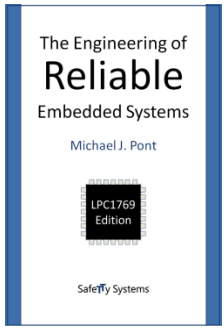
The Engineering of Reliable Embedded Systems

LPC1769 edition

Michael J. Pont



This document includes extracts from the book:

Pont, M.J. (2014) *"The Engineering of Reliable Embedded Systems: LPC1769 edition"*, Published by SafeTTy Systems Ltd. ISBN: 978-0-9930355-0-0.

Last updated: 9 July 2016

This is an extract from the first edition of 'ERES' ('ERES1').

We have now released the complete 'ERES1' book in PDF form.

You can download the book here (free of charge):

http://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems

The second edition of ERES ('ERES2') is now also available:

http://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems-second-edition

Published by SafeTTy Systems Ltd
www.SafeTTy.net

First published 2014
First printing December 2014
Second printing (with corrections) January 2015

ISBN 978-0-9930355-0-0

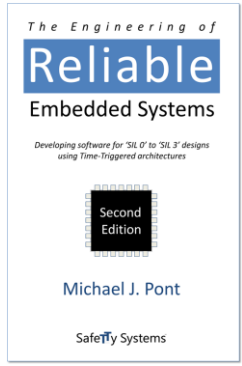
British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library.

This book is dedicated to Benjamin, Timothy, Rowena, Jonathan and Eliza.

This is an extract from the first edition of ‘ERES’.

You’ll find extracts from the second edition of this book here:

http://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems-second-edition



Contents

Definitions

An **Uncontrolled System Failure** means that the system has not detected a System Fault correctly or – having detected such a fault – has not executed a Controlled System Failure correctly, with the consequence that significant System Damage may be caused. The system may be in any mode other than a Fail-Silent Mode when an Uncontrolled System Failure occurs.

A **Controlled System Failure** means that – having correctly detected a System Fault – a reset is performed, after which the system enters a Normal Mode, or a Limp-Home Mode, or a Fail-Silent Mode.
A Controlled System Failure may proceed in stages. For example, after a System Fault is detected in a Normal Mode, the system may (after a system reset) re-enter the same Normal Mode; if another System Fault is detected within a pre-determined interval (e.g. 1 hour), the system may then enter a Limp-Home Mode. Depending on the nature of the fault, the sequence may vary: for example, the system may move immediately from a Normal Mode to a Fail-Silent Mode if a significant fault is detected. The system may be in any mode other than a Fail-Silent Mode when a Controlled System Failure occurs.

A **Normal Mode** means a pre-determined dynamic mode in which the system is fully operational and is meeting all of the expected system requirements, without causing System Damage. The system may support multiple Normal Modes.

A **Limp-Home Mode** means a pre-determined dynamic mode in which – while the system is not meeting all of the expected system requirements – a core subset of the system requirements is being met, and little or no System Damage is being caused. The system may support multiple Limp-Home Modes. In many cases, the system will enter a Limp-Home Mode on a temporary basis (for example, while attempts are made to bring a damaged road vehicle to rest in a location at the side of a motorway), before it enters a Fail-Silent Mode.

A **Fail-Silent Mode** means a pre-determined static mode in which the system has been shut down in such a way that it will cause little or no System Damage. The system will usually support only a single Fail-Silent Mode. In many cases, it is expected that intervention by a qualified individual (e.g. a Service Technician) may be required to re-start the system once it has entered a Fail-Silent Mode.

System Damage results from action by the system that is not in accordance with the system requirements. System Damage may involve loss of life or injury to users of the system, or to people in the vicinity of the system, or loss of life or injury to other animals. System Damage may involve direct or indirect financial losses. System Damage may involve a wider environmental impact (such as an oil spill). System Damage may involve more general damage (for example, through incorrect activation of a building sprinkler system).

A **System Fault** means a Hardware Fault and / or a Software Fault.

A **Software Fault** means a manifestation of a Software Error or a Deliberate Software Change.

A **Hardware Fault** means a manifestation of a Hardware Error, or a Deliberate Hardware Change, or the result of physical damage. Physical damage may arise – for example – from a broken connection, or from the impact of electromagnetic interference (EMI), radiation, vibration or humidity.

A **Deliberate Software Change** means an intentional change to the implementation of any part of the System Software that occurs as a result of a "computer virus" or any other form of malicious interference.

A **Software Error** means a mistake in the requirements, design, or implementation (that is, programming) of any part of the System Software.

A **Deliberate Hardware Change** means an intentional change to the implementation of any part of the System Hardware that occurs as a result of any form of malicious interference.

A **Hardware Error** means a mistake in the requirements, design, or implementation of any part of the System Hardware.

System Software means all of the software in the system, including tasks, scheduler, any support libraries and "startup" code.

System Hardware means all of the computing and related hardware in the system, including any processing devices (such as microcontrollers, microprocessors, FPGAs, DSPs and similar items), plus associated peripherals (e.g. memory components) and any devices under control of the computing devices (e.g. actuators), or providing information used by these devices (e.g. sensors, communication links).

Acronyms and abbreviations

ASIL	Automotive Safety Integrity Level
BCET	Best-Case Execution Time
CAN	Controller Area Network
CBD	Contract-Based Design
CLPD	Complex Programmable Logic Device
CMSIS	Cortex Microcontroller Software Interface Standard
COTS	Commercial 'Off The Shelf'
CPU	Central Processor Unit
DMA	Direct Memory Access
ECU	Electronic Control Unit
EMI	Electromagnetic Interference
ET	Event Triggered
FAP	Failure Assertion Programming
FFI	Freedom From Interference
FPGA	Field Programmable Gate Array
FS	Functional Safety
FSR	Functional Safety Requirement
MC	Mixed Criticality
MCU	Microcontroller (Unit)
MMU	Memory Management Unit
MPU	Memory Protection Unit
PTTES	Patterns for Time-Triggered Embedded Systems
RMA	Rate Monotonic Analysis
SIL	Safety Integrity Level
SoC	System on Chip
STA	Static Timing Analysis
TG	Task Guardian
TSIP	Task Sequence Initialisation Period
TT	Time Triggered
TTC	Time-Triggered Co-operative
TTH	Time-Triggered Hybrid
TTP	Time-Triggered Pre-emptive
TTRD	Time-Triggered Reference Design
WCET	Worst-Case Execution Time
WDT	Watchdog Timer

Reference designs[1]

TTRD02a	TTC scheduler with 'Heartbeat' fault reporting
TTRD02b	TTC scheduler with injected task overrun
TTRD02c	TTC scheduler (porting example)
TTRD03a	Simple framework for washing-machine controller
TTRD05a	Instrumented TTC scheduler (BCET and WCET)
TTRD05b	Instrumented TTC scheduler with task overrun
TTRD05c	Instrumented TTC scheduler (tick jitter)
TTRD05d	Instrumented TTC scheduler (task jitter)
TTRD05e	TTC Dry scheduler
TTRD06a	TTC "Super Loop" scheduler with hardware delay
TTRD06b	Implementing a "Sandwich Delay"
TTRD07a	TTC architecture: Nx3 operating modes
TTRD07b	TTC architecture: Nx3 & "Fail Silent"
TTRD07c	TTC architecture: Nx3, "Limp Home" & "Fail Silent"
TTRD08a	TTC-TC MPU scheduler
TTRD09a	TTC MoniTTor architecture (internal)
TTRD10a:	TTC MoniTTor-PredicTTor (generic)
TTRD10b:	TTC MoniTTor-PredicTTor (generic, async task set)
TTRD10c:	TTC MoniTTor-PredicTTor (TSR protected by MPU)
TTRD10d:	TTC MoniTTor-PredicTTor (TSR on external EEPROM)
TTRD11a:	TTH scheduler with "emergency stop"
TTRD11b:	TTH scheduler with long pre-empting tasks
TTRD11c:	TTP scheduler with example task set
TTRD11d:	TTP scheduler with BCET / WCET monitoring
TTRD12a:	Instrumented TTH scheduler (tick jitter)
TTRD12b:	TTH scheduler with reduced release jitter (idle task)
TTRD12c:	TTP scheduler with reduced release jitter (idle task)
TTRD12d:	TTP scheduler with MoniTTor and PredicTTor
TTRD13a:	TTC scheduler with temperature monitoring
TTRD14a:	System Platform TT01
TTRD14b:	System Platform TT02
TTRD14c:	System Platform TT03
TTRD15a:	Framework for washing-machine controller (TT01)
TTRD15b:	Create Tick List for TTRD15a (Normal Mode)
TTRD15c:	Create Tick List for TTRD15a (Limp-Home Mode)

[1] See: http://www.safetty.net/downloads/reference-designs

International standards and guidelines

Reference in text	Full reference
DO-178C	DO-178C: 2012
IEC 60335	IEC 60335-1:2010 + A1: 2013
IEC 60601	IEC 60601-1-8: 2006 + A1: 2012
IEC 60730	IEC 60730-1: 2013
IEC 61508	IEC 61508: 2010
IEC 62304	IEC 62304: 2006
ISO 26262	ISO 26262: 2011
MISRA C	MISRA C: 2012 (March 2013)

Preface

This book is concerned with the development of reliable, real-time embedded systems. The particular focus is on the engineering of systems based on time-triggered architectures.

In the remainder of this preface, I attempt to provide answers to questions that prospective readers may have about the book contents.

a. What is a "reliable embedded system"?

I have provided a definition of the phrase "System Fault" on Page xiii.

My goal in this book is to present a model-based process for the development of embedded applications that can be used to provide evidence that the system concerned will be able to detect such faults and then handle them in an appropriate manner, thereby avoiding Uncontrolled System Failures.

