cost of the majority of products that utilise embedded systems, and are not capable of meeting the requirements that

embedded systems should have such as reliability[1], product-size limitation, real-time performance and power-consumption constraints (Apneseth, 2006).

1.3 Embedded systems market

Since embedded systems are ubiquitous, their market size today is estimated 100 times larger than the size of desktop market (Eggermont, 2002). This scale was expected to grow exponentially within the next ten years or so (Graaf *et al.*, 2003).

In a report developed by Ravi Krishnan (from BBC research group) in June 2005, the worldwide embedded systems market was estimated at $45.9 billion in 2004 and expected to grow at an average annual growth rate of 14% over the next five years to reach $88 billion by 2009. This total figure was broken down as follows: embedded software market is expected to grow from about $1.6 billion in 2004 to $3.5 billion by 2009 at an average annual growth rate of 16%, embedded hardware market is expected to grow from about $40.5 billion in 2004 to $78.7 billion by 2009 at an average annual growth rate of 14.2%, and embedded board revenues will increase from about $3.7 billion in 2004 to about $6 by 2009 at an average annual growth rate of 10% (see Krishnan, 2005). Figure 1-2 shows the evolution in the worldwide embedded markets from 2003 through to 2009.

[1] Reliability means that the system is able to provide the service to the user whenever requested (Sommerville, 2007).

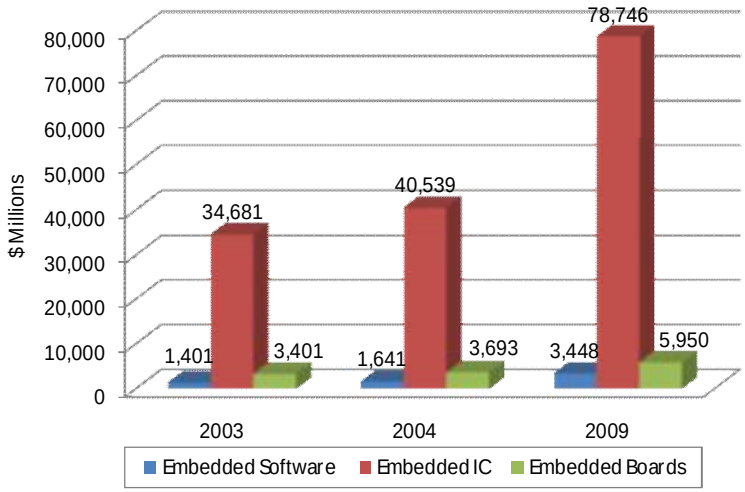


Figure 1-2: Global embedded systems market, 2003 – 2009 (Source: BBC Research Group).

1.4 The need for predictability in embedded systems

Besides the application types listed in Section 1.2, which can be viewed as "non-critical" systems, embedded technology has also been used to develop "safety-critical" systems in which failures can have very serious impacts on human safety. For example, incorrect operation of such systems may endanger human lives or cause catastrophic consequences. Safety-critical systems are typically used in the development of aerospace, automotive, railway, military and medical applications (Redmill, 1992; Profeta *et al.*, 1996; Storey, 1996; Konrad *et al.*, 2004).

The utilisation of embedded systems in safety-critical applications requires that the system should have *real-time* operations to achieve correct functionality and/or avoid any possibility for detrimental consequences. Real-time systems are computer systems which must react (respond) to events in the environment within limited time boundaries (Barr, 1999; Buttazzo, 2005). Real-time behaviour can only be achieved if the system is able to perform *predictable* and *deterministic* processing (Stankovic, 1988; Pont, 2001; Buttazzo, 2005; Phatrapornnant, 2007). More clearly, a given system is described as real-time if it is able to complete the execution of particular activities within specific time intervals. In another word, the system should guarantee that a particular set of activities (e.g. calculating the required throttle settings to control speed in an auto-driver system) will always be completed within (for example) 4 ms or at precisely 3 ms periods. In situations where the system is unable to meet these time constraints, then the

whole application is not simply slower than would be expected, it tends to be entirely useless (Pont, 2001). As a result, the correct behaviour of a real-time system depends on the logical correctness of the output results as well as the time at which these results are produced (Avrunin *et al.*, 1998; Kopetz, 1997).

Overall, real-time systems can be divided into two main classes: ***soft-real-time*** and ***hard-real-time*** systems (Buttazzo, 2005). In soft-real-time systems, timing constraints have to be "generally" met, and failure to do so may only result in reduced system performance but does not cause serious damages or jeopardise correct behaviour. In contrast, in hard-real-time systems such as those related to safety-critical applications, timing constraints must be "deterministically" met in order to achieve correct operations or avoid harmful consequences. For example, consider the Brake-by-Wire system designed for modern passenger cars (Hedenetz and Belschner, 1998), the brake actuators may be required to respond within a fixed amount of time after the brake pedal is pressed. If the system fails to respond within this time bound, then there could be a danger that the vehicle may not stop in time before crashing into another vehicle causing serious damage and possibly loss of passenger lives (Ayavoo, 2007). Another example is an aircraft auto-pilot system in which rapid reactions, involving (for example) rudder, elevator, aileron and engine settings, are necessarily required to keep the aircraft staying on its path. In situations where the system cannot (for example) adjust the rudder setting in millisecond time-scale, the plane may oscillate unpleasantly or even crash in more severe circumstances (Pont, 2001).

In such real-time embedded applications, it is important to predict the timing behaviour of the system to guarantee that the system will behave correctly and as a result the life of the people using the system will be saved. Many researchers indicate that, whilst the most important property of a desktop computing system is its speed, the most important property in a real-time computing system is predictability (Kontak, 1988; Stankovic, 1988; Halang and Stoyenko, 1990). This is clearly stated by Buttazzo (2005) as:
"… rather than being fast, a real-time computing system should be predictable"

Hence, predictability is the key characteristic in real-time embedded systems. Predictability can simply reflect the ability to determine, in advance, exactly what the system will do at every moment of time in which it is running and hence determine

whether the system is capable of meeting all its timing constraints. According to this definition, building an embedded application with highly-predictable system behaviour is, in most cases, a non straightforward process as will be discussed in the next section.

1.5 Challenges in building predictable embedded systems

Embedded systems engineering is viewed as a branch of systems engineering discipline where engineers are concerned with all aspects of the system development including hardware and software engineering. Therefore, activities such as specification, design, implementation, validation, deployment and maintenance will all be involved in the development of an embedded application.

A design of any system usually starts with ideas in people's mind. These ideas need to be captured in requirements specification documents that specify the basic functions and the desirable features of the system. The system design process then determines how these functions can be provided by the system components. Figure 1-3 illustrates the life cycle of a system development process.

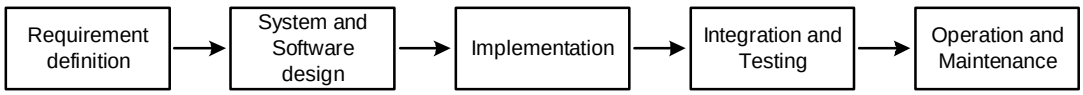


Figure 1-3: The system development life cycle (adapted from Sommerville, 2007).

For successful design, the system requirements have to be expressed and documented in a very clear way. Inevitably, there can be numerous ways in which the requirements for a simple system can be described: this may involve structured natural language, models or graphical representations, formal specification techniques, etc. (Sommerville, 2007).

Once the system requirements have been clearly defined and well documented, the first step in the design process is to design the overall system *architecture*. Architecture of a system basically represents an overview of the system components (i.e. sub-systems) and the interrelationships between these different components. Since embedded engineers are concerned with hardware and software design aspects of the system, they must decide on both the hardware and the software architectures of the intended design.

This thesis is mainly concerned with the software architectures for embedded designs. Douglass (2004) defines architecture as: *"the set of strategic design decisions that affect the structure, behaviour, or functionality of the system as a whole"*.

Clear documentation of the software architecture is paramount as it helps the developers consider key design aspects of the system early in the design process. Since they provide a high-level representation of the system, software architectures allow the developers to establish discussions about the system requirements and begin to predict how the system will operate after implementation (Sommerville, 2007). Determining the most appropriate architecture is a key requirement in the design and implementation processes of a given system. More specifically, embedded systems are often designed and implemented as a collection of processes (called *tasks*) which share the system resources and interact with the system and/or environment in which they operate. The various possible system architectures are then characterised in terms of these tasks. For example, if the tasks are invoked periodically under the control of timer, the system architecture may be described as *time-triggered* (Kopetz, 1997; Albert, 2004). Alternatively, if the tasks are invoked as a response to aperiodic external events, then the system architecture may be described as *event-triggered* (Nissanke, 1997; Albert, 2004). These are the two fundamental architectures used in the design of embedded systems. More details are provided later in Section 2.5.

Once the software architecture is identified, the process of implementing that architecture should take place. This can be achieved using a lower-level system representation such as an operating system or a *scheduler*. Scheduler is a very simple operating system for an embedded application (Pont, 2001). As with desktop operating systems, the scheduler has the responsibility to manage the computational and data resources in order to meet all temporal and functional requirements of the system (Mwelwa, 2006). A vital role of the scheduler is to organise the operation of the tasks running in the system, so as to guarantee that all timing requirements will be met. Building the scheduler would require a *scheduling algorithm* which simply provides the set of rules that determine the order in which the tasks will be executed by the scheduler during the system operating time. It is therefore the most important factor which influences predictability in the system, as it is responsible for satisfying timing and resource requirements (Buttazzo, 2005). However, the actual implementation of the

scheduling algorithm on the embedded microcontroller has an important role in determining the functional and temporal behaviour of the embedded system.

In view of all these different representations for a simple embedded design, the main challenge is to ensure that the various system representations are all matching up and the system would maintain its required behaviour whilst moving between representations. For example, Marwedel (2006) noted that when a system is modelled, each system model views a particular aspect of the system and it is not possible to ensure complete consistency between the various models; although some tools available nowadays can help perform partial consistency checking between the different models. As a consequence, ensuring predictability of the system, whilst translating between its various representations, would require further techniques to be applied at different stages in the development process. This is clearly underlined by Buttazzo (2005) as:
"… one safe way to achieve predictability is to investigate and employ new methodologies at every stage of the development of an application, from design to testing."

1.6 The focus of this thesis

The work described in this thesis seeks to address the process of translating between various possible representations of an embedded system and ensuring predictability during this process. However, the thesis is not attempting (by any mean) to match all system representations which cannot be possible in a single study. Instead, it is mainly concerned with translating between the high-level representation of the system, in terms of its scheduler, and the actual software implementation for that scheduler.

Given that a scheduling "algorithm" is the set of rules that, at every moment in the system run-time, determines which task must be allocated the resources to execute, the scheduler "implementation" is the process of transforming these rules into an executable source code (Sommerville, 2007; Koch, 1999). The source code can hence be viewed as the lower-level software representation of the system which practically dictates its functional and temporal behaviour.

Inevitably, there are many possible behaviour patterns one can get from a very simple software design, not least because of the various possible ways in which the source code for this design can be implemented. Therefore, it has been widely accepted that there is a 'one-to-many' mapping between a scheduling algorithm and its software implementations in practical real-time embedded designs (e.g. Baker and Shaw, 1989; Katcher *et al.*, 1993; Koch, 1999). As a consequence, any – even comparatively small – changes at the implementation stage of a scheduler can have a profound impact on the behaviour of the system which implements this scheduler (see Section 3.3 for more details).

Despite this, the topic of scheduler implementation is rarely considered in detail (Cho *et al.*, 2007). In a research conducted by Katcher and his colleges (Katcher *et al.*, 1993), it was argued that there is a wide gap between scheduling theory and its implementation in operating system kernels running on specific hardware platforms, where this gap must be bridged for meaningful validation of a real-time application. Katcher *et al.* have also noted that the implementation of a particular algorithm can introduce costs which must be taken into account when validating the timing properties of a real-time system.

The aim of this thesis is to bridge the gap between scheduling algorithms and practical scheduler implementations in real-time embedded systems. A main goal is to ensure that precise timing predictions made at the design stage of a system are not lost in the process of creating or maintaining a practical system implementation, thereby ensuring that the implemented scheduler matches the original design specifications and hence meets the user's requirements.

To address these issues, this study proposes the use of "scheduler test cases" as a way for recording and distinguishing the impact of different (scheduler) implementations on the behaviour of embedded systems. The Scheduler Test Case (STC) technique proposed is intended to allow those implementing a system to gain a full understanding of its characteristics by exploring the ways in which the various implementations of the system scheduler can be expected to behave under a range of both normal and abnormal operating conditions.

Note that the particular focus of this study is on resource-constrained, embedded systems which employ time-triggered co-operative (TTC) architectures. These architectures are reviewed in Section 2.8.3. However, the study also considers multi-processor embedded designs which are based on TTC algorithm and a Shared-Clock (S-C) scheduling protocol. These architectures are reviewed in Section 8.4.3.

1.7 Thesis contributions

The main contributions of this thesis are summarised as follows:

- A technique for employing a set of generic Scheduler Test Cases (STCs) is developed and implemented with the intention to facilitate a “black box” comparison between the behaviour of different Time-Triggered Co-operative (TTC) scheduler implementations in single-processor, resource-constrained embedded systems.
- An extension to the STC technique is proposed to allow assessing the behaviour of (distributed) multi-processor embedded designs implemented using a wide range of Shared-Clock (S-C) scheduling architectures built on the Controller Area Network (CAN) protocol.
- A set of standard forms (i.e. representative implementation classes) of TTC schedulers (for single-processor embedded systems) and TTC-SCC schedulers (for multi-processor embedded systems) are fully documented, classified and compared using a systematic approach.
- The development of a flexible (adaptive) TTC architecture that provides extremely predictable task scheduling is described and evaluated. This is aimed towards implementing a “perfect” TTC scheduler.
- A range of data coding techniques are developed to reduce transmission jitter and, hence, increase the predictability of multi-processor embedded networks that employ TTC-SCC scheduling protocols.

1.8 Thesis layout

The remaining chapters of this thesis are organised as follows:

Part B reviews previous work carried out in the areas concerned with in this thesis. It begins, in Chapter 2, by providing essential background material that is necessary to understand the work presented in the remaining chapters of the thesis. It mainly focuses on scheduling algorithms used in real-time embedded systems which have severe resource constraints and require highly-predictable system behaviour. Following this chapter, a more detailed literature review of the previous work in this area is provided. This includes the work on real-time scheduler implementations, with a particular focus on the TTC scheduling algorithm (Chapter 3) and possible ways to match scheduling algorithms and scheduler implementations using generic techniques (Chapter 4). By the end of Part B, the limitations in previous work to address the problems considered in this thesis are clarified.

Part C presents the work carried out in this project for single-processor embedded system implementations. It begins, in Chapter 5, by reviewing a wide range of representative implementation classes for TTC scheduling algorithm. Chapter 6 then describes the STC technique developed in this project to document (and assess) the various TTC scheduler implementations. In Chapter 7, the STC technique is applied to the reviewed TTC scheduler implementations and the output results are presented and analysed. The experimental methodology used to obtain the results is also outlined in this chapter.

Part D considers the work carried out in this project for multi-processor embedded system implementations. This study begins, in Chapter 8, by reviewing various network and scheduling protocols used to implement multi-processor embedded systems which have severe resource constraints and highly-predictable behaviour requirements. The focus in this chapter will particularly be on systems employing Controller Area Network (CAN) communication protocol and Shared-Clock (S-C) scheduling protocol. The next three chapters then follow the same layout as in the single-processor study. In particular, Chapter 9 reviews a wide range of representative implementation classes for

S-C scheduling protocol as implemented with TTC algorithm on CAN network (the resulting system will be referred to as TTC-SCC scheduler). Chapter 10 describes a possible modification to the STC technique to allow documenting (and assessing) the various TTC-SCC scheduler implementations. In Chapter 11, the modified STC technique is applied to the reviewed TTC-SCC scheduler implementations and the output results are presented and analysed. The methodology used to obtain the results is outlined at the beginning of this chapter.

Part E contains the discussion and conclusions of the thesis. In particular, Chapter 12 summarises the work presented in the previous chapters and discusses the overall findings of the project. Finally, Chapter 13 draws the overall thesis conclusions and suggests some work for future research projects.

Part F contains Appendices which provide supplementary materials and summarise additional work which has been carried out during the course of this project, but is indirectly related to the studies presented in the thesis chapters.

1.9 Conclusions

This introductory chapter has discussed the overall theme of the work described in this PhD thesis. It provided an introduction to embedded systems and discussed the challenges involved in the process of creating a predictable embedded application.

The discussions indicated that, despite the importance of scheduling algorithms in managing the operation of real-time embedded systems, scheduler implementations have a major role in determining the actual run-time behaviour of the system: however, there is still a wide gap between scheduling algorithms and their practical implementations which must be addressed to achieve correct validations of embedded systems.

Based on these discussions, the main goal and key contributions of this thesis were stated, and the layout for the remaining chapters provided.

PART B:

LITERATURE REVIEW

Chapter 2

Real-time scheduling algorithms

2.1 Introduction

As previously noted, this thesis is mainly concerned with the process of translating between scheduling algorithms and scheduler implementations in practical real-time embedded systems which employ time-triggered software architectures. This chapter introduces the concepts of scheduling and discusses scheduling algorithms which are widely used in the design of real-time, resource-constrained embedded systems when highly-predictable system behaviour is a key requirement. The chapter begins by providing some essential background material and definitions.

2.2 Tasks

The most important software entity of the real-time embedded system is the *process* or *task* which is a computation that is executed by the CPU in a sequential manner (Buttazzo, 2005). Most embedded systems are assembled from collections of tasks. For example, complex systems (such as aircraft control) may have hundreds of tasks, possibly distributed across a number of CPUs: see Section 2.11 . However, the tasks executed by a single CPU may still need to exchange data between them and access shared resources, e.g. ports, serial interfaces, digital-to-analogue converters, and so forth. The interaction between the various tasks depends on the method of scheduling that is employed: this is discussed further shortly.

Real-time tasks are divided into three main categories:

- **Periodic tasks:** tasks implemented as functions which are called at regular intervals (e.g. every millisecond or every 100 milliseconds) during some or all of the time that the system is active. Periodic tasks usually have critical timing constraints which must be met precisely (Cottet, 2002). Figure 2-1 shows an example of a periodic task. Note that the task is ready at a_i, must complete its execution before d_i (where $i = 1, 2, 3, \ldots, n$) and is called every T interval.

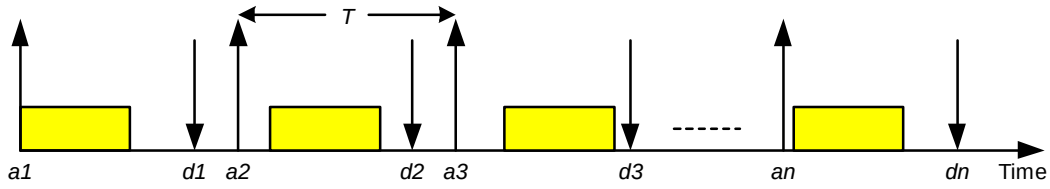


Figure 2-1: Sequence for a periodic task. The figure is adapted from (Buttazzo, 2005).

- **Aperiodic tasks:** tasks implemented as functions which may be activated if a particular event takes place. For example, an aperiodic task might be activated when a switch is pressed, or a character is received over a serial connection. Timing constraints for aperiodic tasks can be less critical than those for periodic tasks (Cottet, 2002). Figure 2-2 shows an example of an aperiodic task.

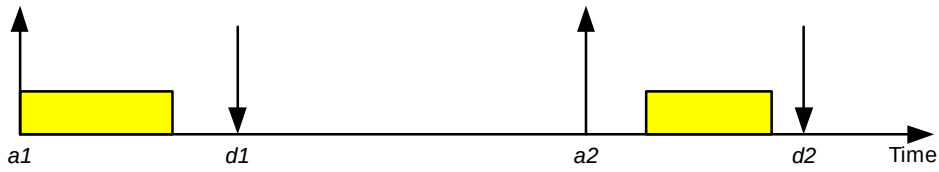


Figure 2-2: Sequence for an aperiodic task. The figure is adapted from (Buttazzo, 2005).

- **Sporadic tasks:** tasks implemented as functions which are called repeatedly at variable intervals. However, the minimum interval between any two successive occurrences of the sporadic task is known. Figure 2-3 shows an example of a sporadic task. Note that the task is called at variable periods with the minimum value of T_m.

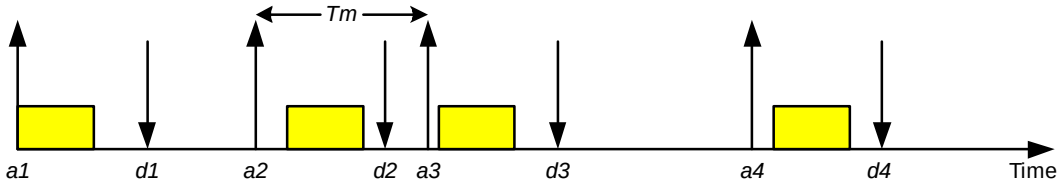


Figure 2-3: Sequence for a sporadic task.

2.3 Timing constraints

For any type of tasks running in a real-time system, timing constraints are often a key concern. Tasks can be divided – according to the implications of missing their timing constraints – into two main classes: *soft* and *hard*. In particular, a task is said to be soft if meeting its timing constraints is desirable for performance, but missing these constraints does not affect the correctness of the system behaviour. In contrast, a task is said to be hard if missing its timing constraints can result in harmful consequences or misbehaviour of the system (Buttazzo, 2005).

The typical timing constraints associated with each task in real-time systems are:

- **Release time**: the time after which a task can start its execution. This parameter is sometimes called *request* time, *ready* time, or *arrival* time.
- **Deadline**: the time before which a task must complete its execution.

Some other timing parameters, which are used to characterise tasks in real-time systems, are:

- **Start time**: the time at which a task starts its execution.
- **Completion time:** the time at which a task completes its execution. This parameter is also called *finishing* time.
- **Execution time**: the time taken by the processor to execute a task without interruption. This parameter is also called *computation* time.
- **Lateness**: the delay between the deadline and the completion time. A task is considered late if its lateness value is positive.

2.4 Jitter

Jitter is a term which describes variations in the timing of activities (Wavecrest, 2001). For some periodic tasks, such variations are more important than the absolute deadline. For example, suppose that some activity should occur at times:
t = {1.0 ms, 2.0 ms, 3.0 ms, 4.0 ms, 5.0 ms, 6.0 ms, 7.0 ms, …}.

Suppose, instead, that the activity occurs at times:
t = {11.0 ms, 12.0 ms, 13.0 ms, 14.0 ms, 15.0 ms, 16.0 ms, 17.0 ms, …}.

In this case, the activity has been delayed (by 10 ms). For some applications – such as data, speech or music playback (for example) – this delay may make no measurable difference to the user of the system. However, suppose that – for a data playback system – same activities were to occur as follows:
t = {1.0 ms, 2.1 ms, 3.0 ms, 3.9 ms, 5.0 ms, 6.1 ms, 7.0 ms, …}.

In this case, there is a variation in the activity timing which is referred to as jitter. In real-time embedded systems, various system activities are identified as tasks that need to be performed at precise timing without delays or, more importantly, jitter.

The present work is concerned with implementing highly-predictable embedded systems. As previously introduced, predictability is one of the most important objectives of real-time embedded systems which can simply be defined as the ability to determine, in advance, exactly what the system will do at every moment of time in which it is running. One way in which predictable behaviour manifests itself is in low levels of task jitter. As jitter is used in this study as a way of assessing timing behaviour, previous work in this area is briefly reviewed in this section and later in Section 2.9. This section, in particular, discusses the impact of jitter on the performance of real-time embedded systems.

Jitter is a key timing parameter that can have detrimental impacts on the performance of many applications, particularly those involving period sampling and/or data generation (e.g. data acquisition, data playback and control systems: see Torngren, 1998). The need for high-speed systems has enforced the embedded processors to operate in multi-gigahertz frequency range, and reliable operation of such high-frequency systems would require substantial understanding of timing jitter characteristics (Ong *et al.*, 2004). For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly, Jerri (1977) discusses the serious impact of jitter on applications such as spectrum analysis and filtering. Also, in control systems, jitter can greatly degrade the performance by

varying the sampling period (Torngren, 1998; Marti *et al.*, 2001b). The serious impacts of jitter on a wide range of applications have been discussed in a number of previous studies (e.g. Jerri, 1977; Hong, 1995; Stothert, 1998; Gulliver and Ghinea, 2007; Phatrapornnant, 2007). For example, Gulliver and Ghinea (2007) exemplify that applications – such as distributed multimedia communications – are highly sensitive to jitter, where the presence of even low amounts of jitter may result in a severe degradation in perceptual video quality.

2.5 Software architectures

Embedded systems are composed of hardware and software components. The success of an embedded design, thus, depends on the right selection of the hardware platform(s) as well as the software environment used in conjunction with the hardware. The selection of hardware and software architectures of an application must take place at early stages in the development process (typically at the design phase). Hardware architecture relates mainly to the type of the processor (or microcontroller) platform(s) used and the structure of the various hardware components that are comprised in the system: see Mwelwa (2006) for further discussion about hardware architectures for embedded systems.

Provided that the hardware architecture is decided, an embedded application requires an appropriate form of software architecture to be implemented. To determine the most appropriate choice for software architecture in a particular system, this condition must be fulfilled (Locke, 1992):
"The [software] architecture must be capable of providing a provable prediction of the ability of the application design to meet all of its time constraints."

Since embedded systems are usually implemented as collections of real-time tasks, the various possible system architectures may then be determined by the characteristics of these tasks. In general, there are two main software architectures which are typically used in the design of embedded systems:

- **Event-triggered (ET):** tasks are invoked as a response to aperiodic events. In this case, the system takes no account of time: instead, the system is controlled purely by the response to external events, typically represented by interrupts which can

arrive at anytime (Bannatyne, 1998; Kopetz, 1991b). Generally, ET solution is recommended for applications in which sporadic data messages (with unknown request times) are exchanged in the system (Hsieh and Hsu, 2005).

- **Time-triggered (TT):** tasks are invoked periodically at specific time intervals which are known in advance. The system is usually driven by a global clock which is linked to a hardware timer that overflows at specific time instants to generate periodic interrupts (Bennett, 1994). In distributed systems, where multi-processor hardware architecture is used, the global clock is distributed across the network (via the communication medium) to synchronise the local time base of all processors. In such architectures, time-triggering mechanism is based on time-division multiple access (TDMA) in which each processor-node is allocated a periodic time slot to broadcast its periodic messages (Kopetz, 1991b). TT solution can suit many control applications where the data messages exchanged in the system are periodic (Kopetz, 1997).

To better explain the differences between the TT and ET software architectures, consider the following example (Pont, 2001). A hospital doctor is required to look after a group of seriously ill patients overnight, with a support of some nursing staff. With ET solution, the doctor might arrange to go to sleep and only if a significant problem occurs with one patient a nurse can waken him up to deal with the problem. An alternative solution to this is TT in which the doctor might set his alarm to ring every hour. When the alarm rings, the doctor wakes up and begins to check the status of all patients in sequence before going to sleep for the rest of the hour.

Many researchers argue that ET architectures are highly flexible and can provide high resource efficiency (Obermaisser, 2004; Locke, 1992). However, ET architectures allow several interrupts to arrive at the same time, where these interrupts might indicate (for example) that two different faults have been detected at the same time. Inevitably, dealing with an occurrence of several events at the same time will increase the system complexity and reduce the ability to predict the behaviour of the ET system (Scheler and Schröder-Preikschat, 2006). In more severe circumstances, the system may fail completely if it is heavily loaded with events that occur at once (Marti, 2002). In

contrast, using TT architectures helps to ensure that only a single event is handled at a time and therefore the behaviour of the system can be highly-predictable.

To make this point clearer, reconsider the hospital doctor example. With a TT solution, where the doctor visits all patients at hourly intervals, each patient will be checked and appropriate treatment is hence arranged before serious problems arise. With this process, the doctor's workload is spread out equally throughout the night making all patients survive without difficulty. On the contrary, using ET solution may cause serious problems. For example, assume that a minor problem occurs with one patient while the doctor is asleep and the nursing staff decide not to waken the doctor but to solve the problem themselves. Few hours later, several patients have minor problems after which the nurses decide to wake the doctor up to look at those problems. Once the doctor sees the patients, he realises that some of them have severe complications and they need surgery. One implication of this process is that before the doctor can deal with the first patient, the second one gets very close to death, and so on.

Since highly-predictable system behaviour is an important design requirement for many embedded systems, TT software architectures have become the subject of considerable attention (e.g. see Kopetz, 1997). In particular, it has been widely accepted that TT architectures are a good match for many safety-critical applications, since they can help to improve the overall safety and reliability (Allworth, 1981; Storey, 1996; Nissanke, 1997; Bates; 2000; Obermaisser, 2004). For example, Time-Triggered Group (TTG) – established by Airbus, Audi, Delphi, Honeywell, PSA Peugeot Citroën, Renault and TTTech companies – promotes cross-industry technologies for a TT solution on many safety-critical industries including aerospace, railway and automotive where safety requirements must be satisfied at low cost (TTA-Group, 2007). In the automotive industry, as an example, TT architectures have been recently accepted as a generic solution for highly dependable systems such as X-by-Wire systems (see Ayavoo, 2006; Mwelwa, 2006). The main reason why the TT approaches are preferred in such applications is that they result in systems which have very predictable and deterministic behaviour. Liu (2000) highlights that TT systems are easy to validate, test, and certify because the times related to the tasks are deterministic.

Moreover, it was pointed out that fault tolerance (which requires a proper synchronism of the redundant component) can be easily achieved with TT systems without requiring additional CPU overhead (Scheler and Schröder-Preikschat, 2006). Detailed comparisons between the TT and ET concepts were performed by Kopetz (1991a and 1991b), Albert (2004) and Scheler and Schröder-Preikschat (2006). Scheler and Schröder-Preikschat (2006) went further to outline a method which helps describing the real-time system independent of its architecture and therefore eases the process of migrating between TT and ET architectures later in the development process. Before wrapping up this discussion, it should be noted that in some applications, a mix of TT and ET system architectures can be an optimal design solution (for more details see Pop *et al.*, 2002).

Over recent years, the ESL researchers have considered various ways in which TT architectures can be employed in low-cost embedded systems (see ESL, 2008 for the full list of ESL publications). The techniques described in these studies have involved the development of software for industry-standard commercial-off-the-shelf (COTS) hardware platforms, such as the 8051 microcontroller (Pont, 2001), ARM processor (Pont and Mwelwa, 2003) or PC platform (Pont *et al.*, 2003). For example, Pont (2001) provides a wide range of design pattern[2] collections to support the software development of embedded systems which are based on TT architectures. Recently, in (Mwelwa, 2006), a tool to support pattern-based code generation of TT embedded systems is developed and assessed. More recently, Phatrapornnant (2007) looked at ways in which dynamic voltage scaling (DVS) techniques – for reducing system power consumption – can be incorporated in simple TT scheduling algorithms.

Nonetheless, previous work in this area has also focused on the development of multi-processor designs. For such designs, it has been demonstrated that a “Shared-Clock” (S-

[2] Patterns describe a solution to a frequently recurring design problem that can be applied in different contexts. The first “pattern language” was described by Christopher Alexander, an architect who intended to link between architectural problems and good design solutions (Pont, 2001). Software patterns are hence used in software systems to facilitate design reuse by providing developers with previously successful design solutions (see Mwelwa, 2006 for more information).

C) architecture provides a simple and low-cost software framework for TT systems without requiring specialised hardware (Pont, 2001). S-C protocols are further discussed in Chapter 8.

2.6 Schedulers

As previously noted, most embedded systems involve several tasks that share the system resources and communicate with one another and/or the environment in which they operate. For many projects, a key challenge is to work out how to schedule tasks so that they can meet their timing constraints. This process requires an appropriate form of *scheduler*[3]. A scheduler can be viewed as a very simple operating system which calls tasks periodically (or aperiodically) during the system operating time. Moreover, as with desktop operating systems, a scheduler has the responsibility to manage the computational and data resources in order to meet all temporal and functional requirements of the system (Mwelwa, 2006).

According to the nature of the operating tasks, any real-time scheduler must fall under one of the following types of scheduling policies:

- **Pre-emptive scheduling:** where a multi-tasking process is allowed. In more details, a task with higher priority is allowed to pre-empt (i.e. interrupt) any lower priority task that is currently running. The lower priority task will resume once the higher priority task finishes executing. For example, suppose that – over a particular period of time – a system needs to execute four tasks (Task A, Task B, Task C, Task D) as illustrated in Figure 2-4.

[3] Note that schedulers represent the core components of "Real-Time Operating System" (RTOS) kernels. Examples of commercial RTOSs which are used nowadays are: VxWorks (from Wind River), Lynx (from LynxWorks), RTLinux (from FSMLabs), eCos (from Red Hat), and QNX (from QNX Software Systems). Most of these operating systems require large amount of computational and memory resources which are not readily available in low-cost microcontrollers like the ones considered in this study.

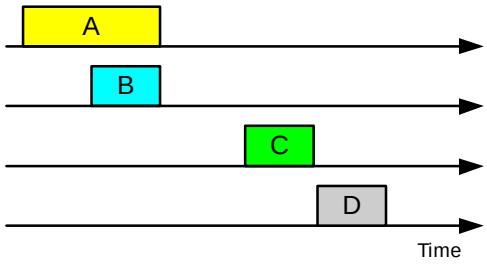


Figure 2-4: A schematic representation of four tasks which need to be scheduled for execution on a single-processor embedded system.

Assuming a single-processor system is used, Task C and Task D can run as required where Task B is due to execute before Task A is complete. Since no more than one task can run at the same time on a single-processor, Task A or Task B has to relinquish control of the CPU. In pre-emptive scheduling, a higher priority might be assigned to Task B with the consequence that – when Task B is due to run – Task A will be interrupted, Task B will run, and Task A will then resume and complete (Figure 2-5).

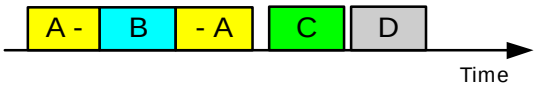


Figure 2-5: Pre-emptive scheduling of Task A and Task B in the system shown in Figure 2-4: Task B, here, is assigned a higher priority.

- **Co-operative (or "non-pre-emptive") scheduling:** where only a single-tasking process is allowed. In more details, if a higher priority task is ready to run while a lower priority task is running, the former task cannot be released until the latter one completes its execution. For example, assume the same set of tasks illustrated in Figure 2-4. In the simplest solution, Task A and Task B can be scheduled co-operatively. In these circumstances, the task which is currently using the CPU is implicitly assigned a high priority: any other task must therefore wait until this task relinquishes control before it can execute. In this case, Task A will complete and then Task B will be executed (Figure 2-6).

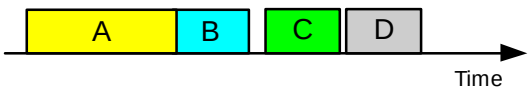


Figure 2-6: Co-operative scheduling of Task A and Task B in the system shown in Figure 2-4.

- **Hybrid scheduling:** where a limited, but efficient, multi-tasking capabilities are provided (Pont, 2001). That is, only one task in the whole system is set to be pre-emptive (this task is best viewed as "highest-priority" task), while other tasks are running co-operatively (Figure 2-7). In the example shown in the figure, suppose that Task B is a short task which has to execute immediately when it arrives. In this case, Task B is set to be pre-emptive so that it acquires the CPU control to execute whenever it arrives and whether (or not) other task is running.

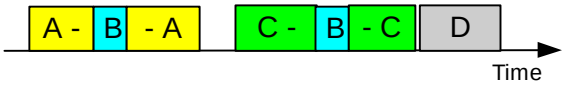


Figure 2-7: Hybrid scheduling of four-tasks: Task B is set to be pre-emptive, where Task A, Task C and Task D run co-operatively.

Overall, when comparing co-operative with pre-emptive schedulers, many researchers have argued that co-operative schedulers have many desirable features, particularly for use in safety-related systems (Allworth, 1981; Ward, 1991; Nissanke, 1997; Bates, 2000; Pont, 2001). For example, Bates (2000) identified the following four advantages of co-operative scheduling over pre-emptive alternatives:

- The scheduler is simpler.
- The overheads are reduced.
- Testing is easier.
- Certification authorities tend to support this form of scheduling.

Similarly, Nissanke (1997) noted: *"[Pre-emptive] schedules carry greater runtime overheads because of the need for context switching - storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overheads. Other advantages of co-operative algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for guaranteeing exclusive access to any shared resource or data."*

Many researchers still, however, believe that pre-emptive approaches are more effective than co-operative alternatives (Allworth, 1981; Cooling, 1991; Bannet, 1994). This can

be due to different reasons. As in (Pont, 2001), one of the reasons why pre-emptive approaches are more widely discussed and considered is because of confusion over the options available. Pont gave an example that the basic cyclic scheduling, which is often discussed by many as an alternative to pre-emptive, is not a representative of the wide range of co-operative scheduling architectures that are available.

Moreover, one of the main issues that concern people about the reliability of co-operative scheduling is that long tasks can have a negative impact on the responsiveness of the system. This is clearly underlined by Allworth (1981):
"[The] main drawback with this co-operative approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired."

However, in many practical embedded systems, the process (task) duration is extremely short. For example, calculations of one of the very complicated algorithms, the proportional integral differential (PID) controller, can be carried out on the most basic (8-bit) 8051 microcontroller in around 0.4 ms: this imposes insignificant processor load in most systems – including flight control – where 10 ms sampling rate is adequate (Pont, 2001). Pont has also commented that if the system is designed to run long tasks, *"this is often because the developer is unaware of some simple techniques that can be used to break down these tasks in an appropriate way and – in effect – convert long tasks called infrequently into short tasks called frequently"*: some of these techniques are introduced and discussed in Pont (2001).

Moreover, if the performance of the system is seen slightly poor, it is often advised to update the microcontroller hardware rather than to use a more complex software architecture. However, if changing the task design or microcontroller hardware does not provide the level of performance which is desired for a particular application, then more than one microcontroller can be used. In such cases, long tasks can be easily moved to another processor, allowing the host processor to respond rapidly to other events as required. Further discussions about multiple-processor designs, in this thesis, are provided in Chapter 8.

Please note that the very wide use of pre-emptive schedulers can simply be resulted from a poor understanding and, hence, undervaluation of the co-operative schedulers. For example, a co-operative scheduler can be easily constructed using only a few hundred lines of highly portable code written in a high-level programming language (such as 'C'), while the resulting system is highly-predictable (Pont, 2001).

It is also important to understand that sometimes pre-emptive schedulers are more widely used in RTOSs due to commercial reasons. For example, companies may have commercial benefits from using pre-emptive environments. Consequently, as the complexity of these environments increases, the code size will significantly increase making 'in-house' constructions of such environments too complicated. Such complexity factors lead to the sale of commercial RTOS products at high prices (Pont, 2001). Therefore, further academic research has been conducted in this area to explore alternative solutions. For example, over the last few years, the ESL researchers have considered various ways in which simple, highly-predictable, non-pre-emptive (co-operative) schedulers can be implemented in low-cost embedded systems (see Section 2.8.3).

2.7 Schedule design

When any type of scheduler is to be employed in a real-time system, a number of key scheduler parameters must be determined: e.g. the task order, initial delay (i.e. phase or offset) and period of each task. The aim with this design process is to ensure that all tasks are able to meet their deadlines and that the simplest scheduler architecture is employed (Gendy and Pont, 2008). It is so important to realise that inappropriate choices of such design parameters may mean that a given task set cannot be scheduled at all.

Automatic generation of schedules and schedulers is less common than general-purpose code generation, but work has been done in this area too. Examples of methods used in automatic schedule / scheduler generation include: simulated annealing (Tindell *et al.*, 1992), constraint programming heuristics (Ekelin and Jonsson, 2001), branch and bound algorithm (Xu and Parnas, 1990) and genetic algorithm (Sandström and Norström, 2002): for mode details see Gendy and Pont (2008). The work described in the previous

studies is relevant to a discussion about tool support for scheduler design. However, none of this previous work relates directly to time-triggered architectures that form the key focus of this study. In fact, previous studies have tended to focus on "conventional" RTOSs (e.g. VxWorks: Sandström and Norström, 2002). Such operating systems greatly exceed the resource requirements available in the types of processors considered in this study.

Two recent studies within the ESL group (Gendy *et al.*, 2007; Gendy and Pont, 2008) have explored scheduler design for time-triggered (co-operative and hybrid) architectures. The approach involves a "best characteristics first" search intended to identify a good (but not necessarily optimal) set of scheduler parameters while maintaining low levels of system power consumption.

2.8 Scheduling algorithms

2.8.1 Introduction

A key component of the scheduler is the *scheduling algorithm* which basically determines the order in which the tasks will be executed by the scheduler (Buttazzo, 2005). More specifically, a scheduling algorithm is the set of rules that, at every instant while the system is running, determines which task must be allocated the resources to execute.

Developers of embedded systems have proposed various scheduling algorithms that can be used to handle tasks in real-time applications. The selection of appropriate scheduling algorithm for a set of tasks is based upon the capability of the algorithm to satisfy all timing constraints of the tasks: where these constraints are derived from the application requirements. Examples of common scheduling algorithms are: Cyclic Executive (Locke, 1992), Rate Monotonic (Liu and Layland, 1973), Earliest-Deadline-First (Liu and Layland, 1973; Liu, 2000), Least-Laxity-First (Mok, 1983), Deadline Monotonic (Leung, 1982) and Shared-Clock (Pont, 2001) schedulers (see Rao *et al.*, 2008 for a simple classification of scheduling algorithms).

This section outlines two key examples of scheduling algorithms that are widely used in the design of real-time embedded systems when highly-predictable system behaviour is an essential requirement: these are the rate monotonic and a form of cyclic executive.

2.8.2 Rate monotonic (RM) scheduler

The rate-monotonic (RM) (Liu and Layland, 1973) is a well-known fixed-priority scheduling algorithm. RM is a time-triggered, pre-emptive algorithm in which task priorities are fixed and inversely proportional to their periods. Liu and Layland demonstrated that – with a set of *n* tasks – every task in the RM scheduler will meet its deadline if the total CPU utilisation is less than or equal to *n(2^(1/n)-1)*; all tasks in the system are independent of one another; the deadline for each task is equal to its period; the worst-case execution time of all tasks is known; and context switching time can be ignored[4] (see Liu and Layland, 1973). As a result, highly-predictable system behaviour can be achieved when RM algorithm is employed in hard real-time systems. This is simply because it provides a guarantee that all tasks will complete execution before their deadlines, if all conditions are met. A key advantage of RM, as observed by Locke (1992) and Bate (1998), is that it is so flexible and as a result of its simple schedulability definition, the only process required to schedule a new task is the recalculation of the CPU utilisation value.

However, as with most pre-emptive schedulers, the RM algorithm may carry large scheduling overhead due to the context switching required to store (and retrieve) the partially computed results (Locke, 1992; Nissanke, 1997). Wendorf (1988) also emphasised that, despite many advantages, fixed-priority schedulers may not perform well (or even fail to meet system requirements) under overload conditions. Moreover, even if all schedulability conditions are met, RM only provides a guarantee that each task will execute once at some point in its execution "slots" and does not guarantee any more precise control over timing behaviour. For example, when a higher priority task pre-empts a lower priority task, this may cause a delay in the output results expected

[4] In RM algorithm, if the number of tasks *n* goes to infinity, then the task set is schedulable if the total CPU utilisation does not exceed 69% (Liu and Layland, 1973).

from the latter task or unwanted jitter in the release time of this task. Lin and Herkert (1996) note that in RM scheduler:
"Although every task must be completed before the end of each period, there is no constraint on when in the period it must be executed. This is because the completion time of a lower priority task in each period depends on if and when some higher priority tasks may arrive (…). Therefore, task execution jitters are unavoidable using RM."

Another problem with such a scheduling algorithm is that a high priority task can be entirely blocked by a low priority task if the former requires access to a shared resource (e.g. analogue-to-digital converter, serial port, etc) while the latter is using it, causing an inversion in the task priorities. Such "priority inversion" consequently produces very high levels of task jitter and hence affects system predictability. Although priority inversion problem can be solved using different techniques, e.g. Priority Ceiling Protocol, (Sha *et al.*, 1990), the impact of such techniques on jitter is not always easy to predict (Phatrapornnant, 2007).

An alternative to this scheduling algorithm is the time-triggered co-operative (TTC) scheduler.

2.8.3 Time-triggered co-operative (TTC) scheduler: Cyclic executive

A key defining characteristic of a time-triggered (TT) system is that it can be expected to have highly-predictable patterns of behaviour. This means that when a computer system has a time-triggered architecture, it can be determined in advance – before the system begins executing – exactly what the system will do at every moment of time while the system is operating. Based on this definition, completely defined TT behaviour is – of course – difficult to achieve in practice. Nonetheless, approximations of this model have been found to be useful in a great many practical systems. The closest approximation of a "perfect" TT architecture which is in widespread use involves a collection of periodic tasks which operate co-operatively (or "non-pre-emptively"). Such a time-triggered co-operative (TTC) architecture has sometimes been described as a cyclic executive (e.g. Baker and Shaw, 1989; Locke, 1992).

According to Baker and Shaw (1989), the cyclic executive scheduler is designed to execute tasks in a sequential order that is defined prior to system activation; the number

of tasks is fixed; each task is allocated an execution slot (called a *minor cycle* or a *frame*) during which the task executes; the task – once interleaved by the scheduler – can execute until completion without interruption from other tasks; all tasks are periodic and the deadline of each task is equal to its period; the worst-case execution time of all tasks is known; there is no context switching between tasks; and tasks are scheduled in a repetitive cycle called *major cycle*. The major cycle can be defined as the time period during which each task in the scheduler executes – at least – onece and before the whole task execution pattern is repeated. This is numerically calculated as the lowest common multiple (LCM) of the periods of the scheduled tasks (Baker and Shaw, 1989; Xu and Parnas, 1993). Koch (1999) emphasised that cyclic executive is a "proof-by-construction" scheme in which no schedulability analysis is required prior to system construction.

Figure 2-8 illustrates the (time-triggered) cyclic executive model for a simple set of four periodic tasks. Note that the final task in the task-group (i.e. Task D) must complete execution before the arrival of the next timer interrupt which launches a new (major) execution cycle.

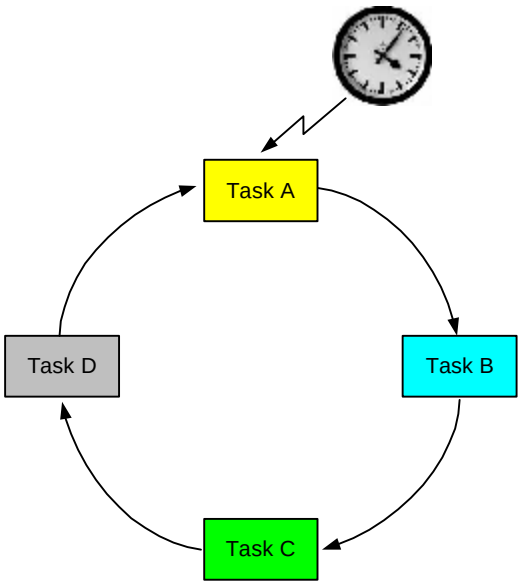


Figure 2-8: A time-triggered cyclic executive model for a set of four periodic tasks (adapted from Kalinsky, 2001).

In the example shown, each task is executed only once during the whole major cycle which is, in this case, made up of four minor cycles. Note that the task periods may not

always be identical as in the example shown in Figure 2-8. When task periods vary, the scheduler should define a sequence in which each task is repeated sufficiently to meet its frequency requirement (Locke, 1992).

Figure 2-9 shows the general structure of the time-triggered cyclic executive (i.e. time-triggered co-operative) scheduler. In the example shown in this figure, the scheduler has a minor cycle of 10 ms, period values of 20, 10 and 40 ms for the tasks A, B and C, respectively. The LCM of these periods is 40 ms, therefore the length of the major cycle in which all tasks will be executed periodically is 40 ms. It is suggested that the minor cycle of the scheduler (which is also referred to as the *tick interval*: see Pont, 2001) can be set equal to or less than the greatest common divisor value of all task periods (Phatrapornnant, 2007). In the example shown in Figure 2-9, this value is equal to 10 ms. In practice, the minor cycle is driven by a periodic interrupt generated by the overflow of an on-chip hardware timer or by the arrival of events in the external environment (Locke, 1992; Pont, 2001). The vertical arrows in the figure represent the points at which minor cycles (ticks) start.

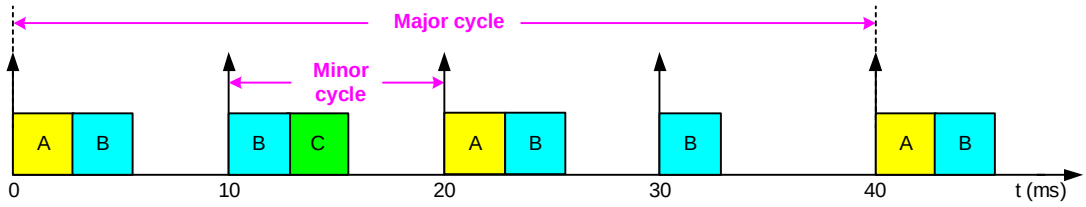


Figure 2-9: A general structure of the time-triggered co-operative (TTC) scheduler.

Overall, TTC schedulers have many advantages. A key recognisable advantage is its simplicity (Baker and Shaw, 1989; Liu, 2000; Pont, 2001). Furthermore, since pre-emption is not allowed, mechanisms for context switching are, hence, not required and, as a consequence, the run-time overhead of a TTC scheduler can be kept very low (Locke, 1992; Buttazzo, 2005). Also, developing TTC schedulers needs no concern about protecting the integrity of shared data structures or shared resources because, at a time, only one task in the whole system can exclusively use the resources and the next due task cannot begin its execution until the running task is completed (Baker and Shaw, 1989; Locke, 1992).

Since all tasks are run regularly according to their predefined order in a deterministic manner, the TTC schedulers demonstrate very low levels of task jitter (Locke, 1992; Bate, 1998; Buttazzo, 2005) and can maintain their low-jitter characteristics even when complex techniques, such as dynamic voltage scaling (DVS), are employed to reduce system power consumption (Phatrapornnant and Pont, 2006). Therefore, as would be expected (and unlike RM designs, for example), systems with TTC architectures can have highly-predictable timing behaviour (Baker and Shaw, 1989; Locke, 1992). Locke (1992) underlines that with cyclic executive systems *"it is possible to predict the entire future history of the state of the machine, once the start time of the system is determined (usually at power-on). Thus, assuming this future history meets the response requirements generated by the external environment in which the system is to be used, it is clear that all response requirements will be met. Thus it fulfils the basic requirements of a hard real time system."*

Provided that an appropriate implementation is used, TTC architectures can be a good match for a wide range of low-cost embedded applications. For example, previous studies have described – in detail – how these techniques can be applied in various automotive applications (e.g. Ayavoo *et al.*, 2006; Ayavoo, 2006), a wireless (ECG) monitoring system (Phatrapornnant and Pont, 2004; Phatrapornnant, 2007), various control applications (e.g. Edwards *et al.*, 2004; Key *et al.*, 2004; Short and Pont, 2008), and in data acquisition systems, washing-machine control and monitoring of liquid flow rates (Pont, 2002). Outside the ESL group, Nghiem *et al.* (2006) described an implementation of PID controller using TTC scheduling algorithm and illustrated how such architecture can help increase the overall system performance as compared with alternative implementation methods.

However, TTC architectures have some shortcomings. For example, many researchers argue that running tasks without pre-emption may cause other tasks to wait for sometime and hence miss their deadlines. However, the availability of high-speed, COTS microcontrollers nowadays helps to reduce the effect of this problem and, as processor speeds continue to increase, non-pre-emptive scheduling approaches are expected to gain more popularity in the future (Baruah, 2006).

Another issue with TTC systems is that the task schedule is usually calculated based on estimates of Worst Case Execution Time (WCET) of the running tasks. If such estimates prove to be incorrect, this may have a serious impact on the system behaviour (Buttazzo, 2005). Further discussions on possible solutions to this problem are provided later in Section 2.10.

One recognised disadvantage of using TTC schedulers is the lack of flexibility (Locke, 1992; Bate, 1998). This is simply because TTC is usually viewed as 'table-driven' static scheduler (Baker and Shaw, 1989) which means that any modification or addition of a new functionality, during any stage of the system development process, may need an entirely new schedule to be designed and constructed (Locke, 1992; Koch, 1999). This reconstruction of the system adds more time overhead to the design process: however, with using tools such as those developed recently to support "automatic code generation" (Mwelwa *et al.*, 2006; Mwelwa, 2006; Kurian and Pont, 2007), the work involved in developing and maintaining such systems can be substantially reduced.

Another drawback of TTC systems, as noted by Koch (1999), is that constructing the cyclic executive model for a large set of tasks with periods that are prime to each other can be unaffordable. However, in practice, there is some flexibility in the choice of task periods (Xu and Parnas, 1993; Pont, 2001). For example, Gerber *et al.* (1995) demonstrated how a feasible solution for task periods can be obtained by considering the period harmonicity relationship of each task with all its successors. Kim *et al.* (1999) went further to improve and automate this period calibration method. Please also note that using a table to store the task schedule is only one way of implementing TTC algorithm where, in practice, there can be other implementation methods (Baker and Shaw, 1989; Pont, 2001). For example, Pont (2001) described an alternative to table-driven schedule implementation for the TTC algorithm which has the potential to solve the co-prime periods problem and also simplify the process of modifying the whole task schedule later in the development life cycle or during the system run-time (more details about this type of implementation are presented in Chapter 5).

Furthermore, it has also been reported that a long task whose execution time exceeds the period of the highest rate (shortest period) task cannot be scheduled on the basic TTC scheduler (Locke, 1992). As previously discussed (see Section 2.6), one solution to this

problem is to break down the long task into multiple short tasks that can fit in the minor cycle. Also, possible alternative solution to this problem is to use a Time-Triggered Hybrid (TTH) scheduler (Pont, 2001) in which a limited degree of pre-emption is supported. One acknowledged advantage of using TTH scheduler is that it enables the designer to build a static, fixed-priority schedule made up of a collection of co-operative tasks and a single (short) pre-emptive task (Phatrapornnant, 2007). Note that TTH architectures are not covered in the context of this thesis. For more details about these scheduling approaches, see (Pont, 2001; Maaita and Pont, 2005; Hughes and Pont, in press; Phatrapornnant, 2007).

Please note that later in this thesis, it will be demonstrated how, with extra care at the implementation stage, one can easily deal with many of the TTC scheduler limitations indicated above.

2.9 Jitter in scheduling algorithms

Having discussed the impact of jitter on the performance of real-time embedded systems (Section 2.4), this section goes on to review some previous work that attempted to deal with jitter in scheduling systems. Note that during the scheduler design process, while the schedule parameter set ensures that all tasks can be scheduled, inappropriate decisions may still lead (for example) to high levels of task jitter.

Recently, Dr. Teera Phatrapornnant, has carried out a detailed research on possible sources of jitter (Phatrapornnant, 2007). A brief summary of his findings is presented here.

Overall, jitter is a common problem which faces the developers of modern systems. For example, in digital wireless communication systems, jitter can be found in the form of a phase noise of the local oscillator. In practice, noise may come from the power supply lines or interference from other nearby signals. Such noise may have a direct impact on timing margins and, consequently, limit the system performance. In high-speed-digital systems, jitter can arise from crosstalk, caused by electromagnetic interference (EMI) along a circuit or a cable pair. Another example affected by EMI is a high-speed optical transmitter which converts data from electrical to optical format at speeds of 10

Gigabits per second. EMI can cause excessive clock jitter that may lead to errors in the optical transmitted data. To overcome this type of jitter, a suggested solution is to enclose the transmitter oscillator in a metal shield.

Jitter is also a common problem in the implementations of real-time control systems. In control systems, there are three main processes performed: sampling, control computation, and actuation. Delays in the operation of these processes can result in degraded performance and hence instability of the system. The main source of such delays is the scheduling algorithm employed. An example of this can be the dynamic scheduling (as opposed to static scheduling such as that used in RM and TTC) where activities such as context switches can cause delays to the operating tasks. Moreover, since the three main processes in the control loop execute in a sequential manner, variation in their execution times may lead to sampling jitter and sampling-actuation delays.

In ideal real-time systems, tasks must be scheduled and executed very precisely. In practice, however, accurate executions may not be achievable: not least because of inappropriate selection of the scheduling algorithm or due to imperfect implementations of the designed scheduler (this issue is further highlighted in Chapter 3). Such imprecise executions of the tasks can, in turn, result in considerable amounts of jitter and hence cause a reduction in the overall system performance.

In real-time tasks, jitter can be associated with different parameters such as release time, execution time and finishing time. For example, every task is ideally required to begin execution immediately after it is released. If the task execution deviates from its ideal release time, then this time deviation (variation) is described as *release jitter*. Similarly, *execution jitter* and *finishing jitter* describe the deviation of the execution duration and the completion time of the task, respectively.

Real-time systems are typically made up of periodic (and possibly aperiodic) tasks. As an example, in many real-time control systems, sampling and actuating tasks are run periodically and have hard timing constraints. These tasks are expected to execute repeatedly at their own periods. Figure 2-10 illustrates a periodic task that is intended to run with period T_i. The task is characterised by its starting time s, finishing time f and

deadline *d*. The figure shows that delays in s_1 and s_3 (and variations in the task durations: i.e. $f_i - s_i$, where i = 1, 2, 3, …) mean that these tasks show evidence of jitter in both release and completion times. For deterministic execution of a periodic task, intervals between its successive execution times must be kept constant (i.e. $P_i = P_{i+1}$, where i = 1, 2, 3, …).

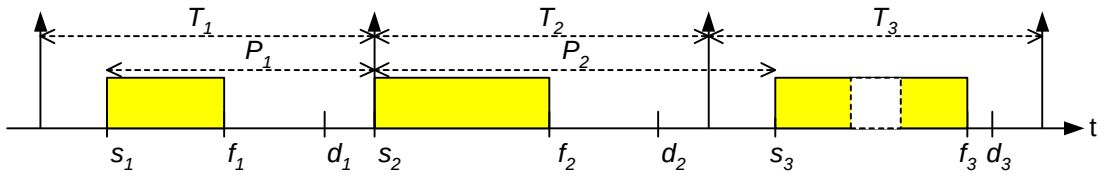


Figure 2-10: Task period jitters (adapted from Mart, 2002).

When TTC architectures (which represent the main focus of this thesis) are employed, possible sources of task jitter can be divided into three main categories:

- Scheduling overhead variation.
- Task placement.
- Clock drift.

The overhead of a conventional (non-co-operative) scheduler arises mainly from context switching. However, in some TTC systems the scheduling overhead is comparatively large and may have a highly variable duration due to code branching or computations that have non-fixed lengths. As an example, Figure 2-11 illustrates how a TTC system can suffer release jitter as a result of variations in the scheduler overhead (this relates to DVS system).

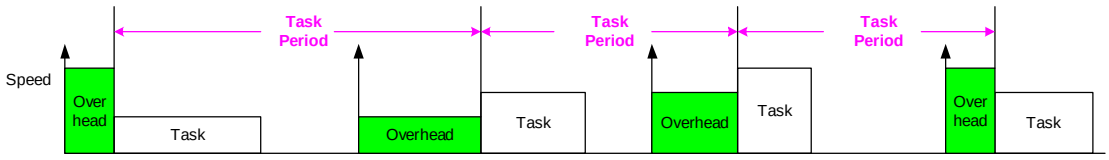


Figure 2-11: Release jitter caused by variation of scheduling overhead (Phatrapornnant, 2007).

Even if the scheduler overhead variations can be avoided, TTC designs can still suffer from jitter as a result of the task placement. To illustrate this, consider Figure 2-12. In

this schedule example, Task C runs sometimes after A, sometimes after A and B, and sometimes alone. Therefore, the period between every two successive runs of Task C is highly variable. Moreover, if Task A and B have variable execution durations (as in Figure 2-10), then the jitter levels of Task C will even be larger.

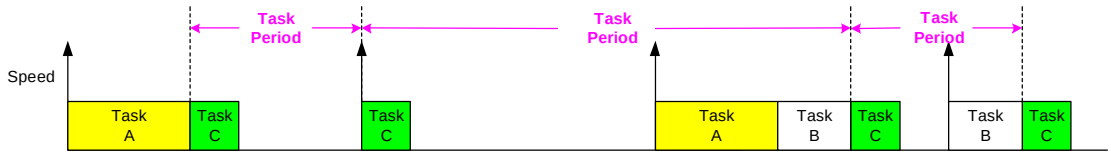


Figure 2-12: Release jitter caused by task placement in TTC schedulers.

For completeness of this discussion, it is also important to consider clock drift as a source of task jitter. In the TTC designs, a clock "tick" is generated by a hardware timer that is used to trigger the execution of the cyclic tasks (Pont, 2001). This mechanism relies on the presence of a timer that runs at a fixed frequency. In such circumstances, any jitter will arise from variations at the hardware level (e.g. through the use of a low-cost frequency source, such as a ceramic resonator, to drive the on-chip oscillator: see Pont, 2001). In the TTC scheduler implementations considered in this study, the software developer has no control over the clock source. However, in some circumstances, those implementing a scheduler must take such factors into account. For example, in situations where DVS is employed (to reduce CPU power consumption), it may take a variable amount of time for the processor's phase-locked loop (PLL) to stabilise after the clock frequency is changed (see Figure 2-13). As discussed elsewhere, it is possible to compensate for such changes in software and thereby reduce jitter (see Phatrapornnant and Pont, 2006; Phatrapornnant, 2007).

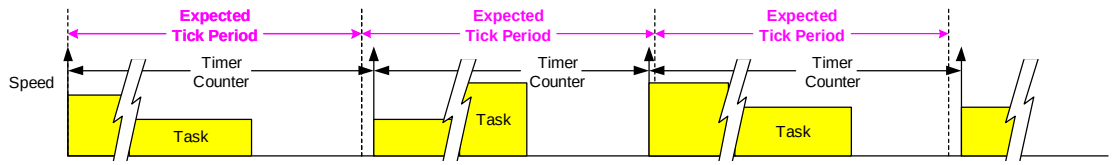


Figure 2-13: Clock drift in DVS systems (Phatrapornnant, 2007).

As a general summary, jitter in embedded systems has been found to arise due to clock drift, branching in the code, the scheduling algorithm employed, or as a consequence of using specific hardware (Sanfridson, 2000). In real-time systems, where real-time

schedulers are employed, the jitter is mainly considered at task level (e.g. release time), and most concern about task jitter has been in the context of scheduling (Lin and Herkert, 1996). For example, standard scheduling algorithms based on fixed timing constraints (e.g. fixed periods and deadlines) can induce jitter if a task is blocked in a high-load situation: to deal with such issues, a range of flexible solutions have been proposed for use at run-time (Marti, 2001a). In distributed systems, reducing the variations in message transmission times can help to reduce the jitter levels (Nolte *et al.*, 2001; Nolte *et al.*, 2002; Nolte, 2003; Nahas and Pont, 2005). Jitter in multi-processor systems is further discussed in Chapter 10.

2.10 Error detection and error recovery mechanisms

So far, it has been assumed that the system always operates correctly. Of course, this may not always be the case. For example, TTC architectures employ static scheduling and no task pre-emption. The schedule is calculated based on estimates of task "worst case execution time" (WCET). If such estimates prove to be incorrect, the problem may not even be detected in a basic TTC implementation. In hard real-time systems, it is essential to monitor the execution times of all tasks and detect overrun situations in which the estimated WCET of a task is exceeded (Burns and Wellings, 2007a). Such a task overrun error may have serious impact on system behaviour. For example, as Buttazzo (2005) has noted: *"[Co-operative] scheduling is fragile during overload situations, since a task exceeding its predicted execution time could generate (if not aborted) a domino effect on the subsequent tasks."*

As many researchers have observed (Becker *et al.*, 2003; Becker and Gergeleit, 2001; Domaratsky and Perevozchikov, 2000; Engblom *et al.*, 2001; Gergeleit and Nett, 2002; Kirner and Puschner, 2003; Liu and Layland, 1973; Nett *et al.*, 1996; Puschner, 2002), determining the WCET of tasks is rarely straightforward. This is therefore a significant concern and – if implementing a TTC scheduler – the user needs to appreciate this potential risk, and understand precisely how the scheduler will behave if such an error occurs. It should be noted that lack of knowledge about WCET is a problem which faces the developers of many embedded systems (not just those based on TTC). For example, as Gergeleit and Nett (2002) have noted: *"Nearly all known real-time scheduling approaches rely on the knowledge of WCETs for all tasks of the system."*

One simple solution to this problem is to err on the side of caution when employing WCET estimates, thereby reducing the chances of an overrun occurrence. Typical "safety margins" used in this way are said to be around 20% (Vallerio and Jha, 2003).

Such an approach is simple and can be effective, but inevitably adds to costs. An alternative is to be slightly more conservative when estimating WCET values (e.g. add 5% to accurate estimates) and then extend the scheduler (or add additional hardware) in such a way that (at run time) any overrunning tasks can be shut down, and/or the schedule can be adjusted (Gendy and Pont, 2008). Such an approach also allows dealing with error-related overruns (for example, tasks which overrun because of a hardware-related error). In these circumstances, the problem can be addressed (at least in part) by employing some form of "watchdog timer" (e.g. Ganssle, 1992) in a "scheduler watchdog" design (e.g. Pont and Ong, 2003). Alternatively, greater control over the system behaviour can be obtained by using a "task guardian" (Hughes and Pont, 2004).

The use of task guardians in TTC scheduler implementation will be considered in more detail in Chapter 5.

2.11 Scheduling multi-processor embedded systems

In the case of multi-processor embedded systems, where tasks are distributed across a number of CPUs communicating with each other, the need for effective network protocol as well as scheduling algorithm is essential. Further details about scheduling methods for multi-processor embedded systems which have severe resource constraints and require high predictability are provided later in Chapter 8.

2.12 Conclusions

This chapter described in detail the various elements required to build a scheduler for real-time embedded systems. The particular focus was on systems which have severe resource constraints and require high levels of timing predictability. All necessary

definitions have been provided to help understand the scheduling theory which is the central topic of the studies detailed in this thesis.

The chapter began by discussing how embedded systems can be built from scratch. As the chapter moved on, various techniques and architectures used to build a scheduler were described and compared. It was emphasised that scheduling algorithms are key elements in scheduling systems which dictate the way in which real-time tasks must operate during the system run-time. Two particular scheduling algorithms – that provide high predictability – have been outlined and compared in detail. These algorithms are: rate monotonic and time-triggered co-operative (as a form of cyclic executive) schedulers.

The discussions indicated that for the type of embedded systems considered in this project, no scheduling algorithms can be competitive to time-triggered co-operative (TTC) schedulers. This was mainly due to their simplicity, low resource requirements and extreme predictability they can offer. The chapter, however, discussed major problems that can affect the performance of TTC schedulers and reviewed some previously-suggested solutions to overcome such problems. Note that ways to deal with some of these problems will be considered in more detail later in this thesis.

Chapter 3

Real-time scheduler implementations

3.1 Introduction

In Chapter 1, it was noted that once the design specifications of a system are clearly defined and then turned into appropriate design elements, the system implementation process can take place by translating those designs into software and hardware components. People working on the development of embedded systems are often concerned with the software implementation of the system in which the system specifications are converted into an executable system (Sommerville, 2007; Koch, 1999). For example, Koch interpreted the implementation of a system as the way in which the software program is arranged to meet the system specifications.

Chapter 2 provided an overview of a number of effective schedule design techniques and scheduling algorithms used to implement the software architecture of an embedded design. This chapter moves on to discuss the challenges encountered in the process of translating between scheduling algorithms and scheduler implementations in practical real-time embedded systems. It also reviews previous work in the area of scheduler implementations and discusses the main drawbacks and limitations of this work. Please note that the main focus of the discussions is on software methods for scheduler implementations.

3.2 Choice of the programming language

3.2.1 Introduction

Having decided on the software architecture of the embedded design, the next key decision to be made is the choice of programming language to implement the embedded software (including the scheduler code). The choice of programming language is an important design consideration as it plays a significant role in reducing the total development time (Grogono, 1999). This section discusses the key challenges faced by an embedded programmer to select a suitable programming language for their

implementations. The section summarises the main motivations behind using 'C' programming language to implement software codes for the designs considered in this study. Please note that a detailed overview of the available programming languages is provided in Appendix B.

3.2.2 Choosing a language for embedded systems

Overall, it has been widely accepted that the low-level Assembly language suffers high development costs and lack of code portability, and only very few highly-skilled Assembly programmers can be found today (see Barr, 1999 and Walls, 2005). If the decision is therefore made not to use the Assembly language due to its inevitable drawbacks, there is no scientific way to select the most optimal high-level programming language for a particular application (Sammet, 1969; Pont, 2002). Instead, people tend to discuss the important factors which should be considered in the choice of a language. For example, Sammet (1969) indicated that a major factor in selecting a language is the language suitability to solve the particular classes of problems for which it is intended, and the type of the actual user (i.e. user professionalism). It has also been noted by Sammet that factors such as availability on the desired computer hardware, history and previous evaluation, implementation consequences of the language are also key factors to consider in language selection process. However, Sammet stressed that a successful choice can only be made if the language includes the required technical features.

Specifically, when choosing a language for embedded systems, the following factors must be considered (Pont, 2003):

- Embedded processors normally have limited speed and memory, therefore the language used must be efficient to meet the system resource constraints.
- Programming embedded systems require a low-level access to the hardware. For example, there might be a need to read from / write to particular memory locations. Such actions require appropriate accessing mechanisms, e.g. pointers.
- The language must support the creation of flexible libraries, making it easy to re-use code components in various projects. It is also important that the developed software should be easily ported and adapted to work on different processors with minimal changes.

- The language must be widely used in order to ensure that the developer can continue to recruit experienced professional programmers, and to guarantee that the existing programmers can have access to information sources (such as books, manuals, websites) for examples of good design and programming practices.

3.2.3 The 'C' programming language

Of course, there is no perfect choice of the language. However, the chosen language is required to be well-defined, efficient, supports low-level access to hardware, and available for the platform on which it is intended to be used. Against all of these factors, C language scores well, hence it turns out to be the most appropriate language to implement software for small (low-cost) embedded systems like the ones considered in this project. Pont (2003) stated that *"C's strengths for embedded system greatly outweigh its weaknesses. It may not be an ideal language for developing embedded systems, but it is unlikely that a 'perfect' language will be created"*.

The key features of the C language can be summarised as follows:

- C is easy to learn and program by both skilled and unskilled programmers.
- It is very popular as many experienced C programmers can be found today.
- It has well-proven compilers available for almost every embedded processor in use today (e.g. 8-, 16-, 32-bit or more).
- Materials (such as books, training courses, code examples and websites) that discuss the use of the language are all widely available.
- It has efficient run-time performance.
- It is a hardware-independent programming language which allows the programmer to concentrate only on the algorithm instead of the hardware on which the program will operate.
- It is a mid-level language with both high-level features (such as support for functions and modules) and low-level features (such as access to hardware via pointers) that allows the programmer to interact easily with the underlying hardware without sacrificing the benefits of using high-level programming.

For more details, refer to (Barr, 1999; Grogono, 1999; Jones, 2002; Brosgol, 2003; Pont, 2003; Fisher *et al.*, 2004; Ciocarlie and Simon, 2007).

Moreover, since C was recognised as the de facto language for coding embedded systems including those which are safety-related (Jones, 2002; Pont, 2002; Walls, 2005), there have been attempts to make C a standard language for such applications by improving its safety characteristics rather than promoting the use of safer languages that are less popular (such as Ada). For example, The UK-based Motor Industry Software Reliability Association (MISRA) has produced a set of guidelines (and rules) for the use of C language in safety-critical software: such guidelines are well known as "MISRA C". For more details, see (Jones, 2002).

3.2.4 Why does 'C' overwhelm other languages?

When comparing C to other alternative languages such as C++ or Ada, the following observations have been made. C++ is a good alternative to C as it provides better abstraction for data and offers better Object-Oriented (O-O) programming style, but some of its features may cause degradation in program efficiency (Barr, 1999). Also, such a new generation O-O language is not readily available for the small embedded systems, primarily because of the overheads inherent in the O-O approach, e.g. CPU-time overhead (Pont, 2003).

Despite that Ada was the foremost language that provided full support for concurrent and real-time programming, it has not gained much popularity (Brosgol, 2003) and has rarely been used outside the areas related to defence and aerospace applications (Barr, 1999; Ciocarlie and Simon, 2007). Unlike C, not many programmers nowadays are professional in Ada, therefore only a small number of embedded systems are currently developed in this language (Ciocarlie and Simon, 2007). In addition, despite their approved efficiency, Ada's compilers are not widely available for small embedded microcontrollers and usually need hard work to accept the program; especially by new programmers (Dewar, 2006). Indeed, both Ada and C++ have too large demand on low-

cost embedded systems resources (e.g. memory requirements) and therefore they cannot be suitable languages for such applications[5] (Walls, 2005).

In a survey carried out recently by Embedded Systems Design (ESD) in 2006, it was shown that the majority of existing and future embedded projects to which the survey applied were programmed (and likely to be programmed) in C. In particular, the figures show that for 2006 projects, 51% were programmed in C, 30% in C++, and less than 5% were programmed in Ada. The survey shows that 47% of the embedded programmers were likely to continue to use C in their next projects. See Figure 3-1 for further details.

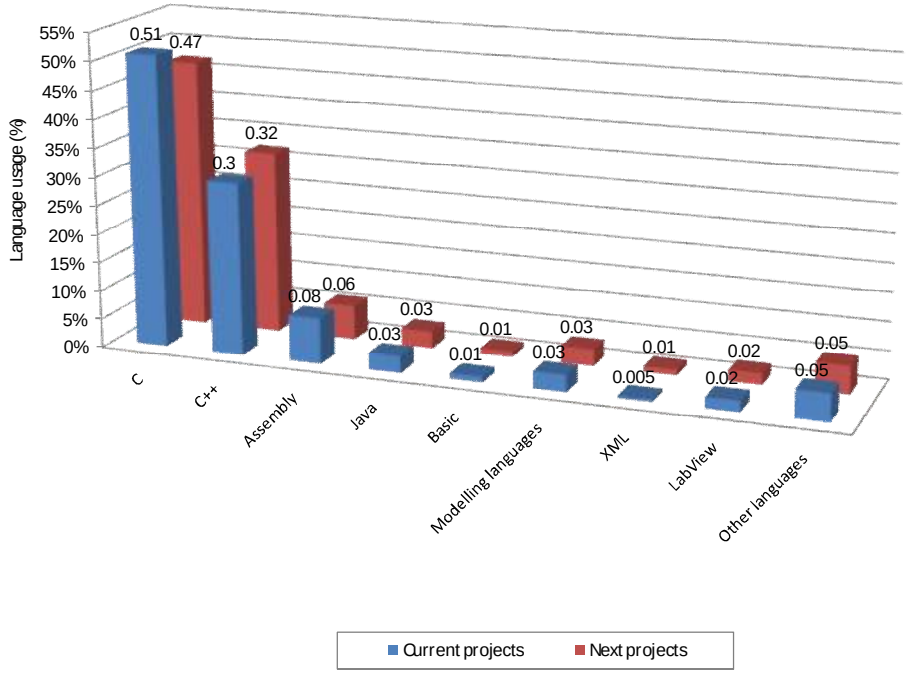


Figure 3-1: Programming languages used in embedded system projects surveyed by ESD in 2006. The figure is derived from the data provided in (ESD, 2006).

[5] However, despite the indicated limitations of Ada, there has recently been a great deal of work on assessing a new version of Ada language (i.e. Ada-2005) to widen its application domain (see Burns, 2006; Taft *et al.*, 2007). It has been noted that Ada-2005 – once standardised – will have enough power to overwhelm the use of C and its descendants in embedded systems programming (Brosgol and Ruiz, 2007).

3.3 Scheduling algorithms and scheduler implementations

3.3.1 Introduction

As discussed in Chapter 2, implementing the software architecture of an embedded design requires a simple form of operating system such as a scheduler. It was also noted that the core component of a scheduler is the scheduling algorithm which mainly determines the run-time execution order of the application tasks executed by the scheduler. In Chapter 2, a range of widely-used scheduling algorithms in the development of embedded systems were listed and two key algorithms discussed in more detail.

The discussion in Chapter 2 remarked that when high predictability is an important design feature of the embedded system, time-triggered co-operative (TTC) schedulers can be a good design solution, if an appropriate implementation is used. This section discusses the differences and relationships between scheduling algorithms and scheduler implementations in practical real-time embedded systems, with a particular focus on software implementations.

The implementation of schedulers is a major problem which faces designers of real-time scheduling systems (for example, see Cho *et al.*, 2005). In their useful publication, Cho and colleges clarified that the well-known term *scheduling* is used to describe the process of finding the optimal schedule for a set of real-time tasks, while the term *scheduler implementation* refers to the process of implementing a physical (software or hardware) scheduler that enforces – at run-time – the task sequencing determined by the designed schedule (Cho *et al.*, 2007).

Generally, it has been argued that there is a wide gap between scheduling theory and its implementation in operating system kernels running on specific hardware, and for any meaningful validation of timing properties of real-time applications, this gap must be bridged (Katcher *et al.*, 1993).

3.3.2 The ‘one-to-many’ relationship

To begin to address the gap between scheduling algorithms and scheduler implementations, it must be noted that the relationship between any scheduling

algorithm and the number of possible implementation options for that algorithm – in practical designs – has generally been viewed as 'one-to-many', even for very simple systems (Baker and Shaw, 1989; Koch; 1999; Pont, 2001; Baruah, 2006; Pont *et al.*, 2007; Phatrapornnant, 2007).

For example, the cyclic executive is a very simple scheduling algorithm in widespread use which can, in practice, be implemented using several forms. In (Baker and Shaw, 1989), it was stated that task schedule in cyclic executive can be constructed either by a pre-processor, manually, or using a table of actions that is generated offline and interpreted by the executive. Pont (2001) provided an alternative implementation in which the task schedule in the cyclic executive system can be constructed and/or modified at run-time, allowing more flexibility and responsiveness to system changes. Various possible ways to implement software for a cyclic executive scheduler were discussed in (Kontak, 1988; Baker and Shaw, 1989; Pont, 2001). In the same way, Koch (1999) has viewed the cyclic executive as one of only four high-level software architecture families[6] used in real-time systems, where each of these architectures can, in practice, have many variations. In (Pont *et al.*, 2007), it was clearly mentioned that if someone was to use a particular scheduling architecture, then there are many different implementation options which can be available. This claim was also supported by Phatrapornnant (2007) by noting that the TTC scheduler (which is a form of cyclic executive) is only an algorithm where, in practice, there can be many possible ways to implement such an algorithm.

Of course, the one-to-many relationship is not only limited to the cyclic executive. For example, Baruah (2006) has demonstrated how the Earliest-Deadline-First[7] (EDF) algorithm (Liu, 2000) – which often schedules tasks pre-emptively in single-processors – can be implemented using other forms, such as using non-pre-emptive scheduling

[6] These architecture families are: [1] cyclic executive, [2] concurrent task systems activated by events, [3] message passing systems, and [4] client-server systems. For more details, see (Koch, 1999).
[7] In EDF system, the task priorities are dynamically allocated, at each time instant, so that the task with the closest deadline will be assigned the highest priority to run first (Liu, 2000).

architectures upon multi-processor platforms. Section 3.4 (and Appendix C) reviews studies which looked at various possible ways for implementing scheduling algorithms including fixed-priority schedulers.

The project described in this thesis was mainly concerned with linking scheduling algorithms and their software implementations. Therefore, the term "scheduler implementation" used in the context of this thesis will refer to the process of implementing scheduler in software in which the scheduling algorithm is translated into a general-purpose executable source code (using C language).

It is worth noting that the source code of a scheduler may be implemented using an appropriate collection of "software design patterns" (see Section 3.5.3 for more information). Mwelwa (2006) discussed in detail how a design pattern can have almost an infinite number of implementation options and therefore the relationship between any pattern and its implementation is best described as "one pattern, many implementations". Figure 3-2 illustrates how a simple TTC algorithm can be implemented using a range of software patterns, each of which is related to a different scheduler implementation (e.g. TTC-1, TTC-2, etc) and has a number of Pattern Implementation Examples (PIEs) associated with different hardware platforms (e.g. 8051, c167 or ARM microcontroller). This example can provide the basis for understanding the one-to-many relationship between a scheduling algorithm and its software implementation in practical real-time embedded systems.

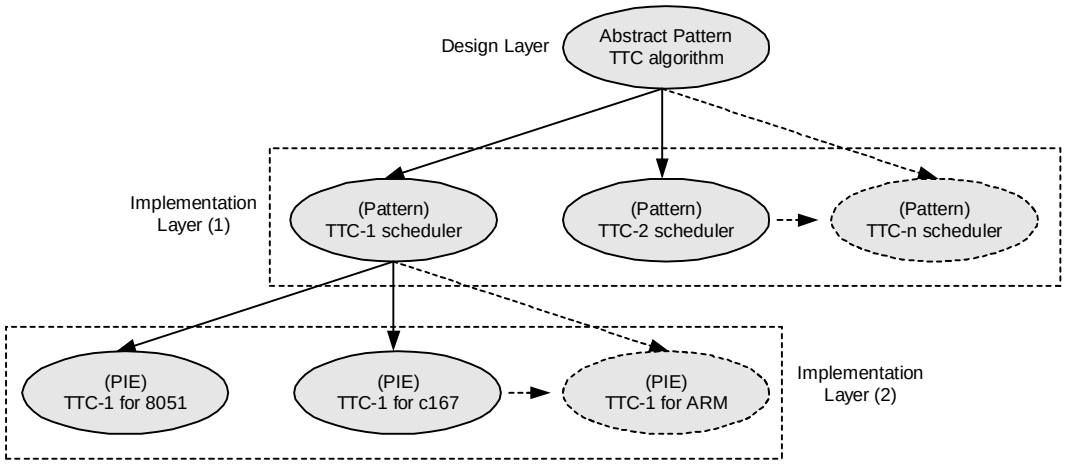


Figure 3-2: The one-to-many relationship between the TTC scheduling algorithm and its implementations using patterns. This figure is adapted from (Mwelwa, 2006).

3.3.3 The importance of scheduler implementation process

Overall, the scheduling algorithm can only be viewed as a "high-level" mathematical description of the scheduling policy, and the scheduler behaviour can only be defined through the implementation process of this algorithm (e.g. through the source code implementation which is the executable software product par excellence at the moment: see Bloomfield *et al.*, 2004). The scheduler source code can hence be described as the lower-level software representation of the system which has the responsibility of determining the functional and temporal behaviour of the system in practical use.

Avrunin *et al.* (1998) underlined that the performance of a real-time system depends crucially on implementation details that cannot be captured at the design level, thus it is more appropriate to evaluate the real-time properties of the system after it is fully implemented. This simply means that ensuring predictability in system behaviour would require an additional care to be taken during the (software) coding process.

3.4 General scheduler implementation approaches

3.4.1 Scheduler implementations in Ada

Early work on software scheduler implementation is referred back to 1980s when researchers attempted to implement scheduling systems using Ada programming language. Many researchers began by identifying the main shortcomings and limitations of the original Ada definitions (i.e. Ada, 1980) to fulfil the requirements of real-time embedded systems, especially those which have hard timing constraints. For example, it was widely accepted that despite many useful advantages of Ada in supporting software engineering principles, Ada language was yet fragile in supporting the software development for real-time scheduling systems (Burns and Wellings, 1987; Cornhill and Sha, 1987; Locke and Vogel, 1987; McCormick, 1987; Borger *et al.*, 1988; LeGrand, 1988; Baker and Shaw, 1989). This had, in turn, driven researchers to explore techniques which can address the real-time deficiencies of Ada. For example, many studies proposed useful extensions to Ada language to enable it facilitate a real-time software programming (Burns and Wellings, 1987; Locke and Vogel, 1987; McCormick, 1987; Cornhill *et al.*, 1987; Borger *et al.*, 1988; LeGrand, 1988; Baker and Shaw, 1989). Some of this work is reviewed here.

McCormick (1987) discussed a number of limitations in Ada task timing which made it inadequate to fulfil the requirements of hard real-time scheduling systems: e.g. nondeterminism of the Ada tasking model. One major problem which received a particular concern was the inability of Ada to detect and hence deal with the situations of task overruns. McCormick therefore proposed a method for providing finer control of task timing to circumvent such an Ada limitation with only little impact on the existing language definitions. In another study, Cornhill *et al.* (1987) proposed some modifications to the existing Ada language to support the implementation of hard real-time, fixed-priority scheduling algorithms such as rate monotonic.

LeGrand (1988) discussed the main features of Ada in supporting real-time task scheduling and outlined main committee and research activities in this area. Goodenough and Sha (1988) described one way in which priority ceiling protocol can be implemented in Ada to address the priority inversion problem in fixed-priority schedulers. In (Kontak, 1988; Baker and Shaw, 1989), possible ways for implementing cyclic executive scheduling algorithm in Ada language were presented (see Section 3.5 for more details). Moreover, there has been some effort made towards developing tools for automatic Ada code generation for real-time scheduling systems (for more details, see Cross II *et al.*, 1989).

Later on, Sha and Goodenough (1990) explored alternative ways for implementing rate monotonic scheduling algorithms using Ada model. Burns (1991) reviewed the results in the application of scheduling algorithms to hard real-time systems (including both static and dynamic algorithms) with a particular consideration to Ada tasks scheduling. Baker and Pazy (1991) provided an overview of a later generation of Ada (i.e. Ada-9X) and discussed the changes that this version brought to the Ada priority scheduling model. The practicality of using pre-emptive, priority-based scheduling techniques in on-board space application, using Ada (i.e. Ada-83 and Ada-9X) as the implementation language, was studied by the European Space Agency (ESA) in the early 1990s (Bailey *et al.*, 1993). In (Vardanega, 1996), the design and implementation of pre-emptive, priority-based scheduler to use for on-board satellite control systems were presented.

Over the last years, people in the Real-Time Systems (RTS) Group, The University of York, UK, have been greatly involved in the design and implementation of real-time

scheduling systems using Ada language. For example, Burns (1999) began to describe Ada tasking features which are designed specifically for safety-critical, hard real-time systems (i.e. Ravenscar Profile). Real and Wellings (1999a) studied the impact of using Ada to implement inheritance priority ceiling protocol on the task schedulability. Real and Wellings (1999b) described the importance of mode-change support in real-time systems in which the system functionality can vary as the operation progresses, and discussed how this feature can be implemented safely in Ada.

McElhone and Burns (2000) discussed requirements for ***adaptive*** real-time systems and demonstrated how adaptivity can be achieved (in Ada system) by using spare resource capacity to schedule optional computations. Burns (2001) suggested new modifications to Ada language to allow the implementation of non-pre-emptive schedulers. Bernat and Burns (2001) demonstrated how Ada tasking facilities enables the implementation of ***flexible*** scheduling schemes in which hard deadlines must always be met and any spare capacity (typically CPU resources) must be used to maximise the total utility of the application (see Davis ***et al.***, 1995). Burns ***et al.*** (2003a) described how Ada can be used to implement a form of round-robin scheduling algorithm in which tasks are executed in a circular queue and each task is allocated a bounded time slot for execution. Burns and Wellings (2003) and Burns ***et al.*** (2004) described possible extensions to Ada language to facilitate the implementation of non-fixed-priority scheduling algorithms such as EDF.

Overall, the RTS group has shown a great deal of interest in implementing flexible real-time systems using Ada (e.g. Burns and Wellings, 2002; Burns ***et al.***, 2003a; Burns and Wellings, 2003; Burns ***et al.***, 2003b; Wellings, 2003; Burns ***et al.***, 2004). However, it has been emphasised that most of the mechanisms explored and applied in these studies are incorporated in Ada-2005 standard (Burns, 2006): for Ada-2005 user manual, see (Taft ***et al.***, 2007). The use of Ada-2005 in real-time implementations has also been discussed in a range of papers published by RTS group (e.g. Burns and Wellings, 2007b; Zerzelidis ***et al.***, 2007; Wellings and Burns, 2007a; Wellings and Burns, 2007b). For the full list of RTS publications, see RTS (2008).

3.4.2 Scheduler implementations in 'C'

As remarked in Section 3.2, despite the strengths of Ada, C remains the most popular means of developing software for real-time and embedded systems. Therefore, C has been extensively used in the implementation of real-time schedulers and operating systems for embedded applications. In general, C was adopted in the software development of almost all operating systems (including RTOSs) in which schedulers are the core components (Laplante, 2004).

In Michael Barr's book on embedded systems programming (i.e. Barr, 1999), it was noted that C is the main focus of any book about embedded programming. Therefore, most of the sample codes presented in Barr's book – for both schedulers and operating systems – were written in C and the key focus of the discussion was on how to use C language for 'in-house' embedded software development. However, some of the example code presented later in the book was written in C++ while Assembly language was avoided as much as possible. In (Barr and Massa, 2006), possible ways for implementing the eCos and the Embedded Linux, as a small and a large open-source operating systems (respectively), in C language were discussed. Other books which discuss the use of C language in the software implementation of real-time embedded systems include (Ganssle, 1992; Brown, 1994; Sickle, 1997; Zurell, 2000; Labrosse , 2000; Samek, 2002; Barnett *et al.*, 2003; Laplante, 2004).

More specifically, in the field of embedded systems development, using C language to implement the software code for particular scheduling algorithms is quite common. For example, Mooney *et al.* (1997) described a strategy for implementing a dynamic run-time scheduler using both hardware and software components: the software part was implemented using C language. Kravetz and Franke (2001) described an alternative implementation of Linux operating system scheduler using C programming. It was emphasised that the new implementation can maintain the existing scheduler behavior / semantics with very little changes in the existing code.

Rao *et al.* (2008) discussed the implementation of a new pre-emptive scheduler framework using C language. The study basically reviewed and extracted the positive characteristics of existing pre-emptive algorithms (e.g. rate monotonic, EDF and LLF)

to implement a new robust, fully pre-emptive real-time scheduler aimed at providing better performance in terms of timing and resource utilisation.

As will be shown in the next section (Section 3.5), the ESL researchers have been greatly concerned with developing techniques and tools to support the design and implementation of reliable embedded systems, mainly using C programming language.

3.5 TTC scheduler implementations

3.5.1 Introduction

From the previous section, it can be clearly seen that pre-emptive scheduling architectures have received a widespread attention by embedded systems developers and researchers while non-pre-emptive schedulers have almost been ignored. More specifically, the software implementations of time-triggered co-operative (TTC) scheduling architectures and their implications in practical real-time embedded systems have rarely received any coverage. This is an unfortunate trend because TTC architectures are widely used in practical embedded applications (see Section 2.8.3) and – as a consequence of the resource, timing, and power constraints – the implementation of such designs is often far from trivial. For example, Pont (2001) has provided a general discussion of the challenges involved in practical implementations of TTC architectures. Previous work on the implementation of such systems is reviewed in this section.

3.5.2 Early work on TTC scheduler implementations

Some early work concerning the implementation of TTC architectures (in the Ada programming language) was carried out by Baker and Shaw (1989). Baker and Shaw began by outlining the problems in Ada which impose challenges on such type of work. For example, tasks in Ada usually execute indefinitely and non-periodically, and the tasking system is based on nondeterministic event-triggered scheduling approach. The authors then demonstrated how such and other limitations in Ada language can be solved to facilitate the implementation of a cyclic executive (that is periodic and time-triggered-based) scheduler in such a programming language.

The work by Baker and Shaw also proposed and evaluated two standard methods (within Ada) to implement a code for cyclic executive scheduler: one method was based on using delay statement while the other was based on using timer interrupts. The paper then described several ways in which the task overrun problems in TTC systems can be addressed in Ada language.

3.5.3 Recent work on TTC scheduler implementations

Recently, the ESL researchers have widely considered the implementation process of TTC schedulers on a broad range of low-cost embedded microcontroller platforms. An early work in this area was carried out by Pont (2001) which described techniques for implementing TTC architectures using a comprehensive set of "software design patterns" written in C programming language. The resulting "pattern language" was referred to as "PTTES[8] Collection" which contained more than seventy different patterns.

Pont has demonstrated that the main aim with this language was to facilitate a reliable implementation of TT systems in low-cost, resource-constrained embedded applications with a particular focus on TTC architectures. Since then, as experience in this area has grown, this pattern collection has expanded and subsequently been revised in a series of ESL publications (e.g. Pont and Ong, 2003; Pont and Mwelwa, 2003; Mwelwa *et al.*, 2003; Mwelwa and Pont, 2003; Pont *et al.*, 2003; Pont and Banner, 2004; Mwelwa *et al.*, 2004; Kurian and Pont, 2005; Kurian and Pont, 2006b; Pont *et al.*, 2006; Wang *et al.*, 2007).

One main objective of introducing patterns was to describe various ways in which simple embedded software can be implemented in practical systems. For example, Pont (2001) and Kurian and Pont (2007) introduced a range of different patterns which describe some of the possible ways in which a TTC scheduler can be implemented. It was illustrated how these implementations have significant differences in their resource

[8] PTTES stands for Patterns for Time-Triggered Embedded Systems.

requirements (e.g. data and code memory overhead). Note that an overview of these and some other TTC implementations will be provided later in this thesis.

Another example of studies that considered the implementation process of TTC systems is a work carried out by Key *et al.* (2003) which addressed the problems, and possible solutions, when attempting to implement TTC architectures in Assembly language.

In (Nahas *et al.*, 2004), a low-jitter TTC scheduler framework was described and compared with an early scheduler implementation (as in Pont, 2001) that took no account of the impact of scheduler overhead variation on the timing behaviour of the co-operative tasks running in the system.

Phatrapornnant and Pont (2004a and 2004b) looked at ways for implementing low-power TTC schedulers by applying "dynamic voltage scaling" (DVS) algorithm. In (Phatrapornnant and Pont, 2006), the authors went further to describe techniques which can maintain low jitter behaviour when the DVS algorithm is employed in a TTC system to reduce the system power consumption. The study also considered ways in which the low-jitter DVS algorithm on TTC can be applied using a range of System-on-Chip (SoC) embedded platforms (in addition to the COTS microcontroller platforms).

In another project, Hughes and Pont (2004 and 'in press') described an implementation of TTC schedulers with a wide range of "task guardian" mechanisms that aimed to reduce the impact of a task-overrun problem on the real-time performance of a TTC system.

Moreover, Dr Michael Pont and his PhD students have also considered the design and implementation of a time-triggered hybrid (TTH) scheduler which allows a single, time-triggered, pre-emptive task to be scheduled in the TTC scheduling framework. This architecture can sometimes be viewed as an extended version, or simply a modified implementation, of the original TTC scheduler. Various ways in which such a TTH scheduler can be implemented in practice have been described in (Pont, 2001; Maaita and Pont, 2005; Hughes and Pont, in press; Phatrapornnant, 2007).

