The Engineering of Reliable Embedded Systems

Michael J. Pont

SafeTTy Systems™

Preface to the PDF release (2016-12-11)

This is a 'PDF edition' of the following book:

Pont, M.J. (2014) "*The Engineering of Reliable Embedded Systems: LPC 1769 edition*", SafeTTy Systems. ISBN: 978-0-9930355-0-0.

This book has been made freely available in this format following publication of the second edition in November 2016:

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.

The 'TTRDs' associated with this book can be downloaded here:

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

Information about 'ERES2' can be found here:

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

Information about the differences between 'ERES1' and 'ERES2' is here:

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

Please refer to Page xxv of this document for information about use of the techniques and code presented here (and in the associated TTRDs) in commercial projects.

I hope that some readers find this book to be of interest.

Michael J. Pont
11 December 2016.

This file may be freely distributed, provided that it is not altered in any way.

The Engineering of Reliable Embedded Systems

LPC1769 edition

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

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

The Engineering of Reliable Embedded Systems

LPC1769 edition

Michael J. Pont

Safetty Systems™
2014

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

Published by SafeTTy Systems Ltd
www.SafeTTy.net

First published 2014

First printing December 2014
Second printing January 2015
Third printing March 2015
Fourth printing April 2015

Copyright © 2014-2015 by SafeTTy Systems Ltd

The right of Michael J. Pont to be identified as Author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

ISBN 978-0-9930355-0-0

All rights reserved; no part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of the publishers. This book may not be lent, resold, hired out or otherwise disposed of in any form of binding or cover other than that in which it is published, without the prior consent of the publishers.

Trademarks
MoniTTor® is a registered trademark of SafeTTy Systems Ltd.
PredicTTor® is a registered trademark of SafeTTy Systems Ltd.
WarranTTor® is a registered trademark of SafeTTy Systems Ltd.
ReliabiliTTy® is a registered trademark of SafeTTy Systems Ltd.
SafeTTy Systems® is a registered trademark of SafeTTy Systems Ltd.
ARM® is a registered trademark of ARM Limited.
NXP® is a registered trademark of NXP Semiconductors.

All other trademarks acknowledged.

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

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

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

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

Contents

Definitions ..... **xv**

Acronyms and abbreviations ..... **xvii**

Reference designs ..... **xix**

International standards and guidelines ..... **xxi**

Preface ..... **xxiii**

a. What is a "reliable embedded system"? xxiii
b. Who needs reliable embedded systems? xxiii
c. Why work with "time-triggered" systems? xxiv
d. How does this book relate to international safety standards? xxiv
e. What programming language is used? xxv
f. Is the source code "freeware"? xxv
g. How does this book relate to other books in the "ERES" series? xxv
h. What processor hardware is used in this book? xxvi
i. How does this book relate to "PTTES"? xxvi
j. Is there anyone that you'd like to thank? xxvii

PART ONE: INTRODUCTION **1**

CHAPTER 1: Introduction **3**

1.1. Introduction 3
1.2. Single-program, real-time embedded systems 4
1.3. TT vs. ET architectures 6
1.4. Modelling system timing characteristics 7
1.5. Working with "TTC" schedulers 9
1.6. Supporting task pre-emption 11
1.7. Different system modes 11
1.8. A "Model-Build-Monitor" methodology 12
1.9. How can we avoid Uncontrolled System Failures? 14
1.10. Conclusions 16

CHAPTER 2: Creating a simple TTC scheduler **17**

2.1. Introduction 17
2.2. A first TTC scheduler (TTRD02a) 21
2.3. The scheduler data structure and task array 21
2.4. The 'Init' function 21
2.5. The 'Update' function 23
2.6. The 'Add Task' function 24
2.7. The 'Dispatcher' 24
2.8. The 'Start' function 26
2.9. The 'sleep' function 26
2.10. Where is the "Delete Task" function? 26
2.11. Watchdog timer support 28
2.12. Choice of watchdog timer settings 29

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

2.13. The 'Heartbeat' task (with fault reporting) 30
2.14. Detecting system overloads (TTRD02b) 31
2.15. Example: Injected (transitory) task overrun (TTRD02b) 32
2.16. Task overruns may not always be "A Bad Thing" 33
2.17. Porting the scheduler (TTRD02c) 33
2.18. Conclusions 34
2.19. Code listings (TTRD02a) 35
2.20. Code listings (TTRD02b) 60
2.21. Code listings (TTRD02c) 61

CHAPTER 3: Initial case study 63
3.1. Introduction 63
3.2. The focus of this case study 63
3.3. The purpose of this case study 63
3.4. A summary of the required system operation 65
3.5. The system architecture 65
3.6. The system states 66
3.7. Implementation platform for the prototype 66
3.8. The "system" task 68
3.9. The "selector dial" task 68
3.10. The "start switch" task 69
3.11. The "door lock" task 69
3.12. The "water valve" task 69
3.13. The "detergent hatch" task 69
3.14. The "water level" task 69
3.15. The "water heater" task 69
3.16. The "water temperature" task 69
3.17. The "drum motor" task 69
3.18. The "water pump" task 69
3.19. The Heartbeat task 69
3.20. Communication between tasks 70
3.21. Where do we go from here? 71
3.22. Conclusions 72
3.23. Code listings (TTRD03a) 73

PART TWO: CREATING RELIABLE TTC DESIGNS 91

CHAPTER 4: Modelling system timing characteristics 93
4.1. Introduction 93
4.2. Basic Tick Lists 93
4.3. Determining the required tick interval 94
4.4. Working with "Short Tasks" 94
4.5. The hyperperiod 95
4.6. Performing GCD and LCM calculations 95
4.7. Synchronous and asynchronous task sets 95
4.8. The importance of task offsets 96
4.9. The Task Sequence Initialisation Period (TSIP) 97
4.10. Modelling CPU loading 97

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

4.11. Worked Example A: Determining the maximum CPU load 98
4.12. Worked Example A: Solution 99
4.13. Modelling task jitter 100
4.14. Worked Example B: Modelling task release jitter 104
4.15. Worked Example B: Solution 104
4.16. Modelling response times 105
4.17. Worked Example C: An “emergency stop” interface 107
4.18. Worked Example C: Solution 110
4.19. Generating Tick Lists 111
4.20. Conclusions 111

CHAPTER 5: Obtaining data for system models 113
5.1. Introduction 113
5.2. The importance of WCET / BCET information 113
5.3. Challenges with WCET / BCET measurements 114
5.4. Instrumenting a TTC scheduler: WCET-BCET (TTRD05a) 116
5.5. Example: An injected task overrun (TTRD05b) 117
5.6. Obtaining jitter measurements: Tick jitter (TTRD05c) 117
5.7. Example: The impact of idle mode on a TTC scheduler 117
5.8. Obtaining jitter measurements: Task jitter (TTRD05d) 118
5.9. Example: The impact of task order on a TTC scheduler 119
5.10. Traditional ways of obtaining task timing information 121
5.11. Generating a Tick List on an embedded platform (TTRD05e) 121
5.12. Creating a Tick List that meets your requirements 123
5.13. Conclusions 124

CHAPTER 6: Timing considerations when designing tasks 125
6.1. Introduction 125
6.2. Design goal: “Short Tasks” 126
6.3. The need for multi-stage tasks 126
6.4. Example: Measuring liquid flow rates 127
6.5. Example: Buffering output data 129
6.6. Example: DMA-supported outputs 131
6.7. The need for timeout mechanisms 131
6.8. Example: Loop timeouts 134
6.9. Example: Hardware timeout 135
6.10. Handling large / frequent data inputs 135
6.11. Example: Buffered input 135
6.12. Example: DMA input 136
6.13. Example: Multi-core input (“Smart” buffering) 136
6.14. Example: Customised hardware support 137
6.15. Execution-time balancing in TTC designs (task level) 137
6.16. Execution-time balancing in TTC designs (within tasks) 138
6.17. ASIDE: Execution-time balancing in TTH / TTP designs 139
6.18. Example: Execution-time balancing at an architecture level 139
6.19. Example: Manual execution-time balancing 140
6.20. Example: Sandwich delays for execution-time balancing 141

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

6.21. Appropriate use of Sandwich Delays 142
6.22. Conclusions 142
6.23. Code Listings (TTRD06a) 143
6.24. Code Listings (TTRD06b) 146

CHAPTER 7: Multi-mode systems..... 147
7.1. Introduction 147
7.2. What does it mean to change the system mode? 147
7.3. Mode change or state change? 148
7.4. The timing of mode changes..... 149
7.5. Implementing effective multi-mode designs 149
7.6. The architecture of a multi-mode system 150
7.7. Different system settings in each mode (if required) 151
7.8. Design example with multiple Normal Modes (TTRD07a)..... 151
7.9. Design example with fault injection (TTRD07b)..... 153
7.10. The process of “graceful degradation” 153
7.11. Design example supporting graceful degradation (TTRD07c)..... 154
7.12. Mode changes in the presence of faults..... 155
7.13. Conclusions 155
7.14. Code listings (TTRD07a) 156
7.15. Code listings (TTRD07c) 165

CHAPTER 8: Task Contracts (Resource Barriers)..... 177
8.1. Introduction 177
8.2. Origins of “Contracts” in software development..... 178
8.3. What do we mean by a “Task Contract”? 178
8.4. Numerical example 179
8.5. Control example 179
8.6. Timing is part of the Task Contract 180
8.7. Implementing Task Contracts (overview) 181
8.8. Implementing Task Contracts (timing checks) 181
8.9. Implementing Task Contracts (checking peripherals)..... 182
8.10. Example: Feeding the WDT..... 184
8.11. One task per peripheral 184
8.12. What about shared data? 185
8.13. Implementing Task Contracts (protecting data transfers) 186
8.14. An alternative way of detecting corruption in shared data 187
8.15. How can we detect corruption of the scheduler data? 187
8.16. Making use of the MPU 187
8.17. Supporting Backup Tasks 188
8.18. What do we do if our Resource Barrier detects a fault? 188
8.19. Task Contracts and international standards 189
8.20. Conclusions 190
8.21. Code listings (TTRD08a) 191

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

CHAPTER 9: Task Contracts (Time Barriers)..... 235
9.1. Introduction 235
9.2. An evolving system architecture..... 236
9.3. System operation..... 237
9.4. Handling execution-time faults..... 238
9.5. Example: TTC scheduler with MoniTTor (TTRD09a) 239
9.6. Working with long tasks 240
9.7. External MoniTTor solutions..... 241
9.8. Alternatives to MoniTTor..... 241
9.9. Conclusions..... 241
9.10. Code listings (TTRD09a) 242