The end result is what I mean by a reliable embedded system.

b. Who needs reliable embedded systems?

Techniques for the development of reliable embedded systems are – clearly – of great concern in safety-critical markets (e.g. the automotive, medical, rail and aerospace industries), where Uncontrolled System Failures can have immediate, fatal, consequences.

The growing challenge of developing complicated embedded systems in traditional "safety" markets has been recognised, a fact that is reflected in the emergence in recent years of new (or updated) international standards and guidelines, including IEC 61508, ISO 26262 and DO-178C.

As products incorporating embedded processors continue to become ever more ubiquitous, safety concerns now have a great impact on developers working on devices that would not – at one time – have been thought to require a very formal design, implementation and test process. As a consequence, even development teams working on apparently "simple" household appliances now need to address safety concerns. For example, manufacturers need to ensure that the door of a washing machine cannot be opened by a child during a "spin" cycle, and must do all they can to avoid the risk of fires in "always on" applications, such as fridges and freezers. Again, recent standards have emerged in these sectors (such as IEC 60730).

Reliability is – of course – not all about safety (in any sector). Subject to inevitable cost constraints, most manufacturers wish to maximise the reliability of the products that they produce, in order to reduce the cost of warranty repairs, minimise product recalls and ensure repeat orders. As

systems grow more complicated, ensuring the reliability of embedded systems can present significant challenges for any organisation.

I have found that the techniques presented in this book can help developers (and development teams) in many sectors to produce reliable and secure systems.

c. Why work with "time-triggered" systems?

As noted at the start of this Preface, the focus of this book is on TT systems.

Implementation of a TT system will typically start with a single interrupt that is linked to the periodic overflow of a timer. This interrupt may drive a task scheduler (a simple form of "operating system"). The scheduler will – in turn – release the system tasks at predetermined points in time.

TT can be viewed as a subset of a more general event-triggered (ET) architecture. Implementation of a system with an ET architecture will typically involve use of multiple interrupts, each associated with specific periodic events (such as timer overflows) or aperiodic events (such as the arrival of messages over a communication bus at unknown points in time).

TT approaches provide an effective foundation for reliable real-time systems because – during development and after construction – it is (compared with equivalent ET designs) easy to model the system and, thereby, determine whether all of the key timing requirements have been met. This can help to reduce testing costs – and reduce business risks.

The deterministic behaviour of TT systems also offers very significant advantages at run time, because – since we know precisely what the system should be doing at a given point in time – we can very quickly determine whether it is doing something wrong.

d. How does this book relate to international safety standards?

Throughout this book it is assumed that some readers will be developing embedded systems in compliance with one or more international standards.

The standards discussed during this book include following:

- IEC 61508 (industrial systems / generic standard)
- ISO 26262 (automotive systems)
- IEC 60730 (household goods)
- IEC 62304 (medical systems)
- DO-178C (aircraft)

No prior knowledge of any of these standards is required in order to read this book.

Please note that full references to these standards are given on p.xix.

e. What programming language is used?

The software in this book is implemented almost entirely in 'C'.

For developers using C, the "MISRA C" guidelines are widely employed as a "language subset", with associated coding guidelines (MISRA, 2012).

f. Is the source code "freeware"?

This book is supported by a complete set of "Time-Triggered Reference Designs" (TTRDs).

Both the TTRDs and this book describe patented[2] technology and are subject to copyright and other restrictions.

The TTRDs provided with this book may be used without charge: [i] by universities and colleges in courses for which a degree up to and including "MSc" level (or equivalent) is awarded; [ii] for non-commercial projects carried out by individuals and hobbyists.

All other use of any of the TTRDs associated with this book requires purchase (and maintenance) of a low-cost, royalty free ReliabiliTTy Technology Licence:

http://www.safetty.net/products/reliabilitty

g. How does this book relate to other books in the "ERES" series?

The focus throughout all of the books in the ERES series is on single-program, real-time systems.

Typical applications for the techniques described in this series include control systems for aircraft, steer-by-wire systems for passenger cars, patient monitoring devices in a hospital environment, electronic door locks on railway carriages, and controllers for domestic "white goods".

Such systems are currently implemented using a wide range of different hardware "targets", including various different microcontroller families (some with a single core, some with multiple independent cores, some with "lockstep" architectures) and various FPGA / CPLD platforms (with or without a "soft" or "hard" processor core).

Given the significant differences between the various platforms available and the fact that most individual developers (and many organisations) tend to work in a specific sector, using a limited range of hardware, I decided that I would simply "muddy the water" by trying to cover numerous microcontroller

[2] Patents applied for.

families in a single version of this book. Instead, ERES will be released in a number of distinct editions, each with a focus on a particular (or small number) of hardware targets and related application sectors.

You'll find up-to-date information about the complete book series here: http://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems

h. What processor hardware is used in this book?

In this edition of the book, the main processor target is an NXP® LPC1769 microcontroller. In almost all cases, the code examples can be executed on a low-cost and readily-available evaluation platform.[3]

The LPC1769 is an ARM® Cortex-M3 based microcontroller (MCU) that operates at CPU frequencies of up to 120 MHz.

The LPC1769 is intended for use in applications such as: industrial networking; motor control; white goods; eMetering; alarm systems; and lighting control.

The two case studies in this book focus on the use of the LPC1769 microcontroller in white goods (specifically, a washing machine). However, the TT software architecture that is employed in these examples is generic in nature and can be employed in many different systems (in various sectors).

Many of the examples employ key LPC1769 components – such as the Memory Protection Unit – in order to improve system reliability and safety.

i. How does this book relate to "PTTES"?

This book is not intended as an introductory text: it is assumed that readers already have experience developing embedded systems, and that they have some understanding of the concept of time-triggered systems. My previous book "Patterns for Time-Triggered Embedded Systems" (PTTES) can be used to provide background reading.[4]

It is perhaps worth noting that I completed work on PTTES around 15 years ago. Since then, I estimate that I've worked on or advised on more than 200 'TT' projects, and helped around 50 companies to make use of a TT approach for the first time. I've learned a great deal during this process. In the present book, I've done my best to encapsulate my experience (to date) in the development of reliable, real-time embedded systems.

[3] Further information about this hardware platform is presented in Appendix 2.

[4] "PTTES" can be downloaded here: http://www.safetty.net/publications/pttes

j. Is there anyone that you'd like to thank?

As with my previous books, I'd like to use this platform to say a public "thank you" to a number of people.

In total, I spent 21 years in the Engineering Department at the University of Leicester (UoL) before leaving to set up SafeTTy Systems. I'd like to thank the following people for their friendship and support over the years: Fernando Schlindwein, John Fothergill, Len Dissado, Maureen Strange, Barrie Jones, Ian Postlethwaite, Andrew Norman, Simon Hogg, Simon Gill, John Beynon, Hans Bleis, Pete Barwell, Chris Marlow, Chris Edwards, Julie Hage, Matt Turner, Bill Manners, Paul Lefley, Alan Stocker, Barry Chester, Michelle Pryce, Tony Forryan, Tom Robotham, Geoff Folkard, Declan Bates, Tim Pearce, Will Peasgood, Ian Jarvis, Dan Walker, Hong Dong, Sarah Hainsworth, Paul Gostelow, Sarah Spurgeon, Andy Truman, Alan Wale, Alan Cocks, Lesley Dexter, Dave Siddle, Guido Herrmann, Andy Chorley, Surjit Kaur, Julie Clayton, Andy Willby, Dave Dryden and Phil Brown.

In the Embedded Systems Laboratory (at the University of Leicester), I had the opportunity to work with an exceptional research team. I'd particularly like to thank Devaraj Ayavoo, Keith Athaide, Zemian Hughes, Pete Vidler, Farah Lakhani, Aley Imran Rizvi, Susan Kurian, Musharraf Hanif, Kam Chan, Ioannis Kyriakopoulos, Michael Short and Imran Sheikh, many of whom I worked with for many years (both in the ESL and at TTE Systems). I also enjoyed having the opportunity to work with Tanya Vladimirova, Royan Ong, Teera Phatrapornnant, Chisanga Mwelwa, Ayman Gendy, Huiyan Wang, Muhammad Amir, Adi Maaita, Tim Edwards, Ricardo Bautista-Quintero, Douglas Mearns, Yuhua Li, Noor Azurati Ahmad, Mouaaz Nahas, Chinmay Parikh, Kien Seng Wong, David Sewell, Jianzhong Fang and Qiang Huang.

In 2005, I was asked by staff in what became the "Enterprise and Business Development Office" (at the University of Leicester) to begin the process that led to the formation of TTE Systems Ltd. Tim Maskell was there from the start, and it was always a great pleasure working with him. I also enjoyed working with David Ward, Bill Brammar and James Hunt.

The "TTE" team involved a number of my former research colleagues, and I also had the pleasure of working with Muhammad Waqas Raza, Anjali Das, Adam Rizal Azwar, Rishi Balasingham, Irfan Mir, Rajas More and Vasudevan Pillaiand Balu. At this time, I also enjoyed having the opportunity to work with my first team of Board members and investors: I'd particularly like to thank Alan Lamb, Clive Smith, Penny Attridge, Jonathan Gee,

Tim Maskell (again), Viv Hallam, Chris Jones and Ederyn Williams for their support over the lifetime of the company.

Since the start of 2014, I've been focused on getting SafeTTy Systems off the ground. Steve Thompson and Farah Lakhani joined me at the start of this new project and it has been a pleasure to have the opportunity to work with them again.