For example, Maaita and Pont (2005) described two possible ways for implementing a TTH scheduler in low-cost embedded systems. They then described a technique (called "planned pre-emption") that can be applied to both TTH implementations considered in order to reduce jitter in the release time of the pre-emptive task. Hughes and Pont (in press) described a novel TTH implementation which incorporates "task guardian" mechanisms to deal with overruns in both the co-operative and the pre-emptive tasks running in the system. Likewise, Phatrapornnant (2007) considered the implementation of low-jitter DVS algorithm (developed originally for TTC architectures) on the equivalent TTH architectures.

Note that the source codes in all the outlined scheduler implementations (unless stated) were written in C programming language. Please also note that since 2001, the ESL researchers have also been concerned with the implementation of TTC architectures upon multi-processor embedded platforms. More details about the researches conducted in this area and the results obtained are provided later in Part D.

3.6 Hardware-based scheduler implementations

For completeness of the current research, it is worth noting that there has been a great deal of previous work on hardware-based scheduler implementation techniques. This work is beyond the scope of this thesis. However, Appendix C reviews some of the key studies carried out in this area.

3.7 The impact of scheduler implementation decisions on system behaviour

Considering the one-to-many mapping between the scheduling algorithm and its implementations in practical systems, it has been widely argued that particular implementation decisions for a given scheduler can have a profound impact on the behaviour of the system which implements this scheduler. This section discusses such an impact in a little more detail and provides an illustrative example.

Katcher *et al.* (1993) stated that the implementation of a particular algorithm can introduce costs which must be taken into account when validating the timing

correctness properties of a real-time system. Katcher *et al.* also argued that while the task periods (which are design parameters) are a function of the environment and the task specification, the actual execution times of tasks are a function of the particular implementation of the designed scheduler. In (Koch, 1999), it was reported that the choice of particular scheduler implementation can have a major impact on the critical success factors for a real-time system.

Xu (2003) emphasised that "the simplified high-level abstraction of code" is only an approximation of "the actual real-time software implementation" which does not take into account all the implementation details that may affect timing. Xu also reported that, in most cases, there is no proof that design abstractions such as specifications, models, algorithms and protocols have the same timing properties as the actual implementation code. Phatrapornnant (2007) noted that in the ideal case a real-time scheduler must schedule and execute tasks precisely, where in practice (given that the scheduling algorithm is selected properly) accurate execution of tasks cannot be achieved due to factors such as imperfect scheduler implementation.

Moreover, the different implementation options of a particular scheduler would have different resource requirements and performance results (Pont *et al.*, 2007). Diversity of implementations would also result in diverse complexity levels. For example, Baker and Shaw (1989) went through different possible cases in which the implementation of a cyclic executive scheduler requires additional complexity. Therefore, the performance of the real-time system would critically depend on implementation details of the task scheduler (Avrunin *et al.*, 1998). In addition, as the system expands, the scheduler design and implementation processes will increase in complexity and, consequently, the impact on the entire system performance becomes more significant (Cho *et al.*, 2007).

A very well-known example on how scheduler implementation can affect the overall system behaviour is the widely-publicised problems encountered during the Mars Pathfinder mission in 1997 (Jones, 1997). The Mars Pathfinder used VxWorks real-time operating system kernel which provides pre-emptive, fixed-priority scheduling of tasks. There were three tasks running in the system:

- A frequent bus management task (with high priority).

- An infrequent meteorological data gathering task (with low priority).
- An infrequent communications task (with medium priority).

The pathfinder used one shared bus for passing information between different components of the spacecraft.

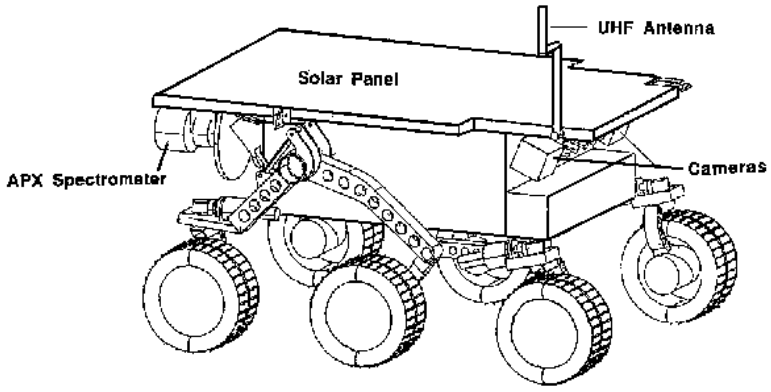


Figure 3-3: Mars pathfinder spacecraft (Source: NASA Jet Propulsion Laboratory).

On arrival of the management task, it was always being blocked by the meteorological data task for a very short time before the latter one releases the bus. This worked fine most of the time. However, at a particular time later on, the long-running communication task was ready to execute during the short interval while the (high priority) management task was blocked, awaiting the (low priority) meteorological data task. What happened is that this medium priority task pre-empted the low priority meteorological data task and began to execute. Consequently, the awaiting high-priority management task was prevented from running: this caused what is known as “priority inversion” (see Section 2.8.2). After sometime, it was detected that the data bus task had not been executed for a while and hence a total system reset was initiated. For more details, see (Jones, 1997). This example illustrates how an improper (or incomplete)

implementation of the task scheduler can have the potential to jeopardise the correct behaviour of the whole system[9].

3.8 Discussion

Despite the usefulness of the studies carried out in the area of schedulers, the topic of scheduler implementation and its implications in practical real-time embedded systems has not been discussed thoroughly. More specifically, while there has been a great deal of interest in the development, assessment and refinement of real-time scheduling algorithms, the process of translating between algorithms and implementations has not been widely considered. This claim is supported by Cho *et al.* (2007) who clearly stated that only very few researches address the architecture and the implementation of the schedulers. The great majority of the studies reviewed during the course of this project tend to focus mainly on design issues and only discuss implementation issues from a high perspective without considering the potential impact a particular software implementation would have on the actual run-time behaviour of the system implementing the scheduler.

Moreover, despite the usefulness of the studies carried out in the area of TTC schedulers, there are apparent limitations. For example, it can be noticed that the number of TTC scheduler implementations developed in the ESL group has significantly increased over the past few years. Due to the high experience gained with TTC, this trend is expected to continue over the next few years or even grow as concerns about predictability in real-time embedded systems is growing. Therefore, the

[9] Once debugged, the problem was solved by amending the software code of the spacecraft from the lab. In more details, the access to the bus was synchronised with mutual exclusion locks (mutexes). In the VxWorks mutex object, there was a parameter that indicates whether priority inheritance should be performed by the mutex. This parameter, which was initially set ‘off’, had been set ‘on’. This caused the low-priority meteorological data task to inherit the priority of the high-priority management task, while the latter one is blocked, and hence complete execution before the medium-priority communication task: thus, the priority inversion was prevented (Jones, 1997).

need for documenting, categorising and comparing the various TTC scheduler implementations in a systematic way becomes of vital importance.

It should also be underlined that each of the TTC scheduler implementations developed previously was dedicated specifically to solve a particular class of problems, while have not been assessed against other issues. This leaves a gap that when a particular TTC scheduler is to be selected for a project, the user will not have sufficient information about the way the system is expected to behave in the future when different operating conditions apply. So far, there has been no attempts within the ESL group to combine all achievements in a single study in which all TTC schedulers are linked and compared systematically to help potential users understand all aspects of the TTC system and consequently be able to decide whether (or not) to use a particular TTC scheduler implementation in a given project.

3.9 Conclusions

Having discussed scheduling algorithms in Chapter 2, this chapter moved on to discuss the various issues related to the process of implementing real-time schedulers in practical embedded systems, with a particular focus on software implementation process.

The chapter began by discussing the process of selecting a programming language to implement software for low-cost, resource-constrained embedded designs like those dealt with in this project. The key features of C language, which made it an appropriate choice for such designs, were summarised.

The chapter then discussed the challenges that might arise when implementing software for a particular scheduler. Such challenges were mainly caused by the broad range of possible implementation options a scheduler can have in practice, and the impact of such implementations on the overall system behaviour. It was hence noted that source code implementation is a crucial component to take care of during the implementation process for achieving predictable behaviour.

A wide range of previous work on scheduler implementations, using different programming languages, was then reviewed. This work considered various scheduling algorithms including cyclic executive, fixed- and dynamic-priority scheduling algorithms. The discussions then focused on previous work carried out in the area of TTC scheduler implementations, particularly within the ESL research group.

Thereafter, the impact of scheduler implementation on system behaviour was discussed in detail with presenting a famous example that shows how improper implementation decisions can detrimentally affect the system operation.

Before concluding the chapter, the gaps in the previous work have been emphasised. In summary, the topic of scheduler implementation requires further consideration, and the work conducted in the ESL group on TTC implementations can be improved by employing further techniques which provide a deep understanding of the scheduler implementation process for use in future applications.

Chapter 4
Linking scheduling algorithms and scheduler implementations

4.1 Introduction

Having discussed the relationship between scheduling algorithms and scheduler implementations in practical real-time embedded systems, the main focus of this chapter is on generic methods that link these two system representations in a systematic way.

In real-time operating environments, it is important to ensure that – after the scheduler software is implemented – it will behave as required. Generally, there are two main processes to evaluate the operation of any software-based system: *validation*, to ensure that the right system is built, and *verification*, to ensure that the system is built right (Boehm, 1981; Hessel, 2007). The primary purpose of the validation and verification processes is to establish confidence that the software system is adequate for its intended use (Sommerville, 2007). Therefore, validation and verification can hold the promise to narrow the gap between the processes of scheduler design and scheduler implementation in real-time, resource-constrained embedded systems.

Before beginning to review and analyse results from previous work in this area, it must be pointed out that there has been confusion in the use of the terms “validation” and “verification” by many people working on the evaluation of software systems. For example, some people tend to think that “validation” and “verification” are synonyms (Sommerville, 2007). The discussion in this chapter begins by reviewing the various definitions for these two terms using a collection of recognised sources. The chapter then reviews prevalent techniques for verifying software systems with a particular focus on real-time embedded software systems. The chapter finally concludes by discussing the limitations of these techniques in addressing the problems concerned with in this project.

4.2 Definitions

In the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std, 1990), validation is defined as: *"The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements."* Similarly, the STING software engineering glossary (STING, 1996) defines validation as: *"The process of evaluating software at the end of the development process to ensure compliance with software requirements."* The Glossary of Computerized System and Software Development Terminology (GCSSDT, 1995), which is based mainly on IEEE and many other international standards, defines software validation as: *"Determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements."* The National Institute of Standards and Technology (NIST, 2007) defines validations as*: "The process of determining whether or not the standard at a given phase of its development fulfils the established requirements."*

In contrast, verification is defined in the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std, 1990) as: *"(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness."* The Glossary of Computerized System and Software Development Terminology (GCSSDT, 1995) defines software verification as: *"The demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle."* The STING software engineering glossary (STING, 1996) defines verification as: *"The process of determining whether or not the products of a given phase in the life-cycle fulfil a set of established requirements."* Moreover, verification is also defined in the Software Testing Glossary (STG, 2008) as: *"The process of determining whether or not the products of a given phase of the software development cycle meet the implementation steps and can be traced to the incoming objectives established during the previous phase."*

Despite this, Ian Sommerville (in his famous book on "Software Engineering", Eighth Edition, 2007) defines validation as a general process which shows that the software meets the customer needs, while verification is the process which ensures that the

software conforms to its specification. He also notes that a key role of the verification process is to check that the software meets specified functional and non-functional requirements.

According to the discrepancy in the way validation and verification are defined, it will not be possible to pursue the discussion in this chapter before distinguishing between these two terms and providing a more generic definition for each term. By reviewing the list of definitions stated above, it can be concluded that a system is said to be valid (or validated) if its final software product meets the user's needs and requirements. Any process involved in checking this is described as a validation process. In contrast, to verify the system, it means checking whether the implementation of a system component matches its defined specifications which have originally been derived from the user's requirements.

This interpretation of the terms validation and verification might sound more compatible with the definitions provided by Sommerville (2007). However, it does not necessarily contradict the concepts behind the IEEE and the other international standards definitions. For example, the IEEE, STING and STG glossaries indicate that verification is used mainly to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase or at the end of the previous phase. This has clearly been asserted by Sommerville that while validation is a general process which checks the consistency of the system – as a whole – with its requirements, verification is a more detailed process which must be applied at each stage in the software development process to check the conformance of that stage with its predefined specification.

Tran (1999) makes this point clearer by noting that *"validation usually takes place at the end of the development cycle, and looks at the complete system as opposed to verification, which focuses on smaller sub-systems"*. The same point is made in the Glossary of Computerized System and Software Development Terminology (GCSSDT, 1995), one of which the definitions provided here are compared to, as *"Validation is usually accomplished by verifying each stage of the software development life cycle."* In another word, validation can be viewed as "end-to-end" verification process (Bloomfield *et al.*, 2004). This concept is illustrated schematically in Figure 4-1 which

shows one possible way of including validation and verification processes in the software development life cycle model.

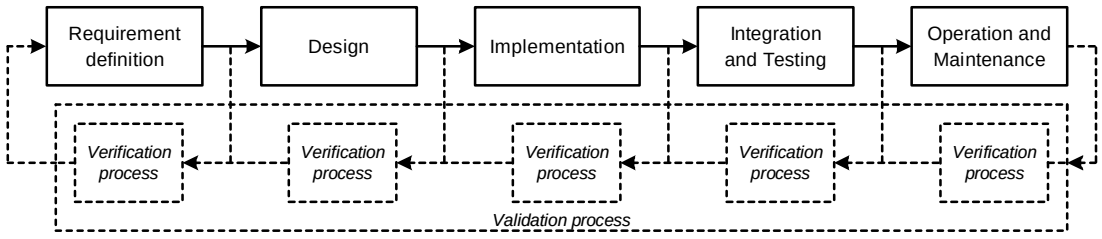


Figure 4-1: Integrating validation and verification in the software development life cycle (adapted from Sommerville, 2007).

Both validation and verification (which is commonly referred to as V&V process) are required in the evaluation of any software system to make sure that the whole software product fulfils the system requirements and operates as the user wants it to operate. In the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std, 1990), it is pointed out that V&V describes *"The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfil the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements."* Sommerville (2007) also reported that, during and after implementation, the developed software needs checking to ensure that it meets its specification and delivers the functionality expected by its user, where this can only be achieved through combining validation and verification processes together.

Since this thesis is concerned with matching scheduling algorithms and scheduler implementations, a software verification technique may be applied between the system design and system implementation stages (see Figure 4-1).

4.3 Software verification techniques

4.3.1 Introduction

For safety-critical embedded systems, a high degree of reliability is needed for software implementations since faults can result in catastrophes. Therefore, the validation and

verification of software used in this type of applications poses challenges which are not usually addressed in conventional software engineering (Dai and Scott, 1995).

Overall, verification of real-time embedded system designs was found to be typically performed by prototyping and simulation or, more effectively, by means of formal methods which express the behaviour of the system mathematically (Balarin *et al.*, 1996). Balarin and colleges underlined that prototyping is expensive and cannot be performed until the design is almost completed and, for complex systems, simulation becomes less effective as only limited number of patterns can be tried.

By referring to (Sommerville, 2007), there are two main approaches which can be used for system verification: "static", through software *inspections* and *formal methods*, and "dynamic", through *software testing*. Software inspections and formal methods are static verification techniques simply because they check and analyse the system without running its software on a computer, while software testing is a dynamic verification technique that mainly examines the output of the system after running its software implementation on a computer with test data. It has been argued that any product obtained during development (e.g. specification document or source code) can be evaluated using static analysis, while dynamic analysis (namely testing) almost evaluates software code only (Bloomfield *et al.*, 2004). In a project carried out recently by Hessel (2007), it has been affirmed that testing is the foremost software verification method in computer and real-time applications.

Moreover, in embedded software development, it has been argued that techniques for "automated code generation" can be an effective way of verifying the correctness of the implemented software: this is simply because they help to ensure that the generated source code is error-free and matches the requirements specified at the design phase of the system development process (Mwelwa, 2006).

This section reviews each of the outlined software verification techniques and related research in this area.

4.3.2 Software inspections

4.3.2.1 Introduction

According to the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std, 1990), inspection is a *"static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems."* Inspections (which are sometimes described as "peer reviews") mainly check and analyse the source code representation of the system but can also be used to review other readable system representations such as specification document or design diagrams. Performing inspections is highly based on previous knowledge about the system and its application domain as well as familiarity with the programming language used to implement its source code (i.e. program). The main goal of program inspection is to reveal errors, omissions and anomalies. Overall, program inspection is a dedicated verification method in which only defects in the program are detected: such defects may include logical errors, invalid conditions or incompliance with organisational standards (Sommerville, 2007).

4.3.2.2 Software inspection process

Generally, software inspection is a formal process which requires a team of people to analyse the software component and hence find possible defects in it. The concept of formal inspection process was first developed in the 1970s at IBM (Fagan, 1976). Fagan suggested that a team of – at least – four people would be required for a complete program inspection activity: e.g. "author", who writes the source code, "reader", to read the program to the team, "tester", to inspect the program from a testing point of view and "moderator", to organise the whole process. Moreover, it was suggested that any inspection process would be divided into six stages: *planning*, by the moderator who selects the inspection team and organises the inspection meeting, *overview*, by the author who describes the objective of the written program to the team, *individual preparation*, by inspection team members who study the program and begin to look for defects, *inspection meeting*, where discovered defects are announced by the inspectors, *rework*, by the author who corrects the identified problems and, finally, *follow-up*, by the moderator who decides whether the corrected program needs a new inspection or the defects have successfully been fixed by the author (Fagan, 1976 and 1986; Ackerman *et al.*, 1989; Sommerville, 2007).

In a later study, Grady and Van Slack (1994) suggested other set of roles which offers more flexibility to the number of inspection team members. Gilb and Graham (1993) described alternative approaches and provided many examples and case studies based on actual experience at well-known software development companies such as IBM, AT&T and others. A general comment they made was that the inspection team should be selected in such a way that they can reflect different perspectives about the program.

Before any inspection process starts, the specification of the program to be inspected must be defined accurately, the members of the inspection team must be familiar with the organisational standards, and each member must have an up-to-date version of the program (Sommerville, 2007). A set of defined checklists are typically used in the inspection process to focus the inspectors on the common errors that are likely to exist in a particular application domain and/or a programming language. This combination of well-structured team and checklists made the formal inspections distinguished from other types of software reviews (Dyer, 1992). A detailed description of the inspection theory and practice is provided in (Wheeler *et al.*, 1996).

Although originally designed to verify the system at the code level, inspection practices were extended to cover earlier stages of the system development life-cycle such as the design process (Fagan, 1976 ; Ackerman *et al.*, 1989; Grady , 1992; Gilb and Graham, 1993; Ebenau and Strauss, 1994; Cheng and Jeffery 1996). Kelly *et al.* (1992) reported that defects in software requirement specifications are likely to be more than in any other document produced later in the development process. Therefore, inspection techniques can always be applied to verify software design and software implementation, where design inspections check the translation of the requirements into a software design, and code inspections check the translation of that design into a program implementation (Dyer, 1992).

4.3.2.3 Enhancement of inspection process

Despite their noticeable advantages, conventional inspection activities (like the ones described earlier) are known to have a number of drawbacks. For example, Nunamaker *et al.* (1991), Gilb and Graham (1993) and Harjumaa and Tervonen (1998) highlighted the most common problems that might arise during a conventional inspection process,

e.g. insufficient knowledge about the process or the document to be inspected, geographical distribution of the inspection team members and possible conflicts between the inspectors during the inspection meetings.

To deal with such problems and hence ensure the cost-effectiveness of an inspection process, a number of computer-based tools have been developed and employed. One recognised work in this field is that carried out by Harjumaa and Tervonen (1998) in which a cost-effective tool – based on the World Wide Web (WWW) – was developed to provide a set of well-defined functions for distributing the document to be inspected, annotating it, searching related documents, choosing the checklist and gathering inspection statistics easily.

As previously noted, inspections are driven from checklists of errors relate to various application domains and programming languages. In some cases, it is possible to automate the process of checking the program against the listed errors. This led to the development of static analysers for different programming languages. Static analysers are basically software tools which scan the program and detect possible errors. They typically utilise the error detection facilities provided by the language compiler to detect if any statement in the program produces errors or formed incorrectly. One of the main purposes of automatic static analysis is to detect errors that might cause problems later when the program is executed and which cannot be detected by manual inspection activities (e.g. data whose value goes out of range). Static analysers are more effective when used with programming languages such as C as an error-prone language whose compilers have limited checking capabilities (for more details, see Sommerville, 2007).

4.3.2.4 Strengths and weaknesses of software inspections

Overall, the significant role of inspection in the software verification process has been recognised by foremost researchers in this area. For example, Fagan (1976) has made it clear that if errors are not detected close to their place of origin, the cost of rework as a fraction of the overall development cost can be incredibly high. Boehm (1981) provided data which illustrated that repairing a software error, after the software production, can be 100 times more costly than if early error-detection mechanisms are employed. Therefore, the use of software inspections (where errors are detected and eliminated near the point of their introduction) helps to increase productivity and improve the

overall quality of the produced software (Fagan, 1976 and 1986; Ackerman *et al.*, 1989; Dyer, 1992).

Moreover, there has been an argument that software inspection can be more effective than testing (Section 4.3.4), due to the following advantages (Sommerville, 2007):

- Inspection allows many errors in the system to be detected at once where there are no worries about interactions between errors.
- It is a low-cost process in which the software can be verified before completion.
- It helps developers consider other quality attributes of the system (e.g. poor programming) along with finding program defects.

In addition, inspections are very cost-effective methods for software defect detection and elimination, where between 50% and 90% of the errors in the program can be discovered using these techniques (Fagan, 1986; Mills *et al.*, 1987; Dyer, 1992; Gilb and Graham, 1993). Sommerville has noted that the effort devoted to static verification techniques generally increases as the system goes more critical, however, the effort required for program inspection can always be 50% less than would be required for equivalent testing process.

Despite the recognised advantages of using program inspections to verify software, more formalised techniques will still be required to address the verification problems that cannot be addressed by basic software inspection activities, e.g. the functional correctness of the developed software. This has prompted software engineers to develop formal verification techniques.

4.3.3 Formal methods

4.3.3.1 Introduction

As the complexity and criticalness of the system increase, a more detailed analysis of the system specification and program is required for verification process. Such type of analysis can be achieved by means of formal methods. Formal methods are primarily based on mathematical representation of the system software which are mainly concerned with mathematically analysing the system specification or transforming this

specification into a semantically equivalent representation of the system (Wang and Lin, 2001; Wang, 2004; Sommerville, 2007). Note that using formal methods in the verification process is a static approach in which a detailed analysis of the system components is carried out without executing them on a computer.

Balarin *et al.* (1996) emphasised that formal verification is a set of techniques that allow for proving mathematically that specifications are fit for a design. This process can hence be used at various points during the software development process. For example, formal methods can be used – at the design level – to discover errors / omissions in the specification document and at the implementation level to check that the software program is compatible with its specification.

It is accepted that formal approaches (which incorporate formal specification and formal verification) are well suited for safety-related systems which almost require a high degree of reliability (Hevner *et al.*, 1992; Bowen, 1993; Sommerville, 2007). Bowen (1993) surveyed a number of safety-related standards in terms of their utilisation of formal methods, e.g. 00-55 (Mod, UK), IEC880 (IEC, Europe), MIL-STD-882B / 882C (DoD, US) and P1228 (IEEE, US).

4.3.3.2 Cleanroom development process

Formal methods have been used in various software development processes such as VDM (Jones, 1989), Z (Spivey, 1992) and B (Wordsworth, 1996). A well-known example of development processes which rely on formal methods is the *Cleanroom* process developed by IBM and aimed at producing zero-defect software system with high reliability (Dyer and Mills, 1981; Dyer, 1982; Currit *et al.*, 1986; Mills *et al.*, 1987; Linger, 1994; Spangler; 1996). Briefly, Cleanroom software development combines three main processes: formal method for specification and design, non-execution-based program development, and statistically-based independent testing (Selby *et al.*, 1987). In the Cleanroom development process, the life cycle mainly consists of executable product increments which all accumulate to yield the final product with full functionality (Currit *et al.*, 1986; Selby *et al.*, 1987; Linger, 1994). Each software product increment is specified formally and this specification is then transformed into an implementation.

Sommerville (2007) summarised the processes involved in any Cleanroom development environment as: *formal specification*, in which the software is formally specified, *incremental development*, in which the software is partitioned into increments to be validated separately, *structured programming*, for stepwise refinement of the specification in which only few constructs are used for systematically transforming the specification into a source code, *static verification*, in which software inspections are used to verify the software increment, and finally *statistical testing*, where the integrated software increment is verified statistically using an "operational profile" which has already been developed simultaneously with system specification. He also suggested that three different teams would ideally be required for a Cleanroom process: *specification team*, which develop the system specification and its equivalent mathematical model, *development team*, which develop and verify the software using formal approach to verification, and *certification team*, which develop statistical tests to exercise the developed software for reliability certification.

Dyer (1992) pointed out that the mathematical-based design of the Cleanroom method results in a more correctly developed software with a significantly reduced number of errors in comparison with the level of errors assumed in conventional development practices. He therefore suggested that the role of inspection in such a formal development process should change from error detection only to confirmation of software correctness. In his useful study, Dyer discussed how the use of verification-based software inspection (instead of formal software inspection) can be more effective in Cleanroom environments as it prevents the introduction of errors while the software design is being constructed, thereby achieving error-free software products. Likewise, Powell (2002) reported that inspection techniques mainly focus on quality attributes that affect readability and maintainability where only limited amount of work on inspection considered the functional correctness. To verify the functional correctness of software in Cleanroom process, systematic code reading techniques were found to be more effective than the traditional code reading techniques based on checklists (Dyer, 1992; Jackson and Hoffman, 1994; Porter *et al.*, 1995; Powell, 2002).

The integration of Computer-Aided Software Engineering (CASE) environments to support Cleanroom development process has also been considered and found to be very

useful especially for solving complex development problems (Fuhrer *et al.*, 1992; Hevner *et al.*, 1992)

Overall, it has been widely agreed that Cleanroom approach is an effective development method for systems which have stringent safety and reliability requirements such as the majority of real-time embedded systems (Hevner *et al.*, 1992; Sommerville, 2007). The Cleanroom's ease-of-use, low-cost and effectiveness in reducing failure rate have been proven practically in a number of studies (e.g. Selby *et al.*, 1987; Cobb and Mills, 1990; Linger, 1994; Stavely, 1999). For example, Cobb and Mills (1990) summarised the results from a list of previously-conducted projects which utilised Cleanroom development process. They show how the use of such development environments had the potential to improve software quality and productivity in all the surveyed projects.

4.3.3.3 Formal verification of real-time embedded systems

The use of formal methods in the verification of real-time embedded systems has received widespread consideration. For example, Dai and Scott (1995) developed a CASE tool (called "Automation of the Verification, Validation and Testing – AVAT") to verify real-time embedded software using the Program Function (PF) table method (Pamas, 1994) as a widely-used formal method in industrial applications. Bradley *et al.* (1996) indicated that testing only is insufficient to provide confidence that a real-time system would always meet its deadlines and therefore formal methods would be required in the verification of such systems. Authors however noted that despite that formal methods can result in assured designs, they do not necessarily lead to assured implementations. The study therefore proposed a formal-based technique, based on Application Oriented Real-Time Algebra (AORTA) as a formal language, to verify implementation of particular real-time designs.

Liu *et al.* (1998) described a technique with a case study to verifying safety-critical embedded software using the practical formal method "Structured-Object-based-Formal Language" (SOFL). In their technique, three verification processes were applied consecutively as to verify both functionality and safety properties of the software: data flow reachability checking, specification testing, and rigorous proofs (see Liu *et al.*, 1998 for further details). Formal verification of a large-scale, fault-tolerant embedded

system was carried out in (Shi *et al.*, 1999) where the developed technique was based on using CSP model-checker (Roscoe, 1994).

Clarke *et al.* (2000) and Cortès (2001) argued that techniques such as simulation and testing are often insufficient as they evaluate the system performance for only selected subsets of operating conditions. Clarke *et al.* (2000) hence proposed a tool which complements simulation and testing methods for embedded systems with formal verification methods that analyse the behaviour of the system over a large set of operating conditions without recourse to exhaustive simulation. Cortès (2001) noted that formal methods had extensively been used in software development as well as in hardware verification but not widely used in embedded systems design. Consequently, Cortès (2000) and Cortès *et al.* (2001) proposed a modelling formalism for real-time embedded systems – based on Petri Net modelling language (Marwedel, 2006) – and introduced an approach to solve the problem of formal verification of real-time embedded systems represented in his modelling formalism.

Likewise, Xu (2003) stated that while they had been used successfully to verify hardware designs, formal methods were rarely used to verify actual software code and were not used at all to verify timing properties of large-scale, real-time software implementations. Xu listed the following examples of proposed formal methods for real-time systems: timed and hybrid automata, timed transition systems / temporal logic, timed Petri-nets, theorem proving techniques using PVS to analyse real-time protocols and algorithms and model checking. However, he underlined that such methods mainly focus on the verification of high-level abstractions of the system but not its actual software code. Xu proposed a "pre-run-time" scheduling framework that imposes restrictions on the software structures to reduce the complexity of large-scale embedded software and hence simplify the process of formally verifying its functional as well as timing correctness properties.

Broadfoot and Broadfoot (2003) published a useful paper which attempted to link between academic research in formal methods and their practical use in embedded software development. They discussed the main reasons why formal verification methods are not widely utilised in industry, some of which were the lack of scalability, limited accessibility to non-specialists and immaturity of the available tools and

techniques. However, the two key problems the authors identified as the main challenges for industrial software practices were the need for specialists who have sound mathematical background to create the systems formal specifications and the complexity of using available methods and tools to verify the system correctness even after the formal specifications are developed. The study then discussed the applicability of combining the two formal methods, namely "Cleanroom" and "CSP" (with its model checker FDR) to overcome the outlined shortfalls. The proposed approach was then applied to a number of industrial case studies to show its effectiveness in practical applications.

Wang (2004) reviewed previous works on formal verification of real-time systems. More specifically, he discussed a wide range of research papers on various topics in formal verification including formal modelling, specification languages, verification frameworks, state-space representations and some others.

Arons *et al.* (2006) noted that although simulation is not adequate to verify large, complex embedded systems due to its limited coverage metrics of the system, simulation techniques can still be very effective if appropriately combined with formal verification techniques: such an approach is referred to as hybrid verification. Arons *et al.* described a hybrid approach – supported by automatic tools – to improve traditional simulation-based validation techniques for complex embedded software designs. The approach is based on formally analysing the software program to generate a coverage space for all feasible control paths of the program and then building architectural tests (Section 4.3.4). Ribeiro and Fernandes (2007) proposed an approach for applying a particular model checker (called "Spin": Holzmann, 1997) to verify some key properties of embedded systems, such as deadlock-freedom, where the behaviour of the embedded system is modelled using a variant of Petri Nets modelling language.

Crocker and Carlton (2007) suggested that before any testing commences, the correctness of the system must be verified formally to ensure that the executed software will meet the stated requirements. Such an approach was described as "correctness-by-construction". Crocker and Carlton discussed their software development tool "Perfect Developer" which was developed in 2004 to reason about requirements and specifications of the system by using a single formal notation for specification, design

and refinement, followed by automatic translation of the refined design to source code. The study moved on to investigate the applicability of this automated reasoning approach on verifying programs written in C, as a popular implementation language for embedded software. It was found that automated reasoning can still achieve an acceptable degree of success in the verification of software written in conventional programming languages such as C.

In another study, Gargantini *et al.* (2008) proposed a validation and verification tool that supports high level formal analysis of model-driven embedded system designs. The tool was based on a formal method called "Abstract State Machine" and aimed at providing a design and analysis environment for HW/SW co-design where both software application and hardware architecture are described using UML model.

4.3.3.4 Strengths and weaknesses of formal methods

It has been argued that formal specification is a very effective way to discover problems in the specification which is the most common cause of system failures, and formal verification increases the confidence in the most critical component of the system which is the software program (Sommerville, 2007). It is also believed that formal methods are so beneficial especially in the development of critical systems such as safety-related embedded systems (Hevner *et al.*, 1992; Bowen and Hinchey, 1995).

Nonetheless, the use of formal specification and corresponding formal verification is often time-consuming and expensive, and its cost would increase as the complexity and criticalness of the system increase (Broadfoot and Broadfoot, 2003; Sommerville, 2007). There is a feeling among some researchers that software systems can still be effectively verified using cheaper verification techniques such as inspections and testing (Sommerville, 2007).

Moreover, formal methods require specialised engineers with solid mathematical expertise to create formal specification, whereas this specification is almost not understandable by domain users. The need for specialised people to create the formal model of a design makes it impractical to adopt formal techniques in the verification of many software systems.

More importantly, despite that they can lead to highly-reliable and safe systems, formal methods do not guarantee software reliability in practical use. For example, although they can assure matchup between formal specification and generated code, the developers cannot guarantee matchup between their formal model and the original system (and user) requirements (Lutz, 1993) or that the code – once running on COTS microcontroller hardware – would behave as expected since the hardware is not formally modelled in the system design process.

As a consequence, it seems that dynamic verification techniques are likely to remain a key way of checking the correctness of the system behaviour while it is operational for the foreseeable future. In 2004, Farn Wang noted that *"in the foreseeable future, it will be difficult to use formal techniques alone for decisive answers to complex verification tasks."* Wang hence predicted that formal verification would be used in the future as a superior technology to guarantee the quality of software systems but not to verify them (Wang, 2004).

In addition (as previously noted in Chapter 1), in real-time environments, besides the correct functionality of the software, it is essential to make sure that the system fulfils its predefined timing constraints and hence behave deterministically. It is unlikely that formal methods – as a static verification technique – can guarantee this. Instead, dynamic techniques (such as testing) would be required in which the developed software is executed on a computer to check its real-time behaviour while it is operational. As a result, software testing would remain – in many cases – the most effective way to achieve a confidence that the system is "completely" fit for its intended use, especially in real-time applications.

4.3.4 Software testing

4.3.4.1 Introduction

Software testing is an essential part in any evaluation process of software systems. In the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std, 1990), testing is defined as *"(1) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. (2) The process of analyzing a software item to*

detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items." Hessel (2007) defined testing as the process of exercising the system in a controlled environment and examine if its behaviour complies with the requirements of the system.

It has been argued that while static analysis techniques are used to evaluate static criteria of the software product at rest (e.g. errors in software documents), dynamic analysis, namely testing, is used to evaluate dynamic criteria related to the properties that can only manifest when the system is running (e.g. failure when the software does not behave as required): see Bloomfield *et al.* (2004). Sommerville (2007) reported that both processes, static and dynamic, are needed for successful verification of real-time systems. For example, inspections can be used to verify the system in the early stages of its development process, while testing is needed after the whole system is integrated in order to verify the behaviour of the final system before deployment.

4.3.4.2 Testing process

Sommerville (2007) pointed out that there are two key activities for testing: *component testing*, which tests particular parts of the system, and *system testing*, which tests the system as a whole. In system testing, the system is checked against its functional and non-functional requirements to ensure that the system behaves correctly for practical use. The main objectives of software testing are:

- To ensure that the software meets its requirements.
- To discover any faults or defects in the software.

In general, software testing cannot discover all errors but can guarantee that the software is good enough for operational use. For complex systems, system testing is usually divided into *integration testing*, which is concerned with finding problems that arise as a result of integrating the system components, and *release testing*, which is concerned with validating that the system meets the user requirements and is dependable. For more details, refer to (Sommerville, 2007).

As the complexity of software application increases, software testing process becomes less trivial. For example, various researchers argued that, in the software development

process, testing can consume up to 50% of the total development cost (e.g. Singh *et al.*, 1997; Liu *et al.*, 2005; Pringsulaka and Daengdej, 2006; Sommerville, 2007).

For any testing process, a set of suitable test cases must be designed. A test case is *"(1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. (2) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item."* (IEEE Std, 1990). A comprehensive testing using every possible execution of the software is unfeasible. Therefore, only an effective subset of possible test cases is used. For successful test cases design, the feature of the system to be tested must be selected along with the inputs that will execute that feature. In addition, the expected outputs of the test cases must also be known. A general model for software testing process is illustrated in Figure 4-2.

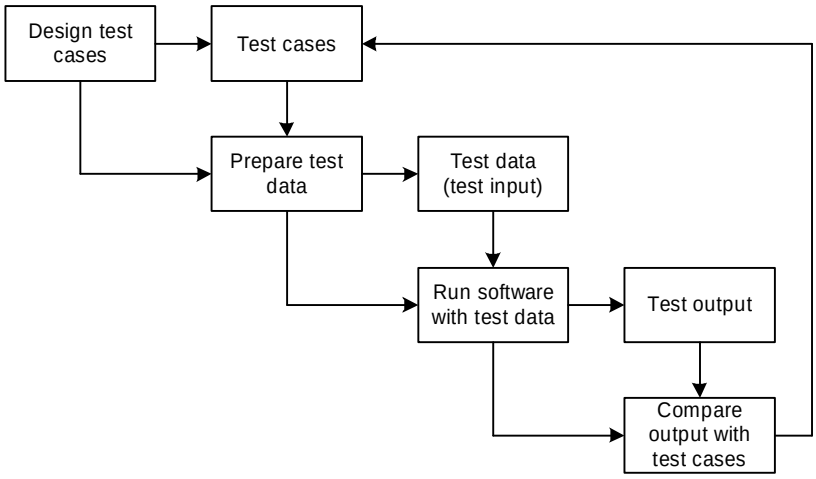


Figure 4-2: Testing process model (adapted from Sommerville, 2007).

The figure shows the key elements in the testing process which include: test cases, test data (or test input) and test output. The obtained output from the test must be compared to the output predicted – during the test case design phase – by those who possess a full understanding of the system and how it should operate after implementation.

Overall, test cases can be generated either manually or automatically. Many studies however demonstrated that automating the test processes has the potential to reduce

time, effort and costs (e.g. Cunning and Rozenblit, 1999; Rayadurgam, 2001; Tsai *et al.*, 2003; Do and Rothermel, 2006; Sommerville, 2007).

4.3.4.3 Test cases and test-case generation

There has been a great deal of interest on both test cases and test-case generation. For example, Beck's (2001) work on "extreme programming" had at heart a view that test cases for the system should be produced early in the product life cycle. Hassel (2007) argued that the most challenging phase in the test process is the selection and execution of test cases. Many studies have, therefore, proposed techniques for automatically generating the test cases (Tsai *et al.*, 2003; Bai *et al.*, 2002, Cunning and Rozenblit, 1999; Ince, 1987; Poston, 1986; Tai, 1993; White and Sahay, 1985; Munoz, 1988; Singh *et al.*, 1997; Kim *et al.*, 1998; Hessel, 2007).

Test cases can be generated from the source code, control flow graphs, design representations and specifications (Offutt and Liu, 1999). In specification-based test case generation approach – which is popular and widely used – test cases are derived from a specification model that provides a high-level abstraction of the desired system behaviour. One advantage of using specification-based test generation (over the code-based test generation where test cases are directly derived from the software code) is that the output test data will be independent of any particular implementation of the system: e.g. the source code (Offutt and Liu, 1999). This also means that test cases can be generated earlier in the development process, even before the coding is finished, allowing more utilisation of the time and resources.

To implement and test the specification model, formal specification languages can be used such as Z (Spivey, 1988), VDM (Jones, 1989), and RAISE (Nielsen *et al.*, 1988). Once the specification model – for the application under test – is verified, test cases can then be automatically (or manually) generated using the appropriate test-case generation method.

There has also been work carried out in the area of testing other features of the system. Such studies are concerned mainly with testing the "non-functional" (i.e. "quality") requirements of the application software, such as scalability, reliability, maintainability, availability and portability: a detailed list of non-functional requirements of a

computing system is provided in (Chung *et al.*, 2000). Just as an example, Laria (2005) has argued that architectural decisions affect the quality of software systems and that it is important to detect the potential risks of using particular architectures as early as possible during the development process. Examples of widely-used approaches for evaluating software architectures are: Architecture Tradeoff Analysis Method (ATAM: Kazman *et al.*, 2000) and Cost Benefit Analysis Method (CBAM: Bass *et al.*, 2003).

4.3.4.4 Testing real-time embedded systems

Rayadurgam (2001) noted that testing is one of the most widely used V&V techniques for verifying embedded systems. There has been a great deal of interest in generating test cases for real-time embedded systems (Cunning and Rozenblit, 1999; Larsen *et al.*, 2005; Nielsen and Skou, 2003; Shere and Carlson, 1994; En-Nouaary *et al.*, 2002; Hessel, 2007). In real-time testing activities, testers take into account the time at which the input parameters are supplied to the system and, therefore, the correct behaviour is achieved when the test verifies that the output values are produced in the correct time.

Overall, the testing of real-time embedded systems has been based on creating timed models (Clarke and Lee, 1997; Springintveld *et al.*, 1997; En-Nouaary *et al.*, 1998; Cunning and Rozenblit, 1999; En-Nouaary *et al.*, 1999; En-Nouaary *et al.*, 2000; Larsen *et al.*, 2005; Hessel, 2007). Hessel (2007) noted that *"Model-based testing is a black box testing technique where test cases are derived from a model that specifies the expected behaviour of a system."* It was also shown that it is sometimes required to used model checking tools (e.g. "Spin": Holzmann, 1997 or "UPPAAL": Larsen *et al.*, 1997) to ensure correctness of the specification model before moving to the implementation stage.

In a study carried out by Cunning and Rozenblit (1999), approaches for model-based automatic test-case generation for event-triggered, real-time embedded systems were presented. They attempted to generate a set of test cases that provide a complete coverage of the system requirements. The work was though based on software / hardware codesign in contrast to that presented earlier by Chandrasekharan *et al.* (1985) and Hsia *et al.* (1994 and 1997) which were concerned with testing software designs only.

En-Nouaary *et al.* (2000) clearly stated that the correctness of safety-critical embedded systems can be improved by verifying the system specification and testing the system implementation: that is to assess the conformance of the system implementation to its specified requirements. In their paper, En-Nouaary *et al.* focused on testing of real-time embedded systems. The embedded system in their study was modelled using Communicating Timed Input Output Automata (CTIOA) model. The study also described an approach for test-cases generation from an embedded CTIOA model. Larsen *et al.* (2005) described a tool for online testing of real-time systems based on UPPAAL model checker.

Hessel (2007) considered testing the functionality (i.e. logical and temporal correctness) of real-time systems based on state-based formalism. In particular, he developed a test generation tool called "COVER" which was built on UPPAAL model checker and used the formal method "timed automata". The tool proved to be effective in generating test suites with full coverage and minimal cost. Hessel also provided a detailed literature review of previous work carried out in the areas of model-based testing of timed systems, test-case generation with coverage criteria, and tools for model-based testing.

Please note that all of this previous work is mainly based on formal modelling of the real-time system properties.

4.3.4.5 Strengths and weaknesses of software testing

The strengths and weaknesses of software testing have already been discussed in the previous sections (e.g. Section 4.3.3.4 and Section 4.3.4.1). Overall, the discussions indicate that testing can be the most superior verification technique to achieve a "complete" confidence in a developed real-time software product, as it verifies properties that can only be revealed when the system is in operation. Please note that the decision to use testing would critically depend on the type of application in which software is used. For example, if the software is developed to operate an aircraft, then formal verification would be the only way to ensure correct operations before the plane flies in the air. In such circumstances, testing would not be the appropriate verification solution.

However, if testing is to be applied, developing test cases with full coverage of system requirements can often be a time-consuming and expensive process. This in turn induces further research interest in this area, where the advantages of testing can be utilised, but with using simple test case generation approaches in order to test only a small set of system properties for certain purposes.

Before testing takes place, and to avoid the need for applying ordinary static verification techniques (such as inspections and formal methods), one can use recently developed techniques for automated software generation. Assuming that design specifications are derived accurately from user requirements, such techniques can help guarantee matchup between design specifications and the produced software.

4.3.5 Automated code generation

4.3.5.1 Introduction

As previously noted, the current project considers the implementation process of TTC scheduling algorithm. One suggested approach to verify the software implementations of such a system is to use automated code generation in the creation of scheduler code (Mwelwa, 2006). By doing so, the generated source code is guaranteed to have zero-defects and thus only testing will be required to verify correctness of the system behaviour when it is operational. Automated code generation approach is reviewed in this section.

4.3.5.2 Manual generation of code using software patterns

An early work on patterns, in the ESL group, has resulted in a collection of several tens of software design patterns aimed at supporting the development of reliable TT embedded systems (see Section 3.5.3). These patterns were documented in a structured manner so that any developer – who wishes to use them – can refer to the relevant publication in which the patterns are detailed. However, the process of manually referring to pattern collection before using a pattern increases the probabilities of coding errors and hence becomes inefficient software development method for systems that require high level of reliability (Kurian and Pont, 2006a; Mwelwa, 2006). Therefore, it was necessary to develop techniques for automatically generating source code from a selected set of design patterns (Mwelwa *et al.*, 2005; Mwelwa *et al.*, 2007).

4.3.5.3 Automating the code generation process

Overall, it has previously been argued that automated code generation holds the promise of reducing the time and effort required to implement safety-critical systems, while at the same time eliminating errors introduced in this stage of development (Whalen and Heimdahl, 1999). Industries such as aerospace and automotive have made extensive use of automatic code generation tools aimed at control and signal processing systems (Marsh, 2003; O'Halloran, 2000; Schatz *et al.*, 2003): these are typical application areas for the TTC schedulers considered in this study. Such tools are used first to model systems and then to generate code. Originally, code was generated automatically for prototyping platforms or PCs. More recently, code generation has become a more practical means of generating production code for embedded hardware. It is thought that hundreds of thousands of cars now rely on code generated using these techniques (Marsh, 2003).

4.3.5.4 Automated code generation techniques

According to (Mwelwa, 2006), two main approaches are used in automated code generation, model-based and pattern-based code generation. In model-based code generation, models are used to represent the system at the abstract level thus allowing developers to design applications based on requirements only, discuss the design ideas among the design team, and validate the design even before it is implemented. The Unified Modelling Language (UML), which is used for organising and communicating design ideas, has become the de facto technology for design, analysis and modelling of various software architectures and more recently for model-based code generation. In model-based code generation method, source code is automatically generated from the UML design models and therefore errors caused by hand-coding are eliminated. Other modelling languages and frameworks which help in model-based software development are discussed in (Mwelwa, 2006).

Mwelwa has made it clear that despite many advantages of model-based code generation, such as the provision for software maintenance, these techniques have limited effects in embedded system designs. This is mainly due to the limitations experienced with UML such as inability to address timing, memory and power constraints, and handling of periodic time-triggered tasks (note that there has recently been significant work on extending the UML to embedded systems domain: e.g. UML

2.0). Mwelwa has also argued that model-based code generation does not necessarily promote software reuse where software reuse is recognised as an important factor for improving the quality of software and reducing development costs. Therefore, pattern-based code generation can offer an alternative method to model-based code generation.

It was argued that pattern-based code generation has the potential to produce codes with high quality. Previous work in this area has led to the development of a tool for "automatic" creation of systems with TTC architecture (e.g. Mwelwa *et al.*, 2003; Mwelwa, 2006; Mwelwa *et al.*, 2007). Such work enables the developer to employ a collection of "design patterns" to support the creation of code for complete TTC systems (including the system scheduler). Kurian and Pont (2006a) noted that pattern-based tools provide support for automatic code generation from pattern-based designs, where the full potential of pattern-based design was still to be fully realised in such tools. Kurian and Pont therefore began to explore the challenges involved in engaging tool support in the design phase of pattern-based software development.

In (Kurian and Pont, 2007) it was underlined that most previous work on pattern-based software development had focused on the process of creating a system but not on the post-creation project phases. The study therefore explored techniques for automatically replacing an existing core scheduler pattern with a suitable alternative pattern in a design after the project has been completed.

4.3.5.5 Strengths and weaknesses of automated code generation

Automated code generation can be viewed as a feed forward process which substantially helps the developer create (or manipulate) the source code for their application with minimal amount of time and effort. They also help to verify the embedded software by ensuring that the implementation of the system scheduler matches the predefined design specification.

However, automated code generation techniques cannot guarantee that the implemented software meets the user requirements and the software product is hence validated. For example, automated code generation techniques take no account of the possible behaviour patterns a particular code may produce during the system operating time. In another word, such techniques do not involve any feedback process from which the

developer can understand (or predict) the implications of using particular source code implementations in their system.

4.4 Discussion

The work described in this thesis is mainly concerned with software verification of real-time schedulers as a potential means for bridging the gap between scheduling algorithms and scheduler implementations in practical embedded systems. By going back to Figure 1-3 in Chapter 1 (the life cycle of a system development process), it can be clearly seen that after the system design process completes, the implementation process should begin. As in Figure 4-1, to address the process of translating between design and implementation of a given embedded software (e.g. scheduler), an appropriate verification technique must be applied during this process to check whether the resulting implementation matches the requirements specified at the end of the design phase (which is obviously the previous phase to implementation in the development life cycle).

Given that real-time embedded software often requires a high degree of reliability and predictability, dynamic approaches (namely testing) remain in many cases the most effective means for verifying that the software – after implementation – will match the original design specifications and hence fulfil the desired user requirements.

As the literature indicates, previous work on testing and test cases was concerned mainly with testing the detailed operations (i.e. functionality) or checking quality attributes of a given software application.

Although there has been considerable effort made towards testing real-time embedded software, this work was mainly based on developing a formal specification model of the embedded design from which suitable suite of test cases can then be generated. As previously noted, formal modelling of a system is a complicated process that requires a specialist to describe and analyse the system specifications using mathematical notations.

In particular, the author found no previous work which considered the creation of test cases to specifically address the impact of using a given scheduler software on the operational behaviour of embedded systems, particularly when algorithms such as TTC schedulers are employed.

On the other hand, despite the great potential of work carried out on automated code generation to link TTC designs and implementations, such work suffers considerable limitations, some of which have already been discussed in Section 4.3.5.5. Moreover, it must be noted that the automated code generation tools developed in the ESL group so far have been proven to work successfully with simple TTC scheduler implementations while have not been used to generate / verify codes for large-scale, complex TTC designs like the majority of those considered in this project. This imposes the demand to explore alternative techniques for verifying such manually-developed complex implementations without use of any code generation tool.

4.5 Conclusions

This chapter reviewed general software verification methods with a particular focus on verifying real-time embedded software. Both static and dynamic verification techniques have been reviewed, and their advantages and disadvantages discussed.

The discussions suggest that there are inevitable limitations in previous work to address verification problem in real-time, resource-constrained embedded systems. In particular, none of the previous work attempted to verify the embedded software in such a way that it helps to understand the impact of using various software implementations on the operational timing behaviour of the whole system. This fact prompts further research interests in this area where the timing behaviour of embedded software can be verified dynamically using simple, cheap and efficient “testing” techniques. Such techniques would be expected to explore the impact of using particular implementation decisions on the run-time behaviour of systems employing real-time scheduling algorithms.

The next chapters begin to describe and evaluate a testing technique developed in this project as one way to verify software implementations for embedded systems employing a TTC scheduling algorithm.

PART C:

SINGLE-PROCESSOR SYSTEMS

Chapter 5

TTC scheduler implementations

5.1 Introduction

As discussed in Chapter 3, the Time-Triggered Co-operative (TTC) scheduling algorithm can have a wide range of possible implementation options, each with a different set of distinctive features as well as behaviour patterns. Of course, it is not feasible to cover all possible implementation options for TTC scheduler in a single study. Thus, only a set of "representative" examples of the various classes of TTC implementations are reviewed in this chapter. Such a representative collection of TTC schedulers will be used as a basis for assessing the testing technique proposed in this project for single-processor designs.

Note that this chapter reviews six different implementations for TTC scheduler. Four implementations have been taken / modified from studies conducted previously in the ESL research group, where the remaining two are new TTC implementations developed in this project. Three further implementation options, which have less distinctive features, are outlined in Appendix D[10].

5.2 A general structure of TTC scheduler implementation

This section begins by introducing a simple approach for implementing TTC scheduler software in low-cost embedded microcontrollers and then describes the main structure used in the TTC implementations reviewed in this chapter.

[10] The work described in this chapter has been adapted from the study presented in the author's publications [1] and [3] listed in page xvi.

As in (Pont, 2001; Kurian and Pont, 2007), The majority of embedded systems run only one program where this program usually starts to execute when power is applied to the microcontroller and stops executing when the power is removed (or some error occurs). Moreover, there is no operating system returned to by the program, and allowing the program to terminate might have undesirable consequences. In order to avoid this, a form of endless "Super Loop" is usually employed (see Listing 5-1). In the example shown in the listing, the application has a "one-shot" task to be executed only once and then the program will remain in the super loop doing nothing until the whole system is reset. It is obvious that the super loop is employed mainly to "stop" the system.

```
int main(void)
   {
   Do_X();

   while(1);

   // Should never reach here
   return 1
   }
```

Listing 5-1: Use of a "Super Loop" to avoid termination of a simple embedded application.

However, the super loop can be used as the basis for implementing a simple TTC scheduler (e.g. Pont, 2001; Kurian and Pont, 2007). A possible implementation of such a scheduler is illustrated in Listing 5-2.

```
int main(void)
   {
   ...
   while(1)
      {
      TaskA();
      Delay_6ms();
      TaskB();
      Delay_6ms();
      TaskC();
      Delay_6ms();
      }

   // Should never reach here
   return 1
   }
```

Listing 5-2: A very simple TTC scheduler which executes three periodic tasks, in sequence.

By assuming that each task in Listing 5-2 has a fixed duration of 4 ms, a TTC system with a 10 ms "tick interval" has been created using a combination of super loop and delay functions. Note that if task durations are variable, then it is almost impossible to

achieve a precisely fixed tick interval with this approach, making the use of such a super-loop-based scheduler inappropriate for systems which have rigid timing constraints[11].

In general, software architectures based on super loop can be seen simple, highly efficient and portable (Pont, 2001; Kurian and Pont, 2007). However, these approaches lack the provision of accurate timing and the efficiency in using the power resources, as the system always operates at full-power which is not necessary in many applications.

An alternative (and more efficient) solution to this problem is to make use of the hardware resources to control the timing and power behaviour of the system. For example, a TTC scheduler implementation can be created using "Interrupt Service Routine" (ISR) linked to the overflow of a hardware timer. In such approaches, the timer is set to overflow at regular "tick intervals" to generate periodic "ticks" that will drive the scheduler. The rate of the tick interval can be set equal to (or higher than) the rate of the task which runs at the highest frequency (Phatrapornnant, 2007).

When the timer overflows and a tick interrupt occurs, the ISR will be called, and awaiting tasks will then be activated either from the ISR directly or from a scheduler function (this depends on the implementation class used as will be discussed in the next sections). Moreover, when not executing ISR or scheduler functions, the system is usually placed in a low-power sleep ("idle") mode in order to reduce system operating power (Pont, 2001). Most processors have idle modes, and their use can (for example) greatly increase battery life in embedded designs: however, use of idle modes is common but not essential. Once entered the idle mode, the system will only wake up when the next tick interrupt takes place.

[11] Ways in which a Super Loop approach can be used to implement a TTC system with variable task durations are discussed in detail in Appendix D. Such an implementation is referred to as TTC-SL scheduler.

Figure 5-1 illustrates a general structure of the TTC scheduler implementations considered in this chapter which is based on using the timer interrupts. The vertical arrows in the figure represent the points at which timer interrupts and, hence, ticks occur. The tick intervals are often numbered (starting from 0). The figure also shows how a system is placed in the idle mode when not executing tasks.

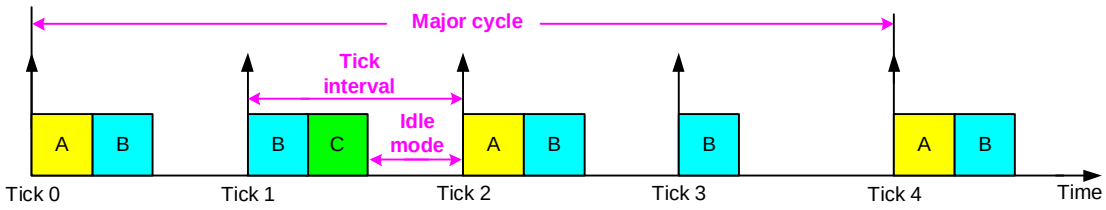


Figure 5-1: A general structure of the TTC scheduler considered in this study.

5.3 A TTC-ISR scheduler

5.3.1 Introduction

The TTC-ISR scheduler describes a very simple software implementation of the TTC scheduling algorithm. The particular implementation discussed in this section is based on that described in detail elsewhere (Pont, 2002; Kurian and Pont, 2007).

5.3.2 Overview of the scheduler implementation

As the name indicates, the basis of a TTC-ISR scheduler is an Interrupt Service Routine (ISR) which is linked to the overflow of a hardware timer. Figure 5-2 shows how such a scheduler can be implemented in software. In this example, it is assumed that one of the microcontroller's timers has been set to generate an interrupt once every 10 ms, and thereby call the function `Update()`. This `Update()` function represents the scheduler ISR. At the first tick, the scheduler will run Task A then go back to the while loop in which the system is placed in the idle mode waiting for the next interrupt. When the second interrupt takes place, the scheduler will enter the ISR and run Task B, then the cycle continues. The overall result is a system which has a 10 ms "tick interval" and three tasks executed in sequence (see Figure 5-3).

BACKGROUND PROCESSING

```
while(1)
   {
   Go_To_Sleep();
   }
```

FOREGROUND PROCESSING

10ms timer

```
void Update(void)
   {
   Tick_G++;

   switch(Tick_G)
      {
      case 1:
         Task_A();
         break;
      case 2:
         Task_B();
         break;
      case 3:
         Task_C();
         Tick_G = 0;
      }
   }
```

Figure 5-2: A schematic representation of a simple TTC-ISR scheduler.

Whether or not the idle mode is used in TTC-ISR scheduler, the timing observed is largely independent of the software used but instead depends on the underlying timer hardware (which will usually mean the accuracy of the crystal oscillator driving the microcontroller). One consequence of this is that, for the system shown in Figure 5-2 (for example), the successive function calls will take place at precisely-defined intervals, even if there are large variations in the duration of tasks which are run from the `Update()` function (Figure 5-3). This is very useful behaviour which is not easily obtained with implementations based on super loop.

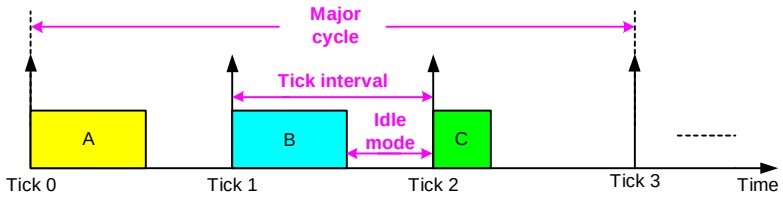


Figure 5-3: The task executions expected from the TTC-ISR scheduler code shown in Figure 5-2.

The function call tree for the TTC-ISR scheduler is shown in Figure 5-4.

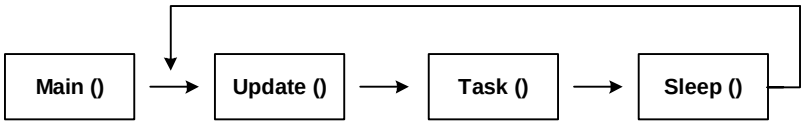


Figure 5-4: Function call tree for the TTC-ISR scheduler.

5.4 A TTC-Dispatch scheduler

5.4.1 Introduction

Implementation of a TTC-ISR scheduler requires a significant amount of hand coding (to control the task timing), and there is no division between the "scheduler" code and the "application" code (i.e. tasks). The TTC scheduler implementation referred to here as a TTC-Dispatch scheduler provides a more flexible alternative. The particular implementation discussed in this section has been adapted from an original version of TTC scheduler described in detail in (Pont, 2001) [12].

5.4.2 Overview of the scheduler implementation

The TTC-Dispatch scheduler implementation considered in this section is characterised by distinct and well-defined scheduler functions (see Listing 5-3).

[12] The modified TTC implementation considered in this section has previously been published in the author's publications [4] listed in page xvii. The original TTC scheduler as in (Pont, 2001) is described in detail in Appendix D.

```
void main(void)
   {
   // Set up the scheduler
   SCH_Init_T2();

   // Init tasks
   TaskA_Init();
   TaskB_Init();

   // Add tasks (5 ms ticks)
   // Parameters are <task name>, <offset in ticks>, <period in ticks>
   SCH_Add_Task(TaskA, 0, 3);
   SCH_Add_Task(TaskB, 1, 3);
   SCH_Add_Task(TaskC, 2, 3);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      SCH_Go_To_Sleep();
      }
   }
```

Listing 5-3: An overview of a possible TTC Dispatch scheduler implementation: see Pont (2001) for details.

Like TTC-ISR, the TTC-Dispatch scheduler is driven by periodic interrupts generated from an on-chip timer. When an interrupt occurs, the processor executes an `Update()` function (see Listing 5-5). In the scheduler implementation discussed here, the `Update()` function simply keeps track of the number of ticks. A `Dispatch()` function (Listing 5-6) will then be called, and the due tasks (if any) will be executed one-by-one. Note that the `Dispatch()` function is called from an "endless" loop placed in the function `Main()`: see Listing 5-3 and Figure 5-5. When not executing the `Update()` or `Dispatch()` functions, the system will usually enter the low-power idle mode.

In this TTC implementation, the software employs a `SCH_Add_Task()` and a `SCH_Delete_Task()` functions to help the scheduler add and/or remove tasks during the system run-time. Such scheduler architecture provides support for "one shot" tasks and dynamic scheduling where tasks can be scheduled online if necessary (Pont, 2001). To add a task to the scheduler, two main parameters have to be defined by the user in addition to the task's name: task's *offset*, and task's *period*. The offset specifies the time (in ticks) before the task is first executed. The period specifies the interval (also in ticks) between repeated executions of the task. In the `Dispatch()` function, the scheduler checks these parameters for each task before running it. Please note that information about tasks is stored in a user-defined scheduler data structure: see Listing

5-4. Both the "sTask" data type and the "SCH_MAX_TASKS" constant are used to create the "Task Array" which is referred to throughout the scheduler as "sTask SCH_tasks_G[SCH_MAX_TASKS]". See (Pont, 2001) for further details.

```
// Total memory per task is >>> bytes
typedef struct
   {
   // Pointer to the task (must be a 'void (void)' function)
   void (*pTask) (void);

   // Delay (ticks) until the function will (next) be run
   // - see SCH_Add_Task() for further details
   int Delay;

   // Interval (ticks) between subsequent runs.
   // - see SCH_Add_Task() for further details
   int Period;

   } sTask;

   ...

 // Define the maximum number of tasks
 #define SCH_MAX_TASKS   (3)
```

Listing 5-4: Data structure in the TTC-Dispatch scheduler.

The function call tree for the TTC-Dispatch scheduler is shown in Figure 5-5.

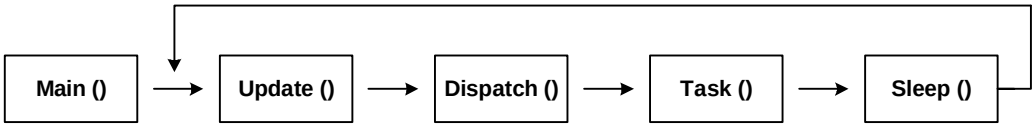


Figure 5-5: Function call tree for the TTC-Dispatch scheduler.

Figure 5-5 illustrates the whole scheduling process in the TTC-Dispatch scheduler. For example, it shows that the first function to run (after the startup code) is the `Main()` function. The `Main()`calls `Dispatch()`which in turn launches any tasks which are currently scheduled to execute. Once these tasks are complete, the control will return back to `Main()` which calls `Sleep()` to place the processor in the idle mode. The timer interrupt then occurs which will wake the processor up from the idle state and invoke the ISR `Update()`. The function calls then returns all the way back to `Main()`, where `Dispatch()` is called again and the whole cycle thereby continues.

`Update()` and `Dispatch()` codes for the TTC-Dispatch scheduler is shown in Listing 5-5 and Listing 5-6.

```
void SCH_Update(void)
   {
   // Note that an interrupt has occurred
   Tick_count_G++;

   // After interrupt, reset interrupt flag (by writing “1”)
   T0IR = 0x01;
   }
```

Listing 5-5: “Update” ISR of the TTC-Dispatch scheduler.

```
void SCH_Dispatch_Tasks(void)
   {
   int Index;
   int  Update_required = 0;

   // Need to check for a timer interrupt since this
   // function was last executed (in case idle mode is not being used)

   // Disable timer interrupt
   VICIntEnClr = 0x10;

   if (Tick_count_G > 0)
     {
     Tick_count_G--;
     Update_required = 1;
     }

   // Re-enable timer interrupts
   VICIntEnable = 0x10;

   while (Update_required)
      {
      // Go through the task array
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
         {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
               // The task is due to run
               (*SCH_tasks_G[Index].pTask)();  // Run the task

               if (SCH_tasks_G[Index].Period != 0)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
               }
            }
         }

      // Disable timer interrupt
      VICIntEnClr = 0x10;

      if (Tick_count_G > 0)
         {
         Tick_count_G--;
         Update_required = 1;
         }
      else
         {
         Update_required = 0;
         }

     // Re-enable timer interrupts
     VICIntEnable = 0x10;
     }

   }
```

Listing 5-6: Dispatch function of the TTC-Dispatch scheduler.

5.5 Applying Dynamic Voltage Scaling (DVS)

5.5.1 Introduction

In order to reduce the CPU power consumption in TTC schedulers, a Dynamic Voltage Scaling (DVS) approach can be employed. The particular implementation discussed in this section – which will be referred to as TTC-DVS – has been described in detail elsewhere (Phatrapornnant and Pont, 2006; Phatrapornnant, 2007).

5.5.2 Overview of the scheduler implementation

The full details about implementing dynamic voltage scaling in the TTC scheduler framework have been provided in the PhD thesis written by Dr. Teera Phatrapornnant, while he was working in the ESL research group (e.g. Phatrapornnant, 2007). A brief summary of this work is presented here.

Dynamic Frequency Scaling (DFS) involves reducing the operating frequency of a processor in order to reduce the power consumption when full performance is not required. DFS forms the starting point for a wide range of power-saving techniques. For example, many designs combine frequency changes and (CPU) voltage changes, resulting in what is usually referred to a Dynamic Voltage Scaling (DVS).

When employing any approaches which build on DFS, the designer faces some significant challenges, if precise control of system timing is an important consideration. This is because – in modern, general-purpose processors – the CPU core and "peripherals" (such as a timer, UART, analogue-to-digital converter, CAN module, etc) are tightly integrated onto a single chip, in order to maximize performance and minimize cost. In almost all cases, the CPU core and peripherals share a common clock source which is expected to remain largely fixed as the device operates. In the event of high-frequency changes to this clock source (as occurs when DFS-based techniques are employed) it becomes very difficult to maintain fixed timing in peripheral components

(such as timers), with the consequence that some level of jitter in task timing is unavoidable.

The key to applying DVS in a TTC application is the presence of *slack time*[13] (Phatrapornnant and Pont, 2006). Under DVS, tasks – which normally run at the same, fixed, CPU speed – will be stretched to fill the available slack time (see Figure 5-6). Therefore, the speed-setting policy is determined by the available slack time for a task (or multiple tasks) in each slot.

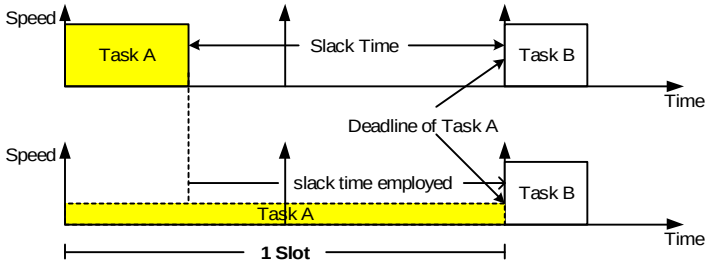


Figure 5-6: Example illustrating the possibility of task stretching in a slot (Phatrapornnant, 2007).

The TTC-DVS is applicable to periodic tasks and can be implemented using a circular array of the size equal to the number of task slots in a complete cycle to store the required CPU speed for each task. Before running tasks, the system runs the speed-finding process for a full cycle, hence calculates and stores the required speed values. Such calculations need information about the WCET and deadline of each task. The WCET is assumed to be provided by the user while the deadline of a given task is the release time of the next task. In this algorithm, the speed is only altered once per tick interval, causing all tasks in the same tick to run at the same speed.

Moreover, a reduced-jitter implementation of the TTC-DVS scheduler was developed and aimed to minimise jitter in systems using this scheduler. The low-jitter TTC-jDVS scheduler includes a timer-adjustment process to load new timer values whenever the frequency is changed. The TTC-jDVS then reduces the variation in scheduler overhead

[13] Slack time is the spare processing time during which the scheduler is in its idle state (Davis, 1993).

prior to the release of such tasks by means of a "jitter guardian", which is a form of "sandwich" delay (Pont *et al.*, 2006). Finally, TTC-jDVS deals with the problems caused by variations in the task duration by running all "reduced-jitter" tasks at the same speed every time they are released. This can (for example) reduce the impact of frequency changes on a system involving data sampling within a task.

Please note that this implementation has been described in brief. For a complete description and code listings of the TTC-DVS scheduler implementation, refer to Phatrapornnant and Pont (2006) and Phatrapornnant (2007).

Note that the scheduler structure used in TTC-DVS scheduler is same as that employed in the TTC-Dispatch scheduler which is simply based on ISR Update linked to a timer interrupt and a Dispatch function called periodically from the Main code (Section 5.4.2).

5.6 Adding Task Guardians (TGs)

5.6.1 Introduction

Despite many attractive characteristics, TTC designs can be seriously compromised by tasks that fail to complete within their allotted periods. This section reviews a TTC implementation which employs a Task Guardian (TG) mechanism to deal with the impact of such task overruns. The particular implementation discussed in this section – which will be referred to as TTC-TG – has been described in detail elsewhere (Hughes and Pont, 2004; Hughes and Pont, in press).

5.6.2 Overview of the scheduler implementation

When dealing with task overruns, the TG mechanism is required to shutdown any task which is found to be overrunning. The proposed solution also provides the option of replacing the overrunning task with a backup task (if required).

The implementation is again based on TTC-Dispatch (Section 5.4). In the event of a task overrun with ordinary Dispatch scheduler, the timer ISR will interrupt the overrunning task (rather than the `Sleep()` function). If the overrunning task keeps executing then it will be periodically interrupted by `Update()` while all other tasks will

be blocked until the task finishes (if ever): this is shown in Figure 5-7. Note that (a) illustrates the required task schedule, and (b) illustrates the scheduler operation when Task A overrun by 5 tick interval. .

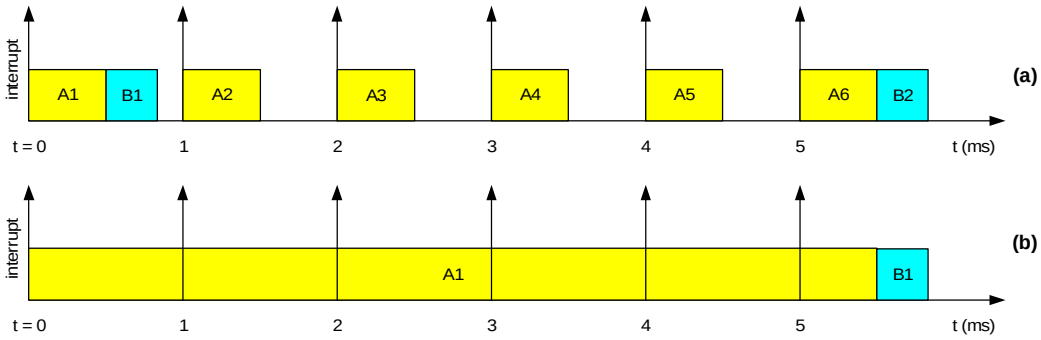


Figure 5-7: The impact of task overrun on a TTC scheduler.

In order for the TG mechanism to work, various functions in the TTC-Dispatch scheduler are modified as follows:

- `Dispatch()` indicates that a task is being executed.
- `Update()` checks to see if an overrun has occurred. If it has, control is passed back to `Dispatch()`, shutting down the overrunning task.
- If a backup task exists it will be executed by `Dispatch()`.
- Normal operation then continues.

In a little more detail, detecting overrun in this implementation uses a simple, efficient method employed in the `Dispatch()` function. It simply adds a "Task_Overrun" variable which is set equal to the task index before the task is executed. When the task completes, this variable will be assigned the value of (for example) 255 to indicate a successful completion. If a task overruns, the `Update()` function in the next tick should detect this since it checks the Task_overrun variable and the last task index value. The `Update()` then changes the return address to an `End_Task()` function instead of the overrunning task. The `End_Task()` function should return control to Dispatch. Note that moving control from `Update()` to `End_Task()` is a nontrivial process and can be done by different ways (Hughes and Pont, 2004).

The `End_Task()` has the responsibility to shutdown the overrunning task. Also, it determines the type of function that overrun and begins to restore register values accordingly. This process is complicated which aims to return the scheduler back to its normal operation making sure the overrun has been resolved completely. Once the overrun is dealt with, the scheduler replaces the overrunning task with a backup task which is set to run immediately before running other tasks. If there is no backup task defined by the user, then the TTC-TG scheduler implements a mechanism which turns the priority of the task that overrun to the lowest so as to reduce the impact of any future overrunning by this task. The function call tree for the TTC-TTG scheduler can be shown in Figure 5-5.

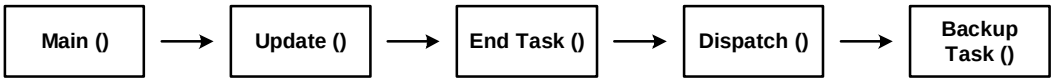


Figure 5-8: Function call tree for the TTC-TG scheduler.

Please note that this implementation has been described in brief. For a complete description and code listings of the TTC-TG scheduler implementation, refer to Hughes and Pont (2004) and Hughes and Pont (in press).

Note that the scheduler structure used in TTC-TG scheduler is same as that employed in the TTC-Dispatch scheduler which is simply based on ISR Update linked to a timer interrupt and a Dispatch function called periodically from the Main code (Section 5.4.2).

5.7 Working with Multiple Timer Interrupts (MTIs)

5.7.1 Introduction

In Chapter 2, the impact of task placement on “low-priority” tasks running in TTC schedulers was considered. The TTC schedulers described in the previous sections lack the ability to deal with jitter in the starting time of such tasks. In order to address this problem, a “gap insertion” mechanism that uses “Multiple Timer Interrupts” (MTIs) was developed and implemented in this project. The particular TTC implementation

which employs MTI technique is called TTC-MTI scheduler and described in detail in this section.

5.7.2 Overview of the scheduler implementation

In the TTC-MTI scheduler, multiple timer interrupts are used to generate the predefined execution "slots" for tasks. This allows more precise control of timing in situations where more than one task executes in a given tick interval. The use of interrupts also allows the processor to enter an idle mode after completion of each task, resulting in power saving[14].

In order to implement this technique, two interrupts are required:

- **Tick interrupt:** used to generate the scheduler periodic tick.
- **Task interrupt:** used – within tick intervals – to trigger the execution of tasks.

The process is illustrated in Figure 5-9. In this figure, to achieve zero jitter, the required release time prior to Task C (for example) is equal to the WCET of Task A plus the WCET of Task B plus scheduler overhead (i.e. ISR `Update()` function). This implies that in the second tick (for example), after running the ISR, the scheduler waits – in idle mode – for a period of time equals to the WCETs of Task A and Task B before running Task C. Figure 5-9 shows that when an MTI method is used, the periods between the successive runs of Task C (the lowest priority task in the system) are always equal. This means that the task jitter in such implementation is independent on the task placement or the duration(s) of the preceding task(s).

[14] Note that similar results can be obtained using "sandwich delays" (Pont *et al.*, 2006). However, this approach does not give such a precise control over timing and can significantly increase the levels of CPU power consumption. An example of TTC implementation – which employs sandwich delays to reduce task jitter – is described in detail in Appendix D: this is called TTC-SD scheduler.

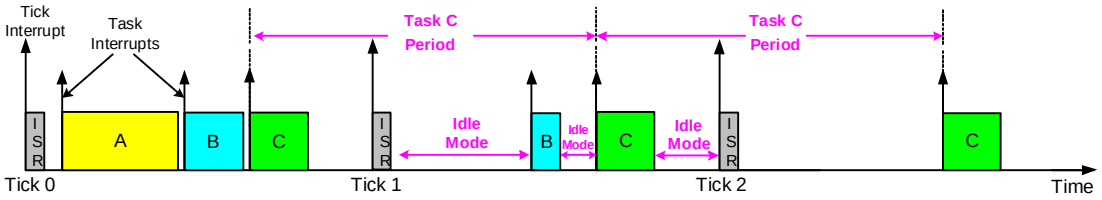


Figure 5-9: Using MTIs to reduce release jitter in TTC schedulers.

In general, it can be argued that the use of multiple timers to execute multiple tasks is not practical since most embedded microcontrollers have limited number of hardware timers (Eswaran *et al.*, 2005). Indeed, the method described here requires no more than two timers in total. Alternatively, one timer – with multiple channels – can adequately do the job. Like many modern processors, the hardware used in this study to implement this scheduler (i.e. LPC21xx microcontroller: see Section 7.2.1) supports multiple channels per timer, allowing efficient use of the available resources.

In the implementation considered in this section, the WCET for each task is input to the scheduler through `SCH_Task_WCET()` function placed in the `Main()` code. The scheduler then employs `Calc_Sch_Major_Cycle()` and `Calculate_Task_RT()` functions to calculate the scheduler major cycle and the required release time for the tasks, respectively (Listing 5-7). Moreover, there is no `Dispatch()` called in the `Main()` code: instead, "interrupt request wrappers" – which contain Assembly code – are used to manage the sequence of operation in the whole scheduler. The function call tree for the TTC-MTI scheduler is shown in Figure 5-10 (compare with Figure 5-5).

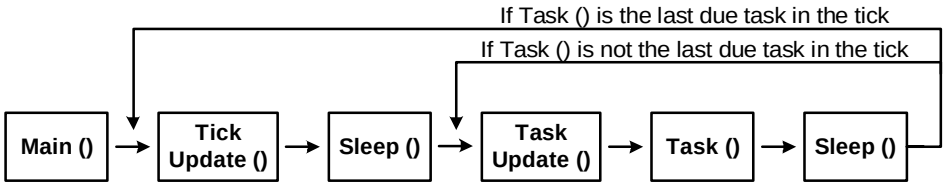


Figure 5-10: Function call tree for the TTC-MTI scheduler (in normal conditions).

Code for the TTC-MTI scheduler is shown in Listing 5-7 to Listing 5-9.

```
int main (void)
   {
   ...

   // Add tasks
   // Delay and Period values are in *ticks*
   SCH_Add_Task(Task_A, 0, 1);
   ...

   // Input duration for tasks
   // Values are in *microseconds*
   SCH_Task_WCET(Task_A, 2000);
   ...

   // Calculate the Scheduler Major Cycle
   Calc_Sch_Major_Cycle(SCH_MAX_TASKS);

   // Calculate the required release time for each task
   Calculate_Task_RT();

   // Start the scheduler
   SCH_Start();

   // The scheduler may enter idle mode at this point (if used)
   SCH_Go_To_Sleep();

   return 0;
   }
```

Listing 5-7: "Main" function in the TTC-MTI scheduler.

```
void SCH_Tick_Update(void)
   {
   int i = 0;
   int Index;

      // Go through the task array
      for (Index = 0; Index < SCH_MAX_TASKS ; Index++)
         {
           // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
               // indicate the task is to be run
               runme[i++] = Index;

               if (SCH_tasks_G[Index].Period != 0)
                 {
                 // Schedule period tasks to run again
                 SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                 }
               else
                 {
                 // Delete one-shot tasks
                 SCH_tasks_G[Index].pTask  = 0;
                 }
               }
           }
          }

          // Indicate no more tasks in runme queue
          runme[i] = SCH_MAX_TASKS;

          /* If there are tasks in current tick interval */
          if (runme[0] != SCH_MAX_TASKS)
            {
            // Setup Match Register 1 - interrupt in uS from tick
            T1MR1 = SCH_tasks_G[runme[0]].Rls_time + 50*(runme[0]+1);

            // Interrupt on match 1
            T1MCR |= 0x08;
            }

   // Return to sleep
   cTask = SCH_Go_To_Sleep;

   // Reset the task index
   Index_G = 0;
   }
```

Listing 5-8: “Update” ISR of the Tick-Timer-Interrupt in the TTC-MTI scheduler.

```
void SCH_Task_Update(void)
  {
  // Run task after this function
  cTask = SCH_tasks_G[runme[Index_G]].pTask;

  // Setup Match Register 1 - for the next task
  T1MR1 = SCH_tasks_G[runme[Index_G+1] % SCH_MAX_TASKS].Rls_time +
          50*(runme[Index_G]+2);

  // Increment task count
  Index_G++;

  // Disable Interrupt on match 1
  T1MCR &= 0xFFFFFFF7;

  // Enable Interrupt on match 1
  T1MCR |= (1 & (tLong) (runme[Index_G] != SCH_MAX_TASKS)) << 3;
  }
```

Listing 5-9: "Update" ISR of the Task-Timer-Interrupt in the TTC-MTI SCHEDULER.

Unlike the normal Dispatch schedulers, Figure 5-10, Listing 5-8 and Listing 5-9 show that the implementation relies on two interrupt `Update()` functions: `Tick Update()` and `Task Update()`. The `Tick Update()` – which is called every tick interval (as normal) – identifies which tasks are ready to execute within the current tick interval. Before placing the processor in the idle mode, the `Tick Update()` function sets the match register of the task timer according to the release time of the first due task running in the current interval. Calculating the release time of the first task in the system takes into account the WCET of the `Tick Update()` code.

When the task interrupt occurs, the `Task Update()` sets the return address to the task that will be executed straight after this update function, and sets the match register of the task timer for the next task (if any). The scheduled task then executes as normal. Once the task completes execution, the processor goes back to `Sleep()` and waits for the next task interrupt (if there are following tasks to execute) or the next tick interrupt which launches a new tick interval. Note that the `Task Update()` code is written in such a way that it always has a fixed execution duration for avoiding jitter at the starting time of tasks.

It is worth highlighting that the TTC-MTI scheduler described here employs a form of "task guardians" which help the system avoid any overruns in the operating tasks. More specifically, the described MTI technique helps the TTC scheduler to shutdown any overrunning task by the time the following interrupt takes place. For example, if the

overrunning task is followed by another task in the same tick, then the task interrupt – which triggers the execution of the latter task – will immediately terminate the overrun. Otherwise, the task can overrun until the next tick interrupt takes place which will terminate the overrun immediately.

The function call tree for the TTC-MTI scheduler – when a task overrun occurs – is shown in Figure 5-11. The only difference between this process and the one shown in Figure 5-10 is that an ISR will interrupt the overrunning task (rather than the `Sleep()` function). Again, if the overrunning task is the last task to execute in a given tick, then it will be interrupted and terminated by the `Tick Update()` at the next tick interval: otherwise, it will be terminated by the following `Task Update()`.

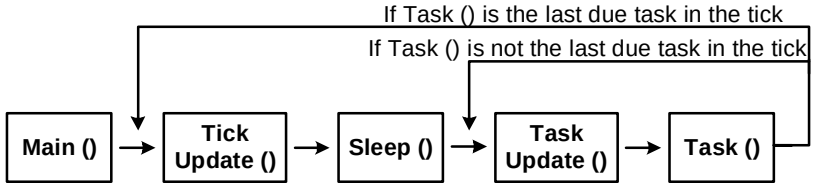


Figure 5-11: Function call tree for the TTC-MTI scheduler (with task overrun).

Please note that the complete code for this scheduler implementation is provided later in Appendix H.

5.8 Towards a "perfect" TTC implementation

5.8.1 Introduction

It can be noticed that each of the previous scheduler implementations was created to deal with one particular problem in TTC algorithm. For applications which require extremely high degree of reliability, a combinational TTC architecture – which incorporates multiple features – can be an appropriate solution. This section describes

the implementation of a highly flexible TTC implementation aimed towards achieving "perfect" time-triggered behaviour in resource-constrained embedded systems[15].

This new scheduler implementation will be called TTC-Adaptive scheduler in this thesis. This is because, unlike previous schedulers, this scheduler is self-adapted to changes in task execution times. A full description of this particular TTC implementation is provided in this section. Note that the idea behind this implementation has been developed from the concepts used in the implementations of the two schedulers described in Section 5.6 and Section 5.7 with a further (substantial) modification.

5.8.2 Overview of the scheduler implementation

The architecture of this TTC implementation was based on that used in TTC-MTI scheduler (see Figure 5-10). The present scheduler however employs a simple, but effective, mechanism for calculating the WCET of each task at the beginning of the system operating period. Remember that in the previous scheduler implementations, WCET information is input to the system by the user.

Overall, there are two different modes in which the system can operate: Calculating Mode (CM) and Operating mode (OM). Each of these modes is described as follows.

a) Calculating mode (CM)

The system runs the calculating mode for a short period of time, allowing the scheduler to perform an online calculation of the WCET for each co-operative task, and the required release time at which the task must start its execution. That is, once the system starts (power is up), the scheduler takes short time to measure the WCETs and release times of all tasks before switching into a normal operating mode. The calculating time period must be defined by the user in "number of ticks", based on system specifications.

[15] The work described in this section has been carried out in collaboration with Zemian Hughes, a member of the ESL research group.

The scheduler structure, described in Section 5.7.2, is used here but with some modification (Figure 5-12).

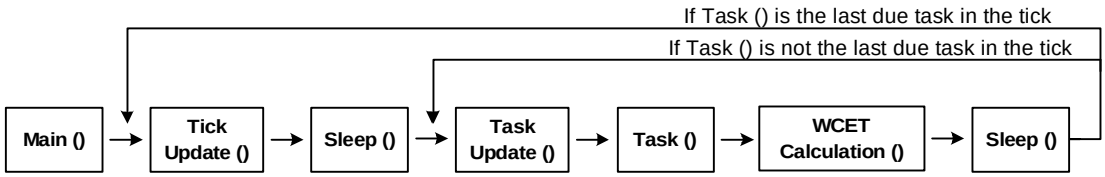


Figure 5-12: Function call tree for the TTC-Adaptive scheduler (calculating mode).

In this process, after the task is executed, `SCH_WCET()` function is called to calculate the WCET of the completed task and its release time required for low jitter characteristics. The WCET of a task is measured by recording the time just before and after the task execution (using, for example, the Timer Control Register "TCR": see Philips Semiconductors, 2003). The WCET is then calculated, in the "`SCH_WCET()`" function, by subtracting the stop time from the start time. In the same way, release time of a task is measured by recoding the time just after the `Task Update()` function begins to execute. The `SCH_WCET()` stores the maximum WCET and the maximum release time for each task in the task array. Note that the release time of the first task in the system is based on the worst case duration of the `Tick Update()` function. After calculating the WCET of the current task, the processor is placed in the idle mode for a very short period before the next task interrupt occurs (see Listing 5-10).

Please note that the WCET value computed in this algorithm is basically the longest possible execution time of the task obtained during the measurement period. As previously underlined in Section 2.10, calculating the accurate WCET of a particular activity is often a complicated process.

```
void SCH_WCET(void)
{
   tLong Duration;

   // Record Stop time
   Stop_Time = T1TC;

   // Calculate duration for no overrun
   Duration = Stop_Time - Start_Time;

   // Calculate duration of Task Update
   Task_Update_Duration = Start_Time - Release_Time;

   // If index is larger than 0
   if (Index_G)
   {
      // If the measured WCET is larger than recorded
      if (SCH_tasks_G[runme[Index_G - 1]].WCET < Duration)
      {
         // Modifiy the recorded WCET
         SCH_tasks_G[runme[Index_G - 1]].WCET = Duration+1;
      }

      // If release time is less than the tasks start time
      if (SCH_tasks_G[runme[Index_G - 1]].Req_Rls_Tm <  (Release_Time))
      {
         // Modify the release time
         SCH_tasks_G[runme[Index_G - 1]].Req_Rls_Tm = Release_Time+2;
      }

      // set the match register to current time plus little margin: this is
      // because we want the Task_Update to be called immediately
      if (runme[Index_G] != SCH_MAX_TASKS)
      {
         // Set the timer to interrupt almost immediately so we can run next
task
         // Set timer match register to current time + 4
         T1MR1 = T1TC + 4;
      }
   }

   // Disable any interrupt and send the scheduler to sleep
   SCH_End_Task();
}
```

Listing 5-10: WCET-calculation function in the TTC-Adaptive scheduler.

b) Operating mode (OM)

This relates to the normal operation mode of the scheduler. It is assumed here that the user has set the duration of the calculating mode long enough to obtain a correct set of WCET values: this must be estimated by the user based on their knowledge about the system specifications. Once the calculation time completes, the system is switched into the operating mode during which scheduled tasks run in their allotted time “slots” with no release jitter.

The function call tree for the operating mode is identical to that illustrated in Figure 5-10. Note that, without any addition to the design, the system is expected to behave in

the same way as the TTC-MTI scheduler. This means that a very simple task guardian mechanism is employed in which the scheduler allows an overrunning task to run until the next task (or tick) interrupt. This solution will be called ‘Option 1’.

However, a more effective task guardian solution is still required. One suggested way is to employ a mechanism which detects the overrun once occurred and shutdown the overrunning task immediately whether or not there are scheduled tasks to run afterwards in the same tick interval. This solution will be called ‘Option 2’. In this solution, the scheduler employs three interrupts: “Tick” interrupt and “Task” interrupt (as before) and a third interrupt called “Task Overrun” interrupt. The ISR functions for the Tick and Task interrupts (i.e. `Tick Update()` and `Task Update()`) are very similar to those used in the TTC-MTI scheduler. However, the `Tick Update()` function here keeps track of the number of ticks for the calculating mode. Once the calculation time (defined by the user) is over, the scheduler switches into operating mode.

In addition to setting the match register of the task timer to be equal to the RT of the next due task, the `Task Update()` function also sets the match register of the “task-overrun” timer to be equal to the task release time plus the task WCET plus the duration of the task update function. This simply implies that if a task exceeds its measured WCET it will be interrupted immediately by a `Task_Overrun_Update()` function which is linked to the “Task Overrun” timer interrupt. This function reports the overrun and sends the scheduler to sleep. If everything goes smoothly and no overrun occurs, an `End_Task()` function is called after the completion of each task which will simply disable the task-overrun timer interrupt and send the scheduler to sleep. Note that the `Tick Update()` function sets the return address after each task to be for the `End_Task()` function.

Figure 5-13 and Figure 5-14 illustrate the sequence of functions in ‘Option 2’ implementation with and without overrun.

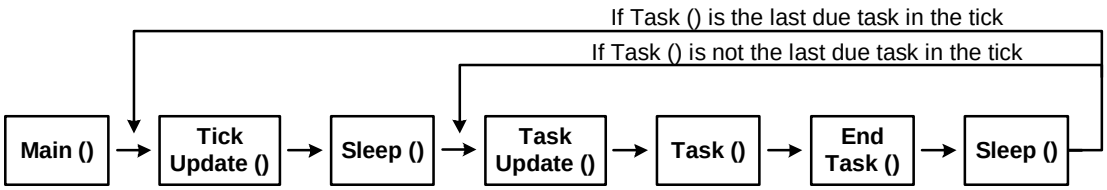


Figure 5-13: Function call tree for the TTC-Adaptive scheduler 'Option 2' (normal operation).

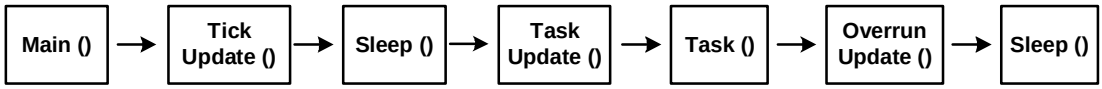


Figure 5-14: Function call tree for the TTC-Adaptive scheduler 'Option 2' (with task overrun).

In order to offer a complete task guardian mechanism, a third solution which includes support for backup tasks has been proposed: this is called 'Option 3'. In this solution, once an overrun is detected, the `Task_Overrun_Update()` function will report the overrun, set "backup" task to be the next due task to run and send the scheduler to sleep. In the next Tick interrupt, the scheduler executes the backup task before continuing to execute the following tasks (if any). Please note that the tasks which have already been executed in the tick – in which the overrun took place – will not be re-executed in the following tick. Overall, with this approach, the scheduler imposes a one-tick delay for the whole scheduler. This can still maintain a high determinism assuming that overruns occur very occasionally. The sequence of functions in 'Option 3' implementation is illustrated in Figure 5-15.

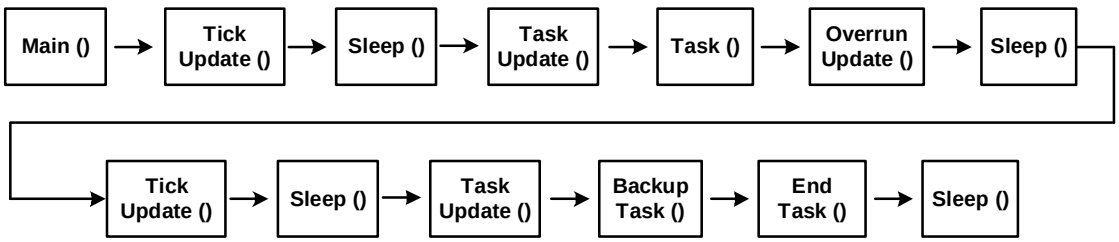


Figure 5-15: Function call tree for the TTC-Adaptive scheduler 'Option 3' (with task overrun).