CHAPTER 10: Monitoring task execution sequences..... 247
10.1. Introduction 247
10.2. Implementing a predictive monitor 249
10.3. The importance of predictive monitoring..... 251
10.4. The resulting system architecture 251
10.5. Handling task-sequence faults..... 252
10.6. Example: Monitoring a 3-mode system (TTRD10a) 252
10.7. Creating the Task-Sequence Representation (TSR) 252
10.8. Side effects of the use of a PredicTTor unit..... 253
10.9. Synchronous vs. asynchronous task sets revisited 253
10.10. The Task Sequence Initialisation Period (TSIP) 254
10.11. Worked example..... 255
10.12. Solution..... 256
10.13. Example: Monitoring another 3-mode system (TTRD10b) 256
10.14. Where should we store the TSR?..... 256
10.15. Links to international standards 257
10.16. Conclusions..... 257
10.17. Code listings (TTRD10a) 258

PART THREE: CREATING RELIABLE TTH AND TTP DESIGNS 263

CHAPTER 11: Supporting task pre-emption 265
11.1. Introduction 265
11.2. Implementing a TTH scheduler 267
11.3. Key features of a TTH scheduler 268
11.4. TTH example: Emergency stop (TTRD11a)..... 269
11.5. TTH example: Medical alarm in compliance with IEC 60601 270
11.6. TTH example: Long pre-empting section (TTRD11b) 271
11.7. Protecting shared resources 272
11.8. A TTP scheduler with shared resources (TTRD11c1)..... 273
11.9. The challenges of priority inversion..... 273
11.10. Implementing a “ceiling” protocol (TTRD11c2) 274
11.11. Monitoring task execution times (TTRD11d) 274
11.12. Use of watchdog timers in TTH and TTP designs 277
11.13. Conclusions..... 278

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

CHAPTER 12: Maximising temporal determinism 279
12.1. Introduction 279
12.2. Jitter levels in TTH designs (TTRD12a) 279
12.3. Reducing jitter in TTH designs (TTRD12b) 280
12.4. How to avoid PI in TT systems (Overview) 281
12.5. Using code balancing with a PredicTTor unit 281
12.6. Do you need to balance the code in your system? 283
12.7. Using code balancing to prevent PI 284
12.8. How to avoid PI in TT systems (TRA protocols) 284
12.9. How to incorporate a TRAP in your design 285
12.10. A complete TTP design (TTRD12d) 285
12.11. Conclusions 285

PART FOUR: COMPLETING THE SYSTEM 287

CHAPTER 13: From Task Contracts to System Contracts 289
13.1. Introduction 289
13.2. What is a "System Contract"? 290
13.3. Generic POST operations 290
13.4. Example: POST operations that meet IEC 60335 requirements 292
13.5. Checking the system configuration 293
13.6. Example: Check the system configuration 294
13.7. Generic periodic checks (BISTs) 295
13.8. Example: BISTs in compliance with IEC 60335 295
13.9. Additional periodic tests 296
13.10. Example: Monitoring CPU temperature (TTRD13a) 296
13.11. System modes 296
13.12. Tasks and backup tasks 296
13.13. Example: Design of a backup task for analogue outputs 299
13.14. Shutting the system down 299
13.15. Performing initial system tests 300
13.16. International standards 301
13.17. Conclusions 302

CHAPTER 14: TT platforms 303
14.1. Introduction 303
14.2. Platform TT00 303
14.3. Platform TT01 304
14.4. Platform TT02 304
14.5. Platform TT03 305
14.6. Platform TT04 307
14.7. Platform TT05 309
14.8. Design example: Avoiding common-cause failures 312
14.9. Design example: Implementing a "1oo2p" architecture 312
14.10. Selecting an MCU: General considerations 313
14.11. Selecting an MCU: Supporting a TT scheduler 314
14.12. Selecting an MCU: WarranTTor platform 315
14.13. Conclusions 316

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

CHAPTER 15: Revisiting the case study 317
15.1. Introduction 317
15.2. An overview of the development process 317
15.3. The system requirements 319
15.4. Considering potential threats and hazards 319
15.5. Considering international safety standards 319
15.6. Potential system platform 320
15.7. Does the team have the required skills and experience? 321
15.8. Shutting the system down 321
15.9. Powering the system up 323
15.10. Periodic system checks 323
15.11. The system modes 324
15.12. The system states 324
15.13. The task sets 327
15.14. Modelling the task set and adjusting the task offsets 327
15.15. Fault-detection and fault handling (overview) 329
15.16. Using Lightweight Resource Barriers 329
15.17. A MoniTTor unit 331
15.18. A PredicTTor unit 331
15.19. A simple system model and fault-injection facility 331
15.20. Fault codes and fault reporting 335
15.21. Revisiting the system requirements 335
15.22. Directory structure 339
15.23. Running the prototype 339
15.24. International standards revisited 340
15.25. Conclusions 340

PART FIVE: CONCLUSIONS 341

CHAPTER 16: Conclusions 343
16.1. The aim of this book 343
16.2. The LPC1769 microcontroller 343
16.3. From safety to security 344
16.4. From processor to distributed system 344
10.5. Conclusions 344

APPENDIX 345

APPENDIX 1: LPC1769 test platform 347
A1.1. Introduction 347
A1.2. The LPC1769 microcontroller 347
A1.3. LPCXpresso toolset 347
A1.4. LPCXpresso board 348
A1.5. The EA Baseboard 348
A1.6. Running TTRD02a 351
A1.7. Running TTRD05a 352
A1.8. Conclusions 353

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

Full list of references and related publications 355

Index 367

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

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.

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

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).

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

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

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

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

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
TTRD11c1:	TTP scheduler with shared resources
TTRD11c2:	TTP scheduler with a "ceiling" protocol
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 TT03
TTRD14b:	System Platform TT04
TTRD14c1:	System Platform TT05 (Control Unit)
TTRD14c2:	System Platform TT05 (WarranTTor Unit)
TTRD15a:	Framework for washing-machine controller (TT03)
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

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

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

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)

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

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

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 xv.

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

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

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)

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

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.xxi.

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).

---

2 Patents applied for.

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 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" (Pont, 2001) 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

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

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,

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

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
April 2015

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

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.

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

CHAPTER 1: Introduction

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

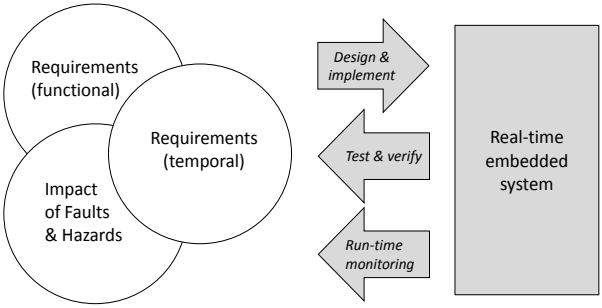


Figure 1: The engineering of reliable real-time embedded systems (overview). In this book, our focus will be on the stages shown on the right of the figure (grey boxes).

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.

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

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.

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

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

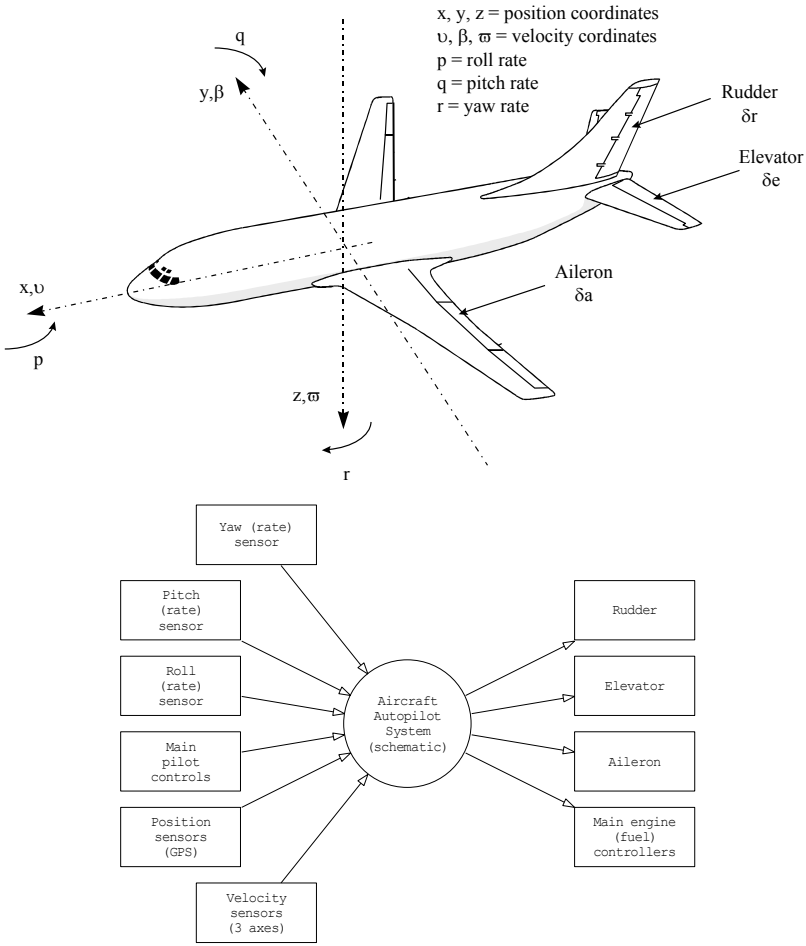


Figure 2: A high-level schematic view of an autopilot system.

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

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.

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

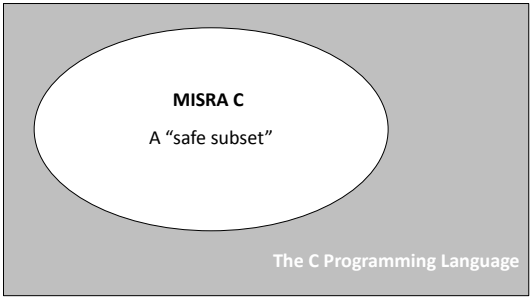


Figure 3: Safer language subsets (for example, MISRA C) are employed by many organisations in order to improve system reliability. See MISRA (2012).

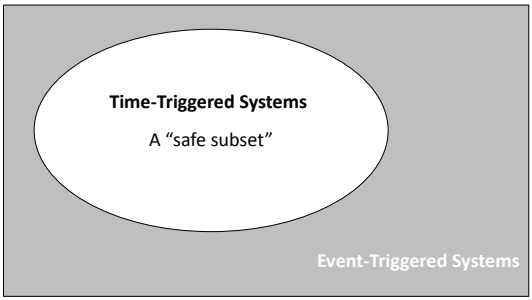


Figure 4: In a manner similar to MISRA C (Figure 3), TT approaches provide a "safer subset" of ET designs, at the system architecture level.

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.

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

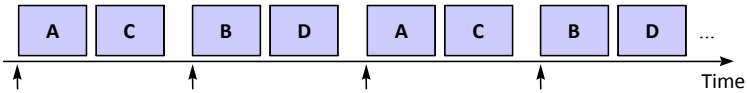


Figure 5: A set of tasks being released according to a pre-determined schedule.

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.

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

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

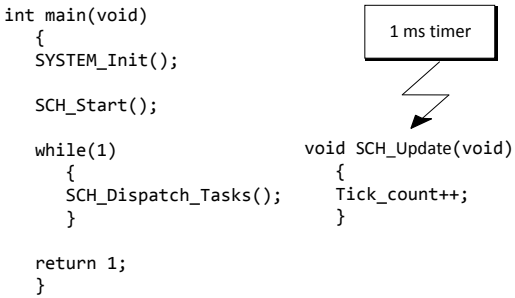


Figure 6: A schematic representation of the key components in a TTC scheduler.