I'm grateful to Cass and Kynall (for being there when I have needed them – I hope to return the favour before too long), and to Bruce and Biggles (for keeping my weight down). I'd like to thank David Bowie for "The Next Day", Thom Yorke and Radiohead for "Kid A", and Sigur Rós for "()".

Last but not least, I'd like to thank Sarah for having faith in me in the last two years, as I took our lives "off piste".

Michael J. Pont
January 2015

PART ONE: INTRODUCTION

"If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with."

Edsger W. Dijkstra, 1972.

CHAPTER 1: Introduction

In this chapter we provide an overview of the material that is covered in detail in the remainder of this book.

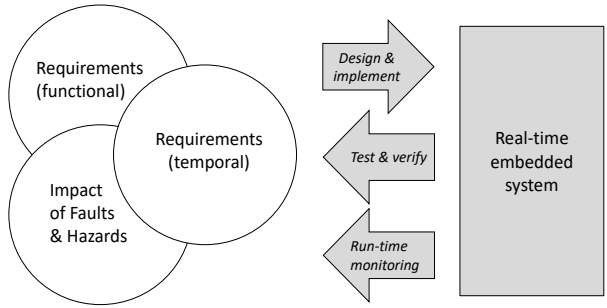


Figure 1: The engineering of reliable real-time embedded systems (overview). In this book, our focus will be on the stages shown on the right of the figure (grey boxes).

1.1. Introduction

The process of engineering reliable real-time embedded systems (the focus of this book) is summarised schematically in Figure 1. Projects will typically begin by recording both the functional requirements and the timing requirements for the system, and by considering potential faults and hazards. Design and implementation processes will then follow, during and after which test and verification activities will be carried out (in order to confirm that the requirements have been met in full). Run-time monitoring will then be performed as the system operates.

The particular focus of this book is on the development of this type of system using time-triggered (TT) architectures.

What distinguishes TT approaches is that it is possible to model the system behaviour precisely and – therefore – determine whether all of the timing requirements have been met. It is important to appreciate that we can use our models to confirm that the system behaviour is correct both during development and at run time. This can provide a very high level of confidence that the system will either: [i] operate precisely as required; or [ii] move into a pre-determined Limp-Home Mode or Fail-Silent Mode.

In this chapter, we explain what a time-triggered architecture is, and we consider some of the processes involved in developing such systems: these processes will then be explored in detail in the remainder of the text.

1.2. Single-program, real-time embedded systems

An embedded computer system ("embedded system") is usually based on one or more processors (for example, microcontrollers or microprocessors), and some software that will execute on such processor(s).

Embedded systems are widely used in a variety of applications ranging from brake controllers in passenger vehicles to multi-function mobile telephones.

The focus in this text is on what are sometimes called "single-program" embedded systems. Such applications are represented by systems such as engine controllers for aircraft, steer-by-wire systems for passenger cars, patient monitoring devices in a hospital environment, automated door locks on railway carriages, and controllers for domestic washing machines.

The above systems have the label "single-program" because the general user is not able to change the software on the system (in the way that programs are installed on a laptop, or "apps" are added to a smartphone): instead, any changes to the software in the steering system – for example – will be performed as part of a service operation, by suitably-qualified individuals.

What also distinguishes the systems above (and those discussed throughout this book) is that they have real-time characteristics.

Consider, for example, the greatly simplified aircraft autopilot application illustrated schematically in Figure 2. Here we assume that the pilot has entered the required course heading, and that the system must make regular and frequent changes to the rudder, elevator, aileron and engine settings (for example) in order to keep the aircraft following this path.

An important characteristic of this system is the need to process inputs and generate outputs at pre-determined time intervals, on a time scale measured in milliseconds. In this case, even a slight delay in making changes to the rudder setting (for example) may cause the plane to oscillate very unpleasantly or, in extreme circumstances, even to crash.

In order to be able to justify the use of the aircraft system in practice (and to have the autopilot system certified), it is not enough simply to ensure that the processing is 'as fast as we can make it': in this situation, as in many other real-time applications, the key characteristic is *deterministic* processing. What this means is that in many real-time systems we need to be able to *guarantee* that a particular activity will always be completed within – say – 2 ms (+/- 5 μs), or at 6 ms intervals (+/- 1 μs): if the processing does not match this specification, then the application is not just slower than we would like, it is simply not fit for purpose.

Reminder

1 second (s)	= 1.0 second (100 seconds) = 1000 ms.
1 millisecond (ms)	= 0.001 seconds (10-3 seconds) = 1000 µs.
1 microsecond (µs)	= 0.000001 seconds (10-6 seconds) = 1000 ns.
1 nanosecond (ns)	= 0.000000001 seconds (10-9 seconds).

Box 1

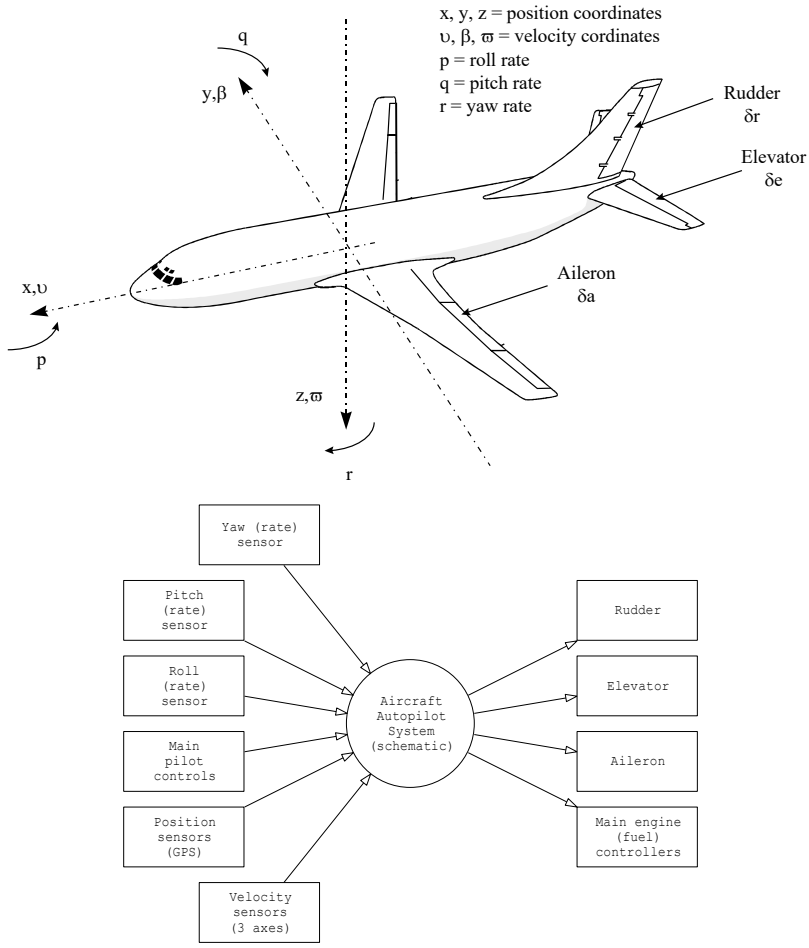


Figure 2: A high-level schematic view of an autopilot system.

Tom De Marco has provided a graphic description of this form of hard real-time requirement in practice, quoting the words of a manager on a software project:

> "We build systems that reside in a small telemetry computer, equipped with all kinds of sensors to measure electromagnetic fields and changes in temperature, sound and physical disturbance. We analyze these signals and transmit the results back to a remote computer over a wide-band channel. Our computer is at one end of a one-meter long bar and at the other end is a nuclear device. We drop them together down a big hole in the ground and when the device detonates, our computer collects data on the leading edge of the blast. The first two-and-a-quarter milliseconds after detonation are the most interesting. Of course, long before millisecond three, things have gone down hill badly for our little computer. We think of that as a real-time constraint."
>
> [De Marco, writing in the foreword to Hatley and Pirbhai, 1987]

In this case, it is clear that this real-time system must complete its recording on time: it has no opportunity for a "second try". This is an extreme example of what is sometimes referred to as a 'hard' real-time system.

1.3. TT vs. ET architectures

When creating a single-program design, developers must choose an appropriate system architecture. One such architecture is a "time-triggered" (TT) architecture. Implementation of a TT architecture will typically involve use of a single interrupt that is linked to the periodic overflow of a timer. This interrupt will be used to drive a task scheduler (a simple form of "operating system"). The scheduler will – in turn – release the system tasks at predetermined points in time.

TT architectures can be viewed as a "safer subset" of a more general event-triggered architecture (see Figure 3 and Figure 4). Implementation of a system with an event-triggered architecture will typically involve use of multiple interrupts, each associated with specific periodic events (such as timer overflows) and aperiodic events (such as the arrival of messages over a communication bus at random points in time). ET designs are traditionally associated with the use of what is known as a real-time operating system (or RTOS), though use of such a software platform is not a defining characteristic of an ET architecture.

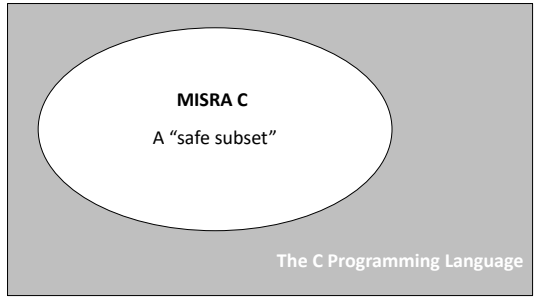


Figure 3: Safer language subsets (for example, MISRA C) are employed by many organisations in order to improve system reliability. See MISRA (2012).