The code for the TTC-Adaptive scheduler is shown in Listing 5-11 to Listing 5-14.

```
void SCH_Tick_Update(void)
   {
   tByte i = 0;
   tByte Index;
   static tWord Tick_Count = 0;

   // If tick is not paused (no overruns)
   if (!PauseTick)
      {
      // Go through the task array
      for (Index = 0; Index < SCH_MAX_TASKS - 1; Index++)
         {
           // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
               // indicate the task is to be run
               runme[i++] = Index;

               if (SCH_tasks_G[Index].Period != 0)
                 {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
                }
            }
          }

          // Indicate no more tasks in runme queue
          runme[i] = SCH_MAX_TASKS;

          /* If there are tasks in current tick interval */
          if (runme[0] != SCH_MAX_TASKS)
                 {
                 // If task is 0
                 if (runme[0] == 0)
                   {
                    // If release time is less than current time + 3
                    if (SCH_tasks_G[0].Req_Rls_Tm < (Tick_Update_Duration))
                     {
                     // Modify release time to be current + 3
                      SCH_tasks_G[0].Req_Rls_Tm = Tick_Update_Duration+3;
                     }
                   }

                 // Setup Match Register 1 - interrupt in uS from tick
                 T1MR1 = SCH_tasks_G[runme[0]].Req_Rls_Tm;

                  // Interrupt on match 1
                 T1MCR |= 0x08;
                 }
      // Reset the task index
      Index_G = 0;
      }

   // If tick is paused, set release time to backup task so that the backup
task runs
   // first and then the next tasks in the schedule can carry on as normal
   else
     {
     // Setup Match Register 1 - interrupt in uS from tick
     T1MR1 = SCH_tasks_G[runme[Index_G]].Req_Rls_Tm;

     // Interrupt on match 1
     T1MCR |= 0x08;
```

```
     // Enable tick to run next time
     PauseTick=0;
     }

   // Return to sleep
   cTask = SCH_Go_To_Sleep;

   // Keep track of the number of ticks for the calculating mode.
   //Once the calculation time (defined by the user) completes, the scheduler
goes to
   //operating (normal) mode.
   if (Mode_G == CALCULATING_MODE)
     {
    // If ticks is larger than calculation time
    if (Tick_Count++ > CALCULATION_TIME)
      {
      // Change mode to operating mode
      Mode_G = OPERATING_MODE;
      }
    }

    // If the scheduler goes into the operating mode
   if (Mode_G == OPERATING_MODE)
      {
      // Run End_Task after evry task
      mTask = SCH_End_Task;
      }

   // Record the duation of the Tick Update
   Tick_Update_Duration = T1TC;
   }
```

Listing 5-11: "Update" ISR of the Tick-Timer-Interrupt in the TTC-Adaptive scheduler.

```
void SCH_Task_Update(void)
{
   Release_Time = T1TC;

   // Run task after this function
   cTask = SCH_tasks_G[runme[Index_G]].pTask;

   // Setup Match Register 1 - for the next task
   T1MR1 = SCH_tasks_G[runme[Index_G+1]].Req_Rls_Tm;

   // Setup Match Register 2 - for WCET for end task
   T1MR2 = SCH_tasks_G[runme[Index_G]].Req_Rls_Tm +
SCH_tasks_G[runme[Index_G]].WCET
           + Task_Update_Duration + 4;

   // Increment task index
   Index_G++;

   // Disable Interrupt on match 1
   T1MCR &= 0xFFFFFFF7;

   // Enable Interrupt on match 1
   T1MCR |= (1 & (tLong) (runme[Index_G] != SCH_MAX_TASKS)) << 3;

   // Disable Interrupt on match 2
   T1MCR &= 0xFFFFFFBF;

   // Enable WCET end_task interrupt for current task
   T1MCR |= (1 & (tLong) (Mode_G == OPERATING_MODE)) << 6;

   // Record start time
   Start_Time = T1TC;
}
```

Listing 5-12: "Update" ISR of the Task-Timer-Interrupt in the TTC-Adaptive scheduler.

```
void SCH_End_Task(void)
{
   // Disable Interrupt on match 2
    T1MCR &= 0xFFFFFFBF;

   // Goto Sleep
   SCH_Go_To_Sleep();
}
```

Listing 5-13: End-Task function in the TTC-Adaptive scheduler.

```
void SCH_Task_Overrun_Update(void)
{
   // Goto sleep after ISR
   cTask = SCH_Go_To_Sleep;

   // Increment task overrun flag
   SCH_tasks_G[Index_G-1].Overrun++;

   // If there exists a backup task
   if (SCH_tasks_G[Index_G-1].bTask)
   {
      // Disable task interrupt on match 1
      T1MCR &= 0xFFFFFFF7;

      // Set backup task to run
      SCH_tasks_G[Index_G-1].pTask = SCH_tasks_G[Index_G-1].bTask;

      // Point index back to overruning task
      Index_G--;

      // Pause the next tick
      PauseTick = 1;
   }
}
```

Listing 5-14: "Update" ISR of the Task-Overrun-Interrupt in the TTC-Adaptive scheduler.

5.9 Conclusions

This chapter reviewed a selective set of implementation classes for TTC scheduling algorithm. The chapter began by a general overview of a simple TTC implementation using a few lines of software code. Such an implementation provided the introduction to a more complicated implementation options which make utilisation of the available hardware resources, such as a hardware timer, to control the system timing in a more precise manner. The description of various TTC scheduler implementations – which are based on such a concept – then followed.

It has been highlighted that the majority of the TTC implementations discussed in this chapter were taken (or adapted) from previous studies carried out in the ESL research group. Such implementations included: TTC-ISR, TTC-Dispatch, TTC-DVS and TTC-TG schedulers. Thereafter, two now implementations were presented which suggest

useful additions to the range of TTC schedulers developed within the group over the past few years. These implementations were: TTC-MTI and TTC-Adaptive schedulers.

Finally, it is important to note that this chapter provides the basis for the practical work presented in Chapter 6 and 7, in which the reviewed TTC implementations form the testbeds for assessing the effectiveness of the testing technique introduced in this project for single-processor designs.

Chapter 6

Scheduler Test Cases (STCs) for TTC schedulers

6.1 Introduction

As the introduction and literature review chapters indicate, the studies detailed in this thesis attempt to bridge the gap between scheduling algorithms and scheduler implementations when TTC software architectures are considered. This process requires an investigation of dedicated techniques that link between such system representations in a systematic way. One way of doing so is to find an appropriate method which can be applied to prove that the predictions made at the design stage of the TTC scheduler are maintained during (and after) the scheduler implementation process. In another word, the technique employed must provide an assurance that a practical TTC implementation matches the underlying characteristics of the TTC algorithm, e.g. high predictability.

In Chapter 4, it has been decided that testing (as a dynamic verification technique) is the most effective way to check the correct behaviour of many systems and, hence, gain a full confidence that those systems meet their desirable features. This was because testing allows verifying properties which can only manifest during the normal operation of the system. In TTC systems, predictable and deterministic behaviour is a key design objective. Therefore, the system must be tested with respect to its operational timing behaviour. To begin to address this issue, Scheduler Test Case (STC) technique has been developed and applied in this project. Such a testing technique is specifically intended to explore the impact of using a given TTC scheduler implementation on the predictability behaviour of the running application.

This chapter describes, in detail, the STC technique and the set of Scheduler Test Cases (STCs) developed to assess the behaviour of the TTC scheduler implementations

described in Chapter 5 for single-processor embedded designs. Remember that such implementations were identified as representative examples of the wide range of possible TTC implementation options[16].

6.2 Overview of the Scheduler Test Case (STC) technique

It is important to begin this section by underlining that the concept of testing in the STC technique, developed in this study, substantially differs from that used elsewhere (e.g. in studies reviewed in Chapter 4). For example, unlike previous studies, testing here is not aiming to check the correct functionality of the application software or evaluate its quality attributes. Instead, it is mainly used to assess the execution behaviour of the system as a result of employing a particular software implementation of TTC scheduler on generic processor hardware.

According to the discussions in Chapter 4, testing requires an appropriate set of test cases which specify the system inputs, predicted results, and execution conditions, aimed to verify (for example) the system's compliance with specific requirements. It was also mentioned that only a selective subset of possible test cases can be used as a comprehensive testing is not viable. The feature of the system to be tested must be selected along with the inputs that will execute that feature, and the expected outputs of the test cases must be known in advance. All of these test case elements have been considered in the process of developing the STCs presented in this chapter. This is further described as follows.

The STC is a simple technique which employs a collection of test cases to examine the output behaviour of a wide range of TTC scheduler implementations. The STCs developed in this study have been generated manually based on previous experience and knowledge (i.e. full understanding) about the characteristics and requirements of the TTC scheduling algorithm (see Section 2.8.3). In this sense, the STC technique is

[16] The work described in this chapter has been adapted from the study presented in the author's publication [1] listed in page xvi.

considered a component testing – as opposed to system testing – since it specifically tests the scheduler component in the whole software and that it is based on an intuitive understanding of how this particular component should operate after the whole system is integrated (see Sommerville, 2007 for more details).

The STC technique employs different scheduling examples (with "dummy tasks") that produce different behaviour patterns as the scheduler implementation varies: such task sets represent the test inputs (or test data). Each STC comprises a different set of tasks with different characteristics. Bloomfield *et al.* (2004) documented that *"The task performed by a system during execution for the purposes of running a dynamic analysis is what is known as workload or simply test cases."*

Once the tasks in each STC are input to the test item (which is, in this case, the scheduler), the system will be executed on the target hardware. The system response will then be monitored over a sufficient period of time and the output behaviour recorded and compared to the predicted behaviour (that has been documented at the test case design stage). The complete process of STC testing is illustrated in Figure 6-1.

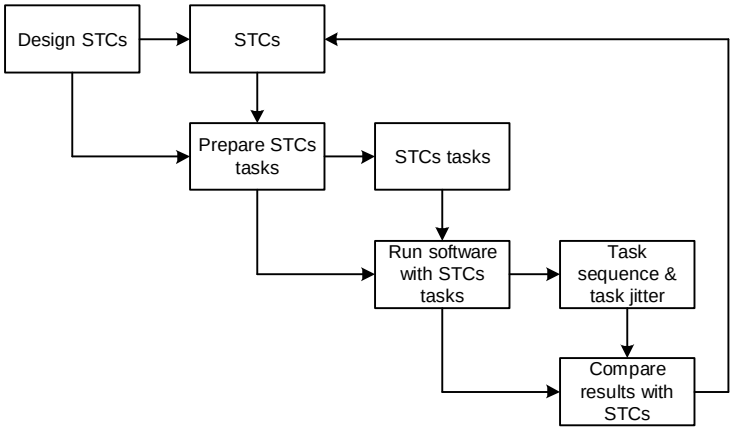


Figure 6-1: The testing process in STC technique (adapted from Sommerville, 2007).

The STC technique has been designed to test the system behaviour under both normal and abnormal operating conditions. Normal operations refer to the situations during which the scheduler operates in an absence of any errors, while abnormal operations relate to the occurrence of errors. The error mode in any scheduling algorithm, for which the STCs are developed, must be defined by the developer where it has to

represent acknowledged problem(s) facing the implementers of such an algorithm. For example, in TTC systems, “task overrun” is a major problem which can cause measurable degradation in the system performance or jeopardise the system functionality (Section 2.10). Therefore, task overrun has been used in this study to define the error mode of the TTC scheduler.

The key criteria against which the TTC scheduler behaviour is assessed include the task sequencing, jitter and ability to deal with overruns: these are the main tested features of the TTC scheduling algorithm in this study. Such criteria have been used as a practical means to assess the predictability of the TTC schedulers. In more detail, the task sequencing checks whether the scheduler executes the required tasks in the required order. Jitter in the task timing is used to assess the timing performance of the system. In Chapter 2, jitter was defined as variations in the timing of tasks. Three different jitter types were also listed: release jitter, execution jitter and finishing jitter. In this study, the jitter is considered at the release time of tasks running in each TTC scheduler. Release jitter – as in Chapter 2 – describes the deviation of the start time of a task from its release time. Remember that tasks with low-jitter characteristics can lead to highly-predictable behaviour in many embedded system designs. In addition to task sequencing and jitter tests, the STC tests the capability of the scheduler to handle task overrun error. The system can be more predictable if it is able to reduce the impact of such an error.

6.3 The Scheduler Test Cases (STCs) for TTC algorithm

6.3.1 Introduction

This section describes the STCs developed in this study for TTC algorithm. The total number of STCs described here is four. More specifically, STC A, STC B and STC C are intended to test the system behaviour under normal operating conditions, where STC D was intended to test the system behaviour during the occurrence of error. Each STC was aimed to address a different type of problem which might have a negative impact on the overall system predictability.

6.3.2 STC A (Task-induced jitter)

STC A explores the potential impact of variations in the execution time of tasks on the jitter levels of subsequent tasks in the schedule.

A summary of the task characteristics for this STC is presented in Table 6-1 and schematically illustrated in Figure 6-2.

Considering STC A in more detail, all tasks execute with a "tick offset" of 0: that is, each task executes for the first time in tick interval 0 and continues to execute in each tick interval.

Table 6-1: Task set (test input) for STC A (Major cycle = 1 Tick).

Task Name	Period (Ticks)	Offset (Ticks)	Priority (1 = High)	ET_{17}	Allowable jitter in start time of task
A	1	0	1	ET(A) – variable (0.01 - 0.4 Ticks)	Low
B	1	0	2	ET(B) – variable (0.01 - 0.2 Ticks)	Low
C	1	0	3	ET(C) – variable (0.01 - 0.2 Ticks)	Low

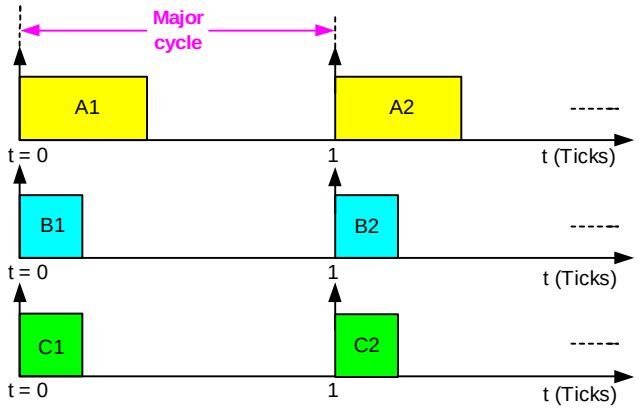


Figure 6-2: Graphical representation of the task set in STC A.

Examples of possible schedules obtained with this task set (using different implementations) are given in Table 6-2 and Table 6-3.

[17] ET denotes the actual execution time of a task on a given run (this figure will vary between runs in most cases).

Table 6-2: Example schedule A1

	Start time (after due tick)	Jitter
Ax	0	Low (related to Tick jitter + scheduler overhead)
Bx	ET(Ax)	Potentially high (varies with ET of preceding task)
Cx	ET(Ax) + ET(Bx)	Potentially high (varies with ET of preceding tasks)

Comment:

In a basic scheduler implementation, it is likely to see significant levels of jitter in the start times of tasks executed later in the tick interval, if the execution time of the earlier tasks varies. This is illustrated in Figure 6-3. Obviously, such a scheduler implementation is not suitable for use with jitter-sensitive tasks.

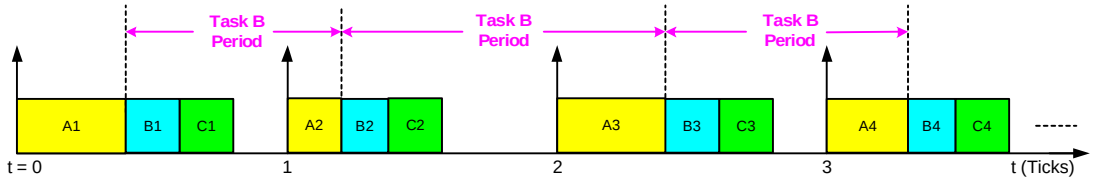


Figure 6-3: Graphical representation of Example schedule A1.

Table 6-3: Example schedule A2

	Start time (after due tick)	Jitter
Ax	0	Low (may be related to scheduler overhead)
Bx	WCET (A)	Low (may be related to scheduler overhead)
Cx	WCET(A) + WCET(B)	Low (may be related to scheduler overhead)

Comment:

In a low-jitter scheduler implementation, the scheduler will compensate for variations in the execution time of tasks. Lower priority tasks in the schedule will have low-jitter characteristics. This is illustrated in Figure 6-4.

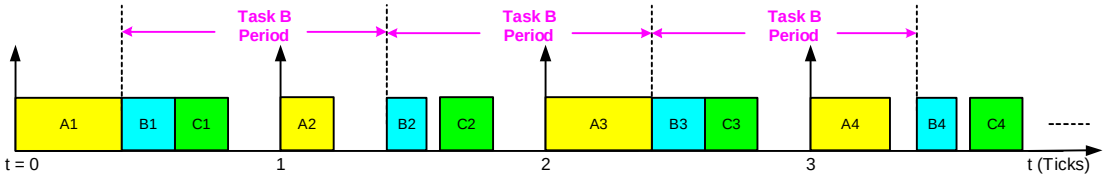


Figure 6-4: Graphical representation of Example schedule A2.

It is clear from Figure 6-4 that with this schedule, Task B (and hence Task C) will be free of jitter if the scheduler overhead is fixed.

6.3.3 STC B (Schedule-induced jitter)

STC A explores the potential impact of variations in the execution time of tasks on the jitter levels in the system. By contrast, STC B explores the potential impact of variations in the schedule on the jitter levels of tasks.

A summary of the task characteristics for this STC is presented in Table 6-4 and schematically illustrated in Figure 6-5.

Table 6-4: Task set (test input) for STC B (Major cycle = 2 Ticks).

Task Name	Period (Ticks)	Offset (Ticks)	Priority (1 = High)	ET	Allowable jitter in start time of task
A	2	0	1	ET(A) – variable (0.01 - 0.4 Ticks)	Low
B	1	0	2	ET(B) – variable (0.01 - 0.2 Ticks)	Low
C	1	0	3	ET(C) – variable (0.01 - 0.2 Ticks)	Low

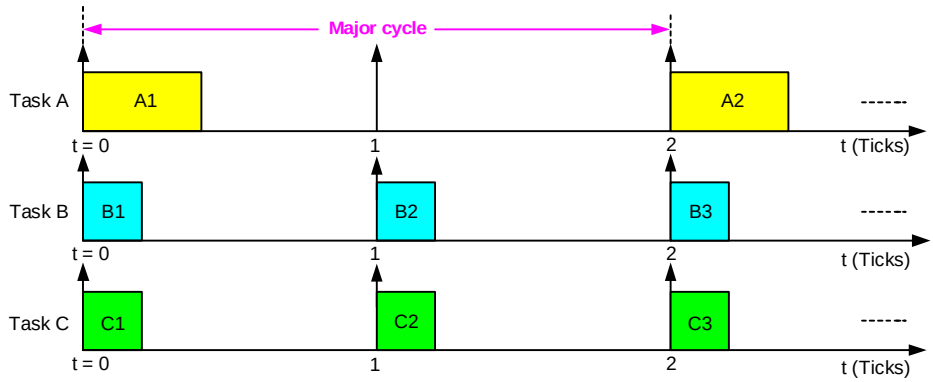


Figure 6-5: Graphical representation of the task set in STC B.

Examples of possible schedules obtained with this task set (using different implementations) are given in Table 6-5 and Table 6-6.

Table 6-5: Example schedule B1 (Basic scheduler)

	Start time (after due tick)	Jitter
Ax	0	Low (related to Tick jitter + scheduler overhead)
Bx	0 or ET(Ax)	High (start time of task varies on alternate Ticks)
Cx	ET(Bx) or ET(Ax) + ET(Bx)	High (start time of task varies on alternate Ticks)

Comment:

Here, Task B will suffer from high levels of release jitter (because it executes sometimes after Task A and sometimes at the start of the tick: Figure 6-6).

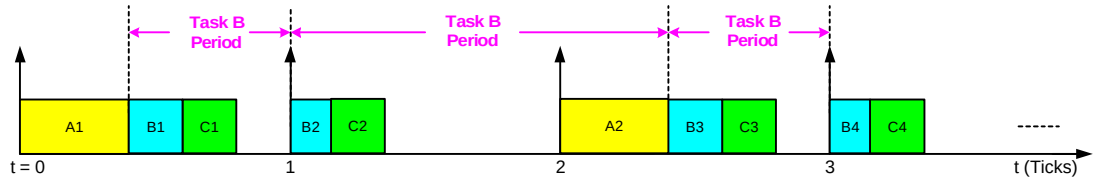


Figure 6-6: Graphical representation of Example schedule B1.

Table 6-6: Example schedule B2 (TTC scheduler with gap insertion)

	Start time (after due tick)	Jitter
Ax	0	Low (related to Tick jitter + scheduler overhead)
Bx	WCET(A)	Low (if WCET estimates are accurate)
Cx	WCET(A) + WCET (B)	Low (if WCET estimates are accurate)

Comment:

This low-jitter scheduler implementation satisfies the jitter requirements for Task B and Task C (Figure 6-7). This is because low-priority tasks always run in fixed time slots independent on any preceding task executions.

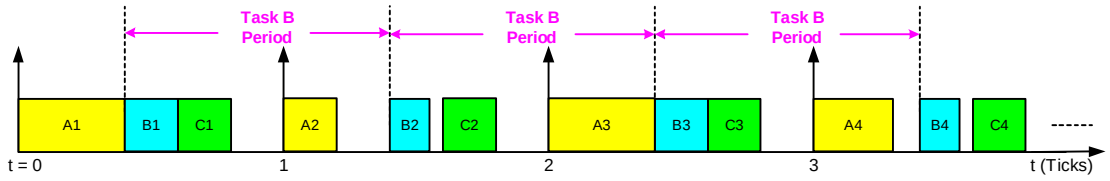


Figure 6-7: Graphical representation of Example schedule B2.

6.3.4 STC C (Long tasks)

The majority of TTC scheduler implementations (including all of those considered in this study) involve the use of scheduler ticks generated by means of a periodic timer overflow, linked to an interrupt service routine. In STC A and STC B, it is assumed that all tasks which begin execution in a given tick interval will be intended to complete

their execution before the next tick occurs. Such a restriction is not an essential requirement in TTC designs[18], but can be a limiting factor in some TTC implementations.

The scheduler's ability to handle "long tasks" is tested in STC C. A summary of the task characteristics for this test is presented in Table 6-7 and schematically illustrated in Figure 6-8.

Table 6-7: Task set (test input) for STC C (Major cycle = 4 Ticks).

Task Name	Period (Ticks)	Offset (Ticks)	Priority (1 = High)	ET	Allowable jitter in start time of task
A	2	1	1	ET(A) – fixed (0.2 Ticks)	Low
B	4	0	2	ET(B) – fixed (2.4 Ticks)	Low
C	2	1	3	ET(C) – fixed (0.2 Ticks)	High

Comment

In this task set, Task B runs for 2.4 ticks. During this time, Task A (assumed to be a low-jitter task) becomes due to run. In this STC, it can be determined how the scheduler will deal with tasks which are (deliberately) designed to have durations longer than the tick interval. It can also be determined how the scheduler manages task priorities: Task A has a higher priority than Task C and – following completion of Task B – Task A should execute before Task C.

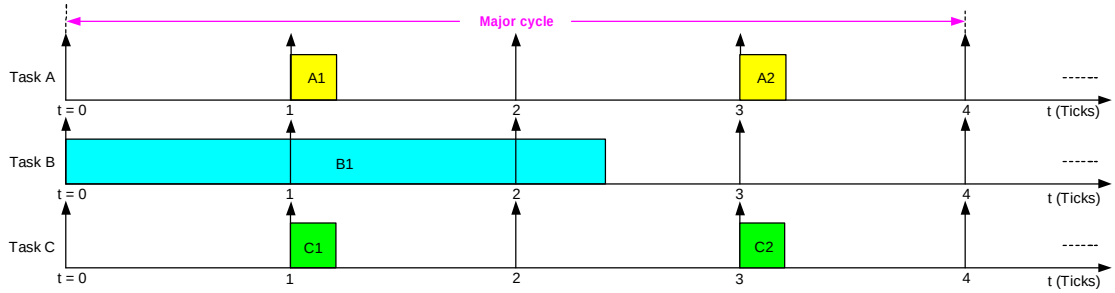


Figure 6-8: Graphical representation of the task set in STC C.

Examples of possible schedules obtained with this task set (using different implementations) are given in Table 6-8 to Table 6-13.

[18] A TTC design is co-operative in nature. Pre-emption of one task by another is not permitted. However, in the case of "long tasks", a task is interrupted by the scheduler (not by another task).

Table 6-8: Example schedule C1 (Basic scheduler)

	Start time (after due tick)	Jitter
Ax	0 or 1.4 Ticks	High (start time of task varies on alternate Ticks)
Bx	0	Low (related to Tick jitter + scheduler overhead)
Cx	ET(Ax) or ET(Ax) + 1.4 Ticks	High (start time of task varies on alternate Ticks)

Comment

This behaviour (illustrated in Figure 6-9) will be expected from a basic TTC scheduler.

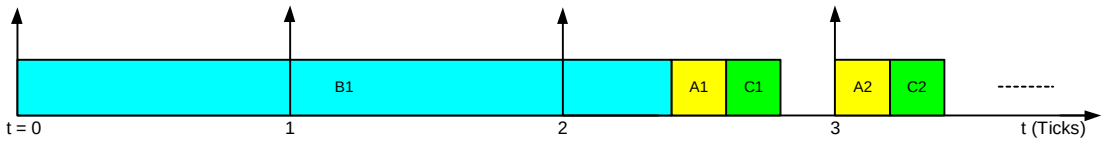


Figure 6-9: Graphical representation of Example schedule C1.

Table 6-9: Example schedule C2

	Start time (after due tick)	Jitter
Ax	0 or ET(Cx) + 1.4 Ticks	High (start time of task varies on alternate Ticks)
Bx	0	Low (related to Tick jitter + scheduler overhead)
Cx	ET(Ax) or 1.4 Ticks	High (start time of task varies on alternate Ticks)

Comment

This behaviour will be observed with many TTC implementations which check each task, in sequence, to see if they are due to run: in this case, Task C's status is tested, and the task is executed, before the status of Task A is tested. This is illustrated in Figure 6-10

.

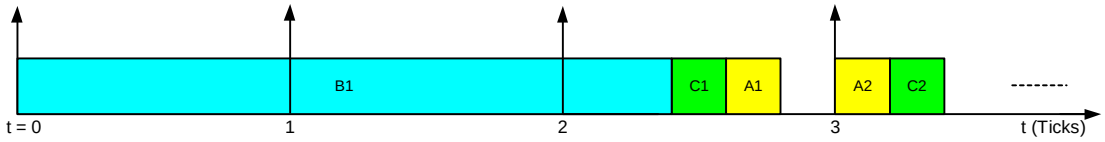


Figure 6-10: Graphical representation of Example schedule C2.

Table 6-10: Example schedule C3

	Start time (after due tick)	Jitter
Ax	0	Low (related to Tick jitter + scheduler overhead)
Bx	0	Low (related to Tick jitter + scheduler overhead)
Cx	ET(Ax) or 1.4 Ticks	High (start time of task varies on alternate Ticks)

Comment

This behaviour can also be observed with many TTC implementations which check the status of all tasks before beginning to execute the due tasks: in this case, after completing Task B, Task C is executed while Task A is omitted from the schedule. So, although jitter in Task A is low, its period is doubled, a result which may not be tolerated in many systems. This is illustrated in Figure 6-10.

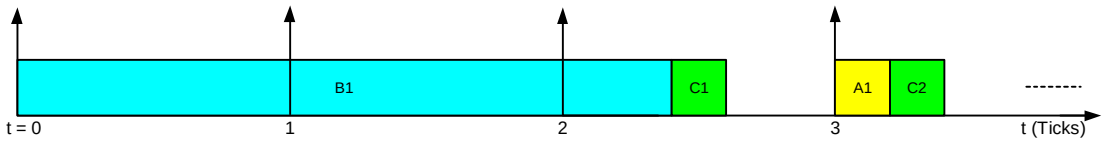


Figure 6-11: Graphical representation of Example schedule C3.

Table 6-11: Example schedule C4

	Start time (after due tick)	Jitter
Ax	0	Low (related to Tick jitter + scheduler overhead)
Bx	0	Low (related to Tick jitter + scheduler overhead)
Cx	ET(Ax)	Low (since ET(Ax) is fixed)

Comment

In each major cycle, the first execution of both Task A and Task C is omitted from the schedule. So, although jitter in Task A and Task C is low, their periods are doubled. This is illustrated in Figure 6-12.

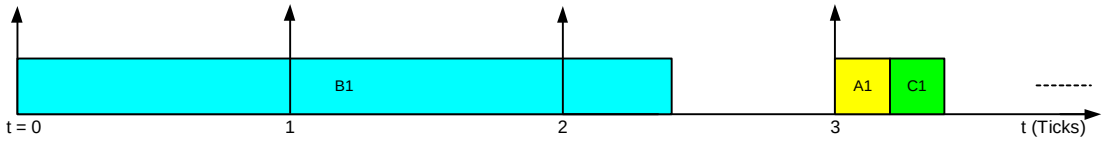


Figure 6-12: Graphical representation of Example schedule C4.

Table 6-12: Example schedule C5

	Start time (after due tick)	Jitter
Ax	2 Tick or ET(Ax)	High (task runs twice in the same Tick at different start times)
Bx	0	Low (related to Tick jitter + scheduler overhead)
Cx	1.4 Tick or 2*ET(Ax)	High (start time of task varies on alternate Ticks)

Comment

The first execution of Task A is delayed by one tick. Thus, Task A will run twice in the following tick before Task C runs (see Figure 6-13).

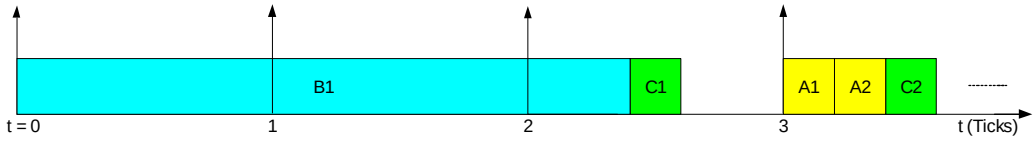


Figure 6-13: Graphical representation of Example schedule C5.

Table 6-13: Example schedule C6

	Start time (after due tick)	Jitter
Ax	0	Low (related to Tick jitter + scheduler overhead)
Bx	0	Low (related to Tick jitter + scheduler overhead)
Cx	ET(Ax)	Low (since ET(Ax) is fixed)

Comment

The scheduler shuts down any task still executing when the next tick occurs (see Figure 6-14).

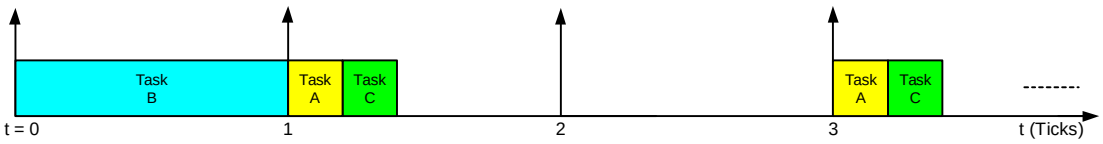


Figure 6-14: Graphical representation of Example schedule C6.

6.3.5 STC D (Task overruns)

STC A, STC B and STC C all assume normal system operation. The goal with STC D is to explore the potential impact of unplanned task overruns.

A summary of the task characteristics for this test is presented in Table 6-14 and schematically illustrated in Figure 6-15.

Table 6-14: Task set (test input) for STC D (Major cycle = 20 Ticks).

Task Name	Period (Ticks)	Offset (Ticks)	Priority (1 = High)	ET	Overrun duration (in Ticks)
A	20	0	1	ET(A) – fixed (0.2 Ticks)	10
B	1	0	2	ET(B) – fixed (0.2 Ticks)	0

Comment

In this task set, Task A is designed to run for the duration of 0.2 Tick. When an error occurs, Task A overruns by 10 Ticks. This is illustrated in Figure 6-15.

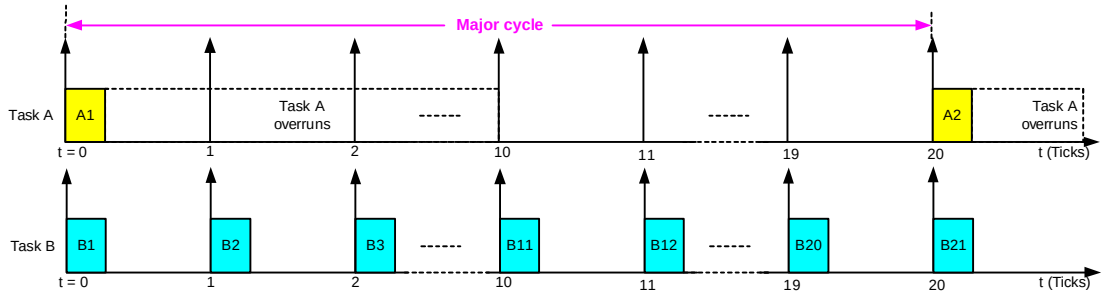


Figure 6-15: Graphical representation of the task set in STC D.

Examples of possible schedules obtained with this task set (using different implementations) are given in Table 6-15. Note that jitter characteristics are not considered in this STC as such values would have no meaning in this test.

Table 6-15: Example schedule D1a, D1b, D2a, D2b, D3a and D3b

Schedule Name	Shut down time (after Ticks)	Backup task	Comment
D1a	---	Not applicable	Overrunning task is not shut down. The number of elapsed ticks – during overrun – is not counted and therefore tasks due to run in these ticks are ignored.
D1b	---	Not applicable	Overrunning task is not shut down. The number of elapsed ticks – during overrun – is counted and therefore tasks due to run in these ticks are executed immediately after overrunning task ends.
D2a	1 Tick	Not available	Overrunning task is detected at the time of the next tick and shut down.
D2b	1 Tick	Available – BK(A)	Overrunning task is detected at the time of the next tick and shut down: a replacement (backup) task is added to the schedule.
D3a	WCET(Ax)	Not available	Overrunning task is shut down immediately after it exceeds its estimated WCET.
D3b	WCET(Ax)	Available – BK(A)	Overrunning task is shut down immediately after it exceeds its estimated WCET. A backup task is added to the schedule.

6.3.6 CPU, memory and power requirements

In resource-constrained embedded systems (of the type considered in this project), designers are frequently concerned about CPU and memory requirements. These requirements are therefore reported for all of the schedulers considered in this study.

In mobile applications (for example), average power consumption is also a key concern, as this is related to the system battery life (Phatrapornnant, 2007). The average power consumption figures will therefore be reported for all schedulers considered in this study.

6.4 Conclusions

This chapter described the STC technique and the set of Scheduler Test Cases (STCs) developed in this project to test the run-time behaviour of the various TTC scheduler implementations considered in Chapter 5. The chapter highlighted the key criteria against which the TTC scheduler behaviour will be assessed. These include task sequencing, jitter and ability to deal with a task overrun. Specifically, task jitter was given a particular consideration as a key practical measure for evaluating the system predictability. It is useful to highlight – at this point – that while jitter has been widely discussed in the literature, the impact of particular scheduler implementations on jitter behaviour has not received widespread attention. Thereby, the current study attempts to address this issue in greater depth.

Through the application of the various STCs described in this chapter, the study presented in this thesis was intended to facilitate a meaningful comparison between the different behaviour patterns in a wide range of “standard” TTC scheduler implementations. The results obtained from a practical application of the STC technique are provided in the next chapter (Chapter 7).

Chapter 7
Assessing the behaviour of TTC scheduler implementations

7.1 Introduction

In Chapter 5, a set of representative implementation classes of the TTC scheduling algorithm was described. Chapter 6 then described a set of generic "scheduler test cases" (STCs) used to facilitate a meaningful comparison between the various TTC schedulers.

This chapter provides the output results obtained when the described STCs are employed in each TTC implementation considered. The aim of this chapter is to show the effectiveness of the proposed STC technique in assessing (and distinguishing) the behaviour of the various implementation classes of TTC scheduler.

The chapter begins by describing the experimental methodology used to obtain the results presented later in the chapter[19].

7.2 Experimental methodology

7.2.1 Hardware platform

It is assumed in this project that the target platform for the embedded system is a small microcontroller (e.g. 8051, Infineon C16x, Philips LPC2xxx, or PH Processor: Hughes *et al.*, 2005) which will be programmed in the C language.

[19] The work described in this chapter has been adapted from the study presented in the author's publications [1] and [3] listed in page xvi.

In particular, the empirical studies reported in this thesis for the single-processor systems were conducted using Ashling LPC2000 evaluation board supporting Philips LPC2106 processor (Ashling Microsystems, 2007). The LPC2106 is a modern 32-bit microcontroller with an ARM7 core which can run – under control of an on-chip PLL – at frequencies from 12 MHz to 60 MHz (Philips Semiconductors, 2003). Except where otherwise noted, the processor used an oscillator frequency of 12 MHz, and a CPU frequency of 60 MHz.

The compiler used was the GCC ARM 4.1.1 operating in Windows by means of Cygwin (a Linux emulator for windows). The IDE and simulator used was the Keil ARM development kit (v3.12).

7.2.2 Task sequencing and overrun tests

In each TTC scheduler implementation, the task sequencing – in both normal and abnormal operations – was measured directly from the simulator by using breakpoints at each task to observe the order (and the tick) at which the tasks execute. The measurements were taken over a number of successive major cycles to ensure that the observed behaviour is repetitive. The results obtained when executing each STC were then reported and compared to the example schedules discussed in Section 6.3.

7.2.3 Jitter tests

In this test, two jitter measures were recorded:

- **Tick jitter:** represented by the variations in the interval between the release times of the periodic tick.
- **Task jitter:** represented by the variations in the interval between the release times of periodic tasks.

To obtain a meaningful set of task jitter results, Task A, Task B and Task C were set to have variable durations in STC A and STC B. In STC C, the impact of long tasks on the jitter levels of the scheduler tick and tasks were studied. It should be noted that the jitter levels are only considered when the scheduler operates in normal conditions. Therefore, in STC D – where errors relate to task overrun take place – jitter levels are not discussed.

In order to measure the jitter on the tick and tasks experimentally, a pin was set high at the beginning of the tick or task (for a short time) then the periods between every two successive rising edges were measured (Figure 6-1). In each experiment, 5000 samples were recorded: this was found sufficient for the purpose of this study. The periods were measured using a National Instruments data acquisition card 'NI PCI-6035E' (National Instruments, 2006), used in conjunction with appropriate software LabVIEW 7.1 (LabVIEW, 2007).

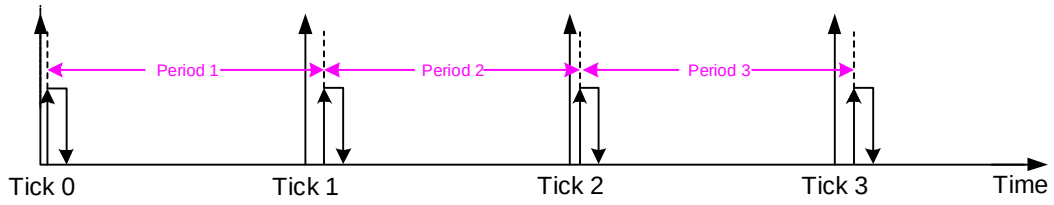


Figure 7-1: The technique used to measure release jitter in tick.

To assess the jitter levels, two values were reported:

- **Difference jitter:** obtained by subtracting the minimum period from the maximum period obtained from the measurements in the sample set. This jitter is sometimes referred to as "absolute jitter" (Buttazzo, 2005).
- **Average jitter:** represented by the standard deviation in the measure of average periods.

Note that there are many other measures that can be used to represent the levels of task jitter, but these measures were felt to be appropriate for this study.

7.2.4 CPU test

The CPU overhead is one of the cost parameters that have been used to differentiate between different TTC scheduler implementations. To obtain CPU overhead measurements in each scheduler, STC A was run for 25 seconds and then, using the performance analyser supported by the Keil simulator, the total time required by the scheduler in the measurement period was measured (Figure 7-2). The percentage of the measured CPU time was then reported to indicate the overhead (i.e. computational cost) required in each scheduler implementation.

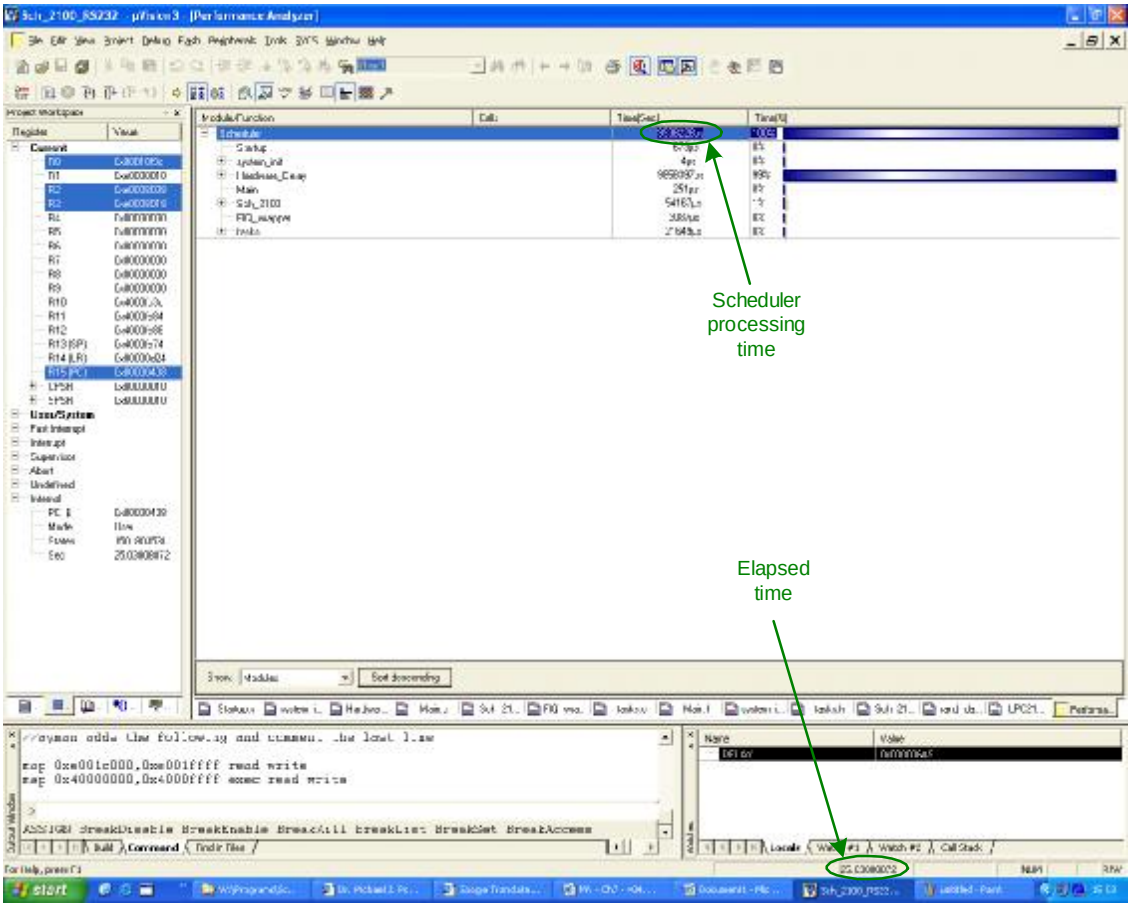


Figure 7-2: Measuring CPU overhead in the Keil simulator.

7.2.5 Memory test

In this test, the CODE and DATA memory values required to implement STC A for each scheduler were recorded. Note that these figures are independent of the STC used. Memory values were obtained using the ".map" file which is created when the source code is compiled (Figure 7-3 and Figure 7-4).

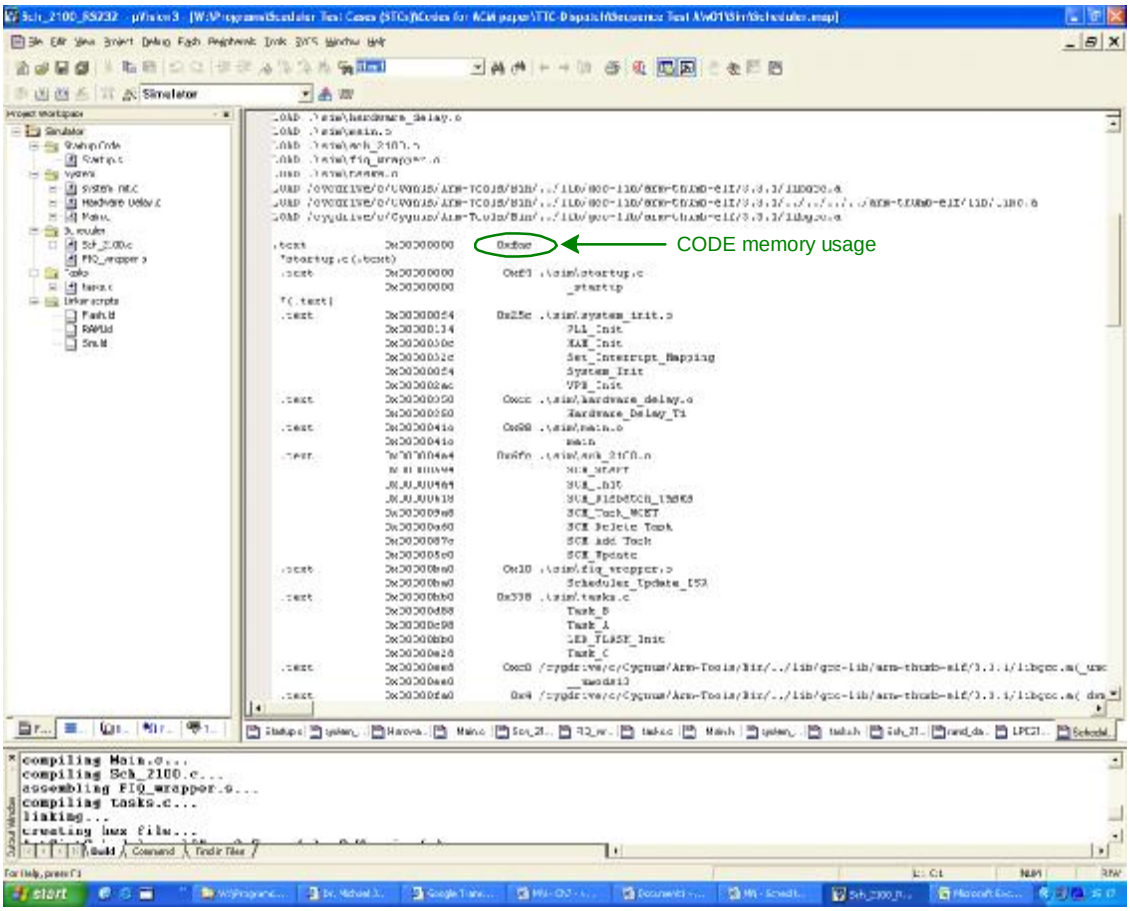


Figure 7-3: Measuring CODE memory overhead from the ".map" file.

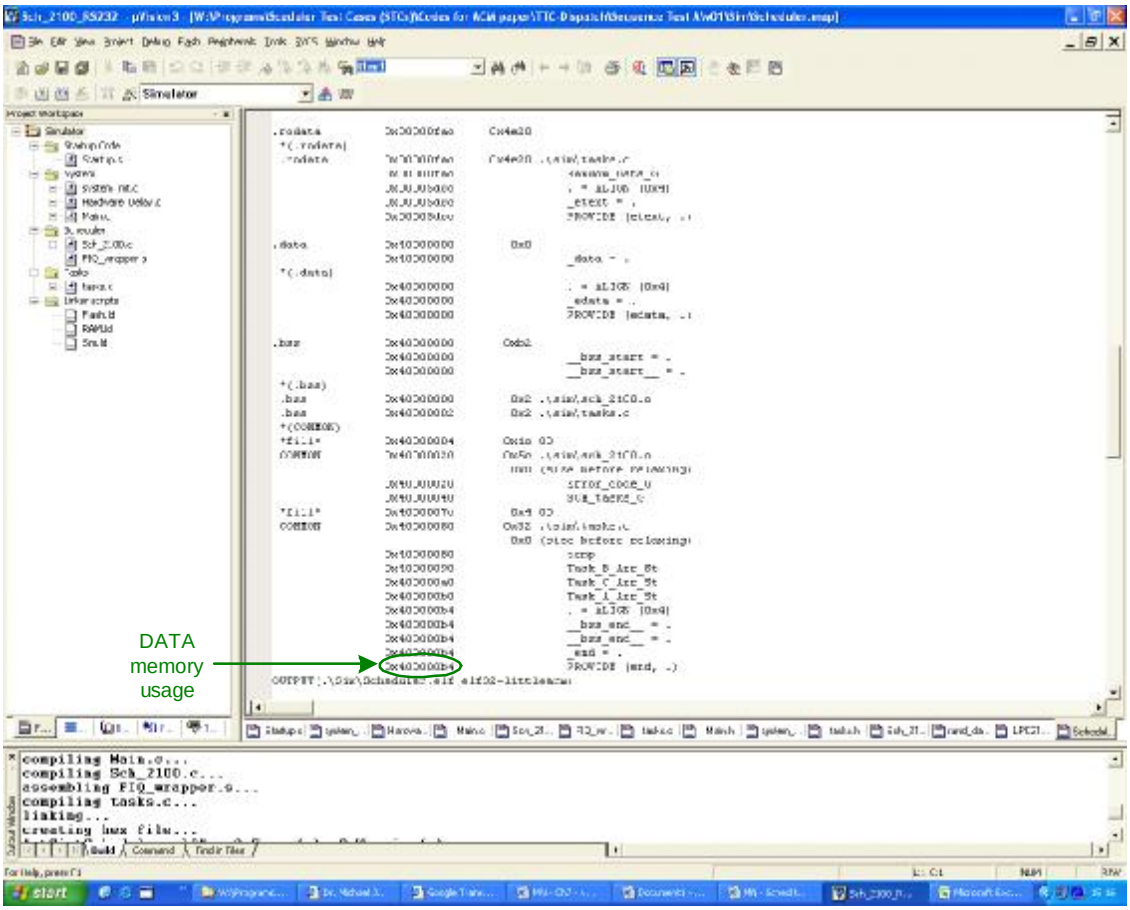


Figure 7-4: Measuring DATA memory overhead from the ".map" file.

The STACK usage was also measured (as DATA memory overhead) by initially filling the data memory with 'DEAD CODE' and then reporting the number of memory bytes that had been overwritten after running the scheduler for sufficient period (Figure 7-5).

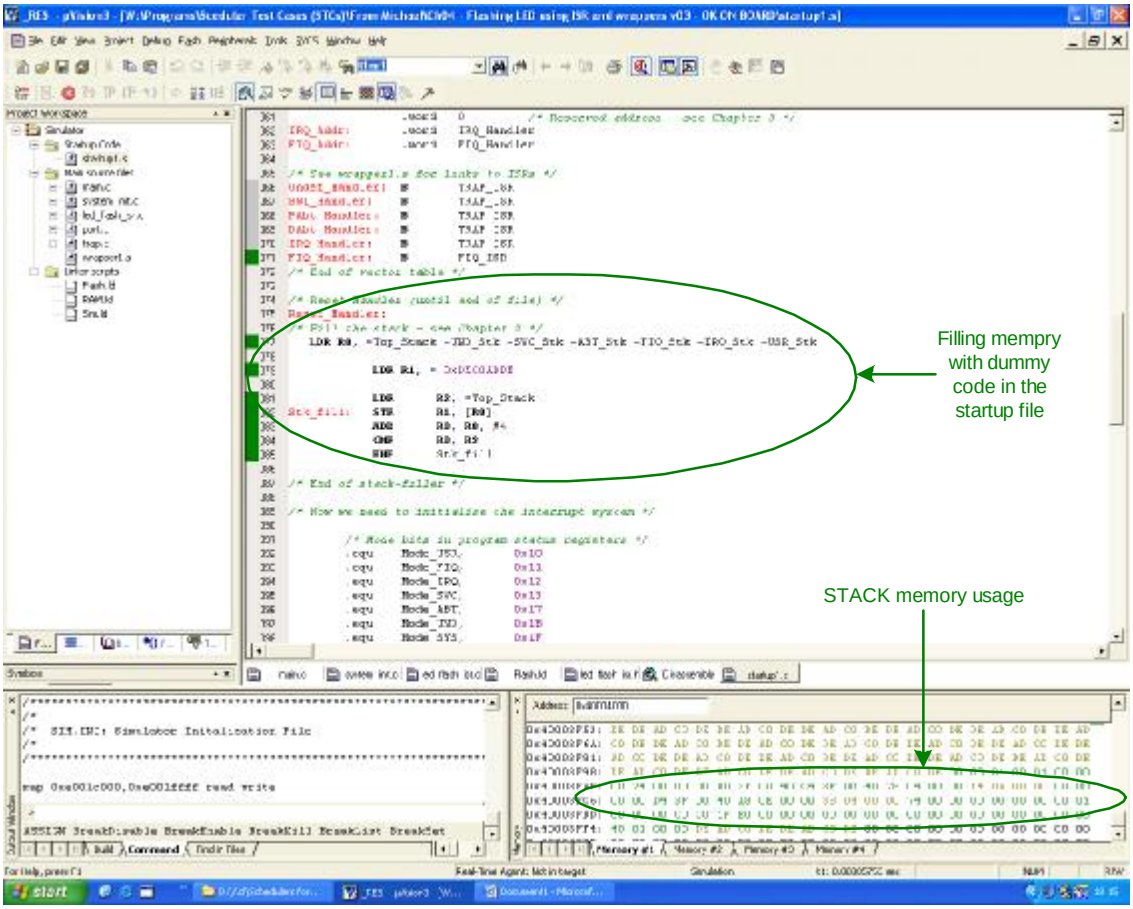


Figure 7-5: Measuring STACK overhead from the Keil simulator.

7.2.6 Power test

To obtain representative values of power consumption, the input current and voltage to the LPC2106 CPU core were measured while executing STC A and STC B: this is because the power measures vary as the task schedule varies (from one STC to another). Figure 7-6 shows one way of measuring the CPU power consumption in the embedded designs considered in this study. Again, the voltage measurements were obtained by using the National Instruments data acquisition card 'NI PCI-6035E' in conjunction with LabVIEW 7.1 software. The sampling rate of 10 KHz was used over a period equal

to 5000 major cycles. Values for currents and voltages were then multiplied and then averaged out to give the power figures presented in power result tables[20].

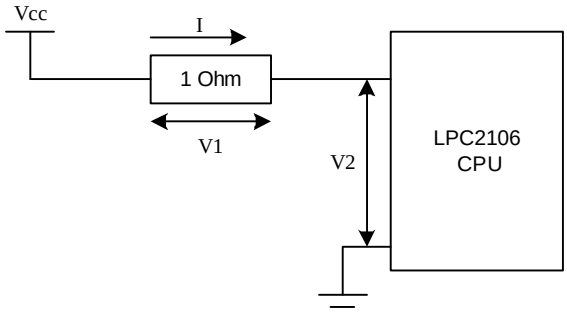


Figure 7-6: The circuit used to measure the system power consumption in each TTC scheduler.

7.3 Results

7.3.1 Applying STC to the TTC-ISR scheduler

This section discusses the implementation of STCs in the TTC-ISR scheduler and presents the output results from such an implementation.

7.3.1.1 Implementing the test cases

Implementing STC A and STC B with the TTC-ISR scheduler was straightforward (and very similar to the example shown in Figure 5-2). Listing 7-1 and Listing 7-2 show how STC C and STC D were implemented, respectively, using a TTC-ISR scheduler.

[20] The method used to obtain the power results was suggested by Dr. Teera Phatrapornnant, an ex-member of the ESL research group working on reducing power consumption in low-cost embedded systems (Phatrapornnant, 2007).

```
void ISR_Update(void)
   {
   switch(Tick_G)
     {
     case 0:
     // Treat first three ticks as one long interval
     Task_B(); // Long task (lose 2 ticks)
     Task_A();
     Task_C();
     Tick_G = 3;
     break;

     case 3:
     // Add Tasks in the fourth tick interval
     Task_A();
     Task_C();

     // Reset Tick count
     Tick_G = 0;
     }

   // After interrupt, reset interrupt flag (by writing “1”)
   T0IR = 0x01;
   }
```

Listing 7-1: One way of implementing STC C using the TTC-ISR scheduler.

```
void ISR_Update(void)
   {
   switch(Tick_G)
      {
      case 0:
      // Add Tasks in the first tick interval
      Task_A();
      Task_B();
      Tick_G++;
      break;

      default:
      // Add Tasks in the rest of tick intervals
      Task_B();
      break;
      }

   // Reset Tick count after 20 ticks to start a new cycle
   Tick_G %= 20;

   // After interrupt, reset interrupt flag (by writing “1”)
   T0IR = 0x01;
   }
```

Listing 7-2: One way of implementing STC D using the TTC-ISR scheduler.

The WCETs of tasks were defined as constants and, by using a “hardware delay” function (Pont, 2001), the execution time of each task was controlled. For example, to implement the tasks in STC C, the WCETs for Tasks A, B and C were entered to the scheduler using the code in Listing 7-3:

```
// Define WCET of each task in microsecond
#define Task_A_WCET 1000
#define Task_B_WCET 12000
#define Task_C_WCET 1000
```

Listing 7-3: Definition of task WCETs in STC C in the TTC-ISR scheduler.

The duration of (for example) Task A was adjusted using a hardware delay as shown in Listing 7-4.

```
void Task_A(void)
   {
   …
   // Delay to control the duration of the task based on hardware timer 1
   Hardware_Delay_T1(Task_A_WCET);
   …
   }
```

Listing 7-4: Adjusting the duration of Task A in STC C in the TTC-ISR scheduler.

Where `Hardware_Delay_T1()` is a function implemented particularly to generate N microsecond delay (approximately) based on hardware Timer 1 (see Listing 7-5).

```
void Hardware_Delay_T1(const unsigned int DELAY)
   {

   // Start timer 1
   T1TCR &= 0x00;
   T1TCR |= 0x01;

   // Set the match register to current time plus required delay
   T1MR0 = T1TC + ((PCLK / 1000000U) * DELAY) ;

   // On match, nothing occurs
   T1MCR &= !0x07; // to make sure that no intrrupt, no reset, no stop on match
                   // register 0

   while ((T1TC < T1MR0));

   }
```

Listing 7-5: One way to implement a hardware delay function (see Pont, 2001 for more details).

Remember that, in STC C, the execution times of tasks were fixed. In situations where tasks have variable durations, such as STC A and STC B, code example shown in Listing 7-6 was used to manipulate the execution time of the tasks. For example, in STC A, Task A's duration varies between 0.01 to 0.4 Ticks (i.e. maximum duration is 2 ms).

```
void Task_A(void)
   {
   int i = 0;
   …

   // Delay to control the duration of the task
   // A random data array was generated to produce 5000 integers with a maximum
   // duration of 2000 µs
   Hardware_Delay_T1( (Random_Data_G[i] % 2000)); //

   // increment i up to 5000 then repreat from 0
   i = (i+1) % 5000;

   }
```

Listing 7-6: Varying the duration of Task A in STC A in the TTC-ISR scheduler.

7.3.1.2 Task sequencing and overrun behaviour

The sequence behaviour of the TTC-ISR scheduler when applying STC A, STC B, STC C and STC D is summarised in Table 7-1.

Table 7-1: Task schedule in TTC-ISR scheduler.

STC	Scheduler behaviour
A	A1
B	B1
C	C1
D	D1a

The results in Table 7-1 show that – as expected – the TTC-ISR scheduler performs the standard scheduler tests (STC A, STC B and STC C) without problems.

The results in the table also illustrate that – in the event of task overrun – the scheduler cannot recover. Referring back to Listing 7-2, it can be seen that – during the overrun – the TTC-ISR scheduler will lose count of the missing ticks (see Figure 7-7). After the overrun completes, the schedule will continue but will always be delayed by 10 ticks.

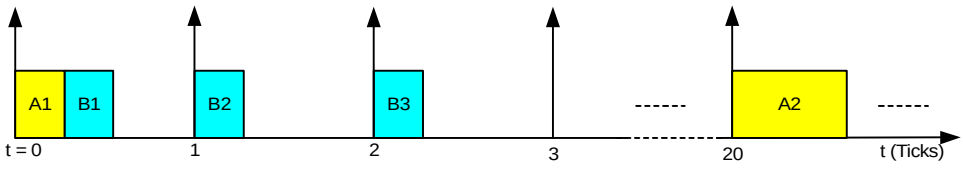


Figure 7-7: The behaviour of TTC-ISR scheduler with STC D (D1a schedule class).

7.3.1.3 Jitter

Table 7-2 shows the periods and jitter measurements for the tick and the tasks for STC A, STC B and STC C when implemented using the TTC-ISR scheduler.

Table 7-2: Task jitter from the TTC-ISR scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	3029.1	2481.1
	Max Period	4999.7	4999.7	6966.1	7595.1
	Average Period	4999.7	4999.7	4938.2	4949.3
	Diff. Jitter	0.0	0.0	3937.0	5114.0
	Avg. Jitter	0.0	0.0	819.6	912.5
Test B	Min Period	4999.7	9999.4	2992.5	2146.1
	Max Period	4999.7	9999.5	7009.2	7761.9
	Average Period	4999.7	9999.5	4845.3	4498.0
	Diff. Jitter	0.0	0.1	4016.7	5615.8
	Avg. Jitter	0.0	0.0	1167.7	1156.4
Test C	Min Period	4999.7	2994.2	19998.9	2994.1
	Max Period	14999.2	17004.7	19998.9	17004.8
	Average Period	7953.6	5193.9	19998.9	4908.2
	Diff. Jitter	9999.5	14010.5	0.0	14010.7
	Avg. Jitter	4562.5	5097.5	0.0	4812.0

The jitter values in STC A and STC B show that with the TTC-ISR scheduler, the tick interrupts occur at precisely-defined intervals with no measurable delays or jitter. The release jitter in Task A is also equal to zero, while low-priority tasks (Task B and Task C) always suffer high jitter in their release times caused by variations in the execution times of the preceding tasks.

In situations where a task required multiple ticks to execute (as with STC C), the resulting tick jitter was significantly increased and the system timing no longer matched the specification. Note that the tick interval in STC C (for example) is not fixed to 5 ms as required but instead varies between 5 and 15 ms resulting in only two (rather than four) ticks per major cycle.

7.3.1.4 CPU, memory and power requirements

Table 7-3 shows the CPU overhead for the TTC-ISR scheduler (with STC A).

Table 7-3: CPU overhead for the TTC-ISR scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	10.09	25.54	39.5

Table 7-4 summarises the memory required to implement STC A using the TTC-ISR scheduler.

Table 7-4: Memory requirements (ROM and RAM) for the TTC-ISR scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	2256	127

Table 7-5 shows the power consumption levels from the STC A and STC B.

Table 7-5: Power requirements for the TTC-ISR scheduler.

Method	Power consumption (mW)
Test A	39.7
Test B	36.4

7.3.2 Applying STC to the TTC-Dispatch scheduler

This section discusses the implementation of STCs in the TTC-Dispatch scheduler and presents the output results from such an implementation.

7.3.2.1 Implementing the test cases

Implementing the STCs was straightforward using `SCH_Add_Task()`function. As an example, STC C was implemented as follows:

```
// Add tasks (5 ms ticks)
 // Parameters are <task name>, <offset in ticks>, <period in ticks>
 SCH_Add_Task(TaskA, 1, 2);
 SCH_Add_Task(TaskB, 0, 4);
 SCH_Add_Task(TaskC, 1, 2);
```

Listing 7-7: Implementing STC C using the TTC-Dispatch scheduler: task's offset and period.

It can be noted that the implementation of STC C was rather more straightforward than was the case with the corresponding TTC-ISR scheduler implementation (see Listing 7-1). Similarly, the WCETs of tasks were entered to the system using a `SCH_Task_WCET` function as in Listing 7-8.

```
// Input duration for tasks
// Values are in *microseconds*
SCH_Task_WCET(Task_A, 1000);
SCH_Task_WCET(Task_B, 12000);
SCH_Task_WCET(Task_C, 1000);
```

Listing 7-8: Implementing STC C using the TTC-Dispatch scheduler: task's WCET.

Also in the TTC-Dispatch scheduler, hardware delays were used, in the same way as in the TTC-ISR scheduler, to adjust the tasks WCETs.

7.3.2.2 Task sequencing and overrun behaviour

The sequence behaviour of the TTC-Dispatch scheduler when applying STC A, STC B, STC C and STC D is summarised in Table 7-6.

Table 7-6: Task schedule in TTC-Dispatch scheduler.

STC	Scheduler behaviour
A	A1
B	B1
C	C1
D	D1b

When executing STC A, STC B and STC C, the TTC-Dispatch scheduler behaves in the same way as the TTC-ISR scheduler. However, when executing STC D, the Dispatch scheduler keeps track of the number of elapsed ticks during the overrun, and – once the overrunning task (Task A) completes – the scheduler performs all missing executions for Task B (in this case, 10 executions), before continuing to serve the tasks in the following ticks. This means that the scheduler has the potential to "catch up" in the event of certain (infrequent and temporary) errors: see Figure 7-8.

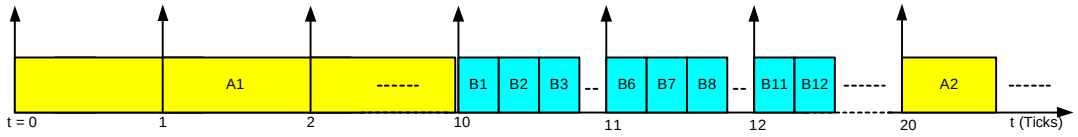


Figure 7-8: The behaviour of Dispatch scheduler with STC D (D1b schedule class).

7.3.2.3 Jitter

Table 7-7 shows the periods and jitter measurements for the tick and the tasks in STC A, STC B and STC C implemented using the TTC-Dispatch scheduler.

Table 7-7: Task jitter from the TTC-Dispatch scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	3029.1	2480.5
	Max Period	4999.7	4999.7	6966.1	7595.1
	Average Period	4999.7	4999.7	4950.9	4917.0
	Diff. Jitter	0.0	0.0	3937.0	5114.6
	Avg. Jitter	0.0	0.0	823.9	921.0
Test B	Min Period	4999.7	9999.4	2988.4	2164.3
	Max Period	4999.7	9999.5	7011.1	7864.1
	Average Period	4999.7	9999.5	4882.0	4799.3
	Diff. Jitter	0.0	0.1	4022.7	5699.8
	Avg. Jitter	0.0	0.0	1172.7	1226.9
Test C	Min Period	4999.4	2978.7	19998.9	2978.6
	Max Period	4999.9	17020.2	19998.9	17020.3
	Average Period	4999.7	5326.5	19998.9	5155.3
	Diff. Jitter	0.5	14041.5	0.0	14041.7
	Avg. Jitter	0.2	5240.1	0.0	5082.2

The jitter values presented in the table show that with the TTC-Dispatch implementation, the duration of the tick interval – in all cases – is constant and equal to 5 ms. However, the tick suffers small jitter when STC C is employed. This jitter is mainly caused by the variation in time taken to leave Task B – rather than leaving the idle mode as in the normal situations – and run the ISR Update function. Note that in this implementation, when the interrupt occurs while Task B is running, the Update function is executed then the scheduler returns back to continue the execution of Task B.

The table also shows that Task A has consistently low (release) jitter levels while the jitter for Task B and Task C is rather high in STC A and STC B.

7.3.2.4 CPU, memory and power requirements

Table 7-8 shows the CPU overhead for the TTC-Dispatch scheduler (with STC A).

Table 7-8: CPU overhead for the TTC-Dispatch scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	9.93	25.01	39.7

The CPU overheads results show that the overall processing time is very similar to that observed with the TTC-ISR scheduler.

Table 7-9 presents the memory required to implement STC A using the TTC-Dispatch scheduler. Inevitably, these figures are somewhat larger than those required to implement the TTC-ISR scheduler.

Table 7-9: Memory requirements (ROM and RAM) for the TTC-Dispatch scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	4012	325

Table 7-10 shows the power consumption levels for STC A and STC B when implemented using the TTC-Dispatch scheduler.

Table 7-10: Power requirements for the TTC-Dispatch scheduler.

Method	Power consumption (mW)
Test A	39.3
Test B	35.7

7.3.3 Applying STC to the TTC-DVS scheduler

This section discusses the implementation of STCs in the TTC-DVS scheduler and presents the output results from such an implementation.

7.3.3.1 Implementing the test cases

Since the TTC-DVS scheduler is adapted from the TTC-Dispatch scheduler, the code examples shown in Listing 7-7 and Listing 7-8 were used to implement the STCs in TTC-DVS.

It is important to note that the results obtained here were based on an LPC2106 board with a 10 MHz crystal oscillator frequency (the other examples in this chapter used a 12 MHz crystal: see Section 7.2.1). The 10 MHz oscillator was used in this case to simplify the process of implementing DVS (see Phatrapornnant and Pont, 2006).

7.3.3.2 Task sequencing and overrun

The sequence behaviour of the TTC-DVS scheduler when applying STC A, STC B, STC C and STC D is summarised in Table 7-11.

Table 7-11: Task schedule in TTC-DVS scheduler.

STC	Scheduler behaviour
A	A1
B	B1
C	C1
D	D1b

Since the DVS scheduler implementation used is based upon the TTC-Dispatch scheduler, the task behaviour observed is identical to that shown in Table 7-6.

7.3.3.3 Jitter

Table 7-12 shows the periods and jitter measurements for the tick and the tasks in STC A, STC B and STC C implemented on TTC-DVS scheduler.

Table 7-12: Task jitter from the TTC-DVS scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.8	4999.8	3029.3	2416.7
	Max Period	4999.8	4999.9	6966.3	7595.2
	Average Period	4999.8	4999.8	4926.3	4937.6
	Diff. Jitter	0.0	0.1	3937.0	5178.5
	Avg. Jitter	0.0	0.0	821.3	913.9
Test B	Min Period	4999.8	9999.6	2904.6	2011.9
	Max Period	4999.8	9999.7	7097.5	7951.0
	Average Period	4999.8	9999.7	4701.0	4718.3
	Diff. Jitter	0.0	0.1	4192.9	5939.1
	Avg. Jitter	0.0	0.0	1167.7	1268.9
Test C	Min Period	4999.5	3457.5	19999.3	3457.3
	Max Period	5000.1	16541.9	19999.4	16542.0
	Average Period	4999.8	6241.9	19999.3	5006.5
	Diff. Jitter	0.6	13084.4	0.1	13084.7
	Avg. Jitter	0.2	5355.8	0.0	4227.8

Despite the use of DVS, the jitter values shown in the table are similar to those presented in Table 7-7. Remember that the results obtained here were based on an LPC2106 board running at 10 MHz oscillator frequency. This explains the little differences in some values between the results obtained from the TTC-DVS and those obtained from the TTC-Dispatch schedulers. For example, the difference Tick jitter in STC C is equal to 0.6 µs at the used frequency. Since the jitter is inversely proportional to the operating frequency, such a value would be equal to 0.5 µs if 12 MHz oscillator frequency is used (as with the other implementations). Please compare to the equivalent Tick jitter value in Table 7-7.

7.3.3.4 CPU, memory and power requirements

Table 7-13 shows the CPU overhead for the TTC-DVS scheduler (with STC A). Inevitably, the CPU overhead for the TTC-DVS scheduler is greater than that for the TTC-ISR and TTC-Dispatch schedulers.

Table 7-13: CPU overhead for the TTC-DVS scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	10.16	25.03	40.6

Table 7-14 shows the memory required to implement the TTC-DVS scheduler: again, this is greater than for previous implementations.

Table 7-14: Memory requirements (ROM and RAM) for the TTC-DVS scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	17460	767

Overall, the memory results can demonstrate the complexity of implementing a TTC-DVS scheduler in the used embedded hardware platform. Table 7-15 shows the power consumption levels from the STC A and STC B when implemented using the TTC-DVS scheduler.

Table 7-15: Power requirements for the TTC-DVS scheduler.

Method	Power consumption (mW)
Test A	24.8
Test B	16.6

The results show that the power consumption levels in the TTC-DVS scheduler are low compared to the previous TTC implementations considered in this chapter. Compared to the TTC-Dispatch scheduler, the values in the table show that in STC A, the average power was reduced by the factor of 37%, where in STC B it was reduced by 53%. This reduction may be significant in a wide range of mobile embedded applications.

7.3.4 Applying STC to the TTC-TG scheduler

This section discusses the implementation of STCs in the TTC-TG scheduler and presents the output results from such an implementation.

7.3.4.1 Implementing the test cases

Since the TTC-TG scheduler is adapted from the TTC-Dispatch scheduler, the code example shown in Listing 7-7 and Listing 7-8 were used to implement the STCs in TTC-TG.

7.3.4.2 Task sequencing and overrun behaviour

The sequence behaviour of the TTC-TG scheduler when applying STC A, STC B, STC C and STC D is summarised in Table 7-16.

Table 7-16: Task schedule in TTC-TG scheduler.

STC	Scheduler behaviour
A	A1
B	B1
C	C6
D	D2b

The results illustrate that in STC C, Task B is terminated when the next tick interrupt takes place. This is because the TG scheduler is designed to support tasks with a WCET of at most one Tick.

In STC D, it is clear that the scheduler detects and hence terminates the overrunning task (Task A) at the beginning of the tick following the one in which Task A overruns. Moreover, the scheduler allows running a backup task BK(A) to replace Task A in the same tick in which the overrun is detected and hence continues to run the following tasks (Figure 7-9 (a)). This means that one tick shift is added to the schedule.

However, in some cases where (for example) the schedule is heavily loaded with tasks, the insertion of a backup task in the next tick of overrun may cause a domino effect. To reduce the impact of such a problem, the whole schedule can be extended for one tick to allow the backup task to complete before the scheduler goes back to its normal operation. In the case of STC D, the whole schedule will be extended for two ticks: one for the backup task and one to run the missed task B1 (Figure 7-9 (b)).

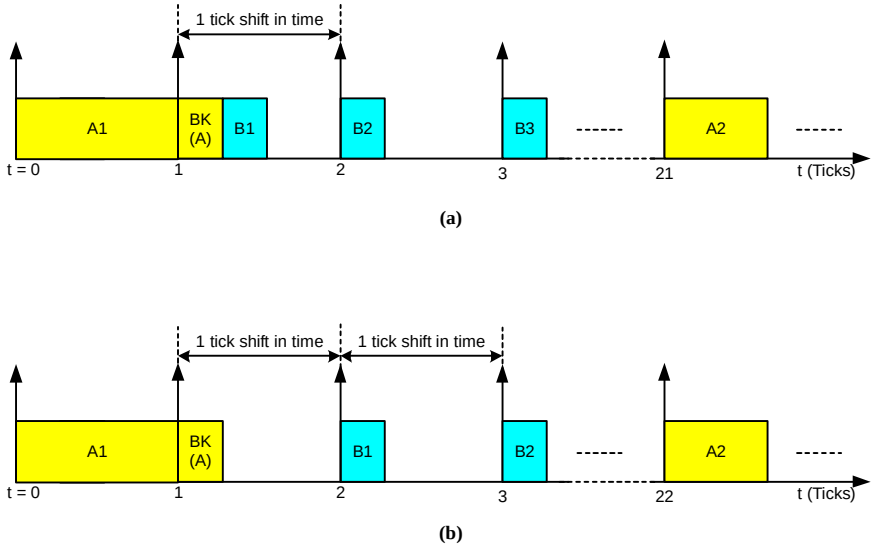


Figure 7-9: The behaviour of TG scheduler with STC D (D2b schedule class).

7.3.4.3 Jitter

Table 7-17 shows the periods and jitter measurements for the tick and the tasks in STC A, STC B and STC C implemented on TTC-TG scheduler.

Table 7-17: Task jitter from the TTC-TG scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	3029.1	2415.6
	Max Period	4999.7	4999.7	6966.1	7541.6
	Average Period	4999.7	4999.7	4933.3	4905.7
	Diff. Jitter	0.0	0.0	3937.0	5126.0
	Avg. Jitter	0.0	0.0	822.0	922.7
Test B	Min Period	4999.7	9999.4	2985.5	2096.2
	Max Period	4999.7	9999.5	7011.7	7848.1
	Average Period	4999.7	9999.5	4922.7	4595.6
	Diff. Jitter	0.0	0.1	4026.2	5751.9
	Avg. Jitter	0.0	0.0	1175.3	1203.3
Test C	Min Period	4999.6	9990.2	19998.9	9990.1
	Max Period	4999.9	10008.7	19998.9	10008.9
	Average Period	4999.7	9999.6	19998.9	9999.3
	Diff. Jitter	0.3	18.5	0.0	18.8
	Avg. Jitter	0.1	9.2	0.0	9.4

Comparing these results with those obtained from the TTC-Dispatch scheduler – upon which this scheduler implementation was based – it can be seen that most are similar. A key difference in the results was the STC C jitter values. The table shows that although Tasks A and Task C no longer suffer very high release jitter (as in the TTC-Dispatch scheduler), they still have variation in their release times. The 19 µs variations observed here were caused by the modified Update function which, in this implementation, differs in length when a task exceeds the tick interval (Hughes and Pont, 2004; Hughes and Pont, in press).

7.3.4.4 CPU, memory and power requirements

Table 7-18 shows the CPU overhead for the TTC-TG scheduler (with STC A).

Table 7-18: CPU overhead for the TTC-TG scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	9.95	25.03	39.8

The CPU overheads results show that the overall processing time required in the implemented TTC-TG scheduler is similar to that of the TTC-Dispatch scheduler.

Table 7-19 presents the memory requirements for implementing the STC A for the TTC-TG scheduler. Compared with the memory requirements in Dispatch schedulers, these figures are slightly larger.

Table 7-19: Memory requirements (ROM and RAM) for the TTC-TG scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	4296	446

Table 7-20 shows the power consumption levels from the STC A and STC B when implemented using the TTC-TG scheduler.

Table 7-20: Power requirements for the TTC-TG scheduler.

Method	Power consumption (mW)
Test A	38.9
Test B	35.7

Since the TG scheduler is based on the Dispatch approach (the TTC-Dispatch scheduler), the same levels of CPU power consumption were observed with the TTC-TG scheduler.

7.3.5 Applying STC to the TTC-MTI scheduler

This section discusses the implementation of STCs in the TTC-MTI scheduler and presents the output results from such an implementation.

7.3.5.1 Implementing the test cases

In MTI scheduler, task parameters are defined in the same way as with Dispatch scheduler. Therefore, the code shown in Listing 7-7 and Listing 7-8 were also used here to implement the STCs.

7.3.5.2 Task sequencing and overrun behaviour

The sequence behaviour of the TTC-MTI scheduler when applying STC A, STC B, STC C and STC D is summarised in Table 7-21.

Table 7-21: Task schedule in TTC-MTI scheduler.

STC	Scheduler behaviour
A	A2
B	B2
C	C6
D	D3a

With this scheduler implementation, it can be seen that the gap insertion mechanism employed (through the multiple timer interrupts) has helped to achieve low jitter at the release time of all tasks running in the system (both in STC A and STC B). However, it should be noted that, like the TG scheduler, the TTC-MTI scheduler cannot support tasks which require multiple ticks to execute (as in STC C). This is because the

scheduler employs a simple TG mechanism and – once an interrupt occurs – the running task (if any) will be terminated. Note that the implementation employed here did not support backup tasks (Figure 7-10).

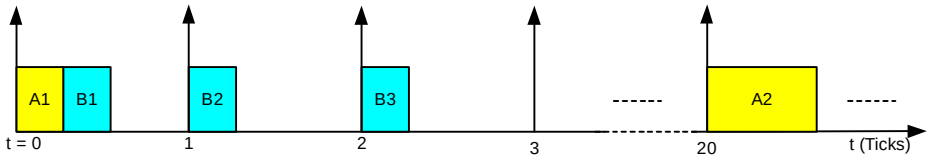


Figure 7-10: The behaviour of MTI scheduler with STC D (D3a schedule class).

7.3.5.3 Jitter

Table 7-22 shows the periods and jitter measurements for the tick and the tasks in STC A, STC B and STC C implemented using the TTC-MTI scheduler.

Table 7-22: Task jitter from the TTC-MTI scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	4999.7	4999.7
	Max Period	4999.7	4999.7	4999.7	4999.7
	Average Period	4999.7	4999.7	4999.7	4999.7
	Diff. Jitter	0.0	0.0	0.0	0.0
	Avg. Jitter	0.0	0.0	0.0	0.0
Test B	Min Period	4999.7	9999.4	4999.7	4999.7
	Max Period	4999.7	9999.5	4999.7	4999.7
	Average Period	4999.7	9999.5	4999.7	4999.7
	Diff. Jitter	0.0	0.1	0.0	0.0
	Avg. Jitter	0.0	0.0	0.0	0.0
Test C	Min Period	4999.6	9999.4	19998.9	9999.4
	Max Period	4999.9	9999.5	19998.9	9999.5
	Average Period	4999.7	9999.5	19998.9	9999.5
	Diff. Jitter	0.3	0.1	0.0	0.1
	Avg. Jitter	0.1	0.0	0.0	0.0

The jitter values in the table clearly show how the TTC-MTI scheduler helped to remove jitter in the release time of all tasks running in the system, causing a significant increase in the overall system predictability. Note that, in STC C, the tick jitter was

caused by the difference between the time taken to leave Task B and service the interrupt (Tick 1 in the cycle) and the time taken to leave the idle mode and service the interrupt (Tick 3).

7.3.5.4 CPU, memory and power requirements

Table 7-23 shows the CPU overhead for the TTC-MTI scheduler (with STC A).

Table 7-23: CPU overhead for the TTC-MTI scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	9.9	25.01	39.6

The CPU overhead results show that the overall processing time required for the TTC-MTI scheduler is similar to that required for the other schedulers.

Table 7-24 presents the memory requirements for implementing the STC A for the TTC-MTI scheduler. These ROM figures are slightly smaller than those used to implement any of the previous Dispatch schedulers while the RAM figures are larger (remember that the overall architecture is rather different in TTC-MTI: see Section 5.7).

Table 7-24: Memory requirements (ROM and RAM) for the TTC-MTI scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	3620	514

Table 7-25 shows the power consumption levels from the STC A and STC B when implemented using the TTC-MTI scheduler.

Table 7-25: Power requirements for the TTC-MTI scheduler.

Method	Power consumption (mW)
Test A	40.3
Test B	36.3

7.3.6 Applying STC to the TTC-Adaptive implementation

This section discusses the implementation of STCs in the TTC-Adaptive scheduler and presents the output results from such an implementation.

7.3.6.1 Implementing the test cases

Since the structure of the TTC-Adaptive scheduler is adapted from the TTC-MTI scheduler, where the tasks are defined based on the approaches used in TTC-Dispatch, the code example shown in Listing 7-7 and Listing 7-8 were also used here to implement the STCs.

7.3.6.2 Task sequencing and overrun behaviour

The sequence behaviour of the TTC-Adaptive scheduler when applying STC A, STC B, STC C and STC D is summarised in Table 7-26.

Table 7-26: Task schedule in TTC-Adaptive scheduler.

STC	Scheduler behaviour
A	A2
B	B2
C	C6
D	D3b

Since TTC-Adaptive scheduler is based on the TTC-MTI and TTC-TG, it lacks the support for running long tasks which require multiple ticks to execute (as in STC C).

As expected, this scheduler implementation provides low jitter at the release time of all tasks running in the system and provides an efficient solution for task overruns problem. For example, unlike TTC-MTI, such an implementation provides a support for backup task that will replace the overrunning task once shut down. In this scheduler, there can be three different options:

- If it is not dependent on the output from Task A, Task B1 can still be scheduled to run in the same tick as Task A1 (Figure 7-11(a)).
- If it is dependent on the output from Task A, Task B1 must be scheduled to run in the next tick after task BK(A) completes execution (Figure 7-11(b)). This will obviously add one tick shift to the schedule.

- To avoid any possibility for a domino effect to take place, the whole schedule can be extended for one more tick to allow a completion of BK(A) before returning to the normal schedule (Figure 7-11(c)). The figure shows that, in the case of STC D, two tick shifts will be added to the whole schedule.

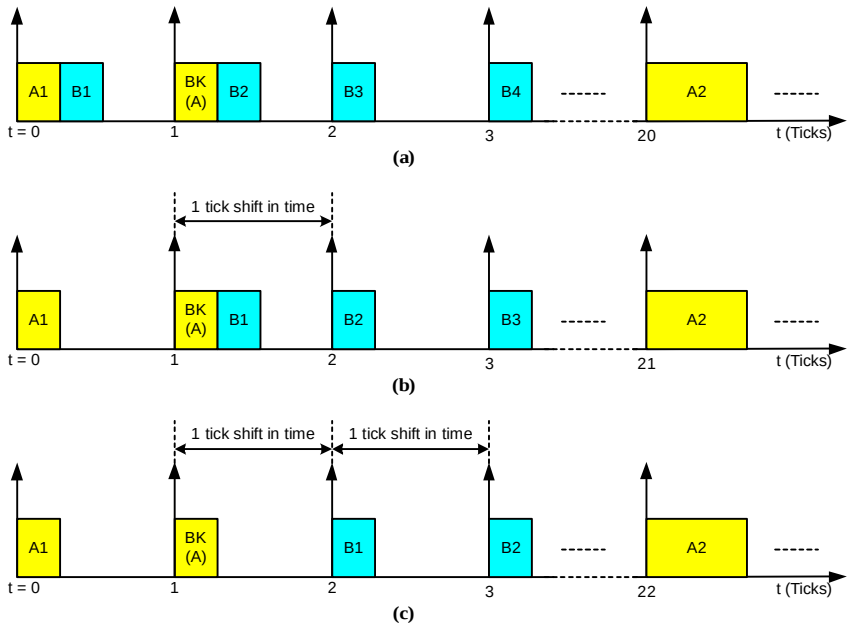


Figure 7-11: The behaviour of MTI scheduler with STC D (D3b schedule class).

Note that the TTC-Adaptive implementation presented in Chapter 5 considered the second option (Figure 7-11(b)). However, the scheduler framework developed in this study has been made flexible so that the user can – with a little modification – adopt any of the three proposed solutions.

Remember that, in addition to low-jitter provision and overrun prevention, the most advantageous feature of the TTC-Adaptive scheduler is its ability to control the timing behaviour of tasks based on accurate "online" measurements (not estimations) of their WCETs.

7.3.6.3 Jitter

Table 7-27 shows the periods and jitter measurements for the tick and the tasks in STC A, STC B and STC C implemented using the TTC-Adaptive scheduler.

Table 7-27: Task jitter from the TTC-Adaptive scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	4999.7	4999.7
	Max Period	4999.7	4999.7	4999.7	4999.7
	Average Period	4999.7	4999.7	4999.7	4999.7
	Diff. Jitter	0.0	0.0	0.0	0.0
	Avg. Jitter	0.0	0.0	0.0	0.0
Test B	Min Period	4999.7	9999.4	4999.7	4999.7
	Max Period	4999.7	9999.5	4999.7	4999.7
	Average Period	4999.7	9999.5	4999.7	4999.7
	Diff. Jitter	0.0	0.1	0.0	0.0
	Avg. Jitter	0.0	0.0	0.0	0.0
Test C	Min Period	4999.6	9999.4	19998.9	9999.4
	Max Period	4999.9	9999.5	19998.9	9999.5
	Average Period	4999.7	9999.5	19998.9	9999.5
	Diff. Jitter	0.3	0.1	0.0	0.1
	Avg. Jitter	0.1	0.0	0.0	0.0

The jitter values in the table also show how the TTC-Adaptive scheduler can remove jitter in the release time of all tasks running in the system. Overall, the jitter behaviour here is seen similar to that obtained with the TTC-MTI scheduler.

7.3.6.4 CPU, memory and power requirements

Table 7-28 shows the CPU overhead for the TTC-Adaptive scheduler (with STC A).

Table 7-28: CPU overhead for the TTC-Adaptive scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	9.95	25.01	39.8

The CPU overhead results show that the implementation of TTC-Adaptive scheduler requires no additional processing time as compared to previous schedulers.

Table 7-29 presents the memory requirements for implementing the STC A for the TTC-Adaptive scheduler. The figures in the table show insignificant increase in the

code memory overhead when compared to those required for TTC-MTI scheduler. The data memory is however as low as that required for the TTC-MTI scheduler.

Table 7-29: Memory requirements (ROM and RAM) for the TTC-MTI scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	5364	510

Table 7-30 shows the power consumption levels from the STC A and STC B when implemented using the TTC-Adaptive scheduler.

Table 7-30: Power requirements for the TTC-Adaptive scheduler.

Method	Power consumption (mW)
Test A	40.5
Test B	36.5

7.4 Summary of the results

This section summarises the results detailed in the previous sections. The first four columns in the summary table (Table 7-31) report the sequence behaviour from the STCs in all TTC implementations. The remaining columns include CPU, jitter, memory and power requirements. Since it is difficult to list all jitter results, the jitter columns only present the Difference Tick jitter levels from STC C, and the Difference release jitter levels for Task A and Task B from STC B as representative jitter values across all jitter measurements.

Table 7-31: Summary of results obtained in this chapter.

Scheduler name	STC A	STC B	STC C	STC D	CPU %	Tick Jitter (µs)	Task A Jitter (µs)	Task B Jitter (µs)	ROM (Bytes)	RAM (Bytes)	Power (mW)
TTC-ISR	A1	B1	C1	D1a	39.5	9999.5	0.1	4016.7	2256	127	36.4
TTC Dispatch	A1	B1	C1	D1b	39.7	0.5	0.1	4022.7	4012	325	35.7
TTC-DVS	A1	B1	C1	D1b	40.6	0.5	0.1	4192.9	17460	767	16.6
TTC-TG	A1	B1	C6	D2b	39.8	0.3	0.1	4026.2	4296	446	35.7
TTC-MTI	A2	B2	C6	D3a	39.6	0.3	0.1	0.0	3620	514	36.3
TTC-Adaptive	A2	B2	C6	D3b	39.8	0.3	0.1	0.0	5364	510	36.5

Jitter in Task A has been included in the table to allow a comparison with the jitter levels in low-priority tasks. Key jitter results are shown in Figure 7-12 for comparison purposes. It can be clearly noticed that with the new TTC implementations, namely TTC-MTI and TTC-Adaptive schedulers, release jitter in all tasks running in the system is minimised.

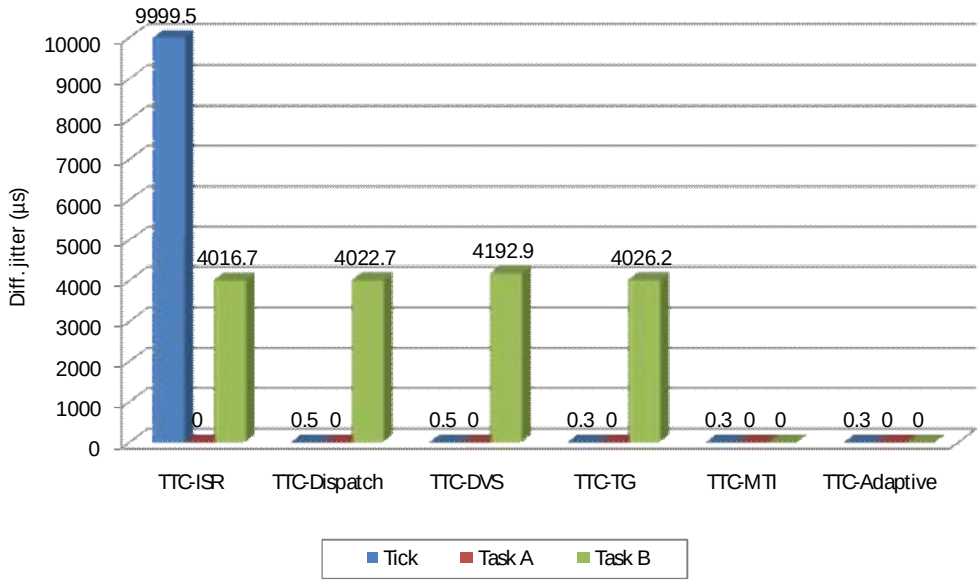


Figure 7-12: Summary of key jitter results in all TTC implementations.

The results for CPU, memory and power requirements are shown in Figure 7-13 to Figure 7-16 for comparison purposes.

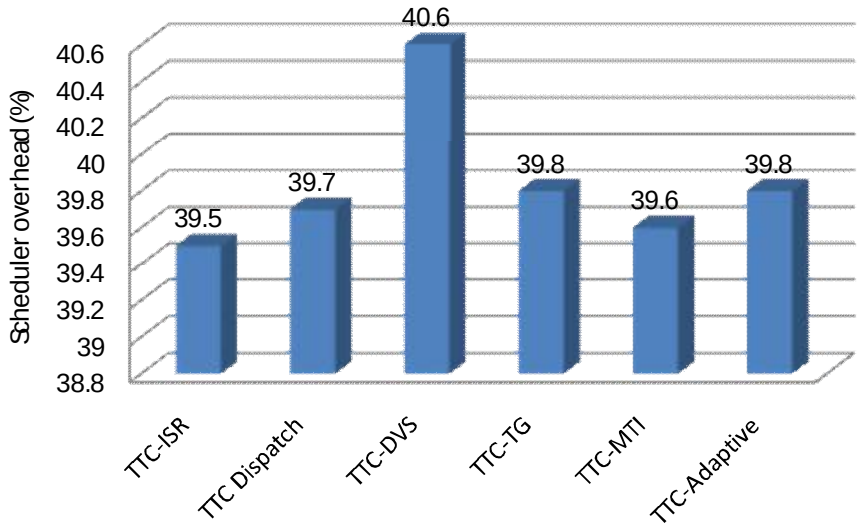


Figure 7-13: Summary of CPU requirements in all TTC implementations.

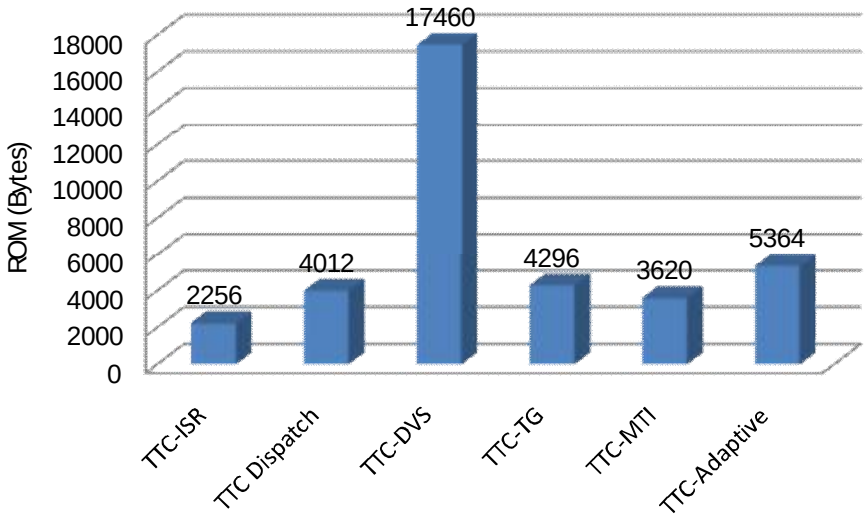


Figure 7-14: Summary of ROM requirements in all TTC implementations.