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.

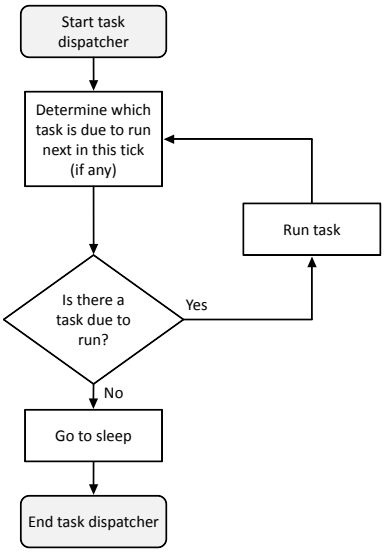


Figure 7: The operation of a task dispatcher.

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.

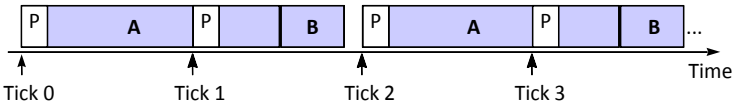


Figure 8: Executing tasks using a TTH scheduler. See text for details.

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, higher-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.

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

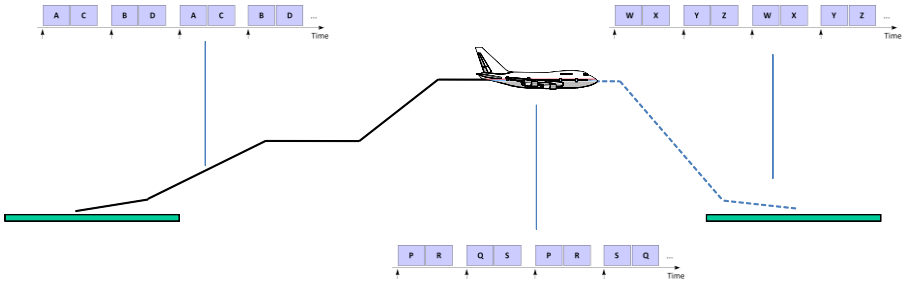


Figure 9: An example of a system with multiple operating modes.

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 xv). 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.

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

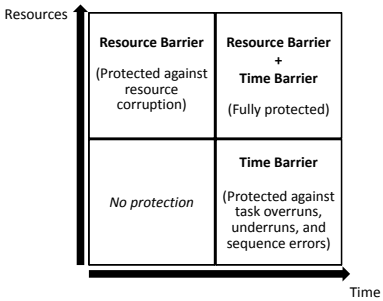


Figure 10: Run-time monitoring is employed to guard against Uncontrolled System Failures. In this book, we will check for faults that are evidenced by incorrect CPU usage (task overruns, task underruns and task sequencing) as well as faults related to other hardware resources (such as CAN controllers,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

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

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].

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

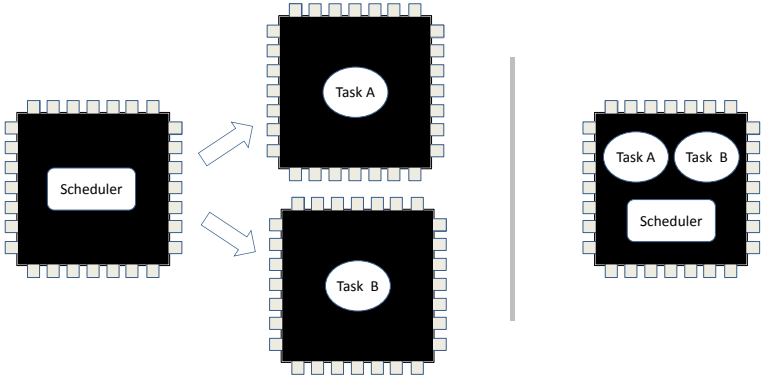


Figure 11: An "ideal" implementation of an embedded system with two tasks (left) along with a more practical implementation (right).

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 <u>any</u> 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 <u>preventable</u> – it may be <u>detectable</u>.

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 <u>evidence</u> 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 <u>evidence</u> 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 <u>all</u> 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.

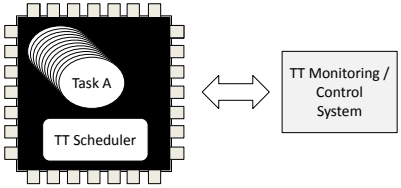


Figure 12: A schematic representation of one of the platforms that we will use in this book in order to avoid Uncontrolled System Failures.

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 "PTTES" (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.

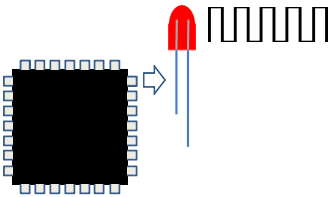


Figure 13: A schematic representation of a microcontroller running a TTC scheduler and executing a "Heartbeat" task.

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

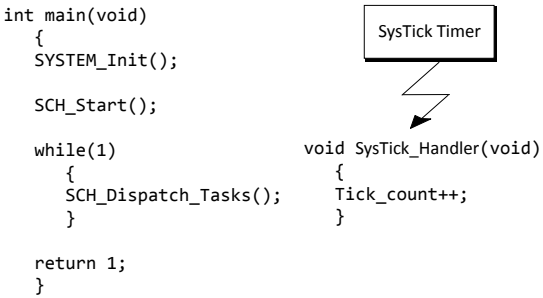


Figure 14: An overview of the structure and use of a TTC scheduler.

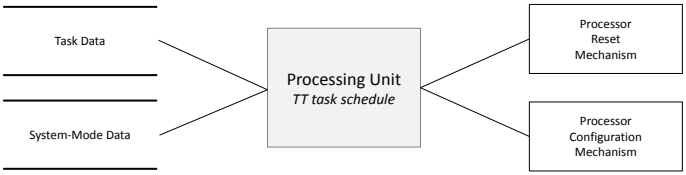


Figure 15: The foundation of the scheduler designs presented in this book.

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.

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

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.

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

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).

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

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.

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

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.

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).

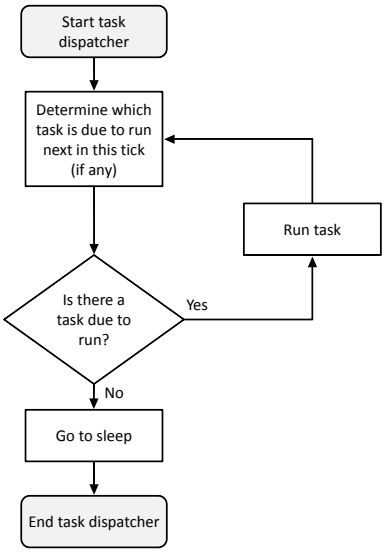


Figure 16: The operation of the dispatcher in the TTC scheduler described in this chapter.

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.

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

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).

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

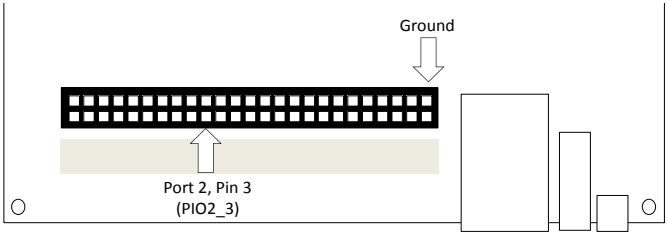


Figure 17: Jumper connections on the EA Baseboard that can be used to enable WDT support. Please refer to Appendix 1 for further information about the baseboard.

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).

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.

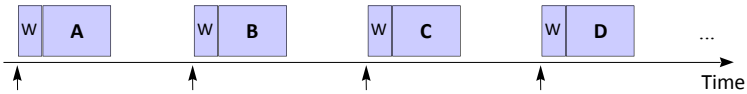


Figure 18: Running a WDT refresh task (shown as Task W) at the start of each tick interval.

```
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.

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.

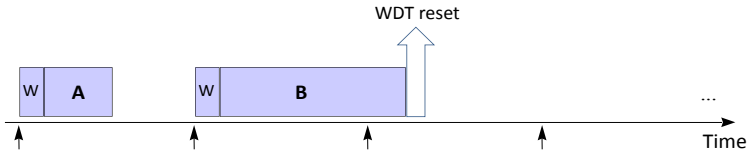


Figure 19: A WDT reset caused by a task overrun.

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

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).

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.

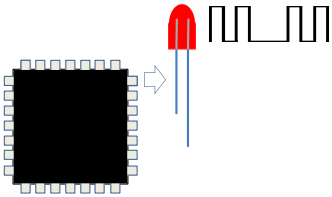


Figure 20: Output from a Fault LED (displaying "Fault Code 2").

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.

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.

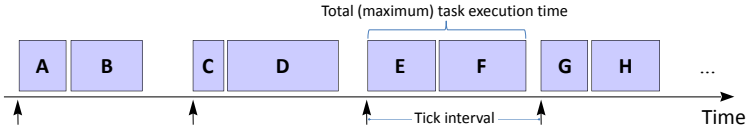


Figure 21: In most TTC designs, we expect that all tasks released in a given tick will complete their execution by the end of the tick.

```
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.

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.

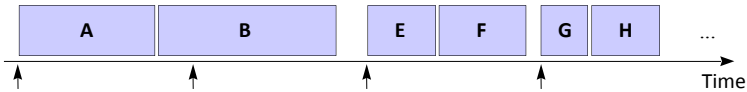


Figure 22: A system design in which there is a "theoretical" system overload.

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 example, 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 unintended 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.

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

2.19. Code listings (TTRD02a)

```
/*------------------------------------------------------------------*-

   main.h (Released 2015-01)

  ------------------------------------------------------------------

   This is the Project Header file.

-*------------------------------------------------------------------*/
#ifndef _MAIN_H
#define _MAIN_H 1

// Links to target libraries
#include <lpc17xx.h>
#include <cr_section_macros.h>
#include <NXP/crp.h>
#include <lpc17xx_gpio.h>

// Required system operating frequency (in Hz)
// Will be checked in the scheduler initialisation file
#define Required_SystemCoreClock (100000000)

// System header
#include "../system/system_1769_001-2_c02a.h"

// Scheduler header
#include "../scheduler/ttc_sch_1769_001-2_c02a.h"

// Port header
#include "../port/port_1769_001-2_c02a.h"

//------------------------------------------------------------------
// System fault codes
//------------------------------------------------------------------

// Scheduler fault codes
#define FAULT_SCH_TOO_MANY_TASKS (1)
#define FAULT_SCH_ONE_SHOT_TASK (2)

// Other fault codes may be added here, if required

//------------------------------------------------------------------
// Project constants
//------------------------------------------------------------------

#define RETURN_NORMAL 0
#define RETURN_FAULT 1

#endif
/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 1: TTRD02a (main.h).