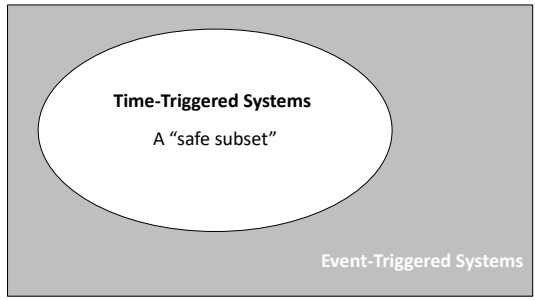


Figure 4: In a manner similar to MISRA C (Figure 3), TT approaches provide a “safer subset” of ET designs, at the system architecture level.

Whether TT or ET architectures are employed, the system tasks are typically named blocks of program code that perform a particular activity (for example, a task may check to see if a switch has been pressed): tasks are often implemented as functions in programming languages such as ‘C’ (and this is the approached followed in the present book).

It should be noted that – at the time of writing (2014) – the use of ET architectures and RTOS solutions is significantly more common than the use of TT solutions, at least in projects that are not safety related.

1.4. Modelling system timing characteristics

TT computer systems execute tasks according to a predetermined task schedule. As noted in Section 1.3, TT systems are typically (but not necessarily) implemented using a design based on a single interrupt linked to the periodic overflow of a timer.

For example, Figure 5 shows a set of tasks (in this case Task A, Task B, Task C and Task D) that might be executed by a TT computer system according to a predetermined task schedule.

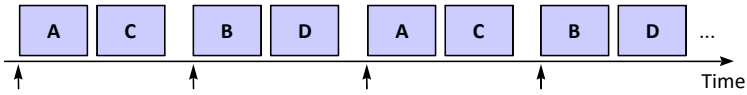


Figure 5: A set of tasks being released according to a pre-determined schedule.

In Figure 5, the release of each sub-group of tasks (for example, Task A and Task B) is triggered by what is usually called a timer "tick". In most designs (including all of those discussed in detail in this book), the timer tick is implemented by means of a timer interrupt. These timer ticks are periodic. In an aerospace application, the "tick interval" (that is, the time interval between timer ticks) of 25 ms might be used, but shorter tick intervals (e.g. 1 ms or 100 μs) are more common in other systems.

In Figure 5, the task sequence executed by the computer system is as follows: Task A, Task C, Task B, Task D. In many designs, such a task sequence will be determined at design time (to meet the system requirements) and will be repeated "forever" when the system runs (until the system is halted or powered down, or a System Failure occurs).

Sometimes it is helpful (not least during the design process) to think of this task sequence as a "Tick List": such a list lays out the sequence of tasks that will run after each system tick.

For example, the Tick List corresponding to the task set shown in Figure 5 could be represented as follows:

```
[Tick 0]
Task A
Task C
[Tick 1]
Task B
Task D
```

Once the system reaches the end of the Tick List, it starts again at the beginning.

In Figure 5, the tasks are co-operative (or "non-pre-emptive") in nature: each task must complete before another task can execute. The design shown in these figures can be described as "time triggered co-operative" (TTC) in nature.

We say more about designs that involve task pre-emption in Section 1.6.

The importance of Tick Lists

The creation and use of Tick Lists is central to the engineering of reliable TT systems.

Through the use of this simple model, we can determine key system characteristics – such as response times, task jitter levels and maximum CPU loading – very early in the design process.

We can then continue to check these characteristics throughout the development process, and during run-time operation of the system.

We will consider the creation and use of Tick Lists in detail in Chapter 4.

Box 2

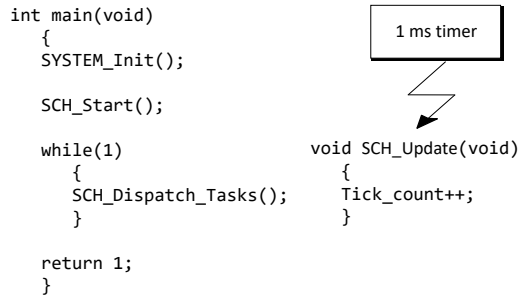


Figure 6: A schematic representation of the key components in a TTC scheduler.

1.5. Working with "TTC" schedulers

Many (but by no means all) TT designs are implemented using co-operative tasks and a "TTC" scheduler.

Figure 6 shows a schematic representation of the key components in such a scheduler. First, there is function SCH_Update(): in this example, this is linked to a timer that is assumed to generate periodic "ticks" – that is, timer interrupts – every millisecond.

The SCH_Update() function is responsible for keeping track of elapsed time.

Within the function main() we assume that there are functions to initialise the scheduler, initialise the tasks and then add the tasks to the schedule.

In Figure 6, function main(), the process of releasing the system tasks is carried out in the function SCH_Dispatch_Tasks().

The operation of a typical SCH_Dispatch() function is illustrated schematically in Figure 7. In this figure, the dispatcher begins by determining whether there is any task that is currently due to run. If the answer to this question is "yes", the dispatcher runs the task.

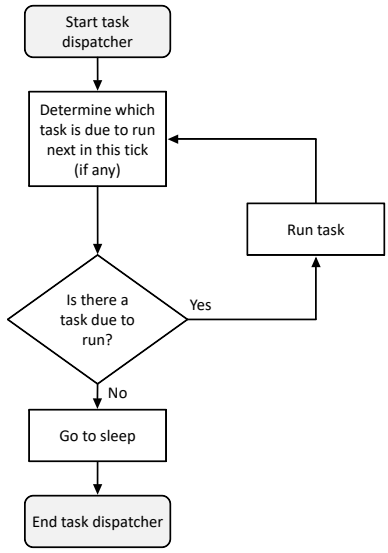


Figure 7: The operation of a task dispatcher.

The dispatcher repeats this process until there are no tasks remaining that are due to run. The dispatcher then puts the processor to sleep: that is, it places the processor into a power-saving mode. The processor will remain in this power-saving mode until awakened by the next timer interrupt: at this point the timer ISR (Figure 6) will be called again, followed by the next call to the dispatcher function (Figure 7).

It should be noted that there is a deliberate split between the process of timer updates and the process of task dispatching (shown in main() in Figure 6 and described in more detail in Figure 7). This split means that it is possible for the scheduler to execute tasks that are longer than one tick interval without missing timer ticks. This gives greater flexibility in the system design, by allowing use of a short tick interval (which can make the system more responsive) and longer tasks (which can simplify the design process). This split may also help to make the system more robust in the event of run-time faults: we say more about this in Chapter 2.

Flexibility in the design process and the ability to recover from transient faults are two reasons why "dynamic" TT designs (with a separate timer ISR and task dispatch functions) are generally preferred over simpler designs in which tasks are dispatched from the timer ISR.

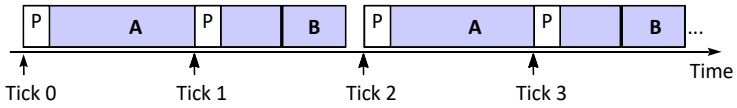


Figure 8: Executing tasks using a TTH scheduler. See text for details.

1.6. Supporting task pre-emption

The designs discussed in Section 1.4 and Section 1.5 involve co-operative tasks: this means that each task "runs to completion" after it has been released. In many TT designs, high-priority tasks can interrupt (pre-empt) lower-priority tasks.

For example, Figure 8 shows a set of three tasks: Task A (a low-priority, co-operative task), Task B (another low-priority, co-operative task), and Task P (a higher-priority pre-empting task). In this example, the lower-priority tasks may be pre-empted periodically by the higher-priority task. More generally, this kind of "time triggered hybrid" (TTH) design may involve multiple co-operative tasks (all with an equal priority) and one or more pre-empting tasks (of higher priority).

We can also create "time-triggered pre-emptive" (TTP) schedulers: these support multiple levels of task priority.

We can – of course – record the Tick List for TTH and TTP designs. For example, the task sequence for Figure 8 could be listed as follows: Task P, Task A, Task P, Task B, Task P, Task A, Task P, Task B.

We say more about task pre-emption in Part Three.

1.7. Different system modes

Almost all practical embedded systems have at least two system modes, called something like "Normal mode" and "Fault mode". However most have additional system modes. For example, Figure 9 shows a schematic representation of the software architecture for an aircraft system with system modes corresponding to the different flight stages (preparing for takeoff, climbing to cruising height, etc).

In this book, we consider that the system mode has changed if the task set has changed. It should therefore be clear that we are likely to have a different Tick List for each system mode.

Please note that – even in a TT design – the timing of the transition between system modes is not generally known in advance (because, for example, the time taken for the plane shown in Figure 9 to reach cruising height will vary with weather conditions), but this does not alter the development process.

The key feature of all TT designs is that – whatever the mode – the tasks are always released according to a schedule that is determined, validated and verified when the system is designed.

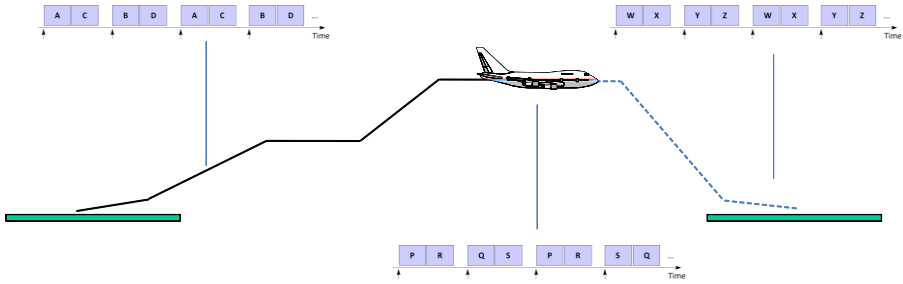


Figure 9: An example of a system with multiple operating modes.

We say more about system modes in Chapter 7.

1.8. A "Model-Build-Monitor" methodology

A three-stage development process is described in detail during the course of this book.

