The Engineering of

Reliable

Embedded Systems

Developing software for 'SIL 0' to 'SIL 3' designs
using Time-Triggered architectures

Michael J. Pont

SafeTTy Systems

The Engineering of Reliable Embedded Systems

Developing software for 'SIL 0' to 'SIL 3' designs using time-triggered architectures

---

SECOND EDITION

Michael J. Pont

This document contains extracts from the following book:

Pont, M.J. (2016) *"The Engineering of Reliable Embedded Systems: Developing software for 'SIL 0' to 'SIL 3' designs using Time-Triggered architectures"*, (Second Edition) SafeTTy Systems.
ISBN: 978-0-9930355-3-1.

This extract was released: **16 May 2017**

Further information about this book can be found here:

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

SafeTTy Systems

2016

The Engineering of Reliable Embedded Systems

Developing software for 'SIL 0' to 'SIL 3' designs using time-triggered architectures

---

SECOND EDITION

Michael J. Pont

SafeTTy Systems

2016

Published by SafeTTy Systems Ltd
www.SafeTTy.net

First published 2016

First printing November 2016 (Edition 2.0)
Second printing December 2016 (Edition 2.1) with corrections.
Third printing April 2017 (Edition 2.2) with corrections.
Fourth printing May 2017 (Edition 2.3) with corrections.

ISBN 978-0-9930355-3-1

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

In memory of David Robert Jones, 1947-2016

Contents

Acronyms and abbreviations

ASIL	Automotive Safety Integrity Level
BCET	Best-Case Execution Time
BIST	Built-In Self Test
CAN	Controller Area Network
CBD	Contract-Based Design
CCF	Common Cause Failure
CMSIS	Cortex Microcontroller Software Interface Standard
COTS	Commercial Off The Shelf
CPU	Central Processor Unit
DAL	Design Assurance Level
DiV	DISTRIBUTED VARIABLE
DMA	Direct Memory Access
DTI	Diagnostic Test Interval
DuV	DUPLICATED VARIABLE
ECU	Electronic Control Unit
EMI	Electromagnetic Interference
ET	Event Triggered
FAP	Failure Assertion Programming
FFI	Freedom From Interference
FIFO	First-In First-Out (buffer arrangement)
FPGA	Field Programmable Gate Array
FSR	FUNCTIONAL SAFETY REQUIREMENT
HMI	Human-Machine Interface
HRS	HARDWARE REQUIREMENTS SPECIFICATION
IoT	Internet of Things
LINAC	Linear Accelerator
MC	Mixed Criticality
MCU	Microcontroller (Unit)
MMU	Memory Management Unit
MPU	Memory Protection Unit
MST	MULTI-STATE TASK
PFC	PROCESSOR FAULT CODE
POST	Power-On Self Test
PTTES	Patterns for Time-Triggered Embedded Systems
RMA	Rate Monotonic Analysis
SCS	SHARED-CLOCK SCHEDULER
SD	SLAVE DELAY
SIL	Safety Integrity Level
SJ	SLAVE JITTER
SoC	System on Chip

SoRS	SOFTWARE REQUIREMENTS SPECIFICATION
STA	Static Timing Analysis
SyRS	SYSTEM REQUIREMENTS SPECIFICATION
T&V	Test & Verification
TET	TASK Execution Time
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
WDC	Watchdog Controller
WDT	Watchdog Timer
WMC	Washing-Machine Controller

International standards and guidelines

Reference in text	*Full reference*
<u>Industrial / Machinery</u>	
IEC 61508	IEC 61508: 2010
ISO 13849-1	ISO 13849-1: 2015
<u>Automotive</u>	
ISO 26262	ISO 26262: 2011
<u>Household goods</u>	
IEC 60730	IEC 60730-1: 2013
IEC 60335	IEC 60335-1: 2010 + A1: 2013
<u>Medical</u>	
IEC 60601-1	IEC 60601-1: 2005 + AMD1: 2012
IEC 60601-1-8	IEC 60601-1-8: 2006 + AMD1: 2012
IEC 60601-2-1	IEC 60601-2-1: 2009 + AMD1: 2014
IEC 62304	IEC 62304: 2006 + AMD1: 2015
<u>Civil aerospace</u>	
DO-178C	DO-178C: 2012
<u>Generic (coding)</u>	
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' software 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'?

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 determine at run time that it has entered an ABNORMAL PLATFORM STATE[1] and handle this situation in a manner that reduces the risk of UNCONTROLLED PLATFORM FAILURES to an acceptable level.

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 an UNCONTROLLED PLATFORM FAILURE may 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 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).

[1] Definitions for terms that appear within the text in SMALL CAPITALS (such as ABNORMAL PLATFORM STATE and UNCONTROLLED PLATFORM FAILURE) can be found in Appendix 1.

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.

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 software for 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 SCHEDULER (a simple form of 'operating system'). The SCHEDULER will – in turn – release the TASKS at predetermined points in time.

A TT architecture 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 it is possible to model the expected system behaviour precisely. This means that: [i] during the development process, it is possible to demonstrate that all of the requirements have been met; and [ii] at run time, problems can be detected very quickly.

The end result is that we can have a high level of confidence that a TT System will either: [i] operate precisely as required; or [ii] react appropriately if a problem occurs.

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

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

The standards discussed during this book include those listed in Table 1: full references to these standards are given on Page xxiii.

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

Table 1: A rough comparison of the different 'Safety Integrity Levels' (SILs) in some of the international safety standards and guidelines that are considered in this book.

Generic (IEC 61508)	**(SIL 0)**	**SIL 1**	**SIL 2**	**SIL 3**	SIL 4
Civil Aerospace (DO-178C)	Level E	Level D	Level C	Level B	Level A
Medical (IEC 62304)	Class A	Class B		Class C	
Automotive (ISO 26262)	QM	ASIL A	ASIL B / ASIL C	ASIL D	--
Machinery (ISO 13849)	PL a	PL b / PL c	PL d	PL e	--
Household (IEC 60730)	Class A	Class B		Class C	--

e. What microcontroller hardware is used in this book?

Most of the code examples in the book target microcontrollers (MCUs) from STMicroelectronics (STM32F0, STM32F4), NXP / Freescale (LPC17xx), Infineon (XMC4000), and Texas Instruments (TMS570).

For safety-related projects, I would aim to employ an MCU with a PROCESSOR SAFETY MANUAL where this is possible. Such a manual is available for the majority of the MCUs that I consider in this book.

Where safety is not a direct concern, the techniques presented in this book with virtually any MCU.

I say more about selection of suitable MCUs for your project in Appendix 4.

f. What programming language is used?

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

g. Where can I find the code examples?

This book is accompanied by a set of 'Time-Triggered Reference Designs' (TTRDs). The latest set of TTRDs can be found here:

https://www.safetty.net/ttrds

h. Is the code 'freeware'?

Both the TTRDs and this book describe implementations of patented 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 or patented technology associated with this book requires purchase of an appropriate ReliabiliTTy Technology Licence:

https://www.safetty.net/reliabilitty-technology-licences

i. How does this book relate to 'ERES'?

In 2014, I planned to write a number of 'ERES' books, each with a focus on a different market sector (e.g. household goods, automotive, industry). The aim was to focus each book on an appropriate MCU target.

Inevitably, as I began to get a new company off the ground and support a number of challenging new customer projects, I found that I had no time to create more than one book.

When writing 'ERES2' I have tried to be more realistic: I planned for a single book, covering a wider range of sectors and MCUs.

The end result is that the techniques presented in the present book are – at times – a little more advanced than those presented in ERES.

j. Do you plan to write any further books?

You'll find up-to-date information about any future books here: https://www.safetty.net/publications

k. Can you help us build our TT system?

Through my company – SafeTTy Systems Ltd – I have helped many companies to develop embedded systems using TT software architectures.

Please visit the company website for further information about the products, technology and services that we offer: https://www.safetty.net/

l. Did you take all of the photographs?

Various photographs and other images that appear in this book are used under a licence from Dreamstime.com or iStockphoto.

m. 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 who have all contributed to the book (directly and indirectly).

I'm grateful to the people who have invited me to work with the companies that they represent on a range of interesting (and sometimes very challenging) TT projects in the period since January 2014.

I'm grateful to the people who provided comments on the draft chapters of this book (prior to publication) and on earlier print runs, including Prof. Daniel López Amado, Amr Ali Abdelnaby, David Bennetts and Ahmed Abdelfatah.

I'm grateful to Prof. Peter Bernard Ladkin for drawing my attention to the papers by Royce (1970) and Parnas & Clements (1986).

A few years later than originally intended (sorry …), I'd like to thank Chris Hills for his help with the preparation of 'Embedded C'.

Since 2014, I've enjoyed numerous interesting discussions about embedded systems with Attila Gönczi (thank you).