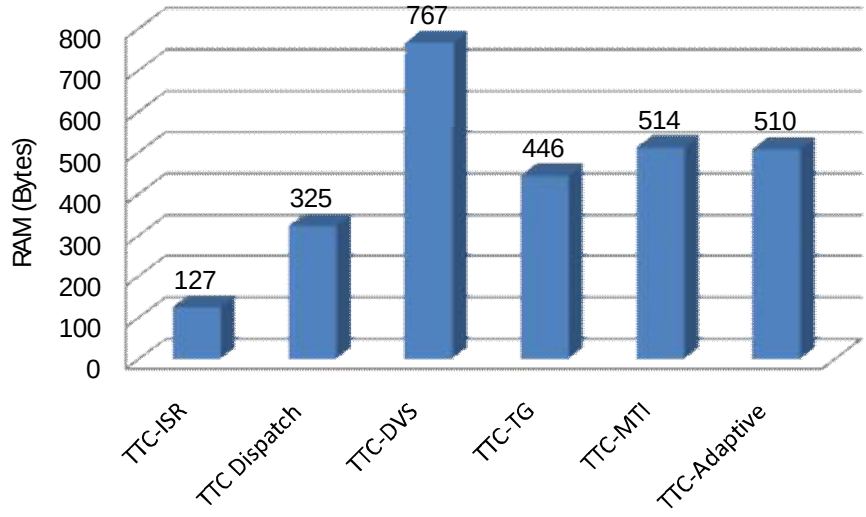


Figure 7-15: Summary of RAM requirements in all TTC implementations.

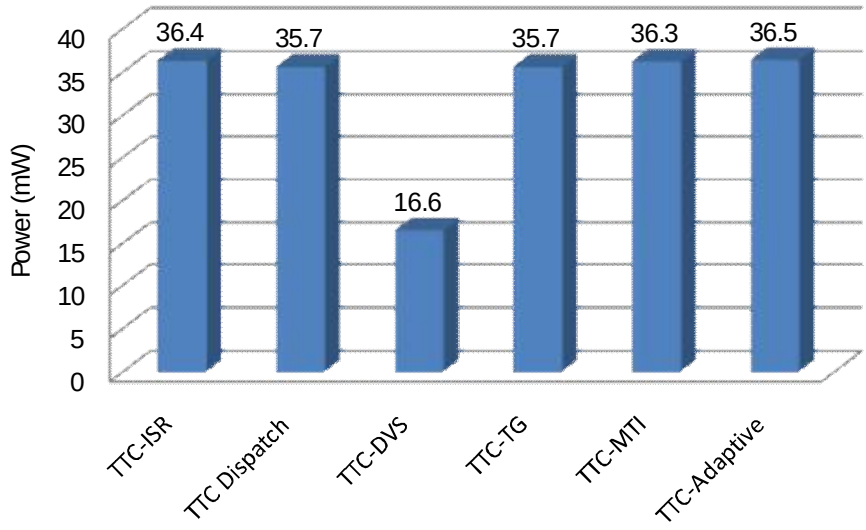


Figure 7-16: Summary of power requirements in all TTC implementations.

It can be clearly seen that the CPU utilisation in all assessed TTC implementations was almost the same. The reason why CPU overheads have been included in the table is to show that, despite the improvement that TTC-MTI and TTC-Adaptive schedulers can offer to the system, such implementations do not compromise the resource efficiency as a price for achieving such an improvement. Instead, it can be seen that the code memory required to implement (for example) the TTC-MTI schedulers was even smaller than was used for the majority of other schedulers. In the TTC-Adaptive scheduler, the little

increase in the code memory as compared to other schedulers is outweighed by the improvement it provides to the scheduler behaviour.

7.5 Conclusions

This chapter began by providing an overview of the experimental methodology used to obtain the empirical results from single-processor TTC implementations considered in this thesis. It discussed in detail how each parameter used to assess the behaviour of the schedulers was measured using practical methods.

The chapter then provided the output results from the application of the STC technique, detailed in Chapter 6, to the range of TTC scheduler implementations described in Chapter 5.

The results presented in this chapter clearly show that even a small (and by no means exhaustive) selection of TTC scheduler implementations demonstrated a wide range of different patterns of behaviour. The results also suggested that a “one size fits all” TTC implementation does not exist in practice, since each implementation has advantages and disadvantages. The selection of a particular implementation will, hence, be decided based on the requirements of the application in which the TTC scheduler is employed, e.g. timing and resource requirements.

Note that, in this chapter, the STC technique was shown to be effective in assessing the behaviour of a simple scheduling algorithm employed in a single-processor system. It would only make sense to adopt such a technique if its applicability on wider (more complicated) architectures can be proven. The next part of the thesis begins to look at ways in which the STC method can be used in assessing the behaviour of scheduling algorithms in multi-processor embedded systems.

PART D:

MULTI-PROCESSOR SYSTEMS

Chapter 8

Network and scheduling protocols for multi-processor embedded systems

8.1 Introduction

Previous chapters in this thesis considered the use of the STC technique in assessing the behaviour of simple embedded system implementations based on single-processor architectures and TTC scheduling algorithms. This part of the thesis begins to investigate the applicability of such a technique when more complicated embedded implementations are considered: for example, when the system is based on distributed architectures.

This chapter reviews a collection of network protocols that are widely used in the design and implementation of distributed real-time embedded systems with high reliability requirements. The focus of the discussion will, however, be on systems using Controller Area Network (CAN) protocol (Bosch, 1991). The chapter provides a brief overview of CAN and compares its features to alternative commercial network protocols. The chapter then describes "high-level" scheduling protocols that can be implemented on the CAN hardware to achieve time-triggered system operations for high predictability. A particular focus of this discussion will be on the Shared-Clock (S-C) scheduling protocol (Pont, 2001) as a simple and effective software platform for many low-cost, reliable embedded systems.

8.2 Overview of multi-processor embedded systems

With the rapid growth of technology, the development of huge and complex embedded systems – that are physically distributed over wide areas – has become quite common. Leen *et al.* (1999) provided two familiar examples:

- A typical passenger car might contain more than 40 processor devices that control brakes, door windows and mirrors, steering, air bags, wheels and so forth.

- An industrial fire detection system might have up to 200 or more processors associated with different sensors and actuators.

Having several microcontrollers communicating with one another in a system is referred to as a ***distributed*** or a ***multi-processor*** system[21]. Ayavoo (2006) highlighted that the number of embedded processors used in a single automotive system (e.g. passenger vehicle) had been steadily increasing over the past few years and predicted that this growth would continue over the next few years as the complexity and functionality of the system increase. A distributed solution helps to reduce the complexity and increase the reliability of the entire system where one transmission medium is shared by all processors. Advantages of using distributed systems are discussed by Tanenbaum (1995).

Historically, multi-processor systems were first developed in the early 1970s, when Moore's law[22] did not work any longer, and people believed that the use of single-processors cannot provide the level of performance that future applications would demand (Ravikumar, 2004).

In order to connect various processors in an embedded control system, an effective network architecture and communication medium are required. Perfect implementations of multi-processor systems are not always justified, not least because of the very wide range of implementation options which are available. For example, the software engineer working on the design of a modern passenger car may need to choose between the use of one (or more) network protocols based on Controller Area Network "CAN"

[21] Please note that the term "multi-processor" is also used to describe System-on-Chip (SoC) designs with multiple CPU cores (i.e. MP SoC). Such designs are not considered in this study. Thus, the term multi-processor used in the context of this thesis is only referred to distributed systems where a umber of physically-distributed microprocessors are connected via a communication network.

[22] Moore's law – which refers back to Gordon E. Moore in 1965 – says: "the complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer" (Moore, 1965).

(Bosch, 1991), Time-Triggered CAN "TTCAN" (Fuhrer *et al.*, 2000), Local Interconnect Network "LIN" (Specks and Rajnak, 2000), FlexRay (FlexRay, 2004) or Time-Triggered Protocol - Class C "TTP/C" (Kopetz, 2001). The resulting network may be connected in (for example) a bus or star (Tanenbaum, 1995) topology. The individual processor nodes in the network may use event-triggered (Nissanke, 1997) or time-triggered (Kopetz, 1997) software architectures – or some combination of the two. The clocks associated with these processors may be linked using (for example) shared-clock techniques (Pont, 2001) or synchronisation messages (Fuhrer *et al.*, 2000). These individual processors may (for example) be C16x (Siemens, 1996), ARM (ARM, 2001), MPC555 (Bannatyne, 2004) or 8051 (Pont, 2001).

8.3 Network protocols for multi-processor systems

8.3.1 Introduction

This section begins by providing a detailed overview of the Controller Area Network (CAN) protocol as a genuine, well-designed hardware platform for multi-processor embedded systems. It then outlines a few other network protocols which have also been used (or recommended) in the design and implementation of such systems.

The section concludes by highlighting the main advantages of CAN, over other network protocols, which made it an appropriate solution for a wide range of embedded designs including those considered in this study.

8.3.2 Controller Area Network (CAN) protocol

8.3.2.1 Introduction

Controller Area Network (CAN) is a cost-effective protocol which is widely used in embedded systems (Farsi and Barbosa, 2000; Fredriksson, 1994; Thomesse, 1998; Sevillano *et al.*, 1998). The CAN protocol was introduced by Robert Bosch GmbH in the 1980s (Bosch, 1991). Although originally designed for automotive applications, CAN is now widely used in process control and many other industrial areas (Farsi and Barbosa, 2000; Fredriksson, 1994; Thomesse, 1998; Sevillano *et al.*, 1998; Pazul, 1999; Zuberi and Shin, 1995; Misbahuddin and Al-Holou, 2003; Short and Pont, 2007). As a consequence of its popularity and widespread use, most modern microcontroller

families now include one or more members with on-chip hardware support for this protocol (e.g. Philips, 1996; Siemens, 1997; Infineon, 2000; Philips, 2004).

In many distributed systems, the CAN protocol provides high reliability communications at very low cost (Farsi and Barbosa, 2000; Fredriksson, 1994; Thomesse, 1998; Sevillano *et al.*, 1998). For example, in automotive vehicles, CAN allows a huge reduction in wiring complexity as communicating devices are connected through a single pair of wire (Farsi and Barbosa, 2000).

8.3.2.2 CAN features and operational principles

The main features and operational principles of CAN have been discussed in detail in a number of recognised publications (e.g. Bosch, 1991; Farsi *et al.*, 1999; Farsi and Barbosa, 2000; Kopetz, 2001; CiA, 2008). The main features of CAN protocol can be summarised as follows.

- High-integrity serial data communication bus for real-time applications.
- Communication speed up to 1 Mbps transmission rate (this speed is also referred to as baudrate).
- Low-cost physical medium: simple twisted wire pair is used.
- Short data length: very low latency compared to other protocols.
- Fast reaction times: no token or permission required from a bus arbiter.
- Multi-master and peer-to-peer communication: broadcast to all or subset of nodes.
- Error detection and correction: high level of error detection and confinement.

CAN is usually viewed as an “event-triggered” protocol (Leen and Heffernan, 2002) which has the following operational principles. Any transmitted message is defined by an identifier which is unique throughout the network. This identifier defines the message contents as well as the message priority. CAN follows Carrier Sense Multiple Access / Collision Avoidance (CSMA/CA) protocol. Under such a protocol, when several nodes compete for bus access, the higher priority message is guaranteed to gain the bus access where lower priority messages have to wait until the bus becomes in the idle state (CiA, 2008; Farsi *et al.*, 1999; Bosch, 1991). When a message wins the bus

access, it has an opportunity to complete transmission without destruction. Such a mechanism is known as “non-destructive bitwise arbitration” (NDBA) which helps resolve conflicts in bus access (Leen and Heffernan, 2001). Once a message is received, each node performs an acceptance test to determine if the received message is relevant to that particular node. The ability to send data on an event basis means that the bus load utilisation is kept to the minimal level. One advantage of CAN over other fieldbus solutions is that this mechanism requires no interaction from a bus master or arbiter (see Farsi *et al.*, 1999; Egan-Krieger, 1994 for more information).

8.3.2.3 CAN layers

As shown in Figure 8-1, CAN standard is a two-layer protocol as compared to the seven layers of the ISO/OSI Reference Model (Farsi and Barbosa, 1999; Bosch, 1991; CiA, 2008). These two layers are: Data Link Layer and Physical Layer. The data link layer consists of two sub-layers: Medium Access Control (MAC) and Logical Link Control (LLC). The LLC sub-layer is concerned with the errors detection and correction when data is exchanged in the network. The MAC sub-layer is responsible for message framing, arbitration, acknowledgement, error detection and signalling. For example, the MAC layer decides which node has the bus control for transmission (see Farsi and Barbosa, 2000 for more details).

Since higher layer services are needed for some applications, CAN in Automation (CiA) defined the CAN Reference Model which incorporates the CAN Application Layer (CAL). The CAL layer employs a large number of services and strategies which achieve the communications between applications. For more details, refer to Farsi *et al.* (1999).

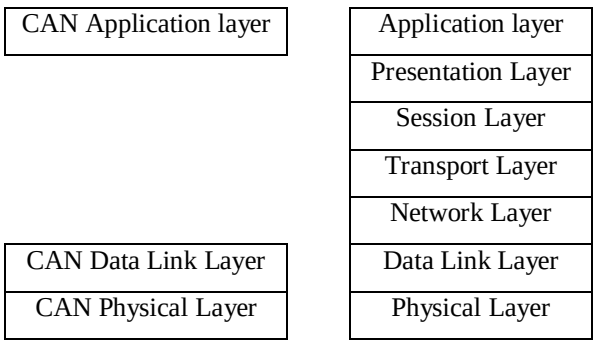


Figure 8-1: Comparison between CAN layers and ISO/OSI Model.

8.3.2.4 CAN format

Briefly, any transmitted CAN frame has the format shown in Figure 8-2. The function of each field illustrated in the figure is as follows (Busch, 1991, Egan-Krieger, 1994):

- SOF indicates the start of frame.
- Identifier (ID) is used to arbitrate access to the bus. This ID can be 11-bits (in standard CAN frames) or 29-bits (in extended CAN frames).
- The Remote Transmission Request (RTR) bit indicates whether the frame is a request frame or a data frame.
- Identifier Extension (IDE) indicates whether the frame is a standard or an extended format.
- The length of the data field (in bytes) is contained in Data Length Code (DLC).
- The Data field contains the message data which can be between 0 and 8 bytes in length.
- To check the frame integrity, a 15-bits Cyclic Redundancy Check (CRC) checksum is calculated and transmitted with each frame. CRC Delimiter bit is always equal to 1.
- ACK slot is transmitted as a recessive bit (value of 1). Receivers that retrieve the message correctly should overwrite this field with a dominant bit (value of 0).
- End of Frame (EOF) bit denotes that the whole frame has been terminated.

	Arbitration field		Control field			Data field	CRC field			
SOF	11 or 29 bits Identifier	RTR	IDO	RO	4-bits DLC	0 – 8 bytes	15-bits CRC	CRC del - bit	Ack	End of Frame

Figure 8-2: Layout of the CAN frame.

8.3.2.5 Message response time in CAN network

When calculating the response time for CAN messages, the deadline monotonic approach described in (Audsley, 1991) is considered. In this approach, a message of shorter deadline is given a higher priority for transmission. Tindell *et al.* (1994 and 1995) provided a detailed analysis of the CAN-message response time. Based on his model, the worst-case response time for a CAN message is represented by the

summation of message jitter (*Jm*), physical transmission time (*Cm*) and the time delay caused by bus arbitration (*Wm*) as in Equation 8-1.

$$Rm = Jm + Cm + Wm$$

Equation 8-1

Note that *Jm* takes into account the variation in the queuing time, and *Cm* is a function of the CAN baudrate as well as the message length. Punnekkat *et al.* (2000) and Nolte *et al.* (2001 and 2002) have also provided a simple analysis for CAN message response time, and demonstrated that the physical transmission time of a particular message on the CAN bus is equal to the message length (in bits) multiplied by the bit-time (T_{bit}). The bit-time was defined as the worst-case time spent by a bit to travel on the CAN bus. T_{bit} can be calculated as follows:

$$T_{bit} = \frac{1}{CAN\ Baudrate}$$

Equation 8-2

For example, at the maximum CAN baudrate (1 Mbps), T_{bit} = 1 µs. Therefore, the transmission time (*Cm*) of a CAN message with 8 data bytes and standard format (111 bits length) can be calculated as:

$$C_m = 111\ (bits) * 1\ (ms/bit) = 111\ ms$$

Similarly, the transmission time (*Cm*) of a CAN message with 8 data bytes and extended format (129 bits length) can be calculated as:

$$C_m = 129\ (bits) * 1\ (ms/bit) = 129\ ms$$

Note that the actual (physical) transmission time of a CAN message also depends on the number of any additional hardware bits inserted by the CAN physical layer for purposes such as clock synchronisation (Nolte *et al.*, 2001): this process is called *bit-stuffing* and is further described in Section 8.3.2.6.

8.3.2.6 Error-handling mechanisms in CAN

CAN protocol employs several mechanisms for error detection and correction which make it quite robust and, hence, a good match for many time-critical applications (Farsi and Barbosa, 2000). Such error detection mechanisms are: bit error and bit-stuffing error (at the bit level), and CRC error, format error and acknowledgement error (at the message level). These mechanisms are described briefly as follows.

Bit error

When the transmitter places a bit on the bus, it monitors and compares this bit with the actual bit on the bus. If the two bit levels are unequal, a bit error is flagged.

Bit-stuffing error

A mechanism known as *bit-stuffing* is used by CAN hardware for clock synchronisation (Bosch, 1991). This mechanism is described here.

Since CAN protocol uses "Non Return to Zero" (NRZ)[23] coding for bit representation, a drift in the receiver's clock may occur when a long sequence of identical bits has been transmitted on the bus. Such a drift might, in turn, result in message corruption. To avoid the possibility of such a scenario, the CAN communication protocol (at the physical layer) employs a bit-stuffing mechanism which operates as follows. After five consecutive identical bits have been transmitted in a given frame, the sending node adds an additional bit of the opposite polarity afterwards. All receiving nodes remove the stuffed (inserted) bits to recover the original data (Farsi and Barbosa, 2000; Nolte *et al.*, 2001; Nolte *et al.*, 2002; CiA, 2008). Figure 8-3 shows the basic operation of the bit-stuffing mechanism carried out in the sending CAN controller.

[23] In NRZ coding, logic "1" is implemented as high-voltage and logic "0" is implemented as low-voltage. This is the simplest way of bit encoding in data communication which can provide maximum data throughput.

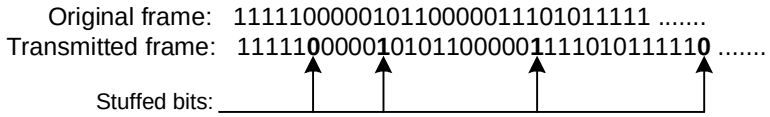


Figure 8-3: The basic operation of bit-stuffing in the sending CAN controller.

If any six identical bits are detected by the receiver, this means that an error has occurred during the transmission, therefore a bit-stuffing error is flagged.

CRC error

Every CAN message contains a 15-bit Cyclic Redundancy Check (CRC) code (see Figure 8-2). The CRC is computed by the sending controller based on the message content. All receivers that accept the message perform a similar calculation to check the integrity of the received data and flag any error.

Format error

There are certain predefined bit values that must be transmitted at certain points within any CAN message frame. If a receiver detects an invalid bit in one of these positions, a format error will be flagged.

Acknowledgement error

If a transmitter determines that a message has not been acknowledged, then an acknowledgement error is flagged.

Please note that the CAN protocol employs complex algorithms to distinguish between temporary errors and permanent failures. More specifically, each node in the network implements one counter for transmit-error and one counter for receive-error. The values of these counters increase when errors occur and decrease when a message is successfully transmitted. When the network starts, all nodes are in the Error Active Mode. When errors begin to occur, error counters begin to count until it reaches a certain threshold after which the node should enter the Error Passive Mode. If the errors continue to occur, then the device will take itself off the bus by going to Bus-Off Mode (Bosch, 1991; Farsi and Barbosa, 2000).

8.3.3 Alternative network protocols

8.3.3.1 Introduction

In this section, some alternative network protocols to CAN are described briefly. Such protocols mainly include: Time-Triggered Protocol (TTP) and FlexRay as dedicated platforms intended to meet the standard for highly-reliable, safety-critical embedded systems. Other protocols which are generally used in multi-processor embedded designs are also discussed in outline. These include: RS-485, Local Interconnect Network (LIN) and Ethernet.

8.3.3.2 Time-Triggered Protocol (TTP)

Time-Triggered Protocol (TTP) (Kopetz, 2001; TTTech, 2008) was originally developed for high-dependability, hard real-time applications. Kopetz provided a detailed comparison between the CAN and TTP protocols in terms of operational principle, protocol services, dependability and system level properties. He concluded that CAN is more suitable for soft real-time systems where flexibility is essential, while TTP is more suitable for hard real-time systems where composability and dependability are more essential than flexibility.

As a summary, the main characteristics of TTP protocol are:

- Applicable to hard real-time systems.
- Provides a time-triggered communication strategy.
- Extendable (new nodes can be easily added to the system if transmission slots for the added nodes have been reserved in the original design).
- Latency jitter is constant.
- Provides fault-tolerant clock synchronisation service (in microsecond range).
- Speed up to 2 Mbps. TTTech (2008) stated that the controllers used today can support 25 Mbps synchronous and 5 Mbps asynchronous transmissions.
- Frame size from 21 – 149 bits (21 control bits and between 0 – 128 data bits). Any frame can be either initialisation frame or normal frame (only two frame types).
- Uses Modified Frequency Modulation (MFM) bit coding.

Briefly, media-access in TTP is controlled by conflict-free TDMA strategy (Kopetz, 2001). Within the TDMA cycle (round), every node is allocated a time slot for transmission. Every TTP controller contains a Message Descriptor List (MEDL) which holds information about the node that is allowed to send at a particular point in time and which message it will send. Every TTP controller also contains two replicated channels in order to be able to tolerate a loss of one channel if occurred (see Kopetz, 2001 for additional explanation).

TTP/C (Kopetz, 2001) is an integrated communication protocol for hard real-time, fault-tolerant distributed systems. TTP/C is a member of the TTP protocol family where C indicates that it satisfies SAE (Society of Automotive Engineers) Class C requirements for hard real-time, fault-tolerant communication in the automotive area. Excellent features can be provided by the use of this protocol. For example, it provides hard real-time message delivery with minimal jitter and supports fault-tolerant communication mechanisms, such as clock synchronisation (Poledna and Kroiss, 1998). The overhead in the TTP/C is kept to minimum levels with the provision of highest data efficiency.

8.3.3.3 FlexRay

FlexRay (FlexRay, 2005; Litterick and Brenner, 2005) is a fault-tolerant, time-triggered communication protocol developed to meet the standard for safety-critical, real-time control systems (e.g. X-by-wire systems). It is a combination of Byteflight (Byteflight, 2008) protocol and TTP/C (Kopetz, 2001) protocol. Like TTP, FlexRay supports a number of fault-tolerant mechanisms which made it suitable for systems requiring high degree of robustness and dependability.

The time line in FlexRay is divided into two channels allowing synchronous (time-triggered) and asynchronous (event-triggered) communications. In the asynchronous, data bandwidth is shared by all nodes to provide high bandwidth efficiency, and the media-access is controlled by Byteflight and mini-slotting protocols. Speed in the FlexRay can be more than 10 Mbps.

In particular, a bit-synchronisation feature in FlexRay – as opposed to bit-stuffing mechanism in CAN – is considered in a little bit more detail. To achieve higher

synchronisation and avoid any possibility of drift in the receiving clocks, the FlexRay protocol implements a Byte Start Sequence (BSS) mechanism which provides the receiving nodes with bit timing information. The BSS contains two alternating bits (logic "1" followed by logic "0"), allowing a transition in the signal level around each byte. This bit encoding, along with other encoding mechanisms (see FlexRay, 2004), result in a constant transmission delay for each message scheduled to transmit in the "static" segment of the communication cycle while maintaining the synchronisation between communicating clocks. For more details see (Rushby, 2001; Kopetz, 2001; FlexRay, 2004).

8.3.3.4 Other network protocols

Other network protocols which have also been used in multi-processor embedded designs include: RS-485 (Leen *et al.*, 1999), Local Interconnect Network (LIN) (Specks and Rajnak, 2000) and Ethernet (Metcalfe and Boggs, 1976) protocols. These protocols are described briefly here.

RS-485 is a serial communication standard produced by the Electronics Industry Association (EIA, 2005) and aimed at transferring data between the desktop PC and microprocessors, or between two (or more) microprocessors in a distributed embedded system. RS-485 can have a data rate of up to 10 Mbps (or even more with new transceivers), has a one twisted-pair line allowing simple implementation and is a "multipoint" communication standard in which up to 256 nodes can be connected to the network (Pont, 2001; Leen *et al.*, 1999).

Local Interconnect Network (LIN) is a UART-based communication protocol developed for automotive sensors and actuators. LIN provides a cost-effective communication choice for one Master and multi-Slaves in a local connection. The Master node in LIN usually communicates with high-level networks such as CAN to achieve more benefits to the local sensors and actuators (Specks and Rajnak, 2000). Figure 8-4 shows a schematic example of a LIN network.

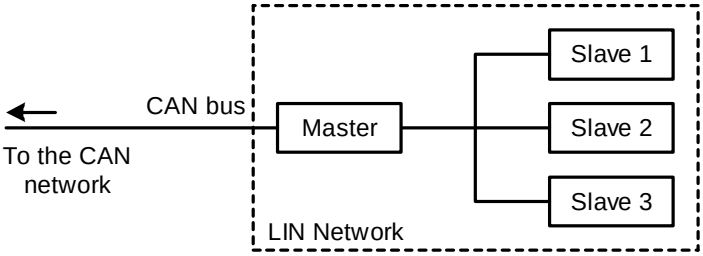


Figure 8-4: Example of a local LIN network connected to a major CAN network.

Ethernet is the most Local Area Network (LAN) protocol used across the world (CISCO, 2008). It provides a transfer rate of up to 100 Mbps based on using twisted pair and fibre optic media. Ethernet is known to be very widely used as a consequence of its widespread availability, high flexibility and easy installation, usage, maintenance and management. In Ethernet, multiple LANs can be linked together via advanced switching devices to create extended LAN networks.

In contrast to CAN, Ethernet networks apply a Carrier Sense Multiple Access / Collision Detection (CSMA/CD) protocol to provide equal chance for each node in the system to access the communication bus. This protocol provides the capability to detect frame collisions when two (or more) nodes begin to transmit simultaneously. If a collision occurs, each station will be notified to reschedule its transmission after a random period of time. Such a mechanism has the potential to minimise the possibility of further collisions. Although more collisions are likely to take place as the network expands, the CSMA/CD can resolve majority of collisions in microseconds to avoid frame losses.

Note that Ethernet is a seven-layer protocol based on the ISO/OSI model shown in Figure 8-1. The only difference between the two models is that the data link layer in Ethernet is divided into two sub-layers: Media Access Control (MAC) and MAC-Client. For more information, see (CISCO, 2008). The basic frame format of Ethernet, as defined in IEEE 802.3 standard, is shown in Figure 8-5.

Preamble	Start Of Frame	Destination Address	Source Address	Length/Type	Data	Pad	Frame Check Sequence
7 bytes	1byte	6 bytes	6 bytes	4 bytes	64 – 1500 bytes		4 bytes

Figure 8-5: IEEE802.3 frame format.

Briefly, Preamble section contains alternating ones and zeros (e.g.1010101…) that notify the receiver of a coming frame and also synchronise its clock. Start-Of-Frame also contains alternating ones and zeros and ends with two successive ones to indicate that the next bit will be the first bit of destination address. Destination Address field identifies the node(s) to which the message is transmitted. Source Address identifies the sending node. Data field contains the transmission information which must not be less than 64 bytes: otherwise, this field must be padded to reach the 64 bytes size. Frame Check Sequence contains a 32-bit Cyclic Redundancy Check (CRC) which is used, as in CAN, to check that the received frame is free of errors (CISCO, 2008).

An important issue, which must be taken into account, is that Ethernet bus may generate EMI and be susceptible to crosstalk due to its very fast transmission rate. These effects are usualy minimised by using Shielded Twisted Pair Cabling (STP) (TechFest, 1999).

8.3.4 Why does CAN remain the most attractive solution?

This section highlights the key limitations of the discussed CAN alternative protocols in fulfilling the requirements for embedded designs which are concerned with in this project: such designs are basically built on low-cost, resource-constrained embedded microcontrollers and have a high degree of predictability and reliability requirements.

Although TTP can provide an excellent platform for hard real-time applications, its high implementation costs and less availability (compared to CAN) have made it less adopted in the design of many applications. In 2002, an article written by Charles J. Murray in EE Times website highlighted that:
“TTTech's AS8202 communication processor, unveiled last spring, supports the TTP for a cost of about $3 per chip, but such prices are based on volumes of 5 million chips, the company said.” (Murray, 2002).

On September 2004, soon after this project started, a personal contact with TTTech staff was made by Devaraj Ayavoo (an ex-member of the ESL research group) enquiring about the price for a complete TTP-development set. He found out that a TTP-development cluster – which supports up to 64 nodes – would cost around €15000 (this was a TURN price for universities only). For the TTP/C controller, he was told that there were only prototyping samples available for testing which would cost around €20 per chip. The TTTech staff also pointed out that network configuration tools would also be required with the TTP hardware set. This seemed likely to add additional cost to a system implemented with this protocol. Later in 2006, Devaraj Ayavoo (in his PhD thesis) confirmed that the cost of the TTP protocol was still seen much higher than CAN protocol. Furthermore, it is worth noting that TTP controllers are still not widely supported by COTS microcontroller boards.

Similarly, FlexRay has not gained widespread popularity and is still difficult to be found on COTS microcontroller platforms. Ayavoo (2006) provided a comparison between some of the characteristics of four well-known network protocols: CAN, TTCAN, TTP/C and FlexRay. He underlined that finding the most optimal solution is not a straightforward decision since each protocol of these has strengths and weaknesses. Such an argument was also supported by Short and Pont (2007). However, Ayavoo highlighted that CAN network seems to be a good match for many automotive systems mainly due to its low implementation cost. Short and Pont (2007) argued that under some circumstances, where (for example) cost is not an issue or the bandwidth of CAN network is insufficient, using advanced protocols such as FlexRay or TTP/C may be an appropriate solution. They, however, noted that:
"Due to a lack of user experience with [FlexRay and TTP] protocols, and their comparatively high cost, it may be desirable for system developers to continue to use CAN where this is practical."

If more basic protocols such as RS-485, LIN and Ethernet are compared with CAN, the following observations can be made.

Unlike CAN, the UART-based protocols, such as RS-485, are so simple and have no error-checking capabilities. This simply means that, for high determinism, error handling mechanisms would be carried out in the application (software) layer leading to

an increased complexity of the system implementation in terms of processing and memory overheads. Moreover, the data bandwidth used in such a protocol is often insufficient as only one data byte can be transmitted at a time. Accordingly, RS-485 cannot compete with well-designed protocols like CAN for high determinism (reliability). One more issue with this protocol is that the use of UART to connect the microcontroller to the network may not be practical, since the number of available on-chip UART transceivers are usually limited and may be needed for communication with a PC during the operation time.

LIN network is, indeed, not designed to operate at high speeds. The maximum reachable speed in LIN can be up to 20 Kbps (STMicroelectronics, 2002). Overall, LIN is a low-cost solution used only to connect sensors and actuators locally with the embedded processor that is connected to a larger network such as CAN.

Despite the great advantages of Ethernet, it can still have some drawbacks. For example, the CSMA/CD mechanism to solve frame conflict problems in Ethernet makes it very sensible to high bus load during which only 60% of the bus throughput is actually utilised: to reduce the impact of such a mechanism, the network must run at very high speeds (Shandle, 2003). In contrast, CAN addresses this problem by employing a clever principle of arbitration based on message priorities (CSMA/CA: see Section 8.3.2.2). On the other hand, Ethernet is not widely available on COTS microcontroller boards. In general, Ethernet is less popular than CAN and cannot be used to achieve the level of determinism that CAN achieves.

In the study carried out recently by Short and Pont (2007), it has been argued (and practically demonstrated) that experience gained with CAN over the past years allows the creation of extremely reliable systems using this protocol, only with a little more care to be taken at the design and implementation process. An example of a highly-reliable CAN system implementation was described and proved to be effective in dealing with major CAN limitations such as inability to support reliable group communication and bus redundant arrangements.

As a result, CAN remains the most preferred network protocol by many engineers mainly due to its simplicity, low-cost, widespread availability and extensive use in

industrial systems as compared to other protocols. In his article published in 2003 and titled "CAN: Network for Thousands of Applications outside Automotive", Jack Shandle noted that:

"CAN may be overlooked by some design engineers because of its simplicity and modest bus speeds compared to Ethernet. But considering the fact that it has its roots deep in the automotive industry, dismissing it may be a mistake. The benefits of the automotive connection mean both mass-market semiconductor pricing and rock solid infrastructure support. So it's not surprising that CAN continues to grow in both market size and application diversity"

8.4 Scheduling protocols for multi-processor systems

8.4.1 Introduction

Although CAN supports event-triggered communication between nodes, it can be set to work in a time-triggered way by employing high-level protocols on the existing CAN hardware. There has been a great deal of previous work on developing techniques that enable time-triggered communication on CAN fieldbus (e.g. Turski, 1994; Broster and Burns; 2001; Broster, 2003; Donnelly and Cosgrove, 2004).

However, key successful work in this area have led to the development of well-designed, high-level protocols such as TTCAN (Fuhrer *et al.*, 2000) and S-C scheduler (Pont, 2001). Each of these protocols is outlined in this section. However, for the purpose of this study, more consideration is given to the S-C scheduling protocol.

8.4.2 Time-Triggered Controller Area Network (TTCAN)

8.4.2.1 Introduction

One consequence of employing NDBO mechanism in CAN arbitration (see Section 8.3.2.2), is a creation of distributed network-wide message queue (Leen and Heffernan, 2001). This, in turn, leads to a possible scenario where some messages – of lower priorities – are delayed indefinitely by higher-priority messages and, hence, miss their deadlines. From this example, two drawbacks of CAN are addressed: the possibility of missing deadline, and the non-deterministic message transmission latency time.

To allow better scheduling strategy for high-reliability and safety-critical embedded applications, time-triggered CAN (TTCAN) protocol was developed (Kopetz, 1997; Fuhrer *et al.*, 2000; Hartwich *et al.*,2002; Leen and Heffernan, 2001; Leen and Heffernan, 2002; Muller *et al.*, 2002; CiA, 2008; Ryan *et al.*, 2004; Short and Pont, 2007).

8.4.2.2 Overview of TTCAN operation

TTCAN is viewed as a "time-triggered" communication protocol in which the message transactions are initiated based on the time progression (Kopetz, 1997). TTCAN provides a high level protocol built on the CAN data link layer and physical layer to allow communication in time-triggered fashion as well as in event-triggered fashion (CiA, 2008). Time-triggered communication in TTCAN is basically achieved by employing a Time Division Multiple Access (TDMA) communication scheme (Ryan *et al.*, 2004).

The communication process in TTCAN is based on the following principles (Fuhrer *et al.*, 2000; Ryan *et al.*, 2004). A time Master transmits a regular *reference message* in order to create a global time base. The nodes across the network must synchronise their clocks according to this reference message. Each node in the system can only transmit data within a pre-allocated time slot (window) following the reference message. The pattern of a reference message followed by time windows is called basic cycle (BC) (Fuhrer *et al.*, 2000). The sequence of basic cycles forms a matrix cycle (MC). Ryan *et al.* (2004) defined the MC as the fixed pre-defined schedule for message exchange. They also provided an example of a matrix cycle consisting of four basic cycles (see Figure 8-6). Note that "merged arbitration" windows may contain more than one CAN message. For further information, refer to Ryan *et al.* (2004). The TTCAN protocol uses a synchronisation method with a maximum accuracy of +/- 1 bit time using a combination of hardware and software. The protocol supports a static TDMA schedule and provides "empty" slots that allow normal message arbitration for dynamic messages (Short and Pont, 2007).

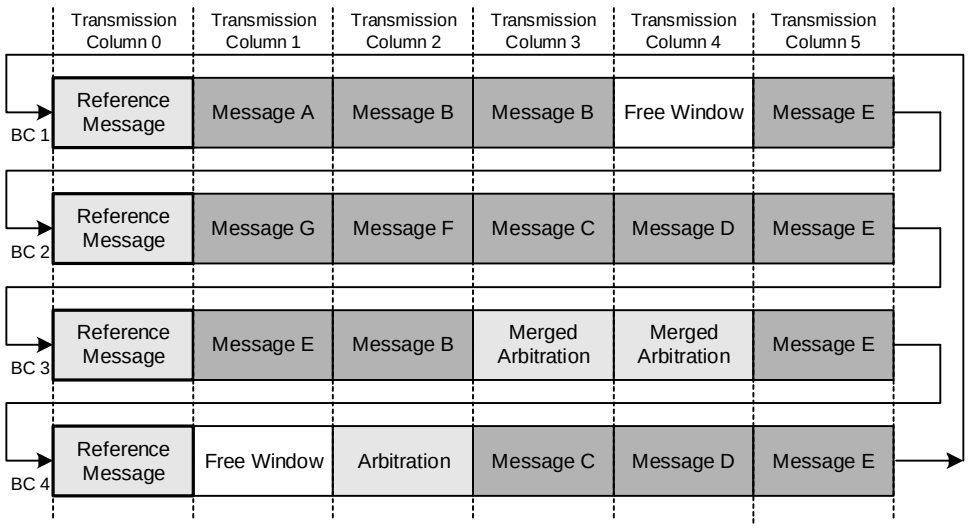


Figure 8-6: Example of TTCAN Matrix Cycle. The figure is reproduced from (Ryan *et al.*, 2004).

One clear advantage of TTCAN is that it exploits the underlying CAN error handling capabilities, whilst improving the overall timing performance of CAN aiming to achieve highly-predictable and deterministic network operations.

Various studies have considered the use of TTCAN in various application domains due to its simplicity and robustness: for more details, see (Fuhrer *et al.*, 2000; Ryan *et al.*, 2004; Rodriguez-Navas *et al.*, 2003).

8.4.3 Shared-Clock (S-C) protocol

8.4.3.1 Introduction

Despite that TTCAN provides a good communication platform for many real-time systems, the TTCAN hardware is not widely supported by the COTS microcontroller boards. Moreover, it was argued that a full implementation of TTCAN requires dedicated hardware that is not yet widely available (Short and Pont, 2007). This, in turn, increases the complexity and, hence, the cost of implementations using this protocol.

An alternative solution to provide a reliable time-triggered communication on CAN without the need for additional hardware or software clock synchronisation algorithms is to implement a “software-based only” protocol which will basically organise the transmission of messages (and hence the operations of tasks) in a timely manner whilst maintaining a high resource efficiency.

Over the last years, the ESL researchers have been involved in the development of reliable embedded systems based on time-triggered software architectures. The previous work in this area has considered the development of techniques for both single- and multi-processor embedded designs. In the case of multi-processor designs, they have demonstrated that a "Shared-Clock" (S-C) scheduling protocol – used in conjunction with the TTC scheduling algorithm – can provide a simple, flexible and predictable platform for many real-time embedded systems (Pont, 2001). This protocol is described in this section.

8.4.3.2 Shared-clock (S-C) scheduler

The "Shared-Clock" (S-C) architecture, developed by Pont (2001), was aimed to provide a simple and low-cost software framework for time-triggered systems without requiring specialised hardware. The S-C scheduler operates as follows (Figure 8-7). On the Master node, a conventional (co-operative or hybrid) scheduler operates and the system is driven by periodic interrupts generated from an on-chip timer. On the Slave nodes, a very similar scheduler operates. However, on the Slaves, no timer is used: instead, the Slave scheduler is driven by interrupts generated through the arrival of periodic "Tick" messages sent from the Master node. By doing so, all nodes will be synchronised according to one reference clock (which is the Master clock).

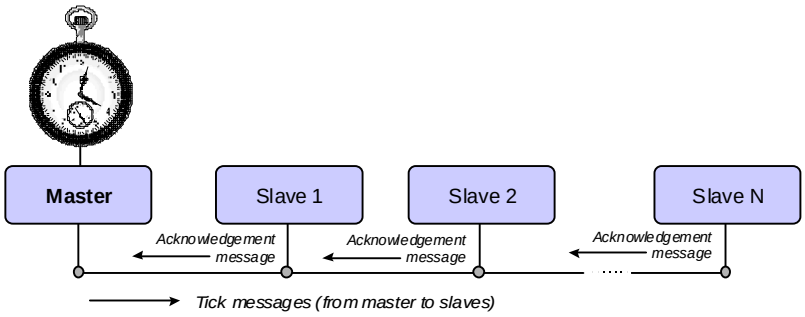


Figure 8-7: Simple architecture of Shared-Clock (S-C) scheduler.

Overall, the S-C scheduler is extremely simple and supports a number of low cost (but effective) error-handling mechanisms (Section 8.4.3.3). The network communications follow a Time-Division Multiple Access (TDMA) protocol, and the system behaviour is highly-predictable (Ayavoo *et al.*, 2007). In such a scheduling protocol, the Master Tick message holds data for a particular Slave or a group of Slaves. The first byte of the

transmitted data is therefore reserved for the Slave or Group identifier (ID) to which the tick message is addressed. Only the addressed Slave(s) must reply a form of acknowledgement "Ack" message to the Master straight after the Tick message is received (see Pont, 2001 for more details).

Please note that in such architectures, the jitter between the timing of Master and Slave nodes can be quite high due to bit stuffing mechanism in CAN hardware. To provide an effective solution to this problem, a range of data coding techniques have been proposed and evaluated in this project. These are "XOR masking", "software bit stuffing" and "eight-to-eleven modulation". Such techniques are described in detail in Appendix E.

8.4.3.3 Error-handling mechanisms in S-C protocol

To achieve high degree of reliability, the S-C scheduler applies several error detection and recovery mechanisms. For example, a Slave can detect an error on the Master by measuring the time period between every two ticks and once it exceeds the tick interval (which is deterministic) the Slave knows that an error has occurred in the Master node. Since the S-C follows the TDMA protocol allowing the Master to talk to each node individually, the Master can easily detect an error on any Slave if no "Ack" message is received from a particular Slave within its allocated time interval.

Once an error is detected in the S-C network, appropriate handling mechanism(s) must be employed. For example, when a Slave detects a failure in the Master, it enters a "safe state" and waits until an appropriate series of "start" commands are received from the Master. The situation is more complicated when a Master detects a failure in one of the Slaves. In this case, the Master can have three options (Pont, 2001):

- Enter a safe state then shut down the whole network.
- Reset the network.
- Start a backup Slave.

8.5 Conclusions

Having completed the work on single-processor embedded systems in the previous chapters, this chapter began to address the implementation issues for (distributed) multi-

processor embedded systems. The chapter reviewed key previous work in this area and linked it to the work concerned with in this thesis.

The focus of the discussions in this chapter was on systems using Controller Area Network (CAN) protocol for message transmission. The chapter reviewed CAN in detail and compared it with other network protocols. The key features of CAN – over alternative protocols – were summarised as simplicity, low-cost, availability and widespread use in industry.

The discussions then moved on to consider ways for implementing "high-level" protocols on the CAN hardware to improve its operational characteristics: in particular, to allow the network operate in time-triggered manner rather than event-triggered. Particular concern was given to the Shared-Clock (S-C) scheduling protocol which offers a very flexible and predictable platform for many real-time embedded systems.

The next chapter describes how the S-C scheduling protocol can be implemented on CAN network and reviews a number of possible implementations for such a protocol.

Chapter 9

TTC-SCC scheduler implementations

9.1 Introduction

As in the discussions provided in Chapter 8, despite that CAN supports an event-triggered communication, time-triggered behaviour can be achieved if simple, cost-effective software protocols (such as Shared-Clock (S-C) schedulers) are implemented with the CAN hardware (Pont, 2001).

Like any other scheduler, the S-C scheduling protocol can have a large number of possible implementation options, where each implementation is expected to produce different patterns of behaviour at the system run-time. In another word, the 'one-to-many' relationship (discussed in Chapter 3) does also apply between the S-C scheduling protocol and its low-level implementations.

As noted earlier in this thesis, it is impossible to cover all possible implementation options for a given scheduler in a single study. Therefore, this chapter reviews a selective set of the various possible ways in which S-C scheduler can be implemented in low-cost embedded systems. Such a representative set of S-C schedulers will be used as a basis for assessing the effectiveness of the STC technique in testing multi-processor embedded designs.

Note that this chapter reviews five different implementations for S-C scheduler. Four implementations have been taken from studies conducted previously in the ESL research group, while only one implementation is proposed in this project[24].

[24] The work described in this chapter has been adapted from the study presented in the author's publication [2] listed in page xvi.

9.2 Implementing S-C scheduler on CAN protocol

The S-C scheduler can be implemented on a wide-range of network protocols used in the design of multi-processor embedded systems, such as CAN, RS-485, TTP and FlexRay. The work presented in this study is, however, focused on implementations using CAN network protocol. The multi-processor systems considered in this study are based on the following three-level implementations:

- TTC-Dispatch scheduler implemented in each individual node to achieve time-triggered operations of scheduled tasks.
- CAN network protocol implemented as a hardware platform on which the communicating nodes transmit their messages.
- S-C scheduling protocol – implemented on top of the CAN – as a software platform to achieve time-triggered communications between the nodes connected in the embedded network.

The resulting system is best described as a "TTC-SCC" scheduler[25] (Ayavoo *et al.*, 2007). Overall, the use of TTC-SCC scheduler can be so attractive due to its exploitation of the error handling features offered by the underlying CAN hardware, whilst – at the same time – allowing the network to behave in a highly-predictable time-triggered manner. The TTC-SCC scheduler has been widely adopted by ESL researchers to implement various distributed embedded applications. For example, Ayavoo *et al.* (2004); Short and Pont (2005) demonstrated how such a scheduling protocol can be used to implement different versions of automotive cruise control system for use in passenger car. The testbed considered in (Ayavoo *et al.*, 2005; Short *et al.*, 2006; Short *et al.*, 2007), which was based on X-by-wire control system, also used this protocol. Similarly, Edward (2004) built an inverted pendulum testbed using several nodes, associated with different sensors and actuators, which communicated with each other through a TTC-SCC scheduling protocol.

[25] TTC-SCC is an abbreviation for Time-Triggered Co-operative, Shared-Clock, CAN.

9.3 TTC-SCC1 scheduling protocol

9.3.1 Introduction

The implementation of the first version of the TTC-SCC scheduler – as described in (Pont, 2001; Ayavoo *et al.*, 2007) – is presented in this section. This particular implementation will be referred to as TTC-SCC1.

9.3.2 Overview of the scheduler implementation

The TTC-SCC1 scheduler is a simple version of the TTC-SCC scheduling protocol. TTC-SCC1 follows a Time Division Multiple Access (TDMA) protocol in which the Master node communicates with only one Slave node per tick interval. The scheduler is based on the following arrangements: first byte of the transmitted data is reserved for the Slave Identifier (ID) to which the Master "Tick" message is addressed. Only the addressed Slave will reply an acknowledgement "Ack" message to the Master where this message must be sent back within the same tick interval in which the "Tick" message is received.

The described mechanism is used by the Master to detect network and node failure. More clearly, at each tick interval, the Master node checks if a valid "Ack" message is received from the addressed Slave in the previous tick. If not, then the necessary actions might be taken, for example, starting a backup Slave, or going into a safe mode. If a correct "Ack" message has been received from that Slave, the Master will send Tick message on the CAN bus which addresses the next Slave node, and so on.

Figure 9-1 illustrates an example of the TDMA round (cycle) for a TTC-SCC1 network with one Master and three Slaves, where "Tick" messages originate from the Master and the "Ack X" message is transmitted back from "Slave X". The figure shows that TTC-SCC1 follows a round-robin message scheduling approach in which all Slaves are given equal time to transmit their messages. The figure clearly shows that the TDMA round in the TTC-SCC1 is equal to the number of Slaves multiplied by the width of the tick interval. Given that N is the number of Slaves and T is the tick interval, the TDMA round can be calculated as follows:

$$TDMA1 = NT$$

Equation 9-1

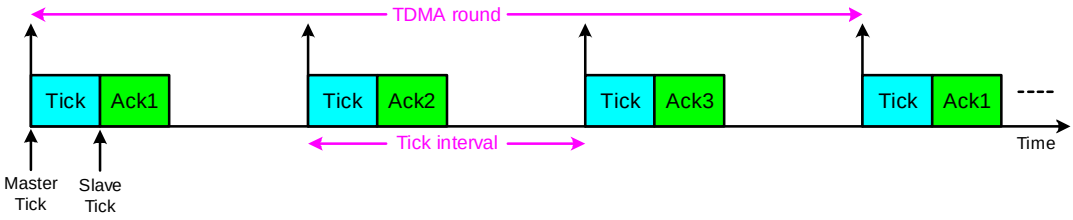


Figure 9-1: TDMA round for a four-node system using TTC-SCC1 scheduler.

To implement TTC-SCC1 scheduler, only two CAN messages are exchanged within a tick interval: “Tick” and “Ack” messages. The “Tick” message is assigned a higher priority than the “Ack” message. This is because the Master Tick messages are used to generate the timing beat of the whole network and manage the transmission of messages. Therefore, the first CAN Message Object (CMO 0) in the Master node must be configured to send “Tick” messages where the second CAN Message Object (CMO 1) must be configured to receive “Ack” messages. The same configurations are to be considered in the Slave nodes. However, in Slaves, CMO 0 is configured to receive “Tick” messages from the Master and CMO 1 is configured to send “Ack” messages to the Master. Furthermore, the timer interrupt on the Master node is enabled to generate periodic interrupts for triggering the Master scheduler and, hence, sending “Tick” messages to the Slaves. On the Slave nodes, the CAN interface will be configured to generate a CAN interrupt on arrival of a valid “Tick” message, while Slave timer interrupts are totally disabled.

Overall, CAN messages can have up to eight bytes data bandwidth. However, in any S-C scheduler, one byte in each (Tick or Ack) message is reserved for Slave ID. This allows up to seven bytes per message for data transfers between nodes. Please note that the Slave ID byte in the Ack message is used by the Master to check that a given Slave has responded correctly and hence has no failure.

9.4 TTC-SCC2 scheduling protocol

9.4.1 Introduction

The TTC-SCC2 scheduler provides a small (but effective) modification to the original TTC-SCC1 scheduler. An overview of the TTC-SCC2 scheduling protocol is presented in this section. The particular implementation discussed in this section has been described in detail elsewhere (Pont, 2001; Ayavoo *et al.*, 2007).

9.4.2 Overview of the scheduler implementation

The round-robin approach used in the TTC-SCC1 scheduler to communicate with the Slave nodes may not be efficient in some networks. For example, in some applications, the Master node may need to communicate with a particular Slave node more frequently than the other Slaves. This is (for example) to check the Slave's status or acquire some data samples. In order to achieve this, an enhanced implementation of the scheduler is required: this is referred to here as "TTC-SCC2".

The TTC-SCC2 scheduler provides a flexible TDMA round. For example, the status of Slave 1, in the example shown in Figure 9-1, may need to be checked more frequently than the status of Slave 2 and Slave 3. In this case, the TDMA round used must be amended to meet such an application requirement. An example of appropriate TDMA round that can be used for such a system is illustrated in Figure 9-2. In the example in the figure, the TDMA round is equal to four tick intervals (i.e. 4T). This can be broken down into 2T (for Slave 1 Ack message which is allowed to transmit twice in the TDMA round) plus 2T (for Slaves 2 and Slave 3 Ack messages, each transmitted once in the TDMA round). More generally, for N Slaves, the TDMA round can be calculated as follows:

$$TDMA2 = (2N\text{-}2)T$$

Equation 9-2

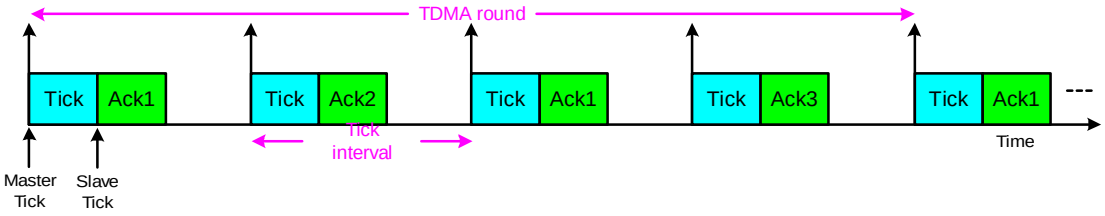


Figure 9-2: A simple TDMA configuration for a four-node system using TTC-SCC2 scheduler.

Figure 9-2 shows a simple example of TTC-SCC2 scheduler for a system which has a small number of Slave nodes, and the Master node communicates with only one of the Slaves more frequently than communicating with the other Slaves (in this case, at every other tick).

In a more complicated scenario, assume that the system has five Slave nodes, and the Master is required to check the status of (for example) Slave 1 at every three ticks (instead of two ticks as in the example shown in Figure 9-2). This is illustrated in Figure 9-3. In the example shown in the figure, the TDMA round is equal to 6*T*. This can be broken down into 2*T* (for Slave 1 Ack message, the only message which is designated two tick intervals) plus 4*T* (for Slaves 2, 3, 4 and 5 Ack messages).

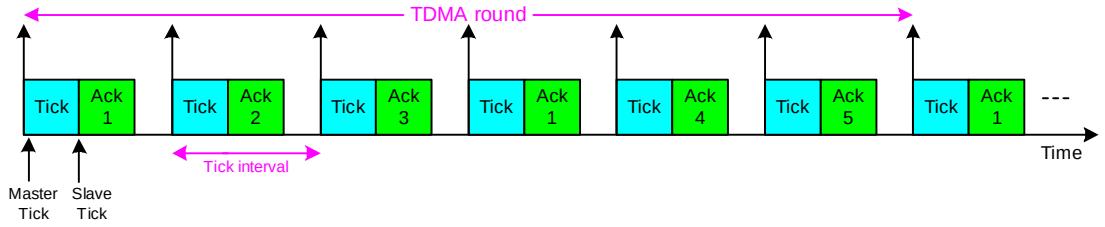


Figure 9-3: A more complicated TDMA configuration for a six-node system using TTC-SCC2 scheduler.

Figure 9-3 shows an example of TTC-SCC2 scheduler for a system which has a comparatively large number of Slave nodes and one of these Slaves requires checking more frequently but at a lower rate than that required in the previous example (in this case, at every three ticks).

To make the calculations more general, given that *N* is the number of Slaves, *T* is the tick interval, S_F is the frequently checked Slave, and *F* is the frequency of S_F "Ack" messages (in "ticks"), the TDMA round can be calculated as follows:

$$TDMA2 = (N - 1 + \frac{N-1}{F-1}).T = (N + \frac{N-F}{F-1}).T = N.T + (\frac{N-F}{F-1}).T$$

Equation 9-3

Of course, the number of Slaves which need to be checked at higher frequency is not limited to one: there might be a few other Slaves which the Master is required to communicate more frequently than the other Slaves. This would add more complexity to the TDMA calculation expressed in Equation 9-3.

In general, TTC-SCC2 scheduler has been intended to meet the requirements of any real-time control application. Therefore, the configuration of the TDMA round in such a scheduler is considered an application-specific design parameter which allows the Master to communicate with Slaves in an arbitrary way. For example, consider the system illustrated in Figure 9-4. Here, the system has five Slaves and the TDMA round is equal to 8*T*. It is impossible to find a general formula which can be used to calculate the TDMA round for any system implemented using TTC-SCC2 scheduler. Instead, the TDMA round for a given system will be dependent on the number of Slaves as well as the message scheduling pattern used for that particular system.

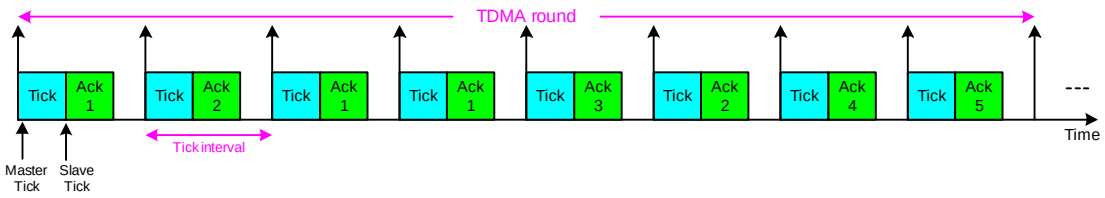


Figure 9-4: A TDMA configuration for a six-node system with arbitrary pattern using TTC-SCC2 scheduler.

Overall, to implement the TTC-SCC2 scheduler, the same configuration for CAN message objects – as described in Section 9.3.2. – is used. The only difference between the two schedulers is, again, the way the system talks to the various Slaves.

9.5 TTC-SCC3 scheduling protocol

9.5.1 Introduction

Overall, The TTC-SCC1 and TTC-SCC2 schedulers are very simple and allow the creation of low-cost, time triggered CAN-based networks with highly-predictable patterns of behaviour (Ayavoo *et al.*, 2007).

However, the two scheduling protocols have some limitations. For example, in TTC-SCC1 and TTC-SCC2 schedulers, Slave-to-Slave communication is not permitted as all communication is directed via the Master node (through "Tick" and "Ack" messages). This causes the transmission time of data between any two Slaves to be comparatively long. Moreover, the time taken to detect the failure of any Slave node can be very long, since the Master checks the status of all (or some) Slaves only once per TDMA round. As the TDMA round goes larger, the failure detection time would increase correspondingly. Finally, tasks running on the Slave nodes will suffer from high jitter due to CAN bit stuffing in the Master Tick messages (Ayavoo *et al.*, 2007). Note that a complete set of results which show such characteristics is provided in Chapter 11.

To resolve some of the outlined shortcomings of the TTC-SCC1 and TTC-SCC2 schedulers, the TTC-SCC3 was developed. An overview of this scheduling protocol is presented in this section. Note that the particular implementation discussed here has been described in detail elsewhere (Ayavoo *et al.*, 2007).

9.5.2 Overview of the scheduler implementation

The TTC-SCC3 scheduler provides the facility for all Slave nodes to transmit their Ack messages within one tick interval. As with TTC-SCC1 and TTC-SCC2, each time a Tick message is sent from the Master, an ID is also sent within the message. However, with TTC-SCC3, this is a "Group ID" (rather than a Slave ID). This simply means that – if there is more than one Slave in a particular group – all Slaves in the group will send their Ack messages simultaneously. In this case, it is the responsibility of the CAN controller to deal with any collision between messages. Thereafter, the Master node needs to ensure that all Slaves in the group addressed in the Tick message have replied back before transmitting the next Tick message, and so on.

To better explain the TTC-SCC3 scheduler, assume a four-node system as illustrated in Figure 9-5. The figure shows how Slave Ack messages can be scheduled in a simple TTC-SCC3 scheduler, where the three Slaves are permitted to transmit in the same tick interval. In this case, the TDMA round is equal to the tick interval.

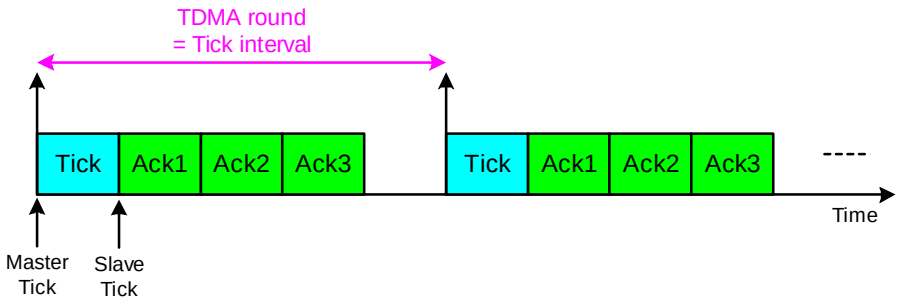


Figure 9-5: A simple TDMA configuration for a four-node system using TTC-SCC3 scheduler.

In a more complicated scenario, assume that a system has *N* Slaves. The scheduler has the option to schedule the Ack messages for all *N* Slaves in one tick interval, or alternatively divide them between two tick intervals. For example, *m* Slaves can send Ack messages in the first tick interval while the remaining *N*-*m* Slaves send Ack messages in the second tick interval (where $m < N$). In general, the TDMA in such a scheduler can be extended across multiple tick intervals. Figure 9-6 illustrates two possible ways to schedule messages in a seven-node system using TTC-SCC3 scheduler. In Configuration A, the TDMA round consists of two tick intervals, each allocated for three Slaves to send their Ack messages. In contrast, the TDMA round in Configuration B is extended across three tick intervals, so that in each interval only two Slaves can send their Ack messages.

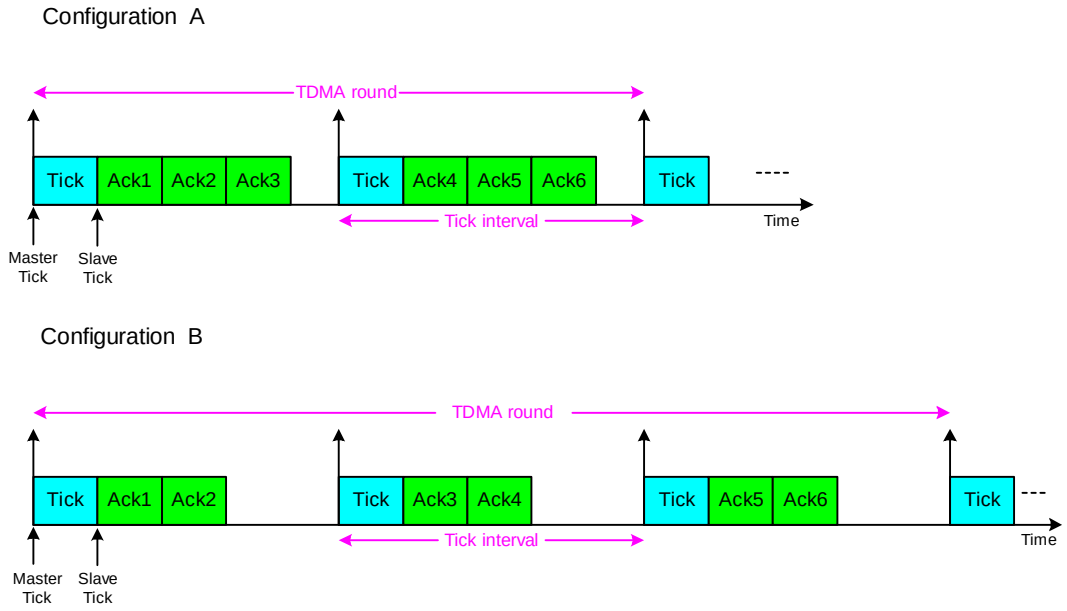


Figure 9-6: Two possible TDMA configurations for a seven-node system using TTC-SCC3 scheduler.

More generally, given that N is the total number of Slaves, m is the maximum number of Slaves replying per tick and T is the tick interval, the TDMA round can be calculated as follows:

$$TDMA3 = \frac{N\,T}{m}$$

Equation 9-4

Please note that the TDMA in TTC-SCC3 can be much shorter than TDMA in TTC-SCC1 and TTC-SCC2. For example, $TDMA1 = NT$ and $TDMA3 = NT/m$. Thus, the relationship between the two TDMA rounds can be expresses as:

$$TDMA3 = \frac{1}{m} \times TDMA1$$

Equation 9-5

Remember that in the case where $m = N$ (as in the example shown in Figure 9-5), then $TDMA3 = T$.

Overall, the TTC-SCC3 scheduler allows that messages sent from the Slave nodes can be broadcasted to both Master and all other Slave nodes. In order to allow practical

implementation for the TTC-SCC3 scheduler, each Slave Ack message must be assigned a unique CMO. Note that, as with TTC-SCC1 and TTC-SCC2 schedulers, such Ack messages should not generate CAN interrupts on arrival at other nodes.

9.6 TTC-SCC4 scheduling protocol

9.6.1 Introduction

The TTC-SCC4 scheduler is another implementation of the S-C algorithm which was adapted from the TTC-SCC3 scheduler. This section describes TTC-SCC4 scheduler briefly. The particular implementation discussed in this section has been described in detail elsewhere (Ayavoo *et al.*, 2007).

9.6.2 Overview of the scheduler implementation

The motivation behind the development of TTC-SCC4 scheduler is to separate between data messages and time-control messages in order to achieve higher predictability. More specifically, the Master node in a TTC-SCC4 scheduler is set to transmit Tick messages which contain no data. Such messages are used only to synchronise the local time of all other nodes. In another word, the Master node has the responsibility to generate the "heartbeat" of the network and then control the message transmissions over the network. For example, it still has the responsibility to check the status of all Slave nodes and deal with any node-failure. Moreover, it decides which Slaves must transmit in each tick interval if the TDMA round is extended across multiple tick intervals (as in Figure 9-6). In this case, the Master will use only one data byte for "Group ID" to which particular messages are sent. Also, a new Slave node is needed to transmit the Master data messages. Figure 9-7 illustrates how the TDMA round in the system shown in Figure 9-5 will look like if TTC-SCC4 is used.

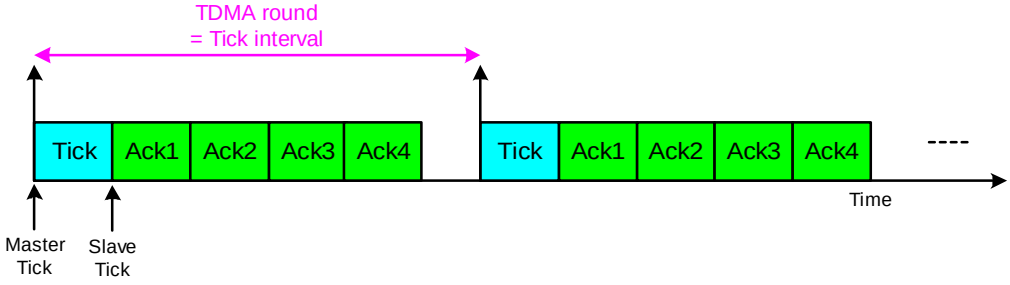


Figure 9-7: A simple TDMA configuration for a four-node system using TTC-SCC4 scheduler.

It can be clearly noticed from the figure that the number of Slaves has increased by one. This implies that the TDMA round in this scheduler is calculated as:

$$TDMA4 = \frac{(N+1)T}{m}$$

Equation 9-6

Where N is the number of original Slaves, m is the maximum number of Slaves replying per tick and T is the tick interval.

However, this simple modification to the previous S-C schedulers allows the Tick messages to be of short and fixed lengths with the result that jitter caused by CAN bit stuffing would be minimised. Remember that, in any S-C scheduler, Tick messages are sent from the Master at each tick interval to drive the Slave schedulers. If such messages have variable lengths, this is likely to introduce jitter in the timing of tasks running in the Slave nodes (see Appendix E).

Since this scheduler is built on the TTC-SCC3, it provides the same features as those outlined in 9.5.2 For example, a direct communication between any two Slaves is permitted.

On the other hand, to implement such a scheduler in practice, an additional microcontroller will be required as the number of nodes in the system has increased by one. This is a disadvantage of such a scheduling protocol.

9.7 TTC-SCC5 scheduling protocol

9.7.1 Introduction

Despite the fact that the TTC-SCC4 scheduler helps to substantially reduce the jitter in the Tick messages, the system requires – at least – one additional processor to generate the timing beat of the network. In order to maintain the low levels of jitter without using additional hardware, the TTC-SCC5 scheduler has been proposed in this project. This scheduling protocol is described in this section.

9.7.2 Overview of the scheduler implementation

In the TTC-SCC5 scheduler, the Master is configured to send out two types of messages: Tick messages and Data messages. As with the TTC-SCC4 scheduler, the Tick messages are configured to have "empty" data. This, again, means that these messages are only used to generate the time-reference for the whole network while processing no data. After a Tick message is sent out to all Slaves at each tick, the Master can then send its data in its Data message (see Figure 9-8). The TDMA round in TTC-SCC5 scheduler is calculated in the same way as in TTC-SCC3 scheduler (i.e. *TDMA*5 = *TDMA*3)

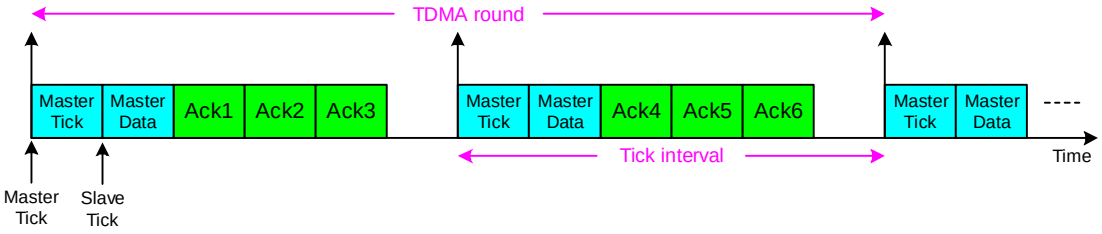


Figure 9-8: A TDMA configuration for a seven-node system using TTC-SCC5 scheduler.

Note that the TTC-SCC5 design is adapted mainly from the TTC-SCC3 and TTC-SCC4 schedulers. Therefore, using this scheduler, jitter in the Slave ticks can be significantly reduced. Moreover, messages sent by a given Slave will be broadcasted to all other Slaves, allowing a direct communication (and hence reduced message transmission times) between the Slaves.

To implement this scheduler practically, the Master node will have the following CAN message Objects (CMOs):

- 'CMO 0' which is configured to send Master "Tick" messages.
- 'CMO 1' which is configured to send Master "Data" messages.
- 'CMO 2 – CMO N+1' which are configured to receive "Ack" messages from N Slaves.

In the Slave nodes, the same configurations are to be considered. However, in Slave, 'CMO 0' is configured to receive "Tick" messages from Master, 'CMO 1' is configured to receive "Data" messages from Master, 'CMO 2' is configured to send "Ack"

messages to all nodes, and 'CMO 3 – CMO N+1' are configured to receive "Ack" messages from the other Slaves. Note that – as with TTC-SCC3 and TTC-SCC4 – each Slave node in the network is assigned a unique CMO for its Ack message in order to achieve a Slave-to-Slave communication. Also note that, when this scheduler is used, the Master Data messages and the Slaves Ack messages should not trigger CAN interrupts.

9.8 Conclusions

This chapter reviewed a selective set of implementation classes for TTC-SCC scheduling protocol for multi-processor embedded designs. The various implementations were based on the architecture described in Section 8.4.3.2 which was originally developed to provide a predictable software platform for real-time embedded applications.

It has been made clear that the majority of the TTC-SCC implementations discussed in this chapter were taken from previous studies carried out in the ESL research group. Such implementations included: TTC-SCC1, TTC-SCC2, TTC-SCC3 and TTC-SCC4 schedulers. Subsequently, one new implementation was presented which suggests a useful addition to the range of TTC-SCC schedulers. Such an implementation was called TTC-SCC5 scheduler. The key feature of this scheduler is that it provides a reduced jitter characteristic in the message transmission and maintains high resource efficiency, having the network timing controlled by one of the existing system nodes without the need for additional hardware as with TTC-SCC4 alternative.

It is essential to highlight that, although jitter due to bit stuffing in CAN hardware can be minimised by altering the scheduler architecture (as discussed in this chapter), there can be other possible ways to deal with such a problem. One suggested solution is to pre-process the transmitted data so as to ensure that CAN bit stuffing mechanism would always have the minimum effect (Nahas and Pont, 2005; Nahas *et al.*, submitted). For further reference, Appendix E reviews a wide range of data coding techniques developed in this project to reduce the impact of CAN bit stuffing on task jitter in S-C scheduling protocols.

Like Chapter 5 in the thesis, this chapter provides the basis for the work presented in Chapter 10 and 11, in which the reviewed TTC-SCC implementations form the testbeds for assessing the effectiveness of the testing technique introduced in this project for multi-processor designs.

Chapter 10

Scheduler Test Cases (STCs) for TTC-SCC schedulers

10.1 Introduction

Because of the complexity of multi-processor embedded systems, developing test cases for such systems is a nontrivial process. The work presented in this thesis attempts to address this problem (in part) by considering the combination of TTC scheduling algorithms and S-C scheduling protocols implemented on CAN networks. In more detail, the study explores the impact of using particular implementations of TTC-SCC scheduling protocol on the overall timing behaviour of multi-processor embedded designs by means of scheduler test cases.

Based on the approach discussed for single-processor systems, this chapter begins to explore ways in which the STC methodology can be extended to assess the behaviour of multi-processor embedded systems which are based on TTC-SCC scheduler. In particular, a set of proposed "scheduler test cases" (STCs) is described in this chapter with the aim to help in distinguishing the behaviour of the selective TTC-SCC scheduler implementations discussed in Chapter 9[26].

10.2 The Scheduler Test Cases (STCs) for TTC-SCC protocol

10.2.1 Introduction

This section describes the various STCs developed in this study for TTC-SCC scheduling protocol. The total number of STCs described here is five. More specifically, STC A, STC B, STC C and STC D are intended to test the system behaviour under

[26] The work described in this chapter has been adapted from the study presented in the author's publication [2] listed in page xvi.

normal operating conditions, where STC E was intended to test the system behaviour during the occurrence of errors.

As in the single-processor systems, each of the STCs presented was intended to address a particular problem that have the potential to degrade the overall system predictability.

Note that in all test cases presented here, it is assumed that the system consists of one Master and three Slaves connected via CAN fieldbus (Figure 10-1). The Master Tick message is used to drive the local time of each of the three Slaves (as discussed in Section 8.4.3.2).

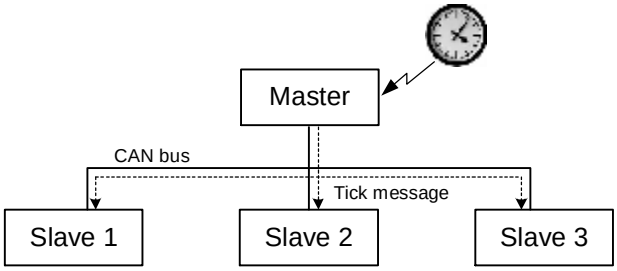


Figure 10-1: Hardware architecture of the multi-processor system used for the STCs.

10.2.2 STC A (jitter behaviour)

As indicated in Chapter 3, a particular focus of this study is on the impact of implementation decisions on system predictability which can often be measured by the levels of jitter. Generally, jitter in multi-processor embedded systems can arise from different sources, such as delay in network caused by a route consisting of several hops (Baruah *et al.*, 1999), network protocol (e.g. Ethernet, CAN) (Tindell and Burns, 1994), and the variations in message transmission times (Nolte *et al.*, 2002).

Remember that in the case of single-processor architectures, the jitter was measured at the task level. In contrast, people working on multi-processor architectures are often concerned with the jitter caused by message transmission. Such a jitter can be defined as the variation in the time taken to transmit a message from one node to another. In many applications, exchanged data messages are used by the networked processors to adjust their timing base according to one time reference (Fuhrer *et al.*, 2000). In situations

where exchanged messages have highly-variable transmission durations (i.e. jitter), this can result in unpredictable operation of the whole network.

Particularly, in TTC-SCC systems, the timing of the individual nodes is synchronised by sharing a single clock source between the various processor boards in the system: this clock is distributed through the Master Tick message (Pont, 2001). If the Master Tick message varies in length, then the Slave scheduler ticks and, hence, the execution time of tasks running on the Slave will suffer from jitter. This can be further illustrated in Figure 10-2.

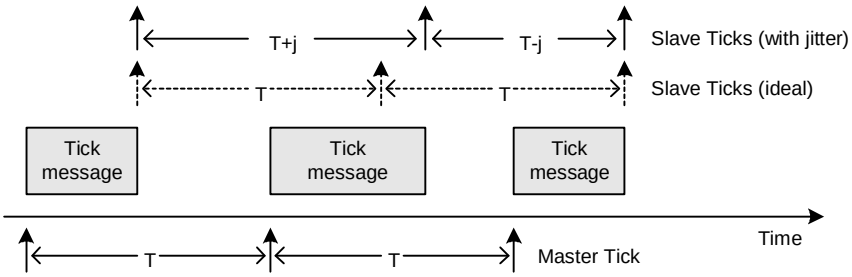


Figure 10-2: Impact of Tick message variation on the timing of Slave ticks in TTC-SCC systems.

In a CAN bus, the bit-stuffing mechanism (introduced in Section 8.3.2.6) inserts an additional bit of opposite polarity when five consecutive bits with the same polarity are transmitted on the bus (e.g. 11111 or 00000). This is aimed at providing edges to allow receivers to re-synchronise their internal timing. Whilst providing an effective mechanism for clock synchronisation in the CAN hardware, such a bit-stuffing mechanism causes the frame length to become (in part) a complex function of the data contents.

It is useful to understand the level of message variation that this process may induce. When using (for example) 8-byte data and standard CAN identifiers, the minimum message length will be 111 bits (without bit-stuffing) and the maximum message length will be 135 bits (with the worst-case level of bit-stuffing) (Nolte *et al.*, 2002). This

translates to a possible variation of approximately 22% of the total message length. For example, at the maximum CAN baud rate (1 Mbps), the variation in the message length will be 24 μs[27]. Once transmission starts, a CAN message cannot be interrupted, and the variation in transmission times therefore has the potential to have a significant impact on the predictability of systems using CAN protocol.

Overall, obtaining full synchronisation between the communicating nodes in a multi-processor embedded design is a key factor to achieve predictability. Therefore, STC A is developed to assess the jitter levels in the relative timing of Master and Slave ticks in a TTC-SCC network. As in the single-processor study, results from this STC are empirical.

In STC A, the system has one task (`Master_Task_A`) running on the Master node and a corresponding task (`Slave1_Task_A`) running on Slave 1 node. Given that "`Master_Task_A`" sends random data to "`Slave1_Task_A`" every time it is called, jitter test assesses the variation in the time delay between these two communicating tasks. Please recall that all other Slaves will receive Master data at the same instant over the CAN link. Also note that the overheads of the Master and the Slave schedulers – which are based on TTC-Dispatch (5.4) – do not introduce any jitter and, hence, the jitter observed is only caused by the communication protocol (Nahas *et al.*, 2004) [28].

10.2.3 STC B (Master-to-Slave message latency)

STC B is developed to assess the communication latency between the Master node and any Slave node in the network. Since such message latency times can be described mathematically, the output results from this STC are in the form of mathematical equations. It is worth highlighting that as the complexity of the scheduling algorithm –

[27] Note that when the baudrate is below 1 Mbps, jitter levels will be seen higher.
[28] In the implementation discussed in (Nahas *et al.*, 2004), the ISR Update function which is called prior to the tasks is set to have a fixed duration, therefore, no jitter is expected to arise from the "scheduler" code.

under test – increases, the use of theoretical (as well as empirical) results can help to provide more information about the behaviour of the system.

In this test case, the message latency between the Master and Slaves in all TTC-SCC protocols is calculated. Assume that a given Slave node needs to respond to a switch-press activity occurred on the Master board by performing some activities. If the switch-press takes place at arbitrary instants, then STC B evaluates the best-case (minimum) and the worst-case (maximum) message transmission times between the Master and the Slave node.

10.2.4 STC C (Slave-to-Master message latency)

STC C is developed to assess the communication latency between any Slave node and the Master node in the network. Results from this STC are also in the form of mathematical equations.

In this test case, the message latency between any Slave and the Master in all TTC-SCC protocols is calculated. Assume that the Master node needs to respond to a switch-press activity occurred on a given Slave board by performing some activities. If the switch-press takes place at arbitrary instants, then STC C evaluates the best-case (minimum) and the worst-case (maximum) message transmission times between the Slave and the Master node.

10.2.5 STC D (Slave-to-Slave message latency)

STC D is developed to assess the communication latency between Slave ‘X’ and Slave ‘Y’ in the network. Results from this STC are also in the form of mathematical equations.

In this test case, the message latency between any two Slaves in all TTC-SCC protocols is calculated. Assume that the Slave ‘Y’ needs to respond to a switch-press activity occurred on Slave ‘X’ board by performing some activities. If the switch-press takes place at arbitrary instants, then STC D evaluates the best-case (minimum) and the worst-case (maximum) message transmission times between Slave ‘X’ and Slave ‘Y’.

10.2.6 STC E (node-failure detection time)

Having considered that STCs B, C and D assess the behaviour of the TTC-SCC system under normal conditions, STC E is developed to assess the behaviour of the system when an error takes place. As discussed in Chapter 8, node failure is a common error in communication networks that might, in turn, reduce the overall reliability and predictability of the system. Node failure describes a situation where one or more nodes do not respond to messages sent from other nodes due to hardware / software error occurred in the receiving node.

The STC E is developed to assess the behaviour of a TTC-SCC protocol when one of the Slaves becomes temporarily out of order. The test case evaluates the worst-case time taken by the network-Master to detect the failure and hence begin to handle it. Results from this STC are also in the form of mathematical equations.

10.2.7 Memory and network bandwidth requirements

As previously done for single-processor schedulers, the memory requirements were also reported here as a means for measuring the complexity of the various TTC-SCC schedulers.

Moreover, in communication network, the utilisation of the available network bandwidth is a major factor that affects network efficiency. Therefore, the bandwidth utilisation in each TTC-SCC scheduling protocol was also reported.

Please note that, in the single-processor study, CPU overhead and average power consumption in each TTC scheduler were measured and reported. In the multi-processor study, it has been felt that such measurements would have no meaning and therefore no CPU or power results were reported.

10.3 Conclusions

This chapter began to explore the applicability of the STC technique developed for single-processor systems in wider embedded architectures. The chapter proposed a set of Scheduler Test Cases (STCs) to help evaluate multi-processor embedded systems when the TTC-SCC scheduler implementations reviewed in Chapter 9 are employed.

The discussions emphasised that the aim with these STCs was to assess the timing behaviour of CAN-based networks implemented using TTC-SCC scheduling protocols. The criteria considered in such an evaluation process included: the levels of jitter in the relative timing of Master and Slave ticks, message latencies between any two communicating nodes, and node-failure detection time.

It is important to note that such criteria were selected as key factors which can somewhat help to assess the predictability of the system as a whole. For example, in S-C networks, tasks in the receiving nodes are triggered by the arrival of messages sent from the Master node. Assessing the jitter levels in the transmission time of such messages allows predicting the time at which tasks in the receivers will execute and whether or not they can meet their timing constraints. Also testing the ability of the network to detect and hence deal with a node failure within a short time bound can help assess the level of predictability in the whole system.

The results obtained from the practical application of the STCs described in this chapter are provided in the next chapter (Chapter 11).

Chapter 11
Assessing the behaviour of TTC-SCC scheduler implementations

11.1 Introduction

As in Chapter 7, this chapter provides the output results from the TTC-SCC implementations (discussed in Chapter 9) when the STCs (described in Chapter 10) are employed. The aim of this chapter is to explore the effectiveness of the extended STC technique for multi-processor systems in assessing (and distinguishing) the behaviour of the various implementation classes for TTC-SCC scheduler.

The chapter also begins by outlining the methodology used to obtain the results presented later in the chapter[29].

11.2 Methodology

11.2.1 Introduction

It is worth highlighting that the key results in the multi-processor study in this thesis are presented using mathematical equations. Such equations were intended to provide a description of the behaviour of each TTC-SCC scheduler considered. Using mathematical approach, as previously noted, was found more meaningful in this particular study than using only empirical approaches (as in the single-processor study).

This section describes the methodology used to obtain both the experimental and theoretical results.

[29] The work described in this chapter has been adapted from the study presented in the author's publication [2] listed in page xvi.

11.2.2 Hardware and software setup

The experimental measurements in this study were conducted using Phytec boards supporting Infineon C167 microcontrollers. The C167 is a 16-bit microcontroller with a 20 MHz crystal oscillator. The C167 board has additional on-chip support for CAN protocol. The network nodes (one Master and three Slaves) were connected using a twisted-pair CAN link. The CAN baudrate used was 1 Mbps, and 8-byte "Tick" messages were used, with one byte reserved for the Slave ID, while the remaining data bytes contained random values (see Section 8.4.3). The tick interval used was 4 ms.

Note that Pont (2001) provided a complete set of codes required to implement the TTC-SCC protocol on 8051 processor hardware. For the 16-bit system considered here, the 8051 design was ported to the C16x family. The Keil C166 compiler was used (Keil Software, 1998).

11.2.3 Jitter tests

To provide an indication of the timing behaviour of each system, two sets of parameters were measured: the first one was corresponding to transmission times between distributed nodes, and the second one was corresponding to the timing jitter.

To present the transmission periods between the Master and Slave nodes, three values were recorded:

- **Worst-case transmission time (WCTT)**: represented by the longest delay between the execution of Task A on the Master node and the execution of Task A on the Slave node.
- **Best-case transmission time (BCTT)**: represented by the shortest delay between the execution of Task A on the Master node and the execution of Task A on the Slave node.
- **Average transmission time (AVTT)**: represented by the average delay between the execution of Task A on the Master node and the execution of Task A on the Slave node.

As mentioned previously, jitter in multi-processor systems can be represented by the variation in time between an event in the Master and its corresponding event in the Slave. To assess the jitter levels, two values were recorded:

- **Difference (absolute) jitter:** obtained by subtracting the best-case (minimum) transmission time from the worst-case (maximum) transmission time obtained from the measurements in the sample set.
- **Average jitter:** represented by the standard deviation in the measure of average message transmission time.

Again, note that there can be many other measures to represent the levels of task jitter, but these measures were felt to be appropriate for this study.

To make transmission delay measurements experimentally, a pin on the Master node was set high (for a short period) at the start of the "`Master_Task_A`". Another pin on the Slave (initially high) was set low at the start of the "`Slave1_Task_A`". The signals obtained from these two pins were then AND-ed (using a 74LS08N chip: Texas Instruments, 1988), to give a pulse stream with widths that represent the transmission delays (Figure 11-1).

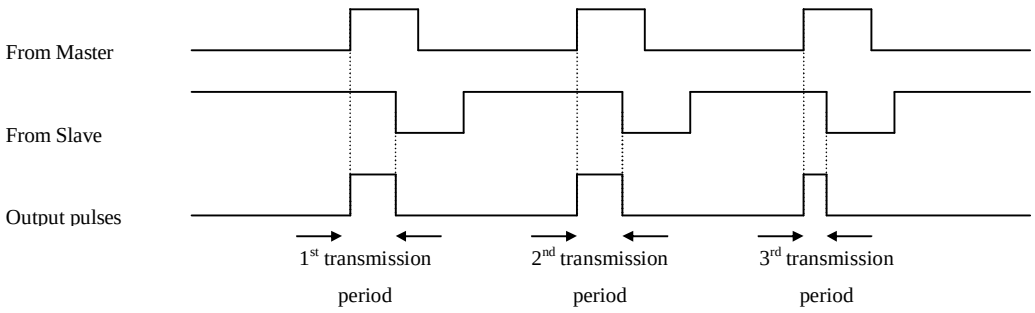


Figure 11-1: The method used to measure the transmission time in TTC-SCC schedulers.

These widths were measured using a National Instruments data acquisition card 'NI PCI-6035E' (National Instruments, 2006), used in conjunction with appropriate software LabVIEW 7.1 (LabVIEW, 2007). In each study, 5000 consecutive pulse widths were measured and recorded: this, again, was found sufficient for the purpose of this study.

11.2.4 Message latency calculations

By going back to the STCs description in Chapter 10, the transmission delays must be calculated between the time at which an activity takes place in one node and the response to this activity in another node. This means that precise results can be obtained if the delays are calculated between the time when data is generated in the sending node and the time when the receiving node begins to handle this data.

Generally, if tasks have long execution durations, the data can be generated at any point in time during the tick interval. In order to simplify the calculations, it has been assumed – throughout this study – that all tasks have reasonably short execution times, thus the data is always generated close to the start of tick interval. Moreover, it has been assumed that the scheduler overhead time on Master and Slaves are very small and can hence be neglected. Based on these assumptions, the message latencies will be calculated between the start of the tick in which data is generated and the tick in which data is received.

11.2.5 Node-failure detection time calculation

In this study, the error mode in the TTC-SCC scheduler has been represented by node failure. Such an error occurs when a Slave node fails to respond to messages sent from the Master or other Slaves in the network.

To assess the behaviour of the scheduler in the event of such an error, it has been decided to calculate the worst-case time the Master processor would take to detect the failure and begin to react to it (see STC E). To obtain worst-case scenario, it was assumed that the Slave fails immediately after it has sent its Ack message to the Master. The worst-case node-failure detection time will hence be calculated between this failure time and the start of the tick in which the Master checks the status of this Slave.

11.2.6 Network utilisation tests

Network (i.e. bandwidth) utilisation in each protocol is also reported. The network utilisation values are represented mathematically as functions of the lengths of the various messages exchanged in the network (assuming 1 Mbps CAN speed) and the scheduler tick interval. Note that network utilisation in each scheduler implementation was presented as the average bandwidth per tick interval.

11.2.7 Memory test

To reflect the scheduler complexity, the CODE and DATA memory values required to implement each of the described scheduling protocol were recorded. The experimental methodology described in Section 7.2.5 to obtain memory requirement results was again used here.

11.3 Results

11.3.1 Applying STC to the TTC-SCC1 scheduler

This section presents the output results from the TTC-SCC1 scheduler.

11.3.1.1 Jitter

This section presents the empirical results obtained from the STC A (jitter test) implemented with the TTC-SCC1 scheduling protocol.

Table 11-1: Task jitter from the TTC-SCC1 scheduler (all values in µs).

	µs
BCTT	162.9
WCTT	173
AVTT	166.3
Diff. Jitter	10.1
Avg. Jitter	1.5

The table shows that the difference and average jitter obtained from the TTC-SCC1 scheduler were 10 µs and 1.5 µs, respectively. This jitter was due to high variation in Tick message lengths: such a variation was caused by the variation in the number of bits stuffed by the CAN hardware in random data bytes. Remember that transmission times here were measured between the Master's task and Slave's task (not between the ticks). However, since Master and Slave schedulers did not vary in time, the jitter observed was due to CAN bit-stuffing only.

The jitter values presented are seen significant (as will be shown later). Remember that the CAN baudrate used in this study was set to its maximum value which is 1 Mbps. If the network is set to run at lower speeds, then such jitter levels would increase, with having more impact on the timing performance.

11.3.1.2 Message latencies

This section presents the results obtained from the STC B, STC C and STC D implemented with the TTC-SCC1 scheduling protocol.

Given that *M* is the Master Tick message length, *T* is the tick interval, *TDMA*1 is the Time Division Multiple Access round and *N* is the number of Slaves, the message latencies between any two nodes in the network are calculated as follows.

STC B: Master-to-Slave message latency

In the best-case scenario, the data to be transmitted from the Master to a Slave at a given tick must be ready at the start of the tick interval and, therefore, it has already been generated in the task(s) executed within the preceding tick interval. The Master will hence be able to send this data with the Tick message due to transmit in the current tick. In contrast, in the worst-case scenario, the Master decides to send data to a given Slave straight after it has sent Tick message to that Slave.

Figure 11-2 shows an example where the Master node wants to communicate with S2 (i.e. Slave number 2). The best-case transmission process is illustrated using the "blue" colour while the worst-case process is illustrated using the "red" colour.

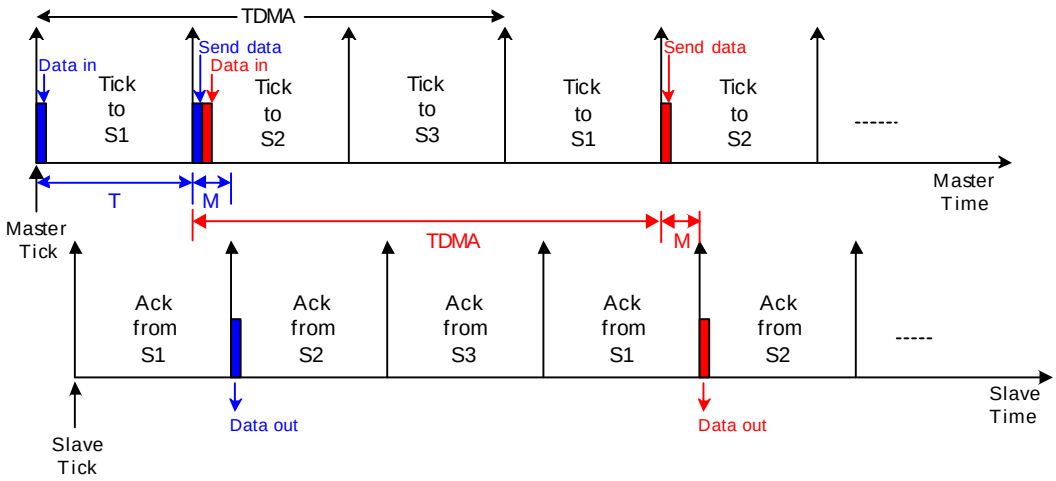


Figure 11-2: Master-to-Slave message latency in TTC-SCC1.

The figure clearly shows that in the best-case scenario, the data – generated in the first tick – can be sent to S2 at the beginning of the second tick (where this tick is allocated

to exchange data with S2). S2 can then extract the data on arrival of the Master Tick message. In the worst-case scenario, the Master needs to wait until the tick allocated for S2 – in the next TDMA round – arrives during which it can send a Tick message with data allocated for S2. Remember that Slave ticks are always delayed by *M*, since Slave scheduler is triggered by the arrival of the Master Tick message.

The equations for the best- and the worst-case message latencies between the Master and a given Slave are presented in the following table. Note that these equations are simply derived from the graphical representation illustrated in Figure 11-2.

Table 11-2: Master-to-Slave latency equations in TTC-SCC1.

	Best-case latency	Worst-case latency
Master-to-Slave latency	$T + M$	$TDMA1 + M$

STC C: Slave-to-Master message latency

Based on the explanation provided for STC B, the best- and the worst-case message transmissions can be derived using Figure 11-3. Note that, in this case, although S2 replies its Ack message to the Master in the second tick interval (straight after receiving the Tick message), the Master will not check the contents of S2 Ack message until the start of the next tick (just before sending data to S3).

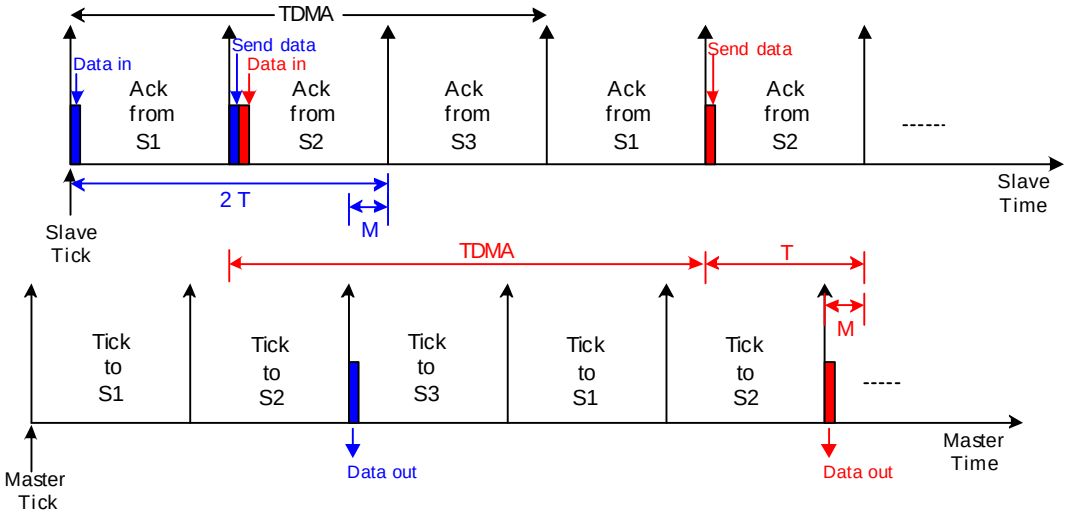


Figure 11-3: Slave-to-Master message latency in TTC-SCC1.