The first stage involves modelling the system (using one or more Tick Lists, depending on the number of modes), as outlined in Section 1.4. The second stage involves building the system (for example, using a simple TTC scheduler, as outlined in Section 1.5). The third stage involves adding support for run-time monitoring.

Run-time monitoring is essential because we need to ensure that the computer system functions correctly in the event that Hardware Faults occur (as a result, for example, of electromagnetic interference, or physical damage: see "Definitions" on Page xiii). In addition, as designs become larger, it becomes unrealistic to assume that there will not be residual Software Errors in products that have been released into the field: it is clearly important that there should not be an Uncontrolled System Failure in the event that such errors are present. Beyond issues with possible Hardware Faults and residual errors in complex software, we may also need to be concerned about the possibility that attempts could be made to introduce Deliberate Software Changes into the system, by means of "computer viruses" and similar attacks.

The approach to run-time monitoring discussed in this book involves checking for resource-related faults and / or time-related faults (Figure 10).

As an example of resource-related fault, assume that Pin 1-23 on our microcontroller is intended to be used exclusively by Task 45 to activate the steering-column lock in a passenger vehicle. This lock is intended to be engaged (to secure the vehicle against theft) only after the driver has parked and left the vehicle.

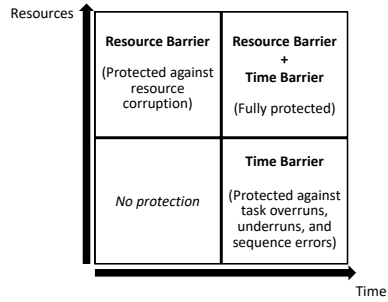


Figure 10: Run-time monitoring is employed to guard against Uncontrolled System Failures. In this book, we will check for faults that are evidenced by incorrect CPU usage (task overruns, task underruns and task sequencing) as well as faults related to other hardware resources (such as CAN controllers, analogue-to-digital convertors, port pins and memory).

A (potentially very serious) resource-related fault would occur if Pin 1-23 was to be activated by another task in the system while the vehicle was moving at high speed.

Of course, both ET and TT designs need to employ mechanisms to check for resource-related faults. However, this process is much more easily modelled (and therefore more deterministic) in TT designs. For instance, in "TTC" designs, precise control of access to shared resources is intrinsic to the architecture (because all tasks "run to completion"). Even in TT designs that involve task pre-emption, controlling access to shared resources is much more straightforward than in ET designs. One very important consequence is that while the impact of priority inversion (PI) can be ameliorated in ET designs through the use of mechanisms such as "ceiling protocols" (as we will discuss in Chapter 12), PI problems can be eliminated only through the use of a TT solution.

Beyond this, we have found that a design approach based on the concept of "Task Contracts" can help developers to implement effective "Resource Barriers" in TT systems. We say more about this approach in Chapter 8.

In addition to resource-related faults, we also need to consider timing related faults (please refer again to Figure 10). Here, a second – very significant – advantage of TT designs comes into play.

As we have seen in this chapter, TT systems are – by definition – designed to execute sets of tasks according to one or more pre-determined schedules: in each system mode, the required sequence of tasks is known in advance. During the design process, the task schedules are carefully reviewed and assessed against the system requirements, and at run time a simple task scheduler is used to release the tasks at the correct times. If the task set is not then executed in the correct sequence at run time, this may be symptomatic of a serious fault. If such a situation is not detected quickly, this may have

severe consequences for users of the system, or those in the vicinity of the system.

For example, consider that we have detected a fault in the braking system of a passenger car: if the driver is already applying the brakes in an emergency situation when we detect the fault, the fault-detection mechanism is of little value. Similarly, late detection (or lack of detection) of faults in aerospace systems, industrial systems, defence systems, medical systems, financial systems or even household goods may also result in injury, loss of human life and / or very significant financial losses.

Using the techniques presented in Chapter 10, we can perform "predictive monitoring" of the task execution sequence during the system operation. In many cases, this means that we can detect that the system is about to run an incorrect task before this task even begins executing.

This type of solution can greatly simplify the process of achieving compliance with international standards and guidelines. For example, to achieve compliance with the influential IEC 61508 standard, many designs require the use of a "diverse monitor" unit. Such a unit is intended to prevent the system from entering an unsafe state[5], which is precisely what we can achieve using predictive monitoring of a TT architecture.

Similar requirements arise from other standards (for example, the need to implement "Safety Mechanisms" in ISO 26262).

1.9. How can we avoid Uncontrolled System Failures?

As we conclude this introductory chapter, we'll consider one of the key challenges facing developers of modern embedded systems.

As we have discussed in previous sections, embedded systems typically consist of: [i] a set of tasks; [ii] a scheduler, operating system or similar software framework that will have some control over the release of the tasks.

In an ideal world, the resulting architecture might look something like that illustrated in Figure 11 (left). From the developer's perspective, such a design may be attractive, because each software component is isolated: this – for example – makes run-time monitoring straightforward, and means that it is easy to add further tasks to the system (for example, during development or later system upgrades) with minimal impact on the existing tasks.

[5] It is sometimes assumed that a "diverse monitor" is intended to detect when a system has entered into an unsafe state, but that is not what is required in IEC 61508 [2010].

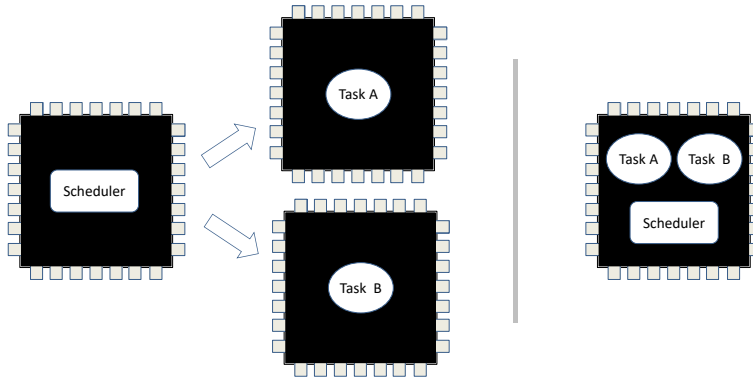


Figure 11: An "ideal" implementation of an embedded system with two tasks (left) along with a more practical implementation (right).

In practice, such a "one processor per task" design would prove to be impossibly expensive in most sectors. Instead, we are likely to have multiple tasks and the scheduler / operating system sharing the same processor (Figure 11, right).

In this real-world design, a key design aim will be to ensure that no task can interfere with any other task, and that no task can interfere with the scheduler: this is sometimes described as a goal of "Freedom From Interference" (FFI).

FFI is a very worthy goal. Unfortunately, in any non-trivial embedded system, there are a great many ways in which it is possible for tasks (and other software components) that share a CPU, memory and other resources to interact. As a consequence, any claim that we can prevent any interference would be likely to be met with some scepticism.

This does not mean that we need to dismiss FFI as "unachievable", because – while interference may not be preventable – it may be detectable.

More specifically, we will argue in this book that – through use of an appropriate implementation of a TT design, with a matched monitoring system – we will often be able to meet FFI requirements. We can do this because the engineering approach described in the following chapters can be used to: [i] provide evidence of the circumstances in which we will be able to detect any interference between tasks, or between tasks and the scheduler, in a given system; and [ii] provide evidence of our ability to move the system into Limp-Home Mode or Fail-Silent Mode in the event that such interference is detected.

Overall, the goal – in this FFI example and with all of the systems that we consider in this book – is to prevent Uncontrolled System Failures. We will do this by building upon a TT system platform, such as that illustrated schematically in Figure 12.

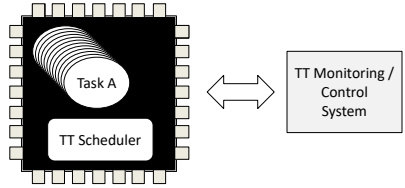


Figure 12: A schematic representation of one of the platforms that we will use in this book in order to avoid Uncontrolled System Failures.

1.10. Conclusions

In this introductory chapter, we've provided an overview of the material that is covered in detail in the remainder of this book.

In Chapter 2, we will introduce our first task scheduler.

CHAPTER 2: Creating a simple TTC scheduler

In Chapter 1, we noted that the implementation of most time-triggered embedded systems involves the use of a task scheduler. In this chapter, we explore the design of a first simple scheduler for use with sets of periodic, co-operative tasks.

Related TTRDs

A list of the TTRDs discussed in this chapter is included below.

C programming language (LPC1769 target):

- TTRD02a: TTC scheduler with WDT support and 'Heartbeat' fault reporting
- TTRD02b: TTC scheduler with injected task overrun
- TTRD02c: TTC scheduler (porting example)

2.1. Introduction

In this chapter, we'll start by exploring the TTC scheduler "TTRD02a". To introduce TTRD02a, we will present a simple "Heartbeat" example in which the scheduler is used to flash an LED with a 50% duty cycle and a flash rate of 0.5 Hz: that is, the LED will be "on" for 1 second, then "off" for one second, then "on" for one second, and so on (Figure 13).

Figure 14 provides an overview of the structure and use of this scheduler.

If you have previously used one of the schedulers described in "Patterns for Time-Triggered Embedded Systems" (Pont, 2001), then much of the material presented in this chapter will be familiar. However, there are some important differences between the PTTES schedulers and those presented in this chapter. The main differences arise as a result of the new system foundation: this is illustrated schematically in Figure 13.

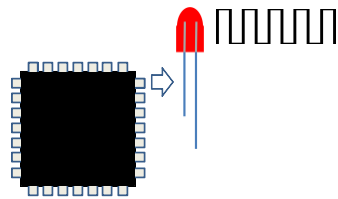


Figure 13: A schematic representation of a microcontroller running a TTC scheduler and executing a "Heartbeat" task.