I'd like to thank David Bowie for 'Blackstar' (the 'last show of the tour' – perhaps 43 years later than originally announced, but still far too early); Beth Orton for 'Kidsticks'; Hilary Mantel for 'Wolf Hall'; and Benedict Cumberbatch for restoring my faith in Hamlet.

I'd also like to thank: Mum and Dad, for theatre trips in Pitlochry (and everything else); Andrew, Genevieve, Timothy, Jonathan and Eliza for reintroducing me to Obi-Wan Kenobi ('Now there's a name I've not heard in a long, long time'); Susan, John, Benjamin and Rowena for kayaking and dog walks; Anna, Nick and Ella, for Wales; Jane and Harry for Portsmouth and other adventures; Dev for Hardwick; Anna and Stavros for lending us their very pleasant 'holiday cottage'; Hazel and Tracey for happy weekends; Ginny, for Fleet; Tim for tea and chats in Langham; Anjali, Salil and Elaanya, for dinners in Cambridge (and a fishing trip); Len and Enid, for wet barbecues in Leicester; Andy and James, for looking after Penny; Volker, for dog walks in Germany; Mr Wahl and his colleagues for 'Room 213'; Biggles and Bruce, for walks up a not-so-artificial hill (and Charlotte for making this possible); Cass and Kynall, for mind games; and Sarah, for Bath in an unreliable British sports car (and the 30 years since then).

Michael J. Pont
May 2017 (Edition 2.3)

PART ONE: INTRODUCTION

"Everything should be made as simple as possible but no simpler."

Albert Einstein

While this quotation has been widely attributed to Einstein, it is not clear that he ever actually used this precise form of words. The underlying sentiments have a lot in common with what is usually called 'Occam's Razor'. William of Ockham (c. 1287–1347) was an English Franciscan monk. His 'razor' states that – when selecting between competing hypotheses – the one that requires the fewest assumptions should be selected.

CHAPTER 1: Introduction

This chapter provides an overview of the material that is covered in detail in the remainder of this book.

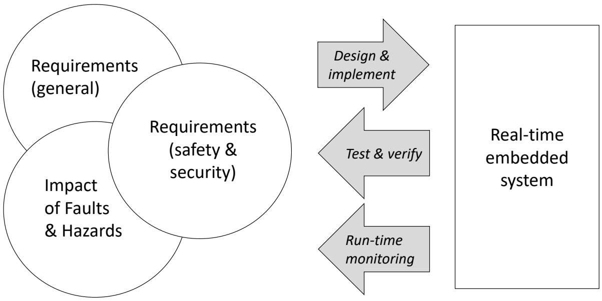


Figure 1: The engineering of reliable real-time embedded systems (overview). In this book, our focus will be on the stages shown on the right of the figure (grey arrows).

1.1. Introduction

The process of engineering reliable, real-time, embedded systems is summarised schematically in Figure 1. Projects will typically begin by recording the requirements for safety, security and general system operation. The impact of potential faults and hazards will be considered. 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 various 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 software for this type of system using time-triggered (TT) architectures.

> What distinguishes TT approaches is that it is possible to model the expected system behaviour precisely. This means that: [i] during the development process, it is possible to demonstrate that all of the requirements have been met; and [ii] at run time, problems can be detected very quickly.
>
> The end result is that we can have a high level of confidence that a TT SYSTEM will either: [i] operate precisely as required; or [ii] react appropriately if a problem occurs.

In this chapter, we explain what a TT software 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 embedded PROCESSOR(S). Such PROCESSORS provide capabilities such as 'anti-lock' behaviour for brake controllers in passenger vehicles, and the features that have transformed basic mobile phones into ubiquitous 'smartphones' in recent years.

The focus in this text is on what are sometimes called 'single-program' embedded 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. These systems can be labelled 'single-program' because the general user is not able to change the software on the system (in the way that 'apps' are added to a smartphone): instead, any upgrades to the steering system – for example – will be performed as part of a service operation, by suitably-qualified technicians.

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 s	= 10^{0} s	= 1000 ms
1 millisecond (ms)	= 0.001 s	= 10^{-3} s	= 1000 µs
1 microsecond (µs)	= 0.000001 s	= 10^{-6} s	= 1000 ns
1 nanosecond (ns)	= 0.000000001 s	= 10^{-9} s	

Box 1

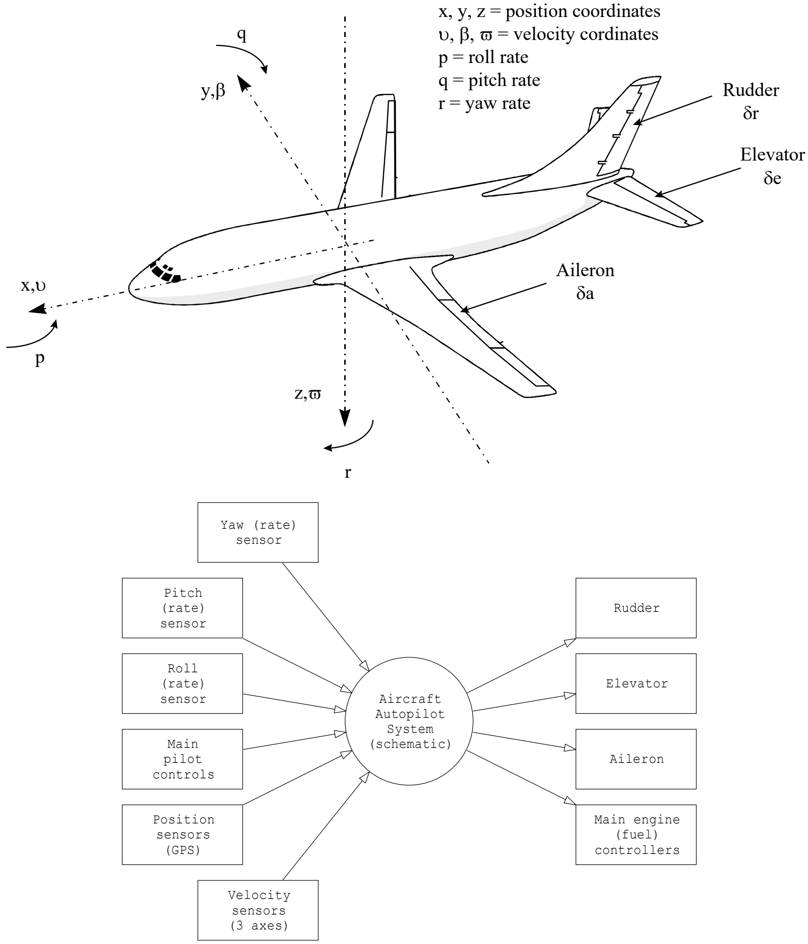


Figure 2: A high-level schematic view of an autopilot system.

1.3. Working with TASKS

A TASK is a 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).

1.4. TT vs. ET architectures

Two software architectures are used in modern embedded systems: these can be labelled as 'event triggered' (ET) and 'time triggered' (TT). The key differences between ET and TT systems arises from the way that the TASKS are released.

For many developers, ET architectures are more familiar. A typical ET design will be required to handle multiple interrupts. For example, interrupts may arise from periodic timer overflows, the arrival of messages on a CAN bus, the pressing of a switch, the completion of an analogue-to-digital conversion and so on. To create such systems, the developer may employ a TASK to handle each event directly: this may involve creating an 'interrupt service routine' (ISR) to deal with each event. The developer may also decide to employ a conventional real-time operating system (RTOS) to support the event handling. Whether an RTOS is used or not, the end result is the same: the system must be designed in such a way that TASK releases – which may occur at 'random' points in time, and in various combinations – can be handled correctly.

> We take the view in this book that a key advantage of ET designs is that they are easy to build. On the other hand, a key challenge with ET designs is that there may be a very large number of possible system states: this can make it difficult to verify that the system will always operate correctly.

The alternative to an event-triggered architecture is a time-triggered ('TT') architecture. When saying that an embedded system has a TT architecture we mean that it executes at least one set of TASKS according to a predetermined schedule. The TASKS must have: [i] well-defined functional behaviour, and [ii] well-defined timing behaviour. The schedule will determine the order of the TASKS are released, the time at which each TASK is released, and whether one TASK can interrupt (pre-empt) another TASK.

In most cases, the starting point for the implementation of a TT design is a 'bare metal' software framework: that is, the system will not usually employ a conventional RTOS, Linux™ or Windows®. In the software framework, a single interrupt will be used, linked to the periodic overflow of a timer. A 'polling' process will then allow interaction with peripherals.