```
/*-------------------------------------------------------------------*-

   port_1769_001-2_c02a.h (Released 2015-01)

  -------------------------------------------------------------------

   This is the "Port Header" file: it documents usage of port pins
   in the project.

-*-------------------------------------------------------------------*/
#ifndef _PORT_H
#define _PORT_H 1

// Project header
#include "../main/main.h"

// Heartbeat LED
// Connected to "LED2" on LPC1769 board
// Port 0, Pin 22
#define HEARTBEAT_LED_PORT (0)
#define HEARTBEAT_LED_PIN (0b1000000000000000000000)

// 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)

#endif

/*-------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-------------------------------------------------------------------*/
```

Listing 2: TTRD02a (port_1769_001-2_c02a.h).

```
/*------------------------------------------------------------------*-

   main.c (Released 2015-01)

  --------------------------------------------------------------------

   main file for TT project.

   See _readme.txt for project information.

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

// Project header
#include "main.h"

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

int main(void)
   {
   // Check mode, add tasks to schedule
   SYSTEM_Init();

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }

   return 1;
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 3: TTRD02a (main.c).

```
/*------------------------------------------------------------------*-

   ttc_sch_1769_001-2_c02a.h (Released 2015-01)

  --------------------------------------------------------------------

   See ttc_sch_1769_001-2_c02a.c for details.

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

#ifndef _SCHEDULER_H
#define _SCHEDULER_H 1

#include "../main/main.h"

// ------ Public data type declarations ----------------------------

// User-define 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
   // - see SCH_Add_Task() for further details
   uint32_t Delay;

   // Interval (ticks) between subsequent runs.
   // - see SCH_Add_Task() for further details
   uint32_t Period;

   // Worst-case execution time (microseconds)
   uint32_t WCET;

   // Best-case execution time (microseconds)
   uint32_t BCET;
   } sTask;
```

Listing 4: TTRD02a (ttc_sch_1769_001-2_c02a.h) [Part 1 of 2]

```
// ------ Public function prototypes -------------------------------

void     SCH_Init(const uint32_t TICKms);    // Tick interval (ms)
void     SCH_Start(void);
void     SCH_Dispatch_Tasks(void);

uint32_t SCH_Add_Task(void (* pTask)(),
                      const uint32_t DELAY,  // Offset (in ticks)
                      const uint32_t PERIOD, // Ticks
                      const uint32_t WCET,   // Microseconds
                      const uint32_t BCET    // Microseconds
                      );

// ------ Public constants ----------------------------------------

// The maximum number of tasks required at any one time
// during the execution of the program
//
// MUST BE CHECKED FOR EACH PROJECT (*not* dynamic)
#define SCH_MAX_TASKS (20)

#endif

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 4 TTRD02a (ttc_sch_1769_001-2_c02a.h) [Part 2 of 2]

```
/*------------------------------------------------------------------*-

  ttc_sch_1769_001-2_c02a.c (Released 2015-01)

  --------------------------------------------------------------------

  Time-Triggered Co-operative (TTC) task scheduler for LPC1769.

  See "ERES (LPC1769)" book (Chapter 2)
  for further information about this scheduler.

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

// Project header
#include "../main/main.h"

// ------ Public variable definitions ------------------------------

// May be used (for example) to report faults using Heartbeat LED
// See Heartbeat task (if used) for basic fault-reporting mechanism
uint32_t Fault_code_G;

// ------ Private variable definitions -----------------------------

// The array of tasks
// Check array size in scheduler header file
sTask SCH_tasks_G[SCH_MAX_TASKS];

// The current tick count
static volatile uint32_t Tick_count_G = 0;

// ------ Private function prototypes ------------------------------

static void SCH_Go_To_Sleep(void);

void SysTick_Handler(void);
```

Listing 5: TTRD02a (ttc_sch_1769_001-2_c02a.c) [Part 1 of 7]

```
/*------------------------------------------------------------------*-

  SCH_Init()

  Scheduler initialisation function.  Prepares scheduler
  data structures and sets up timer interrupts every TICKms
  milliseconds.

  You must call this function before using the scheduler.

  [Required_SystemCoreClock frequency can be found in main.h.]

-*------------------------------------------------------------------*/
void SCH_Init(const uint32_t TICKms)
   {
   uint32_t i;

   // Reset the global fault variable
   Fault_code_G = 0;

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

   // Using CMSIS

   // Must check board oscillator frequency, etc
   // - see "system_lpc17xx.c" (in linked CMSIS project)
   //
   // *If* these values have been set correctly for your hardware
   // SystemCoreClock gives the system operating frequency (in Hz)
   if (SystemCoreClock != Required_SystemCoreClock)
      {
      // Fatal fault
      SYSTEM_Perform_Safe_Shutdown();
      }

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

Listing 5: TTRD02a (ttc_sch_1769_001-2_c02a.c) [Part 2 of 7]

```
/*------------------------------------------------------------------*-

  SCH_Start()

  Starts the scheduler, by enabling SysTick interrupt.

  NOTES:
  * All tasks must be added before starting scheduler.
  * Any other interrupts MUST be synchronised to this tick.

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

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


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

  SysTick_Handler()

  [Function name determined by CMIS standard.]

  This is the scheduler ISR.  It is called at a rate
  determined by the timer settings in the SCH_Init() function.

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

Listing 5: TTRD02a (ttc_sch_1769_001-2_c02a.c) [Part 3 of 7]

```
/*------------------------------------------------------------------*-

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.

-*------------------------------------------------------------------*/
void SCH_Dispatch_Tasks(void)
   {
   uint32_t Index;
   uint32_t Update_required = 0;

   __disable_irq();  // Protect shared resource (Tick_count_G)
   if (Tick_count_G > 0)
      {
      Tick_count_G--;
      Update_required = 1;
      }
   __enable_irq();

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

               // All tasks are periodic in this design
               // - schedule task to run again
               SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
               }
            }
         }

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

   SCH_Go_To_Sleep();
   }
```

Listing 5: TTRD02a (ttc_sch_1769_001-2_c02a.c) [Part 4 of 7]

```
/*------------------------------------------------------------------*-

  SCH_Add_Task()

  Causes a task (function) to be executed at regular intervals.

  pTask  - The name of the task (function) to be scheduled.
           NOTE: All scheduled functions must be 'void, void' -
           that is, they must take no parameters, and have
           a void return type (in this design).

  DELAY  - The interval (ticks) before the task is first executed.

  PERIOD - Task period (in ticks).  Must be > 0.

  WCET   - Worst-Case Execution Time (microseconds)
           [Used only for documentation in this design.]

  BCET   - Best-Case Execution Time (microseconds)
           [Used only for documentation in this design.]

  RETURN VALUE:
  Returns the position in the task array at which the task has been
  added.  If the return value is SCH_MAX_TASKS then the task could
  not be added to the array (there was insufficient space, or the
  requested task period was 0).
  If the return value is < SCH_MAX_TASKS, then the task was added
  successfully.

  Note: this return value may be used (in later designs) to
  support the use of backup tasks.

-*------------------------------------------------------------------*/
uint32_t SCH_Add_Task(void (* pTask)(),
                      const uint32_t DELAY,
                      const uint32_t PERIOD,
                      const uint32_t WCET,
                      const uint32_t BCET
                      )
   {
   uint32_t Return_value = 0;
   uint32_t Index = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
      {
      Index++;
      }
```

Listing 5: TTRD02a (ttc_sch_1769_001-2_c02a.c) [Part 5 of 7]

```
// Have we reached the end of the list?
if (Index == SCH_MAX_TASKS)
   {
   // Task list is full
   //
   // Set the global fault variable
   Fault_code_G = FAULT_SCH_TOO_MANY_TASKS;

   // Also return a fault code
   Return_value = SCH_MAX_TASKS;
   }

// Check for "one shot" tasks
// - not permitted in this design
if (PERIOD == 0)
   {
   // Set the global fault variable
   Fault_code_G = FAULT_SCH_ONE_SHOT_TASK;

   // Also return a fault code
   Return_value = SCH_MAX_TASKS;
   }

if (Return_value != SCH_MAX_TASKS)
   {
   // If we're here, there is a space in the task array
   // and the task to be added is periodic
   SCH_tasks_G[Index].pTask  = pTask;

   SCH_tasks_G[Index].Delay  = DELAY + 1;
   SCH_tasks_G[Index].Period = PERIOD;
   SCH_tasks_G[Index].WCET   = WCET;
   SCH_tasks_G[Index].BCET   = BCET;

   Return_value = Index;
   }

return Return_value;
}
```

Listing 5: TTRD02a (ttc_sch_1769_001-2_c02a.c) [Part 6 of 7]

```
/*------------------------------------------------------------------*-

  SCH_Go_To_Sleep()

  This scheduler enters 'sleep mode' between clock ticks
  to [i] reduce tick jitter; and [ii] save power.

  The next clock tick will return the processor
  to the normal operating state.

  Note: a slight performance improvement may be achieved
  if this code is pasted into the 'Dispatch' function
  (this may be at the cost of code readability & portability)

  *** May be able to make further improvements to the jitter
  *** behaviour depending on the target hardware architecture

  *** Various power-saving options can be added
  *** (e.g. shut down unused peripherals)

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

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 5: TTRD02a (ttc_sch_1769_001-2_c02a.c) [Part 7 of 7]

```
/*------------------------------------------------------------------*-

   system_1769_001-2_c02a.h (Released 2015-01)

  ------------------------------------------------------------------

   See system_1769_001-2_c02a.c for details.

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

#ifndef _SYSTEM_H
#define _SYSTEM_H 1

// Two possible system modes
typedef enum {FAIL_SILENT, NORMAL} eSystem_mode;

// ------ Public function prototypes -------------------------------

void SYSTEM_Init(void);
void SYSTEM_Perform_Safe_Shutdown(void);

#endif

/*------------------------------------------------------------------*-
  ---- END OF FILE ----------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 6: TTRD02a (system_1769_001-2_c02a.h)

```
/*------------------------------------------------------------------*-

  system_1769_001-2_c02a.c (Released 2015-01)

  Controls system configuration after processor reset.

  [Two modes supported - "Normal" and "Fail Silent".]

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

// Project header
#include "../main/main.h"

// Task headers
#include "../tasks/heartbeat_1769_001-2_c02a.h"
#include "../tasks/watchdog_1769_001-2_c02a.h"

// ------ Public variable definitions ------------------------------

// In many designs, System_mode_G will be used in other modules.
// - we therefore make this variable public.
eSystem_mode System_mode_G;

// ------ Private function declarations ----------------------------

void SYSTEM_Identify_Required_Mode(void);
void SYSTEM_Configure_Required_Mode(void);

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

  SYSTEM_Init()

  Wrapper for system startup functions.

-*------------------------------------------------------------------*/
void SYSTEM_Init(void)
   {
   SYSTEM_Identify_Required_Mode();
   SYSTEM_Configure_Required_Mode();
   }
```

Listing 7: TTRD02a (system_1769_001-2_c02a.c) [Part 1 of 4]

```
/*------------------------------------------------------------------*-

  SYSTEM_Identify_Required_Mode()

  Try to work out the cause of the system reset.
  Set the system mode accordingly.