Using the graphical representation illustrated in Figure 11-3, the equations for the best- and the worst-case message latencies between a given Slave and the Master are presented in the following table.

Table 11-3: Slave-to-Master latency equations in TTC-SCC1.

	Best-case latency	Worst-case latency
Slave-to-Master latency	$2T - M$	$TDMA1 + T - M$

STC D: Slave-to-Slave message latency

In the Slave-to-Slave communication, the situation is more complicated. To be able to work out the message latency between any two Slaves in the TTC-SCC1 network, the shortest distance between their corresponding tick intervals (i.e. the tick intervals in which Slaves can send their "Ack" messages) must be calculated.

Given that X is the transmitting Slave and Y is the receiving Slave, the distance D_{XY} between "Ack-X" and "Ack-Y" is calculated as follows:

$$D_{XY} = \big((Y - X)\bmod(N)\big)T$$

Equation 11-1

For example, consider the example shown in Figure 9-1. The distance between Ack-1 and Ack-3, where X = 1 and Y = 3, is calculated as: ((3–1) mod (3)) T = (2 mod (3)) T = 2T.

In contrast, the distance between Ack-3 and Ack-1, where X = 3 and Y = 1, is calculated as: ((1-3) mod (3)) T = (-2 mod (3)) T = T. Note that the message latency between any two communicating Slaves is calculated as a function of D_{XY}. This is further illustrated in Figure 11-4 below.

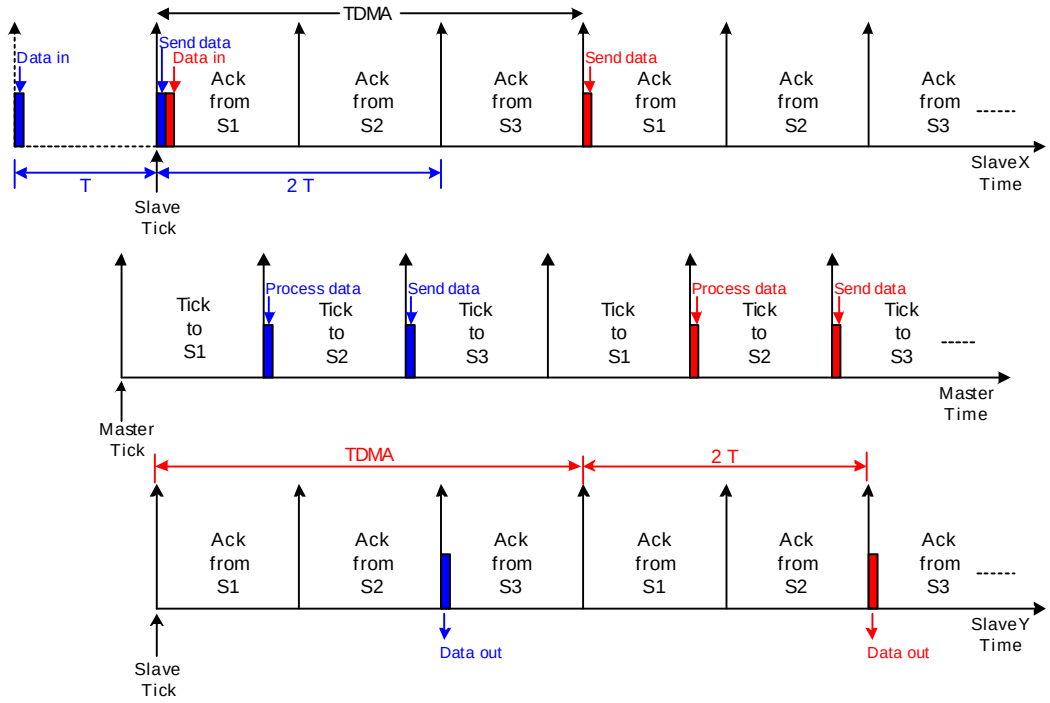


Figure 11-4: Slave-to-Slave message latency in TTC-SCC1.

The figure illustrates the communication process between S1 and S3 in the TTC-SCC1 scheduler. In the best-case scenario (blue colour), S1 sends the data (which was generated in the preceding tick interval) with its Ack-1 message straight after the Master Tick message is received. The Master will check the contents of Ack-1 message in the following tick before it sends a Tick message addresses S2. The data can then be completely processed and placed in the corresponding CAN data registers for transmission with the following Tick message intended for S3. The diagram shows that this process takes time equals to 2*T* (the distance between Ack-1 and Ack-3) plus one additional tick interval.

In the worst-case scenario (red colour), the data is generated in S1 after it has already sent its Ack-1 to the Master. This means that S1 can only send its data after a full TDMA round. This results in increasing the message latency between S1 and S3 to be equal to *TDMA*1 plus the distance between Ack-1 and Ack-3. Note that the process shown in Figure 11-4 presents the communication between any two Slaves when D_{XY} is larger than *T* (i.e. Ack messages for the communicating Slaves are not transmitted in consecutive tick intervals). When D_{XY} is equal to *T*, then the communication process in the described TTC-SCC1 becomes more complicated. This is simply because when the Master receives data from S1 – as an example – it cannot send it immediately to S2

since the data intended for S2 has already been configured and placed in the CAN data registers. This means that the Master always needs to wait for an extra TDMA round so it can complete processing the data received from S1 and configure the Tick data message.

The equations for the best- and the worst-case message latencies between the Master and a give Slave are presented in the following table.

Table 11-4: Slave-to-Slave latency equations in TTC-SCC1.

		Best-case latency	Worst-case latency
Slave-to-Slave latency	$D_{XY} > T$	$D_{XY} + T$	$D_{XY} + TDMA1$
	$D_{XY} = T$	$2\ T + TDMA1$	$T + 2\ TDMA1$

Remember that in TTC-SCC1, $TDMA1 = NT$. By substituting this value in the equations shown, the results can be simplified as follows:

Table 11-5: Slave-to-Slave latency equations in TTC-SCC1 (simplified formula).

		Best-case latency	Worst-case latency
Slave-to-Slave latency	$D_{XY} > T$	$D_{XY} + T$	$D_{XY} + NT$
	$D_{XY} = T$	$(N+2)T$	$(2N+1)$T

Note that when the number of Slaves N significantly increases, the message latencies between the communicating Slaves will also increase by significant factors (except in the best-case scenario when $D_{XY} > T$). This implies that the described TTC-SCC1 may not be the appropriate solution for multi-processor designs with a large number of Slave nodes connected up in the network.

11.3.1.3 Node-failure detection time

This section presents the results obtained from the STC E implemented with the TTC-SCC1 scheduling protocol.

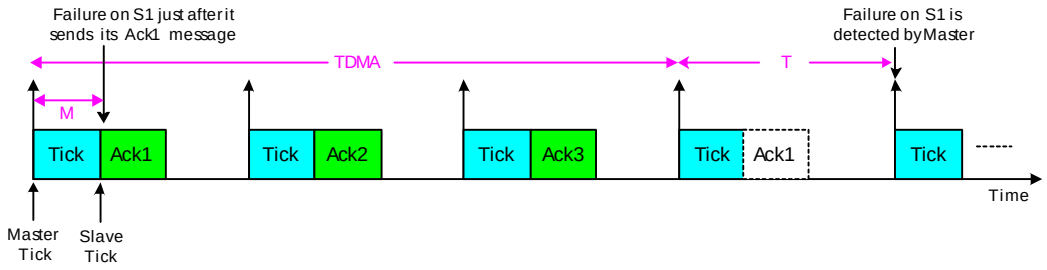


Figure 11-5: Failure detection time in TTC-SCC1.

In TTC-SCC1, the Master node has to wait for a complete TDMA round before the status of all the Slaves can be checked. Using Figure 11-5, the worst-case failure detection time for the TTC-SCC1 scheduler is calculated as:

$$\text{Worst-case failure detection time} = TDMA1 + T - M = (N+1)\,T - M$$

Equation 11-2

In the example shown in Figure 11-5, the Master would take four Tick intervals (i.e. TDMA plus one additional tick) to detect a failure on S1. The situation would become worse if the number of Slave nodes *N* increases.

11.3.1.4 Network utilisation and memory requirements

Assume that all Ack message lengths are equal, and each one is represented by *S*, then the network utilisation in TTC-SCC1 can be calculated as follows:

$$Network\ utilisation = \frac{NM + NS}{TDMA1} = \frac{NM + NS}{NT} = \frac{M + S}{T}$$

Equation 11-3

Remember that the number of Tick messages in each TDMA round was equal to the number of Slaves. If the length of the Tick message is assumed equal to the length of Ack message, then Equation 11-3 can be simplified as:

$$Network\ utilisation = \frac{2M}{T}$$

Equation 11-4

Table 11-6 summarises the memory required to implement the TTC-SCC1 scheduler.

Table 11-6: Memory requirements (ROM and RAM) for the TTC-SCC1 scheduler.

	ROM requirements (Bytes)	RAM requirements (Bytes)
Master	1666	30
Slave	1590	108

11.3.2 Applying STC to the TTC-SCC2 scheduler

This section presents the output results from the TTC-SCC2 scheduler.

11.3.2.1 Jitter

This section presents the empirical results obtained from the STC A (jitter test) implemented with the TTC-SCC2 scheduling protocol.

Table 11-7: Task jitter from the TTC-SCC2 scheduler (all values in µs).

	µs
BCTT	163
WCTT	173.1
AVTT	166
Diff. Jitter	10.1
Avg. Jitter	1.4

The jitter levels in this scheduler implementation are seen similar to those obtained from TTC-SCC1 scheduler. This is again due to CAN bit-stuffing impact on the Tick messages which contained random data set.

11.3.2.2 Message latencies

This section presents the results obtained from the STC B, STC C and STC D implemented with the TTC-SCC2 scheduling protocol.

Calculating message latencies in TTC-SCC2 scheduler is not straightforward. This is because the Master communicates with Slaves in a random way depending on the specification of the system for which the scheduler is used.

To get on with the calculations, it is important to define two parameters:

- The distance between successive ticks allocated for a given Slave: this is referred to as D_{XX}.
- The shortest distance between Ack messages from any two communicating Slaves: this is referred to as D_{XY} (as in TTC-SCC1).

Given that M is the Master Tick message length, T is the tick interval and $TDMA2$ is the Time Division Multiple Access round, the message latencies between any two nodes in the network are calculated as follows.

STC B: Master-to-Slave message latency

In the best-case scenario, the behavior is exactly the same as observed with TTC-SCC1 scheduler. However, in the worst-case scenario, after data is generated in a given tick, the Master needs to wait until the following tick in which it can communicate with the target Slave. This delay does not have to be as long as the TDMA round: instead, it depends on D_{XX}. The value of D_{XX} must lie between T and $TDMA2$. For example, if the Master communicates with the Slave only once in the TDMA round, D_{XX} will be equal to $TDMA2$. In contrast, if the Slave is allocated adjacent tick intervals to transmit its Ack message, then D_{XX} will be equal to T.

Figure 11-6 illustrates the process of Master to Slave 2 communication in the system shown in Figure 9-4. In the worst-case scenario, data – which is generated immediately after the Master sent Tick to S2 – can only be sent to S2 in the next tick allocated for this Slave. In the example shown, this delay is equal to $4T$. For S3, where only one tick in the whole TDMA round is allocated, D_{XX} will be equal to $TDMA2$, and so on.

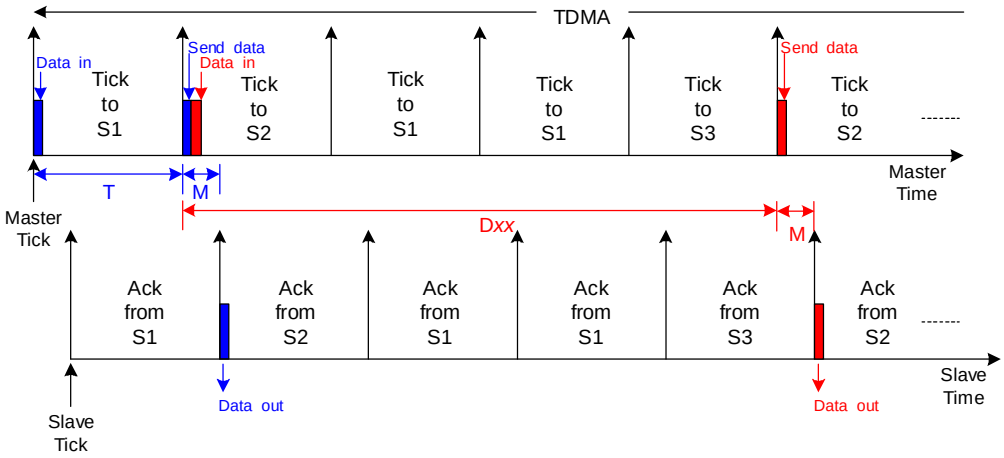


Figure 11-6: Master-to-Slave message latency in TTC-SCC1.

A summary of the results is provided in the following table.

Table 11-8: Master-to-Slave latency equations in TTC-SCC2.

	Best-case latency	Worst-case latency
Master-to-Slave latency	$T + M$	$D_{XX} + M$

Based on the discussion above, the worst-case Master-to-Slave latency will have the minimum value of $T + M$ and the maximum value of $TDMA2 + M$.

STC C: Slave-to-Master message latency

Again, the behavior here is similar to that observed in the TTC-SCC1 scheduler. The only difference – as in Master-to-Slave communication – is that the $TDMA2$ term is replaced by D_{XX} in the equations. A summary of the results is provided in the following table. Remember that D_{XX} can have a value between T and $TDMA$.

Table 11-9: Slave-to-Master latency equations in TTC-SCC2.

	Best-case latency	Worst-case latency
Slave-to-Master latency	$2T - M$	$D_{XX} + T - M$

Similarly, the worst-case Slave-to-Master latency will have the minimum value of $2T - M$ and the maximum value of $TDMA2 + T - M$.

STC D: Slave-to-Slave message latency

The situation here is slightly more complicated. Since the communication between nodes in this scheduler has a random pattern, D_{XY} cannot be calculated as a function of X and Y (as with TTC-SCC1). For example, the distance between the Slave 1 and Slave 3 cannot be calculated as (3-1)T.

In order to present a general formula for Slave-to-Slave message latency, it is important to know the "current" and the "next" distance between the Ack message of the sending Slave and the Ack message of the receiving Slave. The "current" distance is denoted by $D_{Xi\ Yi}$, while the "next" distance is denoted by $D_{X(i+1)\ Y(i+1)}$. For example, consider the communication between S1 and S2 in the system shown in Figure 9-4. For these two Slaves, $D_{Xi\ Yi} = T$ and $D_{X(i+1)\ Y(i+1)} = 3T$. In the same way, considering S1 and S3, $D_{Xi\ Yi}$ = T and $D_{X(i+1)\ Y(i+1)} = 4T$. Note that the "current" distance must be the shortest distance between the two communication Slaves and the "next" distance is the one follows it.

Accordingly, the message latencies between any two Slaves depend of both the "current" and "next" distances. In the best-case scenario, when data is generated in the previous tick to the current one, the message latency between S1 and S2 can be calculated as follows.

Table 11-10: Slave-to-Slave latency equations in TTC-SCC2 (best-case scenario).

		Best-case scenario
Slave-to-Slave latency	$D_{Xi\ Yi} > T$	$D_{Xi\ Yi} + T$
	$D_{Xi\ Yi} = T$	$D_{Xi\ Y(i+1)} + T$

Please note that $D_{Xi\ Yi}$ denotes the distance between the Ack message of the sender and the consecutive Ack message of the receiver, while $D_{Xi\ Y(i+1)}$ is the distance between the Ack message of the sender and the one after the next Ack message of the receiver.

Likewise, in the worst-case scenario, when data is generated in the current tick after Ack message is sent, the message latency between S1 and S2 can be calculated as follows.

Table 11-11: Slave-to-Slave latency equations in TTC-SCC2 (worst-case scenario).

		Worst-case scenario
Slave-to-Slave latency	$D_{X(i+1)\ Y(i+1)} > T$	$D_{Xi\ Y(i+1)}$
	$D_{X(i+1)\ Y(i+1)} = T$	$D_{Xi\ Y(i+2)}$

Please note that the best-case scenario here does not mean the shortest message latency, and the worst-case scenario does not mean longest message latency.

The figure shows the message latency between S1 and S2 in the example provided in Figure 9-4, where $D_{Xi\ Yi} = T$ and $D_{X(i+1)\ Y(i+1)} > T$. The figure shows that the best-case scenario produced longer message latency than in the worst-case scenario.

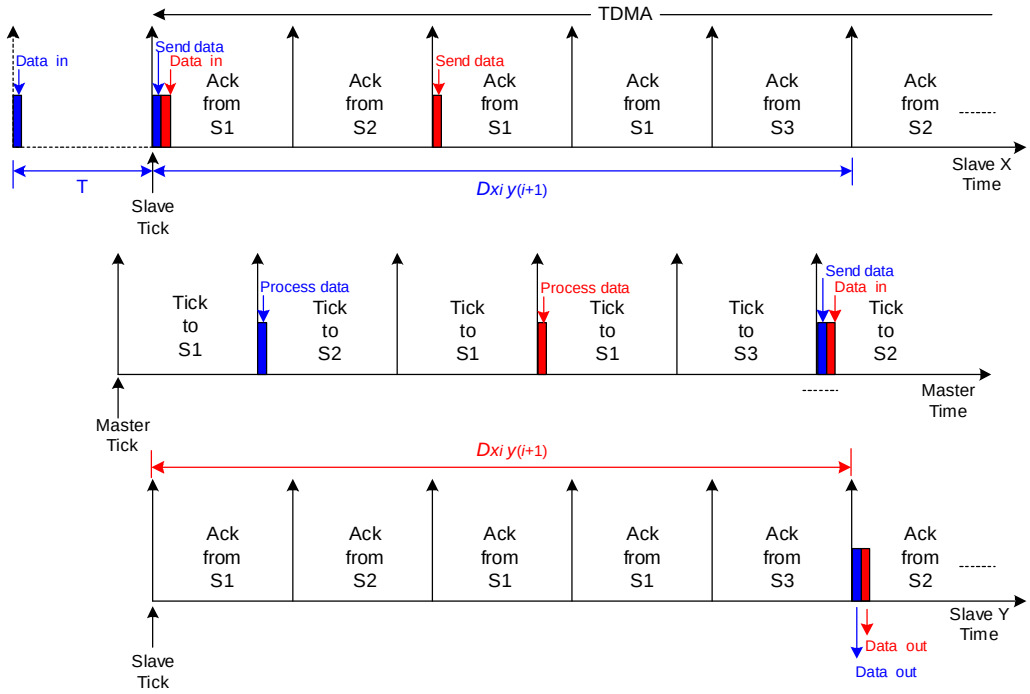


Figure 11-7: Slave-to-Slave message latency in TTC-SCC2.

11.3.2.3 Node-failure detection time

This section presents the results obtained from the STC E implemented with the TTC-SCC2 scheduling protocol.

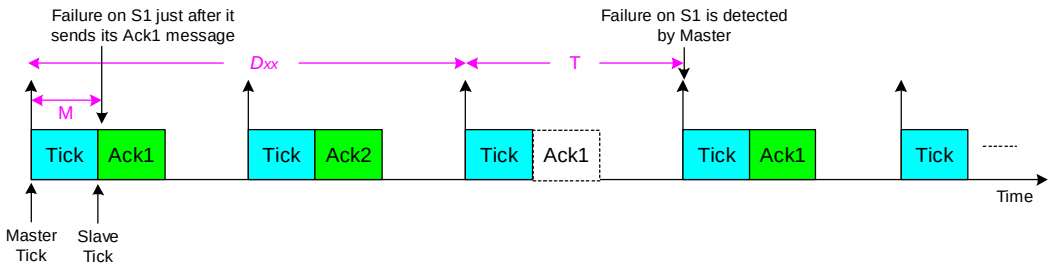


Figure 11-8: Failure detection time in TTC-SCC2.

In TTC-SCC2, the Master node has to wait until the status of the Slave is next checked. Using Figure 11-8, the worst-case failure detection time for the TTC-SCC2 scheduler is calculated as:

$$\text{Worst-case failure detection time} = D_{XX} + T - M$$

Equation 11-5

In the example shown in Figure 11-8, the Master would take approximately three Tick intervals to detect a failure on S1. Failure detection time for a given node in TTC-SCC2 scheduler would depend on the number of Slaves in the network, length of the TDMA round, and the number of ticks – within the TDMA round – used to communicate with that Slave. In some case, where (for example) *TDMA*2 is very long and the Slave is only checked once per TDMA round, detecting failure in such a Slave can be too long. This can have an important impact on the predictability of many networks.

11.3.2.4 Network utilisation and memory requirements

Assume that all Ack message lengths are equal and, where any Ack message is represented by *S*, *k* is the total number of ticks in the TDMA round, *M* is the Master Tick message length, then the network utilisation in TTC-SCC2 can be calculated as follows:

$$Network\ utilisation = \frac{k(M+S)}{kT} = \frac{M+S}{T}$$

Equation 11-6

If the length of the Tick message is assumed equal to the length of Ack message, then Equation 11-6 can be simplified as:

$$Network\ utilisation = \frac{2M}{T}$$

Equation 11-7

Note that the network utilisation here is exactly similar to that with TTC-SCC1 scheduler. This is because, although the TDMA round is configured differently, each tick interval can still not handle more than two messages: Master "Tick" and Slave "Ack".

Table 11-12 summarises the memory required to implement the TTC-SCC2 scheduler.

Table 11-12: Memory requirements (ROM and RAM) for the TTC-SCC2 scheduler.

	ROM requirements (Bytes)	RAM requirements (Bytes)
Master	1710	31
Slave	1590	108

11.3.3 Applying STC to the TTC-SCC3 scheduler

This section presents the output results from the TTC-SCC3 scheduler.

11.3.3.1 Jitter

This section presents the empirical results obtained from the STC A (jitter test) implemented with the TTC-SCC3 scheduling protocol.

Table 11-13: Task jitter from the TTC-SCC3 scheduler (all values in µs).

	µs
BCTT	162.9
WCTT	172.9
AVTT	166.2
Diff. Jitter	10
Avg. Jitter	1.5

The jitter levels in this scheduler implementation are seen similar to those obtained from the previous two schedulers. This is again due to CAN bit-stuffing impact on the Tick messages which contained random data set.

11.3.3.2 Message latencies

This section presents the results obtained from the STC B, STC C and STC D implemented with the TTC-SCC3 scheduling protocol.

Given that M is the Master Tick message length, T is the tick interval, $TDMA3$ is the Time Division Multiple Access round, N is the number of Slaves, and m is the maximum number of Slaves replying per tick interval, the message latencies between any two nodes in the network are calculated as follows.

STC B: Master-to-Slave message latency

Basically, the results obtained here are similar to those obtained from the TTC-SCC1 and TTC-SCC2. However, since the TDMA round is shorter in the TTC-SCC3, this results is a reduced message latency. A summary of the results is provided in the following table.

Table 11-14: Master-to-Slave latency equations in TTC-SCC3.

	Best-case latency	Worst-case latency
Master-to-Slave latency	$T + M$	$TDMA3 + M$

Please note that in the example given in Figure 9-5 (where $TDMA3 = T$), the Master-to-Slave latency is fixed and always equal to $T + M$.

STC C: Slave-to-Master message latency

Again, the equations for the Slave-to-Master message latencies are similar to those derived before (in TTC-SCC1 and TTC-SCC2). But, again, the message latencies are expected to be much shorter in the TTC-SCC3 due to the shorter TDMA round. A summary of the results is provided in the following table.

Table 11-15: Slave-to-Master latency equations in TTC-SCC3.

	Best-case latency	Worst-case latency
Slave-to-Master latency	$2T - M$	$TDMA3 + T - M$

Also note here that in the example given in Figure 9-5 (where *TDMA*3 = *T*), the Slave-to-Master latency is fixed and always equal to *2T – M*.

STC D: Slave-to-Slave message latency

STC D demonstrates a substantial difference between the behaviour of TTC-SCC3 scheduler and the previous schedulers. Since all Slaves are configured to receive Ack messages sent from other Slaves, Slave to Slave message latency is substantially reduced. This is further illustrated in Figure 11-9.

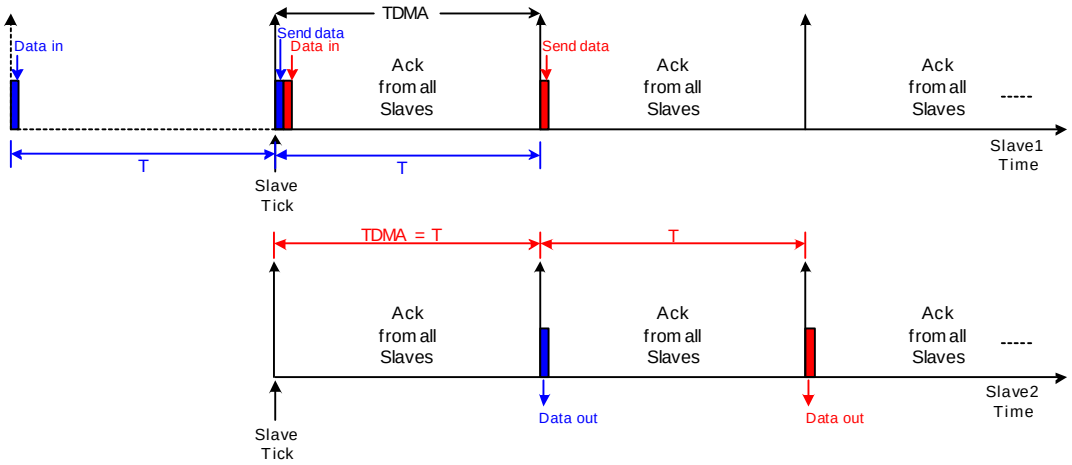


Figure 11-9: Slave-to-Slave message latency in TTC-SCC3.

Assume that S1 wants to send data to S2. In the TTC-SCC3 described, Slave-to-Slave communication can be made directly without going through the Master. More clearly, in the best-case scenario, data on S1 must be ready to transmit at the start of the tick (i.e. data has been generated in the previous tick interval). The data will then be sent out in the Ack-1 message to all nodes. At the beginning of the following tick, S2 (for which the data is intended) will check the contents of Ack-1 message and hence extract the requested data for use in that tick. In the worst-case scenario, where S1 decides to send the data straight after transmitting Ack-1 message, it has to wait for a full TDMA round (which is equal to *T* in the simple implementation shown in the figure) before which it can send the data out with the next Ack-1 message to all Slaves. Once S2 receives the Ack-1 message, the scheduler on S2 needs only one tick to process the Ack-1 message and hence extract the requested data.

Please note that the latencies between Slaves are almost same as the latencies between a given Slave and the Master. These results are summarised in the following table.

Table 11-16: Slave-to-Slave latency equations in TTC-SCC3.

	Best-case latency	Worst-case latency
Slave-to-Slave latency	2*T*	*TDMA*3 + *T*

Again note that in the example given in Figure 9-5 (where *TDMA*3 = *T*), the Slave-to-Slave latency is fixed and always equal to 2*T*.

Please note that a large value of *m* (the number of Slaves replying per tick) would require that the tick interval should be extended to accommodate *m* Ack messages sent from *m* Slaves. This increase in the scheduler tick interval may not be appropriate for some applications where tick interval has to be extremely short. This means that it is always a trade-off between message latencies and tick interval.

11.3.3.3 Node-failure detection time

This section presents the results obtained from the STC E implemented with the TTC-SCC3 scheduling protocol.

The TTC-SCC3 allows the Master node to quickly receive Ack messages from the Slaves. For example, Figure 11-10 illustrates an example where Slave1 suffers a failure as soon as it has sent its Ack message. It is assumed here that the TDMA round is extended across two tick intervals. As a result, the longest possible time for the Master node to detect a failure on the S1 node is calculated as follows.

$$\text{Worst-case failure detection time} = TDMA3 + T - M = (N/m + 1)\,T - M$$

Equation 11-8

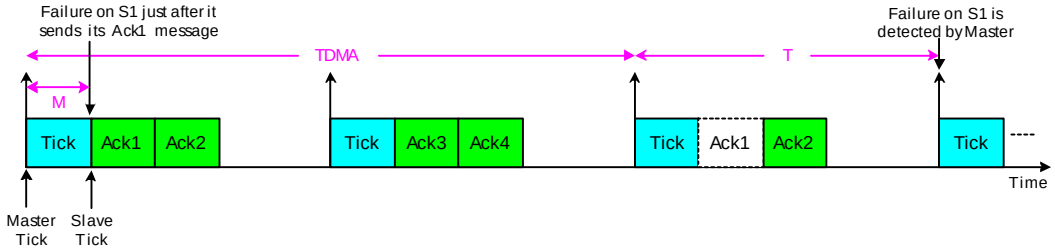


Figure 11-10: Failure detection time in TTC-SCC3.

Remember that TDMA here equals to NT / m. When all Slaves are allowed to reply in one tick (i.e. $N = m$), then the worst-case failure detection time becomes equal to $2T - M$. This duration is slightly less than two Tick intervals (which is significantly less than corresponding time in TTC-SCC1 and TTC-SCC2 for non-trivial networks).

11.3.3.4 Network utilisation and memory requirements

Again, assume that S is the length of any Ack message, then the network utilisation in TTC-SCC3 can be calculated as follows:

$$Network\ utilisation = \frac{\left(\frac{N}{m}\right)M + NS}{TDMA3} = \frac{\left(\frac{N}{m}\right)M + NS}{\left(\frac{NT}{m}\right)} = \frac{M + mS}{T}$$

Equation 11-9

Remember that the number of Tick messages in each TDMA round was equal to the number of tick intervals (which is equal to N/m). If the length of the Tick message is assumed equal to the length of Ack message, then Equation 11-9 can be simplified as:

$$Network\ utilisation = \frac{(1+m)M}{T}$$

Equation 11-10

Table 11-17 summarises the memory required to implement the TTC-SCC3 scheduler.

Table 11-17: Memory requirements (ROM and RAM) for the TTC-SCC3 scheduler.

	ROM requirements (Bytes)	RAM requirements (Bytes)
Master	1838	33
Slave	1722	116

11.3.4 Applying STC to the TTC-SCC4 scheduler

This section presents the output results from the TTC-SCC4 scheduler.

11.3.4.1 Jitter

This section presents the empirical results obtained from the STC A (jitter test) implemented with the TTC-SCC4 scheduling protocol.

Table 11-18: Task jitter from the TTC-SCC4 scheduler (all values in µs).

	µs
BCTT	99.9
WCTT	102
AVTT	101
Diff. Jitter	2.1
Avg. Jitter	0.6

By configuring the Master Tick messages to be empty of data, the results show that the impact of CAN bit stuffing has been significantly reduced. Note that – apart from removing jitter in the data field – all bits in the control fields of CAN messages have been selected with care so that they do not themselves introduce any variation in the number of bit stuffing. Such a bit-value selection approach caused the bit-stuffing jitter in CAN messages to be removed almost completely. However, the residual 2 µs jitter (which is equal to 2 bit times for the CAN bus at 1 Mbps) has been found to be likely generated by a clock-drift between the CAN controller and the microcontroller CPU: this is further discussed in Appendix F.

11.3.4.2 Message latencies

This section presents the results obtained from the STC B, STC C and STC D implemented with the TTC-SCC4 scheduling protocol.

Overall, since the TTC-SCC4 scheduler was developed only to deal with jitter problem in the TTC-SCC3 scheduler, the results obtained from the STCs A, B and C implemented with TTC-SCC4 are expected to be similar to those presented in Section 11.3.3.2. Remember that the TDMA round in TTC-SCC4 is equal to $(N+1)\ T/m$, where N is the original number of Slaves.

However, STC B, C and D assume that Master and Slave should exchange "real" data between them, therefore the equations for Master-to-Slave and Slave-to-Master shown for the TTC-SCC3 do not work here anymore. This is again because the Master node in the TTC-SCC4 scheduler cannot send data to (or respond to data from) Slave nodes. In order to assess the scheduler behavior when the STCs A, B and C are applied to TTC-SCC4 (in the same way as with the previous schedulers), it must be assumed here that the additional Slave node will completely replace the original Master node in processing data, but will still not be superior to other Slaves. Therefore, Master-to-Slave and Slave-to-Master message latencies will be identical to Slave-to-Slave message latencies (in the context discussed here).

STC B: Master-to-Slave message latency

Same as Slave-to-Slave message latency in TTC-SCC3 (see Section 11.3.3.2 c)

STC C: Slave-to-Master message latency

Same as Slave-to-Slave message latency in TTC-SCC3 (see Section 11.3.3.2 c)

STC D: Slave-to-Slave message latency

Same as Slave-to-Slave message latency in TTC-SCC3 (see Section 11.3.3.2 c)

11.3.4.3 Node-failure detection time

This section presents the results obtained from the STC E implemented with the TTC-SCC4 scheduling protocol.

The results here are very similar to those obtained from the TTC-SCC3 scheduler. The only difference is that the Tick message here is extremely short, therefore the worst-

case failure detection time for S1 in the example shown in Figure 11-10 is calculated as follows.

$$\text{Worst-case failure detection time} = TDMA4 + T - M_T = ((N+1)/m + 1)\,T - M_T$$

Equation 11-11

Where N is the original number of Slaves and M_T is the Master Tick message length: this is in order to distinguish it from the ordinary Tick message which contains data in its data field.

11.3.4.4 Network utilisation and memory requirements

Again, assume that S is the length of any Ack message, and M_T is the length of the Master Tick message, then the network utilisation in TTC-SCC4 can be calculated as follows:

$$Network\ utilisation = \frac{\left(\frac{N+1}{m}\right)M_T + (N+1)S}{\left(\frac{N+1}{m}\right)T} = \frac{M_T + mS}{T}$$

Equation 11-12

Remember that the Tick message is sent from a dedicated Master node, and the number of Slaves has increased by one: this is where the term (N+1) comes from. The length of the Master Tick message is assumed shorter than the length of Ack message since it contains no data, so S cannot be substituted by M_T in the equation.

Table 11-19 summarises the memory required to implement the TTC-SCC4 scheduler.

Table 11-19: Memory requirements (ROM and RAM) for the TTC-SCC4 scheduler.

	ROM requirements (Bytes)	RAM requirements (Bytes)
Master	1768	32
Slave	1722	116

11.3.5 Applying STC to the TTC-SCC5 scheduler

This section presents the output results from the TTC-SCC5 scheduler.

11.3.5.1 Jitter

This section presents the empirical results obtained from the STC A (jitter test) implemented with the TTC-SCC5 scheduling protocol.

Table 11-20: Task jitter from the TTC-SCC1 scheduler (all values in µs).

	µs
BCTT	100
WCTT	102.2
AVTT	101.1
Diff. Jitter	2.2
Avg. Jitter	0.6

Jitter results obtained from this scheduler were exactly similar to those obtained from the TTC-SCC4 scheduler. This is obviously due to the same configuration used for the Master Tick messages (i.e. no data-bits were sent and control-bits were selected carefully, thus no CAN bit-stuffing was required).

11.3.5.2 Message latencies

This section presents the results obtained from the STC B, STC C and STC D implemented with the TTC-SCC5 scheduling protocol.

Given that M_T is the Master Tick message length, M_D is the Master Data message length, T is the tick interval, $TDMA5$ is the Time Division Multiple Access round, N is the number of Slaves, and m is the maximum number of Slaves replying per tick interval, the message latencies between any two nodes in the network are calculated as follows.

STC B: Master-to-Slave message latency

Using TTC-SCC5 protocol, the Master-to-Slave communication process will slightly be different than that achieved with the TTC-SCC3 and TTC-SCC4. This process is described here.

By considering the best-case scenario, the data sent from the Master to Slaves is assumed to be generated in the previous tick. However, this data will be sent with the Master Data message (not with the Tick message). The recipient Slaves will, therefore, process the received data in the tick following the tick in which the Master Data message is received (in the same way the Ack messages are treated by the Master and by other Slaves: see Figure 11-9). As a result, the best-case transmission time between the Master and any Slave will be one tick longer than that achieved with the previous S-C protocols.

Since the Master node is allowed to transmit its Data message every tick, it does not have to wait for a full TDMA round before it sends its data in the worst-case scenario: instead, it is able to send its data in the tick following the tick in which the data is generated. Therefore, the best- and the worst-case transmission latencies will always be identical in this scheduler. Please remember that the Slave clocks are always delayed – according to the Master clock – by the value of M_T which represents the length of the Master "empty" Tick message. This is why this term appears in the equations. A summary of the results is provided in the following table.

Table 11-21: Master-to-Slave latency equations in TTC-SCC5.

	Best-case latency	Worst-case latency
Master-to-Slave latency	$2T + M_T$	$2T + M_T$

The results, in the table, show that the Master-to-Slave message latency in this scheduler is always fixed and equal to $2T + M_T$.

STC C: Slave-to-Master message latency

The Slave-to-Master communication process is identical to that achieved when the TTC-SCC3 or TTC-SCC4 is used. Again, remember that the Slave clocks are always delayed by the value of M_T. A summary of the results is provided in the following table.

Table 11-22: Slave-to-Master latency equations in TTC-SCC5.

	Best-case latency	Worst-case latency
Slave-to-Master latency	$2T - M_T$	$TDMA5 + T - M_T$

STC D: Slave-to-Slave message latency

The latencies between the Slaves in this scheduler are similar to those obtained from the TTC-SCC3 and the TTC-SCC4. A summary of the results is provided in the following table.

Table 11-23: Slave-to-Slave latency equations in TTC-SCC5.

	Best-case latency	Worst-case latency
Slave-to-Slave latency	$2T$	$TDMA5 + T$

11.3.5.3 Node-failure detection time

This section presents the results obtained from the STC E implemented with the TTC-SCC5 scheduling protocol.

Figure 11-11 illustrates an example where S1 suffers a failure as soon as it has sent its Ack message. If the TDMA round is extended across two tick intervals, the longest possible time that the Master node takes to detect a failure on the Slave node is calculated as follows.

$$\text{Worst-case failure detection time} = TDMA5 + T - M_T - M_D = (N/m + 1)\, T - M_T - M_D$$

Equation 11-13

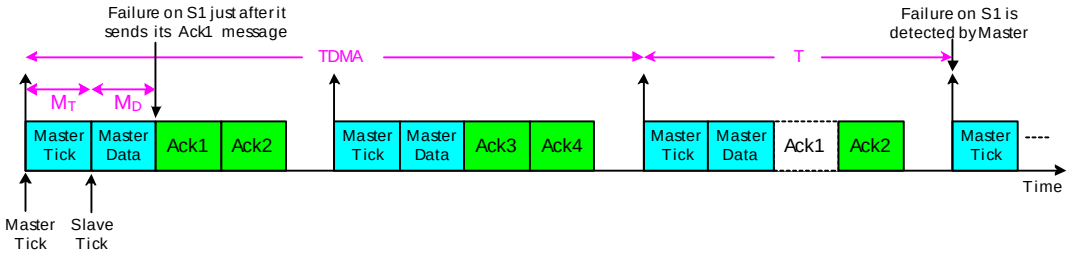


Figure 11-11: Failure detection time in TTC-SCC5.

As discussed in Section 11.3.3.3, when all Slaves are allowed to reply in one tick (i.e. $N = m$), then the worst-case failure detection time becomes equal to $2T - M_T - M_D$.

11.3.5.4 Network utilisation and memory requirements

Again, assume that S is the length of any Ack message, then the network utilisation in TTC-SCC5 can be calculated as follows:

$$Network\ utilisation = \frac{\left(\frac{N}{m}\right)M_T + \left(\frac{N}{m}\right)M_D + NS}{\left(\frac{NT}{m}\right)} = \frac{M_T + M_D + mS}{T}$$

Equation 11-14

Remember that in each tick, the Master sends Tick message and Data message. The Master Tick message is assumed shorter that the Data message, since it contains no data. If the length of the Master Data message is assumed equal to the length of Ack message, then Equation 11-14 can be simplified as:

$$Network\ utilisation = \frac{M_T + (1+m)M_D}{T}$$

Equation 11-15

Table 11-24 summarises the memory required to implement the TTC-SCC5 scheduler.

Table 11-24: Memory requirements (ROM and RAM) for the TTC-SCC5 scheduler.

	ROM requirements (Bytes)	RAM requirements (Bytes)
Master	1884	34
Slave	1722	116

11.4 Summary of the results

This section summarises the results detailed in the previous sections. The section begins by summarising and comparing the results which have been obtained empirically. The jitter column presents the Difference jitter between Master and any Slave in the network.

Table 11-25: Summary of the empirical results from all TTC-SCC schedulers.

		Memory overhead			
Scheduler name	Jitter (µs)	Master		Slave	
		ROM (Bytes)	RAM (Bytes)	ROM (Byte)	RAM (Byte)
TTC-SCC1	10.1	1666	30	1590	108
TTC-SCC2	10.1	1710	31	1590	108
TTC-SCC3	10	1838	33	1722	116
TTC-SCC4	2.1	1768	32	1722	116
TTC-SCC5	2.2	1884	34	1760	118

It is clear from the results that TTC-SCC4 and TTC-SCC5 – where Tick messages transmitted from the Master had fixed lengths – jitter was reduced by approximately 80% when compared to the TTC-SCC1, TTC-SCC2 and TTC-SCC3 schedulers. Again, jitter is an important factor which reveals the predictability level of a system. For memory requirements, it is clear that Slaves required the same memory overheads in TTC-SCC1 and TTC-SCC2, and in TTC-SCC3 and TTC-SCC4. This is because the Slave codes are identical in these cases. In the Master, it can be seen that the memory overheads increased as the scheduler incorporated more features. For example, TTC-SCC5 scheduler required the largest amount of memory overheads to be implemented on the used hardware. However, such increases in memory requirements can still be seen very small (i.e. approx 12% in the ROM and RAM as compared to the basic TTC-SCC1 scheduler).

To provide a practical comparison between the communication behaviour in the various schedulers considered, a small case study is used. The case study is based on the system described in Section 11.2.2. In this system, three Slave nodes are connected up in the network, CAN baudrate is 1 Mbps and the tick interval is 4 ms. Assuming "standard"CAN messages (i.e. 11-bit identifier), "Tick" and "Ack" messages send seven "random" data bytes along with the Slave / Group ID byte (except in the Tick-only message which has no data), then the value of M, M_D and S are equal to 135 µs (with the worst-case level of bit-stuffing) and the value for M_T is equal to 47 µs (without data bytes and any bit-stuffing). The TTC-SCC schedulers used with this small network has the following configurations:

Table 11-26: TTC-SCC models used in the case study to allow a comparison between schedulers.

Scheduler name	Model	TDMA (µs)	Comments
TTC-SCC1	Figure 9-1	12	TDMA round consists of three ticks.
TTC-SCC2	Figure 9-2	16	TDMA round consists of four ticks. S1 is allocated two ticks to send its Ack message, while S2 and S3 only send their Ack once.
TTC-SCC3	Figure 9-5	4	TDMA = T, m = 3. All Slaves send their Ack in the same tick
TTC-SCC4	Figure 9-7	4	TDMA = T, m = 4. The number of Slaves increased by one. Tick message is very short compared to Slaves Ack messages.
TTC-SCC5	Figure 9-8	4	TDMA = T, m = 3. Tick message is also very short compared to Master Data and Slaves Ack messages.

The results obtained from this case study are summarised in the following table. Note that the following abbreviations are used: M-S (Master-to-Slave), S-M (Slave-to-Master), S-S (Slave-to-Slave), NFDT (Node-failure detection time), NU (Network utilisation), BC (Best-case) and WC (Worst-case).

Table 11-27: Results from the case study used to compare between TTC-SCC schedulers.

Scheduler name	M-S1 Latencies (µs)		S1-M Latencies (µs)		S1-S2 Latencies (µs)		NFDT (S1) (µs)	NU (%)
	BC	WC	BC	WC	BC	WC		
TTC-SCC1	4.135	12.135	7.865	15.865	20	28	15.865	6.75%
TTC-SCC2	4.135	8.135	7.865	11.865	20	24	11.865	6.75%
TTC-SCC3	4.135	4.135	7.865	7.865	8	8	7.865	13.5%
TTC-SCC4	8	8	8	8	8	8	7.953	14.675%
TTC-SCC5	8.047	8.047	7.953	7.953	8	8	7.818	14.675%

The results in the table clearly show how the STCs helped to distinguish the behaviour of the various TTC-SCC scheduler implementations evaluated in this study. In more details, it is clear that the TTC-SCC1 – although very simple and efficient – can produce long delays in communication between nodes, especially when the worst-case scenario is considered, and needs comparatively long time to detect a possible failure in a Slave node. The TTC-SCC2 provides an improvement to these parameters while maintaining high network efficiency. The simple case study used here evaluated the communication latencies between S1 (which is more frequently checked) and S2 which is checked once in the TDMA round. If both Slaves are checked only once and the TDMA round increases, the message latencies would be expected to increase correspondingly, with the result that TTC-SCC2 may not be a good alternative to TTC-SCC1 for some systems.

Moving on to the next schedulers, it is clear from the results that in TTC-SCC3, TTC-SCC4 and TTC-SCC5 – where all Slaves are permitted to transmit their Ack messages simultaneously – the message latencies and failure detection time have been reduced significantly. Of course, bandwidth utilisation has increased and would likely increase more depending on the number of Ack messages allowed to transmit per tick interval. This can be a major drawback in applications requiring small tick intervals.

Comparing TTC-SCC4 and TTC-SCC5, the results look almost the same. Remember that TTC-SCC5 was built on TTC-SCC4 and aimed to provide the same level of performance at lower cost. When comparing TTC-SCC5 with TTC-SCC3, Master-to-Slave latencies are shorter (almost by half) in the TTC-SCC3 and the network utilisation

is slightly less. Apart from those, the performance is similar. Remember that in TTC-SCC3, jitter levels are quite high as compared to those obtained from the TTC-SCC5.

11.5 Conclusions

This chapter began by describing the methodology used to obtain the results from multi-processor TTC-SCC implementations considered in this thesis.

The chapter then provided the output results from the application of the STC technique, detailed in Chapter 10, to the TTC-SCC scheduler implementations described in Chapter 9. Again, the results suggested that there is no prefect implementation which can fit all applications. However, according to the features concerned with in this thesis for multi-processor embedded designs, it can be concluded that the TTC-SCC5 scheduler – developed in this project – can be an attractive solution for a wide range of applications due to its low-jitter characteristics, short message latencies and node-failure detection time, and low resource requirements.

Overall, the results presented in this chapter proved – practically – that the use of STC technique is not limited to simple architectures. Instead, it can be easily adapted to evaluate the implementations of scheduling systems with more complex software architectures such as the S-C scheduling protocols considered in this study.

PART E:

DISCUSSION AND CONCLUSIONS

Chapter 12

Discussion

12.1 Introduction

Before concluding this thesis, this chapter discusses the work presented in the previous chapters and highlights the key findings of this project. The studies detailed in this thesis were divided mainly into four parts. Following the thesis introduction, the second part was literature review which consisted of three chapters – according to the covered topics. The topics discussed were real-time scheduling algorithms, scheduler implementations and techniques for linking such two system representations in a systematic way.

Having identified the gaps in previous work in such areas, the third part began to address these gaps for single-processor embedded systems. The fourth and final part then considered the work carried out on multi-processor embedded systems. The same layout is followed in this discussion chapter.

12.2 Literature review

12.2.1 Scheduling algorithms

The literature review began by providing essential background material that is necessary to understand the context of the work presented in the thesis. This material included definitions and classifications of the following items: tasks, timing constraints, jitter, software architectures, schedulers, schedule designs and scheduling algorithms.

It was emphasised that any real-time scheduler incorporates, at its heart, a scheduling algorithm which has a major responsibility of managing the operation of tasks during the system run-time. Since it is responsible of satisfying the timing constraints of tasks, the scheduling algorithm was recognised as the key element that influences system predictability.

The discussion provided a detailed comparison between time-triggered and event-triggered software architectures, and co-operative and pre-emptive schedulers. Based on advantages and disadvantages of these different schemes, it was concluded that systems which employ a combination of time-triggered architectures and co-operative schedulers can have a highly-predictable patterns of behaviour compared with other architectures. Therefore, over many available scheduling algorithms, Time-Triggered Co-operative (TTC) schedulers were found to be a good match for a wide range of applications, in which predictability is a key primary concern, such as safety-related embedded systems.

Problems which might degrade the predictability of TTC schedulers have also been outlined. These mainly included jitter and task overruns. Sources and possible solutions to these problems have been discussed in brief, with a particular focus on the impact that such problems can have on system predictability.

12.2.2 Scheduler implementations

For any project, once the scheduling algorithm has been decided and the task-schedule designed, the next step to take place in the system development process is to implement the scheduler using hardware and software resources. In this project, the target hardware, on which schedulers have been implemented, was based on small (low-cost) COTS embedded microcontrollers. The main focus was therefore on the process of implementing the scheduler software on such hardware platforms using ‘C’ language, being the most suitable language for programming real-time and embedded systems.

Discussions began by emphasising that there can always be a ‘one-to-many’ mapping between any scheduling algorithm and its practical (software) implementations. Evidence was provided that this is true in any scheduling algorithm. As noted, one source of multiple implementations for a give scheduler is the use of “software design patterns” (to create the scheduler code) which themselves can have a vast range of possible implementation options. The discussions indicated that the key component which – in practical use – controls the operational behaviour of an embedded system incorporating a scheduler is the software implementation (typically represented by the scheduler source code). It was also made clear that any – even small and by no means significant – changes in the implementation decisions can have a profound impact on

the overall system behaviour, therefore special consideration must be given to the way the scheduler code is implemented.

Previous work on scheduler implementations using Ada and C languages was reviewed using a substantial number of example studies. The chapter then focused on previous work on TTC scheduler implementations. The key early and recently (by the ESL research group) work in this area was reviewed in detail. It was shown how researchers in the ESL group had a great deal of interest in developing embedded systems which are based on TTC architectures. This was reflected by the number of projects carried out in this particular area.

Discussions about scheduler implementations concluded by pointing out the key limitations experienced in this research area. These can be summarised as follows:

- The process of translating between scheduling algorithms and scheduler implementations, and the impact of using particular implementation decisions on the actual run-time behaviour of embedded systems, have not been considered in detail.
- Despite the useful work on TTC algorithms which is still ongoing, the various TTC implementations developed so far in the ESL group have not been documented, evaluated and linked using a systematic method. In each of the previous projects, only one or two features – as to be addressed by the particular scheduler – were assessed without considering other important features that might affect the user's decision when choosing between the various schedulers.

12.2.3 Linking scheduling algorithms and scheduler implementations

Having discussed the relationship between scheduling algorithms and scheduler implementations, the thesis moved further to review previous work on software evaluation techniques as a practical means for linking these two system representations. The aim of this process was to find ways which help ensure that the features specified at the design stage of a scheduling algorithm are not lost during the scheduler implementation stage.

In general, two main approaches were recognised as ways for evaluating software systems, validation and verification. A clear distinction between these two terms was provided by gathering and comparing numerous definitions from a range of well-recognised dictionaries and reference books. It was concluded that validation is mainly used to ensure that the final software product complies with user's requirements, where such a confidence cannot be achieved unless a verification – which checks the conformance of each stage with its previous one – is applied consistently during the whole development process. As indicated, software verification can be achieved through static techniques, such as software inspections and formal methods, and/or dynamic techniques, such as software testing.

The discussions provided, in detail, the advantages and disadvantages of the three verification techniques outlined. In summary, despite the benefit of using software inspections and formal methods – especially at the early stages in the development process, software testing was recognised as the most effective way to provide a confidence that the implemented software – in many applications – will behave exactly as the user intended. To ensure more efficiency in testing real-time embedded software and avoid the necessity for static verification techniques, automated code generation was suggested as a way to verify that the software – before testing – is error-free and precisely reflects the specifications defined at the design phase of a project.

The limitations in previous work in the area of testing and test cases were then highlighted. These can be summarised as follows:

- Previous work on software testing mainly considered techniques for generating test cases that check the functionality of the software, or its quality attributes.
- Previous work on real-time software testing was based on modelling the properties of the system using formal methods. This inevitably adds complexity to the verification process.
- No previous work was found which considered generating test cases to study the effect of changing some (or all) of the implementation decisions for a given scheduling algorithm on the overall embedded system behaviour which incorporates this scheduler. Even very simple algorithms such as TTC have not been tested in this regard.

- Automated code generation, although useful, cannot provide the user with information about the possible behaviour patterns the generated scheduler code may produce after the system is executed. Also, the use of such techniques has not been evaluated with complex, large-scale TTC scheduler implementations.

12.3 Single-processor study

12.3.1 TTC scheduler implementations

12.3.1.1 Overview

The work on single-processor systems in this thesis began by reviewing a set of selective implementations for TTC scheduling algorithm which were either developed previously in the ESL research group or recently in this project. It was emphasised that the reviewed implementations were selected – according to their features – so that they represent the full range of TTC scheduler implementations developed so far in the ESL group. A brief summary of the features of such TTC implementations is provided in this section.

Note that other examples which were not identified as representative TTC implementations are reviewed and evaluated later in Appendix C. Figure 12-1 summarises all TTC scheduler implementations documented and evaluated in this project. Note that those which were classified as “representative” implementations are shaded to distinguish them from the others.

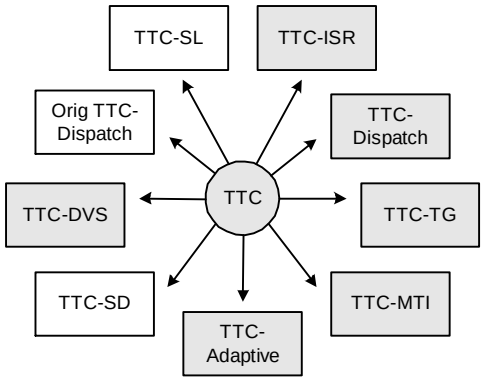


Figure 12-1: All TTC scheduler implementations reviewed in this study.

12.3.1.2 TTC-ISR scheduler

The TTC-ISR scheduler discussed in this thesis is a very simple version of the TTC schedulers used in a wide range of practical systems. It has low resource requirements and the behaviour is easy to understand (and predict).

However, the TTC-ISR implementation has two main drawbacks. The first drawback is that it provides no clear separation between the "scheduler" and the "user" (application) code. One consequence of this is that, if large numbers of tasks are used, the system requires a large amount of hand coding and can be difficult to maintain. The second drawback is that, because tasks are called from an "update" function which is linked to the timer ISR, the system will lose track of any further timer "ticks" which occur during the execution of (for example) long tasks. This second drawback means that the scheduler is very fragile in the event of task overruns.

12.3.1.3 TTC Dispatch scheduler

The TTC-Dispatch scheduler addresses both of the TTC-ISR limitations. First, the scheduler provides a clearer distinction between scheduler and tasks, making the system always easy to maintain and expand. Secondly, the separation of the implementation into an "update" and a "dispatch" functions means that the scheduler is more robust in the event of task overruns.

12.3.1.4 TTC-DVS scheduler

One concern with time-triggered (as opposed to event-triggered) designs is that power consumption can be increased. As discussed in this thesis, use of Dynamic Voltage Scaling (DVS) can help to reduce the power consumption in TTC implementations without jeopardising low-jitter behaviour.

12.3.1.5 TTC-TG scheduler

As noted, a TTC-Dispatch implementation has the ability to tolerate tasks which exceed their predicted WCET values, however, this ability is limited. In the most severe cases, an overrun could mean that a high-priority task tries to execute "for ever", denying lower-priority tasks access to the CPU. As discussed in this thesis, "task guardians" can be added to the TTC-Dispatch design so as to provide a more flexible response in the event of task overruns.

12.3.1.6 TTC-MTI scheduler

A simple TTC implementation based on periodic timer interrupts provides highly-predictable behaviour for the first task in every tick interval. However, if more than one task are executed in a tick interval, the release times of later tasks will depend (in many TTC implementations) on the execution time of earlier tasks. As demonstrated in this thesis, use of multiple timer interrupts can significantly reduce jitter levels in later tasks.

Note that – because this scheduler is similar to the TTC-ISR scheduler in construction (in that tasks are called from the ISR) – it has low resource requirements (i.e. less code memory is required than for the Dispatch scheduler). However, the underlying architecture means that it cannot support tasks which execute for longer than a single tick interval.

12.3.1.7 TTC-Adaptive scheduler

To deal with task overrun problem while maintaining low levels of task jitter, this scheduler has been developed. As previously noted in this thesis, dealing with task overrun and task release jitter requires knowledge about the task WCETs. In previous implementations, it was assumed that such values are provided to the scheduler by the user. The TTC-Adaptive scheduler was developed to offer a flexible implementation in which the user does not need to estimate the task WCETs during the design stage which, in many cases, cannot be accurate and may hence cause a significant degradation in the timing performance of the system.

As discussed in the thesis, the TTC-Adaptive scheduler employs an online measurement method to calculate the WCET for all tasks over a sufficient period of time. Such values are then used by the scheduler to adjust the timing of tasks and protect (guard) any task from overrunning. Since it was adapted from the TTC-MTI scheduler, TTC-Adaptive scheduler also has low resource requirements, e.g. low code memory is required.

12.3.2 STCs for TTC scheduling algorithm

As previously noted, the aim of this project is to bridge the gap between scheduling algorithms and scheduler implementations in embedded systems employing TTC architectures by means of testing. Since testing requires test cases, the work described in this thesis proposed an effective set of “scheduler test cases” (STCs) to evaluate, and

hence, compare the behaviour of the different TTC scheduler implementations reviewed.

The technique presented employs four STCs in total, where each of these test cases attempted to explore the response (behaviour) of the system under different condition. For example, each test case was intended to investigate the impact of a particular problem which might be linked to predictability in TTC schedulers. As previously argued, the main two problems which can have significant impacts on predictability in TTC systems are task jitter and overruns. Thus, STC A and B were developed to test the capability of the system to deal with jitter that arises from tasks and schedule (respectively). STC C investigated the impact of scheduling long tasks on the sequence and timing behaviour of other tasks in the scheduler, where STC D finally tested how the system would behave when a major error – such as a task overrun – took place.

In all STCs, except STC D, the sequence of task executions was recorded along with tick and task jitter. In STC D, the way the system reacts to the task overrun is recorded. In addition to these results, resource requirements to implement each of the compared TTC scheduler implementations were also reported. Such measurements were used as characterising features for each implementation. The aim with this process was to provide the user with as much information as possible about each scheduler implementation so that they can either select the most suitable scheduler for their project or identify which category their existing scheduler belongs to.

12.3.3 Assessing the behaviour of TTC schedulers

The reviewed TTC scheduler implementations outlined in Section 12.3.1 formed the testbed to examine the effectiveness of the STC technique summarised in Section 12.3.2. This section provides a brief summary of the results detailed in Chapter 7 when the STC technique was applied to the target TTC schedulers. In Chapter 7, first, the experimental methodology used to obtain the required measurements was outlined. Before presenting the results in each implementation, the way in which the STCs were employed in that particular implementation was described. Thereafter, results – in terms of task sequencing, overrun-handling, jitter, CPU, memory and power requirements – were presented and analysed.

A summary of the results was provided before Chapter 7 was concluded, where the key results were listed and compared using both a summary table and illustrative graphs. The graphs clearly show how each scheduler implementation can be superior to the others in a particular area. For example, TTC-DVS cannot be competed with when power consumption is the most crucial factor to consider, where this was on the account of (for example) data and code memory overheads. Likewise, TTC-ISR was so simple and required less memory overheads than other schedulers but produced extremely high levels of tick jitter when a long task was scheduled in the system.

The results also show that with TTC-MTI and TTC-Adaptive implementations, the behaviour of the TTC scheduler has been improved in areas including jitter and code memory overheads, without compromising CPU, data memory and power consumption as compared to the majority of schedulers. It is important to highlight that – in addition to the jitter reduction feature – the two developed implementations provided effective solutions to the problem of task overruns (this was shown in the summary table but not in the graphs). Although TTC-TG scheduler provided a complete solution to task overrun by employing appropriate "tasks guardians", it did not address other problems such as jitter which can, in many cases, cause the system behaviour to be entirely unpredictable.

The study concluded by pointing out that a perfect TTC scheduler which matches the requirements of every embedded design cannot be found in practice. It was therefore suggested that the scheduler for a particular application must be selected based on (for example) its timing, power, CPU and memory constraints.

12.4 Multi-processor study

12.4.1 Network and scheduling protocols

Having completed the work on single-processor embedded designs, the thesis moved on to consider the various issues which relate to the process of implementing and evaluating multiple-processor embedded systems. In particular, the thesis began to investigate how far the STC technique (developed for single-processor-based schedulers) can be used as the system goes more complicated. Thus, multi-processor embedded architectures which are based on distributed schedulers were considered.

The study provided by a discussion about the available network and scheduling protocols which were extensively used (or highly recommended) in the design and implementation of such systems.

These discussions began by network protocols for multi-processor designs. It provided the key features of a wide range of protocols with a particular focus on Controller Area Network (CAN) protocol that was selected to represent the hardware platform for all implementations considered in this thesis due to a set of recognised features, such as low-cost, availability and widespread use. CAN was described in a greater detail and then compared to alternative protocols which included Time-Triggered Protocol (TTP), FlexRay, RS-485, Local Interconnect Network (LIN) and Ethernet. It was emphasised that TTP and FlexRay can be competitive solutions to CAN, but their limited availability and use implies that CAN would remain the most appropriate choice for many embedded designs.

The discussions then considered high-level scheduling protocols – which can be used in conjunction with CAN hardware – to obtain time-triggered network operations (as opposed to event-triggered behaviour provided by the original CAN). Such protocols included Time-Triggered CAN (TTCAN) and Shared-Clock (S-C) protocols. Based on the features of each protocol, it was concluded that S-C schedulers can be a good match for many embedded designs: mainly due to their simplicity, low-cost and high predictability.

However, one key issue in the S-C protocol was highlighted: that is the jitter in the relative timing of Master and Slave ticks caused by bit stuffing mechanism employed in the CAN hardware upon which the S-C is implemented. To address such a jitter problem, three new techniques – developed during the course of this project – were listed. As noted, the techniques proposed were based on generic data coding approaches. The complete description and evaluation of the techniques are provided later in Appendices E.

12.4.2 TTC-SCC scheduler implementations

12.4.2.1 Overview

As in the single-processor study, the work on multi-processor systems in this thesis began by reviewing a set of selective implementation options for TTC-SCC scheduling protocol which were either developed previously in the ESL research group or recently in this project. Again, such implementations were identified as representative implementations of the TTC-SCC scheduler. A brief summary of the features of these implementations is provided in this section.

12.4.2.2 TTC-SCC1 scheduler

TTC-SCC1 is a simple version of the S-C protocol. It employs a simple TDMA protocol in which the Master talks to one Slave only in each tick interval. The TDMA round was therefore proportional to the the number of Slaves connected in the system. The two major concerns about this implementation, as noted, were the possibility of having long TDMA round and the lack of support for Slave-to-Slave communication. One consequence of having long TDMA round in this scheduler (where the Master talks to each Slave only once in the TDMA) is that a node failure may not be detected as quickly as required: such behaviour can have a significant impact on system predictability.

12.4.2.3 TTC-SCC2 scheduler

TTC-SCC2 was developed based on the TTC-SCC1 and intended to offer high flexibility in the communication between the nodes. Initially, a simple example was given in which the Master can talk to one particular Slave every other tick interval. Such a Slave was described as a critical Slave which required to be checked by the network-Master regularly at high rates. A slightly more complicated example was then provided in which it was assumed that the Master talks to the critical Slave at any frequency. The TDMA round for both cases was calculated and it was shown that it was longer than that in TTC-SCC1.

It was then emphasised that the two examples represented only limited use of this scheduler. A more general example was therefore given to show how the TDMA round can have a random pattern. This is entirely based on application requirements.

As with TTC-SCC1, this scheduler suffers lack of support for direct communication between Slaves, thereby causing the transmission time between any two Slaves comparatively long. Also the failure detection time of particular Slaves (which are checked less frequently) can be very long.

12.4.2.4 TTC-SCC3 scheduler

The limitations observed in TTC-SCC1 and TTC-SCC2 were addressed through the development of a third implementation of the S-C scheduler which was called TTC-SCC3. Such an implementation allowed each Slave to send its messages to all Slaves in the network. It also allowed the Master to talk to all (or a group of) Slaves within a single tick interval, causing a significant reduction in the length of the TDMA round.

The major problem in this and all the previous schedulers, as discussed in detail, is the high levels of jitter at the release time of Slave tasks due to variations in the length of Tick messages. As previously noted, such variations are dependent on the nature of the transmitted data in the Tick messages. If (for example) data sent in the Tick messages are likely to be random, this can have more impact on the timing of Slave tasks.

12.4.2.5 TTC-SCC4 scheduler

TTC-SCC4 provided an attractive solution to the jitter problem caused by variations in the transmission time of Tick messages. This was achieved by allocating a separate node for generating the heartbeat of the network. This node was seen as a “tick-only-Master” node which processes no data. The Master node in previous implementations becomes an ordinary Slave node which only sends data messages. Apart from this feature, the same message configuration – as in TTC-SCC3 – was used with this scheduler.

The only problem with TTC-SCC4 scheduler is that it required an additional microcontroller board only to send Tick messages while not being involved in any other activities. This obviously caused a reduction in the resource efficiency.

12.4.2.6 TTC-SCC5 scheduler

In order to combine the features of TTC-SCC3 and TTC-SCC4 without adding more cost to the system, the TTC-SCC5 was developed in this project. Simply, such a scheduler allowed the Master node to send two types of messages consecutively. The first message was only to trigger the Slave nodes at precisely-fixed intervals, where the following message was designated for Master data intended for all or some Slaves. The implementation process of such a scheduler was described. It was pointed out that although the bandwidth utilisation might be slightly reduced, due to the scheduling of additional messages in the tick intervals, TTC-SCC5 can provide a highly-predictable message and task operations compared to all previous implementations.

12.4.3 STCs for TTC-SCC scheduling protocol

As in the single-processor study, an appropriate set of "scheduler test cases" (STCs) were developed to assess (and distinguish) the behaviour of the TTC-SCC schedulers outlined in the previous section. The same conditions considered in the process of developing STCs for single-processor systems were also appreciated here. For example, the STCs were selected so that they help to discover the various aspects of the TTC-SCC scheduler when implemented in practical systems.

It was noted that, in such a scheduling protocol, the Master and Slave nodes employ TTC-Dispatch scheduler to manage the operation of their tasks. The aim with the STCs was not to assess the behaviour of individual schedulers (as in single-processor study) but to assess the S-C protocol employed to facilitate the communication between the individual nodes. Therefore, the communication behaviour of each TTC-SCC scheduler implementation was considered to be evaluated using test cases that specifically address the communication latency between any two nodes in the network (STCs B, C and D) and the time required to detect and handle a temporary node failure (STC E). To help address the predictability of the system further, a test case (i.e. STC A) was developed specifically to assess the jitter levels in the transmission time of messages sent from Master to Slaves. Measurements for such kind of jitter were fairly important since its levels influence the overall timing accuracy and hence predictability of the whole network.

12.4.4 Assessing the behaviour of TTC-SCC schedulers

Again, as with single-processor study, the reviewed TTC-SCC scheduler implementations outlined in Section 12.4.2 formed the testbed to test the effectiveness of the STC technique summarised in Section 12.4.3. This section provides a brief summary of the results detailed in Chapter 11 where the STC technique was applied to the various TTC-SCC schedulers. Chapter 11 began by outlining the methodology used in obtaining the required results. Next, results were presented for each scheduler. The results included jitter, Master-to-Slave, Slave-to-Master and Slave-to-Slave latencies, node-failure detection time, bandwidth utilisation and memory requirements.

A summary of the results was presented at the end of Chapter 11 using a small realistic case study that helped understand the different behaviour patterns of the evaluated TTC-SCC schedulers. It was clearly shown that the STC is quite flexible so that it can be adapted for use in any scheduling algorithm, regardless how complicated the system is. Of course, the complexity of the test case design process for a particular system would increase as the system goes more complicated.

12.5 Conclusions

This chapter provided a brief overview of the studies carried out in this PhD research project and summarised the key obtained results. It began by highlighting the main gaps identified in the literature review of research areas related to this project. Then, the approaches proposed to fill these gaps were discussed in summary. For consistency, this chapter followed the same structure used in presenting the previous chapters.

Chapter 13
Conclusions and future work

13.1 Introduction

Chapter 12 provided a summary of the work carried out in this PhD research project and discussed the obtained results. Based on such discussions, this chapter draws the overall thesis conclusions and provides some suggestions for future work in the areas concerned with in this project.

13.2 Main achievements

As clearly stated in this thesis, the work carried out in this project aimed to bridge the gap – which was identified but not systematically addressed – between the scheduling theory and its implementations in practical real-time embedded environments. In order to tackle this problem, the thesis attempted to address the process of translating between the two core system-representations, scheduling algorithm and scheduler implementation, while ensuring highly-predictable system behaviour during this process.

The importance of each of such system components has been emphasised. However, it was underlined that, even if the right scheduling algorithm is selected at the design phase of the system development process, inappropriate decisions at the scheduler implementation phase can lead to undesirable consequences, not least the inability of the system to meet its functional and temporal requirements. This means that it is likely that the system will behave incorrectly during all (or some) of its operating period. The impact of improper scheduler implementation decisions on the system behaviour have been discussed in detail.

The studies presented in this thesis mainly considered the process of implementing Time-Triggered Co-operative (TTC) scheduling algorithm as a simple, low-cost software architecture for many embedded applications which have severe resource

constraints and require very high degrees of predictability. In order to link TTC scheduling algorithm with its practical implementations, the thesis began by categorising the various TTC scheduler implementations developed in the ESL research group since 2001. In total, nine different implementations were collected. However, only six of these were selected for detailed evaluation in this thesis. The particular TTC implementations were selected in such a way that each implementation would be expected to demonstrate recognisably different patterns of behaviour during periods of normal and/or abnormal system operation. This led to a detailed review and evaluation of only TTC schedulers which are representative of all implementations.

To be able to evaluate TTC schedulers, the thesis suggested that an appropriate verification method must be applied between the TTC design and implementation stages to ensure that an implementation matches the original design specification. By reviewing a number of generic verification techniques, it was decided that only dynamic verification techniques – namely testing – would be suitable to address this problem since it facilitates examining the scheduler while it is running on the target hardware. Therefore, the technique proposed to verify the TTC implementations was based on a form of testing. However, it was emphasised that the STC technique was not an ordinary testing method which checks the system against its required functionality. Instead, the STC was specifically developed to assess the behaviour of TTC scheduler with regards to a set of parameters that – in one way or another – influence the system predictability.

The application of STC technique was found to be very effective in both single- and multi-processor embedded systems due to the following reasons:

- It allowed a systematic classification and documentation of the various scheduler implementations which have been developed in the ESL research group over the last eight years.
- It helped identify a small set of “standard” forms of the scheduling algorithm (e.g. TTC) that can satisfy the requirements of a wide range of time-triggered embedded applications.

- It helped understand – practically – the implications of using a particular scheduler implementation on the overall operational behaviour of system implementing this scheduler.
- It facilitated a detailed "black-box" comparison between the various scheduler implementations without the need to access (or attempt to understand) the underlying scheduler source codes.
- It helped assess the predictability levels of systems incorporating particular scheduler implementations.
- It provided the facility to help user select the most appropriate scheduler implementation for a particular project, or alternatively identify which scheduler implementation has been used in their system. By doing so, the user would be able to predict how their system is likely to behave in the future or, alternatively, decide that a different form of system implementation should be employed.

13.3 Limitations and future work

Of course, there are limitations in any research project. This section attempts to identify the key limitations of the project summarised in this thesis and proposes some ideas to address some (or all) of them in a future work.

First, it must be noted that this project was concerned with testing the TTC scheduling algorithm since it represents the simplest form of scheduler that is in widespread use. This is because all tasks are scheduled in predefined sequence without interruption from other tasks. As discussed in Chapter 2, TTC scheduler was seen simple and highly-predictable. This simply means that if a project is to be launched, the designer should begin by considering the use of such a scheduling algorithm for the system tasks. Only in cases where the system cannot meet its requirements or achieve the level of performance expected by implementing TTC scheduler is the scheduler dismissed and alternative architectures considered.

The ESL research members has suggested that if TTC does not match a particular application, where task pre-emption is needed to meet hard deadlines, then Time-Triggered Hybrid (TTH) schedulers – which provide a limited degree of pre-emption –

would be recommended. Having accepted that, the TTH schedulers need to be evaluated in a systematic way. As with every other scheduling algorithm, TTH scheduler can have a wide range of possible implementation options, each with different operational behaviour (see Section 3.5.3). It would therefore be recommended to extend the STC technique to be able to document, assess and compare the various implementation classes of TTH in the same way as with TTC schedulers. For example, a test case would be required to determine which of the co-operative tasks is to be converted into pre-emptive, and how such a modification can affect the system behaviour. Similarly, the STC technique can be extended to evaluate further scheduling algorithms in widespread use such as Rate Monotonic (RM) and Earliest Deadline First (EDF). One of the features that can be tested in such algorithms is the ability of the system to deal with priority inversion problem.

Moreover, the TTC-Adaptive scheduler presented in Chapter 5 can also be extended. For example, the existing version of this scheduler was found effective with co-operative tasks. It would be a good idea if the scheduler framework can be modified to work as a hybrid scheduler. On the other hand, the WCET method implemented in the TTC-Adaptive scheduler only calculates the WCETs and RTs for all tasks during the first phase of the system operation (during the calculating mode). It was assumed that the user had set the duration of the calculating mode long enough to obtain a correct set of WCET values based on their knowledge about the system characteristics. By doing so, there might be a possibility that the calculating period was set incorrectly by the user and thus the actual WCET of all (or some of) the tasks were missed. This means that the behaviour of the system during the operating mode would be unpredictable. For example, high jitter might be observed at task release times and task guardians might detect a task overrun where the task is still within its WCET. In order to avoid such a scenario, the system can be modified so that the whole calculation process is automated while the user's interference is avoided. For example, the scheduler can employ a permanent run-time WECT measurement method so that whenever the WCET value of a task is modified the system adapts itself to this change.

Please note that the TTC-Adaptive scheduler was aimed towards a perfect TTC implementation since it provided effective solutions to jitter and overrun problems. However, the "perfect" TTC scheduler can be achieved if more features are considered.

For example, techniques such as DVS can be incorporated in the scheduler framework to achieve low-power characteristics at zero jitter. Such a modification would require a substantial amount of underlying work in order to avoid any conflicts between timer configurations.

For multi-processor systems, there are some future suggestions. First, the range of TTC-SCC scheduler implementations can be extended to include other possible arrangements for Master and Slave communication messages. Of course, the implementations discussed, although useful, represent only a number of the representative implementation classes for such a scheduler. For example, systems which use dual-CAN bus can also be added to the range of TTC-SCC schedulers and evaluated. This might need an addition of new STCs that explore more features related to this system modification.

Despite the usefulness of the jitter-reduction techniques outlined in Chapter 8, they can still, in some cases, be outweighed by the increases in the resource requirements (such as CPU and memory). To achieve similar levels of performance while reducing the resource overheads, it would be recommended that techniques – such as SBS and EEM – are implemented in hardware using SoC designs. This has the potential to free the system from unnecessary overheads and increase the system predictability.

13.4 Conclusions

The work described in this thesis has mainly considered the development and evaluation of a simple verification technique (STC) aimed at facilitating a meaningful “black-box” evaluation of time-triggered embedded systems. The practical work began by exploring the benefit of using the STC technique in a simple TTC scheduling algorithm used in single-processor embedded architectures. The thesis then explored ways for extending the technique to allow evaluating the behaviour of more complicated embedded designs: for example, when multi-processor architectures are considered. The results show that the proposed technique can have the potential to provide a detailed evaluation of any embedded software design, when appropriate scheduler test cases are employed.

Apart from this, the thesis also described a range of highly-predictable time-triggered scheduler implementations, for both single- and multi-processor systems, and demonstrated how such new implementations can add very useful features to many real-time resource-constrained embedded applications. Some other interesting areas which relate to the work presented in this thesis were also discussed in detail. The thesis concluded by summarising the key achievements of this project and making useful suggestions for future research projects in the same areas.

PART F:

APPENDICES

Appendix A

Overview of system development process

Introduction

This appendix describes in detail the various processes involved in software development with a particular focus on embedded software systems. Sources for this appendix include (McLaughlin and Moore, 1998; Booch *et al.*, 1999; Pont, 2001; Shaw, 2001; Douglass, 2004; Buttazzo, 2005; Marwedel, 2006; Mwelwa, 2006; Sommerville, 2007).

System requirements

Embedded systems engineering is viewed as a branch of system engineering discipline where engineers are concerned with all aspects of computer-based development including hardware, software and process engineering. Therefore, activities such as specification, design, implementation, validation, deployment and maintenance will all be involved in the development of an embedded application. A design of any system usually starts with ideas in people's head. These ideas need to be captured in requirements specification documents that specify the basic functions and the desirable features of the system.

System design process then determines how these functions can be provided by the system components. The following figure illustrates the life cycle of a system development process.

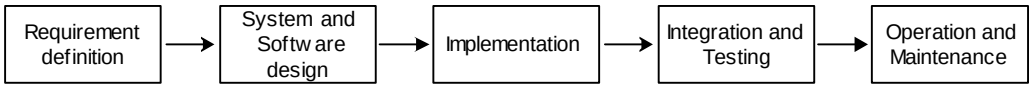


Figure A-1: The system development process (adapted from Sommerville, 2007).

For successful design, the system requirements have to be expressed and documented in a very clear way. Inevitably, there can be numerous ways in which the requirements for a simple system can be described. The simplest and most obvious way to express the system requirements is to use a detailed natural language, such as English, French and

so on. However, due to limitations caused by (for example) overflexibility and ambiguity of the natural languages, systems described using this method are likely to be misunderstood. Furthermore, it is impossible to use the system specification written in a natural language to derive implementations in a systematic way. Therefore, more technical ways of representation might be required. Sommerville (2007) notes that using more specialised notations such as structured natural language, design description language, graphical notations and mathematical specifications can be more effective ways for documenting the designed system. For example, structured natural language – early used in 1980 to describe the requirements for an aircraft system (Heninger, 1980) – uses standard forms that utilise the advantages of natural languages, such as clarity and understandability, while removing some of the language limitations. However, these approaches can still have shortcomings when complex computations are required: for example, as the complexity of the system increases it is so difficult to specify the requirements in an unambiguous way when a natural language text is integrated. One solution to this problem is to use supplementary materials such as tables or graphical models which add extra information about the system.

System modelling

A widely used systematic approach for documenting system requirements is to use a set of system *models* which are basically forms of graphical representations that represent the system from different perspectives. A system model is developed to provide an abstract view of the system while deliberately ignoring some system details. Each system model may cover only one (or sometimes more) particular aspect of the system, while a combination of different models is needed to provide wider scope of the system. Different system models can be classified into: *context* models, *behavioral* models, *data* models and *object* models (see Sommerville, 2007 for more details). Two common types of models which are generally used to describe the overall behaviour of the system are known as *data-flow* and *state machine* models. A data-flow model is very important way to show how data flows in the system through a sequence of processing steps. A state machine model shows events or system states that cause the system to move from one state to another. Since real-time systems are often driven by events in the environment, state machine model is widely used in the design of such systems.

To describe a system model, an appropriate form of specification language is required. There is a wide range of specification languages which are available nowadays, e.g. StateCharts, System Design Language (SDL), Unified Modeling Language (UML), Java, Verilog Hardware Description Language (VHDL), SystemC and SpecC. Note that each of these languages has different capabilities and hence provide different representations of the designed system. As a result, the choice of the language for an actual design will highly depend on the application domain and the environment in which the system has to be operated.

However, one of the most widely used specification language in the fields of system and software engineering is the UML. UML contains a large set of notations and diagram types that make it a de facto standard modelling language for a broad range of application domains. One key advantage of UML, compared to other specification languages, is that it does not require precise knowledge of the system behaviour which is, in most cases, not available at very early stages of the design process. An early version of UML (e.g. UML 1.4) was not supporting the modelling of real-time embedded systems due to a lack of some important features required to model those systems, e.g. timing and concurrency. However, new versions of UML (e.g. UML 2.0) have been developed to support the design modelling for embedded systems. UML 2.0 contains various diagram types such as sequence diagrams, state machine diagrams, activity diagrams, use case diagrams, timing diagrams, object diagrams and some others. Based on the sets of notations and diagram types provided by UML 2.0, data-flow diagram – as an example – can be clearly represented using 'rounded rectangles' for functional processing, 'rectangles' for data stores and 'labelled arrows' for data transfer between different functions. Similarly, state machine diagram can be represented using 'rounded rectangles' for system states, 'labelled arrows' for events that cause transition from one state to another. However, it should be noted that the UML models can only be viewed as 'high-level' representations of the system which need an executable programming language (e.g. C or C++) to be combined with for achieving precise, executable system specifications.

Formal specification

Before beginning the actual design process, *formal specification* techniques can also be used to add details to a system requirements specification. A formal specification of

software is the specification which is expressed in a language whose vocabulary, syntax and semantics are formally defined: this means that the used specification language must be based on mathematical concepts. Examples of formal specification languages are: Larch (Guttag and Homing, 1993), OBJ (Futatsugi *et al.*, 1985), Z (Spivey, 1988), VDM (Jones, 1989), and B (Wordsworth, 1996). The main advantage of using formal specifications is that they help to avoid ambiguities in the system requirements documentation.

System architecture

Once the system requirements have been clearly defined and well documented, the first step in the design process is to design the overall system *architecture*. Architecture of a system basically represents an overview of the system components (i.e. sub-systems) and the interrelationships between these different components. Since embedded engineers are concerned with hardware and software design aspects of the system, they must decide on both the hardware and the software architectures of the intended design.

Douglass (2004) defines architecture as: *"the set of strategic design decisions that affect the structure, behaviour, or functionality of the system as a whole"*. In (Sommerville, 2007), it is highlighted that designing the architecture of a system is the process of creating a basic structural framework which identifies the key components of the system and the communication between these components. It is also noted that the output of this design process is a description of the software architecture which provides a high-level representation of the system. System and software architectures also need appropriate ways of modelling and documentations. It is widely adopted that architectures are illustrated graphically using simple *block diagrams* in which the system components can be represented by a set of 'rectangles' linked to one another by 'arrows'.

Clear documentations of the software architecture help the developers to consider key design aspect of the system early in the design process (Sommerville, 2007). Since the software architecture provides a high-level abstraction of the system, it helps the developers to establish discussions about the system requirements and begin to predict how the system will operate after implementation. Determining the most appropriate architecture is a key part in the design and implementation of a given system. In

embedded systems, there are two fundamental software architectures which are generally used: *time-triggered* and *event-triggered*.

System implementation

Once the software architecture is identified, the process of implementing that architecture should take place. This can be achieved using a lower-level system representation such as a ***scheduler*** or an operating system. A scheduler is a very simple operating system which organises the operation of real-time tasks and manages the computational and data resources in the system. The most key part of the scheduler is the ***scheduling algorithm*** which states the set of rules that specify the order in which the tasks will be executed by the scheduler during the system operating time. Once the scheduling algorithm has been selected and the schedule designed, the low-level implementation of the scheduler will take place by generating the scheduler source code using a software programming language. The scheduler source code is the lower level representation of the system which should determine the actual behaviour of the system once run on the target hardware.

Appendix B

Overview of programming languages

Introduction

This appendix provides an overview of the available programming languages used nowadays in computer science and real-time embedded systems. It discusses the classification of programming languages and provides a historical background. The features of 'C' language (outlined in Chapter 3) are provided here in a little more detail.

What is a programming language?

Simply, programming as a problem has only arisen since computer machines were first created. The magnitude of the problem is however relative to the size (and complexity) of the computer machine used: for example, with using gigantic computers, programming becomes an equally gigantic problem (Cook, 1999). To program a computer system, a programming language is required. The latter is seen as the major way of communication (i.e. interface) between a person who has a problem and the computer system used to solve his problem. Programming language has been defined in several ways. For example, American Standard Vocabulary for Information Processing (ANSVIP, 1970) defined a programming language as *"A language used to prepare computer programs"*. The IFIP-ICC Vocabulary of Information Processing (IFIP-ICC, 1966) defined it as *"A general term for a defined set of symbolic and rules or conventions governing the manner and sequence in which the symbols may be combined into a meaningful communication"*. The IFIP-ICC glossary also noted that *"An unambiguous language, intended for expressing programs, is called a PROGRAMMING LANGUAGE"*. Other definitions for a programming language are:

- *"A computer tool that allows a programmer to write commands in a format that is more easily understood or remembered by a person, and in such a way that they can be translated into codes that the computer can understand and execute."* (Budlong, 1999).
- *"An artificial language for expressing programs."* (ISO, 2001).

- *"A self-consistent notation for the precise description of computer programs"* (Wizitt, 2001).
- *"A standard which specifies how (sort of) human readable text is run on a computer."* (Sanders, 2007).
- *"A precise artificial language for writing programs which can be automatically translated into machine language."* (Holyer, 2008).

However, it was noted elsewhere (e.g. Sammet, 1969) that standard definitions are usually too general as they do not reflect the language usage. A more specific definition for a programming language was given by Sammet as a set of characters and rules (used to combine the characters), with the following characteristics:

- A programming language requires no knowledge of the machine code by the programmer, thus the programmer can write a program without much knowledge about the physical characteristics of the machine on which the program is to be run.
- A programming language should be machine independent.
- When a program written in a programming language is translated to the machine code, each statement should explode to generate a large set of machine instructions.
- A programming language must have problem-oriented notations which are closer to the specific problem intended to be solved.

It is worth mentioning that a vast number of different programming languages have already been created, and new languages are still being created every year.

Classification of programming languages

This section provides an overview of the programming language classifications. Sources for this section include (Sammet, 1969; Booch, 1991; Grogono, 1999; Lambert and Osborne, 2000; Mitchell, 2003; Calgary, 2005; Davidgould, 2008; Wikipedia, 2008; Network Dictionary, 2008).

In general, programming languages can be divided into programming paradigms and classified by their intended domain of use. Paradigms include procedural programming, object-oriented (O-O) programming, functional programming, and logic programming. Note that some languages combine multiple paradigms. Each of these paradigms is briefly introduced here.

Procedural programming (or imperative programming) is based on the concept of decomposing the program into a set of procedures (i.e. series of computational steps). Examples of procedural languages are: FORTRAN (**FOR**mula **TRAN**slator), Algol (**ALGO**rithmic **L**anguage), COBOL (**CO**mmon **B**usiness **O**riented **L**anguage), PL/I (**P**rogramming **L**anguage **I**), Pascal, BASIC (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode), Modula-2, C and Ada. Object-oriented (O-O) programming is a method where the program is organised as a cooperative collections of “objects”. This style of programming was not commonly used in software application development until the early 1990s, but nowadays most of the modern programming languages support this type of programming paradigm. Examples of object-oriented languages are: Simula, Smaltalk, C++, Eiffel and Java. Functional programming treats computation as the evaluation of mathematical functions. In functional programming, a high order function can take another function as a parameter or returns a function. An example of functional languages is LISP (**LIS**t **P**rocessor). Finally, logic programming uses mathematical logic in which the program enables the computer to reason logically. An example of logic languages is Prolog (**PRO**gramming in **LOG**ic). It is often argued that languages with support for an object-oriented (O-O) programming style have advantages over those from earlier generations (Pont, 2003). For example, Jalote (1997) noted that using O-O helps to represent the problem domain, which makes it easier to produce and understand designs.

In addition to programming paradigm, the purpose of use is an important characteristic of a language: it is unlikely to see one language fitting all needs for all purposes (Sammet, 1969). Programming languages can be divided, according to their purpose, into general-purpose languages, system programming languages, scripting languages, domain-specific languages, and concurrent / distributed languages (or a combination of these). General-purpose language is a type of programming language that is capable of creating various types of programs for various applications, e.g. C language. There has

been an argument that some of the general-purpose languages were designed mainly for educational purposes (Wirth, 1993). System programming language is a language used to produce software which services the computer hardware rather than the user, e.g. Assembly and Embedded C. Scripting language is a language in which programs are a series of commands that are interpreted and then executed sequentially at run-time without compilation, e.g. JavaScript (used for web page design). Domain-specific programming languages are, in contrast to general-purpose languages, designed for a specific kind of tasks, e.g. Csound (used to create audio files), and GraphViz (used to create visual representations of directed graphs). Concurrent languages are programming languages that have abstractions for writing concurrent programs. A concurrent program is the program that can execute multiple tasks simultaneously, where these tasks can be in the form of separate programs or a set of processes or threads created by a single program. Concurrent programming can support distributed computing, message passing or shared resources. Examples of concurrent programming languages include Java, Eiffel and Ada.

In his famous book (i.e. “Programming Languages: History and Fundamentals”, 1969), Jean E. Sammet used the following set of defining categories as a way of classifying programming languages: [1] procedural and non-procedural languages; [2] problem-oriented, application-oriented and special purpose languages; [3] problem-defining, problem describing and problem solving languages; [4] hardware, publication and reference languages. Sammet however underlined that any programming language can fall into more than one of these categories simultaneously: for further details see Sammet (1969).

History of programming languages

It has been argued that studying the history of programming languages is paramount as it helps developers avoid previously-committed mistakes in the development of new languages (Wilson and Clark, 2000). It was also pointed out that an unfortunate tendency in Computer Science is to create new language features without carefully studying previous work (Grogono, 1999). Most books and articles on the history of programming languages tend to discuss languages in terms of generations in which languages are classified by age (Cook, 1999). There are many articles and books which have discussed the generations of programming languages (e.g. Wexelblat, 1981;

Martin and Leben, 1986; Watson, 1989; Zuse, 1995; Flynn, 2001). Pont (2003) provides a list of widely-used programming languages classified according to their generations (see Table B-1).

Table B-1: Classification of programming languages by generations

Language generation	Example languages
-	Machine code
First generation language (1GL)	Assembly
Second generation languages (2GL)	COBOL, FORTRAN
Third generation languages (3GL) "process-oriented'	C, Pascal, Ada 83
Fourth generation languages (4GL) 'object-oriented'	C++, Java, Ada 95

A brief history of the most popular programming languages (including the ones presented in Table B-1) is provided in this section. Sources for this section mainly include (Wexelblat, 1981; Martin and Leben, 1986; Watson, 1989; Halang and Stoyenko, 1990; Grogono, 1999; Flynn, 2001; Wikipedia, 2008).

In the 1940s, the first electrically powered digital computers were created. The computers of the early 1950s used machine language which was quickly superseded by a second generation of programming languages known as Assembly languages. The limitations in resources (e.g. computer speed and memory space) enforced programmers to write their hand-tuned assembly programs. However, it was shortly realised that programming in assembly required a great deal of intellectual effort and was prone to error. It is important to note that although many people consider Assembly to be a standard programming language, some others believe it is too low-level to bring satisfactory of communication for user, hence excluded (Sammet, 1969).

1950s saw the development of a range of high-level programming languages (some of which are still in widespread use), e.g. FORTRAN, LISP, and COBOL, and other languages such as Algol 60 that had a substantial influence on most of the lately developed programming languages. In 1960s, languages such as APL (**A P**rogramming **L**anguage), Simula, BASIC and PL/I were developed. PL/I incorporated the best ideas from FORTRAN and COBOL. Simula is considered to be the first language designed to support object-oriented (O-O) programming.

The period between late 1960s and late 1970s brought great prosperity to programming languages most of which are used nowadays. In the mid-1970s, Smalltalk was introduced with a complete design of an object-oriented language. The programming language C was developed between 1969 and 1973 as a systems programming language, and remained popular. In 1972, Prolog was designed as the first logic programming language. In 1978, ML (**M**eta-**L**anguage) was developed to found statically-typed functional programming languages in which type checking is performed during compile-time allowing more efficient program execution. It is important to highlight that each of these languages originated an entire family of descendants. Some other key languages which were developed in this period include: Pascal, Forth and SQL (**S**tructured **Q**uery **L**anguage).

In 1980s, the C++ was developed as a combined object-oriented and systems programming language. Around the same time, Ada was developed and standardised by the United States government as a systems programming language intended for use in defence systems. One noticeable tendency of language design during the 1980s was the increased focus on programming large-scale systems through the use of modules, or large-scale organisational units of code. Therefore, languages such as Modula-2, Ada, and ML were all extended to support such modular programming in 1980s. Some other languages that were developed in this period include: Eiffel, PEARL (**P**ractical **E**xtraction and **R**eport **L**anguage) and FL (**F**unction **L**evel).

In mid-1990s, the rapid growth of the Internet created opportunities for new languages to emerge. For example, PEARL (which is originally a Unix scripting tool first released in 1987) became widely adopted in dynamic web sites design. Another example is Java which was commonly used in server-side programming. These language developments provided no fundamental novelty: instead, they were modifications to existing languages and paradigms and largely based on the C family of programming languages.

It is difficult to determine which programming languages are most widely used, as there have been various ways to measure language popularity (see O'Reilly, 2006; Bieman and Murdock, 2001). Mostly, languages tend to be popular in particular types of applications. For example, COBOL is a foremost language in business applications (Carr and Kizior, 2000), FORTRAN is widely used in engineering and science

applications (Chapman, 2004), and C is a genuine language for programming embedded applications and operating systems (Barr, 1999; Pont, 2002; Liberty and Jones, 2004).

Programming languages for embedded and real-time systems

To develop a real-time embedded system, a number of tools and techniques will be required: the key one is the programming language used to develop the application code (Burns, 2006). Assembly was the first programming languages used to implement the software for embedded applications. However, it was argued that the development environments that used the first generation languages such as Assembly lacked the basic support for debugging and testing (Halang and Stoyenko, 1990). Therefore, in 1960s, the need for high-level programming languages to program real-time systems, instead of continuing to use Assembly language, was agreed among many real-time system designers due to advantages such as ease of learning, programming, understanding, debugging, maintaining, documenting, and code portability (see Boulton and Reid, 1969; Sammet, 1969).