> We view such TT designs as a 'safer subset' of a more general class of ET design (see Figure 3 and Figure 4).
>
> A key advantage of TT designs is that it is (compared with an equivalent ET design) easy to verify that the system will operate correctly. However, we accept that – for teams that lack experience – it can often be more challenging to build a TT design than an equivalent ET design.

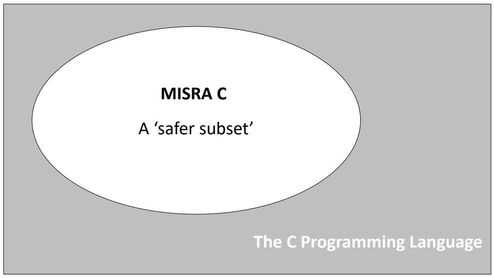


Figure 3: Safer language subsets (for example, MISRA C) are employed by many organisations in order to improve system reliability. See MISRA (2012).

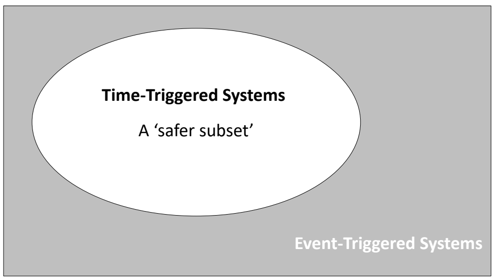


Figure 4: In a manner similar to MISRA C (Figure 3), TT approaches provide a 'safer subset' of ET designs, at the system architecture level.

Our goal in this book is to explore a range of techniques that can facilitate the development of reliable embedded systems using TT software architectures.

1.5. Modelling system timing characteristics

In a TT System, each PROCESSOR releases TASKS in accordance with a predetermined TASK schedule. 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 SYSTEM.

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 TICK. In most designs with a single PROCESSOR, the TICK is implemented by means of a periodic timer interrupt. 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.

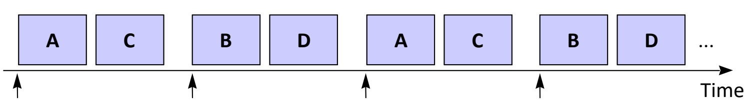


Figure 5: A set of TASKS being released according to a pre-determined schedule.

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 (see Appendix 6) 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 use of TICK LISTS in detail in Chapter 11.

Box 2

In Figure 5, the TASK sequence executed by the PROCESSOR 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, unless: [i] the system changes MODE; [ii] the system is powered down; or [iii] 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 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
```

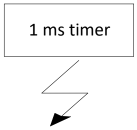
Once the system reaches the end of the TICK LIST, it starts again at the beginning.

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 1.7.

1.6. Working with TTC SCHEDULERS

Many (but by no means all) TT designs are implemented using co-operative TASKS and a 'TTC' SCHEDULER.




```
uint32_t main(void)
   {
   PROCESSOR_Init();