-*------------------------------------------------------------------*/
void SYSTEM_Identify_Required_Mode(void)
   {
   // If "1", reset was caused by WDT
   uint32_t WDT_flag = (LPC_SC->RSID >> 2) & 1;

   if (WDT_flag == 1)
      {
      // Cleared only by software or POR
      // Clear flag (or other resets may be interpreted as WDT)
      LPC_SC->RSID &= ~(0x04);

      // Set system mode (Fail Silent)
      System_mode_G = FAIL_SILENT;
      }
   else
      {
      // Here we treat all other forms of reset in the same way
      // Set system mode (Normal)
      System_mode_G = NORMAL;
      }
   }
```

Listing 7: TTRD02a (system_1769_001-2_c02a.c) [Part 2 of 4]

```
/*------------------------------------------------------------------*-

  SYSTEM_Configure_Required_Mode()

  Configure the system in the required mode.

-*------------------------------------------------------------------*/
void SYSTEM_Configure_Required_Mode(void)
   {
   switch (System_mode_G)
      {
      case default:    // Default to "FAIL_SILENT"
      case FAIL_SILENT:
         {
         // Reset caused by WDT
         // Trigger "fail silent" behaviour
         SYSTEM_Perform_Safe_Shutdown();

         break;
         }

      case NORMAL:
         {
         // Set up WDT (timeout in *microseconds*)
         WATCHDOG_Init(1100);

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

         // Prepare for Heartbeat task
         HEARTBEAT_Init();

         // Add tasks to schedule.
         // Parameters are:
         // 1. Task name
         // 2. Initial delay / offset (in ticks)
         // 3. Task period (in ticks): Must be > 0
         // 4. Task WCET (in microseconds)
         // 5. Task BCET (in microseconds)

         // Add watchdog task first
         SCH_Add_Task(WATCHDOG_Update, 0, 1, 10, 0);

         // Add Heartbeat task
         SCH_Add_Task(HEARTBEAT_Update, 0, 1000, 20, 0);

         // Feed the watchdog
         WATCHDOG_Update();

         break;
         }
      }
   }
```

Listing 7: TTRD02a (system_1769_001-2_c02a.c) [Part 3 of 4]

```
/*------------------------------------------------------------------*-

  SYSTEM_Perform_Safe_Shutdown()

  Attempt to place the system into a safe state.

  Note: Does not return and may (if watchdog is operational) result
  in a processor reset, after which the function may be called again.

  [The rationale for this behaviour is that - after the reset -
   the system MAY be in a better position to enter a safe state.
   To avoid the possible reset, adapt the code and feed the WDT
   in the loop.]

-*------------------------------------------------------------------*/
void SYSTEM_Perform_Safe_Shutdown(void)
   {
   // Used for simple fault reporting
   uint32_t Delay, j;

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

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

   HEARTBEAT_Init();

   while(1)
      {
      // Flicker Heartbeat LED to indicate fault
      for (Delay = 0; Delay < 200000; Delay++) j *= 3;
      HEARTBEAT_Update();
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 7: TTRD02a (system_1769_001-2_c02a.c) [Part 4 of 4]

```
/*------------------------------------------------------------------*-

   heartbeat_1769_001-2_c02a.h (Released 2015-01)

  ------------------------------------------------------------------

   - See heartbeat_1769_001-2_c02a.c for details.

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

#ifndef _HEARTBEAT_H
#define _HEARTBEAT_H 1

// ------ Public function prototypes ------------------------------

void HEARTBEAT_Init(void);
void HEARTBEAT_Update(void);

#endif

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 8: TTRD02a (heartbeat_1769_001-2_c02a.h)

```
/*------------------------------------------------------------------*-

  heartbeat_1769_001-2_c02a.c (Released 2015-01)

  ------------------------------------------------------------------

  Simple 'Heartbeat' task for LPC1769.

  If everything is OK, flashes at 0.5 Hz

  If there is a fault code active, this is displayed.

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

// Project header
#include "../main/main.h"

// Task header
#include "heartbeat_1769_001-2_c02a.h"

// ------ Public variable declarations -----------------------------

// See scheduler for definition
extern uint32_t Fault_code_G;

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

  HEARTBEAT_Init()

  Prepare for HEARTBEAT_Update() function - see below.

-*------------------------------------------------------------------*/
void HEARTBEAT_Init(void)
   {
   // Set up LED2 as an output pin
   // Params: Port : Pin : 1 for o/p, 0 for i/p
   GPIO_SetDir(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN, 1);
   }
```

Listing 9: TTRD02a (heartbeat_1769_001-2_c02a.c) [Part 1 of 3]

```
/*------------------------------------------------------------------*-

  HEARTBEAT_Update()

  Flashes at 0.5 Hz if fault code is 0 (i.e. no fault code).

  Otherwise, displays fault code.

  Must schedule every second (soft deadline).

-*------------------------------------------------------------------*/
void HEARTBEAT_Update(void)
   {
   static uint32_t Heartbeat_state = 0;
   static uint32_t Fault_state = 0;

   if (Fault_code_G == 0)
      {
      // No faults recorded
      // - just flash at 0.5 Hz

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

Listing 9: TTRD02a (heartbeat_1769_001-2_c02a.h) [Part 2 of 3]

```
   else
      {
      // If we are here, there is a (non-zero) fault code ...
      Fault_state++;

      if (Fault_state < Fault_code_G*2)
         {
         Heartbeat_state = 0;
         GPIO_ClearValue(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
         }
      else
         {
         if (Fault_state < Fault_code_G*4)
            {
            // Change the LED from OFF to ON (or vice versa)
            if (Heartbeat_state == 1)
               {
               Heartbeat_state = 0;
               GPIO_ClearValue(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
               }
            else
               {
               Heartbeat_state = 1;
               GPIO_SetValue(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
               }
            }
         else
            {
            Fault_state = 0;
            }
         }
      }
   }

/*-----------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-----------------------------------------------------------------*/
```

Listing 9: TTRD02a (heartbeat_1769_001-2_c02a.h) [Part 3 of 3]

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

```
/*------------------------------------------------------------------*-

   watchdog_1769_001-2_c02a.h (Released 2015-01)

  ------------------------------------------------------------------

   - See watchdog_1769_001-2_c02a.c for details.

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

#ifndef _WDT_H
#define _WDT_H 1

#include <lpc17xx_wdt.h>

// ------ Public constants ----------------------------------------

// From NXP

/** Define divider index for microsecond ( us ) */
#define WDT_US_INDEX ((uint32_t)(1000000))

/** WDT Time out minimum value */
#define WDT_TIMEOUT_MIN ((uint32_t)(0xFF))

/** WDT Time out maximum value */
#define WDT_TIMEOUT_MAX ((uint32_t)(0xFFFFFFFF))

// Jumper connections
// WDT will only be enabled if jumper is inserted
// (connecting the jumper pin to ground)
// - see Port Header for jumper pin details.
#define WDT_JUMPER_INSERTED (0)
#define WDT_PCLK (4000000)

// ------ Public function prototypes ------------------------------

void WATCHDOG_Init(const uint32_t);
void WATCHDOG_Update(void);

#endif

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 10: TTRD02a (watchdog_1769_001-2_c02a.h)

```
/*------------------------------------------------------------------*-

  watchdog_1769_001-2_c02a.c (Released 2015-01)

  ------------------------------------------------------------------

  'Watchdog' library for LPC1769.

  ** Jumper controlled (see below) **

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

// Project header
#include "../main/main.h"

// Task header
#include "watchdog_1769_001-2_c02a.h"

// ------ Public variable declarations -----------------------------

// See scheduler module for definition
extern uint32_t Fault_code_G;


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

  WATCHDOG_Init()

  Set up watchdog timer on LPC1769.

  ******************************************************************
  * Handle with care - if WDT is running, debug links may be lost *
  * In this design, WDT is enable only when jumper is inserted.   *
  ******************************************************************

  The watchdog timer is driven by the Internal RC Oscillator:
  the minimum available timeout is 256 µs.

-*------------------------------------------------------------------*/
```

Listing 11: TTRD02a (watchdog_1769_001-2_c02a.c) [Part 1 of 3]

```
void WATCHDOG_Init(const uint32_t WDT_TIMEOUTus)
   {
   uint32_t wdt_ticks = 0;
   uint32_t Jumper_input;

   // *If* WDT jumper is in place, we start the WDT

   // Read WDT jumper setting
   // - set up jumper pin for input
   // - params: Port : Pin : 1 for o/p, 0 for i/p
   GPIO_SetDir(WDT_JUMPER_PORT, WDT_JUMPER_PIN, 0);

   // Note: we only read the jumper during system init phase
   Jumper_input = (GPIO_ReadValue(WDT_JUMPER_PORT) & WDT_JUMPER_PIN);

   if (Jumper_input != WDT_JUMPER_INSERTED)
      {
      // Jumper not inserted - don't enable WDT
      return;
      }

   // If we are here, we are setting up the WDT

   // Drive WDT from internal RC timer (IRC)
   LPC_WDT->WDCLKSEL = 0x00;

   // Calculate required tick count for WDT timeout
   wdt_ticks  = (((WDT_PCLK) / WDT_US_INDEX) * (WDT_TIMEOUTus / 4));

   // Check if tick count is within allowed range
   if ((wdt_ticks >= WDT_TIMEOUT_MIN) && (wdt_ticks <= WDT_TIMEOUT_MAX))
      {
      LPC_WDT->WDTC = wdt_ticks;
      }
   else
      {
      // We simply "stop" if WDT values are wrong
      // - other solutions may make sense for your application
      // - for example, use closest available timeout.
      while(1);
      }

   // Reset if WDT overflows
   LPC_WDT->WDMOD |= 0x02;

   // Start WDT
   LPC_WDT->WDMOD |= 0x01;

   // Feed watchdog
   WATCHDOG_Update();
   }
```

Listing 11: TTRD02a (watchdog_1769_001-2_c02a.c) [Part 2 of 3]

```
/*------------------------------------------------------------------*-

  WATCHDOG_Update()

  Feed the watchdog timer.

  See Watchdog_Init() for further information.