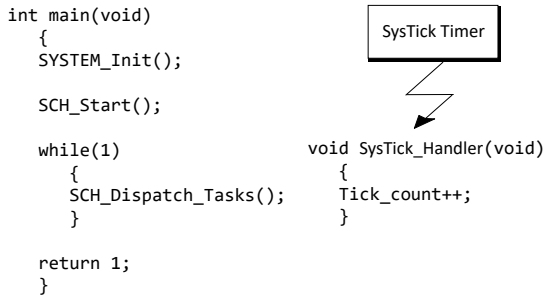


Figure 14: An overview of the structure and use of a TTC scheduler.

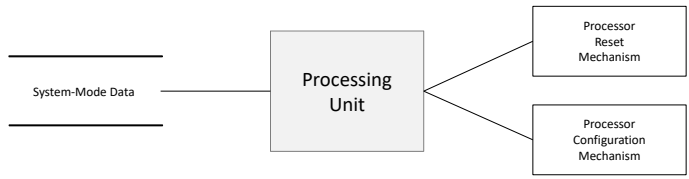


Figure 15: The foundation of the scheduler designs presented in this book.

Perhaps the most immediately obvious differences between the PTTES schedulers and those described here can be seen in the way that the SCH_Update() function and SCH_Dispatch() function are structured: please refer to Section 2.5 and Section 2.7, respectively, for further details.

There is also a significant difference in the recommended initialisation process for the schedulers presented in this book when compared with the approach presented in PTTES. More specifically, we now recommend that the system initialisation process is split between function main() and function SYSTEM_Init().

An example of a suitable implementation of function main() is shown in Figure 14.

We will consider the scheduling functions that are called in main – SCH_Start() and SCH_Dispatch_Tasks() – shortly. In this section, we will focus on the system initialisation function (see Listing 7, on p.48)[6].

The first point to note about the initialisation process illustrated in this chapter is that the system has two operating modes: "FAIL_SILENT" and "NORMAL". In this example, any reset that is caused by the watchdog timer (WDT) causes the system to enter the FAIL_SILENT mode, while a normal power-on reset (and any other reset events) cause the system to enter NORMAL mode.

[6] Throughout this book, longer code listings are located at the end of each chapter.

In FAIL_SILENT mode, the system simply "halts" (Code Fragment 1).[7]

```
case FAIL_SILENT:
   {
   // Reset caused by WDT
   // Trigger "fail silent" behaviour
   SYSTEM_Perform_Safe_Shutdown();
```

Code Fragment 1: Entering "Fail_Silent" mode.

There really isn't very much more that we can do in this mode in the Heartbeat demo, but – in a real system design – this should be where we end up if a serious fault has been detected by the system (and no other way of handling this fault has been identified). Deciding what to do in these circumstances requires careful consideration during the system development process.[8]

As a starting point, we need to consider what to do with the system port pins in these circumstances. Our general goal is to ensure that the pins are left in a state where they can do minimum damage. For example, it may be possible to turn off any dangerous equipment that is under the control of the computer system by setting appropriate levels on the port pins.

Other options may also need to be considered. For example, if the computer system is connected to other devices or equipment over a computer network (wired or wireless) we may wish to try and send out a message to the other components to indicate that the computer system has failed.

When the system reset is not caused by the WDT then – in this example – we enter NORMAL mode.[9]

In this mode, we need to do the following to initialise the system:

- set up the WDT, and associated WATCHDOG_Update() task;
- set up the scheduler;
- call the initialisation functions for all other tasks; and,
- add the tasks to the schedule.

In our example, we first set up the watchdog timer.

---

7 You will find the code for the function SYSTEM_Perform_Safe_Shutdown() in Listing 7 (on p.46).

8 We will consider this matter in detail in Chapter 13.

9 In many system designs, there will be multiple operating modes. We consider how such designs can be implemented in Chapter 7.

When used as described in this chapter, a WDT is intended to force a processor reset (and – thereby – place the system into a FAIL_SILENT mode) under the following circumstances:

- when the system becomes overloaded, usually as a result of one or more tasks exceeding their predicted "worst-case execution time" (WCET): this is a **task overrun** situation; or,
- when the system fails to release the WATCHDOG_Update() task according to the pre-determined schedule for any other reason.

In TTC designs where it has been determined that – for every tick in the hyperperiod – the sum of the task execution times of the tasks that are scheduled to run is less than the tick interval, then a WDT timeout period very slightly larger than the tick interval is often used.

Our example design comfortably meets the above execution-time criteria and has a tick interval of 1 ms: we therefore set the watchdog timeout to just over 1 ms, as follows:

```
// Set up WDT (timeout in *microseconds*)
WATCHDOG_Init(1100);
```

We'll look at the details of the WDT configuration in Section 2.11.

Following configuration of this timer, we then set up the scheduler with 1 ms ticks, using the SCH_Init() function:

```
// Set up scheduler for 1 ms ticks (tick interval *milliseconds*)
SCH_Init(1);
```

Note that if the system cannot be configured with the required tick interval, we force a system reset (using the WDT unit): following the reset, the system will then enter a FAIL_SILENT mode. This type of WDT-induced mode change will be common in many of the designs that we consider in this book (in various different circumstances).

We will provide further information about the SCH_Init() function in Section 2.4.

Assuming that initialisation of the scheduler was successful, we then prepare for the Heartbeat task, by means of the HEARTBEAT_Init() function.

Further information is provided about the Heartbeat task in Section 2.13: for now, we will simply assume that this is used to configure an I/O pin that has been connected to LED2 on the LPC1769 board (please refer to Appendix 1 for details).

SCH_MAX_TASKS

You will find SCH_MAX_TASKS in "Scheduler Header" file in all designs in this book.

This constant must be set to a value that is at least as large as the number of tasks that are added to the schedule: this process is not automatic and must be checked for each project.

Box 3

2.2. A first TTC scheduler (TTRD02a)

Having considered, in outline, how the system will be initialised, we now consider the internal structure and operation of the scheduler itself.

The TTRD02a scheduler presented in this chapter is made up of the following key components:

- A scheduler data structure.
- An initialisation function.
- An interrupt service routine (ISR), used to update the scheduler.
- A function for adding tasks to the schedule.
- A dispatcher function that releases tasks when they are due to run.

We consider each of the required components in the sections that follow.

2.3. The scheduler data structure and task array

At the heart of TTRD02a is a user-defined data type (sTask) that collects together the information required about each task.

Listing 4 shows the sTask implementation used in TTRD02a.

The task list is then defined in the main scheduler file as follows:

```
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

The members of sTask are documented in Table 1.

2.4. The 'Init' function

The scheduler initialisation function is responsible for:

- initialising the global fault variable;
- initialising the task array; and,
- configuring the scheduler tick.

The initialisation process begins as shown in Code Fragment 2.

```
// Reset the global fault variable
Fault_code_G = 0;

for (i = 0; i < SCH_MAX_TASKS; i++)
   {
   SCH_tasks_G[i].pTask = 0;
   }
```

Code Fragment 2: The start of the scheduler initialisation process.

In a manner similar to the PTTES schedulers, a global variable (Fault_code_G) is used to report fault codes, usually via the Heartbeat task: please see Section 2.13 for further information about this.

The fault codes themselves can be found in the Project Header file (main.h, Listing 1).

The next step in the scheduler initialisation process involves setting up the timer ticks.

In TTRD02a, this code is based on the ARM CMSIS[10].

Table 1: The members of the sTask data structure (as used in TTRD02a).

Member	Description
`void (*pTask)(void)`	A pointer to the task that is to be scheduled. The task must be implemented as a "void void" function. See Section 2.13 for an example.
`uint32_t Delay`	The time (in ticks) before the task will next execute.
`uint32_t Period`	The task period (in ticks).
`uint32_t WCET`	The worst-case execution time for the task (in µs). Please note that this information is not used directly in TTRD02a, but is employed during schedulability analysis (see Chapter 4). In addition, in later schedulers in this book, this information is used to assist in the detection of run-time faults (see Chapter 9 and Chapter 11).
`uint32_t BCET`	The best-case execution time for the task (in µs). Again, this information is not used directly in TTRD02a, but is employed during schedulability analysis and – in later schedulers – it is used to assist in the detection of run-time faults.

[10] Cortex® Microcontroller Software Interface Standard.

As part of this standard, ARM provides a template file system_device.c that must be adapted by the manufacturer of the corresponding microcontroller to match their device.

At a minimum, system_device.c must provide:

- a device-specific system configuration function, SystemInit(); and,
- a global variable that represents the system operating frequency, SystemCoreClock.

The SystemInit() function performs basic device configuration, including (typically) initialisation of the oscillator unit (PLL). The SystemCoreClock value is then set to match the results of this configuration process.

In TTRD02a, we record our expected system operating frequency in main.h by means of Required_SystemCoreClock. We then check that the system has been configured as expected, as shown in Code Fragment 3.

As we enable the WDT unit before we call SCH_Init(), we can force a reset (and a transition to FAIL_SILENT mode) if – for some reason – the system operating frequency is not as expected.

CMSIS also provides us with a SysTick timer to drive the scheduler, and a means of configuring this timer to give the required tick rate (Code Fragment 3).

```
// Now to set up SysTick timer for "ticks" at interval TICKms
if (SysTick_Config(TICKms * SystemCoreClock / 1000))
   {
   // Fatal fault
   ...
   while(1);
   }
```

Code Fragment 3: Configuring the SysTick timer.

A key advantage of using the "SysTick" to drive your scheduler is that this approach is widely used and very easily portable between microcontroller families.

Please refer to Section 2.17 for information about the use of other timers as a source of system ticks.

2.5. The 'Update' function

Code Fragment 4 shows the SysTick ISR.