   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }

   return 1;
   }
```

```
void SysTick_Handler(void)
   {
   Tick_count++;
   }
```

Figure 6: A schematic representation of a key components in a simple TTC SCHEDULER.

Figure 6 shows a schematic representation of a key components in such a SCHEDULER.

The SysTick_Handler() function is responsible for keeping track of elapsed time: in this example, this function is linked to a timer that generates interrupts every millisecond.

Within the function PROCESSOR_Init() there will be function calls to initialise the SCHEDULER, initialise the TASKS and then add the TASKS to the schedule.

In function main(), the process of releasing the TASKS is carried out in the function SCH_Dispatch_Tasks().

The operation of a typical SCH_Dispatch_Tasks() function is illustrated schematically in Figure 7. In this figure, 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. The Dispatcher repeats this process until there are no TASKS remaining that are due to run. The Dispatcher then moves the PROCESSOR into a power-saving mode. The PROCESSOR will remain in this mode until awakened by the next timer interrupt: at this point the timer ISR – SysTick_Handler() – will be called again, followed by the next call to the Dispatcher.

It should be noted that there is a deliberate split between the process of timer updates and the process of TASK dispatching. This split means that it is possible for the SCHEDULER to execute TASKS that are longer than one TICK INTERVAL without missing 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 a little more robust in the event of run-time faults.

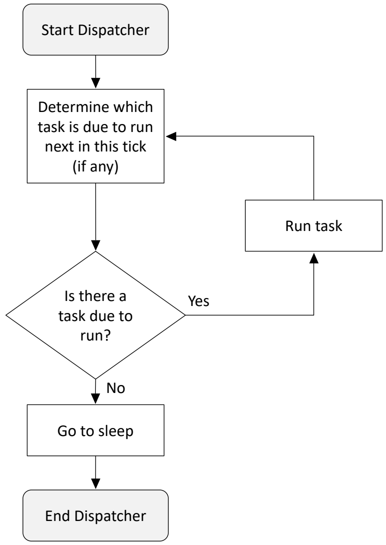


Figure 7: The operation of a Dispatcher.

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.

1.7. 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, higher-priority TASKS can interrupt (pre-empt) lower-priority TASKS.

For example, Figure 8 shows a set of three TASKS: Task A (low-priority), Task B (low-priority), and Task P (high-priority). In this example, the low-priority TASKS may be pre-empted periodically by the high-priority TASK. More generally, this kind of 'time triggered hybrid' (TTH) design may involve multiple co-operative TASKS (all with an equal low priority) and one or more pre-empting TASKS (all with an equal high 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 will focus in this book on TTC designs, but we will say more about TASK pre-emption in Appendix 9 and Appendix 10.

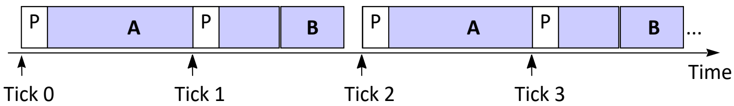


Figure 8: Executing TASKS using a TTH SCHEDULER. See text for details.

1.8. Supporting multiple PROCESSORS and / or multiple cores

Many designs involve the use of more than one PROCESSOR. For example, a modern passenger car might contain 50 or more PROCESSORS, controlling brakes, door windows and mirrors, steering, air bags, and so forth. Similarly, an industrial fire detection system might typically have 200 or more PROCESSORS, associated – for example – with a range of different sensors and actuators.

When developing such 'distributed' designs, we need to consider issues such as the synchronisation of the activities on the different PROCESSORS and the transfer of data between PROCESSORS. We also need to consider how we are going to detect (and respond to) faults on the links between PROCESSORS and on the PROCESSORS themselves. We consider these issues in Chapter 9.

Not all multi-PROCESSOR designs are distributed in nature. In fact, many of the systems that we will consider in this book will employ at least two PROCESSORS that are often located on the same PCB. Such designs are intended to facilitate cross-checking between the PROCESSORS, with the goal of meeting safety requirements (See Figure 9).

In addition to working with multiple PROCESSORS, we may also have more than one core inside each PROCESSOR. We say a little more about this topic in Chapter 3.

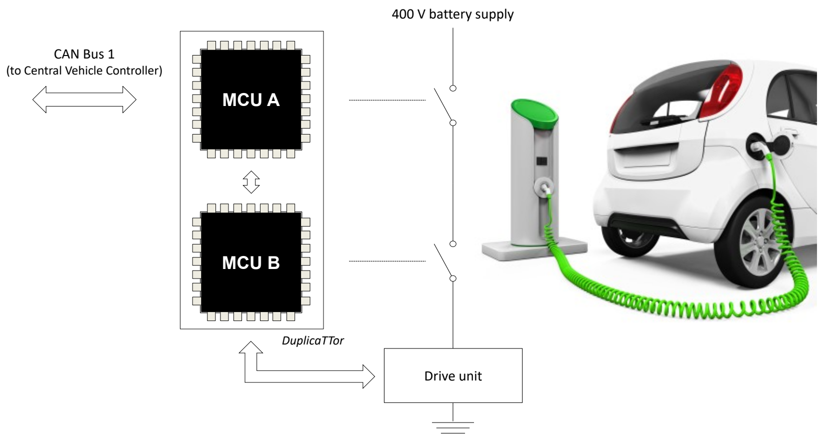


Figure 9: An example of a DuplicaTTor design. We discuss such designs in Appendix 3.
Car image copyright © Nerthuz; licensed from Dreamstime.com.

1.9. Changing MODE

In all of the systems considered in this book, each PROCESSOR will support at least one MODE, called something like 'NORMAL mode'. However many PROCESSORS support additional MODES. For example, Figure 10 shows a schematic representation of a software architecture for an aircraft system with MODES corresponding to the different flight stages (preparing for take off, climbing to cruising height, etc).

In this book, we consider that the MODE is changed if the TASK set is changed. It should therefore be clear that we are likely to have a different TICK LIST for each MODE.

There are two particular features of these MODE changes that should be noted:

- whatever the MODE, the TASKS are **always** released according to a schedule that can be validated and verified when the system is designed;
- the timing of the transition between MODES need **not** be known in advance, a fact that adds significantly to the flexibility of TT systems.

What this means in practice is that – in Figure 10 – the plane can switch between MODES at times that are required by the flying conditions: the timing of such MODE transitions may vary based, for example, on the prevailing weather and / or on the density of the air traffic during the flight. Regardless of the timing of the MODE changes, the TASK schedule in each MODE will have been subject to rigorous test and verification (T&V) processes at design time.

This combination of flexible behaviour combined with the ability to perform rigorous T&V activities is a very effective way of building reliable systems.

We say more about MODES in Chapter 8.

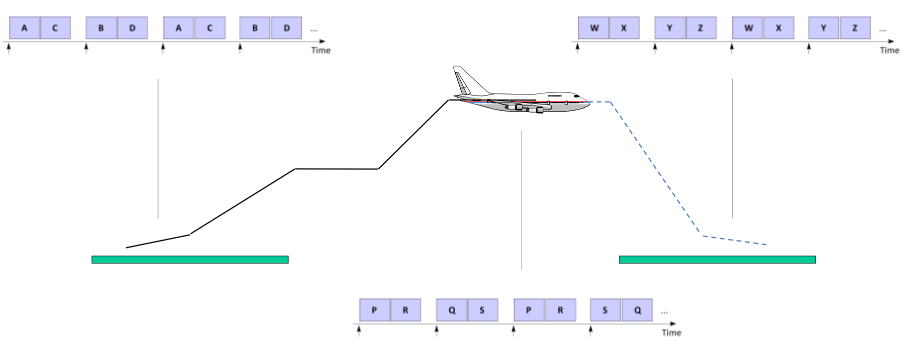


Figure 10: An example of a system with multiple operating MODES.

1.10. The need for run-time monitoring

A three-stage development process is explored in detail during the course of this book:

- the first stage involves modelling the system (using one or more TICK LISTS), as outlined in Section 1.5;
- the second stage involves building the system (for example, using a simple TTC SCHEDULER, as outlined in Section 1.6);
- third stage involves adding support for run-time monitoring.

The last stage in the development process – run-time monitoring – is essential because we need to ensure that the computer system functions correctly 'in the field'.

Some of the threats that we may need to consider are as follows:

- A HARDWARE FAILURE[2] that may result (for example) from electromagnetic interference, or from physical damage;
- A SOFTWARE BUG that may remain in the product even after test and verification processes are complete;
- A DELIBERATE SOFTWARE CHANGE may be introduced into the system, by means of 'computer viruses' and similar security-related attacks.

As an example of a potential 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. 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.

We will explore run-time monitoring solutions in detail in Part Four.

1.11. Bending the rules

Throughout most of this book, we focus on (pure) TT designs. Under normal operation, these designs employ a periodic interrupt to drive a SCHEDULER on each PROCESSOR: where additional interrupt sources are employed, these are synchronised to the TICK.

In Chapter 27 we consider 'Quasi TT' designs. These employ a small number of additional (asynchronous) interrupts. Used with care, these may simplify the design without having a significant (adverse) impact on our ability to model or monitor the system.

---

2 See 'Definitions' in Appendix 1.

1.12. TT Wrappers

In addition to considering Quasi TT designs, we will also consider 'TT Wrappers'.

TT Wrappers can be used to improve confidence in the safety of embedded systems that include components that may have an ET architecture, may be highly adaptive in nature (for example, because they include artificial intelligence components, such as a neural network), and / or may not have been originally developed for use in safety-related systems.

We say more about TT Wrappers in Chapter 20.

1.13. Case studies

This book is intended to present practical advice for developers of reliable embedded systems. In order to 'put theory into practice', the book includes a suite of representative case studies.

These studies explore the development of the following devices:

- An industrial monitoring system (IEC 61508, SIL 2)
- A domestic washing machine (IEC 60730, Class B)
- A hospital radiotherapy machine (IEC 60601-2-1; IEC 62304, Class C)
- A steering-column lock for a passenger car (ISO 26262, ASIL D)
- An aircraft jet engine (DO-178C, Level A)

1.14. Conclusions

In this 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 a first simple TT SCHEDULER.

CHAPTER 2: A simple TTC SCHEDULER

In this chapter, we explore the design and implementation of a TTC SCHEDULER for use with sets of periodic, co-operative TASKS.

2.1. Introduction

In this chapter, we will present a simple TT 'co-operative' SCHEDULER.

Our discussions in this chapter will centre on a 'TT Reference Design' (TTRD): TTRD2-02a. As this design – an implementation of a 'TT02' PLATFORM (see Appendix 2) – will form the foundation for all of the SCHEDULERS presented throughout the remainder of this book, we will explore the operation of this TTRD in detail.

2.2. Hardware target

As noted in the Preface, the TTRDs that are discussed in this book can be applied with a very wide range of PROCESSORS: in this chapter, the introductory example that we present targets an MCU with an ARM Cortex-M0 core. More specifically, we will work with an STM32F091 MCU running on a NUCLEO-F091RC board (Figure 11).

Further information about this MCU (and all of the targets discussed in this book) can be found in Appendix 4.

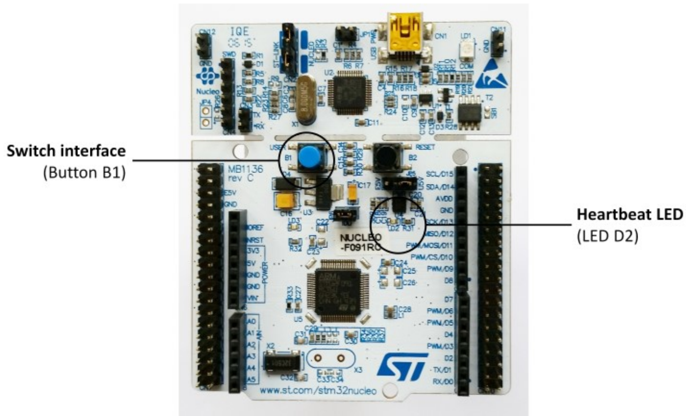


Figure 11: The NUCLEO-F091RC board that is used as the hardware target for the TTRD discussed in this chapter. Photo by MJP.

```
uint32_t main(void)
   {
   PROCESSOR_Init();

   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }

   return 1;
   }
```

1 ms timer

```
void SysTick_Handler(void)
   {
   // Increment tick count and check against limit
   if (++Tick_count_g > SCH_TICK_COUNT_LIMIT)
      {
      // One or more tasks has taken too long to complete
      PROCESSOR_Perform_Safe_Shutdown();
      }
   }
```

Figure 12: An overview of the structure of the TTRD2-02a Scheduler.

2.3. An introduction to TTRD2-02a

TTRD2-02a implements a simple 'Heartbeat' example in which the Scheduler is used to flash an LED ('D2' on the Nucleo board) 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 ... The example also incorporates a switch interface (linked to 'B1' on the board): if the switch is pressed, the LED will stop flashing. As with most of the designs in this book, TTRD2-02a also includes a Task to 'feed' a watchdog timer (WDT).

Figure 12 provides an overview of the structure and use of the Scheduler in this example. Before we consider the internal Scheduler operation, we will consider how the Scheduler is used, starting with the PROCESSOR_Init() function (Code Fragment 1).

```
void PROCESSOR_Init(void)
   {
   PROCESSOR_Identify_Reqd_MoSt();
   PROCESSOR_Configure_Reqd_MoSt();
   }
```

Code Fragment 1: The PROCESSOR_Init() function from TTRD2-02a [STMF091].

As we can see in Figure 12, PROCESSOR_Init() is called at the start of main(). This simple 'wrapper' function is responsible for identifying and configuring the required Mode or State. We will use the same architecture in the great majority of the examples in this book.

In TTRD2-02a, we support only one Mode (NORMAL) and one State (FAIL_SAFE).

Any reset that is caused by the WDT causes the system to enter the FAIL_SAFE State (see Section 2.11), while a power-on reset (and any other reset events in this example) cause the system to enter NORMAL Mode (see Code Fragment 2).

```
void PROCESSOR_Identify_Reqd_MoSt(void)
   {
   // Check cause of reset
   if (RCC_GetFlagStatus(RCC_FLAG_IWDGRST) == SET)
      {
      // Reset was caused by WDT => State ‘Fail Safe’
      Processor_MoSt_g = FAIL_SAFE;
      }
   else
      {
      // Here we treat all other forms of reset in the same way
      // => Mode ‘Normal’
      Processor_MoSt_g = NORMAL;
      }

   // Clear cause-of-reset flags
   RCC_ClearFlag();
   }
```

Code Fragment 2: The PROCESSOR_Identify_Reqd_MoSt() function from TTRD2-02a [STMF091].

In FAIL_SAFE STATE, the system simply ‘halts’ (Code Fragment 3, Code Fragment 4).