The work in this area began by identify the essential requirements for a high-level language to fulfil the objectives of real-time applications (Opler, 1966). Such requirements were summarised by Boulton and Reid (1969) as methods of handling real-time signals and interrupts, and methods of scheduling real-time tasks. Opler (1966) argued that to achieve such requirements, one can make extensions / modifications to an existing programming language, where an alternative solution is to develop new languages dedicated specifically for real-time software. Some success, in extending existing languages to real-time computing, was achieved using languages such as FORTRAN (e.g. Jarvis, 1968; Roberts, 1968; Hohmeyer, 1968; Mensh and Diehl, 1968; Kircher and Turner, 1968) and PL/I (e.g. Boulton and Reid, 1969). Some other studies, however, attempted to develop new real-time languages but with some similarity to existing languages, e.g. PROSPRO (Bates, 1968), SPL (Oerter, 1968) and RTL (Schoeffler and Temple, 1970).

In 1970s, a major concern of many researchers became the programming of real-time applications which involve concurrent processing. Useful work in this area demonstrated that, same as before, concurrent programming can be achieved by either extending available general-purpose languages (e.g. Hansen, 1975; Wirth, 1977) or

developing entirely new concurrent-processing languages (e.g. Schutz, 1979). However, it was noticed that extended general-purpose languages still lacked genuine concurrency and real-time concepts (Steusloff, 1984). This led to the development of more efficient concurrent real-time languages such as PEARL (DIN, 1979), ILIAD (Schutz, 1979) and Ada (Ada, 1980).

Ada is a well-designed and widely used language for implementing real-time systems (Burns, 2006). Therefore, it is worth mentioning it in greater detail. As previously noted, Ada is an object-oriented, high-level programming language which was first developed and adopted by the U.S. Department of Defence (DoD) to implement various defence mission-critical software applications (Ada, 1980; Baker and Shaw, 1989). Ada appeared as a standard language in 1983 – when Ada83 was released – and was later reviewed and improved in 1995 by producing Ada95. Since developed, Ada has gained a great deal of interest by many real-time and embedded systems developers (see Chapter 3 for example studies). It was declared that Ada embodies features which facilitate the achievement of safety, reliability and predictability in the system behaviour (Halang and Stoyenko, 1990). Halang and Stoyenko (1990) carried out a detailed survey on a number of representative real-time programming languages including Ada, FORTRAN, HALL/S, LTR, PEARL, PL/I and Euclid, and concluded that Ada and PEARL were the most widely available and used languages among the others which had been surveyed.

In addition to the previous sets of modified and specialised real-time languages, it was accepted that universal, procedural programming languages (such as C) can also be used for real-time programming although they contain just rudimentary real-time features: this is mainly because such languages are more popular and widely available than genuine real-time languages (Halang and Stoyenko, 1990). Later generations of object-oriented (O-O) languages such as C++ and Java also have popularity in embedded programming (Fisher *et al.*, 2004).

Overview of ‘C’ language

In his famous book “Programming Embedded Systems in C and C++”, Michael Barr (1999) emphasised that C language has been a constant factor across all embedded software development due to the following advantages:

- It is small and easy to learn.
- Its compilers are available for almost every processor in use today.
- There are so many experienced C programmers around the world.
- It is hardware-independent programming language, a feature which allows the programmer to concentrate only on the algorithm rather than on the architecture of the processor on which the program will be running.

Despite this, Barr highlighted that the key advantage of C which made it the favourite choice for many embedded programmers is its low-level nature that provides the programmer with the ability to interact easily with the underlying hardware without sacrificing the benefits of using high-level programming.

In (Grogono, 1999), it was declared that C is based on a small number of primitive concepts, therefore it is an easy language to learn and program by both skilled and unskilled programmers. Moreover, Grogono stated that C can be easily compiled to produce efficient object code.

In a more recent publication, Pont (2002) stated that *"C's strengths for embedded system greatly outweigh its weaknesses. It may not be an ideal language for developing embedded systems, but it is unlikely that a 'perfect' language will be created"*. According to (Pont, 2002 and 2003), the key features of the C language can be summarised as follows.

- It is a mid-level language with both high-level features (such as support for functions and modules) and low-level features (such as access to hardware via pointers).
- It is very efficient, popular and well understood even by desktop developers who programmed on C++ or Java.
- It has well-proven compilers available nowadays for every embedded processor (e.g. 8-, 16-, 32-bit or more).
- Books, training courses, code examples and websites that discuss the use of language are all widely available.

In (Jones, 2002), it was noted that features such as easy access to hardware, low memory requirements, and efficient run-time performance make the C language popular and foremost among other languages. In (Brosgol, 2003), it was made clear that C is the typical choice for programming embedded applications as it is processor-independent, has low-level features, can be implemented on any architecture, has reasonable run-time performance, is an international standard, and is familiar to almost all embedded systems programmers. Fisher *et al.* (2004) emphasised that, in addition to portability and low-level features of the language, C's structured programming drives embedded programmers to choose C language for their designs. Moreover, it has been clearly noted that C cannot be competed in producing a compact, efficient code for almost all processors used today (Ciocarlie and Simon, 2007).

Appendix C
Hardware-based scheduler implementation approaches

This appendix discusses a range of hardware (or a mix of hardware and software) techniques used previously to implement scheduling algorithms in practical real-time embedded systems.

In 1988, Wendorf (1988) considered the practical implementation issues of the Time-Driven Scheduler (TDS) developed originally by Jensen *et al.* (1985). Wendorf began by pointing out that fixed-priority schedulers may not perform well under overload conditions and the TDS had the potential to improve the performance of real-time systems under such conditions. TDS is a time-triggered, pre-emptive scheduler in which each task has a time-varying value function that defines the value of completing task at a given time. In addition, a "best-effort" (BE) scheduling policy (Locke, 1986) was designed and integrated into the TDS scheduler framework to maximise the total value of all completed tasks over a wide range of value functions and workloads. However, as noted by Wendorf, the practical implementation of the best-effort, time-driven scheduler and the impact on the computational overhead were not fully addressed. Therefore, Wendorf discussed the implementation and performance of the BE scheduling policy on a practical real-time system. Experimentally, it was found that under overload conditions, more than 80% of the CPU time could be spent by the scheduler to decide which task to execute next when a single-processor system is used. It was therefore suggested that all scheduling processes are performed in a dedicated scheduling processor and was shown that this solution can help to reduce the CPU overhead of the host processor to less than 2%.

Katcher *et al.* (1993) presented a methodology to incorporate the costs of scheduler implementation in fixed-priority scheduling algorithms. In particular, the study provided a framework to evaluate hardware and software implementation decisions for real-time applications based on quantitative results about implementation costs such as *blocking*

and *overhead*[30]. The proposed methodology was used to compare the real-time performance, in terms of schedulability, of four generic scheduler implementations of a fixed-priority algorithm: two time-triggered and two event-triggered. When the different implementations were applied to two realistic task sets – corresponding to avionics and inertial navigation applications – different levels of schedulability utilisation were obtained. This work was described as a first step toward bridging the gap between real-time scheduling theory and its implementation in real systems.

Later on, Mooney (1999) described one way of implementing a custom run-time scheduler, which dynamically executes tasks in different orders based on the conditional execution path, by using a hardware-software co-design. Along with a real-time analysis tool, the study demonstrated how the suggested implementation helps the system meet its relative timing, control-flow, and rate constraints.

In (Huajin *et al.*, 2002), implementing the classical Round Robin scheduling algorithm on Xilinx Field Programmable Gate Array (FPGA) chip using Verilog Hardware Description Language (VHDL) was discussed. Huajin noted that Round Robin is a very simple and widely used scheduling algorithm in computer systems where tasks are placed in a circular queue and executed in order starting from the first task in the queue. Moreover, it was emphasised that Round Robin algorithm is a time-triggered, pre-emptive scheduler in which each task in the system is allocated one time unit (quantum) to execute, and if the task is not completed at the end of the allocated slot the CPU is pre-empted and the current task is added to the tail of the queue, and so on.

Golatowski *et al.* (2002) presented a framework (with appropriate tools) to help the developers select the appropriate algorithm for their real-time application, among various kinds of dynamic scheduling algorithms (i.e. EDF, LLF), and then choose the

[30] Overhead and blocking are implementation costs which are a function of the underlying hardware. Overhead is the time spent in the kernel performing a service on behalf of a specific task, such as invoking or terminating it. Blocking, or priority inversion, is time spent, either in the kernel or in an application task, when a higher priority task is prevented from running (Katcher *et al.*., 1993).

best hardware / software implementation method for the selected scheduler based on the schedulability analysis of the scheduled task set.

Brinkschulte *et al.* (2002) considered the design and implementation of a real-time scheduling algorithm, called Guaranteed Percentage (GP) scheme, in which each thread is assigned a specific guaranteed percentage of the processor power and the threads are executed in isolation: i.e. threads have no influence on each others. The study demonstrated that when compared with Fixed-Priority Pre-emptive (FPP), Earliest Deadline First (EDF) and Least Laxity First (LLF) scheduling algorithms (all implemented on a *Komodo* microcontroller that features a multithreaded Java processor kernel), the GP scheduling was the only scheme that provided a strict isolation between threads: such an isolation advantage is required to maximise dependability in real-time systems. The results also showed that the hardware implementation costs of the GP scheduler were still reasonable.

Samuelsson *et al.* (2003) presented a performance comparison between a real-time kernel implemented in hardware and an equivalent one implemented in software using a multi-processor hardware platform. The hardware kernel implemented the scheduler, inter-process communication methods, semaphores and timer.

Cho *et al.* (2005) described an approach to implement static scheduler in multi-processor System-on-Chip (SoC). The work introduced efficient hardware and software scheduler architectures and considered the centralised[31] implementation versus distributed implementation of the schedulers. The trade-offs between both types of scheduler implementation was investigated according to area- and scheduler-overhead.

In a study carried out by Silva *et al.* (2005), it was argued that trade-offs between software and hardware implementations of a system are very important to achieve

[31] In centralised implementation, the scheduler is implemented on a single processor, whereas in distributed implementation the scheduler is implemented over multiple processors with one local scheduler for each processor.

flexibility as well as high-performance. The paper considered the implementation of a task scheduler for a real-time embedded system, as defined by the Real-Time Specification for Java (RTSJ), in both hardware and software. The study concluded that, if hardware implementation is used (using co-processor), task latencies can be reduced (regardless the number of scheduler tasks in the system) and the system predictability can be improved. However, such enhancement was at the cost of area-overhead.

Similarly, Vetromille *et al.* (2006) claimed that distributing the critical operating system functionalities between hardware and software implementations can have the potential to improve the overall performance and increase predictability of the real time systems. Therefore, Vetromille *et al.* evaluated the process of migrating RTOS scheduler implementation from software to hardware by considering the pros and cons of three different scheduler implementations as follows: (i) software using a single processor; (ii) software partitioned using two processors; and (iii) hardware / software partitioned using a processor and a dedicated hardware block. It was noticed that scheduler implementation (iii) always achieves better performance results, but is more complex and expensive compared to the other approaches due to the complex nature of the hardware implementation. The study concluded that (ii) and (iii) present the best results for hard real-time applications, where (i) is suitable for soft real-time systems.

In another study, Baruah (2006) presented sufficient conditions for determining whether a given periodic task system will meet all deadlines if non-pre-emptive EDF scheduler implementation is used upon a multi-processor platform. Baruah came to conclude that, if particular conditions are met, non-pre-emptive EDF scheduler implementations can provide the level of performance expected from the pre-emptive scheduler alternatives. Moreover, Baruah made a note that as faster processors become available, non-pre-emptive scheduling would become more popular in the future.

It can be clearly noticed that the outlined studies on scheduler implementation have not looked at the various possible ways in which the software of a given scheduler (or scheduling algorithm) can be implemented in low-cost “commercial of the shelf” COTS microcontroller platforms, and the impact of the various software implementation methods on the run-time behaviour of the systems. The studies presented in this thesis attempt to address such issues.

Appendix D

Additional set of TTC scheduler implementations

TTC-SL scheduler

Introduction

The simplest practical implementation of a TTC scheduler can be created using a "Super Loop" (sometimes called an "endless loop: Kalinsky, 2001). The particular implementation discussed in this section has been adapted from that described in detail elsewhere (Pont, 2001; Kurian and Pont, 2007).

Overview of the scheduler implementation

In Chapter 5, a simple TTC scheduler implementation – using a form of super loop – was outlined (Listing 5-2). Such a system assumed that each task executed would always have 4 ms duration, therefore a system with 10 ms tick interval was implemented using super loop and delay function (see Figure D-1).

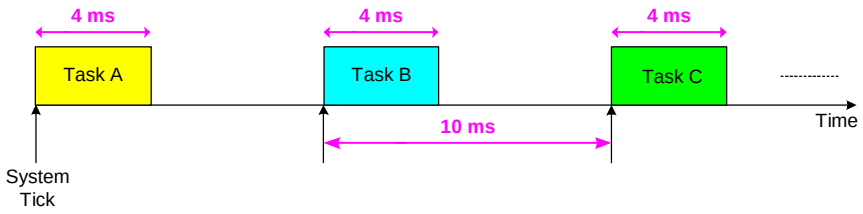


Figure D-1: The task executions resulting from the code in Listing 5-2.

In the case where the scheduled tasks have variable durations, creating a fixed tick interval is not straightforward. One way of doing that is to use a "Sandwich Delay" (Pont *et al.*, 2006) placed around the tasks. Briefly, a Sandwich Delay (SD) is a mechanism – based on a hardware timer – which can be used to ensure that a particular code section always takes approximately the same period of time to execute. The SD operates as follows:

- A timer is set to run.
- An activity is performed.

- The system waits until the timer reaches a pre-determined count value.

In these circumstances – as long as the timer count is set to a duration that exceeds the WCET of the sandwiched activity – SD mechanism has the potential to fix the execution period.

Listing D-1 shows how the tasks in Listing 5-2 can be scheduled – again using a 10 ms tick interval – if their execution durations are not fixed.

```
int main(void)
   {
   ...

   while(1)
      {
     // Set up a Timer for sandwich delay
     SANDWICH_DELAY_Start();

     // Add Tasks in the first tick interval
     Task_A();

     // Wait for 10 millisecond sandwich delay
     // Add Tasks in the second tick interval
     SANDWICH_DELAY_Wait(10);
     Task_B();

     // Wait for 20 millisecond sandwich delay
     // Add Tasks in the second tick interval
     SANDWICH_DELAY_Wait(20);
     Task_C();

     // Wait for 30 millisecond sandwich delay
     SANDWICH_DELAY_Wait(30);
      }
   // Should never reach here
   return 1
   }
```

Listing D-1: A TTC scheduler which executes three periodic tasks with variable durations, in sequence.

Using the code listing shown, the successive function calls will take place at fixed intervals, even if these functions have large variations in their durations (Figure D-2).

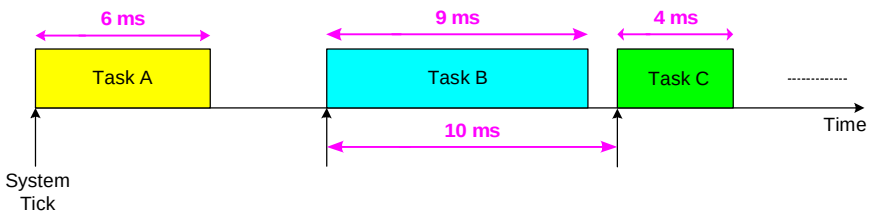


Figure D-2: The task executions expected from the TTC-SL scheduler code shown in Listing D-1.

Original TTC-Dispatch scheduler

Introduction

An early implementation of TTC scheduler, using Dispatch approach, was developed in the ESL group back in 2001. The architecture of such a scheduler provided the basis for TTC implementations which were developed later in the group. The particular implementation discussed in this section has been fully described and documented in (Pont, 2001).

Overview of the scheduler implementation

As in the TTC-Dispatch scheduler (described in Chapter 5), the implementation considered in this section is characterised by distinct and well-defined scheduler functions (see Listing 5-3). The original TTC-Dispatch scheduler is also driven by periodic interrupts generated from an on-chip timer. When an interrupt occurs, the processor executes an `Update()` function (see Listing 5-5). In the `Update()` function, the scheduler checks the status of all tasks to see which tasks are due to run and sets appropriate flags. After these checks are complete, a `Dispatch()` function (Listing 5-6) will be called, and the identified tasks (if any) will be executed in sequence. The `Dispatch()` function here is also called from an "endless" loop placed in the `Main` code (Listing 5-7) and when not executing the `Update()` and `Dispatch()` functions, the system will usually enter a low-power ("idle") mode.

Again, the scheduler implements a `SCH_Add_Task()` and a `SCH_Delete_Task()` functions for adding or removing tasks during the system run-time. In the `Update()` function, the scheduler applies checking on each task's parameters (i.e. task's offset and period) and consequently sets `RunMe` flag to indicate that the checked task is ready to execute in the current tick interval.

Code for the original TTC-Dispatch considered in this section is shown in the following listings.

```
void SCH_Update(void)
   {
   int Index;

   // NOTE: calculations are in *TICKS* (not milliseconds)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      // Check if there is a task at this location
      if (SCH_tasks_G[Index].pTask)
         {
         if (SCH_tasks_G[Index].Delay == 0)
            {
            // The task is due to run
            SCH_tasks_G[Index].RunMe += 1;  // Inc. the 'RunMe' flag

            if (SCH_tasks_G[Index].Period)
               {
               // Schedule regular tasks to run again
               SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period - 1;
               }
            }
         else
            {
            // Not yet ready to run: just decrement the delay
            SCH_tasks_G[Index].Delay -= 1;
            }
         }
      }
   // After interrupt, reset interrupt flag (by writing "1")
   T0IR = 0x01;
   }
```

Listing D-2: "Update" ISR of the original TTC-Dispatch scheduler.

```
void SCH_Dispatch_Tasks(void)
   {
   int Index;

   // Dispatches (runs) the next task (if one is ready)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      if (SCH_tasks_G[Index].RunMe > 0)
         {
         (*SCH_tasks_G[Index].pTask)();  // Run the task

         SCH_tasks_G[Index].RunMe -= 1;   // Reset / reduce RunMe flag

         // Periodic tasks will automatically run again
         // - if this is a 'one shot' task, remove it from the array
         if (SCH_tasks_G[Index].Period == 0)
            {
            SCH_Delete_Task(Index);
            }
         }
      }

   // The scheduler enters idle mode at this point
   SCH_Go_To_Sleep();
   }
```

Listing D-3: Dispatch function of the original TTC-Dispatch scheduler.

Adding “Sandwich Delays”

Introduction

The timing performance of the tasks running in the TTC scheduler can be improved by adding “sandwich delays”. This approach is introduced in this section and will be referred to as TTC-SD scheduler.

Overview of the scheduler implementation

In Chapter 2, the impact of task placement on “low-priority” tasks running in TTC schedulers have been considered. One way to reduce the variation in the starting times of such tasks is to place “Sandwich Delay” (Pont *et al.*, 2006) around tasks which execute prior to other tasks in the same tick interval.

In the TTC-SD scheduler, sandwich delays are used to provide execution “slots” of fixed sizes in situations where there is more than one task in a tick interval. To clarify this, consider the set of tasks shown in Figure D-3. In the figure, the required SD prior to Task C – for low jitter behaviour – is equal to the WCET of Task A plus the WCET of Task B. This implies that in the second tick (for example), the scheduler runs Task A and then waits for the period equals to the WCET of Task B before running Task C. The figure shows that when SDs are placed around the tasks prior to Task C, the periods between successive runs of Task C become equal and hence jitter in the release time of this task is significantly reduced.

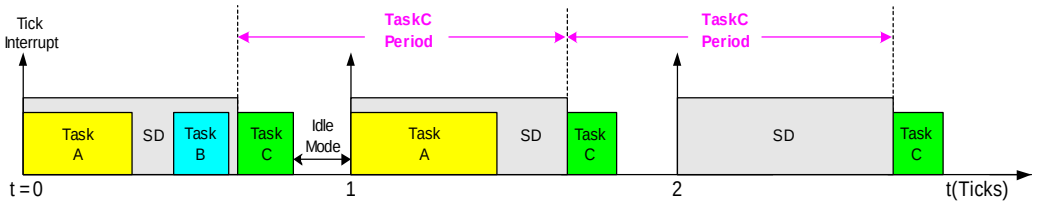


Figure D-3: Using Sandwich Delays to reduce release jitter in TTC schedulers.

Note that – with this implementation – the WCET for each task is input to the scheduler through a `SCH_Task_WCET()` function placed in the `Main` code. After entering task parameters, the scheduler employs `Calc_Sch_Major_Cycle()` and `Calculate_Task_RT()` functions to calculate the scheduler major cycle and the

required release time for the tasks, respectively. The release time values are stored in the "Task Array" using the variable `SCH_tasks_G[Index].Rls_time`.

Code for the TTC-SD scheduler is shown in the following listings.

```
int main (void)
   {
   ...

   // Add tasks
   // Delay and Period values are in *ticks*
   SCH_Add_Task(Task_A, 0, 1);
   ...

   // Input duration for tasks
   // Values are in *microseconds*
   SCH_Task_WCET(Task_A, 2000);
   ...

   // Calculate the Scheduler Major Cycle
   Calc_Sch_Major_Cycle(SCH_MAX_TASKS);

   // Calculate the required release time for each task
   Calculate_Task_RT();

   // Start the scheduler
   SCH_Start();

   // The scheduler may enter idle mode at this point (if used)
   SCH_Go_To_Sleep();

   return 0;
   }
```

Listing D-4: "Main" function in the TTC-SD scheduler.

```
void SCH_Dispatch_Tasks(void)
   {
   int Index;
   int Update_required = 0;

   // Delay margin added to compensate for scheduler overhead
   int Delay_Margin ;

   // Set up Timer 1 for sandwich delay
   SANDWICH_DELAY _Start();

   // Need to check for a timer interuppt since this
   // function was last executed (in case idle mode is not being used)

   // Disable timer interrupt
   VICIntEnClr = 0x10;

   if (Tick_count_G > 0)
      {
      Tick_count_G--;
      Update_required = 1;
     }

   // Re-enable timer interrupts
   VICIntEnable = 0x10;

   while (Update_required)
      {
     // Go through the task array
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
         {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
                  //if(Index>0)
                  {
                  Delay_Margin = 20*Index;

                  // Wait for the required sandwich delay

                 SANDWICH_DELAY_Wait(SCH_tasks_G[Index].Rls_time+Delay_Margin);
                  }

                 // The task is due to run
                 (*SCH_tasks_G[Index].pTask)();  // Run the task

                 if (SCH_tasks_G[Index].Period != 0)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
               }
            }
         }

      // Disable timer interrupt
      VICIntEnClr = 0x10;

      if (Tick_count_G > 0)
        {
         Tick_count_G--;
         Update_required = 1;
       }
      else
```

```
        {
        Update_required = 0;
        }

    // Re-enable timer interrupts
     VICIntEnable = 0x10;
     }

// Stop then reset SD Timer
T1TCR &= ~0x01;
T1TC = 0;

// The scheduler may enter idle mode at this point (if used)
SCH_Go_To_Sleep();
}
```

Listing D-5: Dispatch function of the TTC-SD scheduler.

The code presented in Listing D-5 shows that a SD was placed around each scheduled task, and only when the SD matches the value of the required "release time" of a task (e.g. `SCH_tasks_G[Index].Rls_time`) is the task executed. Note that the required release time of a task is the time between the start of the tick interval and the start time of the task "slot" plus a little safety margin.

Results

Applying STC to the TTC-SL scheduler

This section discusses the implementation of STCs in the TTC-SL scheduler and presents the output results from such an implementation.

Implementing the test cases

Implementing STC A and STC B with the TTC-SL scheduler can be straightforward (and very similar to the example shown in Listing D-1). The following two listings show how STC C and STC D were implemented, respectively, using a TTC-SL scheduler.

```
int main(void)
   {
   ...
   while(1)
      {
     // Set up Timer 1 for sandwich delay
     SANDWICH_DELAY_T1_Start();

     // Add Tasks in the first tick interval
     // Task B executes in approx 2 and 1/2 ticks
     Task_B();

     // Wait for 5 millisecond sandwich delay
      // Since Task B exceeds the 5ms tick, the scheduler goes to run the tasks
       in Tick 2 straight away
     SANDWICH_DELAY_T1_Wait(5);

     // Add Tasks in the second tick interval
     Task_A();
     Task_C();

     // Wait for 10 millisecond sandwich delay
     SANDWICH_DELAY_T1_Wait(10);

     // Add Tasks in the third tick interval

     // Wait for 15 millisecond sandwich delay
     SANDWICH_DELAY_T1_Wait(15);

     // Add Tasks in the fourth tick interval
     Task_A();
     Task_C();

     // Wait for 20 millisecond sandwich delay
            SANDWICH_DELAY_T1_Wait(20);
     }

   return 1;  // Should never reach here ...
   }
```

Listing D-6: One way of implementing STC C using the TTC-SL scheduler.

```
int main(void)
   {
   ...
   while(1)
      {
     // Set up Timer 1 for sandwich delay
     SANDWICH_DELAY_T1_Start();

     // Add Tasks in the first tick interval
     // Task A executes will overrun for 10 ticks
     Task_A();
      Task_B();

     // Wait for 5 millisecond sandwich delay
     SANDWICH_DELAY_T1_Wait(5);

     for(i=2; i<=20; i++)
       {
       // Add Tasks in the next tick interval
       Task_B();

       // Wait for 5 millisecond sandwich delay
       SANDWICH_DELAY_T1_Wait(i*5);
       }
     }

   return 1;  // Should never reach here ...
   }
```

Listing D-7: One way of implementing STC D using the TTC-SL scheduler.

Task sequencing and overrun behaviour

The sequence behaviour of the TTC-SL scheduler when applying STC A, STC B, STC C and STC D is summarised in the following table.

Table D-1: Task schedule in TTC-SL scheduler.

STC	Scheduler behaviour
A	A1
B	B1
C	C1
D	D1b

The results in Table D-1 show that – as would be expected – the TTC-SL scheduler performs the standard scheduler tests (STC A, STC B and STC C) without problems. However, when executing STC D, once the overrunning task (Task A) completes, the scheduler performs all missing executions for Task B (in this case, 10 executions), before continuing to serve the tasks in the following ticks. This means that the behaviour of the SL scheduler with STC D is very much similar to that obtained with the Dispatch scheduler (Section 7.3.2.2) in the sense that the system can “catch up” in

the event of error: see Figure D-4. Please note that there can be various other possible ways to implement the SL scheduler which might not be able to provide such behaviour.

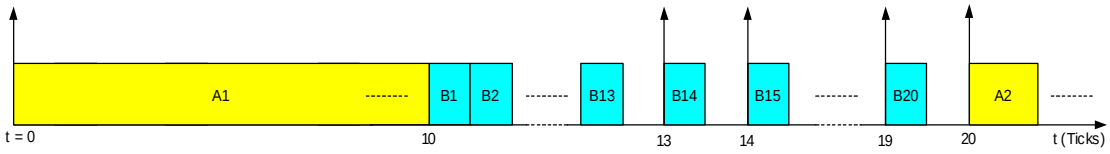


Figure D-4: The behaviour of SL scheduler with STC D (D1b schedule class).

Jitter

Table D-2 shows the periods and jitter measurements for the tick and the tasks for STC A, STC B and STC C when implemented using the TTC-SL scheduler.

Table D-2: Task jitter from the TTC-SL scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	3029.9	2409.1
	Max Period	5000.5	5000.5	6953.6	7368.8
	Average Period	5000.1	5000.1	4935	4836.8
	Diff. Jitter	0.8	0.8	3923.7	4959.7
	Avg. Jitter	0.2	0.2	836.8	900.8
Test B	Min Period	4999.8	10000.1	2993.9	2100.8
	Max Period	5001	10001.6	7010.1	7873
	Average Period	5000.5	10000.9	4923.7	4947.2
	Diff. Jitter	1.2	1.5	4016.2	5772.2
	Avg. Jitter	0.3	0.3	1179	1248.6
Test C	Min Period	972.5	2991.9	20004.4	2991.6
	Max Period	12012.8	17012.8	20004.5	17013.2
	Average Period	2416.1	5184.8	20004.5	5344.5
	Diff. Jitter	11040.3	14020.9	0.1	14021.6
	Avg. Jitter	2159.7	5093.3	0	5240.2

The jitter values in STCs A and B show that with a Super Loop scheduler, it is difficult to obtain zero jitter in the release time of the tick, although the tick jitter can still be very low. Results also show that when the scheduler major cycle had more than one tick (as in STC B) the ‘tick’ and ‘Task A’ jitter values have slightly increased. It can also be

shown that low-priority tasks always suffer high jitter in their release times when they are scheduled to run later in the tick interval. In situations where a task required multiple ticks to execute (as in STC C), the resulting tick jitter has significantly increased. Note that the tick interval in Test C is not fixed to 5 ms as required: instead, it varies between 12, 2, 1 and 5 ms in each major cycle.

CPU, memory and power requirements

Table D-3 shows the CPU overhead for the TTC-SL scheduler (with STC A). From the results shown in the table, the TTC-SL scheduler always requires a full CPU load (~ 100%). This is since the scheduler does not use the low-power "idle" mode when not executing tasks: instead, the scheduler waits in a "while" loop.

Table D-3: CPU overhead for the TTC-SL scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	25.01	25.01	100

Table D-4 summarises the memory required to implement STC A using the TTC-SL scheduler.

Table D-4: Memory requirements (ROM and RAM) for the TTC-SL scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	2264	124

Table D-5 shows the power consumption levels from the STC A and STC B.

Table D-5: Power requirements for the TTC-SL scheduler.

Method	Power consumption (mW)
Test A	62.1
Test B	65.3

The results in the table demonstrate very high levels of CPU power consumption. This is again caused by the inefficient use of the processor: that is, when no tasks are executed, the processor is not sent to sleep (i.e. placed in the low-power "idle" mode).

Applying STC to the original TTC-Dispatch scheduler

This section discusses the implementation of STCs in the original TTC-Dispatch scheduler and presents the output results from such an implementation.

Implementing the test cases

Implementing the STCs is similar to that with the TTC-Dispatch scheduler (Chapter 7).

Task sequencing and overrun behaviour

The sequence behaviour of the original TTC-Dispatch scheduler when applying STC A, STC B, STC C and STC D is summarised in the following table.

Table D-6: Task schedule in TTC-Dispatch scheduler.

STC	Scheduler behaviour
A	A1
B	B1
C	C3
D	D1a

When executing STC A and STC B, the original TTC-Dispatch scheduler behaves in the same way as the TTC-ISR and the TTC-Dispatch schedulers. When executing STC C, since the scheduler checks each task in sequence to see if they are due to run, Task C's status is tested – after Task B – and the task is executed. The scheduler then enters the "idle" mode waiting for a timer interrupt. This means that the first execution of Task A is omitted from the schedule. The system then continues as normal. In STC D, the scheduler does not have a mechanism which counts the missing ticks, therefore tasks which are due to run in these ticks are totally ignored.

Jitter

Table D-7 shows the periods and jitter measurements for the tick and the tasks for STC A, STC B and STC C when implemented using the original TTC-Dispatch scheduler.

Table D-7: Task jitter from the original TTC-Dispatch scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	3029.1	2480.5
	Max Period	4999.7	4999.7	6953.5	7595.1
	Average Period	4999.7	4999.7	4930.99468	4896.26064
	Diff. Jitter	0	0	3924.4	5114.6
	Avg. Jitter	0	0	824.6	911.9
Test B	Min Period	4999.7	9999.4	2990.1	2099
	Max Period	4999.7	9999.5	7011.6	7863.6
	Average Period	4999.7	9999.4849	4805.84748	4762.61818
	Diff. Jitter	0	0.1	4021.5	5764.6
	Avg. Jitter	0	0	1166.3	1204.7
Test C	Min Period	4999.5	4999.2	18991	2994.1
	Max Period	4999.9	14999.7	19998.9	17004.8
	Average Period	4999.7	7680.33856	19998.49684	5370.11318
	Diff. Jitter	0.4	10000.5	1007.9	14010.7
	Avg. Jitter	0.1	4429.8	20.2	5258

The jitter results obtained are similar to those obtained with the TTC-Dispatch scheduler (Chapter 7). For example, the table shows that Task A has consistently low (release) jitter levels while the jitter for Task B and Task C – which are of low priorities – is rather high in STC A and STC B.

However, it is very important to highlight that at any tick, the length of the check activities – and hence the Update ISR – is a function of the number of scheduled tasks to run at this tick. This results in varying the length of the ISR function from one tick to another. One consequence of this variation is that Task A will suffer from jitter in its release time when the tasks, to be scheduled in-phase with it, change from one tick to another. The developed STCs (presented in Chapter 6) do not illustrate this difference in behaviour between the two versions of the Dispatch scheduler.

In order to emphasise this behaviour, a small study was carried out in which a number of tasks (between one and ten) were scheduled in such a way that the impact of jitter would be maximised (Listing D-8 to Listing D-11). In this study, the release jitter for Task A was measured 10 times each with a different set of tasks. For example, in the

first experiment, only Task A was added to the system. In the second experiment, Task A and Task B were added. In the third experiment, Task A, Task B and Task C were added and so on. This was to explore the impact of the number of scheduled task on the jitter behaviour of Task A (which is implicitly the top priority task with hardest timing constraints).

```
// Add tasks in experiment 1 (5 ms ticks)
// Parameters are <task name>, <offset in ticks>, <period in ticks>
SCH_Add_Task(Task_A, 0, 10);
```

Listing D-8: Task list used in experiment 'one'.

```
// Add tasks in experiment 2 (5 ms ticks)
// Parameters are <task name>, <offset in ticks>, <period in ticks>
SCH_Add_Task(Task_A, 0, 10);
SCH_Add_Task(Task_B, 0, 9);
```

Listing D-9: Task list used in experiment 'two'.

```
// Add tasks in experiment 3 (5 ms ticks)
// Parameters are <task name>, <offset in ticks>, <period in ticks>
SCH_Add_Task(Task_A, 0, 10);
SCH_Add_Task(Task_B, 0, 9);
SCH_Add_Task(Task_C, 0, 8);
```

Listing D-10: Task list used in experiment 'three'.

```
// Add tasks in experiment 10 (5 ms ticks)
// Parameters are <task name>, <offset in ticks>, <period in ticks>
SCH_Add_Task(Task_A, 0, 10);
SCH_Add_Task(Task_B, 0, 9);
SCH_Add_Task(Task_C, 0, 8);
SCH_Add_Task(Task_D, 0, 7);
SCH_Add_Task(Task_E, 0, 6);
SCH_Add_Task(Task_F, 0, 5);
SCH_Add_Task(Task_G, 0, 4);
SCH_Add_Task(Task_H, 0, 3);
SCH_Add_Task(Task_I, 0, 2);
SCH_Add_Task(Task_J, 0, 1);
```

Listing D-11: Task list used in experiment 'ten'.

Table D-8: Task A jitter from the original TTC-Dispatch and the TTC-Dispatch schedulers (all values in µs).

		Experiment No.									
		One	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten
Original TTC-Dispatch	Min Period	4999.7	4998.6	4997.7	4996.8	4996.2	4995.8	4995.3	4994.9	4994.5	4994.1
	Max Period	4999.7	5000.8	5000.6	5000.9	5001.2	5001.5	5002.1	5002.4	5002.9	5003.7
	Average Period	4999.7	4999.6	4999.5	4999.7	4999.7	4999.8	4999.6	4999.7	4999.5	4999.7
	Diff. Jitter	0	2.2	2.9	4.1	5	5.7	6.8	7.5	8.4	9.6
	Avg. Jitter	0	1.1	1.4	1.6	1.9	2.2	2.6	2.8	3.2	3.6
TTC-Dispatch	Min Period	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7
	Max Period	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7
	Average Period	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7	4999.7
	Diff. Jitter	0	0	0	0	0	0	0	0	0	0
	Avg. Jitter	0	0	0	0	0	0	0	0	0	0

By analysing the jitter values of Task A in both schedulers, it was seen that, in the original TTC-Dispatch scheduler, the jitter levels increased as further tasks were scheduled to run in the system, while the jitter levels were always constant in the TTC-Dispatch scheduler.

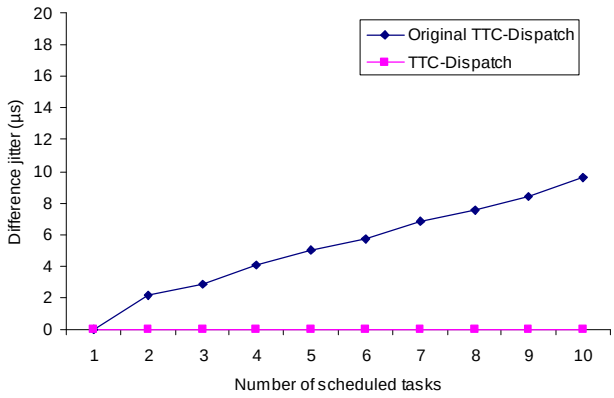


Figure D-5: "Task A" release jitter in the original TTC-Dispatch and the TTC-Dispatch schedulers based on the study shown in Listing D-8 to Listing D-11.

The observed behaviour patterns are caused mainly by the architecture of the system. For example, in the original TTC-Dispatch implementation, the scheduler first determines – in the `Update()` function – which tasks are due to execute and sets the corresponding flags. The system will then execute the flagged tasks from the `Dispatch()` function. A consequence of this arrangement – as previously noted – is that the scheduler overhead (the `Update()` function duration) will vary depending on the number of tasks that are to be implemented in a given tick interval. This means that all tasks (even the first task to be executed which is Task A in this case) will suffer release jitter.

The TTC-Dispatch implementation (described in Chapter 7) controls the jitter in the first task by re-arranging the activities performed in the `Update()` and `Dispatch()` functions, as illustrated in Listing 5-5 and Listing 5-6, respectively. In such an implementation, the `Update()` function is very short and has a fixed duration: it simply keeps track of the number of Ticks. The dispatch activities will then be carried out in the `Dispatch()` function.

CPU, memory and power requirements

Table D-9 shows the CPU overhead for the original TTC-Dispatch scheduler (with STC A).

Table D-9: CPU overhead for the original TTC-Dispatch scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	9.9	25.0	39.8

Table D-10 presents the memory required to implement STC A using the original TTC-Dispatch scheduler.

Table D-10: Memory requirements (ROM and RAM) for the original TTC-Dispatch scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	4112	324

Table D-11 shows the power consumption levels for STC A and STC B when implemented using the original TTC-Dispatch scheduler.

Table D-11: Power requirements for the original TTC-Dispatch scheduler.

Method	Power consumption (mW)
Test A	38.9
Test B	35.7

Applying STC to the TTC-SD scheduler

This section discusses the implementation of STCs in the TTC-SD scheduler and presents the output results from such an implementation.

Implementing the test cases

Implementing the STCs is similar to that with the TTC-Dispatch scheduler (Chapter 7).

Task sequencing and overrun behaviour

The sequence behaviour of the TTC-SD scheduler when applying STC A, STC B, STC C and STC D is summarised in Table D-12.

Table D-12: Task schedule in TTC-SD scheduler.

STC	Scheduler behaviour
A	A2
B	B2
C	C1
D	D1b

It can be clearly seen from the table that – like the MTI approach – using the SD approach helps to reduce the variation in the starting times of tasks running in a TTC system (as in STC A and STC B). In the STC C and STC D, the system behaves in the same way as the TTC-Dispatch scheduler (Chapter 7). This is because the SD scheduler is mainly adapted from the Dispatch scheduler with a little modification.

Jitter

Table D-13 shows the periods and jitter measurements for the tick and the tasks for STC A, STC B and STC C when implemented using the TTC-SD scheduler.

Table D-13: Task jitter from the TTC-SD scheduler (all values in µs).

		Tick	Task A	Task B	Task C
Test A	Min Period	4999.7	4999.7	4999	4999
	Max Period	4999.7	4999.7	5000.5	5000.5
	Average Period	4999.7	4999.7	4999.7	4999.7
	Diff. Jitter	0	0	1.5	1.5
	Avg. Jitter	0	0	0.3	0.3
Test B	Min Period	4999.7	9999.4	4999	4999
	Max Period	4999.7	9999.5	5000.5	5000.5
	Average Period	4999.7	9999.5	4999.8	4999.7
	Diff. Jitter	0	0.1	1.5	1.5
	Avg. Jitter	0	0	0.4	0.3
Test C	Min Period	4999.6	2978.4	19998.9	2978.2
	Max Period	4999.9	17020.5	19998.9	17020.6
	Average Period	4999.7	5312.2	19998.9	5427.3
	Diff. Jitter	0.3	14042.1	0	14042.4
	Avg. Jitter	0	5227.8	0	5328.9

From the values presented in the table, the use of SD mechanism in TTC schedulers can help the low-priority tasks to execute at fixed intervals. This is clear in the results obtained from STC A and STC B. However, the results from these STCs still show little variation (i.e. jitter) in the release times of Tasks B and Task C. This jitter is caused by variation in time taken to leave the software loop – which is used in the SD mechanism to check if the required release time for the concerned task is matched – and begin to execute the task. In Listing D-12, one way of implementing such a SD mechanism is shown.

```
void SANDWICH_DELAY_T1_Wait(const unsigned int DELAY_MS)
  {
   // The timer is set so that one count equals to one microsecond
   int i = DELAY_MS;

   // Wait for Timer 1 count to reach delay
   while (T1TC < i)
     {
     ;
     }
  }
```

Listing D-12: An example of "sandwich delay" function used in the TTC-SD scheduler.

CPU, memory and power requirements

Table D-14 shows the CPU overhead for the TTC-SD scheduler (with STC A).

Table D-14: CPU overhead for the TTC-SD scheduler.

	Scheduler time (s):	Total time (s):	Overhead %
Test A	18.5	25.0	74.0

The CPU overhead results show that the overall processing time required for the TTC-SD scheduler is equal to 74% of the total run-time. This overhead figure is too large compared to that obtained from the most of the schedulers considered in this thesis (which was approximately equal to 39%). The observed increase in processing time is expected when such a SD approach is used: since the CPU is forced to run in normal operating mode while waiting for tasks to start their execution. Nonetheless, this CPU overhead can still be low compared to that required to implement the TTC-SL scheduler in which the processor is not placed in low-power idle mode under any condition.

Table D-15 presents the memory requirements for implementing the STC A for the TTC-SD scheduler.

Table D-15: Memory requirements (ROM and RAM) for the TTC-SD scheduler.

Method	ROM requirements (Bytes)	RAM requirements (Bytes)
Test A	5344	310

Table D-16 shows the power consumption levels from the STC A and STC B when implemented using the TTC-MTI scheduler.

Table D-16: Power requirements for the TTC-SD scheduler.

Method	Power consumption (mW)
Test A	54.4
Test B	54.5

The results in the table show that with SD scheduler, the CPU power consumption is significantly increased. This is, again, because the processor runs in normal operating mode while the SD is executing. Note that the power consumption levels in STC A and STC B, when SD is employed, are equal. This is because whether or not Task A is scheduled, the processor has to operate for the same duration until Task C (the last task in the list) completes execution (see STC A and STC B in Chapter 6).

In order to eliminate jitter completely from the release time of tasks in a TTC scheduler while reducing power consumption, the modified sandwich delay mechanism (described in Section 5.7) which employs “multiple timer interrupts” (MTIs) is recommended.

Appendix E

Techniques for reducing jitter in S-C schedulers

Introduction

This appendix reviews key previous work carried out to reduce jitter in systems using CAN-based networks including the TTC-SCC scheduling protocol. The main focus will, however, be on data coding techniques developed in this project to provide simple and cost-effective solutions to jitter problem in such embedded system architectures.

General jitter-reduction techniques

Generally, there has been a great deal of previous work to address jitter problem in systems implemented using CAN network. For example, ways for bounding the response time of CAN messages to reduce the impact of jitter have been explored in a number of studies (e.g. Tindell *et al.*, 1995; Navet and Song, 1998; Rudiger, 1998). To reduce clock jitter in CAN systems, many studies proposed techniques which help to adjust clocks in the communicating processors (e.g. see Verissimo and Rodrigues, 1992; Rodrigues *et al.*, 1998; Lee and Allan, 2003; Johansson *et al.*, 2005). Barreiros *et al.* (2000) and Coutinho *et al.* (2000) applied ability of genetic algorithms which is a search technique to manage the schedule of message transmission in order to minimise jitter levels.

Nonetheless, if data messages are used to drive the local time base of each other node (as with TTC-SCC protocol), then encoding message data (before transmission) can be a cost-effective approach to reduce jitter by significant factors whilst maintaining high resource efficiency.

Data coding techniques

Introduction

This section describes a range of effective software-based techniques which can be integrated in TTC-SCC1, TTC-SCC2 and TTC-SCC3 (or any CAN-based) networks for reducing jitter caused by the underlying network protocol. In this case, the jitter is observed by bit stuffing mechanism implemented in the CAN hardware for clock

synchronisation. The described techniques are based on generic data coding approaches which can be adapted for use in a wider range of data applications. Such techniques include: XOR masking, Software Bit Stuffing (SBS), and Eight-to-Eleven Modulation (EEM). Ways of implementing each of these techniques in practical designs, using TTC-SCC1 scheduler, are explored and fully documented[32].

It must be emphasised that in all these techniques, data "encoding" and "decoding" activities in the Master and Slave nodes, respectively, are performed in the scheduler *slack time*: that is the spare processing time during which the scheduler is in its idle state (Davis, 1993), see Figure E-1. The reason for this is to avoid any jitter that may be caused by variations in the execution times of such coding activities which are a complex function of the original data-bit values. This means that, at any tick interval, frame that was encoded in the slack time of the previous tick interval is transmitted. Note that code listings for all techniques are presented in Appendix H.

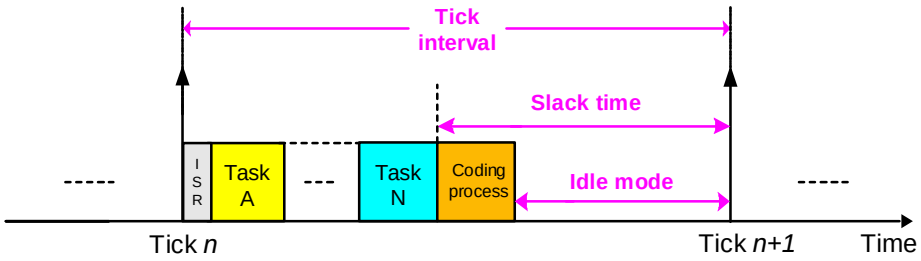


Figure E-1: Tick structure in all coding techniques considered in this thesis.

Masking data using XOR transformation

Introduction

This section explores the benefits of a method proposed originally by Nolte *et al.* (2001 and 2002) for reducing the impact of (hardware) bit stuffing in CAN networks. Modifications to this technique – for achieving better improvement to a wider range of real-time applications – are then discussed and evaluated.

[32] The work described in this appendix has been adapted from the studies presented in the author's publications [7] and [9] listed in page xvii.

Overview of the technique

When attempting to reduce message-length variations (and hence transmission jitter) to low levels without imposing large computational or memory overheads, techniques described by Nolte *et al.* (2001 and 2002) appear to be attractive. Nolte and colleagues (2001 and 2002) have described two mechanisms which can be combined to reduce the impact of bit stuffing in networks employing the CAN protocol. The first approach was based on a careful selection of message priorities aimed to remove the bit stuffing effect in the frame header (i.e. Arbitration and Control fields). The second approach they considered was based on an exclusive-OR (XOR) bit masking applied to the data section of each CAN frame.

When applying both techniques to a particular set of test data, Nolte and colleges have found that worst-case number of stuffed bits in CAN messages was reduced from 17 to 4. This is further discussed as follows.

Nolte XOR transformation

By analysing 25000 CAN frames from a real automotive system, Nolte *et al.* (2001) found that the probability of having bit value of 1 (or 0) in the data section is not 50% as usually assumed in traditional models. More specifically, they observed that the probability of having consecutive bits of the same polarity was high, and that – therefore – the number of stuffed bits is higher than would be expected with random data.

To reduce the number of stuffed bits inserted by the CAN hardware, Nolte suggested a simple encoding scheme based on logical exclusive-OR (i.e. XOR) operation. In this scheme, the data section of each CAN frame is XOR-ed with alternating ones and zeros (i.e. 101010…). At the receiving end, the same bit operation is applied again to extract the original data (see Figure E-2).

Original frame: 000000111110011000000111 ...
XOR with bit-mask: 101010101010101010101010 ...
Transmitted frame: 101010010100110010101101 ...

Figure E-2: Encoding process in Nolte XOR masking.

When applying such an XOR masking technique to the data set considered in their study (along with the message-ID selection technique), Nolte *et al.* found that the worst-case number of stuffed bits in CAN messages was reduced by approximately 76%.

However, in a more general case, the data transmitted may not have the same characteristics as those observed by Nolte. Provided that message identifiers were selected properly, it would not be expected to see a significant reduction in the level of bit-stuffing if the Nolte (XOR) transformation is applied to a data field containing random bytes. This is investigated in the next section.

Applying Nolte transformation to general CAN traffic

Overall, the application of an XOR transformation can help to reduce levels of bit stuffing in frames which are found to contain long sequences of identical bits. In a more general case, the data transmitted may not have the same pattern. Indeed, if a completely general CAN message was modelled using random data, then it would not be expected to see a significant reduction in the level of bit-stuffing when the Nolte (XOR) transformation is applied.

To illustrate this, 10 million pseudo-random data frames – each with eight data bytes – were created and analysed using a 'C' program. The results from a simple analysis of these data are presented in Table E-1.

Table E-1: Bit stuffing results from random CAN frames.

No. of frames exposed to CAN bit stuffing	Maximum number of stuffed bit	Average number of stuffed bit
8,932,166	10	2.27

The table shows that – of the 10,000,000 frames – a total of 8,932,166 (around 89%) would be subject to CAN bit stuffing. In this data set, the maximum number of stuffed bits (for any frame) was 10 and the average number of stuffed bits (across all frames) was 2.27.

Table E-2 then illustrates what happens if the Nolte approach is applied to all frames in the data set.

Table E-2: Results from Nolte XOR transformation technique applied to random CAN frames.

Bit-stuffed frames	Maximum. stuffed bits	Average stuffed bits	Reduction in frames	Reduction in max bits	Reduction in average bits
8,931,642	10	2.27	0.006%	0%	0%

In the table, "Reduction in frames" shows the reduction in the number of frames which are subject to bit stuffing after Nolte XOR transformation is applied: in this case, the result is small (0.006%). Similarly, the reductions in the maximum number of stuffed bits (0%) and the average number of stuffed bits (0%) are also small. Overall, it can be concluded that the direct application of Nolte transformation is having a minimal improvement on the level of bit stuffing for the random data.

Selective "frame-based" application of Nolte transformation

Using the same study, Table E-3 shows the results obtained in response to a selective application of the Nolte method. This table uses the same data set used in Table E-1. This time, however, the frames are tested individually before Nolte XOR transformation is applied: in situations where – for the whole frame – bit stuffing will not occur, the frame is transmitted unaltered. Only where bit-stuffing will be applied (to the "raw" frame) is the frame subject to an XOR transformation. This method will be referred to, in the remainder of the thesis, as "frame-based XOR transformation".

Table E-3: Results from frame-based XOR transformation applied to random CAN frames.

Bit-stuffed frames	Maximum. stuffed bits	Average stuffed bits	Reduction in frames	Reduction in max bits	Reduction in average bits
7,927,015	10	2.22	11.25%	0%	2.2%

In this case, it is noted that "Reduction in frames" and reduction in the average number of stuffed bits are larger after frame-based XOR transformation is applied. However, no reduction is obtained in the maximum number of stuffed bits.

Selective "byte-based" application of Nolte transformation

Here, Table E-4 shows the results obtained in response to a third implementation of the Nolte method. This table again uses the same data set. This time, however, each byte of data in each frame is tested individually before Nolte XOR transformation is applied: in situations where – for the byte – bit stuffing will not occur, the byte is transmitted

unaltered. Only where bit-stuffing will be applied is the byte subject to an XOR transformation. This method will be referred to, in the remainder of the thesis, as “byte-based XOR transformation”.

Table E-4: Results from byte-based XOR transformation applied to random CAN frames.

Bit-stuffed frames	Maximum. stuffed bits	Average stuffed bits	Reduction in frames	Reduction in max bits	Reduction in average bits
5,638,654	6	1.39	36.87%	40%	38.77%

In this case, against all the measures made here, it can be noted that there has been a reduction in the level of bit stuffing. In this case, the “Reduction in frames” is approximately 37%. The reductions in the maximum and average number of stuffed bits are at similar levels.

Implementation

From the small study described in the previous sections, the XOR transformation suggested by Nolte has – as expected – a little impact on the random data set. However, by applying Nolte transformation selectively (when required) the level of bit stuffing can be further reduced.

Of course, the study outlined by Nolte was highly artificial, and took no account of (for example) the need to transmit information about the encoding process to the receiver, to allow successful decoding of the data stream. In this section, the described XOR transformation methods are incorporated in the TTC-SCC1 scheduling protocol described in Chapter 9.

Implementing Nolte XOR transformation

In this method, every byte of the CAN data message (except the Slave ID) is XOR-ed with the bit pattern 10101010. In this method the maximum data bandwidth of eight bytes can be used for real data since there is no need to send any encoding information with the CAN messages.

Implementing the frame-based XOR transformation

In this method, each CAN frame is checked and – if a sequence of five identical bits is detected – the whole frame will be XOR-ed with the bit mask 10101010 ….

To allow decoding, only one bit is required to indicate if the frame is masked or not. To make best use of the available data bandwidth, one bit in Byte 1 (which otherwise contains the Slave ID) was used to store the masking information. Appropriate coding schemes were used to ensure that the bit stuffing was not introduced in the Slave ID byte (see Figure E-3).

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Slave ID	Slave ID	Masking info	Slave ID	Slave ID	Slave ID	Slave ID	Slave ID

Figure E-3: Layout for Byte 1 in the frame-based XOR transformation.

Implementing the byte-based XOR transformation

In this method, each CAN frame is checked on byte-by-byte basis, and once a byte contains a sequence of five identical bits is detected, this particular byte will be masked using Nolte bit-mask (i.e. 10101010).

To hold the masking information, one bit per each byte of data is required. In this case, where 6 bytes were used for data, six bits were needed. However, it was necessary to ensure that these 6 bits did not themselves introduce bit stuffing. Therefore – as with the frame-based method – one bit of the Slave-ID byte was used to store decoding information, along with 5 bits (and appropriate padding) in the last CAN data byte (see Figure E-4 to Figure E-6).

Byte 1	Byte 2 ---------- Byte 7	Byte 8
Slave ID + one bit for masking info	Actual data	Masking info

Figure E-4: Layout for data field in the byte-based XOR transformation.

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Opposite to the previous bit	Relevant to data byte 2	Relevant to data byte 3	Relevant to data byte 4	Opposite to Bit 4	Relevant to data byte 6	Relevant to data byte 7	Opposite to Bit 7

Figure E-5: Layout for Byte 8 in the byte-based XOR transformation.

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Slave ID	Slave ID	Relevant to data byte 5	Slave ID	Slave ID	Slave ID	Slave ID	Slave ID

Figure E-6: Layout for Byte 1 in the byte-based XOR transformation.

For example, if Byte 8 equals to: 01010110 and Byte 1 equals to: 00100010, the receiving node will know that bytes 2, 4, 5, 6 and 7 were masked. The Slave ID value was again selected with extra care to avoid exposure to hardware bit-stuffing.

To implement encoding and decoding processes of this method in practice, there can be two implementation options: (a) offline using lookup table, and (b) online using function call. Both approaches are explained here.

Lookup table

In this approach, byte checking and required XOR transformation are carried out in a separate desktop program for all different combinations of bits. Once all bytes are checked and hence the required bytes are masked, the equivalent codewords are stored in a one dimensional array that is used as a lookup table. For those bytes which need no masking, the equivalent codeword in the table will be equal to the original byte value. In the following table, the first few lines of the lookup table used in the byte-based XOR transformation are presented. Note that the byte values 8, 9 and 10 remain the same since they are not subject to XOR transformation.

Table E-5: Part of the byte-based XOR lookup table.

Original Byte value	Equivalent codeword (Decimal)	Equivalent Codeword (Binary)
0	85	01010101
1	84	01010100
2	87	01010111
3	86	01010110
4	81	01010001
5	80	01010000
6	83	01010011
7	82	01010010
8	8	00001000
9	9	00001001
10	10	00001010

The (embedded) Master and Slave codes will then port this lookup table and use it for encoding and decoding, respectively. However, in the Slave, the decoding function needs to search for the value – of each received byte – in the table and map it to its original byte value (which is represented by the array index).

Function call

In this approach, masking the required bytes (in the encoder) and de-masking them (in the decoder) are performed in the embedded code during the system runtime without prior information. For example, the Master code needs to check the data bytes and applies the XOR transformation when required. The receiver code should hence follow the reverse process to recover the original data bytes.

Software Bit Stuffing (SBS)

Introduction

This section presents an alternative software-based technique, called Software Bit Stuffing (SBS), which can be used in any CAN-based network to minimise the impact of the bit stuffing mechanism implemented in the CAN physical layer. It will be explained how such a technique can be implemented in practical designs, using TTC-

SCC1 scheduler, with the aim to reduce the message-length variations (and hence the transmission jitter) in CAN networks.

Overview of the technique

Although the byte-based XOR transformation was found to reduce the level of message-length variations, it is impossible to guarantee that all (hardware) bit stuffing will be avoided through application of this method. This is because in situations where there are five consecutive bits of the same polarity at the boundary of two adjacent bytes, this method will not detect this.

To completely eliminate the need for hardware bit stuffing in the CAN data segment, Software Bit Stuffing (SBS) technique is proposed. SBS operates as follows. Before transmitting on the CAN bus, the data content of a given frame is checked. If a sequence of four consecutive identical bits is detected, the algorithm adds an additional bit, of opposite polarity, afterwards. By doing so, the transmitted frames will have no bit-sections – in the data field – that will be subject to CAN bit stuffing. Note that the (software) stuffed bits must be removed at the receiving node (using the reverse process) to recover the original data.

Note that after completing the bit stuffing process in the sending node, the message is padded – as necessary – before transmission to ensure that the message length is independent of the level of (software) bit stuffing. Note that in the hardware bit-stuffing mechanism used in CAN, no such padding is employed.

Implementation

The previous section described a simple approach to software bit stuffing. This section describes how such a technique can be implemented on the TTC-SCC1 scheduling protocol.

The analysis in (Nolte *et al.*, 2001) demonstrates that the worst-case bit stuffing occurs if the CAN data contains five “0s” followed by four “1s” followed by four “0s”, and so on. The total number of bits transmitted in the CAN message is therefore calculated as:

$$No.of\ transmitted\ bits = No.of\ bits\ subject\ to\ bit\ stuffing + No.of\ stuffed\ bits$$

Equation E-1

The number of required stuffed bits is calculated as follows:

$$No.of\ stuffed\ bits = \frac{No.of\ bits\ subject\ to\ bit\ stuffing - 1}{Maximum\ No.of\ consecutive\ bits\ allowed - 1}$$

Equation E-2

Note that the maximum number of allowed consecutive bits to transmit is equal to five; since any sequence contains more than five consecutive identical bits will be broken up by the CAN physical layer using an opposite polarity bit.

In the software level, to avoid hardware bit stuffing, a maximum of four consecutive identical bits is allowed to transmit. Remember that in the S-C protocol, one byte is allocated for Slave ID, where the bits value of this byte are selected carefully such that (a) they were not subject to CAN bit stuffing, and (b) the last two bits in the Slave ID had opposite polarities.

Given that B_t is the total number of bits in the data field of the CAN frame, B_{ID} is the number of bits used for Slave ID, B_r is the number of bits used for real data (which will be subject to bit stuffing), and B_s is the number of stuffed bits, by substituting these parameters in Equation Equation E-1 and above, B_t can be formulated as:

$$B_t = B_{ID} + B_r + B_s$$

Equation E-3

Using Equation E-2, B_s will be equal to $\left[\frac{B_r - 1}{3}\right]$. If this term is substituted in Equation E-3, then B_r can be calculated as:

$$B_r = \frac{3(B_t - B_{ID}) - 1}{4}$$

Equation E-4

If B_{yr} is the number of bytes used for real data, then $B_r = 8\ B_{yr}$. By substituting this in Equation E-4, B_{yr} will be calculated as 5.3 Bytes. This means that the maximum bandwidth which can be used for transmitting real data is 5 bytes. This implies that the worst-case number of stuffed bits will be equal to 13.

Note that although the Slave ID byte was not subject to bit stuffing, there is still the possibility that there will be five consecutive bits of the same polarity at the boundary of this byte and the first byte in the real data. To avoid this possibility, software bit stuffing will be applied if the data starts with ***three*** identical bits.

After all the necessary message bytes have been checked and the required bits inserted, the remaining bits in the data section of the CAN message are padded with alternating ones and zeros (i.e. 1010…). These bits are called compensation bits.

The number of compensation bits required (B_c) can be calculated as follows:

$$B_c = B_t - B_{ID} - (B_r + B_s); \qquad 0 \leq B_c \leq 16$$

Equation E-5

In the system considered here where $B_{ID} = 8$, $B_t = 64$ (8 bytes), $B_r = 40$ (5 bytes), and $B_s = 13$, there will be at least three compensation bits needed to pad the message for fixing its length. If the real data are not subject to CAN bit stuffing then the last two "Stuffed coding" bytes (16 bits) in the data section of the CAN message will be filled with alternating ones and zeros (see Figure E-7).

Byte 1	Byte 2 – 6	Byte 7	Byte 8
Slave ID	Data	Stuff coding	Stuff coding

Figure E-7: Layout for data field in the software bit stuffing.

Using an example of pseudo-random data, the entire (encoding) process of SBS is schematically illustrated in Figure E-8. Note that the stuffed bits and the compensation bits are bolded.

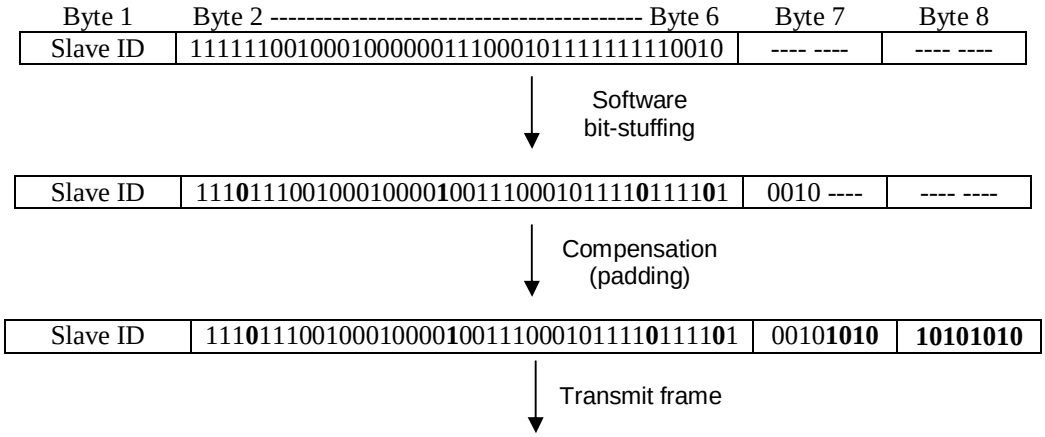


Figure E-8: Encoding process in the software bit stuffing algorithm.

In the same way, reverse (decoding) process must apply in all Slave nodes to extract the original five data bytes sent in the "Tick" message from the Master.

Eight-to-Eleven Modulation (EEM)

Introduction

In this section, Eight-to-Eleven Modulation (EEM) technique, which can also be used in CAN-based systems for reducing the impact of bit stuffing, is described. EEM is, however, found to be more flexible and cost-effective. The flexibility of this method results from the wide range of implementation options which are available to implement the algorithm in resource-constrained embedded microcontrollers. An exploration of various possible ways in which EEM can be implemented in practice, using TTC-SCC1 scheduler, is carried out in this section. Please note that the implementations outlined here have been viewed as representative of all possible implementations methods.

Overview of the technique

Despite the fact that SBS can help to substantially reduce the jitter levels in CAN systems, the data encoding / decoding processes can only be performed at run-time (while the system is in its normal operation) using a function call. Overall, processing data at run-time may result in increased CPU overheads on the used processor: a fact which may not be tolerated in many embedded systems that have extremely limited timing resources.

To reduce the impact of bit stuffing in CAN systems while achieving higher processor utilisation, an alternative coding technique, which is based on X-to-Y modulation

approach, is proposed. The X-to-Y modulation is a general coding method in which "X" represents the number of bits in the original data segment and "Y" represents the number of bits in the encoded data segment. Data stream in some computer applications need to be encoded to meet particular requirements. One of the popular examples is the Eight-to-Fourteen Modulation (EFM) used to represent audio data in compact discs (CDs)[33]: see Watkinson (2002). Applying the same concept to the transmitted data in CAN networks, Eight-to-Eleven Modulation (EEM) would be an effective solution to avoid hardware bit-stuffing. This approach is described here.

EEM is a special case of the X-to-Y modulation approach where "X" equals to 8 (the number of bits per byte) and "Y" equals to 11. Modulating CAN data bytes using EEM can help to ensure that the number of consecutive bits of the same polarity does not exceed four and that, therefore, hardware bit stuffing is no longer applicable. As a consequence, predictability of CAN message transmission times will increase.

By applying the formula in Equation E-2, where the number of bits subject to bit stuffing is equal to 8 and the maximum number of consecutive bits allowed is again equal to 4, the number of required stuffed bits in each byte will be equal to 2.3 (~ 2 bits).

By referring to the analysis provided by Nolte *et al.* (2001), the worst-case scenario for one byte would be four ones followed by three zeros (i.e. 11110001). With adding two

[33] In digital audio CD systems, data is represented by NRZI in which the presence of alternating '1s' and '0s' at high rates can be so fast that the optical system cannot perceive the data. On the other hand, a jitter or data locking can be introduced if the data rate is set too low. In order to keep the data rate between non-too-low and non-too-high frequency ranges, an appropriate coding mechanism is required. For example, Eight-to-Fourteen Modulation (EFM) technique is widely adopted in CD recording systems. In EFM, each data byte (8-bit) is converted to an equivalent 14-bit codeword using lookup table. This conversion ensures that any data word has: [1] No more than 10 zeros between every two ones: this is for synchronisation; [2] No less than 2 zeros between every 2 ones: this is to reduce frequency and help the laser to detect the recorded data. For further details, refer to (Watkinson, 2002).

stuffed bits as calculated above, the resulting byte becomes: '1111**0**000**1**1' (the stuffed bits are bolded). This would work only if the boundaries between adjacent bytes are not taken into consideration. More clearly, in the case where the previous byte to the one shown above ends with the bit "1", CAN hardware will detect five consecutive bits (at the boundaries between this and its preceding byte) and thus insert an additional bit for synchronisation.

There is no way to avoid this unless if a further bit is padded near the most significant bits of each byte. Figure E-9 shows one possible way – which has been considered in this study – for byte encoding. The figure clearly demonstrates that each byte needs to be represented by (at least) 11-bit codeword. This process is referred to, in this thesis, as "eight-to-eleven modulation (EEM)". Note that "SB" in the figure stands for "stuffed bit".

Figure E-9: Encoding a data byte in EEM method.

Same as SBS, when EEM was implemented in practice (using the TTC-SCC1 scheduler) the maximum throughput for the real data was equal to 5 bytes (since stuffed bits require approximately two bytes and one byte is reserved for Slave ID in the S-C protocol).

If B_{yr} is the number of bytes used for real data, then $B_r = 8B_{yr}$, and $B_s = 3\ B_{yr}$ (since 3 bits are required to encode each byte in the EEM). By substituting these terms in Equation E-3, B_{yr} will be calculated as 5.1 bytes. In another word, the number of bytes used for real data cannot be more than 5 when EEM is applied.

Implementation

This section explores possible ways for implementing EEM technique in practice using the TTC-SCC1 scheduling protocol. The two main approaches considered are: lookup table and function call. Each of these approaches is discussed in details.

Lookup table approaches

As in the EFM data modulation method described previously, the EEM algorithm can use a lookup table to store the EEM codes for all possible byte values (between 0 and 255). In this approach, EEM codes are computed offline (using a desktop program) and stored in a one-dimensional array which will then be used by the embedded program. Note that with 16-bit microcontrollers (as used in this study), array with 16-bit integers can be a suitable choice to represent the lookup table. However, using 16-bit array can provide two types of EEM tables: explicit and implicit tables

Explicit EEM table

In the explicit representation, each array element (16-bit) is reserved for one EEM code (11-bit), and the remaining bits in each element are left unused. Since there are 256 possible byte values, the required array size will be 256 integers. Basically, the array index here represents the byte value and the array element represents the EEM code. For example, the byte "0" has the EEM code of "546" (i.e. 01000100010). The first few lines of the explicit lookup table considered here is shown in the following table.

Table E-6: Part of the explicit EEM lookup table.

Index	Array element	Binary value
0	546	01000100010
1	547	01000100011
2	548	01000100100
3	549	01000100101
4	554	01000101010
5	555	01000101011
6	556	01000101100
7	557	01000101101
8	562	01000110010
9	563	01000110011
10	564	01000110100

Implicit EEM table

In the implicit representation of the EEM table, each array integer may include bits for two or three EEM codes (11-bits each). For example, the first array element includes the

EEM code of the byte “0”, and five bits from the EEM code of the byte “1”. In this option, the overall array size is calculated as follows:

$$Array\ size = \frac{(No.\ of\ \ all\ \ possible\ byte\ values) \times (No.\ of\ \ bits\ \ required\ \ for\ \ each\ byte)}{(No.\ of\ \ bits\ \ per\ \ array\ element)}$$

Equation E-6

Since the number of all possible byte values is 256, each byte requires 11 bits for encoding, and the length of each array element is 16 bits, then the size of the EEM array will be 176 integers. Remember that the explicit EEM table contains 256 integers.

Once the lookup table is generated, it is then used by both the Master ansd Slave nodes. Note that although the lookup table methods are offline-based, there are still some processes to be done at runtime. This includes: (1) looking up data from the EEM table, and (2) placing data in (or extracting from) their corresponding 8-bit CAN registers. Unlike the process carried out by the encoder, the lookup process in the decoder may not be straightforward. This study outlines two mechanisms for the lookup process that the Slave’s program can implement.

Searching element:

The one-dimensional array is the simplest (software) method to represent the EEM lookup table. If, for example, the explicit lookup table is used in both the Master and Slave, then the decoder needs to search for the EEM code in the array lines to find its corresponding byte value (i.e. array index).

There are many methods for looking up a particular value in a sorted array. For example, Binary Search Algorithm (BSA) is a widely-used searching method in computer science (Knuth, 1998). It basically rules out half of the data at each search step for reducing the search time. A binary search finds the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner. Note that this algorithm is logarithmic in which it requires at most (1+*log*2 N) iterations to return the answer; where N is the number of records.

If the implicit lookup table is used in the Slave, then a direct searching mechanism has to be applied to determine the original byte values for the receiving EEM codes. In the design considered in this study, in order to simplify the searching process, the implicit lookup table was divided into 16 regions (each contains 16 EEM codes stored in 11 array elements). Therefore, the decoder needs to work out in which region a particular EEM code is stored then performs a search within this region. Overall, searching mechanisms cause low utilisation of the CPU.

Using reverse array:

In this approach, the EEM table – in the Slave – is implemented in the opposite way: i.e. the array index represents the EEM value and the array element represents its corresponding byte value. Since the maximum EEM value is 1501, then the array can have 1501 elements. But in order to save more memory, unnecessary ranges can, if possible, be removed. For instance, since the minimum EEM value is 546 then $f(546)$ can be set to be equal to $f(0)$ to start the table from the minimum EEM code while all elements with indices less than 546 are entirely removed from the table. This implies that, in the decoding process, $f(x) = f(x - 546)$. With this option the EEM array can have 956 bytes (i.e. $Max - Min = 1501 - 546 + 1$) and hence save more memory.

Nonetheless, in order to make a better use of the available memory resources, the EEM technique can also be implemented using online computational methods where the EEM code for an input byte value is calculated at runtime using a function call. This approach is further described here.

Function call approaches

This is an alternative implementation of EEM in which the equivalent EEM code for each data byte is calculated online without the need to store data in a lookup table. In this approach, the Master uses an “encode data” function to perform the EEM conversion and then places the EEM words in the corresponding registers for transmission with the CAN frame. On the Slave node, a reverse process (decoding) takes place to extract the original data bytes. This method can be practically implemented using two approaches: algorithmic and mathematical coding. Both approaches are described here.

Algorithmic coding

In algorithmic coding, a set of logical operations (e.g. SHIFT, AND, OR, etc) are employed to stuff the three required bits in each byte for generating the equivalent EEM code. The complete process of this coding method is illustrated in Figure E-10. Note that “In_index” indicates the bit order in the original byte, while “Out_index” indicates the bit order in the EEM code.

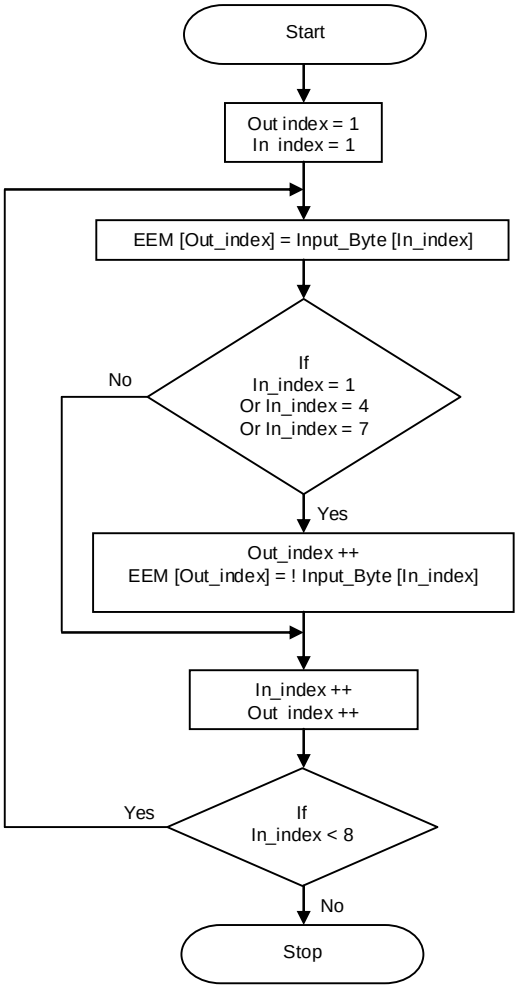


Figure E-10: The process of the EEM algorithmic coding.

On the Slave node, the reverse process is applied to take out the stuffed bits and recover the original byte values. It should be noted that in the offline approaches the desktop program – used to generate the lookup table – was based on this algorithm.

Mathematical coding

The EEM value of a given byte can be calculated using a mathematical formula. To clarify this, by looking at the EEM values in the explicit lookup table, it can be observed that as the byte value (array index) increases the equivalent EEM value increases with a specific trend (Figure E-11). The three graphs show the numerical EEM values in three ranges: (a) full range, (b) byte values between 0 and 12, and (c) byte values between 0 and 32. Note that the trend shown in (c) is repeated 8 times in (a).

Mathematical equations for EEM values can be derived directly from the analysis of the graphs. If the EEM value of a byte x is represented by $f(x)$, then $f(x)$ can be calculated using Equation E-7:

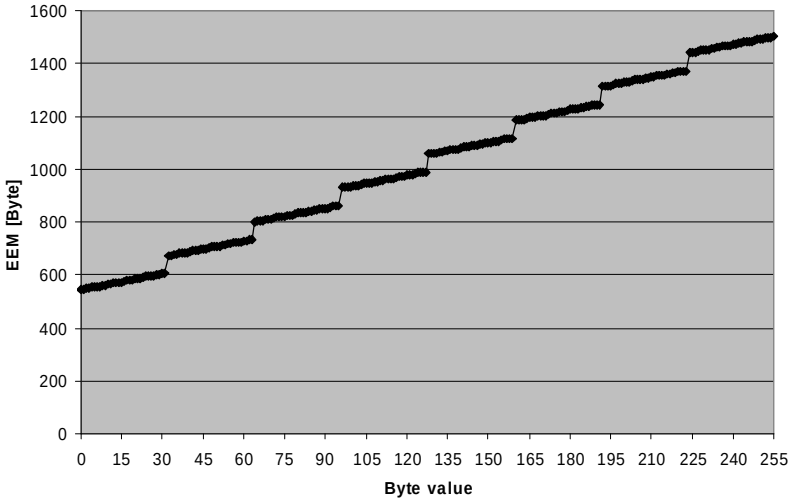
$$f(x) = f(0) + x + \left(\left\lfloor \frac{x}{4} \right\rfloor \times 4\right) + \left(\left\lfloor \frac{x}{32} \right\rfloor \times 64\right)$$

Equation E-7

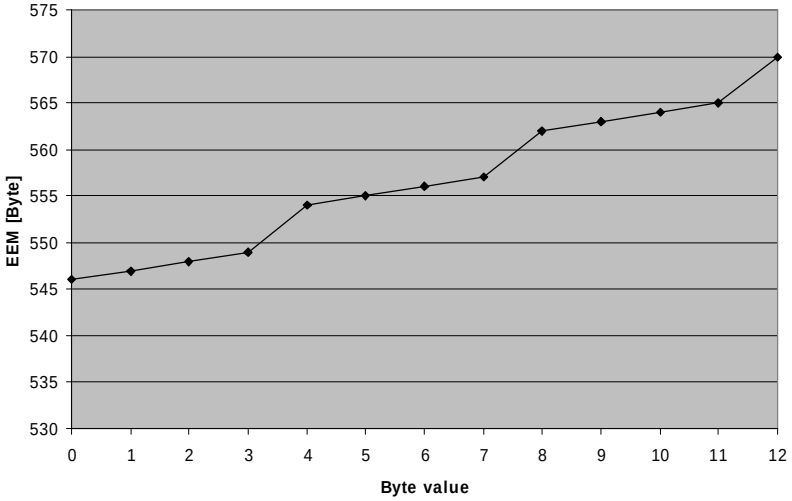
Where $f(0)$ is the initial value which represents the EEM code of the byte "0", and $\lfloor x \rfloor$ is the floor function of x. Floor function is defined as the largest integer less than or equal to x. Similarly, on the Slave, the decoding function employs Equation E-8 to calculate x from the receiving $f(x)$ value.

$$x = f(x) - f(0) - \left(\left\lfloor \frac{f(x) - f(0)}{4} \right\rfloor \times 2\right) - \left(\left\lfloor \frac{f(x) - f(0)}{64} \right\rfloor \times 16\right)$$

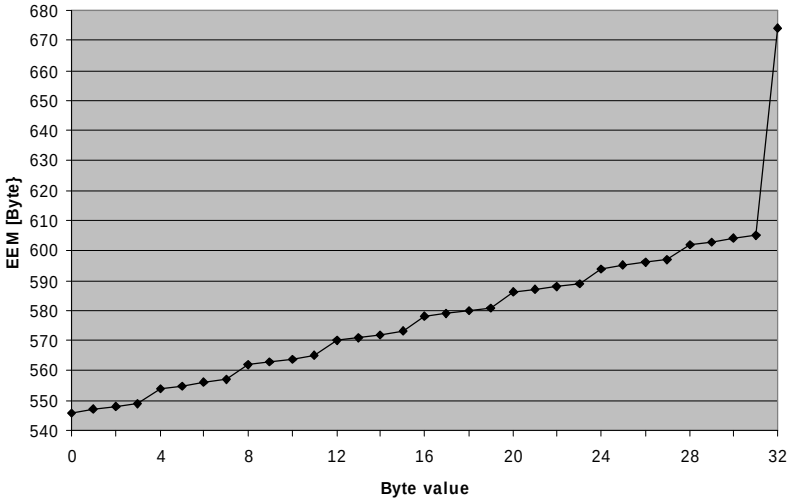
Equation E-8



(a)



(b)



(c)

Figure E-11: EEM values for different byte ranges: (a) full range, (b) bytes between 0 and 12, (c) bytes between 0 and 32.

In general, online solutions require no excessive memory for storing data. However, they impose higher CPU overheads on the Master and Slave microcontrollers. One advantage over the SBS method is that the online implementation of EEM algorithm can be achieved by both algorithmic and mathematical approaches: in SBS, only algorithmic coding can be used. Remember that, unlike SBS, EEM coding processes each byte in the data segment independently.

Summary

In general, the discussions in this appendix suggested that jitter in the transmission times of data messages caused by bit-stuffing mechanism in CAN protocol can be reduced significantly using simple coding techniques.

A schematic illustration of the techniques developed in this study, and their possible implementation methods, is provided in Figure E-12.

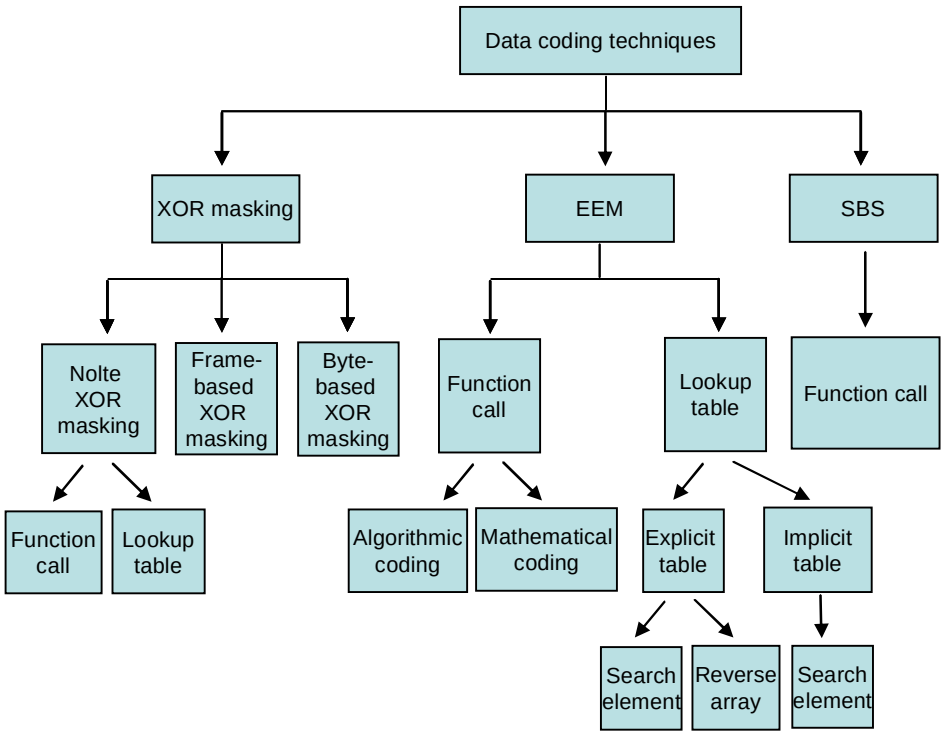


Figure E-12: Summary of the coding techniques described in this appendix.

Appendix F

Results from the jitter-reduction techniques

Introduction

This appendix provides the output results from the data coding techniques introduced in Appendix E for reducing jitter in multi-processor embedded systems employing TTC-SCC architecture. The particular system used, as previously noted, is the TTC-SCC1 scheduling protocol as a representative scheduler which utilises the tick messages for exchanging real data with other nodes. Since TTC-SCC2 and TTC-SCC3 schedulers use the same configuration for the Master tick messages, they can also benefit from such techniques to reduce jitter in the timing of their Slave tasks.

The key parameter against which the various techniques are assessed is the release jitter in Slave tasks. However, the techniques are also weighed up against the required CPU time and memory overheads.

The experimental methodology used to obtain the results is first outlined[34].

Experimental methodology

Hardware setup

In summary, the study used here was based on using two microcontroller nodes: one represents the Master and the other represents the Slave. Each node was based on a Phytec board supporting an Infineon C167 microcontroller with oscillator frequency of 20 MHz. The network nodes were connected using a twisted-pair CAN link. The CAN baudrate was 1 Mbps.

[34] The work described in this appendix has been adapted from the studies presented in the author's publications [7] and [9] listed in page xvii.

Software setup

The data coding techniques discussed in Appendix E were applied to the TTC-SCC1 (described in Chapter 9) to explore their impact on the jitter levels of Slave tasks. The scheduler tick interval used was 4 ms. The Keil C166 compiler was used.

Jitter measurements

The experimental methodology used to obtain jitter results in this study is similar to that described in detail in Section 11.2.3. Remember that jitter levels are measured at the release time of "`Slave1_Task_A`" (the only task) in the Slave node.

Jitter in each method was measured for systems with 8-, 7- and 6-byte data model to allow a meaningful comparison with other coding techniques. For example, the 6-byte data model uses five bytes for real data and one byte for Slave ID. The remaining bytes would either be unused or hold information about encoding process.

Assessing the CPU and memory loads

Since the focus in this study is on embedded systems with limited-resource requirements, consideration of the overheads (in terms of memory and CPU requirements) is important. To do this, the time taken for the encoding and decoding processes was measured from the microcontroller hardware using LabVIEW measurement tools. In each process, a pin was set high at the start of the coding function and low at the end of it. The resulting pulse was then measured which represented the processing time of the function at a given run. In each case, 1000 samples were recorded and then averaged. The CPU overhead values were also represented as percentages of the tick interval used (which is in this case 4 ms).

The memory requirements (ROM and RAM) to implement each technique in the C167 hardware platform used were also reported. In addition to the absolute values of memory requirements in each method, the results were also presented as percentages of the available on-chip ROM and RAM resources. Please note that the C167 boards used have 32 Kbytes ROM and 2 Kbytes RAM. Note that memory results were obtained from the systems using the maximum available data bandwidth only (i.e. 8-byte model).

Data selection

In the studies presented here, the data segment of each message may contain "best-case", "worst-case" or "random" values (and the Slave ID). One example of the best-case scenario is message data which consists of alternating '1s' and '0s' (e.g. 10101010): such data require no bit stuffing. The worst-case situation occurs when the data are set to be 11111000011110000…, since this causes the maximum level of CAN bit stuffing (Nolte *et al.*, 2001).

Both best- and worst-case data were used here for comparison purposes. In the case of random data, pseudo-random values were sent in each data byte. Note that exactly the same set of "random" data was used in each study. In total 50,000 messages were transmitted (and measured) in each experiment.

Benchmark measures

Jitter

The jitter levels resulting from the "best-case" and "worst-case" data sets are first considered. Table F-1 shows these jitter levels from the TTC-SCC1 scheduler. Note that all results are in microseconds (µs). Remember that 8-byte "Tick" messages were used, with one byte reserved for the Slave ID.

Table F-1: Task jitter from the TTC-SCC1 scheduler for best-case and worst-case data.

		8-byte
Best-case data	BCTT	161.4
	WCTT	163.6
	AVTT	162.5
	Diff. Jitter	2.2
	Avg. Jitter	0.6
Worst-case data	BCTT	174.4
	WCTT	176.7
	AVTT	175.5
	Diff. Jitter	2.3
	Avg. Jitter	0.6

It might be expected that the jitter levels when fixed data bytes are transmitted in each CAN frame will be zero. In practice, the difference jitter obtained was found to be approximately 2μs (equal to 2 bit times for the CAN bus at 1 Mbps). These jitter figures are approximately equal to 1% of the whole message length.

Without access to the implementation details for the CAN controller (which are not generally available), the precise cause of these variations cannot be identified. However, it can be noticed that the CAN controller is asynchronous with respect to the CPU. This means that the generation of an interrupt by the controller and its servicing by the CPU takes a period of time which may depend on several factors (for example, the state of the instruction pipeline in the CPU when the interrupt is generated). The reported timing values are compatible with this kind of jitter.

The jitter levels resulting from the "random" data set is now considered. Table F-2 shows these jitter levels from the TTC-SCC1 scheduler. Note that all results are in microseconds (μs). Remember that 8-, 7- and 6- byte data models are used here.

Table F-2: Task jitter from the TTC-SCC1 scheduler for random data.

		8-byte	7-byte	6-byte
Random data	BCTT	162.4	151.5	140.7
	WCTT	172.6	160.6	148.7
	AVTT	166.7	154.4	143.5
	Diff. Jitter	10.2	9.1	8.0
	Avg. Jitter	1.5	1.4	1.3

The results clearly show that as the number of data bytes – sent in the CAN message – increases, the number of bits stuffed by the CAN hardware increases. The maximum number of stuffed bits, when maximum data bandwidth was used, was found to be 10.

CPU and memory requirements

CPU and memory requirements in the TTC-SCC1 before employing any coding technique are presented in Table F-3. Of course, there are no encoding and decoding processes in the original system, therefore, the CPU and memory overheads are equal to zero.

Table F-3: CPU and memory requirements for the TTC-SCC1 scheduler.

	Encoding		Decoding	
	Absolute value:	Percentage value:	Absolute value:	Percentage value:
CPU overhead (ms)	0	0%	0	0%
Data overhead (Byte)	0	0%	0	0%
Code overhead (Byte)	0	0%	0	0%

Nolte XOR transformation

Jitter

The jitter levels from Nolte XOR transformation are presented in Table F-4. Note that all results are in microseconds (µs).

Table F-4: Jitter results from Nolte XOR transformation.

	8-byte	7-byte	6-byte
BCTT	162.4	151.4	140.6
WCTT	172.5	160.4	148.6
AVTT	166.6	154.6	143.4
Diff. Jitter	10.1	9	8
Avg. Jitter	1.4	1.3	1.3

Overall, jitter levels were not reduced as a result of applying Nolte XOR transformation to the random data set considered in this study (compare with the benchmark results presented Table F-2).

CPU and memory requirements

To implement this method in the microcontroller hardware considered in this study, encoding and decoding processes required little amounts of the CPU time. Table F-5 shows the implementation costs of the Nolte method in terms of CPU and memory overheads.

Table F-5: Implementation costs of the Nolte XOR transformation.

	Encoding		Decoding	
	Absolute value:	Percentage value:	Absolute value:	Percentage value:
CPU overhead (ms)	0.0158	0.4%	0.0157	0.4%
Data overhead (Byte)	0	0%	0	0%
Code overhead (Byte)	32	0.1%	24	0.1%

The CPU overhead figures shown in the table are equal to 0.4% from the total CPU time available in each scheduler tick interval. The memory requirements to implement this technique were negligible as compared against the available memory resources.

Frame-based XOR transformation

Jitter

The jitter results from the frame-based XOR transformation are shown in Table F-6. Note that all results are in microseconds (μs).

Table F-6: Jitter results from frame-based XOR transformation.

	8-byte	7-byte	6-byte
BCTT	167.9	157.8	147.1
WCTT	177.9	166.9	155.1
AVTT	171.8	161	149.5
Diff. Jitter	10	9.1	8
Avg. Jitter	1.5	1.4	1.3

As expected, the jitter levels were not improved when the frame-based XOR transformation was applied to random CAN traffic. This is because masking the whole frame (when at least one long sequence of identical bit is detected) had no potential to reduce the number of stuffed bits in that frame. In such a case, the masked frame would also be of random data pattern and as a result the CAN bit-stuffing would be expected to remain at the same level. Frame-based XOR transformation may only fit particular applications that contain few numbers of long sequences in each of their data frames.

CPU and memory requirements

Table F-7 shows the CPU and the memory overheads of implementing this method on the used microcontroller hardware.

Table F-7: Implementation costs of the frame-based XOR transformation.

	Encoding		Decoding	
	Absolute value:	Percentage value:	Absolute value:	Percentage value:
CPU overhead (ms)	0.2457	6.1%	0.0144	0.4%
Data overhead (Byte)	8	0.4%	0	0%
Code overhead (Byte)	338	1%	36	0.1%

Byte-based XOR transformation

Jitter

Table F-8 shows the jitter levels obtained from the byte-based XOR transformation. Remember that in this method, the maximum number of bytes that can be used for real data (including Slave ID) is 7 as one byte is used to store information for decoding. Note that all results are in microseconds (µs).

Table F-8: Jitter results from byte-based Nolte XOR transformation.

	7-byte	6-byte
BCTT	165	155.8
WCTT	172.1	162.9
AVTT	167.1	157.84
Diff. Jitter	7.1	7.1
Avg. Jitter	1.2	1.1

It can be clearly seen from the results that the application of Nolte XOR transformation to particular bytes – when required – has helped to reduce the jitter levels from 9 µs down to 7 µs when the maximum available data bandwidth was used. This reduction in jitter is approximately equal to 20%.

CPU and memory requirements

In both lookup table and function call implementation methods for the byte-based XOR transformation, jitter values are at the same level. However, each approach required different implementation costs. Table F-9 shows the CPU and memory overheads of implementing each method on the microcontroller hardware used.

Table F-9: Implementation costs of the byte-based XOR transformation.

		Encoding		Decoding	
		Absolute value:	Percentage value:	Absolute value:	Percentage value:
Lookup table	CPU overhead (ms)	0.0537	1.3%	0.4	10%
	Data overhead (Byte)	512	25%	512	25%
	Code overhead (Byte)	152	0.5%	104	0.3%
Function call	CPU overhead (ms)	0.2896	7.2%	0.0218	0.5%
	Data overhead (Byte)	3	0.1%	0	0%
	Code overhead (Byte)	244	0.7%	70	0.2%

Note that the decoding process in the function call method was found much faster than encoding. This is because the encoding process involved checking each byte (bit by bit) to see if hardware bit-stuffing will occur: this process took more time than the corresponding checking routine during decoding (which only required testing of a single bit flag). Also note that in the lookup table approach, the required CPU time increased by approximately 10% in the Slave node, and the data memory increased by 25% in both Master and Slave nodes. However, the CPU overhead for the encoding process was seen reduced significantly. The reason why the CPU overhead in the decoding process increased by a large factor is that each received byte must be checked and if it is masked then the Slave will search in the lookup table for its corresponding value. The duration of this search process entirely depends on the combination of the byte values received.

Software bit stuffing (SBS)

Jitter

The task jitter which resulted from the use of SBS is presented in Table F-10. Note that all results are in microseconds (μs).

Table F-10: Jitter results from the SBS technique.

	6-byte
BCTT	161.2
WCTT	166.2
AVTT	162.72
Diff. Jitter	5
Avg. Jitter	0.8

The values in the table show that, in practical implementations used in this study, SBS can reduce the jitter on the Slave task from 8 µs to 5 µs (when the same number of real data bytes is used, in this case 6). This reduction is approximately equal to 40% which is significant in many applications that require high levels of predictability.

CPU and memory requirements

The memory and CPU overheads imposed by SBS on the used microcontroller hardware are presented in Table F-11.

Table F-11: Implementation costs of the SBS technique.

	Encoding		Decoding	
	Absolute value:	Percentage value:	Absolute value:	Percentage value:
CPU overhead (ms)	0.44831	11.2%	0.4232	10.6%
Data overhead (Byte)	0	0%	0	0%
Code overhead (Byte)	250	0.8%	150	0.5%

The SBS technique required approximately 11% of the tick interval to perform the encoding or decoding process. Although this figure is higher than that required in the previous technique (by approximately 4%), it is still very small in comparison with the available time resources. Remember that with SBS, the reduction in jitter was two times that obtained from the byte-based XOR masking.