```
void SysTick_Handler(void)
   {
   // Increment tick count (only)
   Tick_count_G++;
   }
```

Code Fragment 4: The 'Update' function (SysTick_Handler()) from TTRD02a.

This arrangement ensures that the scheduler function can keep track of elapsed time, even in the event that tasks execute for longer than the tick interval.

Note that the function name (SysTick_Handler) is used for compatibility with CMSIS.

2.6. The 'Add Task' function

As the name suggests, the 'Add Task' function – Listing 5 - is used to add tasks to the task array, to ensure that they are called at the required time(s).

The function parameters are (again) as detailed in Table 1.

Please note that this version of the scheduler is "less dynamic" than the version presented in PTTES. One change is that only periodic tasks are supported: "one shot" tasks can no longer be scheduled. This, in turn, ensures that the system can be readily modelled (at design time) and monitored (at run time), processes that we will consider in detail in subsequent chapters.

We say more about the static nature of the schedulers in this book in Section 2.10.

2.7. The 'Dispatcher'

The release of the system tasks is carried out in the function SCH_Dispatch_Tasks(): please refer back to Figure 14 to see this function in context.

The operation of this "dispatcher" function is illustrated schematically in Figure 16.

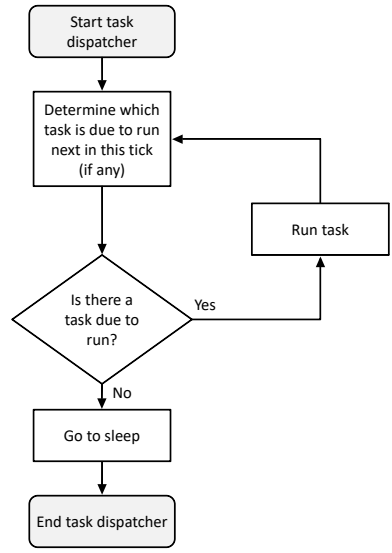


Figure 16: The operation of the dispatcher in the TTC scheduler described in this chapter.

In Figure 16 the dispatcher begins by determining whether there is a task that is currently due to run. If the answer to this question is "yes", the dispatcher runs the task. It repeats this process (check, run) until there are no tasks remaining that are due to run.

The dispatcher then puts the processor to sleep: that is, it places the processor into a power-saving mode. The processor will remain in this power-saving mode until awakened by the next timer interrupt: at this point the timer ISR will be called again, followed by the next call to the dispatcher function (Figure 14).

The SCH_Dispatch function employed in TTRD02a is shown in Listing 5.

Referring again to Figure 14, it should be noted that there is a deliberate split between the process of timer updates and the process of task dispatching.

This division means that it is possible for the scheduler to execute tasks that are longer than one tick interval without missing timer ticks. This gives greater flexibility in the system design, by allowing use of a short tick interval (which can make the system more responsive) and longer tasks (which can – for example – simplify the design process).

Although this flexibility is available in the scheduler described in this chapter, many (but not all) TTC systems are designed to ensure that no tasks are running when a timer interrupt occurs: however, even in such designs, a run-time fault may mean that a task takes longer to complete. Because of the dynamic nature of the scheduler, the system may be able to recover from such run-time faults, provided that the fault is not permanent.

Flexibility in the design process and the ability to recover from faults are two reasons why "dynamic" TT designs (with a separate timer ISR and task dispatch functions) are generally preferred over simpler designs in which tasks are dispatched from the timer ISR.

In addition, separating the ISR and task dispatch functions also makes it very simple to create TT designs with support for task pre-emption (including "time-triggered hybrid" – TTH – architectures): we discuss this process in Chapter 11.

In this listing, please note that Tick_count_G is a "shared resource": it is accessed both in the scheduler Update ISR and in this Dispatcher function. To avoid possible conflicts, we disable interrupts before accessing Tick_count_G in the Dispatcher.

2.8. The 'Start' function

The scheduler Start function (Code Fragment 5) is called after all of the required tasks have been added to the schedule.

```
void SCH_Start(void)
   {
   // Enable SysTick timer
   SysTick->CTRL |= 0x01;

   // Enable SysTick interrupt
   SysTick->CTRL |= 0x02;
   }
```

Code Fragment 5: The SCH_Start() function from TTRD02a. This function should be called after all required tasks have been added to the schedule.

SCH_Start() starts the scheduler timer, and enables the related interrupt.

2.9. The 'sleep' function

In most cases, the scheduler enters "idle" mode at the end of the Dispatcher function: this is achieved by means of the SCH_Go_To_Sleep() function (Code Fragment 6).

```
void SCH_Go_To_Sleep()
   {
   // Enter sleep mode = "Wait For Interrupt"
   __WFI();
   }
```

Code Fragment 6: The SCH_Go_To_Sleep() function from TTRD02a.

The system will then remain "asleep" until the next timer tick is generated.

Clearly, the use of idle mode can help to reduce power consumption. However, a more important reason for putting the processor to sleep is to control the level of "jitter" in the tick timing.

The central importance of jitter in the system operation will be explored in Chapter 4. In Chapter 5, we will explore the use of idle mode in schedulers in more detail.

2.10. Where is the "Delete Task" function?

Traditional approaches to changing system modes in TT designs involve mechanisms for adding / removing tasks from the schedule. For example, the TT task scheduler described in "PTTES" provides SCH_Add_Task() and SCH_Delete_Task() functions that can be called at any time while the scheduler is running.

Such mechanisms for changing system modes have the benefit of simplicity. We will argue throughout this book that simplicity is generally "A Good Thing". However, the author has had the opportunity to review many system designs created using variations on the PTTES schedulers over the years: in doing so, it has become clear that providing an easy way of changing the task set at run-time has had unintended consequences.

TT schedules are – by their very nature – static in nature, and a key strength of this development approach is that a complete task schedule can be carefully reviewed at design time, in order to confirm that all system requirements have been met: we consider this process in detail in Chapter 4. Once the system is in the field, we can then perform monitoring operations to ensure that the run-time behaviour is exactly as expected at design time: we begin to consider how we can achieve this in Chapter 9.

In general, it is extremely difficult to change the system mode in TT designs using conventional methods without significantly undermining this static design process. When tasks can be added or removed from the schedule at "random" times (perhaps – for example – in response to external system events), then the system design becomes dynamic (in effect, it is no longer "time triggered"), and it is not generally possible to predict the precise impact that the mode change will have on all tasks in the schedule.

Even where the perturbations to the behaviour of a TT system during traditional mode changes are short lived, this may still have significant consequences. TT designs are often chosen for use in systems where security is an important consideration. In such designs – because the task schedule is known explicitly in advance – it is possible to detect even very small changes in behaviour that may result from security breaches (for example, if the system code has been changed as the result of a virus, etc). In circumstances where dynamic changes to a task set are permitted (as in traditional mode changes), this may mask security-related issues.

In the approach to system mode changes recommended in this book, we always change task sets (and – therefore – the system mode) by means of a

processor reset. This ensures that the transition is made between one complete set of tasks (that can be subject to detailed analysis, test and verification at design time) and another complete set of tasks (that can be similarly assessed at design time).

2.11. Watchdog timer support

As noted throughout this chapter, TTRD02a requires a watchdog timer. The code used to initialise the LPC1769 watchdog is shown in Listing 11.

The configuration code for the watchdog used in the demo system is straightforward, but it should be noted that this feature of the design is controlled by a jumper: this is needed because watchdog support cannot be enabled when the system is debugged over the JTAG link (if the watchdog is enabled, the processor resets will keep breaking the debug connection). To use this design, you need to insert the jumper (between the pin identified in the Port Header file and ground) in order to enable watchdog support). Please refer to Code Fragment 7 and Figure 17 for further information about this.

```
// Add jumper wire on baseboard to control WDT
// WDT is enabled *only* if jumper is in place.
// (Jumper is read at init phase only)
// Port 2, Pin 3
#define WDT_JUMPER_PORT (2)
#define WDT_JUMPER_PIN (0b1000)
```

Code Fragment 7: WDT jumper settings from the Port Header file (see Listing 1).

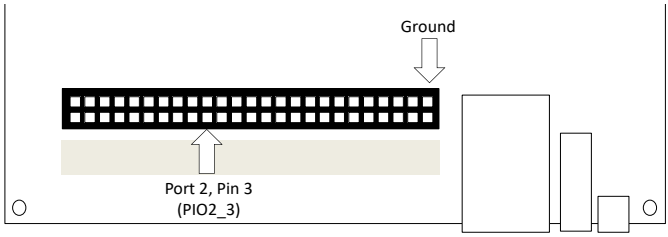


Figure 17: Jumper connections on the EA Baseboard that can be used to enable WDT support. Please refer to Appendix 1 for further information about the baseboard.

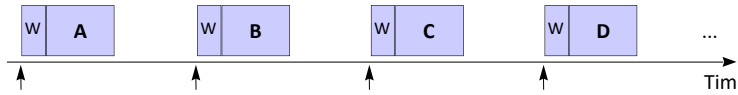


Figure 18: Running a WDT refresh task (shown as Task W) at the start of each tick interval.

The code used to refresh the watchdog is shown in Code Fragment 8. Please note that interrupts are disabled while the WDT is "fed", to avoid the possibility that the timer ISR will be called during this operation. This might be possible (even in a TTC design) in the event of a task overrun.