```
void PROCESSOR_Perform_Safe_Shutdown(void)
   {
   uint32_t Delay1, Delay2, Heartbeat_state;

   // Here we simply "fail safe" with rudimentary fault reporting.
   // OTHER BEHAVIOUR IS LIKELY TO BE REQUIRED IN YOUR DESIGN

   // *************************************
   // NOTE: This function should NOT return
   // *************************************

   // Set up Heartbeat LED pin
   HEARTBEAT_SW_Init();

   while(1)
      {
      // Flicker Heartbeat LED to indicate fault
      for (Delay1 = 0; Delay1 < 1000000; Delay1++)
         {
         Delay2 *= 3;
         }

      // Change the LED from OFF to ON (or vice versa)
      if (Heartbeat_state == 1)
         {
         Heartbeat_state = 0;
         GPIO_ResetBits(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
         }
      else
         {
         Heartbeat_state = 1;
         GPIO_SetBits(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
         }
      }
   }
```

Code Fragment 3: The PROCESSOR_Perform_Safe_Shutdown() function from TTRD2-02a [STMF091].

There really isn't very much more that we can do in this STATE in TTRD2-02a, but – in a real system design – this is where we should end up if a serious problem has been detected by the PROCESSOR (and no other way of handling this problem has been identified). Deciding what to do in these circumstances requires careful consideration during the system development process.

```
void PROCESSOR_Configure_Reqd_MoSt(void)
   {
   switch (Processor_MoSt_g)
      {
      // Default to "Fail Safe" state
      default:
      case FAIL_SAFE_S:
         {
         // Reset caused by iWDT
         // Trigger "fail safe" behaviour
         PROCESSOR_Perform_Safe_Shutdown();

         break;
         }

      // NORMAL mode
      case NORMAL_M:
         {
         // Set up the scheduler for 1 ms Ticks (Tick Interval in *ms*)
         SCH_Init_Milliseconds(1);

         // Set up WDT
         // Timeout is parameter * 100 µs: 25 => ~2.5 ms
         // NOTE: WDT driven by RC oscillator - timing varies with temperature
         WATCHDOG_Init(25);

         // Prepare for switch-reading task
         SWITCH_BUTTON1_Init();

         // Prepare for heartbeat task
         HEARTBEAT_SW_Init();

         // Add tasks to schedule.
         // Parameters are:
         // A. Task name
         // B. Initial delay / offset (in Ticks)
         // C. Task period (in Ticks): Must be > 0
         //           A                         B  C
         SCH_Add_Task(WATCHDOG_Update,       0, 1);    // Feed watchdog
         SCH_Add_Task(SWITCH_BUTTON1_Update, 0, 10);   // Switch interface
         SCH_Add_Task(HEARTBEAT_SW_Update,   0, 1000); // Heartbeat LED

         // Feed the watchdog
         WATCHDOG_Update();

         break;
         }
      }
   }
```

Code Fragment 4: The PROCESSOR_Configure_Reqd_MoSt() function from TTRD2-02a [STMF091].

When the system reset is not caused by the WDT then – in this example – we enter NORMAL MODE (Code Fragment 4).

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

- set up the SCHEDULER;
- call the initialisation functions for the TASKS; and,
- add the TASKS to the schedule.

In our example, we first set up the SCHEDULER with 1 ms TICKS:

```
SCH_Init(1);
```

We say more about the SCH_Init() function in Section 2.6.

Assuming that initialisation of the SCHEDULER was successful, we then set up the WDT: we'll provide details of this process in Section 2.11.

We then prepare for the switch-interface TASK and the 'Heartbeat' TASK, by means of the SWITCH_BUTTON1_Init() and HEARTBEAT_Init() functions. Further information is provided about these TASKS in Section 2.12 and Section 2.13 respectively.

Having called their 'init' functions, we then add all three TASKS to the schedule by means of the SCH_Add_Task() function:

```
SCH_Add_Task(WATCHDOG_Update, 0, 1);
SCH_Add_Task(SWITCH_BUTTON1_Update, 0, 10);
SCH_Add_Task(HEARTBEAT_SW_Update, 0, 1000);
```

We say more about SCH_Add_Task() in Section 2.8.

2.4. The SCHEDULER components

Having summarised the startup process for TTRD2-02a, we will now consider the implementation and operation of the SCHEDULER in more detail.

The SCHEDULER is made up of the following key components:

- a SCHEDULER data structure;
- an initialisation function;
- a function for adding TASKS to the schedule;
- an interrupt service routine (ISR), used to keep track of elapsed time;
- a Dispatcher (function) that releases TASKS when they are due to run.

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

SCH_MAX_TASKS

You will find SCH_MAX_TASKS in the 'SCHEDULER Header' file in the majority of 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 in any of the MODES.

This memory-allocation process is not dynamic and must be checked for each project.

Please note that this process is **deliberately** static in nature, in line with the recommendations of standards such as IEC 61508-3 (Clause C.2.6.3), ISO 26262-6 (Clause 8.4.4) and MISRA C (Dir. 4.12).

Box 3

2.5. The SCHEDULER data structure and TASK array

At the heart of TTRD2-02a is a user-defined data type (sTask) that collects together the information required about each TASK.

Code Fragment 5 shows the sTask_t implementation used in TTRD2-02a. The members of sTask_t are documented in Table 2.

The TASK set is then defined in the main SCHEDULER file as follows:

```
sTask_t SCH_tasks_g[SCH_MAX_TASKS];
```

```
// User-defined type to store required data for each task
typedef struct
   {
   // Pointer to the task (must be a 'void (void)' function)
   void (*pTask) (void);

   // Delay (Ticks) until the task will (next) be run
   uint32_t Delay;

   // Interval (Ticks) between subsequent runs.
   uint32_t Period;
   } sTask_t;
```

Code Fragment 5: The sTask_t data type used in the SCHEDULERS presented in this chapter. Please refer to Table 2 for further information. [STMF091].

Table 2: The members of the sTask_t data structure (as used in TTRD2-02a).

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 2.11 for a first simple example.
uint32_t Delay	The time (in TICKS) before the TASK will next execute.
uint32_t Period	The TASK period (in TICKS).

2.6. The 'Init' function

The SCHEDULER initialisation function is responsible for:

- initialising the TASK array; and,
- configuring the SCHEDULER TICK SOURCE.

The full function listing is given in Code Fragment 6.

The initialisation process begins by setting the `pTask` member of each TASK in the SCHEDULER array to a 'null pointer' value:

```
SCH_tasks_g[Task_id].pTask = SCH_NULL_PTR;
```

The value represents an address at which no TASK can be stored. This address is usually '0' (and that is the case here): a constant value – SCH_NULL_PTR – is used to make the purpose of the code more explicit (and to simplify the process of porting the code in the future should this ever be required).

```
void SCH_Init_Milliseconds(const uint32_t TICKms)
   {
   for (uint32_t Task_id = 0; Task_id < SCH_MAX_TASKS; Task_id++)
      {
      // Set pTask to 'null pointer'
      SCH_tasks_g[Task_id].pTask = SCH_NULL_PTR;
      }

   // Using CMSIS

   // SystemCoreClock gives the system operating frequency (in Hz)
   if (SystemCoreClock != REQUIRED_PROCESSOR_CORE_CLOCK)
      {
      // We treat this as a Fatal Platform Failure
      PROCESSOR_Perform_Safe_Shutdown();
      }

   // Now to set up SysTick timer for Ticks at interval TICKms
   if (SysTick_Config(TICKms * SystemCoreClock / 1000))
      {
      // Cannot configure SysTick as required
      // We treat this as a Fatal Platform Failure
      PROCESSOR_Perform_Safe_Shutdown();
      }