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

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 11: TTRD02a (watchdog_1769_001-2_c02a.c) [Part 3 of 3]

2.20. Code listings (TTRD02b)

```
/*------------------------------------------------------------------*-

  HEARTBEAT_Update()

  Flashes at 0.5 Hz if fault code is 0 (i.e. no fault code).

  Otherwise, displays fault code.

  * Incorporates fault injection (task overrun) after 15 secs *

  Must schedule every second (soft deadline).

-*------------------------------------------------------------------*/
void HEARTBEAT_Update(void)
   {
   static uint32_t Heartbeat_state = 0;
   static uint32_t Fault_state = 0;

   uint32_t Delay, j;
   static uint32_t Task_overrun_counter = 0;

   // Force task overrun after 15 seconds (test / demo purposes)
   if (Task_overrun_counter++ == 15)
      {
      Task_overrun_counter = 0;

      // Trigger temporary task overrun (for demo purposes)
      // This gives delay of ~3.6 ms
      for (Delay = 0; Delay < 20000; Delay++)
         {
         j *= 3;
         }
      }

   if (Fault_code_G == 0)
      {
      // No faults recorded
      // - just flash at 0.5 Hz

      // Remaining code omitted here

...

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 12: TTRD02b (extract from file heartbeat_1769_001-2_c02b.c)

2.21. Code listings (TTRD02c)

```
void SCH_Init(const uint32_t TICKmicroseconds)
   {
   // Used to configure Timer 0
   TIM_TIMERCFG_Type TMR0_Cfg;
   TIM_MATCHCFG_Type TMR0_Match;

   // Code omitted here

   // Power-on Timer 0
   LPC_SC->PCONP |= 1 << 1;

   // Initialise Timer 0, prescale counter = 1 µs
   TMR0_Cfg.PrescaleOption = TIM_PRESCALE_USVAL;
   TMR0_Cfg.PrescaleValue = 1;

   // Use channel 0, MR0
   TMR0_Match.MatchChannel = 0;

   // Enable interrupt when MR0 matches the value in TC register
   TMR0_Match.IntOnMatch = ENABLE;

   // Enable reset on MR0: TIMER will reset if MR0 matches it
   TMR0_Match.ResetOnMatch = TRUE;

   // Don't stop on MR0 if MR0 matches it
   TMR0_Match.StopOnMatch = FALSE;

   // Do nothing for external output pin if match
   TMR0_Match.ExtMatchOutputType = TIM_EXTMATCH_NOTHING;

   // Tick value
   // Set Match value, count value in microseconds in this version.
   TMR0_Match.MatchValue = TICKmicroseconds;

   // Set configuration for Tim_config and Tim_MatchConfig
   TIM_Init(LPC_TIM0, TIM_TIMER_MODE, &TMR0_Cfg);
   TIM_ConfigMatch(LPC_TIM0, &TMR0_Match);

   // Highest priority = Timer 0
   NVIC_SetPriority(TIMER0_IRQn, 0);
   }
```

Listing 13: TTRD02c (extract from file ttc_sch_od_1769_001-2_c02c.c)

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

CHAPTER 3: Initial case study

In this chapter we present an introductory case study.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD03a – Simple framework for washing-machine controller

3.1. Introduction

As noted in the Preface, the goal of 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 System Faults (as defined on Page xv) and then handle them in an appropriate manner, thereby avoiding Uncontrolled System Failures.

Before we introduce the model-based development process, we will present a simple case study in this chapter.

3.2. The focus of this case study

The examples in this book target the LPC1769 microcontroller. As noted in the Preface, this processor is intended for use in applications such as: industrial networking; motor control; white goods; eMetering; alarm systems; and lighting control.

This case study will focus on the use of the LPC1769 in a controller for a domestic washing machine. This is an appropriate application for this microcontroller. However, the TT software architecture that is employed is generic in nature and can also be employed in many other systems.

3.3. The purpose of this case study

We will present a simple TT framework for a control system in this chapter. In subsequent chapters of this book, we will present a range of techniques that will allow us to make this framework much more robust in the presence of various potential threats (such as EMI).

While the framework created here could form the foundation for a reliable embedded system, the initial design presented in this chapter is far from complete and is best viewed as an early system prototype.

We will revisit this study in Chapter 15 and apply the techniques described in the remainder of this book in order to improve the system reliability.

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

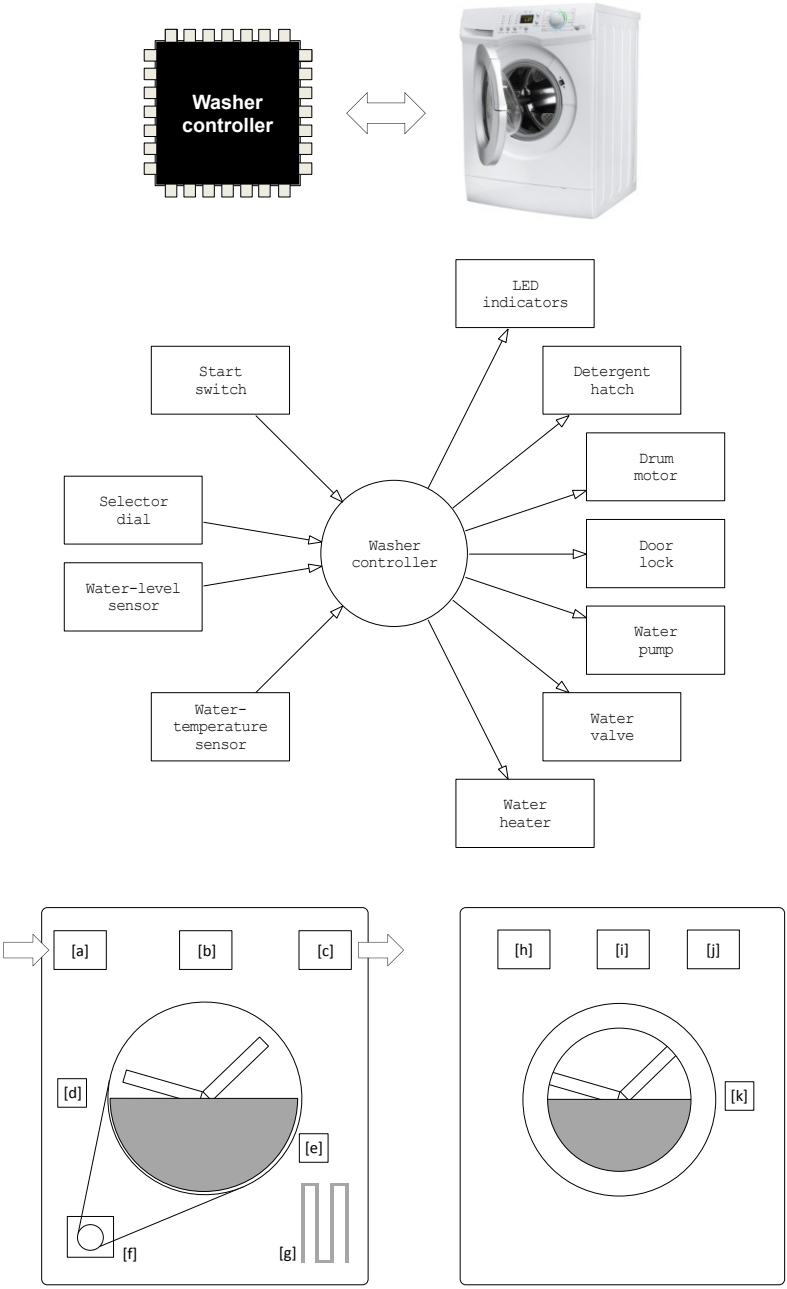


Figure 23: An overview of the components in the washing machine (TTRD3a). Internal view of the system (bottom left): [a] water valve; [b] detergent hatch; [c] water pump; [d] water-level sensor; [e] water-temperature sensor; [f] drum motor; [g] water heater. External view of the system (bottom right): [h] selector dial; [i] LED indicators; [j] start switch; [k] door lock.

3.4. A summary of the required system operation

An overview of the sensor and actuator components in the washer control system is given in Figure 23.

This is a brief summary of the required system operation:

1. The user selects a wash program (e.g. 'Cotton') on the selector dial.
2. The user presses the 'Start' switch.
3. The door lock is engaged.
4. The water valve is opened to allow water into the wash drum.
5. If the wash program involves detergent, the detergent hatch is opened. When the detergent has been released, the detergent hatch is closed.
6. When the 'full water level' is sensed, the water valve is closed.
7. If the program involves warm water, the water heater is switched on.
8. When the water reaches the correct temperature, the water heater is switched off.
9. The washer motor is turned on to rotate the drum. The motor then goes through a series of movements (at various speeds) to wash the clothes. (The precise set of movements carried out depends on the wash program that the user has selected).
10. At the end of the wash cycle, the motor is stopped.
11. The pump is switched on to drain the drum.
12. When the drum is empty, the pump is switched off.
13. The door lock is released.
14. During the operation various LEDs are used to indicate where the system is in the wash cycle.

The description is simplified for the purposes of this example, but it will be adequate for our purposes here.

3.5. The system architecture

Our design consists of a "system" task, plus one module for each component on the context diagram shown in Figure 23.

The system task stores and updates the current state. It obtains information from the various sensor tasks and controls the system operation by means of the various actuator tasks.

The underlying architecture is widely applicable. For example, the sequence of events used to raise the landing gear in a passenger aircraft can be controlled in a similar manner. In this case, basic tests (such as 'WoW' – 'Weight on Wheels') will be used to determine whether the aircraft is on the ground or in the air: these tests will be completed before the operation begins. Feedback from various door and landing-gear sensors will then be used to ensure that each phase of the manoeuvre completes correctly.

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

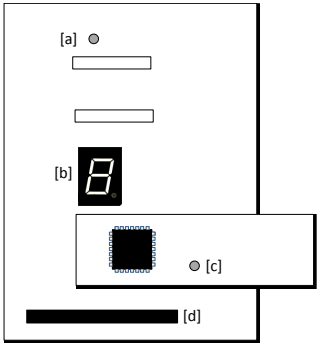


Figure 24: User-interface elements in the washer prototype: [a] RGB LED cluster indicating the state of the motor and door lock [b] 7-segment display indicating the overall system state; [c] the "Heartbeat" LED; [d] connections needed for the WDT (as detailed in Chapter 2).

3.6. The system states

The possible system states are described in Table 2.

The various states are implemented by means of a switch statement in the main "washer controller" task (see Section 3.8).

3.7. Implementation platform for the prototype

The prototype system described in this chapter is designed to run on the EA Baseboard: this is described in Appendix 1.

The key user-interface elements in the prototype are illustrated schematically in Figure 24.

In this design, the overall state of the system is illustrated by the 7-segment LED. For example – as noted in the left column in Table 2 - "2" will be displayed in the FILL_DRUM state, and "E" will be displayed in the FAULT state.

The state of the motor is indicated by the blue LED in the RGB LED cluster. The green LED in this cluster is on (and the red LED is off) when the door is unlocked; the red LED is on (and the green LED is off) when the door is locked.

There is a standard "Heartbeat" LED, and the WDT support requires a jumper to be inserted (as detailed in Chapter 2, Section 2.11).

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

Table 2: Possible system states for the washing-machine controller.

State	Description
INIT [0]	The system enters INIT state when it is powered up. It remains in this state until the Start switch is pressed. Once the Start switch is pressed, the system reads the selector dial: the system then moves into START state.
START [1]	In START state, the door is locked, and the water valve is opened to begin filling the drum with cold water. If detergent is required (depending on the reading from the selector dial) then this will also be released into the drum. The system then enters FILL_DRUM state.
FILL_DRUM [2]	In FILL_DRUM state, the system checks the water-level sensor periodically, to see if the drum is full. If it takes longer than the MAX_FILL_DURATION to fill the drum, then something is wrong (for example, there is no water supply connected): the system then enters FAULT state. If the drum has filled in time, then the water heater may be switched on (if hot water is required in the selected wash routine). If hot water is required, the system will move into HEAT_WATER state. If hot water is not required, the system will move into PRE_WASH state.
HEAT_WATER [3]	In HEAT_WATER state, the system checks the water-temperature sensor periodically, to see if the water has reached the required temperature. If it takes longer than the MAX_WATER_HEAT_DURATION to heat the water, then something is wrong (for example, the water heater has failed): the system then enters FAULT state. If the water reaches the required temperature within the time limit then the system will move into PRE_WASH state.
PRE_WASH [4]	In the PRE_WASH state, the drum motor is activated in order to rotate the drum. In our simplified design, this will simply involve running the motor (constantly) at "medium speed". The system remains in the pre-wash state for a period equal to the PRE_WASH_DURATION. It then moves into the MAIN_WASH state.
MAIN_WASH [5]	Like the PRE_WASH state, the MAIN_WASH state involves turning the drum (in this case at a –medium-high speed). The system remains in this state for a period equal to the MAIN_WASH_DURATION. The system then moves into DRAIN_DRUM state.
DRAIN_DRUM [6]	In DRAIN_DRUM state, the water pump is activated: this begins the process of removing water from the drum. If draining the drum takes longer than DRAIN_DRUM_DURATION, the system moves into FAULT state, otherwise – after the drum has been drained – the system move into SPIN state.
SPIN [7]	In SPIN state, the drum is rotated at full speed. The system remains in this state for a period equal to the SPIN_DURATION. The system then moves into FINISHED state.
FINISHED [-]	In FINISHED state, the various system components are switched off. To avoid any risk of injury when the door is opened (just in case the drum is still rotating), there is a delay (of 10 seconds) before the door lock is released. The system then returns to INIT state, ready for the next wash.
FAULT [F]	In the event that a fault is detected, the system enters the FAULT state. All components are switched off, and there is a delay before the door is unlocked. The system remains (indefinitely) in the fault state.

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

3.8. The “system” task

The system task is shown in Listing 15.

The possible system states are implemented by means of an enumerated type (Code Fragment 11).