```
void WATCHDOG_Update(void)
   {
   // Feed the watchdog
   __disable_irq();  // Avoid possible interruption
   LPC_WDT->WDFEED = 0xAA;
   LPC_WDT->WDFEED = 0x55;
   __enable_irq();
   }
```

Code Fragment 8: The WATCHDOG_Update() function from TTRD02a.

2.12. Choice of watchdog timer settings

It is clearly important to select appropriate timeout values for the watchdog timer (WDT) and to refresh this timer in an appropriate way (at an appropriate interval).

One way to do this is to set up a WDT refresh task (like that shown in Code Fragment 8) and schedule this to be released at the start of every tick (Figure 18). In this case, we would probably wish to set the WDT timeout values to match the tick interval. In this way, a task overrun would delay the WDT refresh and cause a transition to a FAIL_SILENT mode (via a WDT reset): see Figure 19.

Used in this way, the WDT provides a form of "Task Guardian" (TG) that can detect and handle task overruns (that is, tasks that – at run time – exceed the WCET figures that were predicted when the system was constructed). Such overruns can clearly have a very significant impact on the system operation.

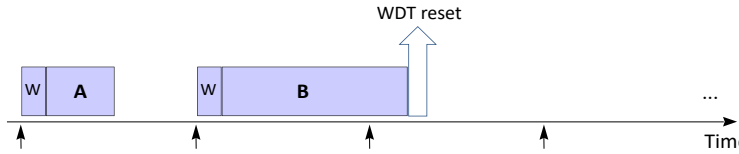


Figure 19: A WDT reset caused by a task overrun.

Please note that this form of TG implementation is effective, but is a rather "blunt instrument": in Chapter 9 we'll begin to explore some techniques that will allow us to identify (and, if required, replace) individual tasks that overrun by more than a few microseconds.

Please also note that – in addition to ensuring that we make a "clean" change between task sets – using a reset between system modes allows us to use appropriate watchdog settings for each system mode. This is possible because the majority of modern COTS processors allow changes to WDT settings (only) at the time of a processor reset.

We will provide examples of designs that use different watchdog timeouts in different modes in Chapter 7.

2.13. The 'Heartbeat' task (with fault reporting)

How can you be sure that the scheduler and microcontroller in your embedded system is operating correctly i.e. how can you get a tangible indication of your systems "health"?

One way to do this is to hook up a JTAG connection to the board, or some other connection (e.g. via a USB port or a UART-based link). Such connections may be straightforward on a workbench during system prototyping, but are not always easy once the embedded processor has been incorporated into a larger piece of equipment.

One effective solution is to implement a "Heartbeat" LED (e.g. see Mwelwa and Pont, 2003).

A Heartbeat LED is usually implemented by means of a simple task that flashes an LED on and off, with a 50% duty cycle and a frequency of 0.5 Hz: that is, the LED runs continuously, on for one second, off for one second, and so on.

Use of this simple technique ensures that the development team, the maintenance team and, where appropriate, the users, can tell at a glance that the system has power, and that the scheduler is operating normally.

In addition, during development, there are two less significant (but still useful) side benefits:

- After a little practice, the developer can often tell "intuitively" – by watching the LED – whether the scheduler is running at the correct rate: if it is not, it may be that the timers have not been initialised correctly, or that an incorrect crystal frequency has been assumed.
- By adding the Heartbeat LED task to the scheduler array *after* all other tasks have been included. This allows the developer to easily see that the task array is adequately large for the needs of the application (if the array is not large enough, the LED will never flash).

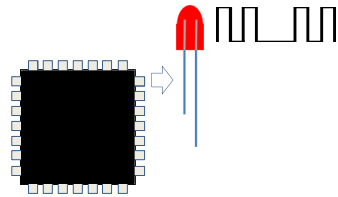


Figure 20: Output from a Fault LED (displaying "Fault Code 2").

We can take this approach one step further, by integrating the Heartbeat task with a fault-reporting function (see Listing 9, p.53). To do this, we maintain a (global) fault variable in the system, and set this to a non-zero value in the event of a fault. If a non-zero fault value is detected by the Heartbeat task, we then reconfigure the task to display this value. For example, we could display "Fault Code 2" on the LED as shown in Figure 20.

2.14. Detecting system overloads (TTRD02b)

When we are using a TTC scheduler, we will generally aim to ensure that all tasks that are scheduled to execute in a given tick have completed their execution time by the end of the tick.

For example, consider Figure 21. In the third tick interval, we would generally expect that the sum of the worst-case execution time of Task E and Task F would be less than the tick interval.

It is very easy to check this during the system execution.

To do so, we set a flag every time a task is released and clear the flag when the task completes. We can then check this flag in the scheduler ISR: if the flag is set, then there is still a task running, and we have an "overload" situation. We can report this using a global fault variable and the "Heartbeat" task that was introduced in Section 2.13.

Please note that this mechanism is intended primarily as a guide to the system loading for use during development, but it can be included in production systems (and, perhaps, checked during scheduled maintenance sessions), if the task overrun situation is a "nuisance" indicator, rather than a safety indicator. This may be the case in systems that have "soft" timing constraints.

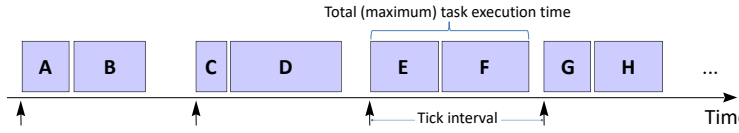


Figure 21: In most TTC designs, we expect that all tasks released in a given tick will complete their execution by the end of the tick.

Please also note that this indicator clearly won't have a chance to work if the WDT setting in your system are set to match the tick interval (as in Figure 19).

You will therefore need to increase the WDT timeout settings if you intend to use this mechanism (during development or in a production system).

TTRD02b includes a complete implementation of the overload detection mechanism.

Code Fragment 9 shows the timer ISR function from TTRD02b: in this ISR the "Task_running_G" flag is checked.

```
void SysTick_Handler(void)
   {
   // Increment tick count (only)
   Tick_count_G++;

   // As this is a TTC scheduler, we don't usually expect
   // to have a task running when the timer ISR is called
   if (Task_running_G == 1)
      {
      // Simple fault reporting via Heartbeat / fault LED.
      // (This value is *not* reset.)
      Fault_code_G = FAULT_SCH_SYSTEM_OVERLOAD;
      }
   }
```

Code Fragment 9: Detecting system overloads: Checking the "Task running" flag.

Code Fragment 10 shows how this flag can be set (when the task is released, in the "Dispatcher" function).

```
// 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

      // Set "Task_running" flag
      __disable_irq();
      Task_running_G = 1;
      __enable_irq();
```

Code Fragment 10: Detecting system overloads: Setting the "Task running" flag.

2.15. Example: Injected (transitory) task overrun (TTRD02b)

We've introduced two simple mechanisms for detecting task overruns in this chapter. In this section, we introduce a simple example that can be used to test these mechanisms. The complete example is illustrated in TTRD02b.

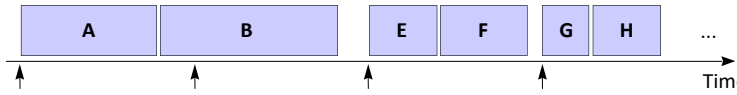


Figure 22: A system design in which there is a "theoretical" system overload.

Please refer to Listing 12. This shows part of a HEARTBEAT_Update task has been adapted to generate a transient overrun event (of duration less than 4 ms), after 15 seconds.

Note that – with the standard WDT settings – this task will only overrun once (because the overrun will be detected by means of the WDT, and the system will be reset: after the reset, the system will enter a fail silent mode).

However, if we extend the WDT setting (to around 10 ms), we can use the mechanisms introduced in Section 2.14 to detect (and report) the system overload situation. These WDT setting are illustrated in TTRD02b.

2.16. Task overruns may not always be "A Bad Thing"

A "soft" TTC design may be able to tolerate occasional task overruns.

In some cases, we can go beyond this. Consider, for example, the design illustrated in Figure 22.

In this design, Task A and Task B are both released in the first tick, but their combined execution time significantly exceeds the tick interval.

In this design (and in many practical cases), this "theoretical" system overload has no impact on the task schedule, because no tasks are due to be released in the second tick. Such a design may well be considered acceptable.

As we will see in Chapter 11, this type of task schedule forms the basis of TTH and TTP scheduler architectures, both of which are in widespread use.

Note that – if you opt to run with longer task combinations in one or more ticks – you may need to adjust the WDT settings for the system (or at least for this system mode), in order to avoid WDT-related resets.

2.17. Porting the scheduler (TTRD02c)

All of the schedulers presented so far in this chapter have employed the SysTick timer to generate the system tick. Such a timer is common across many microcontrollers based on an ARM core and the code can therefore be easily ported.

When you use this approach, you should bear in mind that this solution was intended (by ARM) to be used to generate a 10 ms tick, as is commonly required in conventional operating systems.

As SysTick is based on a 24-bit timer, the maximum interval is (2^{24}-1) ticks: at 100 MHz (the SystemCoreClock frequency used in the majority of the

examples in this book), this provides a maximum tick interval of (16,777,215 / 100,000,000) seconds, or approximately 160 ms.

As an example of an alternative way of generating system ticks, TTRD02c illustrates the use of Timer 0 in the LPC1769 device to drive a TTC scheduler. This is a 32-bit timer. In this design, the required tick interval is provided in microseconds, and the maximum tick interval is $((2^{32} -1) / 100,000,000)$ seconds, or approximately 42 seconds.

Key parts of the scheduler initialisation function for TTRD02c are shown in Listing 13.

2.18. Conclusions

In this chapter, we've introduced some simple but flexible task schedulers for use with sets of periodic co-operative tasks.

In Chapter 3, we present an initial case study that employs one of these schedulers.