   // Timer is started by SysTick_Config():
   // we need to disable SysTick timer and SysTick interrupt until
   // all tasks have been added to the schedule.
   SysTick->CTRL &= 0xFFFFFFFC;
   }
```

Code Fragment 6: The SCH_Init_Milliseconds() function from TTRD2-02a [STMF091].

Selecting an appropriate clock source

The version of TTRD2-02a that is presented in this chapter employs an RC oscillator as the main clock source. Such an oscillator will typically have a frequency variation of 2% or more over the operating temperature of the device.

For an introductory example, this is not an inappropriate clock source. However, because of the frequency variation, an RC oscillator may not be suitable as the main oscillator for systems with 'hard' real-time characteristics, including designs that need to support communication protocols such as USB.

Rather than employing an RC oscillator, most practical designs are driven by a crystal oscillator with a frequency variation typically in the region of '100 ppm' (or better). This means '100 parts per million', or a variation of around 0.01% over the operating temperature of the device. By way of comparison, there are 86,400 seconds in a day: a system based on a basic (100 ppm) crystal oscillator might lose 8.6 seconds in a day; a system based on a 2% RC oscillator might lose 1,728 seconds (= 28 minutes) in a day.

Design choices are rarely completely straightforward. In this case, while crystal oscillators are more stable than RC oscillators, they are also more vulnerable to physical damage (for example, as a result of vibration). Many safety-related designs will therefore employ a crystal oscillator as the main clock source, and will automatically switch to an RC oscillator (and perhaps enter a LIMP-HOME PROCESSOR MODE) if the main clock source fails. We say more about this in Chapter 13.

Box 4

The next step in the SCHEDULER initialisation process involves setting up the timer TICKS. In TTRD2-02a, this code is based on the ARM CMSIS[3]. 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 (such as a PLL). The SystemCoreClock value is then set to match the results of this configuration process.

If you look closely at the Nucleo board that we are using in the introductory SCHEDULER example that is described in this chapter (Figure 11) you will see that the 'X3' crystal is missing. X3 is the external crystal oscillator and is – by default – omitted from this board (presumably on grounds of cost).

---

[3] Cortex® Microcontroller Software Interface Standard.

Rather than requiring that readers of this book add a crystal oscillator to their board in order to try out TTRD2-02a, the code is has been designed to operate with the High-Speed Internal (HSI) oscillator that is incorporated in the MCU (see Box 4). Using this RC oscillator and the PLL will allow us to reach a 48 MHz operating frequency.

We record this expected system operating frequency in main.h by means of the constant REQUIRED_PROCESSOR_CORE_CLOCK (Code Fragment 7).

```
// Required system operating frequency (in Hz)
// Will be checked in the scheduler initialisation file
#define REQUIRED_PROCESSOR_CORE_CLOCK (48000000)
```

Code Fragment 7: Part of the SCH_Init_Milliseconds() function from TTRD2-02a [STMF091].

We then check that the system has been configured as expected, as shown in Code Fragment 8.

```
// SystemCoreClock gives the system operating frequency (in Hz)
if (SystemCoreClock != REQUIRED_PROCESSOR_CORE_CLOCK)
   {
   // We treat this as a Fatal Platform Failure
   PROCESSOR_Perform_Safe_Shutdown();
   }
```

Code Fragment 8: Part of the SCH_Init_Milliseconds() function from TTRD2-02a [STMF091].

As suggested by this code example, we attempt to force a safe shutdown if – for whatever reason – the system operating frequency is not as expected.

There is no 'magic' underlying these checks! As mentioned earlier in this section, there is – in the background – a SystemInit() function that is called by the system startup code, before main() is called. The SystemInit() function is – in this case – responsible for configuring the STM32F091 HSI and PLL to give us the required operating frequency.

The SystemInit() function can be found in the file system_stm32f0xx.c.

The setting for this file can – if required – be adjusted using the STM32F0xx Clock Configuration tool (Figure 13).

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 9). Again, we attempt to force a system shutdown if we cannot achieve the expected rate.

Please note that SysTick_Config() starts the timer. We wish to delay the timer start until we have completed the SCHEDULER configuration: we must therefore stop the timer, as shown at the end of Code Fragment 9.

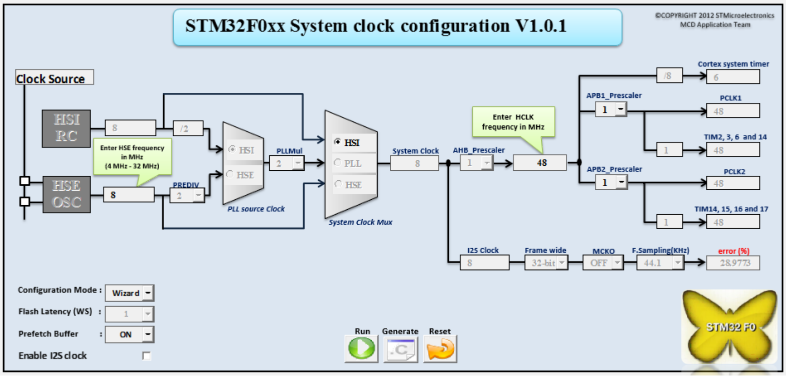


Figure 13: A screenshot from the STM32F0xx Clock Configuration tool.

```
// Now to set up SysTick timer for Ticks at interval TICKms
if (SysTick_Config(TICKms * SystemCoreClock / 1000))
   {
   // Cannot configure SysTick as required
   // We treat this as a Fatal Platform Failure
   PROCESSOR_Perform_Safe_Shutdown();
   }

   // Timer is started by SysTick_Config():
   // we need to disable SysTick timer and SysTick interrupt until
   // all tasks have been added to the schedule.
   SysTick->CTRL &= 0xFFFFFFFC;
```

Code Fragment 9: Part of the SCH_Init_Milliseconds() function from TTRD2-02a [STMF091].

The SysTick timer is widely used and SCHEDULER code based on this component very easily portable between microcontroller families. However, other timers can also be used (without difficulty) to generate the TICK, if required.

2.7. The 'Update' function

Code Fragment 10 shows the SCHEDULER ISR.

This function ensures that the SCHEDULER can keep track of elapsed time (by incrementing the 'tick count' variable): it also uses the same variable to perform a monitoring function.

```
void SysTick_Handler(void)
   {
   // Increment tick count and check against limit
   if (++Tick_count_g > SCH_TICK_COUNT_LIMIT)
      {
      // One or more tasks has taken too long to complete
      PROCESSOR_Perform_Safe_Shutdown();
      }
   }
```

Code Fragment 10: The SysTick_Handler function from TTRD2-02a [STMF091].

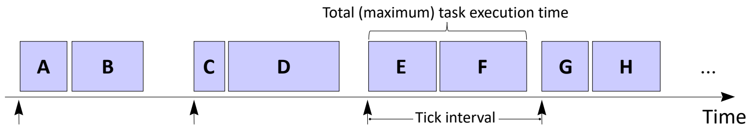


Figure 14: In most TTC designs, we expect that all TASKS released in a given TICK will complete their execution by the end of the TICK.

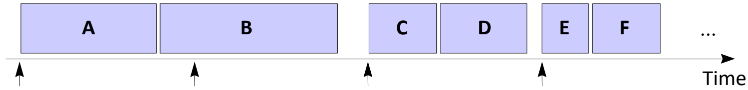


Figure 15: A system design in which the TASKS released in the first TICK INTERVAL have a combined execution time that exceeds the TICK INTERVAL. As this does not (in this case) have any impact on the release of subsequent TASKS (Task C, Task D, …), this behaviour may be acceptable in many designs, not least where Task B has a highly-variable execution time.

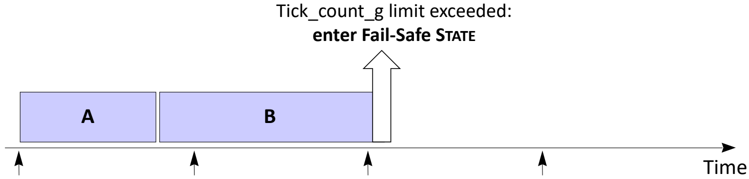


Figure 16: The TASK set from Figure 15, in a situation where Task B exceeds its expected WCET.

To understand the monitoring operation that is performed in the ISR, it should be noted in the majority of TTC designs we expect all TASKS that are released in a TICK INTERVAL to complete before the next TICK (Figure 14). In these circumstances, SCH_TICK_COUNT_LIMIT will be set (in the SCHEDULER header file) to a value of 1:

```
// Usually set to 1, unless 'Long Tasks' are employed
#define SCH_TICK_COUNT_LIMIT (1)
```

In some TTC designs we will **expect** to release a set of TASKS that have a combined 'worst-case execution time' (WCET) that may exceed the TICK INTERVAL: see Figure 15. In these circumstances, we can use a larger tick-count limit. For example, the design illustrated in Figure 15 may be configured as follows:

```
#define SCH_TICK_COUNT_LIMIT (2)
```

This would allow the TASK set to execute as illustrated, but would detect situations in which a longer Task B started to interfere with the execution of Task E. For example, in Figure 16, the variable Tick_count_g would reach a value of 3 before Task E was released (causing the system to enter the STATE FAIL_SAFE in this case).

2.8. The 'Add Task' function

As the name is intended to suggest, the 'Add Task' function – Code Fragment 11 – is used to add TASKs to the schedule.

The function parameters are (again) as detailed in Table 2.