```
// Possible system states
typedef enum {INIT, START, FILL_DRUM, HEAT_WATER, PRE_WASH,
              MAIN_WASH, DRAIN_DRUM, SPIN, FINISHED, FAULT}
              eSystem_state;
```

Code Fragment 11: The possible washer states.

Just in case a value outside the allowed range is assigned to the System_state_G variable, we have a default state (which is FAIL_SILENT): see Code Fragment 12.

```
      switch (System_state_G)
         {
         case INIT:  // 0
            {
            }
...

         default:    // Default state
         case FAULT: // E
            {
            }
         }
```

Code Fragment 12: The washer default state.

Please note that use of default states is a requirement in MISRA C.

3.9. The “selector dial” task

In the final system, the selector-dial task will read an input from a switch something like that shown in Figure 25. In our demo system, we use a simple placeholder task.

Figure 25: A typical selector dial from a domestic washing machine.

3.10. The “start switch” task

In our demonstration system, we use “SW3” on the EA Baseboard as the start switch.

3.11. The “door lock” task

In our demonstration system, the door lock is representation by the state of two LEDs (as detailed in Section 3.7).

3.12. The “water valve” task

A simple placeholder task is used for the water-valve interface in this demo.

3.13. The “detergent hatch” task

A simple placeholder task is used for the detergent-hatch interface in this demo.

3.14. The “water level” task

A simple model is used to implement the water-level task in the demo system (Code Fragment 13).

3.15. The “water heater” task

A simple placeholder task is used for the water-heater interface in this demo.

3.16. The “water temperature” task

Another simple model is used to implement the water-temperature task in this demo system.

3.17. The “drum motor” task

In our demonstration system, the drum-motor state is representation by the state of the blue LED on the EA Baseboard (as detailed in Section 3.7).

3.18. The “water pump” task

A simple placeholder task is used for the water-pump interface in this demo.

3.19. The Heartbeat task

A standard Heartbeat LED is used in the demo (please refer again to Section 3.7 for connection details).

```
void WATER_LEVEL_Update(void)
   {
   // Placeholder code
   static uint32_t Water_level = 0;

   // Is the drum filling?
   if ((Water_valve_required_state_G == OPEN) && !WATER_VALVE_FAULT)
      {
      // Drum is filling
      Water_level++;

      // Is the drum full of water (ready to wash)?
      if (Water_level >= 50)
         {
         Water_level_reading_G = 1;
         }
      }

   // Is the drum being drained?
   if ((Water_pump_required_state_G == ON) && !WATER_PUMP_FAULT)
      {
      // Drum is being drained
      if (Water_level > 0)
         {
         // Drum is draining
         Water_level--;
         }

      if (Water_level == 0)
         {
         // Drum is empty
         Water_level_reading_G = 0;
         }
      }
   }
```

Code Fragment 13: The water-level task, based on a simple model.

3.20. Communication between tasks

Communication between tasks is carried out by means of global variables.

For example, the required state of the water pump is represented as shown in Code Fragment 14.

```
// ------ Public variable definitions -------------------------------

uint32_t Water_pump_required_state_G;
```

Code Fragment 14: The global variable used to control the state of the water pump.

3.21. Where do we go from here?

In this case study we have illustrated how a set of "placeholder" tasks and a TT scheduler can be employed in order to develop a framework for a non-trivial control system.

We emphasise again that – while the framework created here could form the starting point for a reliable embedded system - the result so far (as detailed in full in TTRD03a) is far from complete and is best viewed as an early system prototype.

We will build on this foundation in later chapters, following a three-stage development process:

- The first stage will involve modelling the system timing characteristics: we will begin to explore this process in Chapter 4.
- The second stage will involve building the system using appropriate task designs (Chapter 6), along with a TT scheduler that has support for "Task Contracts" (Chapter 8 and Chapter 9) and – if required – support for task pre-emption (Chapter 11 and Chapter 12).
- The third stage will involve adding comprehensive support for run-time monitoring: we begin to see how this can be achieved in Chapter 8.

The end result will meet our requirements for a reliable embedded system and will be applicable in a wide range of sectors.

As far as this particular study is concerned, two of our goals in the second version of this system (presented in Chapter 15) will be to ensure that: [i] the door of our washing machine cannot be opened while the drum is spinning, an event that would risk injury to users of the product; [ii] the door can be opened – without causing a flood – in circumstances where it is realised that a piece of coloured clothing has been inadvertently included in a white wash, or that a toddler has placed the family cat in the machine with the washing.

It should be emphasised that very similar challenges apply in other sectors. For example, in Chapter 1 (Section 1.8), we discussed a system that is intended to engage the steering-column lock in a passenger vehicle (to secure the vehicle against theft) after the driver has parked and left the vehicle. In this system we need to ensure (for security reasons) that this lock is engaged when the vehicle is parked: however, we also need to ensure (for safety reasons) that this lock can <u>never</u> be engaged when the vehicle is moving.

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

3.22. Conclusions

In this chapter, we have illustrated how a TTC scheduler can be used with an initial set of periodic "placeholder" tasks in order to create a prototype for an embedded control system.

In Chapter 4, we will begin to explore techniques for modelling sets of periodic tasks in order to determine (for example) precisely how long it will take our system to respond to external events.

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

3.23. Code listings (TTRD03a)

```
/*------------------------------------------------------------------*-

  system_1769_001-2_c03a.c (Released 2015-01)

  Controls system configuration after processor reset.

  [Two modes supported - "Normal" and "Fail Silent".]

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

// Project header
#include "../main/main.h"

// Task headers
#include "../tasks/detergent_hatch_1769_001-2_c03a.h"
#include "../tasks/door_lock_1769_001-2_c03a.h"
#include "../tasks/drum_motor_1769_001-2_c03a.h"
#include "../tasks/selector_dial_1769_001-2_c03a.h"
#include "../tasks/start_switch_1769_001-2_c03a.h"
#include "../tasks/washer_controller_1769_001-2_c03a.h"
#include "../tasks/water_heater_1769_001-2_c03a.h"
#include "../tasks/water_level_1769_001-2_c03a.h"
#include "../tasks/water_pump_1769_001-2_c03a.h"
#include "../tasks/water_temperature_1769_001-2_c03a.h"
#include "../tasks/water_valve_1769_001-2_c03a.h"

#include "../tasks/heartbeat_1769_001-2_c02a.h"
#include "../tasks/watchdog_1769_001-2_c02a.h"

// ------ Public variable definitions ------------------------------

// In many designs, System_mode_G will be used in other modules.
// - we therefore make this variable public.
eSystem_mode System_mode_G;

// ------ Public variable declarations ----------------------------

// Actuators
extern uint32_t Drum_motor_required_state_G;
extern uint32_t Drum_motor_required_speed_G;
extern uint32_t Door_lock_required_state_G;
extern uint32_t Detergent_hatch_required_state_G;
extern uint32_t Water_heater_required_state_G;
extern uint32_t Water_pump_required_state_G;
extern uint32_t Water_valve_required_state_G;

// ------ Private function declarations ---------------------------

void SYSTEM_Identify_Required_Mode(void);
void SYSTEM_Configure_Required_Mode(void);
```

Listing 14: TTRD03a (system_1769_001-2_c03a.c) [Part 1 of 5]

```
/*------------------------------------------------------------------*-

  SYSTEM_Init()

  Wrapper for system startup functions.

-*------------------------------------------------------------------*/
void SYSTEM_Init(void)
   {
   SYSTEM_Identify_Required_Mode();
   SYSTEM_Configure_Required_Mode();
   }

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

  SYSTEM_Identify_Required_Mode()

  Try to work out the cause of the system reset.
  Set the system mode accordingly.

-*------------------------------------------------------------------*/
void SYSTEM_Identify_Required_Mode(void)
   {
   // If "1", reset was caused by WDT
   uint32_t WDT_flag = (LPC_SC->RSID >> 2) & 1;

   if (WDT_flag == 1)
      {
      // Cleared only by software or POR
      // Clear flag (or other resets may be interpreted as WDT)
      LPC_SC->RSID &= ~(0x04);

      // Set system mode (Fail Silent)
      System_mode_G = FAIL_SILENT;
      }
   else
      {
      // Here we treat all other forms of reset in the same way
      // Set system mode (Normal)
      System_mode_G = NORMAL;
      }
   }
```

Listing 14: TTRD03a (system_1769_001-2_c03a.c) [Part 2 of 5]

```
/*------------------------------------------------------------------*-

  SYSTEM_Configure_Required_Mode()

  Configure the system in the required mode.