Eight-to-Eleven Modulation (EEM)

Jitter

The task jitter which resulted from the use of EEM is presented in Table F-12. Note that all results are in microseconds (µs).

Table F-12: Jitter results from the EEM technique.

	6-byte
BCTT	163.4
WCTT	168.5
AVTT	165
Diff. Jitter	5.1
Avg. Jitter	0.9

By looking at the results in the table, it is noticeable that – like SBS – when the message data section is encoded using the EEM technique, the jitter is reduced by approximately 40% on the microcontroller platform used in this study.

CPU and memory requirements

The memory and CPU overheads imposed by EEM on the used microcontroller hardware are presented in the following tables.

Table F-13: CPU overhead in all EEM methods (values are in ms).

		Encoding		Decoding		
EEM approach	Coding Method	Absolute value:	Percentage value:		Absolute value:	Percentage value:
Lookup table	Explicit table	0.08	2%	BSA	0.36	9%
				Reverse Array	0.07	1.75%
	Implicit table	0.16	4%	Search Element	0.49	12.25%
Function call	Algorithmic coding	0.42	10.5%	---	0.40	10%
	Mathematical coding	0.10	2.5%	---	0.16	4%

Table F-14: Memory overhead in the encoding process in all EEM methods (values are in Bytes).

EEM approach	Coding Method	Encoding			
		Data		Code	
		Absolute value:	Percentage value:	Absolute value:	Percentage value:
Lookup table	Explicit table	522	25.5%	172	0.5%
	Implicit table	362	17.7%	364	1.1%
Function call	Algorithmic coding	10	0.5%	266	0.8%
	Mathematical coding	10	0.5%	206	0.6%

Table F-15: Memory overhead in the decoding process in all EEM methods (values are in Bytes).

EEM approach	Coding Method	Decoding				
			Data		Code	
			Absolute value:	Percentage value:	Absolute value:	Percentage value:
Lookup table	Explicit table	BSA	522	25.5%	266	0.8%
		Reverse Array	1922	93.8%	164	0.5%
	Implicit table	Search Element	362	17.7%	480	1.5%
Function call	Algorithmic coding	---	10	0.5%	218	0.7%
	Mathematical coding	---	10	0.5%	182	0.6%

The tables demonstrate how much processing time and memory each implementation method of the EEM technique required when implemented on the used C167 hardware. As a general observation, searching elements in the decoder requires quite a long time to recover the original byte (~12% of the used tick interval). Alternatively, a reverse lookup table can be implemented and used to save time but on the account of the used data memory (data memory required is too large). Another observation is that the mathematical coding can provide little CPU overhead and small amounts of memory requirements. Combinations of encoding and decoding methods can be used as

appropriate. Indeed, it must be reported that there is no “best” solution as the selection of the most appropriate implementation is highly dependent on the available resources of the Master and Slave microcontrollers adopted.

Summary of the results

Based on the results presented in this appendix using TTC-SCC1 scheduling protocol and random data messages, it was shown that Nolte XOR transformation and the selective frame-based XOR transformation did not help to reduce the jitter levels. For the frame-based technique, only one bit was required to indicate if the frame was masked or not.

The selective byte-based XOR transformation technique provided a minimum jitter of 7 µs on the C167 microcontrollers used (at 1 Mbps CAN baudrate). This means that the method had the potential to reduce the number of stuffed bits in the data section of a CAN frame by approximately 20% in practical implementations. For this method, one bit per (real) data byte was required to indicate if a byte was masked or not. For example, when 6 “real” data bytes were transmitted, six bits were needed for the encoding data: however, to insure that these six bits do not themselves introduce bit stuffing, one byte (including appropriate padding) was used along with one bit from the Slave ID byte. This caused a loss of 12.5% of the available CAN message bandwidth.

When SBS and EEM coding schemes were applied, the minimum jitter was equal to 5 µs. this means that such techniques had the potential to reduce the levels of jitter in a CAN-based system by approximately 40%. However, for both techniques, up to 5 bytes of “real” data were allowed to transmit in a CAN message. This is because, in SBS, up to 13 (software) stuffed bits were required plus 3 bits for padding, to ensure that the stuffed bits are not themselves subject to hardware bit stuffing. In EEM, 15 bits in total were required to convert the real data bytes into their equivalent codewords to avoid hardware bit stuffing. As a consequence, a loss of 25% of the CAN message bandwidth was caused in both techniques. Figure F-1 summarises the jitter results in all techniques and compares them with those obtained from the original scheduler (without any coding scheme employed).

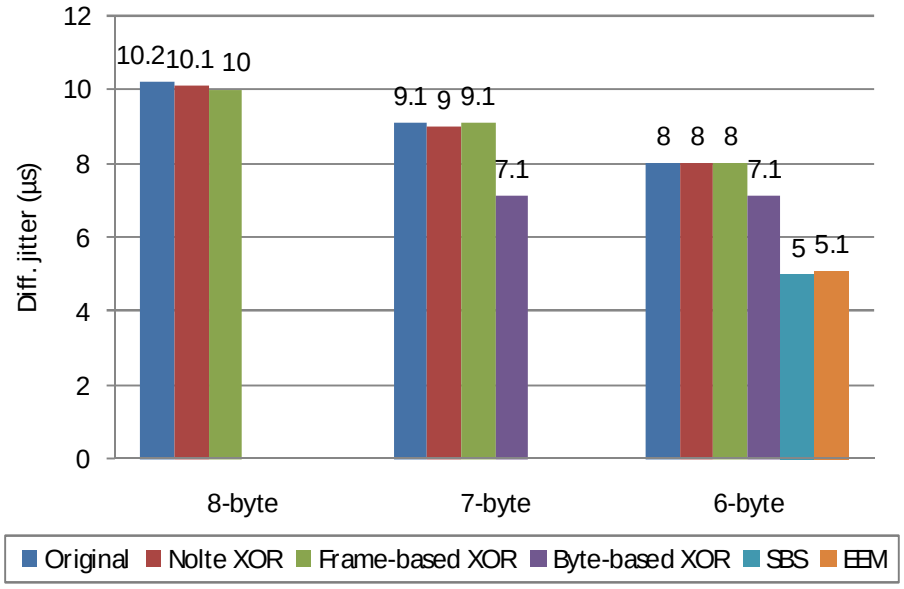


Figure F-1: Jitter results from all coding techniques on C167 platform.

With regards to jitter results, it is worth emphasising that – in C167 microcontrollers – 2 µs (i.e. ±1 bit time for the CAN bus at 1 Mbps) of the jitter values were likely related to clock synchronisation between the CAN controller and the CPU. If this is taken into account, then the remaining 3 µs jitter – in the case of SBS and EEM – are likely to be generated from the control fields of the CAN frame which cannot be fully controlled by the software designers. For example, the CRC field, tailed to the CAN message, contains a 15-bit codeword which is calculated as a function of the bit contents in all fields including the data field. Such 15 bits can, in the worst-case scenario, induce three (hardware) stuffed bits (Nolte *et al.*, 2001). If this proves to be correct, then SBS and EEM techniques had the capability to provide the maximum reduction of jitter which could be possible at the software layer.

The message overhead (bandwidth) required to implement each technique is illustrated in Figure F-2. Note that the results are in “number of bits”.

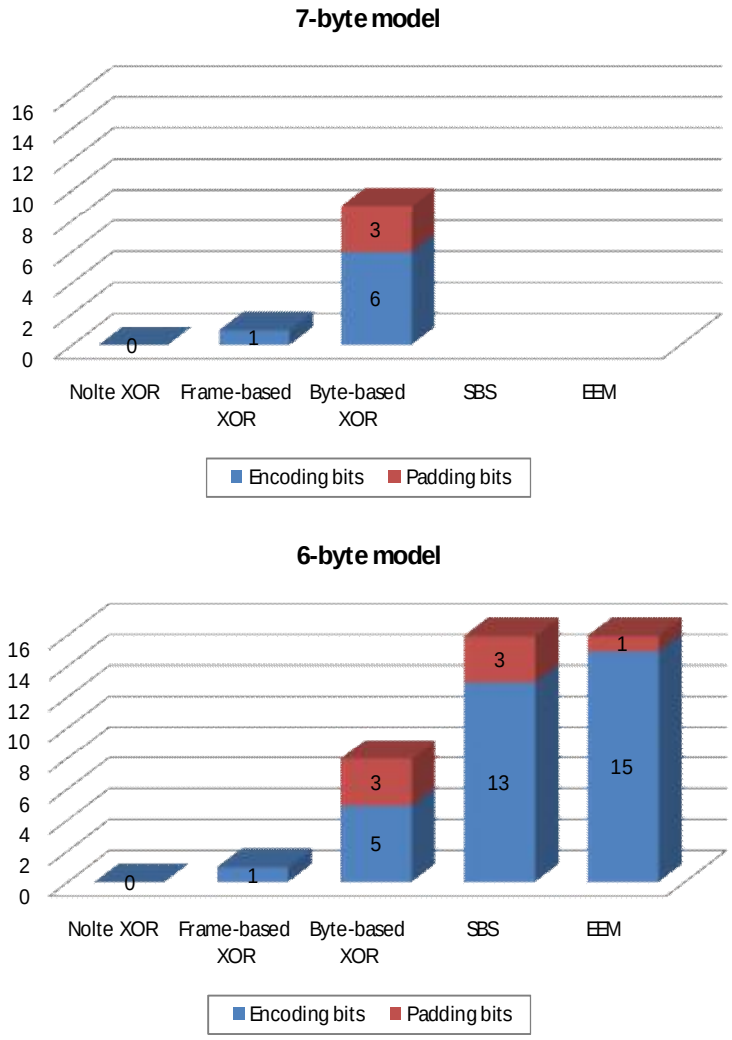


Figure F-2: Message overheads in all coding techniques.

The results can be summarised as follows. In the 7-byte model, the frame-based XOR masking had a 1-bit overhead while the byte-based XOR masking had a 9-bit overhead (6 bits for encoding information and 3 bits for appropriate padding). In the 6-byte model, the frame-based XOR masking (again) had an overhead of 1 bit, the byte-based XOR masking had an overhead of 8 bits, while the overheads for the SBS and EEM were 16 bits.

Guidelines for selecting a suitable method

This section attempts to provide general rules for selecting the most suitable data coding technique for a particular project. It is assumed here that the designer has decided to improve the overall performance of their existing design implemented on CAN

protocol, and not to upgrade the hardware network using a more predictable solution (such as FlexRay).

- If the CAN data messages are likely to contain long sequences of ones and zeros – as in the data set observed by Nolte *et al.* – then the direct application of Nolte (XOR) transformation can significantly reduce the effect of bit stuffing and, therefore, be a suitable solution.
- If the CAN data are likely to have random characteristics (with any degree of randomness), then the designer needs to know the maximum tolerated jitter. If (for example) the standing jitter levels are quite high for the system to tolerate, but a reduction of up to 20% can help, then the selective byte-based XOR masking method can be a good option. Note that this method requires little CPU and memory overheads, but imposes 12.5% loss of the data bandwidth.
- In the case where 20% reduction in jitter is not sufficient, then the designer needs to select between SBS and EEM methods.
- If the CPU time is very limited, then the explicit lookup table (with reverse array in the decoder) can be a good solution. However, if the available data memory is very limited, then the online mathematical coding will be the best option. Designers can also chose to implement the implicit lookup table in the encoder for reducing the data memory overhead (compared to explicit table).
- If the CPU time is flexible but the data memory is very limited, then the user can chose one of these methods (in this order): SBS, EEM mathematical coding, or EEM algorithmic coding. Among these methods, the EEM mathematical can be the best in terms of CPU overhead.
- Combinations between the encoding and decoding implementations can also by applicable depending on the available resources of individual nodes.

Using alternative microcontroller hardware

Introduction

To avoid presenting results which are symptomatic of any particular hardware platform, a comparative study was carried out in which the XOR masking and SBS techniques were also applied to two other CPU families: “8051” (8-bit) and “ARM7” (32-bit).

In the 8-bit system, each node was based on a Phytec board supporting an Infineon 8051 microcontroller with a 10 MHz crystal oscillator. The C515C is based on the 8051 architecture with additional on-chip support for features such as CAN (Siemens, 1997).

In the 32-bit system, each node was based on a Keil MCB2100 board, supporting a Philips LPC2129 microcontroller with an ARM7 core. The oscillator frequency was 12 MHz and, through use of the on-chip PLL, the CPU frequency was 60 MHz.

Benchmark measures

Table F-16 shows jitter results from the TTC-SCC1 scheduler on the C515C (8051-based), and ARM7 microcontrollers when the best-case and worst-case data are transmitted. Note that all results are in microseconds (µs).

Table F-16: Task jitter from the TTC-SCC1 scheduler for best-case and worst-case data on 8051 and ARM.

		8051	ARM
Best-case data	BCTT	220.3	114.5
	WCTT	222.8	115.6
	AVTT	221.6	115.1
	Diff. Jitter	2.5	1.1
	Avg. Jitter	0.6	0.3
Worst-case data	BCTT	231.3	127.6
	WCTT	233.9	128.6
	AVTT	232.6	128.1
	Diff. Jitter	2.6	1.0
	Avg. Jitter	0.6	0.3

It was shown that, like the C167 which is also based on Infineon board, the difference jitter in both the best- and the worst-case data was 2.5µs (approximately equal to 2 bit

times for the CAN bus at 1 Mbps). For the ARM processor, the corresponding value was 1μs (1 bit time). These jitter figures are approximately equal to 1% of the whole message length in all cases. By looking at the results presented, it can also be noted that – as the CPU performance increases (e.g. ARM) – the level of difference jitter is found to fall.

The jitter levels resulting from the "random" data set on the 8051 and ARM microcontrollers are shown in Table F-17. Note that all results are in microseconds (μs). Remember that 8-, 7- and 6- byte data models are used here.

Table F-17: Task jitter from the TTC-SCC1 scheduler for random data on 8051 and ARM.

			8-byte	7-byte	6-byte
8051	Random data	BCTT	221.4	208.3	197.2
		WCTT	231.6	217.3	205.4
		AVTT	225.3	212.0	200.1
		Diff. Jitter	10.2	9	8.2
		Avg. Jitter	1.5	1.4	1.4
ARM	Random data	BCTT	114.6	103.5	92.6
		WCTT	123.6	111.6	99.9
		AVTT	117.3	105.9	94.9
		Diff. Jitter	9.0	8.1	7.3
		Avg. Jitter	1.4	1.3	1.3

The results show that the jitter levels from the 8051 processor were very similar to those obtained from the C167 alternative. However, it can be clearly seen that the jitter levels from the ARM processor was less by approximately one bit time. This can again be due to the way the CAN controller in such hardware synchronises its timing with the microcontroller CPU.

CPU and memory requirements in the TTC-SCC1 before employing any coding technique are presented in Table F-18. Of course, there are no encoding and decoding processes in the original system, therefore, the CPU overhead is equal to zero. Please note that – unlike the previous sections – the memory overheads by the "whole" software program in each technique were recorded. This is to allow a meaningful

comparison with the original system. Also note that the table shows the results as percentages of the available on-chip ROM and RAM resources. For example, the C515C boards used have 64 Kbytes ROM and 256 bytes RAM (Siemens, 1997), and the ARM boards have 128 Kbytes ROM and 16 Kbytes RAM (Philips, 2004).

Table F-18: CPU and memory requirements for the TTC-SCC1 scheduler on 8051 and ARM.

		Encoding		Decoding	
		Absolute value:	Percentage value:	Absolute value:	Percentage value:
8051	CPU overhead (ms)	0	0%	0	0%
	Data overhead (Byte)	52	20.3%	102	39.8%
	Code overhead (Byte)	1582	2.4%	1434	2.2%
ARM	CPU overhead (ms)	0	0%	0	0%
	Data overhead (Byte)	448	2.7%	512	3.1%
	Code overhead (Byte)	11408	8.7%	10208	7.8%

Jitter results from all techniques

The jitter results obtained from the 8051 and ARM microcontrollers are shown in the following tables. Note that all values presented are in microseconds (µs).

Table F-19: Jitter results from 8 byte methods (7 data bytes + Slave ID) using 8051 and ARM.

Platform		Nolte XOR transformation	Frame-based XOR transformation	Byte-based XOR transformation	SBS
C515	BCTT	221.5	221.9	----	----
	WCTT	231.7	232.0	----	----
	AVTT	225.4	225.2	----	----
	Diff. Jitter	10.2	10.1	----	----
	Avg. Jitter	1.5	1.5	----	----
ARM	BCTT	114.7	114.6	----	----
	WCTT	123.7	123.5	----	----
	AVTT	117.3	117.2	----	----
	Diff. Jitter	9.0	8.9	----	----
	Avg. Jitter	1.4	1.4	----	----

Table F-20: Jitter results from 7 byte methods (6 data bytes + Slave ID + encoding information) using 8051 and ARM.

Platform		Nolte XOR transformation	Frame-based XOR transformation	Byte-based XOR transformation	SBS
C515	BCTT	208.4	208.8	221.9	----
	WCTT	217.4	218.0	229.0	----
	AVTT	212.1	211.8	224.1	----
	Diff. Jitter	9	9.2	7.1	----
	Avg. Jitter	1.4	1.4	1.1	----
ARM	BCTT	103.4	103.7	114.6	----
	WCTT	111.5	111.6	120.7	----
	AVTT	105.8	105.8	116.2	----
	Diff. Jitter	8.1	7.9	6.1	----
	Avg. Jitter	1.3	1.4	1.1	----

Table F-21: Jitter results from 6 byte methods (5 data bytes + Slave ID + encoding information) using 8051 and ARM.

Platform		Nolte XOR transformation	Frame-based XOR transformation	Byte-based XOR transformation	SBS
C515	BCTT	197.6	197.8	208.1	220.3
	WCTT	205.8	205.9	215.2	225.4
	AVTT	200.5	200.4	210.4	222.0
	Diff. Jitter	8.2	8.1	7.1	5.1
	Avg. Jitter	1.4	1.3	1.1	0.8
ARM	BCTT	92.6	92.7	103.6	115.2
	WCTT	99.7	99.6	109.5	119.3
	AVTT	94.6	94.6	104.9	116.3
	Diff. Jitter	7.1	6.9	5.9	4.1
	Avg. Jitter	1.3	1.2	1.0	0.7

The results show that the minimum level of jitter obtained from the byte-based XOR transformation – when using 8-bit C515C Infineon boards – was approximately 7 µs (this is similar to the figure obtained from the equivalent C167 boards). When 32-bit ARM boards were used, the minimum jitter was approximately 6 µs. When the SBS method was applied, the jitter levels were further reduced. For example, with the 8-bit

boards, the minimum jitter approached was around 5 µs whereas with the 32-bit boards it came down to 4 µs.

Table F-22: CPU overhead from all techniques on 8051 and ARM (values are in ms).

Platform	Method	Encoding process		Decoding process	
		Absolute value:	Percentage value:	Absolute value:	Percentage value:
8051	Nolte XOR transformation	0.039	1%	0.039	1%
	Frame-based XOR transformation	0.9019	22.5%	0.0471	1.2%
	Byte-based XOR transformation (function call)	1.4932	37.3%	0.1134	2.8%
	SBS	2.9798	74.5%	2.6016	65%
ARM	Nolte XOR transformation	0.0121	0.3%	0.0121	0.3%
	Frame-based XOR transformation	0.1328	3.3%	0.0115	0.3%
	Byte-based XOR transformation (function call)	0.1244	3.1%	0.0175	0.4%
	SBS	0.1585	4%	0.1608	4%

From the table, it can be seen that the byte-based XOR encoding process took approximately 1.5 ms (37% of the 4ms tick interval used) and 0.1 ms (3%) on 8- and 32-bit microcontrollers, respectively. Also, it can be seen that the durations of the SBS encoding processes, on 8- and 32-bit microcontrollers, were 3 ms (74%: this is too large) and 0.16 ms (4%), respectively. The bit de-stuffing processes (on the Slaves) imposed similar CPU loads.

Table F-23: Memory overheads from all techniques on 8051 and ARM.

Platform		Master				Slave			
		ROM		RAM		ROM		RAM	
		Absolute	%	Absolute	%	Absolute	%	Absolute	%
8051	Nolte XOR transformation	1590	2.4	52.6	20.5	1442	2.2	102	39.3
	Frame-based XOR transformation	1788	2.7	60	23.4	1461	2.2	102	39.8
	Byte-based XOR transformation (function call)	1780	2.7	55	21.5	1482	2.3	103	40.2
	SBS	1848	2.8	60	23.4	1547	2.4	110	43
ARM	Nolte XOR transformation	11456	8.7	448	2.7	10304	7.9	512	3.1
	Frame-based XOR transformation	12240	9.3	640	3.9	10320	7.9	512	3.1
	Byte-based XOR transformation (function call)	11776	9	576	3.5	10288	7.8	512	3.1
	SBS	11856	9	480	2.9	10528	8	528	3.2

As can be seen from this table, the absolute values for memory requirements increase on the more powerful processors. For example, to implement the byte-based XOR encoding and decoding on the 32-bit processors, an additional 368 and 80 code bytes would be required (respectively). Similarly, to implement SBS on this processor, an additional 484 code bytes are required for coding and an extra 320 code bytes would be required for decoding.

Comparison between results in all platforms

The following figures summarise the jitter results obtained from the XOR masking and SBS techniques in all microcontroller platforms considered. For meaningful comparisons, the results from 8-, 7- and 6-byte models are illustrated.

The figures clearly show how the application of the byte-based XOR masking and SBS techniques have the capability to reduce the task jitter by significant factors. They also show that, as the processor's speed increases, jitter levels are likely to decrease in all

cases. The results suggest that the application of the EEM technique would provide the same levels of jitter reduction – as with SBS – in 8051 and ARM microcontroller hardware. Of course, this would be on the account of the CPU and memory overheads imposed by practical implementation of such a method.

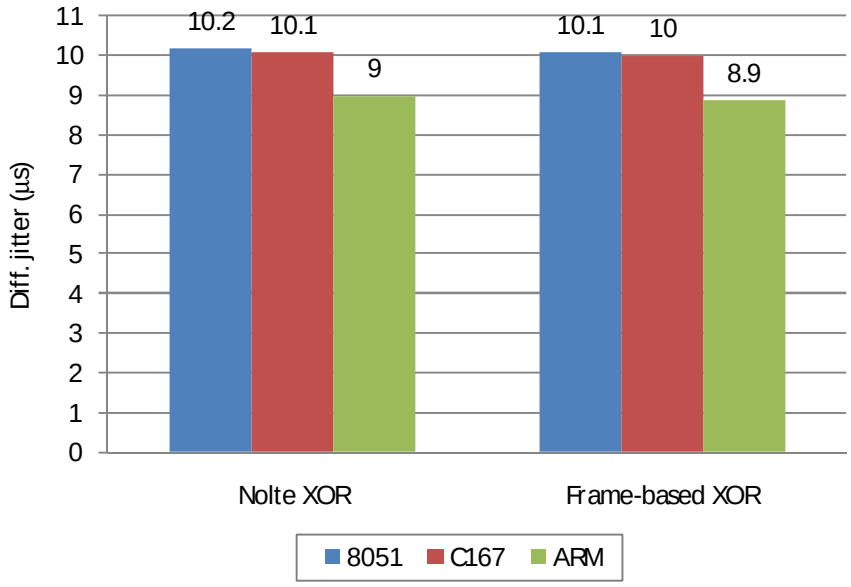


Figure F-3: Jitter results from all techniques on all platforms (8-byte models).

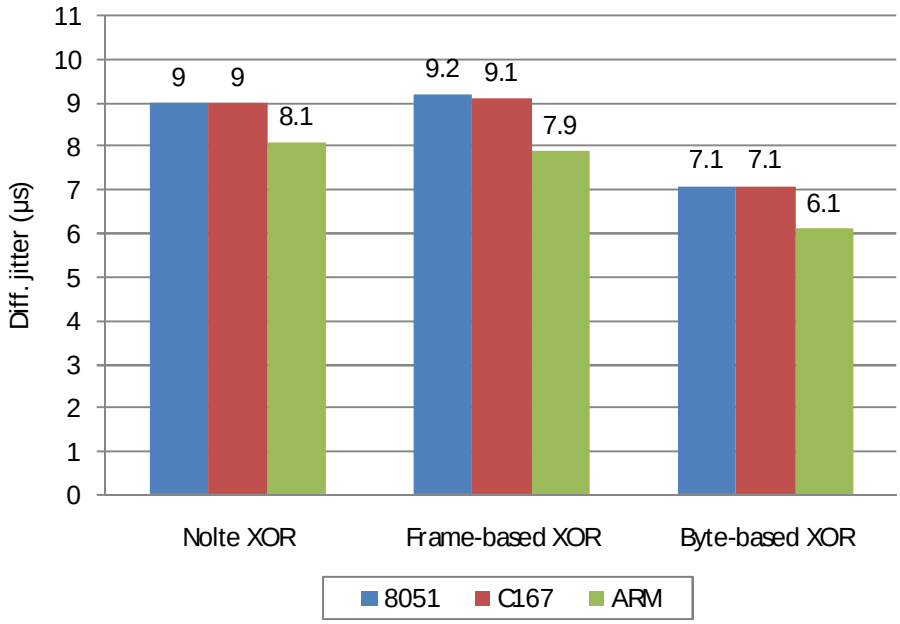


Figure F-4: Jitter results from all techniques on all platforms (7-byte models).

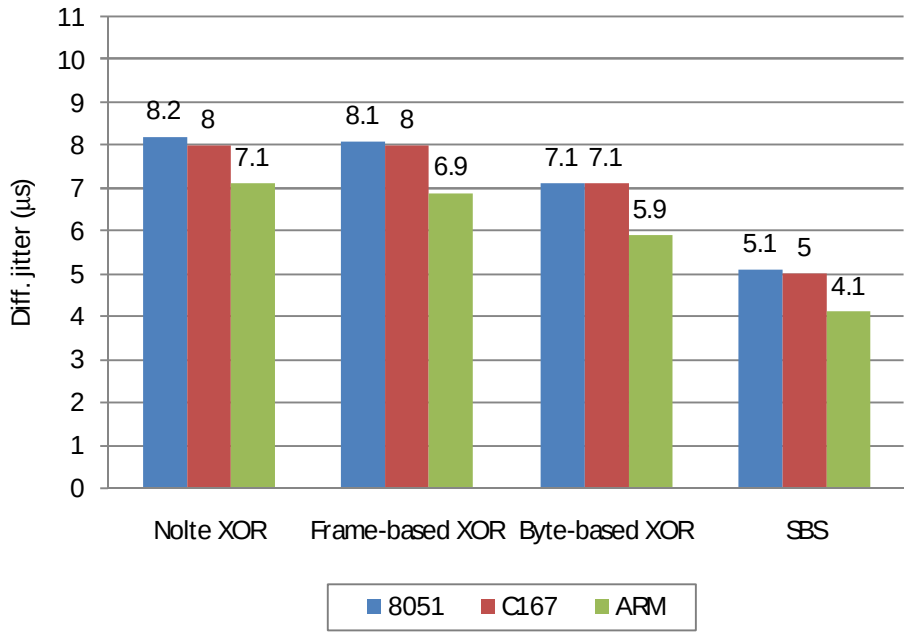


Figure F-5: Jitter results from all techniques on all platforms (6-byte models).

Appendix G

Adaptive Cruise Control (ACC) system: a case study

Introduction

As in Appendix F, the data coding techniques discussed can significantly reduce the levels of hardware bit stuffing. However, it only makes sense to employ such techniques in a particular application if the gains (from the reductions in jitter levels) are not outweighed by the losses that result from the implementation costs (e.g. CPU, memory resources and CAN bandwidth).

In order to begin to address this matter, a detailed case study was carried out in which the Software Bit Stuffing (SBS) technique was applied to an Adaptive Cruise Control (ACC) system developed for use in passenger vehicles. The study was based on using a realistic "hardware in the loop" (HIL) testbed facility[35].

ACC system

HIL testbed

The HIL testbed employed in this study has been previously described in detail (e.g. Short *et al.* 2004a; Short *et al.* 2004b; Short *et al.* 2004c; Short and Pont, 2005). Briefly, the simulation consists of a real-time representation of a motor vehicle travelling down a three-lane motorway, under realistic traffic conditions. It enables different system architectures to be assessed and quantitatively compared in a variety of realistic and repeatable scenarios. In this appendix, HIL simulator is used to represent an automotive Adaptive Cruise Control (ACC) system.

[35] The work described in this chapter has been adapted from the study presented in the author's publication [5] listed in page xvii.

ACC system

Adaptive Cruise Control (ACC) is a relatively new technological development in the automotive field which was claimed to reduce driver fatigue and the rate of auto accidents whilst increasing fuel efficiency (Stanton, 1997). The main system function of ACC is to control the speed of the host vehicle using information about:

- The distance between the subject vehicle and any forward vehicles.
- The motion of the subject vehicle.
- Driver commands.

Based upon the information acquired, the controller sends commands to the vehicle throttle and brakes to either regulate the vehicle speed to a given set value, or maintain a safe distance to a leading vehicle (if the speed of the vehicle in front is slower than the set value). It also sends status information to the driver.

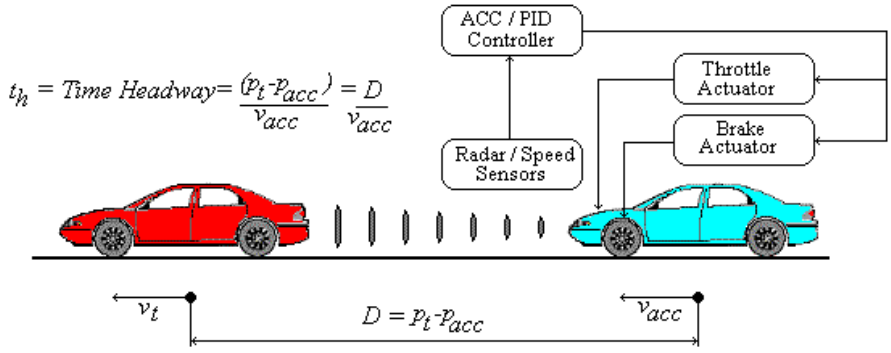


Figure G-1: An overview of the operation of the ACC.

The system under consideration in this study is a Type 2b ACC system: such a system has an automatic gearbox and active braking. Vehicle acceleration is limited to 2.0 m/s^2, deceleration to 3.0 m/s^2 in order to comply with ISO standards (ISO 15622, 2003). Figure G-1 shows the principle of operation. The controller that has been implemented is based of a modified version of that presented by Yi *et al.* (2000) and is shown in schematic form in Figure G-2 (see Short *et al.* 2004a; Short *et al.* 2004b; Short *et al.* 2004c).

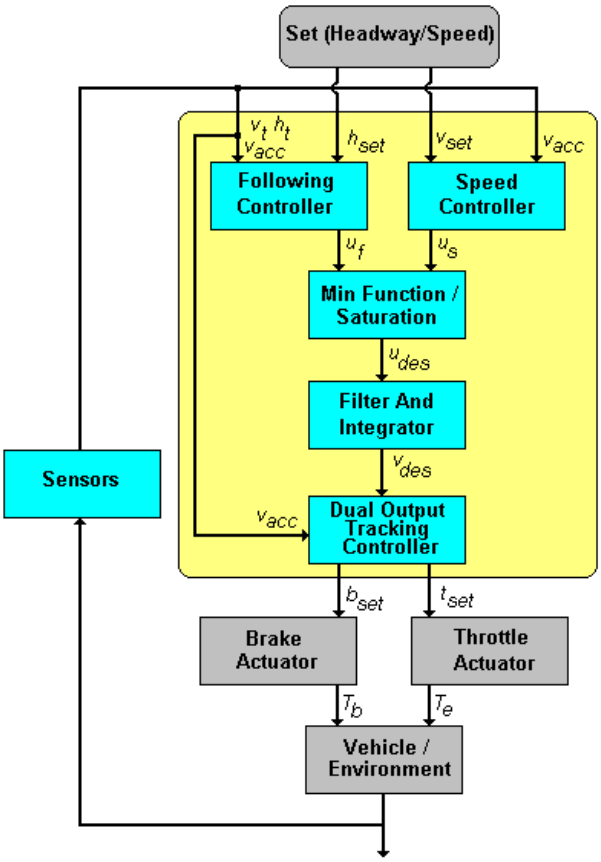


Figure G-2: The ACC implementation: adapted from Yi *et al.* (2000)

This ACC system has been used as an example of a distributed embedded control system which can be based on TTC-SCC architecture (the architecture used was that provided in TTC-SCC1 scheduler). The aim of this study is to demonstrate the effect of the developed SBS technique on the real-time performance of the ACC application.

System implementation

Overview

The ACC testbed was based on Infineon C167CS microcontrollers (one per node) running at a 20 MHz oscillator frequency. Each microcontroller had two on-chip CAN interfaces. In total 10 nodes were used. All nodes were connected using twisted-pair CAN links running at 500 kbaud. The system is schematically illustrated in Figure G-3. In this case study, SBS technique was applied to the ACC system to explore the impact on the real-time behaviour.

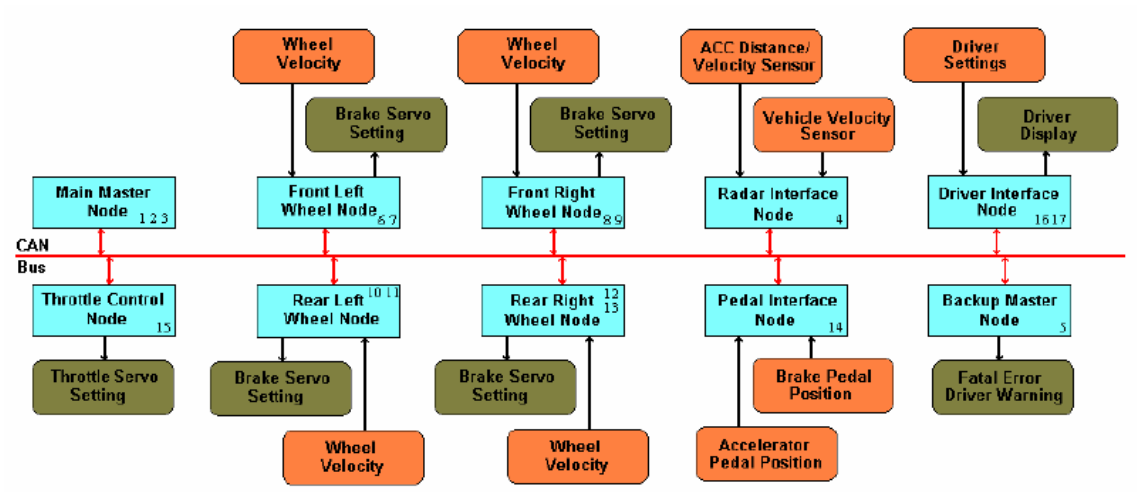


Figure G-3: The 10-nodes ACC implementation

Three systems

One consequence of the use of the SBS methods is that a limit is placed on the amount of user data that may be transferred in each CAN frame, since the encoding process requires two data bytes (see Appendix E).

The original system design had to be altered to accommodate the reduced data payload. This, in turn, resulted in a reduction in the sampling rate of the system traction controller. To enable a meaningful comparison, measurements were also taken for an uncompensated implementation using this reduced sampling rate.

To summarise, the three sets of results were obtained. These are labelled as follows:

- **Original:** the original 8-byte system with no SBS.
- **Uncompensated:** a 6-byte system with no SBS.
- **Compensated:** a 6-byte system with SBS.

Experimental methodology

Jitter measurements

To obtain jitter measurements, the latency between Master and Slave clock ticks was recorded for a period of 10,000 samples for each system. The experimental methodology used here is very similar to that described in Chapter 11. However, here, the delay was measured between the Master ISR and the Slave ISR. This implies that

the “worst-case transmission time” (WCTT) here was represented by the longest delay between the occurrence of a clock Tick on the Master node and the corresponding Tick on the Slaves. Apart from this, the same methodology outlined in Chapter 11 was used. Note that the jitter values presented here reflect the impact of bit-stuffing only.

Jerk and IAE measurement

To provide an indication of the control performance of each system, the maximum positive and negative vehicle ‘jerk’ (rate of change of acceleration) was recorded over a 300 second test period in which the ACC system was put through a series of typical manoeuvres. The jerk was averaged over a 1-second time period in accordance with ISO test specifications (ISO, 2003).

In addition to measuring the vehicle ‘jerk’, the performance of the vehicle while executing speed, and time-gap control was recorded. The Integral of Absolute Error (IAE) criterion was used to provide the performance measure in this case, as defined in Equation G-1. The IAE represents the error between the measured speed (or time-gap) and the reference one, with the test duration, *T*, equal to 300 seconds.

$$IAE = \int_0^T \left| e(t) \right| dt$$

Equation G-1

Each velocity test was for a speed setpoint of 70 MPH. Each distance test was performed whilst following a lead vehicle at 50 MPH (distance setpoint of 33.53 m for 1.5 s headway).

Results

System performance

Using the experimental methodology outlined in the previous section, the results obtained from the described ACC case study are presented in the following table.

Table G-1: Results from the 10-node ACC study

Test	Original	Uncompensated	Compensated
IAE Velocity	1.64	1.68	1.53
IAE Distance	11.60	11.84	10.98
Ave. Jitter (μs)	3.66	3.83	2.40
Diff. Jitter (μs)	23.53	24.03	11.20
WCTT (μs)	339.17	339.57	310.77
Max Pos Jerk (m/s^3)	2.24	2.39	2.37
Max Neg Jerk (m/s^3)	-1.81	-1.75	-1.60

Note that each of the three tests was repeated three times, and the results obtained were averaged. Note that the IAE measurements are “unit less” values, and are best viewed as a performance measure (the lower the better).

It can be seen from the results that the measured WCTT, average and difference jitter have all been reduced considerably by the compensation technique. For example, the overall reduction in the difference jitter was approximately 50%. When comparing the control behaviour of the compensated system to that of the original system, it can be seen that the performance has improved in all areas, except in the case of positive jerk. When comparing the uncompensated system to the original system, it is clear that the control performance of the uncompensated system is comparatively poor: this is a direct consequence of the 25% reduction in the data bandwidth of the network. However, when the compensated and uncompensated systems (with the same bandwidth restrictions) are compared, the use of compensation is seen to improve performance in all areas (including positive jerk).

The following tables show the results of jitter, IAE distance and velocity, and jerk from the three versions of the ACC system considered in this study.

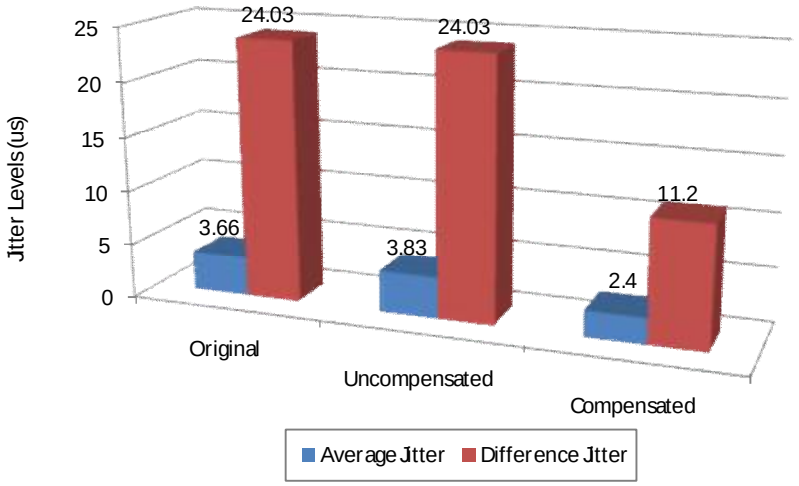


Figure G-4: Jitter levels for all systems.

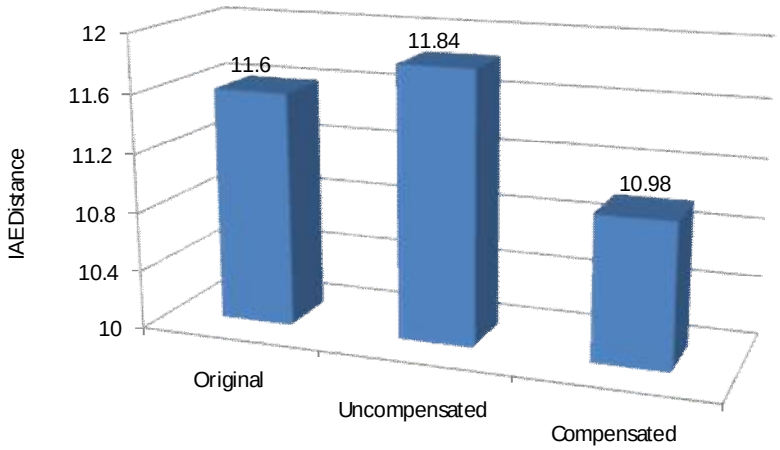


Figure G-5: IAE distance for all systems.

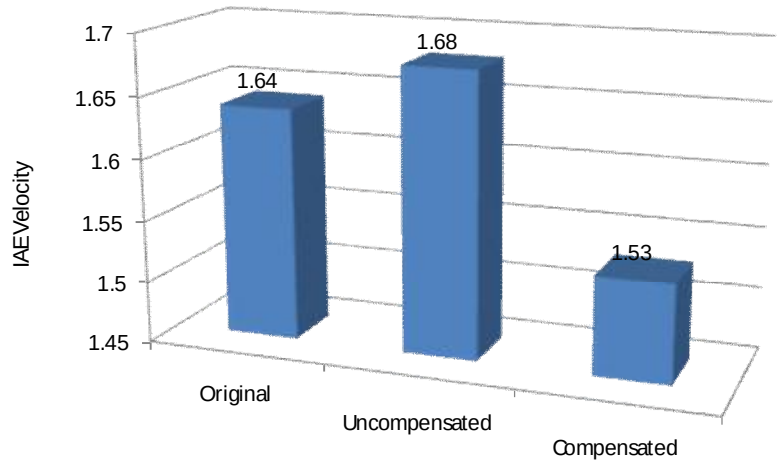


Figure G-6: IAE velocity for all systems.

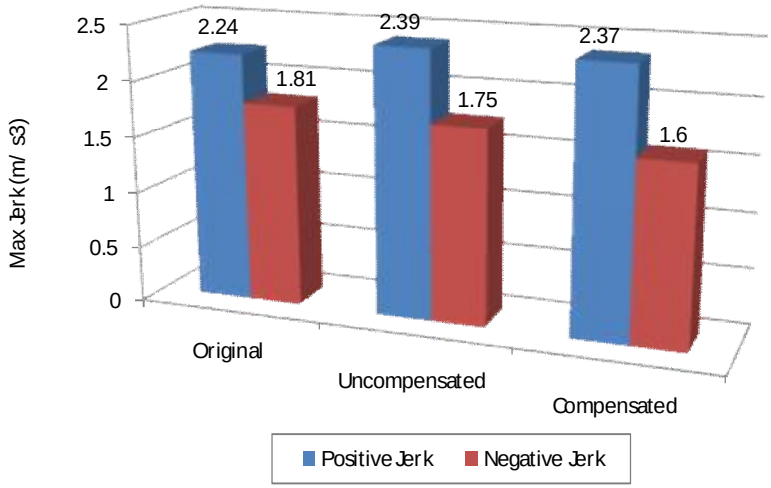


Figure G-7: Positive and negative jerk for all systems.

Memory and CPU requirements

The bit stuffing (SBS encoding) operation took an average of 0.7 ms on the C167 processors used, and the corresponding de-stuffing (SBS decoding) operation had an average duration of 0.6 ms. The extra RAM required by the SBS technique was 72 bytes and 56 bytes for the Master and Slave, respectively. The corresponding ROM (data memory) increases were found to be 1,317 and 985 bytes respectively, for the Master and Slave. Please note that the C167 boards used in this study have 256 kBytes ROM and 256 kBytes RAM (PhyCORE, 2003). The overall increases in memory do not, therefore, represent large percentages of the available resources.

Appendix H

Selective code listings

TTC-MTI scheduler

IRQ Wrapper

```
/*-------------------------------------------------------------------*-
   IRQ Wrapper (v1.00)
-*-------------------------------------------------------------------*/
   .text
   .arm
   .code 32

/* ------ Global function prototypes -------------------------------*/
   .global    IRQ_Wrapper

/* ------ LPC2106 Timer 1 Interrupt Register Address ---------------*/
   .equ       T1IR, 0xE0008000

    .equ    Mode_USR,       0x10
        .equ    Mode_FIQ,       0x11
        .equ    Mode_IRQ,       0x12
        .equ    Mode_SVC,       0x13
        .equ    Mode_ABT,       0x17
        .equ    Mode_UND,       0x1B
        .equ    Mode_SYS,       0x1F

        .equ    I_Bit,          0x80    /* when I bit is set, IRQ is disabled */
        .equ    F_Bit,          0x40    /* when F bit is set, FIQ is disabled */

/*-------------------------------------------------------------------*-
  void IRQ_Wrapper (void)
-*-------------------------------------------------------------------*/
IRQ_Wrapper:
/* Save regs and create stack frame */
      MOV            R12,R13
      STMDB   R13!,{R0-R12,R14}
      MOV            R11,R12

      /* Check if Timer 1 Match Register 0 generated interrupt */
      LDR     R2,=T1IR
      LDR     R3,[R2]
      STR            R3,[R2]
      AND     R4,R3,#0xf
      CMP     R4,#1

      /* If T1IR = 1 call SCH_Tick_Update */
      BLEQ    SCH_Tick_Update
      CMP     R4,#1
      BEQ            IRQ_End_If

      /* Else call SCH_Task_Update */
      BL             SCH_Task_Update

IRQ_End_If:

/* Restore registers from stack frame */
      LDMDB   R11,{R0-R11,R13,R14}

/* Load return address register with pointer to Dispatch */
      MSR     CPSR_c, #Mode_SYS
      LDR            R14,=mTask
```

```
      LDR            R14,[R14]
      LDR            R13,=StackP
      LDR            R13,[R13]
      MSR     CPSR_c, #Mode_FIQ|I_Bit|F_Bit

/* Load return address register with pointer to Task */
      LDR            R8,=cTask
      LDR            R8,[r8]

/* Return from Interrupt */
      SUBS    PC,R8,#0
```

Main code

```
/*-------------------------------------------------------------------*-
   Main.C (v1.00)
-*-------------------------------------------------------------------*/

#include "Main.h"
#include "system_init.h"
#include "Sch_2100.h"
#include "tasks.h"
#include "task_RT.h"
#include "mc_cal.h"

/*-------------------------------------------------------------------*-
  int main (void)
-*-------------------------------------------------------------------*/
int main (void)
   {
   // Set up PLL, VPB divider and MAM (disabled)
   System_Init();

   // Set up the scheduler
   SCH_Init(5);

   // Prepare for the 'Flash_LED' task
   LED_FLASH_Init();

   // Add tasks
   // Delay and Period values are in *ticks*
   SCH_Add_Task(Task_A, 0, 1);
   SCH_Add_Task(Task_B, 0, 1);
   SCH_Add_Task(Task_C, 0, 1);

   // Input duration for tasks
   // Values are in *microseconds*
   SCH_Task_WCET(Task_A, 2000);
   SCH_Task_WCET(Task_B, 1000);
   SCH_Task_WCET(Task_C, 1000);

   // Calculate the Scheduler Major Cycle
   Calc_Sch_Major_Cycle(SCH_MAX_TASKS);

   // Calculate the required release time for each task
   Calculate_Task_RT();

   // Start the scheduler
   SCH_Start();

   // The scheduler may enter idle mode at this point (if used)
   SCH_Go_To_Sleep();

   return 0;
   }
```

Scheduler code

```
/*-------------------------------------------------------------------*-
     sch_2100.H (v1.00)
-*-------------------------------------------------------------------*/

#ifndef _SCH_2100_H
#define _SCH_2100_H

#include "main.h"
#include "lpc21xx.h"

// ------ Public function prototypes -------------------------------

void SCH_Init(const int);
void SCH_Start(void);
void SCH_Update (void);

// ------ Public data type declarations ----------------------------

// Total memory per task is >>> bytes
typedef struct
   {
   // Pointer to the task (must be a 'void (void)' function)
   void (*pTask) (void);

   // Delay (ticks) until the function will (next) be run
   // - see SCH_Add_Task() for further details
   tWord Delay;

   // Interval (ticks) between subsequent runs.
   // - see SCH_Add_Task() for further details
   tWord Period;

   // Incremented (by scheduler) when task is due to execute
   tWord RunMe;

   // Task worst case execution time (in microseconds): the longest time required by
the processor to execute the task
   tWord wcet;

   // Task release time (in microseconds): the time at which can execute with no
jitter
   tWord Rls_time;

   } sTask;

// ------ Public function prototypes -------------------------------

// Core scheduler functions
void  SCH_Dispatch_Tasks(void);
tByte SCH_Add_Task(void (*) (void), const tWord, const tWord);
tByte SCH_Task_WCET(void (*) (void), const tWord);
int   SCH_Delete_Task(const tByte);
void  SCH_Go_To_Sleep(void);

// ------ Public constants -----------------------------------------

// The maximum number of tasks required at any one time
// during the execution of the program
//
// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define SCH_MAX_TASKS   (3)

#endif
```

```
/*-------------------------------------------------------------------*-
   sch_2100.c (v1.00)
-*-------------------------------------------------------------------*/

#include "sch_2100.h"
```

```
// ------ Public variable definitions -----------------------------

// Used to report errors, if required
tByte Error_code_G;

// Current task index
tByte Index_G;

// The current mode
tByte Mode_G;

long StackP;

// Pointer to the task (must be a 'void (void)' function)
void (*cTask) (void);
void (*mTask) (void);

// ------ Private variable definitions ----------------------------

// The array of tasks (see Sch2100.H)
sTask SCH_tasks_G[SCH_MAX_TASKS];

tByte runme[SCH_MAX_TASKS + 1];

// ------ Private function prototypes ----------------------------

void  SCH_Tick_Update(void);
void  SCH_Task_Update(void);
void SCH_End_Task(void);
void SCH_Task_Overrun_Update(void);

/*------------------------------------------------------------------*-
  SCH_Init()

  Scheduler initialisation function.  Prepares scheduler
  data structures and sets up timer interrupts at required rate.

  You must call this function before using the scheduler.
-*------------------------------------------------------------------*/
void SCH_Init(const int TICK_LEN_mS)
   {
   tByte i;

   for (i = 0; i < SCH_MAX_TASKS; i++)
      {
      SCH_Delete_Task(i);
        }

   // Reset the global error variable
   // - SCH_Delete_Task() will generate an error code,
   //   (because the task array is empty)
   Error_code_G = 0;

   // Initialise the mode to calculating mode
   mTask = SCH_Go_To_Sleep;

   /*-------------------------------------------
                                    Timer 1
   -------------------------------------------*/

   // Set prescaler to 0
   T1PR = (PCLK / 1000000) - 1;

   // Setup Match Register 0 - tick in ms multiples
   T1MR0 = ((1000) * TICK_LEN_mS) - 1;

   // Interrupt on match, and automatically restart counter
   T1MCR = 0x03;

   // Set Timer 1 to FIQ
   VICIntSelect = 0x20;

   // Enable Timer 1 interrupt
   VICIntEnable |= 0x20;
```

```
   // Enable Timer 1 to operate in idle mode
   PCONP |= 0x04;
   }

/*-------------------------------------------------------------------*-
  SCH_Start()

  Starts the scheduler, by enabling interrupts.

  NOTE: Usually called after all regular tasks are added,
  to keep the tasks synchronised.

  NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!
-*-------------------------------------------------------------------*/
void SCH_Start(void)
   {
   T1TCR |= 0x01;    // Counter enable (Timer Counter Register)
   StackP = (int) sp;
   }

/*-------------------------------------------------------------------*-
  SCH_Tick_Update()

  This is the scheduler ISR.  It is called at a rate
  determined by the timer settings in the 'init' function.
-*-------------------------------------------------------------------*/
void SCH_Tick_Update(void)
   {
   tByte i = 0;
   tByte Index;

        // Set tick jitter pin to 1
        IOSET0 = Tick_Jitter_pin;
        // Set tick jitter pin to 0
        IOCLR0 = Tick_Jitter_pin;

      // Go through the task array
      for (Index = 0; Index < SCH_MAX_TASKS ; Index++)
         {
              // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
                           // indicate the task is to be run
                           runme[i++] = Index;

                        if (SCH_tasks_G[Index].Period != 0)
                        {
                        // Schedule period tasks to run again
                        SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                        }
                         else
                        {
                        // Delete one-shot tasks
                        SCH_tasks_G[Index].pTask  = 0;
                        }
                       }
                 }
          }

             // Indicate no more tasks in runme queue
             runme[i] = SCH_MAX_TASKS;



             /* If there are tasks in current tick interval */
             if (runme[0] != SCH_MAX_TASKS)
                    {
                    // Setup Match Register 1 - interrupt in uS from tick
                    T1MR1 = SCH_tasks_G[runme[0]].Rls_time + 100*(runme[0]+1);

                 // Interrupt on match 1
                     T1MCR |= 0x08;
                     }

   // Return to sleep
```

```
   cTask = SCH_Go_To_Sleep;

   // Reset the task index
   Index_G = 0;
   }


/*------------------------------------------------------------------*-
  SCH_Task_Update()
-*------------------------------------------------------------------*/
void SCH_Task_Update(void)
{

   // Run task after this function
   cTask = SCH_tasks_G[runme[Index_G]].pTask;

   // Setup Match Register 1 - for the next task
   T1MR1 = SCH_tasks_G[runme[Index_G+1] % SCH_MAX_TASKS].Rls_time +
100*(runme[Index_G]+2);


   Index_G++;


   // Disable Interrupt on match 1
   T1MCR &= 0xFFFFFFF7;

   // Enable Interrupt on match 1
   T1MCR |= (1 & (tLong) (runme[Index_G] != SCH_MAX_TASKS)) << 3;

//if(Index_G == 2){IOSET0 = Task_Jitter_pin;
//      IOCLR0 = Task_Jitter_pin;}

}

/*------------------------------------------------------------------*-
  SCH_Add_Task()

  Causes a task (function) to be executed at regular intervals
  or after a user-defined delay
-*------------------------------------------------------------------*/
tByte SCH_Add_Task(void (* pFunction)(),
                   const tWord DELAY,
                   const tWord PERIOD)
   {
   tByte Index = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
      {
      Index++;
      }

   // Have we reached the end of the list?
   if (Index == SCH_MAX_TASKS)
      {
      // Task list is full
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

      // Also return an error code
      return SCH_MAX_TASKS;
      }

   // If we're here, there is a space in the task array
   SCH_tasks_G[Index].pTask   = pFunction;
   SCH_tasks_G[Index].Delay   = DELAY + 1;
   SCH_tasks_G[Index].Period  = PERIOD;
   SCH_tasks_G[Index].Rls_time = 0;

   return Index; // return position of task (to allow later deletion)
   }

/*------------------------------------------------------------------*-
  SCH_Task_WCET()
```

```
-*---------------------------------------------------------------------*/

tByte SCH_Task_WCET(void (* pFunction)(),
                    const tWord WCET)
   {
   tByte Index = 0;

   for (Index=0; Index<SCH_MAX_TASKS; Index++)
     {
     if(SCH_tasks_G[Index].pTask  == pFunction)
       {
       SCH_tasks_G[Index].wcet  = WCET;
         break;
       }
     }

   return Index; // return position of task (to allow later deletion)


   }

/*---------------------------------------------------------------------*-
  SCH_Delete_Task()

  RETURN VALUE:  RETURN_ERROR or RETURN_NORMAL
-*---------------------------------------------------------------------*/
int SCH_Delete_Task(const tByte TASK_INDEX)
   {
   int Return_code;

   if (SCH_tasks_G[TASK_INDEX].pTask == 0)
      {
      // No task at this location...
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

      // ...also return an error code
      Return_code = RETURN_ERROR;
      }
   else
      {
      Return_code = RETURN_NORMAL;
      }

   SCH_tasks_G[TASK_INDEX].pTask   = 0x0000;
   SCH_tasks_G[TASK_INDEX].Delay   = 0;
   SCH_tasks_G[TASK_INDEX].Period  = 0;

   return Return_code;       // return status
   }

/*---------------------------------------------------------------------*-
  SCH_Go_To_Sleep()
-*---------------------------------------------------------------------*/
void SCH_Go_To_Sleep()
   {
   PCON = 1;
   }
```

Major cycle and task RT calculations

```
/*---------------------------------------------------------------------*-
      MC_Cal.H (v1.00)
-*---------------------------------------------------------------------*/

#include "main.h"

// Scheduler major cycle
tWord SCH_Major_Cycle_G;
```

```
// Function prototype
tWord Calc_Sch_Major_Cycle(tWord);
tWord LCM2Numbers(tWord, tWord);

/*-----------------------------------------------------------------*-
      MC_Cal.C (v1.00)
-*-----------------------------------------------------------------*/
#include "mc_cal.h"
#include "sch_2100.h"

// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// Calculate the scheduler major cycle

tWord Calc_Sch_Major_Cycle(tWord n)
  {
   tWord i;

   SCH_Major_Cycle_G = SCH_tasks_G[0].Period;

   for(i = 1; i < n; i++)
    {
     SCH_Major_Cycle_G = LCM2Numbers( SCH_Major_Cycle_G, SCH_tasks_G[i].Period);
    }//end of for(i..

   return SCH_Major_Cycle_G;

  }//end of function CalcTestPeriod(int n)

/*-----------------------------------------------------------------*-
       LCM2Numbers()
-*-----------------------------------------------------------------*/

tWord LCM2Numbers(tWord n1,tWord n2)
{
 //calc the LCM of any two numbers
 tWord LCM,product,temp;

    product=n1*n2;

      do
              {
              if ( n1 < n2 )
                      {
                      temp=n1;
                      n1=n2;
              n2=temp;
                      }//end of if(n1...

              else
                      {
                      }//end of else-->if(n1...
                      n1=n1%n2;

              }while (n1);

      // now n2 contains the GCD of the two numbers
    LCM= product/n2;

    return LCM;

}//end of function LCM2Numbers(....


/*-----------------------------------------------------------------*-
      Task_RT.H (v1.00)
-*-----------------------------------------------------------------*/

#include "main.h"

// Function prototype
void Calculate_Task_RT(void);

/*-----------------------------------------------------------------*-
Task_RT.C (v1.00)
-*-----------------------------------------------------------------*/
```

```
#include "task_rt.h"
#include "sch_2100.h"
#include "mc_cal.h"

// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];


void Calculate_Task_RT(void)
  {
    tWord Index, k, Temp_Rls_time=0, Ticks ;
      tWord Task_Schedule[SCH_MAX_TASKS][SCH_Major_Cycle_G] ;     // adjust to be
larger than or equal to major cycle


    // Fill the array Task_Schedule[] which contains information about the task
schedule
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
        {
         //SCH_tasks_G[Index].Rls_time = 0;

         // check higher-periority tasks in each tick
         for (Ticks = 0; Ticks < SCH_Major_Cycle_G; Ticks++)
           {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
                {
                         // The task is due to run
                             // Don't run the task at this stage; instead set a flag

                         Task_Schedule[Index][Ticks] = 1;

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
                }
                      else
                             {
                             Task_Schedule[Index][Ticks] = 0;
                             }
             }
         } // end for (Index ...)
     }       // end for (Runs ...)


  // Calculate the RT required for each task

     for (Index = 1; Index < SCH_MAX_TASKS; Index++)
       {

          for (Ticks = 0; Ticks < SCH_Major_Cycle_G; Ticks++)
            {
                     Temp_Rls_time = 0;

          if (Task_Schedule[Index][Ticks] == 1)
                {
                for (k = 0; k < Index ; k++)
                  {
                      if (Task_Schedule[k][Ticks] == 1)
                        {
                             Temp_Rls_time += SCH_tasks_G[k].wcet;

                    // store the maximum (release) time before the current task

                             if((SCH_tasks_G[Index].Rls_time < Temp_Rls_time))
                             {
```

```
                            SCH_tasks_G[Index].Rls_time = Temp_Rls_time;

                            }

              }     // end if()
            }       // end for(k)
          } // end if(Task)
        }   // end for(j)


            //SCH_tasks_G[Index].Rls_time *= (PCLK / 1000000U);
      }     // end for(Index)

  }
```

```
/*-------------------------------------------------------------------*-
      Task_RT.H (v1.00)
-*-------------------------------------------------------------------*/

#include "main.h"

// Function prototype
void Calculate_Task_RT(void);

/*-------------------------------------------------------------------*-
Task_RT.C (v1.00)
-*-------------------------------------------------------------------*/

#include "task_rt.h"
#include "sch_2100.h"
#include "mc_cal.h"

// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];


void Calculate_Task_RT(void)
  {
    tWord Index, k, Temp_Rls_time=0, Ticks ;
      tWord Task_Schedule[SCH_MAX_TASKS][SCH_Major_Cycle_G] ;     // adjust to be
larger than or equal to major cycle


    // Fill the array Task_Schedule[] which contains information about the task
schedule
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
        {
         //SCH_tasks_G[Index].Rls_time = 0;

         // check higher-periority tasks in each tick
         for (Ticks = 0; Ticks < SCH_Major_Cycle_G; Ticks++)
           {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
                {
                         // The task is due to run
                            // Don't run the task at this stage; instead set a flag

                         Task_Schedule[Index][Ticks] = 1;

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
                }
                      else
```

```
                            {
                            Task_Schedule[Index][Ticks] = 0;
                            }
              }
           } // end for (Index ...)
        }       // end for (Runs ...)


  // Calculate the RT required for each task

      for (Index = 1; Index < SCH_MAX_TASKS; Index++)
        {

          for (Ticks = 0; Ticks < SCH_Major_Cycle_G; Ticks++)
            {
                     Temp_Rls_time = 0;

          if (Task_Schedule[Index][Ticks] == 1)
                {
                for (k = 0; k < Index ; k++)
                  {
                      if (Task_Schedule[k][Ticks] == 1)
                        {
                             Temp_Rls_time += SCH_tasks_G[k].wcet;

                       // store the maximum (release) time before the current task

                              if((SCH_tasks_G[Index].Rls_time < Temp_Rls_time))
                              {
                              SCH_tasks_G[Index].Rls_time = Temp_Rls_time;

                              }

                }     // end if()
              }       // end for(k)
            } // end if(Task)
          }  // end for(j)


              //SCH_tasks_G[Index].Rls_time *= (PCLK / 1000000U);
        }     // end for(Index)

  }
```

TTC-SCC1 scheduler

Master code

```
// ------ Public variable definitions -----------------------------

// Four bytes of data (plus ID information) are sent
tByte Tick_message_data_G[NUMBER_OF_SLAVES][8];
tByte Ack_message_data_G[NUMBER_OF_SLAVES][4];

// ------ Public variable declarations ----------------------------

// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

long int Tick_count_G;
// ------ Private variable definitions ----------------------------

static tByte Slave_index_G = 0;
static bit First_ack_G = 1;

// ------ Private function prototypes -----------------------------

static void SCC_A_MASTER_Send_Tick_Message(const tByte);
static bit  SCC_A_MASTER_Process_Ack(const tByte);
static void SCC_A_MASTER_Shut_Down_the_Network(void);
```

```
static void SCC_A_MASTER_Enter_Safe_State(void);
static tByte  SCC_A_MASTER_Start_Slave(const tByte); // reentrant;


// ------ Private constants ---------------------------------------

// Slave IDs may be any NON-ZERO tByte value (all must be different)
// NOTE: Do *not* use ID 0x00 (used to start slaves)
static const tByte MAIN_SLAVE_IDs[NUMBER_OF_SLAVES] = {0x02};

// If there are no backup nodes, this array should be identical
// to the array MAIN_SLAVE_IDs[] - see above.
static const tByte BACKUP_SLAVE_IDs[NUMBER_OF_SLAVES] = {0x02};

#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)

// ------ Private variables ---------------------------------------

// Current slave IDs (Slave or Backup)
static tByte Current_Slave_IDs_G[NUMBER_OF_SLAVES] = {0};


/*---------------------------------------------------------------------*-

  SCC_A_MASTER_Update_T6

  This is the scheduler ISR.  It is called at a rate determined by
  the timer settings in SCC_A_MASTER_Init_T6().  This version is
  triggered by Timer 6 interrupts: timer is automatically reloaded.

-*---------------------------------------------------------------------*/

void SCC_A_MASTER_Update_T6(void) interrupt INTERRUPT_Timer_6_Overflow
   {
   tByte Previous_slave_index;
   bit Slave_replied_correctly;


   // Clear T6 interrupt request flag
   T6IR = 0;


   // Default
   Network_error_pin = NO_NETWORK_ERROR;

   // Keep track of the current slave
   Previous_slave_index = Slave_index_G;   // First value of prev slave is 0...

   if (++Slave_index_G >= NUMBER_OF_SLAVES)
      {
      Slave_index_G = 0;
      }

   // Check that the appropriate slave responded to the previous message:
   // (if it did, store the data sent by this slave)
   if (SCC_A_MASTER_Process_Ack(Previous_slave_index) == RETURN_ERROR)
      {
      Error_code_G = ERROR_SCH_LOST_SLAVE;
      Network_error_pin = NETWORK_ERROR;

      // If we have lost contact with a slave, we attempt to
      // switch to a backup device (if one is available)
      if (Current_Slave_IDs_G[Slave_index_G] != BACKUP_SLAVE_IDs[Slave_index_G])
         {
         // There is a backup available: switch to backup and try again
         Current_Slave_IDs_G[Slave_index_G] = BACKUP_SLAVE_IDs[Slave_index_G];
         }
      else
         {
         // There is no backup available (or we are already using it)
         // Try main device.
         Current_Slave_IDs_G[Slave_index_G] = MAIN_SLAVE_IDs[Slave_index_G];
         }

      // Try to connect to the slave
```

```
      Slave_replied_correctly =
SCC_A_MASTER_Start_Slave(Current_Slave_IDs_G[Slave_index_G]);

      if (!Slave_replied_correctly)
         {
         // No backup available (or backup failed too) - we shut down
         // OTHER BEHAVIOUR MAY BE MORE APPROPRIATE IN YOUR APPLICATION
         SCC_A_MASTER_Shut_Down_the_Network();
         }
      }

   // Send 'tick' message to all connected slaves
   // (sends one data byte to the current slave)
   SCC_A_MASTER_Send_Tick_Message(Slave_index_G);


   // Check the last error codes on the CAN bus via the status register
   if ((C1CSR & 0x0700) != 0)
      {
      Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
      Network_error_pin = NETWORK_ERROR;

      // See Infineon C167CR manual for error code details
      CAN_error_pin0 = ((C1CSR & 0x0100) == 0);
      CAN_error_pin1 = ((C1CSR & 0x0200) == 0);
      CAN_error_pin2 = ((C1CSR & 0x0400) == 0);
      }
   else
      {
      CAN_error_pin0 = 1;
      CAN_error_pin1 = 1;
      CAN_error_pin2 = 1;
      }

   Tick_count_G++;


   }

/*-------------------------------------------------------------------*-

  SCC_A_MASTER_Send_Tick_Message()

  This function sends a tick message, over the CAN network.
  The receipt of this message will cause an interrupt to be generated
  in the slave(s): this invoke the scheduler 'update' function
  in the slave(s).

-*-------------------------------------------------------------------*/
void SCC_A_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
   {

   // Find the slave ID for this slave
   // ALL SLAVES MUST HAVE A UNIQUE (non-zero) ID
   tByte Slave_ID = (tByte) Current_Slave_IDs_G[SLAVE_INDEX];
   CAN_OBJ[0].Data[0] = Slave_ID;


   // Fill the data fields
   CAN_OBJ[0].Data[1] = Tick_message_data_G[0][1];
   CAN_OBJ[0].Data[2] = Tick_message_data_G[0][2];
   CAN_OBJ[0].Data[3] = Tick_message_data_G[0][3];
   CAN_OBJ[0].Data[4] = Tick_message_data_G[0][4];
   CAN_OBJ[0].Data[5] = Tick_message_data_G[0][5];
   CAN_OBJ[0].Data[6] = Tick_message_data_G[0][6];
   CAN_OBJ[0].Data[7] = Tick_message_data_G[0][7];

   // Send the message on the CAN bus
   CAN_OBJ[0].MCR = 0xEFFF;  // Set TXRQ
   }

/*-------------------------------------------------------------------*-

  SCC_A_MASTER_Process_Ack()

  Make sure the slave (SLAVE_ID) has acknowledged the previous
  message that was sent.  If it has, extract the message data
```

```
  from the USART hardware: if not, call the appropriate error
  handler.

  PARAMS:   The index of the slave.

  RETURNS:  RETURN_NORMAL - Ack received (data in Ack_message_data_G)
            RETURN_ERROR  - No ack received (-> no data)

-*--------------------------------------------------------------------*/

bit SCC_A_MASTER_Process_Ack(const tByte SLAVE_INDEX)
{
   tByte Ack_ID, Slave_ID;

   // First time this is called there is no ack tick to check
   // - we simply return 'OK'
   if (First_ack_G)
   {
      First_ack_G = 0;
      return RETURN_NORMAL;
   }

   // Is the NEWDAT flag set?
   if ((CAN_OBJ[1].MCR & 0x0300) == 0x0200)
   {
      // An Ack message was received
      // - Extract the data
      Ack_ID = CAN_OBJ[1].Data[0];   // Get data byte 0

      Ack_message_data_G[SLAVE_INDEX][0] = CAN_OBJ[1].Data[1];
      Ack_message_data_G[SLAVE_INDEX][1] = CAN_OBJ[1].Data[2];
      Ack_message_data_G[SLAVE_INDEX][2] = CAN_OBJ[1].Data[3];
      Ack_message_data_G[SLAVE_INDEX][3] = CAN_OBJ[1].Data[4];

      CAN_OBJ[1].MCR = 0xFDFF;  // Clear NEWDAT flag

      // Find the slave ID for this slave
      Slave_ID = (tByte) Current_Slave_IDs_G[SLAVE_INDEX];

      if (Ack_ID == Slave_ID)
      {
         return RETURN_NORMAL;
      }
   }

   // No message, or ID incorrect
   return RETURN_NORMAL;
}
```

```
// ------ Public variable definitions -----------------------------

// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];

extern long int Tick_count_G;

extern tByte Tick_message_data_G[NUMBER_OF_SLAVES][8];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
tByte Error_code_G = 0;

// ------ Private function prototypes -----------------------------

void SCH_Go_To_Sleep(void);

// ------ Private variables ---------------------------------------

// Keeps track of time since last error was recorded (see below)
static tWord Error_tick_count_G;

// The code of the last error (reset after ~1 minute)
```

```
static tByte Last_error_code_G;

/*------------------------------------------------------------------*-

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.

-*------------------------------------------------------------------*/
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;
   bit Update_again = 0;

   do {
      // NOTE: calculations are in *TICKS* (not milliseconds)
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
         {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
               // The task is due to run
               (*SCH_tasks_G[Index].pTask)();  // Run the task

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask = 0;
                  }
               }
            }
         }

      // Disable Timer 6 interrupt
      T6IE = 0;

      if (--Tick_count_G > 0)
         {
         Update_again = 1;
         }
      else
         {
         Update_again = 0;
         }

      // Re-enable Timer 6 interrupt
      T6IE = 1;

      } while (Update_again);


// Fill the message data with random bytes

     for (i=1; i <= 7; i++)
       {
         Tick_message_data_G[0][i] = (tByte) (rand() % 255); // random data
       }

//-------------------------------------------------------------
// Encoding process – for jitter reduction - is carried out here

Encode_data();

//-------------------------------------------------------------

   // Report system status
   SCH_Report_Status();
```

Slave code

```
// ------ Public variable definitions -----------------------------

// Data sent from the master to this slave
tByte Tick_message_data_G[8];

tByte Dummy_tick_data_G[8];  // dummy array

// Data sent from this slave to the master
// - data may be sent on, by the master, to another slave
tByte Ack_message_data_G[4] = {'1','1','1','1'};

// ------ Public variable declarations ----------------------------

// The array of tasks (see Sch51.c)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// The error code variable (see Sch51.c)
extern tByte Error_code_G;

long int Tick_count_G;

// ------ Private function prototypes -----------------------------
static void  SCC_A_SLAVE_Enter_Safe_State(void);

static void  SCC_A_SLAVE_Send_Ack_Message_To_Master(void);
static tByte SCC_A_SLAVE_Process_Tick_Message(void);

static void  SCC_A_SLAVE_Watchdog_Init(void);
static void  SCC_A_SLAVE_Watchdog_Refresh(void);

// ------ Private constants ---------------------------------------

// Each slave (and backup) must have a unique (non-zero) ID
#define SLAVE_ID 0x02

#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)

/*---------------------------------------------------------------------*-

  SCC_A_SLAVE_Update

  This is the scheduler ISR.  It is called at a rate
  determined by the timer settings in SCC_A_SLAVE_Init().

  This Slave is triggered by CAN interrupts.

-*---------------------------------------------------------------------*/
void SCC_A_SLAVE_Update(void) interrupt INTERRUPT_CAN_C167CR
   {
//   tByte Index;
   tWord uwIntID;
      tWord i;

   // *** DON'T NEED THIS ***
   uwIntID = C1IR & 0x00ff;

   if ((uwIntID & 0x00ff) != 3)
    {
    // Only interested in Message Object 1 Interrupt
      return;
      }

    RECV_LED_pin = 0;

   // Reset this when tick is received
   Network_error_pin = NO_NETWORK_ERROR;

   // Check tick data - send ack if necessary
   // NOTE: 'START' message will only be sent after a 'time out'
```

```
   if (SCC_A_SLAVE_Process_Tick_Message() == SLAVE_ID)
      {
      SCC_A_SLAVE_Send_Ack_Message_To_Master();

      // Feed the watchdog ONLY when a *relevant* message is received
      // (noise on the bus, etc, will not stop the watchdog...)
      //
      // START messages will NOT refresh the slave
      // - Must talk to every slave at regular intervals
      SCC_A_SLAVE_Watchdog_Refresh();
      }

   // Check the last error codes on the CAN bus via the status register
   if ((C1CSR & 0x0700) != 0)
      {
      Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
      Network_error_pin = NETWORK_ERROR;

      // See Infineon C167CR manual for error code details
      CAN_error_pin0 = ((C1CSR & 0x0100) == 0);
      CAN_error_pin1 = ((C1CSR & 0x0200) == 0);
      CAN_error_pin2 = ((C1CSR & 0x0400) == 0);
      }
   else
      {
      CAN_error_pin0 = 1;
      CAN_error_pin1 = 1;
      CAN_error_pin2 = 1;
      }

      Tick_count_G++;

      for(i=0; i<700; i++);
      RECV_LED_pin = 1;
   }


/*-------------------------------------------------------------------*-

  SCC_A_SLAVE_Process_Tick_Message()

  The ticks messages are crucial to the operation of this shared-clock
  scheduler: the arrival of a tick message (at regular intervals)
  invokes the 'Update' ISR, that drives the scheduler.

  The tick messages themselves may contain data.  These data are
  extracted in this function.

-*-------------------------------------------------------------------*/
tByte SCC_A_SLAVE_Process_Tick_Message(void)
   {
   tByte Tick_ID;

      if ((CAN_OBJ[0].MCR & 0x0c00) == 0x0800)  // if MSGLST set
         {
         // Indicates that the CAN controller has stored a new
         // message into this object, while NEWDAT was still set,
         // i.e. the previously stored message is lost.

         // We simply IGNORE this here and reset the flag
         CAN_OBJ[0].MCR = 0xf7ff;  // reset MSGLST
         }

   // The first byte is the ID of the slave for which the data are
   // intended
   Tick_ID = CAN_OBJ[0].Data[0];   // Get data byte 0 (Slave ID)

   if (Tick_ID == SLAVE_ID)
      {
      // Only if there is a match do we need to copy these fields
      Tick_message_data_G[0] = CAN_OBJ[0].Data[0];
      Tick_message_data_G[1] = CAN_OBJ[0].Data[1];
      Tick_message_data_G[2] = CAN_OBJ[0].Data[2];
      Tick_message_data_G[3] = CAN_OBJ[0].Data[3];
      Tick_message_data_G[4] = CAN_OBJ[0].Data[4];
      Tick_message_data_G[5] = CAN_OBJ[0].Data[5];
      Tick_message_data_G[6] = CAN_OBJ[0].Data[6];
```

```
      Tick_message_data_G[7] = CAN_OBJ[0].Data[7];
      }

   else
      {
      // Must do same processing to avoid jitter
      Dummy_tick_data_G[0] = CAN_OBJ[0].Data[0];
      Dummy_tick_data_G[1] = CAN_OBJ[0].Data[1];
      Dummy_tick_data_G[2] = CAN_OBJ[0].Data[2];
      Dummy_tick_data_G[3] = CAN_OBJ[0].Data[3];
      Dummy_tick_data_G[4] = CAN_OBJ[0].Data[4];
      Dummy_tick_data_G[5] = CAN_OBJ[0].Data[5];
      Dummy_tick_data_G[6] = CAN_OBJ[0].Data[6];
      Dummy_tick_data_G[7] = CAN_OBJ[0].Data[7];
        }
        CAN_OBJ[0].MCR = 0xFDFD;  // Reset NEWDAT and INTPND

   return Tick_ID;
   }

/*-------------------------------------------------------------------*-

  SCC_A_SLAVE_Send_Ack_Message_To_Master()

  Slave must send and 'Acknowledge' message to the master, after
  tick messages are received.  NOTE: Only tick messages specifically
  addressed to this slave should be acknowledged.

  The acknowledge message serves two purposes:
  [1] It confirms to the master that this slave is alive & well.
  [2] It provides a means of sending data to the master and - hence
      - to other slaves.

  NOTE: Data transfer between slaves is NOT permitted!

-*-------------------------------------------------------------------*/
void SCC_A_SLAVE_Send_Ack_Message_To_Master(void)
   {
   // First byte of message must be slave ID
   CAN_OBJ[1].Data[0] = SLAVE_ID;   // data byte 0

   CAN_OBJ[1].Data[1] = Ack_message_data_G[0];
   CAN_OBJ[1].Data[2] = Ack_message_data_G[1];
   CAN_OBJ[1].Data[3] = Ack_message_data_G[2];
   CAN_OBJ[1].Data[4] = Ack_message_data_G[3];

   // Send the message on the CAN bus
   CAN_OBJ[1].MCR = 0xE7FF; // Set TXRQ (send message)
   }
```

```
// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
tByte Error_code_G = 0;

extern long int Tick_count_G;

// ------ Private function prototypes -----------------------------

void SCH_Go_To_Sleep(void);

// ------ Private variables ---------------------------------------

// Keeps track of time since last error was recorded (see below)
static tWord Error_tick_count_G;

// The code of the last error (reset after ~1 minute)
static tByte Last_error_code_G;

// ------ Private constants ---
```

```
#define NO_NETWORK_ERROR (1)
#define NETWORK_ERROR (0)


/*------------------------------------------------------------------*-

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.

-*------------------------------------------------------------------*/
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;
   bit Update_again = 0;

   do {
      // NOTE: calculations are in *TICKS* (not milliseconds)
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
         {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
               // The task is due to run
               (*SCH_tasks_G[Index].pTask)();  // Run the task

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
               }
            }
         }


   // Check the last error codes on the CAN bus via the status register
   if ((C1CSR & 0x0700) != 0)
      {
      Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
      Network_error_pin = NETWORK_ERROR;

      // See Infineon C167CR manual for error code details
      CAN_error_pin0 = ((C1CSR & 0x0100) == 0);
      CAN_error_pin1 = ((C1CSR & 0x0200) == 0);
      CAN_error_pin2 = ((C1CSR & 0x0400) == 0);
      }
   else
      {
      CAN_error_pin0 = 1;
      CAN_error_pin1 = 1;
      CAN_error_pin2 = 1;
      }



      // Disable interrupts
      //ET2 = 0;

      if (--Tick_count_G > 0)
         {
         Update_again = 1;
         }
      else
         {
         Update_again = 0;
```

```
         }

      //ET2 = 1;

      } while (Update_again);


//-------------------------------------------------------------
// Decoding process – to recover original data - is carried out here

      Decode_Data();
//-------------------------------------------------------------

   // Report system status
   SCH_Report_Status();
```

Data coding techniques

Nolte XOR masking

Master code

```
// XOR all data bytes (except the Slave ID) with Nolte bit mask
       for (i=1; i <= 7; i++)
       {
       Tick_message_data_G[0][i] ^= 0x55;
       }
```

Slave code

```
for (i=1; i <= 7; i++)
       {
       Tick_message_data_G[i] ^= 0x55; // random data
       }
```

Frame-based XOR masking

Master code

```
if(Data_Config() == YES)
         {
              for (i=1; i <= 7; i++)
             {
             Tick_message_data_G[i] ^= 0x55; // random data
             }

           // send 0010 0000 to indicate that the frame has been masked
           Tick_message_data_G[0][0] = 0x20;
         }
```

```
tByte Data_Config(void)
{
      Consec_Bits = 0;
      Tick_message_data_G[0][0] = 0;

      frame = Tick_message_data_G[0][7];
      Prev_bit = frame & 0x1;
      z = 2;
      if (Check_Bit_Stuff(8,5) == NO)
```

```
        {
        frame = Tick_message_data_G[0][6];
        Prev_bit = Last_Bit;
        z = 1;
        if (Check_Bit_Stuff(8,5) == NO)
          {
          frame = Tick_message_data_G[0][5];
          Prev_bit = Last_Bit;
          z = 1;
          if (Check_Bit_Stuff(8,5) == NO)
            {
             frame = Tick_message_data_G[0][4];
             Prev_bit = Last_Bit;
             z = 1;
             if (Check_Bit_Stuff(8,5) == NO)
               {
                frame = Tick_message_data_G[0][3];
                Prev_bit = Last_Bit;
                z = 1;
                if (Check_Bit_Stuff(8,5) == NO)
                  {
                  frame = Tick_message_data_G[0][2];
                  Prev_bit = Last_Bit;
                  z = 1;
                  if (Check_Bit_Stuff(8,5) == NO)
                      {
                       frame = Tick_message_data_G[0][1];
                       Prev_bit = Last_Bit;
                       z = 1;
              if (Check_Bit_Stuff(8,5) == NO)
                        {
                        return NO;
                        }
                       else {return YES;}
                       }
                    else {return YES;}
                   }
                 else {return YES;}
                 }
              else {return YES;}
              }
          else {return YES;}
          }
       else {return YES;}
}
```

```
tByte Check_Bit_Stuff(tByte frame_size, tByte CONSEC_BITS)
{
      tByte FRAME1;
      tByte k;

      Stuff_bits=0;

      for (k=z; k <= frame_size; k++)
              {

              FRAME1 = frame >> (k-1);

              Curr_bit = FRAME1 & 0x1;

              if (Curr_bit == Prev_bit)
                      {
                      Consec_Bits ++;
                      }
              else
                      {
                      Consec_Bits = 0;
                      Prev_bit = Curr_bit;
                      }

              if (Consec_Bits == (CONSEC_BITS-1))
```

```
                    {
                    Stuff_bits ++;
                    Consec_Bits = 0;
                    }
             }

      Last_Bit = Prev_bit;

      if(Stuff_bits > 0) {return YES;}

      else {return NO;}

}
```

Slave code

```
if (Tick_message_data_G[0] >> 5 == 1)
      {
      for(i=1; i<=7; i++)
             {
             Tick_message_data_G[i] ^= 0x55;
             }
      }
```

Byte-based XOR masking

Online (function call)

Master code

```
Data_Config();
```

```
void Data_Config(void)
{
      tByte i ;

      Tick_message_data_G[0][0] = 0;
      Tick_message_data_G[0][7] = 0;

      for(i=1; i <= NO_OF_BYTES; i++)
      {
      frame = Tick_message_data_G[0][i];
      if (Check_Bit_Stuff(8,5) == YES)
             {
             Tick_message_data_G[0][i] ^= 0x55;
             if(i==4)
                    {
                    Tick_message_data_G[0][0] |= (0x01<<5); // 0010 0000
                    }
             else
                    {
                    Tick_message_data_G[0][7] |= (0x01<<(7-i));
                    }
             }
      }

// bit no 1 in byte7 must oppose last bit in byte6
// bit no 5 in byte7 must oppose bit no 4 in this byte
// last bit in byte7 must oppose bit no 7 in this byte

 Tick_message_data_G[0][7] |= (!(Tick_message_data_G[0][6]&0x01)<<7)
                           |(!((Tick_message_data_G[0][7]>>4)&0x01)<<3)
                           |(!((Tick_message_data_G[0][7]>>1)&0x01));
```

```
}
```

```
/*********************************************************************
Check_Bit_Stuff()

This function goes through the frame bit-by-bit. Whenever it finds
consecutive 1s or 0s, it returns YES, otherwise returns NO.

Note that this function requires two values: the frame size and the number of
consecutive bits to check for. (e.g. 4 or 5)

*********************************************************************/

tByte Check_Bit_Stuff(tByte frame_size, tByte CONSEC_BITS)
{

      tByte frame1;
      bit Curr;
      bit Prev;
      tWord Consec_Bits;
      tWord k;


      Consec_Bits = 0;  // Number of consecutive bits
      k=1;
      Stuff_bits=0;

      Prev = frame & 0x1;

      for (k=2; k <= frame_size; k++)
              {

              frame1 = frame >> (k-1);

              Curr = frame1 & 0x1;

              if (Curr == Prev)
                      {
                      Consec_Bits ++;
                      }
              else
                      {
                      Consec_Bits = 0;
                      Prev = Curr;
                      }

              if (Consec_Bits == (CONSEC_BITS-1))

                      {
                      Stuff_bits++;

                      Consec_Bits = 0;
                      }
              }


              if(Stuff_bits > 0)
                      {
                      return YES;
                      }

      return NO;

}
```

Slave code

```
Data_Config();
```

```
void Data_Config(void)
{

      tByte i;
      for(i=1; i<= NO_OF_BYTES; i++)
              {
              if(i == 4)
                      {
                      if (((Tick_message_data_G[0]>>5) & 0x01) == 1)
                      {
                              Ti ck_message_data_G[4] ^= 0x55;
                              }
                      }

              else
                      {
                      if ((Tick_message_data_G[7]>>(7-i)) & 0x01 == 1)
                              {
                              Tick_message_data_G[i] ^= 0x55;
                              }
                      }
              }
}
```

Software Bit Stuffing (SBS)

Master code

```
Encode_Data();
```

```
void Encode_Data(void)
{
tByte i;

// Consecutive bits counter
tByte Consec_Bits=0;

// Bit status flags
bit Prev_Bit=0;
bit Current_Bit=0;

// Pointers for input data
tByte In_Bit_No=0;
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Bit_No=0;
tByte Out_Byte_No=0;

// Initialise the consec bits counter
Consec_Bits=0;

// Initialise the previous bit (i.e. from Slave ID) - can be modified?
Prev_Bit=0;

// Reset the output data buffers
for(i=0; i<7; i++)
{
Tick_data_after_stuffing[i] = 0;

}
// Begin the bit stuffing
for(In_Byte_No=0;In_Byte_No<CAN_BYTES_USED_FOR_DATA;In_Byte_No++)
        {
        // Loop through the input bits in this byte
```

```
        for(In_Bit_No=0;In_Bit_No<8;In_Bit_No++)
                {
                // Loop through each bit
                Current_Bit=(Tick_message_data_G[0][In_Byte_No]>>(7-In_Bit_No))&0x01;

                // See if current bit is same as previous bit
                if(Current_Bit==Prev_Bit)
                        {
                        // It is
                        Consec_Bits++;
                        }
                // Otherwise we are OK
                else
                        {
                        // Reset counter
                        Consec_Bits=0;
                        Prev_Bit=Current_Bit;
                        }

                // Now we copy the data as normal here
                Tick_data_after_stuffing[Out_Byte_No]|=((tByte)(Current_Bit)<<(7-
Out_Bit_No));

                // Increment and check for roll-over
                if(++Out_Bit_No>7)
                        {
                        // Reset bit counter
                        Out_Bit_No=0;
                        // Increment byte counter
                        Out_Byte_No++;
                        }

                // Are we at bit limit?
                if(Consec_Bits==CONSECUTIVE_BIT_LIMIT)
                        {
                        // Yes-Insert stuff bit

Tick_data_after_stuffing[Out_Byte_No]|=((tByte)(!Current_Bit)<<(7-Out_Bit_No));

                        // Increment and check for roll-over
                        if(++Out_Bit_No>7)
                                {
                                // Reset bit counter
                                Out_Bit_No=0;
                                // Increment byte counter
                                Out_Byte_No++;
                                }

                        // Reset bit counter
                        Prev_Bit=!Current_Bit;
                        Consec_Bits=0;
                        }
                }
        }
// We have now finished doing the bit stuffing
// Begin the compensation stage now
for(Out_Byte_No=Out_Byte_No;Out_Byte_No<CAN_NUM_DATA_BYTES-1;Out_Byte_No++)
        {
        // Add compensation bits to all remaining bytes
        for(Out_Bit_No=Out_Bit_No;Out_Bit_No<8;Out_Bit_No++)
                {
                // Toggle the last bit we sent
                Current_Bit=!Current_Bit;
                Tick_data_after_stuffing[Out_Byte_No]|=((tByte)(Current_Bit)<<(7-
Out_Bit_No));
                }
        // Remember to set Out_Bit_No equal to zero here!
        Out_Bit_No=0;
        }

}
```

Slave code

```
Decode_Data();
```

```
void Decode_Data(void)
{
// Consecutive bits counter
tByte Consec_Bits=0;

// Bit status flags
bit Prev_Bit=0;
bit Current_Bit=0;

// Pointers for input data
tByte In_Bit_No=0;
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Bit_No=0;
tByte Out_Byte_No=0;

// Initialise the consec bits counter
Consec_Bits=0;

// Initialise the previous bit (i.e. from Slave ID) - can be modified?
Prev_Bit=0;

// Init pointers
Out_Bit_No=0;
Out_Byte_No=0;
In_Bit_No=0;
In_Byte_No=0;

// Scan through until we have decoded all usefull information
while(Out_Byte_No<CAN_BYTES_USED_FOR_DATA)
                {
                // Loop through each bit
                Current_Bit=(Tick_data_before_destuffing[In_Byte_No]>>(7-
In_Bit_No))&0x01;

                // See if current bit is same as previous bit
                if(Current_Bit==Prev_Bit)
                        {
                        // It is
                        Consec_Bits++;
                        }

                // Otherwise we are OK
                else
                        {
                        // Reset counter
                        Consec_Bits=0;
                        Prev_Bit=Current_Bit;
                        }

                // Are we at bit limit?
                if(Consec_Bits==CONSECUTIVE_BIT_LIMIT)
                        {
                        // Yes-remove stuff bit
                        // Increment and check for roll-over
                        if(++In_Bit_No>7)
                                {
                                // Reset bit counter
                                In_Bit_No=0;
                                // Increment byte counter
                                In_Byte_No++;
                                }
                        // Reset counter
                        Prev_Bit=!Current_Bit;
                        Consec_Bits=0;
                        }
```

```
                // Now we copy the data as normal here
                 Tick_message_data_G[Out_Byte_No]|=((tByte)(Current_Bit)<<(7-
                            Out_Bit_No));

                // Increment and check for roll-over
                if(++In_Bit_No>7)
                        {
                        // Reset bit counter
                        In_Bit_No=0;
                        // Increment byte counter
                        In_Byte_No++;
                        }

                // Increment and check for roll-over
                if(++Out_Bit_No>7)
                        {
                        // Reset bit counter
                        Out_Bit_No=0;
                        // Increment byte counter
                        Out_Byte_No++;
                        }
                }
}
```

Eight-to-Eleven Modulation (EEM)

Explicit lookup table

Master code

```
// Important definitions

#define CAN_BYTES_USED_FOR_DATA (5)
#define CAN_NUM_DATA_BYTES (8)

#define NO_INPUT_BYTES (5)
#define NO_INPUT_BITS (11)
#define NO_OUTPUT_BITS (8)
#define DIFF_IO_BITS (NO_INPUT_BITS - NO_OUTPUT_BITS)
```

```
// Original data message
tByte Tick_message_data_G[NUMBER_OF_SLAVES][CAN_BYTES_USED_FOR_DATA];

// Encoded data message
tByte Tick_data_after_EEM_encoding[CAN_NUM_DATA_BYTES-1];

/*-------------------------------------------------------------------*-

  SCC_A_MASTER_Send_Tick_Message()

  This function sends a tick message, over the CAN network.
  The receipt of this message will cause an interrupt to be generated
  in the slave(s): this invoke the scheduler 'update' function
  in the slave(s).

-*-------------------------------------------------------------------*/
void SCC_A_MASTER_Send_Tick_Message(const tByte SLAVE_INDEX)
   {

   // Find the slave ID for this slave
   // ALL SLAVES MUST HAVE A UNIQUE (non-zero) ID
   tByte Slave_ID = (tByte) Current_Slave_IDs_G[SLAVE_INDEX];
   CAN_OBJ[0].Data[0] = Slave_ID;


   // Fill the data fields
   CAN_OBJ[0].Data[1] = Tick_data_after_EEM_encoding[0];
```

```
   CAN_OBJ[0].Data[2] = Tick_data_after_EEM_encoding[1];
   CAN_OBJ[0].Data[3] = Tick_data_after_EEM_encoding[2];
   CAN_OBJ[0].Data[4] = Tick_data_after_EEM_encoding[3];
   CAN_OBJ[0].Data[5] = Tick_data_after_EEM_encoding[4];
   CAN_OBJ[0].Data[6] = Tick_data_after_EEM_encoding[5];
   CAN_OBJ[0].Data[7] = Tick_data_after_EEM_encoding[6];

   // Send the message on the CAN bus
   CAN_OBJ[0].MCR = 0xEFFF;  // Set TXRQ
   }



void Encode_data(void)
{
tByte i;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;

// Pointers for output data

tByte Out_Byte_No=0;

// Convert each tick message byte to its equivalent 11-bit EEM code using the EEM_table
(see EEM_table.H)
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{
EEM_data[i] = EEM_Code[Tick_message_data_G[0][i]];
}


// Reset the output data buffers
for(i=0; i<CAN_NUM_DATA_BYTES-1; i++)
{
Tick_data_after_EEM_encoding[i] = 0;
}


RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the encoded CAN data bytes for transmission

      for (In_Byte_No=0; In_Byte_No<NO_INPUT_BYTES; In_Byte_No++)
      {

      //
      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >>
      RS_Index) & (0x7FF >> RS_Index));

      Out_Byte_No++;


      if (RS_Index > NO_OUTPUT_BITS)
              {
              RS_Index -= NO_OUTPUT_BITS;
              Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >>
              RS_Index) & (0x7FF >> RS_Index));

              Out_Byte_No++;
              }

      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] <<
   (NO_OUTPUT_BITS-RS_Index)) & (0x7FF << (NO_OUTPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;

      }

}
```