```
void SCH_Add_Task(void (* pTask)(),
                  const uint32_t DELAY,
                  const uint32_t PERIOD)
   {
   uint32_t Task_id = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_g[Task_id].pTask != SCH_NULL_PTR)
          && (Task_id < SCH_MAX_TASKS))
      {
      Task_id++;
      }

   // Have we reached the end of the list?
   if (Task_id == SCH_MAX_TASKS)
      {
      // Task array is full - we treat this as a Fatal Platform Failure
      PROCESSOR_Perform_Safe_Shutdown();
      }

   // Check for 'one shot' tasks
   if (PERIOD == 0)
      {
      // We do not allow 'one shot' tasks (all tasks must be periodic)
      // We treat this as a Fatal Platform Failure
      PROCESSOR_Perform_Safe_Shutdown();
      }

   // If we're here, there is a space in the task array
   // and the task to be added is periodic
   SCH_tasks_g[Task_id].pTask  = pTask;

   SCH_tasks_g[Task_id].Delay  = DELAY + 1;
   SCH_tasks_g[Task_id].Period = PERIOD;
   }
```

Code Fragment 11: The 'Add Task' function from TTRD2-02a [STMF091].

Please note that:

- if an attempt is made to add too many TASKs to the schedule (see Box 3, p.20), the PROCESSOR shuts down;
- only periodic TASKs are supported in this SCHEDULER (and throughout this book); this helps to ensure that the activities on each PROCESSOR can be readily modelled (at design time) and monitored (at run time), as we will demonstrate in later chapters.

2.9. The Dispatcher

The release of the TASKS is carried out in the function SCH_Dispatch_Tasks(): Figure 12 shows this function in context, and Code Fragment 12 presents the source.

```
void SCH_Dispatch_Tasks(void)
   {
   __disable_irq();
   uint32_t Update_required = (Tick_count_g > 0);  // Check tick count
   __enable_irq();

   while (Update_required)
      {
      // Go through the task array
      for (uint32_t Task_id = 0; Task_id < SCH_MAX_TASKS; Task_id++)
         {
         // Check if there is a task at this location
         if (SCH_tasks_g[Task_id].pTask != SCH_NULL_PTR)
            {
            if (--SCH_tasks_g[Task_id].Delay == 0)
               {
               (*SCH_tasks_g[Task_id].pTask)();  // Run the task

               // All tasks are periodic: schedule task to run again
               SCH_tasks_g[Task_id].Delay = SCH_tasks_g[Task_id].Period;
               }
            }
         }

      __disable_irq();
      Tick_count_g--;                          // Decrement the count
      Update_required = (Tick_count_g > 0); // Check again
      __enable_irq();
      }

   // The scheduler enters idle mode at this point
   __WFI();
   }
```

Code Fragment 12: The Dispatcher from TTRD2-02a [STMF091].

Please note that in Code Fragment 12 we have a 'shared resource' (Tick_count_g) that is accessed from both the SCHEDULER ISR and the Dispatcher. Such resources need to be protected, and the disabling of interrupts before Tick_count_g is accessed in the Dispatcher meets this requirement in an appropriate manner.

In most designs (such as that represented by Figure 14), the SCHEDULER operation is as follows (see Figure 17):

- the PROCESSOR is paused in idle mode (it enters this mode at the end of the Dispatcher, see the final lines in Code Fragment 12);
- the TICK 'wakes' the PROCESSOR and triggers the SCHEDULER ISR, which causes Tick_count_g variable to be incremented and checked (Code Fragment 10);

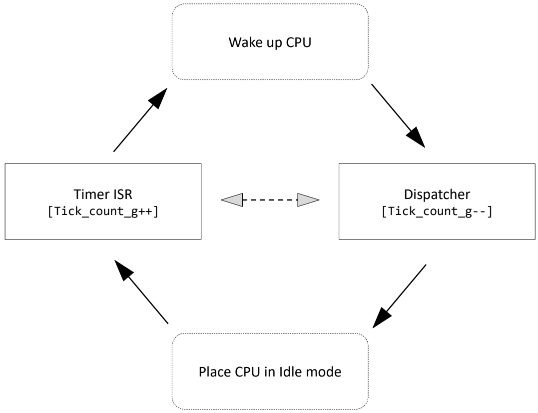


Figure 17: A schematic representation of the SCHEDULER operation.

- assuming that the value of Tick_count_g is within the allowed range, the ISR ends and the Dispatcher starts again (Code Fragment 12);
- within the dispatcher, the value of Tick_count_g is checked and, if this value is greater than 0, the Dispatcher goes through the TASK array in order, updating the 'delay' values for each TASK and releasing any TASKS that are due to run;
- having completed the SCHEDULER update process, the PROCESSOR enters idle mode at the end of the Dispatcher, and the process repeats.

It may seem that the process of checking the value of Tick_count_g at the start of the Dispatcher (and the setting of the Update_required flag) is unnecessary. However, it is possible that the SCHEDULER has not entered idle mode correctly: see Figure 18. Alternatively, the system could be wakened from idle mode by an event other than the SCHEDULER ISR. Without the Update_required flag – or a similar mechanism – it is possible that TASK updates would be carried out more frequently than required in these circumstances. The checks of the value of Tick_count_g are intended to reduce the risk of such problems.

We also need to repeat these checks at the end of the Dispatcher in order to handle TASKS that are still running when the TICK is generated.

Use of idle mode is an important way of controlling jitter (very precisely) in a TTC design, because it allows us to place both the hardware and software into a known configuration. This means that the response time to the timer ISR is (in most MCU architectures) of a fixed duration. For example, in the STM32F091 MCU that is the target for version of TTRD2-02a that is presented in this chapter, the response time to the timer ISR when in idle mode is precisely 20 clock cycles: we would therefore expect to see no TICK jitter.

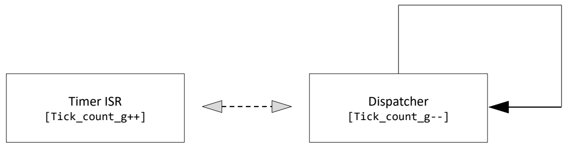


Figure 18: If the system does not enter idle mode, the Dispatcher may be called more frequently than intended. Checks of the value of `Tick_count_g` are used to detect this.

We say more about jitter in real-time systems in Appendix 6. We discuss techniques for measuring such jitter in Appendix 7.

2.10. The 'Start' function

The SCHEDULER Start function (Code Fragment 13) 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 13: The SCH_Start() function from TTRD2-02a. This function should be called after all required Tasks have been added to the schedule [STMF091].

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

2.11. Watchdog timer support

TTRD2-02a includes a TASK to 'feed' the watchdog timer that is incorporated in the PROCESSOR: this is an 'internal WDT', or 'iWDT'. As in the majority of other examples in this book, the iWDT is used in TTRD2-02a: [i] to detect situations in which the SCHEDULER is not operating; and [ii] to trigger a move into a FAIL-SAFE PROCESSOR STATE in these circumstances.

The initialisation function and TASK that make up the TASK MODULE for the iWDT are shown in Code Fragment 14. In TTRD2-02a, we set the iWDT timeout to around 2.5 TICKS, and we 'feed' the timer at the start of each TICK (Figure 19).

Please note that – in a practical design – we would usually aim to use different clock sources for the SCHEDULER and the iWDT: this usually means that we use a crystal oscillator as the clock source for the SCHEDULER, and an RC oscillator to drive the iWDT. As we discussed in Box 4 (p. 22) the stability of RC oscillators is comparatively limited: this means that it is rarely possible to rely on the WDT for precise timing control, and a '2.5 TICK' timeout is usually an effective starting point.

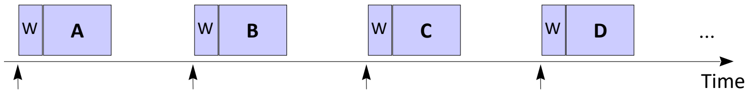


Figure 19: Running a WDT refresh TASK (shown as Task W) at the start of each TICK INTERVAL.

iWDTs are key components in most systems. Unfortunately, in the author's experience, they are very often misused (even in designs that are intended to be safety related). We will say more about the effective use of these simple but important timers in Chapter 16.

2.12. The 'Switch' TASK

We view the process of feeding the iWDT as a 'core TASK' (that will be employed on virtually every system). TTRD2-02a also incorporates two simple 'user TASKS'.

The first user TASK is designed to read the state of a switch that is connected to our microcontroller. In this case, the switch used in the example is 'B1' (which is identified in Figure 11).

B1 is connected on the Nucleo board (essentially) as illustrated in Figure 20. In an ideal world, pressing this button would give rise to a waveform at the port pin which looks something like that illustrated in Figure 21 (top). In practice, all mechanical switch contacts *bounce* after the switch is closed or opened. As a result, the actual input waveform will look more like that shown in Figure 21 (bottom). Usually, switches bounce for less than 20 ms (and this is what we would expect from B1): however large mechanical switches exhibit bounce behaviour for 50 ms or more.

Code Fragment 15 and Code Fragment 16 present the core of the switch-interface TASK MODULE. Code Fragment 16 includes the switch-interface TASK itself – SWITCH_BUTTON1_Update() – that will be called periodically and will report a switch press only after a 'stable' reading has been obtained from the hardware.

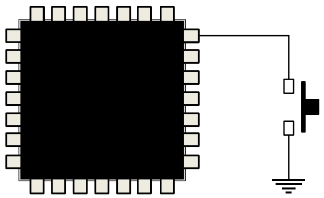


Figure 20: A typical switch connection to a microcontroller. Note that – in practice – the input may be 'opto isolated' (or have some equivalent protection): such an interface would not have an impact on the software architecture that is discussed here.

```
void WATCHDOG_Init(const uint32_t WDT_COUNT)
   {
   // Enable write access to IWDG_PR and IWDG_RLR registers
   IWDG->KR = 0x5555;

   // Set pre-scalar to 4 (‘tick’ is ~100 µs)
   IWDG->PR = 0x00;

   // Counts down to 0 in increments of 100 µs
   // Max reload value is 0xFFF (4095) or ~410 ms (with this prescalar)
   IWDG->RLR = WDT_COUNT;

   // Reload IWDG counter
   IWDG->KR = 0xAAAA;

   // Enable IWDG (the LSI oscillator will be enabled by hardware)
   IWDG->KR = 0xCCCC;

   // Feed watchdog
   WATCHDOG_Update();
   }

/*------------------*/

void WATCHDOG_Update(void)
   {
   // Feed the watchdog (reload IWDG counter)
   IWDG->KR = 0xAAAA;
   }
```

Code Fragment 14: The core of the WDT module from TTRD2-02a [STMF091].

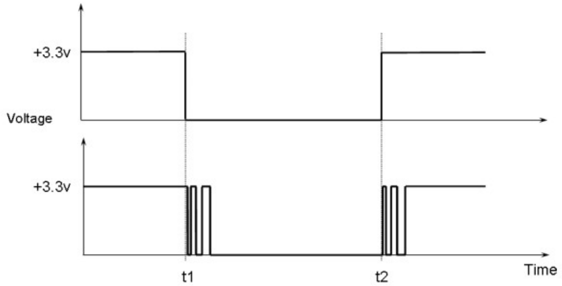


Figure 21: The voltage signal resulting from a mechanical switch. [Top] Idealised waveform resulting from a switch depressed at time t1 and released at time t2 [Bottom] Actual waveform showing leading edge bounce following switch depression and trailing edge bounce following switch release.

```
// Allows NO or NC switch to be used (or other wiring variations)
#define SW_PRESSED (0)

// SW_THRES must be > 1 for correct debounce behaviour
#define SW_THRES (3)

// The current switch state (see Init function)
static uint32_t Switch_button1_pressed_g;

/*-------------------*/

void SWITCH_BUTTON1_Init(void)
   {
   GPIO_InitTypeDef GPIO_InitStruct;

   // Enable GPIOC clock (bit 19)
   RCC->AHBENR |=  (1UL << 19);

   // Configure the switch pin
   GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_IN;
   GPIO_InitStruct.GPIO_Speed = GPIO_Speed_Level_1;
   GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
   GPIO_InitStruct.GPIO_Pin   = BUTTON1_PIN;

   GPIO_Init(BUTTON1_PORT, &GPIO_InitStruct);

   // Set the initial state
   Switch_button1_pressed_g = BUTTON1_NOT_PRESSED;
   }
```

Code Fragment 15: The core of the 'Switch' module from TTRD2-02a, Part 1 of 2 [STMF091]

Please note that the structure of this TASK MODULE is the same as the Heartbeat module: that is, we have an 'Init' function and an 'Update' function (the TASK itself). This is the core structure that we will see for most TASK MODULES in this book.[4] In addition, most of our modules will also include 'Get' / 'Set' functions: in this case, we have a Get function for accessing the switch state.

We will say a little more about Get and Set functions in Section 2.14.

2.13. The 'Heartbeat' TASK

Many PLATFORMS benefit from the inclusion of a 'Heartbeat' LED.

This is usually implemented by means of a TASK that flashes an LED on and off, with a 50% duty cycle and a frequency of 0.5 Hz: that is, the LED is on for one second, off for one second, on for one second …

Use of this simple reporting mechanism 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.

The Heartbeat module from TTRD2-02a is shown in Code Fragment 17.

---

4 Some modules may also require a 'Deinit' function (see Chapter 8) and / or an interface that supports testing, including fault injection (see Chapter 13).

```
void SWITCH_BUTTON1_Update(void)
   {
   // Duration of switch press
   static uint32_t Duration_s = 0;

   // Read the pin state
   uint32_t Button1_input = GPIO_ReadInputDataBit(BUTTON1_PORT, BUTTON1_PIN);

   if (Button1_input == SW_PRESSED)
      {
      Duration_s += 1;

      if (Duration_s > SW_THRES)
         {
         Duration_s = SW_THRES;

         Switch_button1_pressed_g = BUTTON1_PRESSED;
         }
      else
         {
         // Switch pressed, but not yet for long enough
         Switch_button1_pressed_g = BUTTON1_NOT_PRESSED;
         }
      }
   else
      {
      // Switch not pressed - reset the count
      Duration_s = 0;

      // Update status
      Switch_button1_pressed_g = BUTTON1_NOT_PRESSED;
      }
   }

/*-------------------*/

uint32_t SWITCH_BUTTON1_Get_State(void)
   {
   return Switch_button1_pressed_g;
   }
```

Code Fragment 16: The core of the ‘Switch’ module from TTRD2-02a, Part 2 of 2 [STMF091].

In Code Fragment 17, the Heartbeat TASK incorporates a link to the switch-interface TASK (Section 2.12), by means of which we ensure that the LED stops flashing if the switch is pressed.

2.14. Transferring data between TASKS

In previous introductory texts (and the previous edition of this book), the author has used global variables as a means of transferring data between TASKS.

In the present text, we have a focus on the development of reliable and (potentially) safety-related systems: in such environments, we would generally wish to make limited use of global variables. For example, ISO 26262-6 (Table 8) recommends that global variables are avoided (or their use justified) in all safety-related designs (from ‘ASIL A’ to ‘ASIL D’).

```
void HEARTBEAT_SW_Init(void)
   {
   GPIO_InitTypeDef GPIO_InitStruct;

   // Enable GPIOA clock (bit 17)
   RCC->AHBENR |=  (1UL << 17);

   // Configure port pin for the LED
   GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_OUT;
   GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStruct.GPIO_Speed = GPIO_Speed_Level_1;
   GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
   GPIO_InitStruct.GPIO_Pin   = HEARTBEAT_LED_PIN;

   GPIO_Init(HEARTBEAT_LED_PORT, &GPIO_InitStruct);
   }

/*------------------*/

void HEARTBEAT_SW_Update(void)
   {
   static uint32_t Heartbeat_state_s = 0;

   // Check switch (Button 1) state
   if (SWITCH_BUTTON1_Get_State() == SWITCH_NOT_PRESSED)
      {
      // Switch is *not* pressed: normal 'heartbeat' behaviour

      // Change the LED from OFF to ON (or vice versa)
      if (Heartbeat_state_s == 1)
         {
         Heartbeat_state_s = 0;
         GPIO_ResetBits(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
         }
      else
         {
         Heartbeat_state_s = 1;
         GPIO_SetBits(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
         }
      }
   }
```

Code Fragment 17: The core of the 'Heartbeat' module from TTRD2-02a [STMF091]

In place of global variables, we employ 'private' variables in each TASK module, and provide 'Get' and / or 'Set' functions to access these data. It is expected that such a Get / Set arrangement will be familiar to the majority of readers of this book.

As an example, Code Fragment 17 shows use of the SWITCH_BUTTON1_Get_State() function to read the state of the switch: the full function definition can be found in Code Fragment 16.

2.15. Conclusions

In this chapter, we've introduced a simple but flexible SCHEDULER for use with sets of periodic co-operative TASKS. This design will form the foundation for all of the SCHEDULERS presented throughout the remainder of this book.

In Part Two, we start to look at the design of effective TASKS for use with TT systems.

2.16. Further reading

Mwelwa, C. and Pont, M.J. (2003) *'Two new patterns to support the development of reliable embedded systems'* Paper presented at the Second *Nordic Conference on Pattern Languages of Programs,* ('VikingPLoP 2003'), Bergen, Norway, September 2003.

Pont, M.J. (2001) *'Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers'*, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

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 ('VikingPloP 2002')*, pp.159-200. Published by Microsoft Business Solutions. ISBN: 87-7849-769-8.