Slave code (Binary Search Algorithm)

```
// Important definitions

#define CAN_BYTES_USED_FOR_DATA (5)
#define CAN_NUM_DATA_BYTES (8)

#define NO_INPUT_BYTES (5)
#define NO_INPUT_BITS (11)
#define NO_OUTPUT_BITS (8)
#define DIFF_IO_BITS (NO_INPUT_BITS - NO_OUTPUT_BITS)
```

```
// Data sent from the master to this slave
tByte Tick_data_before_EEM_decoding[CAN_NUM_DATA_BYTES-1];
tByte Tick_message_data_G[NUMBER_OF_SLAVES][CAN_BYTES_USED_FOR_DATA];

/*--------------------------------------------------------------------*-

  SCC_A_SLAVE_Process_Tick_Message()

  The ticks messages are crucial to the operation of this shared-clock
  scheduler: the arrival of a tick message (at regular intervals)
  invokes the 'Update' ISR, that drives the scheduler.

  The tick messages themselves may contain data.  These data are
  extracted in this function.

-*--------------------------------------------------------------------*/
tByte SCC_A_SLAVE_Process_Tick_Message(void)
   {
   tByte Tick_ID;

      if ((CAN_OBJ[0].MCR & 0x0c00) == 0x0800)  // if MSGLST set
         {
         // Indicates that the CAN controller has stored a new
         // message into this object, while NEWDAT was still set,
         // i.e. the previously stored message is lost.

         // We simply IGNORE this here and reset the flag
         CAN_OBJ[0].MCR = 0xf7ff;  // reset MSGLST
         }

   // The first byte is the ID of the slave for which the data are
   // intended
   Tick_ID = CAN_OBJ[0].Data[0];   // Get data byte 0 (Slave ID)

   if (Tick_ID == SLAVE_ID)
      {
      // Only if there is a match do we need to copy these fields
      Tick_data_before_EEM_decoding[0] = CAN_OBJ[0].Data[1];
      Tick_data_before_EEM_decoding[1] = CAN_OBJ[0].Data[2];
      Tick_data_before_EEM_decoding[2] = CAN_OBJ[0].Data[3];
      Tick_data_before_EEM_decoding[3] = CAN_OBJ[0].Data[4];
      Tick_data_before_EEM_decoding[4] = CAN_OBJ[0].Data[5];
      Tick_data_before_EEM_decoding[5] = CAN_OBJ[0].Data[6];
      Tick_data_before_EEM_decoding[6] = CAN_OBJ[0].Data[7];
      }

   else
      {
      // Must do same processing to avoid jitter
      Dummy_tick_data_G[1] = CAN_OBJ[0].Data[1];
      Dummy_tick_data_G[2] = CAN_OBJ[0].Data[2];
      Dummy_tick_data_G[3] = CAN_OBJ[0].Data[3];
      Dummy_tick_data_G[4] = CAN_OBJ[0].Data[4];
      Dummy_tick_data_G[5] = CAN_OBJ[0].Data[5];
      Dummy_tick_data_G[6] = CAN_OBJ[0].Data[6];
      Dummy_tick_data_G[7] = CAN_OBJ[0].Data[7];
        }
       CAN_OBJ[0].MCR = 0xFDFD;  // Reset NEWDAT and INTPND
```

```
   return Tick_ID;
   }
```

```
void Decode_data(void)
{
tWord i;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Byte_No=0;


// Reset the EEM_data
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{
EEM_data[i] = 0;
}

RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the decoded CAN data bytes

      for (Out_Byte_No=0; Out_Byte_No<CAN_BYTES_USED_FOR_DATA; Out_Byte_No++)
      {

      //
      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] <<
      RS_Index) & 0x7FF);

      In_Byte_No ++;

      if (RS_Index > NO_INPUT_BITS)
              {
              RS_Index -= NO_INPUT_BITS;
              EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] <<
              RS_Index)& 0x7FF);// & (0xFF << RS_Index));

              In_Byte_No++;
              }

      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] >>
   (NO_INPUT_BITS-RS_Index))& 0x7FF);// & (0xFF >> (NO_INPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;

      }


      // Recover the original byte using binary search algorithm

      for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
      {
      Tick_message_data_G[0][i] = binarySearch(EEM_Code, EEM_data[i],
      ARR_MIN_INDEX, ARR_MAX_INDEX);
      }

}
```

```
/*---------------------------------------------------------------------------
      binarySearch(a, value)

The search begins by examining the value in the center of the list; because the values
are sorted, it then knows whether the value occurs before or after the centre value,
and searches through the correct half in the same way.
```

```
This function determines the 'index' of a given value in a sorted list 'a' between
indices 'left' and 'right'
------------------------------------------------------------------------------*/

tByte binarySearch(tWord* a, tWord value, tWord left, tWord right)
{
      tByte mid=0;

      if (right < left) {return 0;}


      mid = (left + right)/2;

    if (value > a[mid]) {return binarySearch(a, value, mid+1, right);}

    else if (value < a[mid]) {return binarySearch(a, value, left, mid-1);}
    else {return mid;}
}
```

Slave code (reverse array)

```
void Decode_data(void)
{
tWord i;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Byte_No=0;


// Reset the EEM_data
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{
EEM_data[i] = 0;
}

RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the decoded CAN data bytes

      for (Out_Byte_No=0; Out_Byte_No<CAN_BYTES_USED_FOR_DATA; Out_Byte_No++)
      {

      //
      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] << RS_Index)
& 0x7FF);

      In_Byte_No ++;

      if (RS_Index > NO_INPUT_BITS)
              {
              RS_Index -= NO_INPUT_BITS;
              EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] <<
RS_Index)& 0x7FF);// & (0xFF << RS_Index));

              In_Byte_No++;
              }

      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] >>
(NO_INPUT_BITS-RS_Index))& 0x7FF);// & (0xFF >> (NO_INPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;


      // Recover the original CAN data byte using the EEM reverse table
      for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
              {
              Tick_message_data_G[0][i] = Byte_Value[EEM_data[i]-546];
              }
```

```
      }

}
```

Implicit lookup table

Master code

```
void Encode_data(void)
{
tByte i, Input_Byte;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Byte_No=0;

// Convert each tick message byte to its equivalent 11-bit EEM code using the EEM_table
(see EEM_table.H)
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{

Input_Byte = Tick_message_data_G[0][i];

// Use EEM_Code function to find the EEM code for the input bytes
EEM_data[i] = EEM_Code(i, Input_Byte);

}


// Reset the output data buffers
for(i=0; i<CAN_NUM_DATA_BYTES-1; i++)
{
Tick_data_after_EEM_encoding[i] = 0;
}


// Using the EEM data, produce the encoded CAN data bytes for transmission

RS_Index = DIFF_IO_BITS;


      for (In_Byte_No=0; In_Byte_No<NO_INPUT_BYTES; In_Byte_No++)
      {

      //
      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >> RS_Index)
& (0x7FF >> RS_Index));

      Out_Byte_No++;


      if (RS_Index > NO_OUTPUT_BITS)
              {
              RS_Index -= NO_OUTPUT_BITS;
              Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >>
RS_Index) & (0x7FF >> RS_Index));

              Out_Byte_No++;
              }

      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] <<
(NO_OUTPUT_BITS-RS_Index)) & (0x7FF << (NO_OUTPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;

      }

}
```

```
/*-------------------------------------------------------------------------
      EEM_Code()

      This function finds the equivalent EEM code for an input BYTE
-------------------------------------------------------------------------*/

tWord EEM_Code (tByte i, tByte BYTE)
{

tByte Array_Index = (BYTE*11)/16;   // determine the no of the array element in
                                    // which the EEM code is stored
tByte EEM_Start = (BYTE*11)%16;// determine the location of the first
                                      // bit of the EEM code in the array element


// if the EEM code start after 6 bits from left, then the code is split within two
array elements
if(EEM_Start > 5)
      {
      EEM_data[i] = EEM_ARRAY[Array_Index] << (11 - (16 - EEM_Start));
      EEM_data[i] |= EEM_ARRAY[Array_Index+1] >> (16 - (11 - (16 - EEM_Start)));
      }

// if not, then the code is stored in one array element
else
      {
      EEM_data[i] = EEM_ARRAY[Array_Index] >>  ((16 - EEM_Start) - 11);
      }

      // filter the data by taking only the 11 LSB
    EEM_data[i] &= 0x7FF;

return EEM_data[i];
}
```

Slave code (search element)

```
void Decode_data(void)
{
tWord i, Array_Start;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Byte_No=0;

// Variables for searching element
tByte EEM_Start = 0;
tByte Array_Index = 0;
tWord eem_data = 0;


// This is to test the slave decoding
/*Tick_data_before_EEM_decoding[0]=0x67;
Tick_data_before_EEM_decoding[1]=0x4c;
Tick_data_before_EEM_decoding[2]=0xca;
Tick_data_before_EEM_decoding[3]=0x6e;
Tick_data_before_EEM_decoding[4]=0x25;
Tick_data_before_EEM_decoding[5]=0x27;
Tick_data_before_EEM_decoding[6]=0x56;
*/


// Reset the EEM_data
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{
```

```
EEM_data[i] = 0;
}

RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the decoded CAN data bytes

      for (Out_Byte_No=0; Out_Byte_No<CAN_BYTES_USED_FOR_DATA; Out_Byte_No++)
      {

      //
      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] << RS_Index)
& 0x7FF);

      In_Byte_No ++;

      if (RS_Index > NO_INPUT_BITS)
             {
             RS_Index -= NO_INPUT_BITS;
             EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] <<
RS_Index)& 0x7FF);// & (0xFF << RS_Index));

             In_Byte_No++;
             }

      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] >>
(NO_INPUT_BITS-RS_Index))& 0x7FF);// & (0xFF >> (NO_INPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;



      // Determine the range to save time

      if             (EEM_data[Out_Byte_No] < 578)         {Array_Start = 0       ;}
      else if (EEM_data[Out_Byte_No] < 647) {Array_Start = 11      ;}
      else if (EEM_data[Out_Byte_No] < 706) {Array_Start = 22      ;}
      else if (EEM_data[Out_Byte_No] < 802) {Array_Start = 33      ;}
      else if (EEM_data[Out_Byte_No] < 834) {Array_Start = 44      ;}
      else if (EEM_data[Out_Byte_No] < 930) {Array_Start = 55      ;}
      else if (EEM_data[Out_Byte_No] < 962) {Array_Start = 66      ;}
      else if (EEM_data[Out_Byte_No] < 1058)       {Array_Start = 77      ;}
      else if (EEM_data[Out_Byte_No] < 1090)       {Array_Start = 88      ;}
      else if (EEM_data[Out_Byte_No] < 1186)       {Array_Start = 99      ;}
      else if (EEM_data[Out_Byte_No] < 1218)       {Array_Start = 110     ;}
      else if (EEM_data[Out_Byte_No] < 1314)       {Array_Start = 121     ;}
      else if (EEM_data[Out_Byte_No] < 1346)       {Array_Start = 132     ;}
      else if (EEM_data[Out_Byte_No] < 1442)       {Array_Start = 143     ;}
      else if (EEM_data[Out_Byte_No] < 1474)       {Array_Start = 154     ;}
    else     {Array_Start = 165;}


      // Recover the original CAN data byte using the EEM table

      EEM_Start=0;
      Array_Index = Array_Start;


      for (i=0; i<16; i++)
             {

        EEM_Start = (i*11) % 16;


        // if the EEM code start after 6 bits from left, then the code is split within
two array elements
        if(EEM_Start > 5)
             {
             eem_data = EEM_ARRAY[Array_Index] << (11 - (16 - EEM_Start));
             eem_data |= EEM_ARRAY[++Array_Index] >> (16 - (11 - (16 - EEM_Start)));
             }

        // if not, then the code is stored in one array element
        else
             {
             eem_data = EEM_ARRAY[Array_Index] >>  ((16 - EEM_Start) - 11);
```

```
              }

        // filter the data by taking only the 11 LSB
      eem_data &= 0x7FF;



              // Now check if the current EEV value (in the table) matches the received
EEM word

              if (EEM_data[Out_Byte_No] == eem_data)
                     {
                     Tick_message_data_G[0][Out_Byte_No] = i + (Array_Start/11) * 16;
      // (Array_Start/11) is the array division in which the EEM code is located.

                                                                          // (Array_Start/11) *
16  determines the index of the first element in the division.
                     break;
                     }

              }

      }

}
```

Algorithmic coding

Master code

```
void Encode_data(void)
{
tByte i, Current_Bit, Next_Bit;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;
tByte In_Bit_No=0;


// Pointers for output data
tByte Out_Byte_No=0;
tByte Out_Bit_No=0;


// Convert each tick message byte to its equivalent 11-bit EEM code
// Note that the number of input bits = 8, and the output bits = 11

for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{
      Out_Bit_No=0;

      for(In_Bit_No=0; In_Bit_No<8; In_Bit_No++)
      {
              Current_Bit = (Tick_message_data_G[0][i] >> (7-In_Bit_No)) & 0x01;
              EEM_data[i] |= Current_Bit <<(10-Out_Bit_No);


              if((In_Bit_No==0) ||(In_Bit_No==3)||(In_Bit_No==6))
              {

              // Stuff the opposite bit afterward
              Next_Bit = !Current_Bit;
              Out_Bit_No++;

              EEM_data[i] |= Next_Bit <<(10-Out_Bit_No);

              }


              Out_Bit_No++;
      }
```

```
}

// Reset the output data buffers
for(i=0; i<CAN_NUM_DATA_BYTES-1; i++)
{
Tick_data_after_EEM_encoding[i] = 0;
}


RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the encoded CAN data bytes for transmission
// Note that the number of input bits = 11, and the output bits = 8

      for (In_Byte_No=0; In_Byte_No<NO_INPUT_BYTES; In_Byte_No++)
      {

      //
      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >> RS_Index)
& (0x7FF >> RS_Index));

      Out_Byte_No++;


      if (RS_Index > NO_OUTPUT_BITS)
              {
              RS_Index -= NO_OUTPUT_BITS;
              Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >>
RS_Index) & (0x7FF >> RS_Index));

              Out_Byte_No++;
              }

      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] <<
(NO_OUTPUT_BITS-RS_Index)) & (0x7FF << (NO_OUTPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;

      }

}
```

Slave code

```
void Decode_data(void)
{
tWord i, Current_Bit;//Array_Start;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;
tByte In_Bit_No=0;


// Pointers for output data
tByte Out_Byte_No=0;
tByte Out_Bit_No=0;


// Reset the EEM_data
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{
EEM_data[i] = 0;
}

RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the decoded CAN data bytes

      for (Out_Byte_No=0; Out_Byte_No<CAN_BYTES_USED_FOR_DATA; Out_Byte_No++)
      {
```

```
      //
      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] << RS_Index)
& 0x7FF);

      In_Byte_No ++;

      if (RS_Index > NO_INPUT_BITS)
              {
              RS_Index -= NO_INPUT_BITS;
              EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] <<
RS_Index)& 0x7FF);// & (0xFF << RS_Index));

              In_Byte_No++;
              }

      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] >>
(NO_INPUT_BITS-RS_Index))& 0x7FF);// & (0xFF >> (NO_INPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;

    }

      // Recover the original CAN data byte by removing the stuff-bits
  for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
  {
      Out_Bit_No=0;

      for(In_Bit_No=0; In_Bit_No<11; In_Bit_No++)
      {
              Current_Bit = (EEM_data[i] >> (10-In_Bit_No)) & 0x01;

              if (!((In_Bit_No==1) ||(In_Bit_No==5)||(In_Bit_No==9)))
              {

              Tick_message_data_G[0][i] |= Current_Bit <<(7-Out_Bit_No);
              Out_Bit_No++;

              }
      }
  }

}
```

Mathematical coding

Master code

```
void Encode_data(void)
{
tByte i;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Byte_No=0;


// Convert each tick message byte to its equivalent 11-bit EEM code using the EEM
mathematical equation
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{

// The used equation is derived from the lookup table values
// f(x) = f(0) + x + 4*floor(x/4) + 64*floor(x/32)

EEM_data[i] = 546 + Tick_message_data_G[0][i] + ((Tick_message_data_G[0][i]/4)*4) +
((Tick_message_data_G[0][i]/32)*64);

}
```

```
// Reset the output data buffers
for(i=0; i<CAN_NUM_DATA_BYTES-1; i++)
{
Tick_data_after_EEM_encoding[i] = 0;
}


RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the encoded CAN data bytes for transmission

      for (In_Byte_No=0; In_Byte_No<NO_INPUT_BYTES; In_Byte_No++)
      {

      //
      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >> RS_Index)
& (0x7FF >> RS_Index));

      Out_Byte_No++;


      if (RS_Index > NO_OUTPUT_BITS)
              {
              RS_Index -= NO_OUTPUT_BITS;
              Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] >>
RS_Index) & (0x7FF >> RS_Index));

              Out_Byte_No++;
              }

      Tick_data_after_EEM_encoding[Out_Byte_No] |= ((EEM_data[In_Byte_No] <<
(NO_OUTPUT_BITS-RS_Index)) & (0x7FF << (NO_OUTPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;

      }


}
```

Slave code

```
void Decode_data(void)
{

tByte i;
tByte RS_Index;      // right shift index

// Pointers for input data
tByte In_Byte_No=0;

// Pointers for output data
tByte Out_Byte_No=0;


// Reset the EEM_data
for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
{
EEM_data[i] = 0;
}

RS_Index = DIFF_IO_BITS;


// Using the EEM data, produce the decoded CAN data bytes

      for (Out_Byte_No=0; Out_Byte_No<CAN_BYTES_USED_FOR_DATA; Out_Byte_No++)
      {

      //
```

```
      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] << RS_Index)
& 0x7FF);

      In_Byte_No ++;

      if (RS_Index > NO_INPUT_BITS)
              {
              RS_Index -= NO_INPUT_BITS;
              EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] <<
RS_Index)& 0x7FF);// & (0xFF << RS_Index));

              In_Byte_No++;
              }

      EEM_data[Out_Byte_No] |= ((Tick_data_before_EEM_decoding[In_Byte_No] >>
(NO_INPUT_BITS-RS_Index))& 0x7FF);// & (0xFF >> (NO_INPUT_BITS-RS_Index)));

    RS_Index += DIFF_IO_BITS;


      // Recover the original CAN data byte using the EEM table
      for (i=0; i<CAN_BYTES_USED_FOR_DATA ; i++)
              {

              // The encoder equation is used here in reverse way to calculate the
value of each byte
              // x = f(x) - f(0) - 2*floor[(f(x)-f(0))/4] - 16*floor[(f(x)-f(0))/64]
              Tick_message_data_G[0][i] = EEM_data[i] - 546 - ((EEM_data[i]-546)/4)*2 -
((EEM_data[i]-546)/64)*16;
              }

      }


}
```

Appendix I

Bibliography

Ackerman, A.F., Buchwald, L.S. and Lewski, F.H. (1989) “Software inspections: an effective verification process”, IEEE Software, Vol. 6 (3), pp. 31-36.

Ada (1980) “Reference Manual for the Ada Programming Language”, proposed standard document, U.S. Department of Defense.

Albert, A. (2004) “Comparison of event-triggered and time-triggered concepts with regard to distributed control systems,” in Proceedings of Embedded World, Nurnberg, Germany, 17-19 Feb, 2004, pp. 235-252.

Ali, W. (2004) “Embedded Systems”, Programmers Heaven.com (Last accessed: October 2008) http://www.programmersheaven.com/search/Download.asp?FileID=35032

Allworth, S.T. (1981) “An Introduction to Real-Time Software Design”, Macmillan, London.

ANSVIP (1970) “American National Standard Vocabulary for Information Processing”, American National Standards Institute, Inc., 1430 Broadway, New York, N.Y.

Apneseth, C (2006) “Embedded system technology in ABB”, ABB Corporate Research, ABB AS, Billingstad, Norway.

ARM (2001) "ARM7TDMI technical reference manual".

Arons, T., Elster, E., Murphy, T. and Singerman, E. (2006) “Embedded Software Validation: Applying Formal Techniques for Coverage and Test Generation”, Seventh International Workshop on Microprocessor Test and Verification, MTV '06, 4-5 Dec. 2006, pp. 45-51.

Ashling Microsystems (2007) “LPC2000 Evaluation and Development Kits datasheet”, available online (Last accessed: October 2008) http://www.ashling.com/pdf_datasheets/DS266-EvKit2000.pdf

Audsley, N., Burns, A., Richardson, M.F. and Wellings, A.J. (1991) “Hard real-time scheduling: The deadline- monotonic approach”, In Proceedings of the 8th Workshop on Real-Time Operating Systems and Software.

Avrunin, G.S., Corbett, J.C. and Dillon, L.K. (1998) “Analyzing partially-implemented real-time systems”, IEEE Transactions on Software Engineering, Vol. 24 (8), pp.602-614.

Axelson, J. (1994) “The Microcontroller Idea Book: Circuits, Programs & Applications Featuring the 8052-Basic Single-Chip Computer”, Lakeview Research.

Ayavoo, D., Pont, M.J. and Parker, S. (2004) "Using simulation to support the design of distributed embedded control systems: A case study", In: Koelmans, A., Bystrov, A. and Pont,

M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp. 54-65. Published by University of Newcastle upon Tyne.

Ayavoo, D., Pont, M.J., Fang, J., Short, M. and Parker, S. (2005) "A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car", In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 60-90. Published by University of Newcastle upon Tyne.

Ayavoo, D., Pont, M.J. and Parker, S. (2006) "Does a 'simulation first' approach reduce the effort involved in the development of distributed embedded control systems?", 6th UKACC International Control Conference, Glasgow, Scotland, 2006.

Ayavoo, D. (2006) "The Development of Reliable X-by-Wire Systems: Assessing The Effectiveness of a 'Simulation First' Approach", PhD thesis, Department of Engineering, University of Leicester, UK.

Ayavoo, D., Pont, M.J., Short, M. and Parker, S. (2007) "Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems", Microprocessors and Microsystems, Vol. 31(5), pp. 326-334.

Bai, X., Tsai, W.T., Paul, R., Feng, K. and Yu, L. (2002) "Scenario-based modeling and its applications", Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, 2002 (WORDS 2002), 7-9 Jan. 2002, pp. 253-260.

Bailey, C.M., Fyfe, E., Vardanega, T. and Wellings, A.J. (1993) "The use of preemptive priority-based scheduling for space applications", Proceedings of Real-Time Systems Symposium, 1-3 December 1993, pp. 253-257.

Baker, T.P. and Shaw, A. (1989) "The cyclic executive model and Ada. Real-Time Systems", Vol. 1 (1), pp. 7-25.

Baker, T. and Pazy, O. (1991) "Real-time features for Ada 9X", Proceedings of the 12th Real-Time Systems Symposium, 4-6 December 1991, pp. 172-180.

Balarin, F., Hsieh, H., Jurecska, A., Lavagno, L. and Sangiovanni-Vincentelli, A. (1996) "Formal verication of embedded systems based on CFSM networks", In Proceedings of the 33th ACM/IEEE Design Automation Conference, pp. 568-571, June 1996.

Bannatyne, R. (1998) "Time triggered protocol-fault tolerant serial communications for real-time embedded systems", WESCON/98 Conference Proceedings, Anaheim, CA, USA, pp. 86-91.

Bannatyne, R. (2004) "MPC5500 Family of New Generation Embedded Controllers", Micro Control Journal, available online (Last accessed: October 2008) http://www.mcjournal.com/articles/arc109/MPC5500.pdf

Barnett, R.H., O'Cull, L. and Cox, S. (2003) "Embedded C Programming and the Atmel Avr", Thomson Delmar Learning.

Barr, M. (1999) "Programming Embedded Systems in C and C++", O'Reilly Media.

Barr, M. and Massa, A. (2006) “Programming Embedded Systems: With C and GNU Development Tools”, O'Reilly Media.

Barreiros, J., Costa, E., Fonseca, J. and Coutinho, F. (2000) “Jitter reduction in a real-time message transmission system using genetic algorithms”, Proceedings of the CEC 2000 – Conference of Evolutionary Computation, USA, July 2000.

Baruah, S., Buttazzo, G., Gorinsky, S. and Lipari, G. (1999) “Scheduling periodic task systems to minimize output jitter”, The Sixth International Conference on Real-Time Computing Systems and Applications, 1999, RTCSA '99, pp. 62-69.

Baruah S.K. (2006) “The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors”, Real-Time Systems, Vol. 32, pp. 9-20.

Bass, L, Clements, P and Kazman, R. (2003) “Software Architecture in Practice”, Addison-Wesley.

Bate, I.J. (1998), “Scheduling and Timing Analysis for Safety Critical Real-Time Systems”, PhD thesis, Department of Computer Science, University of York.

Bates, D.G. (1968) “PROSPRO/1800”, IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. 15, pp. 70-75.

Bates, I. (2000) “Introduction to scheduling and timing analysis”, in The Use of Ada in Real-Time System, IEE Conference Publication 00/034.

Beck, K (2001) “Extreme Programming Explained: Embrace Change”, Addison-Wesley.

Becker, L.B. and Gergeleit, M. (2001). “Execution Environment for Dynamically Scheduling Real-Time Tasks”. RTSS 2001, 22nd IEEE Real-Time Systems Symposium.

Becker, L.B., Nett, E., Schemmer, S. and Gergeleit, M. (2003). “Robust scheduling in team-robotics”. 11th International Workshop on Parallel and Distributed Real-Time Systems, Nice, France.

Bell, C. G. and Newell, A. (1971) “Computer Structures: Readings and Examples”, McGraw-Hill Book Company, New York. Chapter 5: The DEC PDP-8, pp. 120-136.

Bellis, M. “Inventors of the Modern Computer: Intel 4004 - The World's First Single Chip Microprocessor”, About Website. (Last accessed: October 2008) http://inventors.about.com/library/weekly/aa092998.htm

Bennett, S. (1994), “Real-time Computer Control: An introduction”, Prentice Hall.

Bernat, G. and Burns, A. (2001) “Implementing a Flexible Scheduler in Ada”, Proceedings of Reliable Software Technologies—Ada Europe 2001.

Bieman, J.M., and Murdock, V. (2001) “Finding code on the World Wide Web: a preliminary investigation”, Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, pp. 73-78.

Bloomfield, R., Cazin, J., Craigen, D., Juristo, N., Kesseler, E. and Voas, J. (2004) “Validation, verification and certification of embedded systems”, National Aerospace Laboratory (NLR), The Netherlands.

Boehm, B. W. (1981) “Software Engineering Economics”, Prentice-Hall.

Bolton, W. (2000) “Microprocessor Systems”, Longman.

Booch, G. (1991) “Object Oriented Design with Applications”, Benjamin / Cummings.

Booch, J. Rumbaugh, and I. Jacobson (1999) “The Unified Modeling Language User Guide”, Addison Wesley, 1999.

Borger, M., Klein, M. and Weiderman, N. (1988) “A testbed for investigating real-time Ada issues”, ACM SIGAda Ada Letters, Special edition: International Workshop on Real-Time Ada Issues, Vol. VIII (7), pp. 7-11.

Bosch (1991) “CAN Specification Version 2.0”, Robert Bosch GmbH.

Boulton, P.I.P. and Reid, P.A. (1969) “A Process-Control Language”, IEEE Transactions on Computers, Vol. 18 (11), pp. 1049-1053.

Bowen, J. (1993) “Formal methods in safety-critical standards”, Proceedings of Software Engineering Standards Symposium, 30 Aug.-3 Sept. 1993, pp. 168-177.

Bowen, J. and Hinchey, M. (1995) “Seven More Myths of Formal Methods”, IEEE Software, Vol. 12 (4), pp. 34-41.

Bradley, S., Henderson, W., Kendall, D., Robson, A. and Hawkes, S. (1996) “A formal design and implementation method for real-time embedded systems”, Proceedings of the 22nd EUROMICRO Conference EUROMICRO 96 'Beyond 2000: Hardware and Software Design Strategies', 2-5 Sept. 1996, pp. 77-84.

Brinkschulte, U., Kreuzinger, J., Pfeffer, M. and Ungerer, T. (2002) “A scheduling technique providing a strict isolation of real-time threads”, Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, 2002. (WORDS 2002). 7-9 Jan 2002, pp. 334-340.

Broadfoot, G.H. and Broadfoot, P.J. (2003) “Academia and industry meet: some experiences of formal methods in practice”, Tenth Asia-Pacific Software Engineering Conference, 2003, pp. 49-58.

Brosgol, B.J. (2003) “Ada and Java: real-time advantages”, Embedded.com, WWW website (Last accessed: October 2008) http://www.embedded.com/columns/technicalinsights/16100316?_requestid=169704

Brosgol, B. and Ruiz, J. (2007) “Ada enhances embedded-systems development”, Embedded.com, WWW website (Last accessed: October 2008) http://www.embedded.com/columns/technicalinsights/196800175?_requestid=167577

Broster, I. (2003) “Flexibility in dependable real-time communication”, PhD thesis, University of York, York, U.K.

Broster, I. and Burns, A. (2001) “Timely use of the CAN protocol in critical hard real-time systems with faults”, In Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001), Delft, The Netherlands, Jun. 13–15, 2001, pp. 95-102.

Brown, J.F. (1994) “Embedded Systems Programming in C and Assembly”, Kluwer Academic Publishers.

Budlong, M. (1999) “Teach Yourself COBOL in 21 days”, Sams.

Burns, A. and Wellings, A.J. (1987) “Real-time Ada issues”, ACM SIGAda Ada Letters, Vol. VII (6), pp. 43-46.

Burns, A. (1991) “Scheduling hard real-time systems: a review”, Software Engineering Journal, Vol. 6 (3), pp. 116-128.

Burns, A. (1999) “The Ravenscar Profile”, ACM Ada Letters.

Burns, A. (2001) “Non-Preemptive Dispatching and Locking Policies”, Ada Letters.

Burns, A. and Wellings, A.J. (2002) “Accessing delay queues”, In Proceedings of IRTAW11, Ada Letters, Vol. XX1I (4), pp. 72-76.

Burns, A. and Wellings, A.J. (2003) “Task attribute-based scheduling - extending Ada's support for scheduling”, In T. Vardenega, editor, Proceedings of the 12th International Real-Time Ada Workshop, Vol. XXIII, pp. 36-41. ACM Ada Letters.

Burns, A. Harbour, M.G. and Wellings, A.J. (2003a) “A round robin scheduling policy for Ada”, In Reliable Software Technologies, Proceedings of the Ada Europe Conference, Vol. LNCS 2655, pp. 334-343. Lecture Notes on Computer Science, Springer Verlag, 2003.

Burns, A., Wellings, A.J. and Vardanega, T. (2003b) “Report of session: Flexible scheduling in Ada”, In Proceedings of IRTAW 12, Ada Letters, Vol. XXIII (4), pp. 32-25.

Burns, A., Wellings, A.J. and Taft, T. (2004) “Supporting Deadlines and EDF Scheduling in Ada”, Proceedings of Reliable Software Technologies - Ada Europe 2004.

Burns, A., and Wellings, A.J. (2005) “Programming Execution-Time Servers in Ada 2005”, 27th IEEE International Real-Time Systems Symposium, 2006. RTSS '06, Dec. 2006, pp. 47-56.

Burns, A. (2006) “Real-Time Languages”, Network of Excellence on Embedded Systems Design, WWW website (Last accessed: October 2008) http://www.artist-embedded.org/artist/Real-Time-Languages.html

Burns, A. and Wellings, A.J. (2006) “Programming Execution-Time Servers in Ada 2005”, 27th IEEE International Real-Time Systems Symposium (RTSS '06), December 2006, pp. 47-56.

Burns, A. and Wellings, A.J. (2007a) “Concurrent and Real-time Programming in Ada 2005”, Cambriage University Press.

Burns, A. and Wellings, A.J. (2007b) “Programming Execution-Time Servers and Supporting EDF Scheduling in Ada 2005”, Handbook of Real-Time and Embedded Systems, Chapman and Hall/CRC.

Buttazzo, G. (2005), “Hard real-time computing systems: predictable scheduling algorithms and applications”, Second Edition, Springer.

Byteflight (2008) “Byteflight homepage”, WWW website (Last accessed: October 2008) http://www.byteflight.com/links/index.html

Calgary (2005) “Calgary Ecommerce Services – Glossary”, WWW website (Last accessed: October 2008) http://www.calgary-ecommerce-services.com/glossary.html

Carr, D. and Kizior, R.J. (2000) “The case for continued Cobol education”, IEEE Software, Vol. 17 (2), pp. 33-36.

Chakravarty, D. and Cannon, C. (1994) “PowerPC: Concepts, Architecture, and Design”, J. Ranade workstation series, McGraw-Hill.

Chandrasekharan, M., Dasarathy, B. and Kishimoto, Z. (1985) “Requirements-Based Testing of Real-Time Systems: Modeling for Testability”, IEEE Computer, Vol. 18, pp. 71-80.

Chapman, S.J (2004) “Fortran 90/95 for Scientists and Engineers”, McGraw-Hill Science Engineering.

Cheng, B. and Jeffery, R. (1996) “Comparing inspection strategies for software requirement specifications”, Proceedings of the 1996 Australian Software Engineering Conference, 1996, 14-18 July 1996, pp. 203-211.

Cho, Y., Yoo, S., Choi, K., Zergainoh, N.E. and Jerraya, A. (2005) “Scheduler implementation in MPSoC Design”, In: Asia South Pacific Design Automation Conference (ASPDAC’05), pp. 151-156.

Cho, Y., Zergainoh, N-E., Yoo, S., Jerraya, A.A. and Choi, K. (2007) “Scheduling with accurate communication delay model and scheduler implementation for multiprocessor system-on-chip”, Design Automation for Embedded Systems, Vol. 11 (2-3), pp. 167-191.

Chow, P. (1989) “The MIPS-X RISC Microprocessor”, Springer.

Chung, L., Nixon, B., Yu, E., and Mylopoulos, J. (2000) “Non-functional requirements in software engineering”, Kluwer Academic Publishers.

CiA (2008) “CAN in Automation”, WWW website (Last accessed: October 2008) http://www.can-cia.de/

Ciocarlie, H. and Simon, L. (2007) “Definition of a High Level Language for Real-Time Distributed Systems Programming”, EUROCON 2007 The International Conference on “Computer as a Tool”, Warsaw, September 9-12.

CISCO (2008) “Ethernet Technologies”, available online (Last accessed: October 2008) http://www.cisco.com/en/US/docs/internetworking/technology/handbook/Ethernet.html

Clarke, D. and Lee, I. (1997) “Automatic Generation of Tests for Timing Constraints from Requirements”, In Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems, Newport Beach, Califomia, Feb. 1997.

Clarke, E., Garlan, D., Krogh, B., Simmons, R. and Wing, J. (2000) “Verification Tools for Embedded Systems”, Carnegie Mellon University.

Cobb, R.H. and Mills, H.D. (1990) “Engineering software under statistical quality control”, IEEE Software, Vol. 7 (6), pp. 45-54.

Comhill, D. and Sha, L. (1987) “Priority Inversion in Ada”, ACM SIGAda Ada Letters Vol. VII (7), pp. 30-32.

Cook, D. (1999) “Evolution of Programming Languages and Why a Language is Not Enough to Solve Our Problems”, Software Technology Support Center, available online (Last accessed: October 2008) http://www.stsc.hill.af.mil/crosstalk/1999/12/cook.asp

Cooling, J.E. (1991) “Software design for real time systems”, Chapman and Hall.

Cornhilll, D., Sha, L. and Lehoczky, J.P. (1987) “Limitations of Ada for real-time scheduling”, Proceedings of the first international workshop on Real-time Ada issues, Morehampstead, Devon, United Kingdom, pp. 33-39.

Cortes, L.A., Eles, P. and Peng, Z. (2000) “Verification of embedded systems using a Petri net based representation”, Proceedings of the 13th International Symposium on System Synthesis, 2000. 20-22 Sept. 2000, pp. 149-155.

Cortes, L.A., Eles, P. and Peng, Z. (2001) “Hierarchical modeling and verification of embedded systems”, Proceedings of the Euromicro Symposium on Digital Systems, Design, 2001. 4-6 Sept. 2001, pp. 63-70.

Cottet, F. and David, L. (1999), “A solution to the time jitter removal in deadline based scheduling of real-time applications”, 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, pp. 33-38.

Cottet, F. (2002) “Scheduling in Real-time Systems”, Wiley.

Coutinho, F. M., Fonseca, J., Barreiros, J. and Costa, E. (2000) “Using Genetic Algorithms to Reduce Jitter in Control Variables Transmitted over CAN”, In Proceedings of the 7th International CAN Conference, 2000.

Crocker, D. and Carlton, J. (2007) “Verification of C Programs Using Automated Reasoning”, Fifth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2007, 10-14 Sept. 2007, pp. 7-14.

Cross II, J.H., Morrison, K.I., May, C.H. and Kathryn C. Waddel, K.C. (1989) “A Graphically Oriented Specification Language for Automatic Code Generation”, Department of Computer Science and Engineering, Auburn University, December, 1988.

Cunning, S.J. and Rozenblit, J.W. (1999) “Automatic test case generation from requirements specifications for real-time embedded systems”, Proceedings of the 1999 IEEE International Conference on Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference, 12-15 Oct. 1999, Vol. 5, pp.784–789.

Currit, P.A., Dyer, M., and Mills, H.D. (1986) "Certifying the reliability of software", IEEE Transactions on Software Engineering, Vol. 12, pp. 3-11.

Dai, H. and Scott, C.K. (1995) “AVAT, a CASE tool for software verification and validation”, Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, 10-14 July 1995, pp. 358-367.

Davidgould (2008) “Davidgould – Glossary”, WWW website (Last accessed: October 2008) http://www.davidgould.com/Glossary/Glossary.htm

Davis, R.I. (1993) “Approximate Slack Stealing Algorithms for Fixed Priority Preemptive Systems”, Technical Report YCS 217, Department of Computer Science, University of York, November 1993.

Davis, R., Punnekkat, S., Audsley, N. and Burns, A. (1995) “Flexible scheduling for adaptable real-time systems”, Proceedings of Real-Time Technology and Applications Symposium, 15-17 May 1995, pp. 230-239.

Dewar, R.B.K. (2006) “Safety-critical design for secure systems: The languages, tools and methods needed to build error-free-software”, WWW website (Last accessed: October 2008) http://www.embedded.com/columns/technicalinsights/190400498?_requestid=177701

DIN (1979) “Programming language PEARL”, Part 1. Basic PEARL, Part 2: Full PEARL, Deutsches Institut für Normung (DIN) German Standards Institute, Berlin, DIN 66253, 1979 (in English).

Do, H. and Rothermel, G. (2006) “On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques”, IEEE Transactions on Software Engineering, Vol. 32 (9), pp. 733-752.

Domaratsky, Y. and Perevozchikov, M. (2000). “Highly Dependable Time-Triggered Operating System”, Dedicated Systems Magazine, Vol. 4, pp. 77-80.

Donnelly, B. and Cosgrove, J. (2004) “Achieving microsecond accuracy with 32 bit microcontrollers using the controller area network (CAN)”, In Proceedings of Irish Signals and Systems Conference, Belfast, U.K., Jun./Jul. 2004, pp. 508-513.

Douglass, B., P. (2004) “Real Time UML: Advances in The UML for Real-Time Systems”, Addison-Wesley.

Dyer, M. and Mills, H.D. (1981) "The Cleanroom approach to reliable software development, " in Proc. Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting: Production of Reliable Flight-Crucial Software, Research Triangle Institute, NC, Nov. 2-4, 1981.

Dyer, M. (1982) "Cleanroom software development method", IBM Federal Systems Division, Bethesda, MD, Oct. 14, 1982.

Dyer, M. (1992) “Verification based inspection”, Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992, Vol. 2, pp. 418-427.

Ebenau, R.G. and Strauss, S.H. (1994) “Software Inspection, Process”, McGraw-Hill.

Edwards, T., Pont, M.J., Scotson, P. and Crumpler, S. (2004) "A test-bed for evaluating and comparing designs for embedded control systems", In: Koelmans, A., Bystrov, A. and Pont,

M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp. 106-126. Published by University of Newcastle upon Tyne.

Egan-Krieger, G.V., Stein, T. and Rahn, J. (1994) "Object Oriented device control using the CAN bus", Nuclear Instruments and Methods in Physics Research, North-Holland, A 352, pp. 204-206, 1994.

Eggermont, L. (2002) "Embedded Systems Roadmap 2002", STW Technology Foundation, 2002, (Last accessed: October 2008) http://www.stw.nl/Programmas/Progress/ESroadmap.htm

Ekelin, C. and Jonsson, J. (2001) "Evaluation of search heuristics for embedded system scheduling problems", In Proc. Int. Conf. Principles and Practice of Constraint Programming, Paphos, Cyprus, pp. 640-654.

Engblom, J., Ermedahl, A., Sjoedin, M., Gustafsson, J. and Hansson, H. (2001). "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems". Journal of Software Tools for Technology Transfer.

En-Nouaary , A., Dssouli, R., Khendek, F. and Elqortobi, A. (1998) "Timed Test Cases Generation Based on State Characterisation Technique", In 19th IEEE Real-Time Systems Symposium (RTSS'Y8), Madrid, Spain, December, 2-4 1998.

En-Nouaary, A., Khendek, F. and Dssouli, R. (1999) "Fault Coverage in Testing Real-Time Systems", In 6th International Conference on Real-Time Systems Computing Systems and Applications (RTCSA'YY), Hong Kong, December, 13-15 1999.

En-Nouaary, A., Dssouli, R. and Khendek, F. (2002) "Timed Wp-method: testing real-time systems", IEEE Transactions on Software Engineering, Vol. 28 (11), pp. 1023- 1038.

ESD (2006) "2006 – Embedded Systems Design – State of Embedded Market Survey", Embedded.com, WWW website (Last accessed: October 2008) http://www.embedded.com/columns/survey

ESL (2008) "Embedded Systems Laboratory - Publications", The University of Leicester, Department of Engineering, WWW website (Last accessed: October 2008) http://www.le.ac.uk/eg/embedded/publications.htm

Eswaran, A., Rowe, A. and Rajkumar, R. (2005) "Nano-RK: An Energy-Aware Resource-Centric Operating System for Sensor Networks", Proceedings of the 26th IEEE Real-Time Systems Symposium, December 2005.

Fagan, M. (1976) "Design and code inspections to reduce errors in program development", IBM System Journal, vol. 15 (3).

Fagan, M.E. (1986) "Advances in software inspections", IEEE Transactions on Software Engineering, Vol. 12 (7), pp. 744-751.

Farsi, M., Ratcliff, K. and Barbusa, M. (1999) "An Overview of Controller Area Network", Computing and Control Engineering Journal, June 1999.

Farsi, M. and Barbosa, M. (2000) "CANopen Implementation, applications to industrial networks", Research Studies Press Ltd, England.

Fisher, J.A., Faraboschi, P. and Young, C. (2004) “Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools”, Morgan Kaufmann.

FlexRay (2004) “FlexRay Communications System Protocol Specification Version 2.0”, FlexRay Consortium, 2004.

Flynn, I.M. (2001) “Generations, Languages”, Macmillan Science Library: Computer Sciences, WWW website (Last accessed: October 2008) http://www.bookrags.com/research/generations-languages-csci-01/

Fredriksson, L.B. (1994) “Controller Area Networks and the protocol CAN for machine control systems”, Mechatronics, Vol.4 (2), pp. 159-192.

Fuhrer, D., Mao, H. and Poore, J.H. (1992) “OS/2 cleanroom environment: a progress report on a cleanroom tools development project”, Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992, Vol. 2, pp. 449-458.

Fuhrer, T., Muller, B., Dieterle, W., Hartwich, F., Hugel, R. and Walther, M. (2000) “Time Triggered Communication on CAN”, In Proceedings of the 7th International CAN Conference, 2000.

Futatsugi, K., Goguen, J.A. and Jouannaud, J.P (1985) "Principles of OBJ2", Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, ACM 1985, pp. 52-66.

Ganssle, J. (1992) “The art of programming embedded systems”, Academic Press, San Diego, USA.

Gargantini, A., Riccobene, E. and Scandurra, P. (2008) “A model-driven validation & verification environment for embedded systems”, International Symposium on Industrial Embedded Systems, SIES 200, 11-13 June 2008, pp. 241-244.

GCSSDT (1995) “Glossary of Computerized System and Software Development Terminology”, available online (Last accessed: October 2008) http://www.nthanalytics.com/doc/1995%20software%20validation%20glossary.pdf

Gendy, A. and Pont, M.J. (2007) “Towards a generic 'single-path programming' solution with reduced power consumption”, Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), September 4-7, 2007, Las Vegas, Nevada, USA.

Gendy, A.K. and Pont, M.J. (2008) “Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems”, IEEE Transactions on Industrial Informatics, Vol. 4 (1), pp. 37-46.

Gerber, R., Hong, S., and Saksena, M. (1995) “Guaranteeing real-time requirements with resource-based calibration of periodic processes”, IEEE Transactions on Software Engineering, Vol. 21 (7), pp. 579-592.

Gergeleit, M. and Nett, E. (2002). “Scheduling Transient Overload with the TAFT Scheduler”. GI/ITG specialized group of operating systems, Berlin.

Gilb, T., and Graham, D. (1993) “Software inspections”, Addison-Wesley.

Golatowski, F., Hildebrandt, J., Blumenthal, J. and Timmermann, D. (2002) “Framework for validation, test and analysis of real-time scheduling algorithms and scheduler implementations”, Proceeding of the 13th IEEE International Workshop on Rapid System Prototyping, 1-3 July 2002, pp. 146-152.

Goodenough, J. B. and Sha, L. (1988) “The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks”, in Proc. 2nd ACM Int. Workshop Real-Time Ada Issues, 1988.

Graaf, B., Lormans, M. and Toetenel, H. (2003) “Embedded software engineering: The state of the practice”, IEEE Software, Vol. 20 (6), pp. 61-69.

Grady, R.B. (1992) “Practical Software Metrics for Project Management and Process Improvement”, Prentice Hall.

Grady, R.B. and Van Slack, T. (1994) “Key lessons in achieving widespread inspection use”, IEEE Software, Vol. 11 (4), pp. 46-57.

Grogono, P. (1999) “The Evolution of Programming Languages”, Course Notes, Department of Computer Science, Concordia University, Montreal, Quebec, Canada.

Gulliver, S.R. and Ghinea, G. (2007) “The Perceptual Influence of Multimedia Delay and Jitter”, 2007 IEEE International Conference on Multimedia and Expo, 2-5 July 2007, pp. 2214-2217.

Guttag, J. and Homing, J. (1993) “Larch: Languages and Tools for Formal Specification”, Springer-Verlag.

Halang, W.A. and Stoyenko, A.D. (1990) “Comparative evaluation of high-level real-time programming languages”, Real-Time Systems, Vol. 2 (4), pp. 365-382.

Hall, E.C. (2000) “From the farm to pioneering with digital control computers: an autobiography”, Annals of the History of Computing, IEEE Vol. 22 (2), pp. 22-31.

Hansen, P.B. (1975) “The programming language Concurrent Pascal”, IEEE Transactions on Software Engineering, Vol. 1 (2), pp. 199-207.

Harjumaa, L. and Tervonen, I. (1998) “A WWW-based tool for software inspection”, Proceedings of the Thirty-First Hawaii International Conference on System Sciences, 6-9 Jan 1998, Vol. 3, pp. 379-388.

Hartwich, F., Muller, B., Fuhrer, T., Hugel, R. and GmbH, R.B. (2002). "Timing in the TTCAN network", Proceedings of the 8th International CAN Conference.

Hedenetz, B. and Belschner, R. (1998), “Brake-By-Wire Without Mechanical Backup ByUsing A TTP-Communication Network”, (Last accessed: October 2008) http://www.vmars.tuwien.ac.at/projects/xbywire/projects/new-BBW.html

Heninger, K.L. (1980) “Specifying Software Requirements for Complex Systems: New Techniques and Their Application”, IEEE Transactions on Software Engineering, Vol. 6 (1), pp. 2-13.

Hessel, A. (2007) “Model-Based Test Case Generation for Real-Time Systems”, PhD thesis, Department of Information Technology, Uppsala University, Sweden.

Hevner, A.R. Becker, S.A. and Pedowitz, L.B. (1992) “Integrated CASE for cleanroom development”, IEEE Software, Vol. 9 (2), pp. 69-76.

Hohmeyer, R.E. (1968) “CDC 1700 FORTRAN for process control”, IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. 15, pp. 67-70.

Holyer, I (2008) “Dictionary of Computer Science”, Department of Computer Science, University of Bristol, UK, WWW website (Last accessed: October 2008) http://www.cs.bris.ac.uk/Teaching/Resources/COMS11200/jargon.html

Holzmann, G.J. (1997) “Design and validation of computer protocols”, IEEE Transactions on on Software Engineering, Vol. 23 (5), pp. 279-295.

Hong, S. (1995) “Scheduling Algorithm of Data Sampling Times in the Integrated Communication and Control Systems”, IEEE Transactions on Control Systems Technology, Vol. 3 (2), pp. 225-230.

Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y. and Chen, C. (1994) “Formal Approach to Scenario Analysis”, IEEE Software, Vol. 11, pp. 33-41.

Hsia, P., Kung, D. and Sell, C. (1997) “Software Requirements and Acceptance Testing”, Annals of Software Engineering, Vol. 3, pp. 291-317.

Hsieh, C-C. and Hsu, P-L. (2005) “The event-triggered network control structure for CAN-based motion system”,Proceeding of the 2005 IEEE conference on Control Applications, Toronto, Canada, August 28 – 31, 2005.

Huajin, S., Deyuan, G, Shengbing, Z. and Danghui, W. (2002) “Design fast round robin scheduler in FPGA”, IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions, 29 June – 1 July 2002, Vol. 2, pp.1257-1261.

Huang, H.W. (2004) “PIC Microcontroller: An Introduction to Software & Hardware Interfacing”, Thomson Delmar Learning.

Hughes, Z.M. and Pont, M.J. (2004) “Design and test of a task guardian for use in TTCS embedded systems”. In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp. 16-25. Published by University of Newcastle upon Tyne.

Hughes, Z.M., Pont, M.J. and Ong, H.L.R. (2005) "The PH Processor: A soft embedded core for use in university research and teaching". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 224-245. Published by University of Newcastle upon Tyne.

Hughes, Z.M. and Pont, M.J. (in press) “Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed”, Trans Institute of Measurement and Control.

IEEE Std (1990) “IEEE Standard Glossary of Software Engineering Terminology”, The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA.

IFIP-ICC (1966) “The IFIP-ICC Vocabulary of Information Processing”, North-Holland Pub. Co., Amsterdam.

Ince, D (1987) “the automatic generation of test data”, The Computer Journal, Vol. 30 (1), pp. 63-69.

Infineon (2000) “C167CR Derivatives 16-Bit Single-Chip Microcontroller”, Infineon Technologies.

ISO (2001) “ISO 5127 Information and documentation –Vocabulary”, International Organisation for Standardisation (ISO).

ISO 15622 (2003) “Adaptive Cruise Control Systems – Performance Requirements And Test Procedures”, International Standards Organisation, Geneva, Switzerland.

Jackson, A. and Hoffman, D. (1994) “Inspecting Module Interface Specifications”. Software Testing, Verification and Reliability, Vol. 4 (2), pp. 101-117.

Jalote, P. (1997) “An integrated approach to software engineering”, Springer-Verlag.

Jarvis, P.H. (1968) “Some experiences with process control languages,” IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. 15, pp. 54-56.

Jensen, E.D., Locke, C.D. and Tokuda, H. (1985) “A time-driven scheduling model for real-time operating systems”, In Proceedings of Real-Time Systems Symposium, December 1985, pp. 112-122.

Jerri, A.J. (1977), “The Shannon sampling theorem: its various extensions and applications a tutorial review”, Proc. of the IEEE, Vol. 65, pp. 1565-1596.

Johansson, K.H., Törngren, M. and Nielsen, L. (2005) "Vehicle applications of controller area network", In D. Hristu-Varsakelis and W. S. Levine, Ed., "Handbook of Networked and Embedded Control Systems", Springer.

Jones, C.B. (1989) “Systematic Software Development using VDM”, Prentice Hall.

Jones, M. (1997) “What really happened on Mars?”, WWW website (Last accessed: October 2008) http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

Jones, N. (2002) “Introduction to MISRA C”, Embedded.com, WWW website (Last accessed: October 2008) http://www.embedded.com/columns/beginerscorner/9900659

Kalinsky, D. (2001) “ Context switch, Embedded Systems Programming”, Vol. 14(1), 94-105.

Kamal, R. (2003) “Embedded Systems: Architecture, Programming and Design”, McGraw-Hill.

Katcher, D., Arakawa, H. and Strosnider, J. (1993) “Engineering and analysis of fixed priority schedulers”, IEEE Transactions on Software Engineering, Vol. 19 (9), pp. 920-934.

Kazman, R., Klein, M. and Clements, P. (2000) “ATAM: Method for architecture evaluation”, CMU/SEI, 2000.

Kelly, J.C., Sherif, J.S. and Hops, J. (1992) “An Analysis of Defect Densities Found During Software Inspections”, Journal of Systems Software, Vol. 17, pp. 111-117.

Key, S.A., Pont, M.J. and Edwards, S. (2003) “Implementing low-cost TTCS systems using assembly language”, In: Henney, K. and Schutz, D. (Eds) Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 2003), Germany, June, pp. 667-690, Published by Universitätsverlag Konstanz.

Key, S. and Pont, M.J. (2004) “Implementing PID control systems using resource-limited embedded processors”. In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp. 76-92. Published by University of Newcastle upon Tyne.

Kim, N., Ryu, M., Hong, S. and Shin, H. (1999) “Experimental Assessment of the Period Calibration Method: A Case Study”, Real-Time Systems, Vol. 17 (1), pp. 41-64.

Kim, T.H., Hwang, I.S., Jang, M.S., Kang, S.W., Lee, J.Y. and Lee, S.B. (1998) “Test case generation of a protocol by a fault coverage analysis”, Proceedings of the Twelfth International Conference on Information Networking, 1998 (ICOIN-12), 21-23 Jan 1998, pp. 690-695.

Kircher, O. and Turner, E.B. (1968) “On-line MISSIL”, IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. 15, pp. 80-84.

Kirner, R. and Puschner, P. (2003). “Discussion of Misconceptions about Worst-Case Execution-Time Analysis”. 3rd Euromicro International Workshop on WCET Analysis.

Knuth, D. (1998) “The Art of Computer Programming”, Addison-Wesley.

Koch, B. (1999) “The Theory of Task Scheduling in Real-Time Systems: Compilation and Systematization of the Main Results”, Studies thesis, University of Hamburg.

Konrad, S., Cheng, B.H. C. and Campbell, L.A. (2004) “Object analysis patterns for embedded systems”, IEEE Transactions on Software Engineering, Vol. 30 (12), pp. 970- 992.

Kontak, R.E. (1988) “Applicability of Ada tasking for avionics executives”, Proceedings of the IEEE 1988 National Aerospace and Electronics Conference (NAECON), 23-27 May, Vol. 2, pp. 739-746.

Kopetz, H. (1991a) “Event-triggered versus time-triggered real-time systems”, In: Proceedings of the InternationalWorkshop on Operating Systems of the 90s and Beyond, London, UK, Springer-Verlag, pp. 87-101.

Kopetz, H. (1991b), “Event-triggered versus time-triggered real-time systems”, Technical Report 8/91, Technical University of Vienna, Austria.

Kopetz, H. (1997) “Real-time systems: Design principles for distributed embedded applications”, Kluwer Academic.

Kopetz, H. (2001) “A Comparison of TTP/C and FlexRay”, Real-Time Systems Group, Vienna University of Technology.

Kravetz, M. and Franke, H. (2001) “Implementation of a Multi-Queue Scheduler for Linux”, IBM Linux Technology Center, Version 0.2, April 2001.

Krishnan, R. (2005) “Future of Embedded Systems Technology”, Published by BBC Research Group, WWW website (Last accessed: October 2008) http://www.bccresearch.com/report/IFT016B.html

Kühnel, C. (2006) “AVR RISC Microcontroller Handbook”, Newnes.

Kurian, S. and Pont, M.J. (2005) “Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples”, In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 36-59. Published by University of Newcastle upon Tyne.

Kurian, S. and Pont, M.J. (2006a) “Evaluating and improving pattern-based software designs for resource-constrained embedded systems”, In: C. Guedes Soares & E. Zio (Eds), “Safety and Reliability for Managing Risk: Proceedings of the 15th European Safety and Reliabilty Conference (ESREL 2006), Estoril, Portugal, 18-22 September 2006”, Vol. 2, pp. 1417-1423. Published by Taylor and Francis, London.

Kurian, S. and Pont, M.J. (2006b) “Restructuring a pattern language which supports time-triggered co-operative software architectures in resource-constrained embedded systems”, Paper presented at the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Germany, July 2006.

Kurian, S. and Pont, M.J. (2007) “Maintenance and evolution of resource-constrained embedded systems created using design patterns”, Journal of Systems and Software, Vol. 80 (1), pp. 32-41.

Labrosse, J.J. (2000) “Embedded Systems Building Blocks: Complete and Ready-to-use Modules in C”, Focal Press.

LabVIEW (2007) “LabVIEW 7.1 Documentation Resources”, WWW website (Last accessed: October 2008) http://digital.ni.com/public.nsf/allkb/06572E936282C0E486256EB0006B70B4

Lambert, K.A. and Osborne, M. (2000) “Java: A Framework for Program Design and Data Structures”, Brooks / Cole.

Laplante, P.A. (2004) “Real-time Systems Design and Analysis”, Wiley-IEEE.

Laria, G (2005) “Architectural Evaluation and Assessment”, CRMPA.

Larsen, K.G., Pettersson, P. and Yi, W. (1997) “UPPAAL in a Nutshell”, Int. Journal on Software Tools for Technology Transfer, Vol. 1 (1–2), pp. 134-52.

Larsen, K.G., Mikucionis, M. and Nielsen, B. (2005) “Online testing of real-time systems using Uppaal”, In J. Gabowski and B. Nielsen, editors, Proc. 4, International Workshop on Formal Approaches to Testing of Software 2004 (FATES’04), Vol. 3395 of Lecture Notes in Computer Science, pp. 79-94. Springer–Verlag, 2005.

Lee, D and Allan, G. (2003). "Fault-tolerant clock synchronisation with microsecond-precision for CAN networked systems", International CAN Conference, Munich, Germany, October 2003.

Leen, G., Heffernan, D. and Dunne, A. (1999) “Digital networks in the automotive vehicle”, Computing and Control, Vol. 10 (6), pp. 257-266.

Leen, G. and Heffernan, D. (2001) “Time-Triggered Controller Area Network”, Computing and Control Engineering Journal, December 2001, Vol. 12, (6).

Leen, G. and Heffernan, D. (2002) “TTCAN: a new time-triggered controller area network”, Microprocessors and Microsystems, Vol. 26 (2), pp. 77-94.

LeGrand, S. (1988) “Ada task scheduling: A focused Ada investigation”, technical report published in NASA Technical Reports Server (NTRS), NASA, U.S.

Leung J.Y.T. and Whitehead, J. (1982) “On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks”, Performance Evaluation, Vol. 2, pp. 237-250.

Liberty, J. and Jones, B. (2004) “Teach Yourself C++ in 21 Days”, Sams.

Lin, K.J. and Herkert, A. (1996), “Jitter Control in Time-Triggered Systems”, Proceedings of the 29th Hawaii International Conference on System Sciences, Maui, Hawaii, pp. 451-459.

Linger, R.C. (1994) “Cleanroom process model”, IEEE Software, Vol. 11 (2), pp. 50-58.

Litterick, M. and Brenner, M. (2005). "Utilizing Vera functional coverage in theverification of a protocol engine for the FlexRay automotive communication system", The fourteenth Annual Conference of Synopsys Users Group (SNUG) Europe, Munich, Germany.

Liu, C.L. and Layland, J.W. (1973), “Scheduling algorithms for multi-programming in a hard real-time environment”, Journal of the AVM 20, Vol. 1, pp. 40-61.

Liu, J.W.S. (2000), “Real-time systems”, Prentice Hall.

Liu, S., Asuka, M., Komaya, K. and Nakamura, Y. (1998) “An approach to specifying and verifying safety-critical systems with practical formal method SOFL”, Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems, 1998, ICECCS '98,10-14 Aug. 1998, pp. 100-114.

Liu, Z., Gu, N. and Yang, G. (2005) “An automate test case generation approach: using match technique”, The Fifth International Conference on Computer and Information Technology, CIT 2005, 21-23 Sept. 2005, pp. 922-926.

Locke, C.D. (1986) “Best-effort decision making for real-time scheduling”, PhD thesis, Department of Computer Science, Carnegie Mellon University, USA.

Locke, C.D., Vogel, D.R. and Mesler, T.J. (1991) “Building a Predictable Avionics Platform in Ada: A Case Study”, Proceedings of IEEE Real Time Systems Symposium. pp. 181-189.

Locke, C.D. (1992), “Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives”, Real-Time Systems, Vol. 4, pp. 37-52.

Lutz, R.R. (1993) “Analysing software requirements errors in safety-critical embedded systems”, Proc. RE’93, San Diego CA: IEEE Computer Society Press.

Maaita, A. and Pont, M.J. (2005) “Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler”. In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 18-35. Published by University of Newcastle upon Tyne

Marsh, P. (2003) “Models of control”, IEE Electronics Systems and Software, Vol. 1 (6), pp. 16-19.

Marti, P., Fuertes, J.M., Ramamritham, K. and Fohler, G. (2001a), “Jitter Compensation for Real-Time Control Systems”, 22nd IEEE Real-Time Systems Symposium (RTSS'01), London, England, pp. 39-48.

Marti, P., Fuertes, J.M., Villa, R. and Fohler, G. (2001b), “On Real-Time Control Tasks Schedulability”, European Control Conference (ECC01), Porto, Portugal, pp. 2227-2232.

Marti, P. (2002), “Analysis and design of real-time control systems with varying control timing constraints”, PhD thesis, Automatic Control Department, Technical University of Catalonia.

Martin, J. and Leben, J. (1986) “Fourth Generation Languages Volume 1: Principles”, Prentice Hall.

Marwedel, P (2006) “Embedded system design”, Springer.

McCormick, F. (1987) “Scheduling difficulties of Ada in the hard real-time environment”, Proceedings of the first international workshop on Real-time Ada issues, Morehampstead, Devon, United Kingdom, pp. 49 – 50.

McElhone , C. and Burns, A. (2000) “Scheduling Optional Computations for Adaptive Real-Time Systems”, Journal of Systems Architectures.

McLaughlin, M. and Moore, A. (1998) “Real-Time Extensions to UML”, Published by Dr Dobb’s Potral, The Word for Software development, WWW website (Last accessed: October 2008) http://www.ddj.com/184410749

Mensh, M. and Diehl, W. (1968) “Extended FORTRAN for process control”, IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. 15, pp. 75-79.

Mills, H.D., Dyer, M. and Linger, R.C. (1987) “Cleanroom Software Engineering”, IEEE Software, Vol. 4 (5), pp. 19-25.

Misbauddin, S. and Al-Holou, N. (2003) “Efficient data communication techniques for controller area network (CAN) protocol”, ACS/IEEE International Conference on Computer Systems and Applications, 2003, Book of Abstracts, pp. 22.

Mitchell, J.C. (2003) “Concepts in Programming Languages”, Cambridge University Press.

Mok, A.K. (1983) “Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment”, Ph.D Thesis, MIT, USA.

Mooney, V., Sakamoto, T. and De Micheli, G. (1997) “Run-time scheduler synthesis for hardware-software systems and application to robot control design”, Proceedings of the Fifth International Workshop on Hardware/Software Codesign, (CODES/CASHE '97), 24-26 March 1997, pp. 95-99.

Mooney, V.J. (1999) “Path-based edge activation for dynamic run-time scheduling”, System Synthesis, 1999. Proceedings of the 12th International Symposium on 10-12 Nov. 1999, pp. 30-36

Muller, B., Hartwich, F., Fuehrer, T., Hugel, R., Weiler, H. and Bosch R (2002) “Fault-tolerant TTCAN networks”, Proceedings of the 8th International CAN Conference (iCC), 2002.

Munoz, C.U. (1988) “An approach to software product testing”, IEEE Transactions on Software Engineering, Vol. 14 (11), pp. 1589-1596.

Murray, C.J. (2002) “Time-triggered protocol gains aerospace mileage”, EE Times, WWW website (Last accessed: October 2008) http://www.eetimes.com/story/OEG20020912S0061

Mwelwa, C. and Pont, M.J. (2003) “Two new patterns to support the development of reliable embedded systems”, Paper presented at VikingPLoP 2003 (Bergen, Norway, September 2003).

Mwelwa C., Pont M.J. and Ward D. (2003) “Towards a CASE Tool to Support the Development of Reliable Embedded Systems Using Design Patterns”, In: Bruel, J-M [Ed.] Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering, June 20th 2003, Toulouse, France, Published by Cepadues-Editions, Toulouse.

Mwelwa, C., Pont, M.J. and Ward, D. (2004) “Code generation supported by a pattern-based design methodology”, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp. 36-55. Published by University of Newcastle upon Tyne

Mwelwa, C., Pont, M.J. and Ward, D. (2005) "Developing reliable embedded systems using a pattern-based code generation tool: A case study". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 177-193. Published by University of Newcastle upon Tyne.

Mwelwa, C. (2006) “Development and Assessment of a Tool to Support Pattern-Based Code Generation of Time-Triggered (TT) Embedded Systems”, PhD thesis, Department of Engineering, University of Leicester, UK.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D. (2006) “Rapid software development for reliable embedded systems using a pattern-based code generation tool”, Paper presented at the Society of Automotive Engineers (SAE) World Congress, Detroit, Michigan, USA, April 2006. SAE document number: 2006-01-1457. Appears in: Society of Automotive Engineers (Ed.) “In-vehicle software and hardware systems”, Published by Society of Automotive Engineers.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D. (2007) "Rapid software development for reliable embedded systems using a pattern-based code generation tool". SAE

Transactions: Journal of Passenger Cars (Electronic and Electrical Systems), Vol. 115 (7), pp. 795-803.

Nahas, M., Pont, M.J. and Jain, A. (2004) "Reducing task jitter in shared-clock embedded systems using CAN", In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp. 184-194. Published by University of Newcastle upon Tyne.

Nahas, M. and Pont, M.J. (2005) "Using XOR operations to reduce variations in the transmission time of CAN messages: A pilot study". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 4-17. Published by University of Newcastle upon Tyne.

Nahas, M., Short, M. and Pont, M. J. (2005) "The impact of bit stuffing on the real-time performance of a distributed control system", Proceeding of the 10th International CAN conference iCC (Rome, Italy, March 2005), pp. 10-1 to10-7.

Nahas, M., Short, M. and Pont, M. J. (submitted) "Reducing message-length variations in resource-constrained embedded systems implemented using the CAN protocol", Submitted for a journal.

National Instruments (2006) "Low-Cost E Series Multifunction DAQ – 12 or 16-Bit, 200 kS/s, 16 Analog Inputs", available online (Last accessed: October 2008) http://www.ni.com/pdf/products/us/4daqsc202-204_ETC_212-213.pdf

Navet, N. and Song, Y.Q. (1998) "Design of reliable real time applications distributed over CAN (controller area network)", Proceedings of INCOM'98, IFAC Symposium on Information Control in Manufacturing, Metz 22–24 June, 1998, pp. 391-396.

Nett, E., Streich, H., Bizzarri, P., Bondavalli, A. and Tarini, F. (1996). "Adaptive Software Fault Tolerance Policies with Dynamic Real-Time Guarantees". WORDS 96, IEEE Second Int. Workshop on Object oriented Real-time Dependable Systems, Laguna Beach, California, U.S.A.

Network Dictionary (2008) "Concurrent programming", WWW website (Last accessed: October 2008) http://wiki.networkdictionary.com/index.php/Concurrent_programming

Nghiem, T., Pappas, G.J., Alur, R. and Girard, A. (2006) "Time-triggered implementations of dynamic controllers", Proceedings of the 6th ACM & IEEE International conference on Embedded software, Seoul, Korea, pp. 2-11.

Nielsen, B. and Skou, A. (2003) "Automated test generation from timed automata", International Journal on Software Tools for Technology Transfer, pp. 1023–1038.

Nielsen, M., Havelund, K., Wagner, K.R. and George, C. (1988) "The RAISE Language, Method and Tools", Proceedings of the Europe Symposium on VDM, pp. 376-405.

Nissanke, N. (1997) "Real-time Systems", Prentice-Hall.

NIST (2007) "The National Institute of Standards and Technology", WWW website (Last accessed: October 2008) http://www.nist.gov/

Nolte, T., Hansson, H., Norström, C. and Punnekkat, S. (2001) "Using Bit-stuffing Distributions in CAN Analysis", IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium) London.

Nolte, T., Hansson, H. and Norstrom, C. (2002), "Minimizing CAN response-time jitter by message manipulation", IEEE Real Time Technology and Applications Symposium 2002, pp. 197-206.

Nolte, T. (2003), "Reducing Pessimism and Increasing Flexibility in the Controller Area Network", PhD thesis, Department of Computer Science and Engineering, Malardalen University, Vasteras, SWEDEN.

Nunamaker, J.F., Dennis, A.R., Valacich, J.S., Vogel, D.R., and George, J.F. (1991) "Electronic meeting systems to support group", Communications of the ACM, July 1991, pp. 42-61.

Obermaisser, R (2004) "Event-Triggered and Time-Triggered Control Paradigms", Kluwer Academic.

Oerter, G.W. (1968) "A new implementation of decision tables for a process control language", IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. 15, pp. 57-61.

Offutt, J. and Liu, S. (1999) "Generating test data from SOFL specifications", The Journal of Systems and Software, Vol. 49 (1), pp. 49-62.

O'Halloran, C. (2000) "Issues for the automatic generation of safety critical software", The 15th IEEE International Conference on Automated Software Engineering, Grenoble, France.

Ong, C.K., Hong, D., Cheng, K.T.T. and Wang, L.C. (2004), "Jitter spectral extraction for multi-gigahertz signal", Asia and South Pacific Design Automation Conference (ASP-DAC '04), pp. 298-303.

Opler, A. (1966) "Requirements for real-time languages", Communications of the ACM, Vol. 9 (3), pp. 196-199.

O'Reilly, T. (2006) "Programming Language Trends", WWW website (Last accessed: October 2008) http://radar.oreilly.com/archives/2006/08/programming-language-trends.html

Pamas, D.L. (1994) "Inspection of safety-critical software using program-function tables", CRL Report No. 288, McMaster University, Hamilton, Canada, 1994.

Pazul, K. (1999) "Controller Area Network (CAN) Basics", Microchip Technology Inc. Preliminary DS00713A-page 1 AN713.

Phatrapornnant, T. and Pont, M.J. (2004a) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology", Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989), pp. 3-8.

Phatrapornnant, T. and Pont, M.J. (2004b) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology",

Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989), pp. 3-8.

Phatrapornnant, T. and Pont, M.J. (2006), "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling", IEEE Transactions on Computers, Vol. 55 (2), pp. 113-124.

Phatrapornnant, T. (2007) "Reducing Jitter in Embedded Systems Employing a Time-Triggered Software Architecture and Dynamic Voltage Scaling", PhD thesis, Department of Engineering, University of Leicester, UK.

Philips (1996) "P8x592 8-bit microcontroller with on-chip CAN, datasheet", Philips Semiconductor.

Philips (2004) "LPC2119/2129/2194/2292/2294 microcontrollers user manual", Philips Semiconductor.

Philips Semiconductors (2003) "LPC2106/2105/2104 USER MANUAL", available online (Last accessed: October 2008) http://www.standardics.nxp.com/products/lpc2000/datasheet/lpc2104.lpc2105.lpc2106.pdf

PhyCORE-167 (2003) "QuickStart Instructions", Phytec Technology.

Poledna, S. and Kroiss, G. (1998) "The Time-Triggered Communication Protocol TTP™/C", Real-Time Magazine, available online (Last accessed: October 2008) http://www.realtime-info.be

Pont, M.J. (2001) "Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers", ACM Press / Addison-Wesley.

Pont, M.J. (2002) "Embedded C", Addison-Wesley.

Pont, M.J. (2003) "An object-oriented approach to software development for embedded systems implemented using C", Transactions of the Institute of Measurement and Control, Vol. 25 (3), pp. 217-238.

Pont, M.J. and Mwelwa, C. (2003) "Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language", Paper presented at Viking PLoP 2003 (Bergen, Norway, September 2003).

Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", in Hruby, P. and Soressen, K. E. [Eds.]Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002, pp.159-200. Published by Micrsoft Business Solutions.

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T. (2003) "Prototyping time-triggered embedded systems using PC hardware". Paper presented at EuroPLoP 2003 (Germany, June 2003).

Pont, M.J. and Banner, M.P. (2004) "Designing embedded systems using patterns: A case study", Journal of Systems and Software, Vol. 71 (3), pp. 201-213.

Pont, M.J., Kurian, S. and Bautista-Quintero, R. (2006) "Meeting real-time constraints using 'Sandwich Delays'", In: Zdun, U. and Hvatum, L. (Eds) Proceedings of the Eleventh European conference on Pattern Languages of Programs (EuroPLoP '06), Germany, July 2006: pp. 67-77. Published by Universitätsverlag Konstanz.

Pont, M.J., Kurian, S., Wang, H. and Phatrapornnant, T. (2007) "Selecting an appropriate scheduler for use with time-triggered embedded systems", Paper presented at the twelfth European Conference on Pattern Languages of Programs (EuroPLoP 2007).

Pop, P., Eles, P. and Peng, Z. (2004) "Analysis and Synthesis of Distributed Real-Time Embedded Systems", Springer.

Porter, A.A., Votta, L.G. Jr. and Basili, V.R. (1995) "Comparing detection methods for software requirements inspections: a replicated experiment", IEEE Transactions on Software Engineering, Vol. 21 (6), pp. 563-575.

Poston, R.T. (1986) "The Automatic Test Case Data Generator", Proceedings of the 4th Annual Pacific Northwest Software Quality Assurance Conference, Sept. 1986, pp. 168-176.

Powell, D. (2002) "Deriving verification conditions and program assertions to support software inspection", Ninth Asia-Pacific Software Engineering Conference, 2002, 4-6 Dec. 2002, pp. 447-456.

Pringsulaka, P. and Daengdej, J. (2006) "Coverall algorithm for test case reduction", IEEE Aerospace Conference, 4-11 March 2006, pp. 8-15.

Profeta III, J.A., Andrianos, N.P., Bing, Yu, Johnson, B.W., DeLong, T.A., Guaspart, D. and Jamsck, D. (1996) "Safety-critical systems built with COTS", IEEE Computer, Vol. 29 (11), pp. 54-60.

Punnekkat, S., Hansson, H. and Norström, C. (2000) "Response Time Analysis under Errors for CAN", Proceedings of RTAS'2000- 6th IEEE Real-Time Technology and Applications Symposium, pp. 258-265, June 2000.

Puschner, P. (2002). "Is WCET Analysis a Non-Problem? - Towards New Software and Hardware Architectures". 2nd Intl. Workshop on Worst Case Execution Time Analysis, Vienna, Austria.

Rao, M.V.P, Shet, K.C, Balakrishna, R. and Roopa, K. (2008) "Development of Scheduler for Real Time and Embedded System Domain", 22nd International Conference on Advanced Information Networking and Applications - Workshops, 25-28 March 2008, AINAW, pp. 1-6.

Ravikumar, C. P. (2004) "Multiprocessor architectures for embedded system-on-chip applications", VLSI Design, 2004. Proceedings of the 17th International Conference, pp. 512-519.

Rayadurgam, S. (2001) "Automated test-data generation from formal models of software", Proceedings of the 16th Annual International Conference on Automated Software Engineering, 2001 (ASE 2001), 26-29 Nov. 2001, pp. 438.

Real, J. and Wellings, A.J. (1999a) "The Ceiling Protocol in Multi-moded Real-Time Systems Reliable Software Technologies", Ada-Europe 99, Lecture Notes in Computer Science.

Real, J. and Wellings, A.J. (1999b) “Implementing Mode Changes and Shared Resources in Ada”, Proceedings of the11th Euromicro Conference on Real-Time Systems 1999.

Redmill, F. (1992) “Computers in safety-critical applications”, Computing & Control Engineering Journal, Vol. 3 (4), pp.178-182.

Ribeiro, O.R. and Fernandes, J.M. (2007) “Translating Synchronous Petri Nets into PROMELA for Verifying Behavioural Properties”, International Symposium on Industrial Embedded Systems, SIES '07, 4-6 July 2007, pp. 266-273.

Roberts, B.C (1968) “FORTRAN IV in a process control environment”, IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. 15, pp. 61-63.

Rodrigues, L., Guimarães, M. and Rufino, J. (1998) “Fault-Tolerant Clock Synchronization in CAN”, Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998.

Rodriguez-Navas, G., Barranco, M. and Proenza, J. (2003) “Harmonizing Dependability and Real Time in CAN Networks” Proceedings of the 2nd International Workshop on Real-Time LANs in the Internet Age (RTLIA), 2003.

Roscoe, A.W. (1994) “Model checking CSP”, In A.W. Roscoe, editor, A Classical Mind: Essays in Honour of C.A.R. Hoare, pp. 353-378, Prentice Hall.

RTS (2008) “Real-Time Systems Research Group - Publications”, The University of York, Department of Computer Science, WWW website (Last accessed: October 2008) http://www.cs.york.ac.uk/rts/papers_db_all.php#2008

Rudiger, R. (1998) “Evaluating the temporal behaviour of CAN based systems by means of a cost functional”, Proceedings of the Fifth International CAN Conference, San Jose, CA, USA, November, 1998, pp. 10.09-10.26.

Rushby, J. (2001) “A Comparison of Bus Architectures for Safety-Critical Embedded Systems”, Computer Science Laboratory Technical Report, SRI International, 2001.

Ryan, C., Heffernan, D. and Leen, G. (2004) “Clock synchronisation on multiple TTCAN network channels”, Microprocessors and Microsystems Journal, Vol. 28 (3), pp. 95-146.

Sachitanand, N.N. (2002). “Embedded systems - A new high growth area”. The Hindu. Bangalore.

Samek, M. (2002) “Practical Statecharts in C/C++: Quantum Programming for Embedded Systems”, CMP Books.

Sammet, J.E. (1969) “Programming languages: history and fundamentals”, Prentice-Hall.

Samuelsson, T., Åkerholm, M., Nygren, P., Stärner, J. and Lindh, L. (2003) “A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software”, In: International Workshop on Advanced Real-Time Operating System Services (ARTOSS’03), 2003.

Sanders, J. (2007) “Simple Glossary”, WWW website (Last accessed: October 2007) http://www-xray.ast.cam.ac.uk/~jss/lecture/computing/notes/out/glossary/

Sandström, K. and Norström, C. (2002) “Managing complex temporal requirements in real-time control systems”, 9th IEEE Conf. Engineering of Computer-Based Systems, IEEE, Sweden.

Sanfridson, M. (2000) “Timing problems in distributed real-time computer control systems”. Technical Report, Mechatronics Lab, Department of Machine Design, Royal Institute of Technology, Stockholm, Sweden.

Schatz, B., Hain, T., Houdek, F., Prenninger, W., Rappl, M., Romberg, J., Slotosch, O., Strecker, M. and Wisspeintner, A. (2003) “CASE tools for embedded systems”, Technical University of Munich, Munich.

Scheler, F. and Schröder-Preikschat, W. (2006) “Time-Triggered vs. Event-Triggered: A matter of configuration?”, GI/ITG Workshop on Non-Functional Properties of Embedded Systems (NFPES), March 27 – 29, 2006, Nürnberg, Germany.

Schoeffler, J.D. and Temple, R.H. (1970) “A real-time language for industrial process control”, Proceedings of the IEEE, Vol. 58 (1), pp. 98-111.

Schutz, H.A. (1979) “On the Design of a Language for Programming Real-Time Concurrent Processes”, IEEE Transactions on Software Engineering, Vol. 5 (3), pp. 248-255.

Selby, R.W., Basili, V.R. and Baker, F.T. (1987) "Cleanroom Software Development: An Empirical Evaluation", IEEE Transactions on Software Engineering, Vol. 13 (9), pp. 1027-1037.

Sevillano, J.L., Pascual, A., Jiménez, G. and Civit-Balcells, A. (1998) “Analysis of channel utilization for controller area networks”, Computer Communications, Vol. 21 (16), pp. 1446-1451.

Sha, L. and Goodenough, J.B. (1990) “Real-time scheduling theory and Ada”, Computer, Vol. 23 (4), pp. 53-62.

Sha, L., Rajkumar, R. and Lehoczky, J.P. (1990), “Priority inheritance protocols: an approach to real-timesynchronization”, IEEE Transactions on Computers, Vol. 39 (9), pp. 1175-1185.

Shandle, J. (2003) “CAN: Network for Thousands of Applications outside Automotive”, techonline, WWW website (Last accessed: October 2008) http://www.techonline.com/showArticle.jhtml?articleID=192200347&queryText=CAN%3A+Network+for+Thousands+of+Applications+outside+Automotive

Shaw, A.C. (2001) “Real-time systems and software”, New York, John Wiley & Sons Inc.

Shere, K.D. and Carlson, R.A. (1994) “A methodology for design, test, and evaluation of real-time systems”, Computer, Vol. 27 (2), pp. 35-48.

Shi, H., Peleska, J., and Kouvaras, M. (1999) “Combining methods for the analysis of a fault-tolerant system”, Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing, 16-17 Dec. 1999, pp. 135-142.

Short, M., Pont, M.J. and Huang, Q. (2004a) “Simulation Of Vehicle Longitudinal Dynamics”, Technical report ESL 04/01, Embedded Systems Laboratory, University of Leicester, 2004.

Short, M., Pont, M.J. and Huang, Q. (2004b) “Simulation Of Motorway Traffic Flows”, Technical report ESL 04/02, Embedded Systems Laboratory, University of Leicester, 2004.

Short, M., Pont, M.J. and Huang, Q. (2004c) “Development Of A Hardware-In-The-Loop Test Facility For Distributed Embedded Systems”, Technical report ESL 04/03, Embedded Systems Laboratory, University of Leicester, 2004.

Short, M. and Pont, M.J. (2005) "Hardware in the loop simulation of embedded automotive control systems", In Proceedings of the 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005) held in Vienna, Austria, 13-16 September 2005, pp. 226-231.

Short, M.J., Fang, J., Pont, M.J. and Rajabzadeh, A. (2006) "Assessing the impact of redundancy on the performance of a brake-by-wire system", Paper presented at the Society of Automotive Engineers (SAE) World Congress, Detroit, Michigan, USA, April 2006. SAE document number: 2006-01-0836. Appears in: Pimental, J.R. (Ed.) "Safety-critical automotive systems", Published by Society of Automotive Engineers.

Short, M. and Pont, M.J. (2007) “Fault-Tolerant Time-Triggered Communication Using CAN”, IEEE Transactions on Industrial Informatics, Vol. 3 (2), pp. 13-142.

Short, M.J., Fang, J., Pont, M.J. and Rajabzadeh, A. (2007) "Assessing the impact of redundancy on the performance of a brake-by-wire system", SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems), Vol. 115 (7), pp. 331-338.

Short, M. and Pont, M.J. (2008) “Assessment of high-integrity embedded automotive control systems using Hardware-in-the-Loop simulation”, Journal of Systems and Software, Vol. 81 (7), pp. 1163-1183.

Sickle, T.V. (1997) “Reusable Software Components: Object-Oriented Embedded Systems Programming in C”, Prentice Hall.

Siemens (1996), “C167 Derivatives: 16-Bit CMOS Single-Chip microcontroller”, User’s Manual Version 2.0.

Siemens (1997) “C515C 8-bit CMOS microcontroller, user’s manual”, Siemens.

Silva, E.T., Jr., Wehrmeister, M.A., Becker, L.B., Wagner, F.R. and Pereira, C.E. (2005) “Design exploration in HW/SW co-design of real-time object-oriented embedded systems: the scheduler object”, 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 2005, WORDS 2005, 2-4 Feb. 2005, pp. 378-385.

Singh, H., Conrad, M. and Sadeghipour, S. (1997) “Test case design based on Z and the classification-tree method”, Proceedings of the First IEEE International Conference on Formal Engineering Methods, 1997, 12-14 Nov 1997, pp. 81-90.

Sommerville, I. (2007) “Software engineering”, 8th edition, Harlow: Addison-Wesley.

Spangler, A. (1996) “Cleanroom software engineering-plan your work and work your plan in small increments”, IEEE Potentials, Vol. 15 (4), pp. 29-32.

Specks, J.W. and Rajnak, A. (2000) "LIN — Protocol, Development Tools, and Software Interfaces for Local Interconnect Networks in Vehicles", In Proceedings of the 9th International Conference on Electronic Systems for Vehicles, Baden-Baden, Germany, 2000.

Spivey, J. M. (1988) "The Z Notation: A Reference Manual", Prentice Hall International.

Springintveld, J., Vaadranger, F. and Dargenio, P. (1997) "Testing Timed Automata", Technical Report CTIT97-17, University of Twente, Amesterdam, 1997.

Stankovic, J.A. (1988) "Misconceptions about real-time computing", IEEE Computers, Vol. 21 (10).

Stavely, A.M. (1999) "High-quality software through semiformal specification and verification", Proceedings of the 12th Conference on Software Engineering Education and Training, 1999. 22-24 March 1999, pp. 145-155.

Steusloff, H.U. (1984) "Advanced real time languages for distributed industrial process control", IEEE Computer, pp. 37-46.

STG (2008) "Software Testing Glossary", WWW website (Last accessed: October 2008) http://www.aptest.com/glossary.html

STING (1996) "STING software engineering glossary", WWW website (Last accessed: October 2008) http://www.apl.jhu.edu/Notes/Hausler/web/glossary.html

STMicroelectronics (2002) "AN1278 Application Note, LIN Solutions", WWW website (Last accessed: October 2008) http://www.st.com/stonline/products/literature/an/8130.pdf

Storey, N. (1996) "Safety-critical computer systems", Harlow, Addison-Wesley.

Stothert, A. and MacLeod, I. (1998) "Effect of Timing Jitter on Distributed Computer Control System Performance". Proceedings of DCCS'98 – 15th IFAC Workshop on Distributed Computer Control Systems, September 1998.

Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E. and Leroy, P. (2007) "Ada 2005 Reference Manual: Language and Standard Libraries", Springer.

Tai, K. C. (1993) "Predicate-Based Test Generation for Computer Programs", Proceedings of the 15th Interactional Conference on Software Engineering (ICSE), May 1993, pp. 267-276.

Tanenbaum, A.S. (1995), "Distributed Operating Systems", Prentice Hall

TechFest (1999) "TechFest Ethernet Technical Summary", available online (Last accessed: October 2008) http://www.techfest.com/networking/lan/ethernet5.htm

Texas Instruments (1988) "SN5408, SN54LS08, SN54S08SN7408, SN74LS08, SN74S08 Quadruple 2-Input Positive-AND Gates", available online (Last accessed: October 2008) 74LS08 Datasheet, available on: http://www.cs.amherst.edu/~sfkaplan/courses/spring-2002/cs14/74LS08-datasheet.pdf

Thomesse, J.P. (1998) "A review of the fieldbuses", Annual Reviews in Control, Vol. 22, pp. 35-45.

Tindell, K., Burns, A., and Wellings, A. (1992) "Allocating hard real-time tasks: An NP-hard problem made easy", Real-Time Systems, Vol. 4 (2), pp. 145-165.

Tindell, K.W. and Burns, A. (1994) "Guaranteed message latencies for distributed safety-critical hard real-time control networks", Technical Report YCS229, Dept. of Computer Science, University of York, June 1994.

Tindell, K.W., Hansson, H. and Wellings, A.J. (1994) "Analysing Real-Time Communications: Controller Area Network (CAN)", Proceedings of RTSS'94 - 15th IEEE Real-Time Systems Symposium, pp. 259–265, December 1994.

Tindell, K.W., Burns, A. and A.J. Wellings, A.J. (1995) "Calculating Controller Area Network (CAN) Message Response Times", Control Engineering Practice, Vol. 3 (8), pp. 1163-1169.

Torngren, M. (1998), "Fundamentals of implementing real-time control applications in distributed computer systems", Real-Time Systems, Vol. 14, pp. 219-250.

Tran, E. (1999) "Verification/Validation/Certification", Carnegie Mellon University, Dependable Embedded Systems, Spring 1999, available online (Last accessed: October 2008) http://www.ece.cmu.edu/~koopman/des_s99/verification/

Tsai, W.T., Yu, L., Liu, X.X., Saimi, A. and Xiao, Y. (2003) "Scenario-based test case generation for state-based embedded systems", Conference Proceedings of the 2003 IEEE International Performance, Computing, and Communications Conference, 2003, 9-11 April 2003, pp. 335-342.

TTA-Group (2007) "The Cross-Industry Consortium for Time-Triggered Systems", WWW website (Last accessed: October 2008) http://www.ttagroup.org/index.htm

TTTech (2008) "Time-Triggered Technology - TTP", WWW website (Last accessed: October 2008) http://www.tttech.com/solutions/ttp/

Turski, K. (1994) "A global time system for CAN networks", In Proceedings of the 1st International CAN Conference, 1994, pp. 31-36.

Vallerio, K.S. and Jha, N.K. (2003) "Task graph extraction for embedded system synthesis", Proceedings of the 16th International Conference on VLSI Design concurrently with the 2nd International Conference on Embedded Systems Design, pp. 480-486.

Vardanega, T. (1996) "Tool support for the construction of statically analysable hard real-time Ada systems", 17th IEEE Real-Time Systems Symposium, 4-6 December 1996, pp. 129-135.

Verissimo, P. and Rodrigues, L. (1992) "A posteriori Agreement for Fault-Tolerant Clock Synchronization on Broadcast Networks", the 22nd International Symposium on Fault-Tolerant Computing, Boston, USA, July, 1992.

Vetromille, M., Ost, L., Marcon, C.A.M., Reif, C., Hessel, F. (2006) "RTOS Scheduler Implementation in Hardware and Software for Real Time Applications", Seventeenth IEEE International Workshop on Rapid System Prototyping, 14-16 June 2006, pp. 163-168.

Walls, C. (2005) "Embedded Software: The Works", Newnes.

Wang, B. and Lin, Z.H. (2001) "Formal verification of embedded SoC", Proceedings of the 4th International Conference on ASIC, 23-25 Oct. 2001, pp.769-772.

Wang, F. (2004) "Formal verification of timed systems: a survey and perspective", Proceedings of the IEEE, Vol. 92 (8), pp. 1283-1305.

Wang, H., Pont, M.J. and Kurian, S. (2007) "Patterns which help to avoid conflicts over shared resources in time-triggered embedded systems which employ a pre-emptive scheduler", Paper presented at the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007).

Ward, N.J. (1991) "The static analysis of a safety-critical avionics control systems", Air Transport safety: Proceedings of the Safety and Reliability Society Spring Conference, In: Corbyn D.E. and Bray, N.P. (Eds.)

Watkinson, J. (2002) "Introduction to Digital Audio", Focal Press.

Watson, D. (1989) "High Level Languages and Their Compilers", Addison-Wesley.

Wavecrest (2001), "Understanding Jitter: Getting Started", Wavecrest Corporation.

Wellings, A.J. (2003) "Is Java augmented with the RTSJ a better realtime systems implementation technology than Ada 95?", In Proceedings of IRTAW12, Ada Letters, Vol. XXIII (4), pp. 16-21.

Wellings, A.J. and Burns, A. (2007a) "Beyond Ada 2005: allocating tasks to processors in SMP systems", Proceedings of IRTAW 13, Ada Letters, Vol. XXVII (2).

Wellings, A.J. and Burns, A. (2007b) "A framework for real-time utilities for Ada 2005", Proceedings of IRTAW 13, Ada Letters, Vol. XXVII (2).

Wendorf, J.W. (1988) "Implementation and evaluation of a time-driven scheduling processor", IEEE Real-Time Systems Symposium, December 1988, pp.172-180.

Wexelblat, L. (1981) "History of Programming Languages", Academic Press.

Whalen, M.W. and Heimdahl, M.P.E. (1999) "On the requirements of high-integrity code generation", Proceedings of the 4th High Assurance in Systems Engineering Workshop, Washington DC.

Wheeler, D.A., Brykczynski, B. and Meeson, R.N.Jr. (1996) "Software Inspection: And Industry Best Practice", IEEE Computer Society Press.

White, L.J. and Sahay, P.N. (1985) "A Computer System for Generating Test Data using the Domain Strategy", Proceedings of SOETFADUI - 2nd Conference on Software Development Tools, Techniques and Altematives, 1985, pp. 38-45.

Wikipedia (2008) "Programming Language" WWW website (Last accessed: October 2008) http://en.wikipedia.org/wiki/Programming_language

Wilson, L.B. and Clark, R.G. (2000) "Comparative Programming Languages", Addison-Wesley.

Wirth, N. (1977) “Modula - A programming language for modular multiprogramming”, Software - Practice and Experience, Vol. 7, pp. 3-35.

Wirth, N (1993) “Recollections about the development of Pascal”, Proceedings of the 2nd ACM SIGPLAN conference on history of programming languages, pp. 333-342.

Wizitt (2001) “T223 – A Glossary of Terms (Block 2)”, Wizard Information Technology Training (Wizitt), WWW website (Last accessed: October 2008) http://wizitt.com/t223/glossary/glossary2.htm

Wordsworth, J. (1996) “Software Engineering with B”, Addison-Wesley.

Xu , J. and Parnas, D.L. (1990) “Scheduling processes with release times, deadlines, precedence and exclusion relations“, IEEE Transactions on Software Engineering, Vol. 16 (3), pp. 360-369.

Xu , J. and Parnas, D.L. (1993) “On satisfying timing constraints in hard - real - time systems”, IEEE Transactions on Software Engineering, Vol. 19 (1), pp. 70-84.

Xu, J. (2003) “On Inspection and Verification of Software with Timing Requirements”, IEEE Transactions on Software Engineering, Vol. 29 (8), pp. 705-720.

Yi, K., Cho, Y., Lee, S., Lee, J. and Ryoo, N. (2000) “A Throttle/Brake Control Law for Vehicle Intelligent Cruise Control”, Seoul 2000 FISITA World Automotive Congress, June 12-15, Seoul, Korea.

Zerzelidis, A., Burns, A. and Wellings, A.J. (2007) “Correcting the EDF protocol in Ada 2005”, Proceedings of IRTAW 13, Ada Letters, Vol. XXVII (2).

Zuberi, K.M. and Shin, K.G. (1995) "Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications", In Proceedings of Real-Time Technology and Applications Symposium, pp. 240-249.

Zurell, K. (2000) “C programming for embedded systems”, CMP Books.

Zuse, K (1995) “A Brief History of Programming Languages”, Byte.com, WWW website (Last accessed: October 2008) http://www.byte.com/art/9509/sec7/art19.htm