-*------------------------------------------------------------------*/
void SYSTEM_Configure_Required_Mode(void)
   {
   switch (System_mode_G)
      {
      case default:    // Default to "FAIL_SILENT"
      case FAIL_SILENT:
         {
         // Reset caused by WDT
         // Trigger "fail silent" behaviour
         SYSTEM_Perform_Safe_Shutdown();

         break;
         }
```

Listing 14: TTRD03a (system_1769_001-2_c03a.c) [Part 3 of 5]

```
case NORMAL:
   {
   // Set up WDT (timeout in *microseconds*)
   WATCHDOG_Init(1100);

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

   // Prepare for Heartbeat task
   HEARTBEAT_Init();

   // Prepare to read START switch
   SWITCH_SW3_Init();

   // Prepare for main washer task
   WASHER_CONTROLLER_Init();

   // Prepare for other tasks
   DETERGENT_HATCH_Init();
   DOOR_LOCK_Init();
   DRUM_MOTOR_Init();
   SELECTOR_DIAL_Init();
   WATER_HEATER_Init();
   WATER_LEVEL_Init();
   WATER_PUMP_Init();
   WATER_TEMPERATURE_Init();
   WATER_VALVE_Init();

   // Add tasks to schedule.
   // Parameters are:
   // 1. Task name
   // 2. Initial delay / offset (in ticks)
   // 3. Task period (in ticks) - set to 0 for "one shot" task
   // 4. Task WCET (in microseconds)
   // 5. Task BCET (in microseconds)

   // Add watchdog task first
   SCH_Add_Task(WATCHDOG_Update, 0, 1, 10, 0);

   // Add task to control door lock
   SCH_Add_Task(DOOR_LOCK_Update, 0, 10, 100, 0);

   // Add tasks to read selector dial, then START switch
   SCH_Add_Task(SELECTOR_DIAL_Update, 0, 10, 100, 0);
   SCH_Add_Task(SWITCH_SW3_Update, 0, 10, 100, 0);

   // Remaining tasks called less frequently
   SCH_Add_Task(DETERGENT_HATCH_Update, 1, 100, 100, 0);
   SCH_Add_Task(DRUM_MOTOR_Update, 2, 100, 100, 0);
   SCH_Add_Task(WATER_HEATER_Update, 3, 100, 100, 0);
   SCH_Add_Task(WATER_LEVEL_Update, 4, 100, 100, 0);
   SCH_Add_Task(WATER_PUMP_Update, 5, 100, 100, 0);
   SCH_Add_Task(WATER_TEMPERATURE_Update, 6, 100, 100, 0);
   SCH_Add_Task(WATER_VALVE_Update, 7, 100, 100, 0);
```

Listing 14: TTRD03a (system_1769_001-2_c03a.c) [Part 4 of 5]

```
         // Add main washer task
         SCH_Add_Task(WASHER_CONTROLLER_Update, 0, 1000, 100, 0);

         // Add Heartbeat task
         SCH_Add_Task(HEARTBEAT_Update, 0, 1000, 20, 0);

         // Feed the watchdog
         WATCHDOG_Update();

         break;
         }
      }
   }

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

  SYSTEM_Perform_Safe_Shutdown()

  Attempt to place the system into a safe state.

  Note: Does not return and may (if watchdog is operational) result
  in a processor reset, after which the function may be called again.

  [The rationale for this behaviour is that - after the reset -
   the system MAY be in a better position to enter a safe state.
   To avoid the possible reset, adapt the code and feed the WDT
   in the loop.]

-*------------------------------------------------------------------*/
void SYSTEM_Perform_Safe_Shutdown(void)
   {
   // Used for simple fault reporting
   uint32_t Delay, j;

   // Here we simply "fail silent" with rudimentary fault reporting
   // Other behaviour may make more sense in your design

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

   HEARTBEAT_Init();

   while(1)
      {
      // Flicker Heartbeat LED to indicate fault
      for (Delay = 0; Delay < 200000; Delay++) j *= 3;
      HEARTBEAT_Update();
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 14: TTRD03a (system_1769_001-2_c03a.c) [Part 5 of 5]

```
/*-----------------------------------------------------------------------*-

  washer_controller_1769_001-2_c03a.c (Released 2015-01)

  ---------------------------------------------------------------------

  Main (system state) task.

  Part of simple initial case study (washing machine controller)

  See "ERES (LPC1769)" book, Chapter 3.

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

// Project header
#include "../main/main.h"

// Task header
#include "washer_controller_1769_001-2_c03a.h"

// Support functions
#include "../task_support_fns/report_number_7seg_1769_001-2_c03a.h"

// ------ Public variable declarations -----------------------------

// Sensors
extern uint32_t Start_switch_pressed_G;
extern uint32_t Selector_dial_reading_G;
extern uint32_t Water_level_reading_G;
extern uint32_t Water_temperature_reading_G;

// Actuators
extern uint32_t Drum_motor_required_state_G;
extern uint32_t Drum_motor_required_speed_G;
extern uint32_t Door_lock_required_state_G;
extern uint32_t Detergent_hatch_required_state_G;
extern uint32_t Water_heater_required_state_G;
extern uint32_t Water_pump_required_state_G;
extern uint32_t Water_valve_required_state_G;

// ------ Private data type declarations ---------------------------

// Possible system states
typedef enum {INIT, START, FILL_DRUM, HEAT_WATER, PRE_WASH,
              MAIN_WASH, DRAIN_DRUM, SPIN, FINISHED, FAULT}
              eSystem_state;
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 1 of 8]

```
// ------ Private constants ---------------------------------------

// All durations are in seconds (short times here for demo)
#define MAX_FILL_DURATION (100)
#define MAX_WATER_HEAT_DURATION (100)
#define PRE_WASH_DURATION (20)
#define MAIN_WASH_DURATION (20)
#define DRAIN_DRUM_DURATION (10)
#define SPIN_DURATION (10)

// ------ Private variables ---------------------------------------

static eSystem_state System_state_G;

static uint32_t Time_in_state_G;

static uint32_t Program_G;

// Ten different programs are supported
// Each one may or may not use detergent
static uint32_t Detergent_G[10] = {1,1,1,0,0,1,0,1,1,0};

// Each one may or may not use hot water
static uint32_t Hot_Water_G[10] = {1,1,1,0,0,1,0,1,1,0};

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

  WASHER_CONTROLLER_Init()

  Prepare for WASHER_CONTROLLER_Update() task - see below.

-*------------------------------------------------------------------*/
void WASHER_CONTROLLER_Init(void)
   {
   System_state_G = INIT;

   // Used to report current system state in this demo
   REPORT_NUMBER_7SEG_Init();
   }
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 2 of 8]

```
/*-----------------------------------------------------------------*-

  WASHER_CONTROLLER_Update().

  Main (system state) task for the washing-machine controller.

-*-----------------------------------------------------------------*/
void WASHER_CONTROLLER_Update(void)
   {
   // Call once per second
   switch (System_state_G)
      {
      case INIT: // 0
         {
         // For demo purposes only
         REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

         // Set up initial state
         // Motor is off
         Drum_motor_required_state_G = STOPPED;
         Drum_motor_required_speed_G = 0;

         // Pump is off
         Water_pump_required_state_G = OFF;

         // Heater is off
         Water_heater_required_state_G = OFF;

         // Valve is closed
         Water_valve_required_state_G = CLOSED;

         // Detergent hatch is closed
         Detergent_hatch_required_state_G = CLOSED;

         // Unlock the door
         Door_lock_required_state_G = UNLOCKED;

         // Wait in this state until START switch is pressed
         if (Start_switch_pressed_G == 1)
             {
             // START *has* been pressed.

             // Read the selector dial
             Program_G = Selector_dial_reading_G;

             // Change state
             System_state_G = START;
             }

         break;
         }
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 3 of 8]

```
case START: // 1
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

   // Lock the door
   Door_lock_required_state_G = LOCKED;

   // Start filling the drum
   Water_valve_required_state_G = OPEN;

   // Release the detergent (if any)
   if (Detergent_G[Program_G] == 1)
      {
      Detergent_hatch_required_state_G = OPEN;
      }

   // Ready to go to next state
   System_state_G = FILL_DRUM;
   Time_in_state_G = 0;

   break;
   }

case FILL_DRUM: // 2
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

   // Remain in this state until drum is full
   if (++Time_in_state_G >= MAX_FILL_DURATION)
      {
      // Should have filled the drum by now...
      System_state_G = FAULT;
      }

   // Check the water level
   if (Water_level_reading_G == 1)
      {
      // Drum is full

      // Stop filling the drum
      Water_valve_required_state_G = CLOSED;

      // Does the program require hot water?
      if (Hot_Water_G[Program_G] == 1)
         {
         Water_heater_required_state_G = ON;

         // Ready to go to next state
         System_state_G = HEAT_WATER;
         Time_in_state_G = 0;
         }
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 4 of 8]

```
      else
         {
         // Using cold water only
         // Ready to go to next state
         System_state_G = PRE_WASH;
         Time_in_state_G = 0;
         }
      }
   break;
   }

case HEAT_WATER: // 3
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

   // Remain in this state until water is hot
   // NOTE: Timeout facility included here
   if (++Time_in_state_G >= MAX_WATER_HEAT_DURATION)
      {
      // Should have warmed the water by now...
      System_state_G = FAULT;
      }

   // Check the water temperature
   if (Water_temperature_reading_G == 1)
      {
      // Water is at required temperature
      // Ready to go to next state
      System_state_G = PRE_WASH;
      Time_in_state_G = 0;
      }

   break;
   }

case PRE_WASH: // 4
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

   // All wash program involve WASH_01
   // Drum is slowly rotated at medium speed
   Drum_motor_required_state_G = RUNNING;
   Drum_motor_required_speed_G = 50;

   if (++Time_in_state_G >= PRE_WASH_DURATION)
      {
      System_state_G = MAIN_WASH;
      Time_in_state_G = 0;
      }

   break;
   }
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 5 of 8]

```
case MAIN_WASH: // 5
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

   // Drum is rotated at higher speed
   Drum_motor_required_state_G = RUNNING;
   Drum_motor_required_speed_G = 70;

   if (++Time_in_state_G >= MAIN_WASH_DURATION)
      {
      System_state_G = DRAIN_DRUM;
      Time_in_state_G = 0;
      }

   break;
   }

case DRAIN_DRUM: // 6
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

   // Pump is activated to drain drum
   Water_pump_required_state_G = ON;

   // Drum is rotated at low speed
   Drum_motor_required_state_G = RUNNING;
   Drum_motor_required_speed_G = 10;

   // Check the water level
   if (Water_level_reading_G == 0)
      {
      // Drum is empty
      // Move to "Spin" state
      System_state_G = SPIN;
      Time_in_state_G = 0;
      }

   if (++Time_in_state_G >= DRAIN_DRUM_DURATION)
      {
      // Drum should have drained by now - fault
      System_state_G = FAULT;
      Time_in_state_G = 0;
      }

   break;
   }
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 6 of 8]

```
case SPIN: // 7
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) System_state_G);

   // Drum is rotated at high speed
   Drum_motor_required_state_G = RUNNING;
   Drum_motor_required_speed_G = 100;

   if (++Time_in_state_G >= SPIN_DURATION)
      {
      System_state_G = FINISHED;
      Time_in_state_G = 0;
      }

   break;
   }

case FINISHED: // -
   {
   // For demo purposes only
   REPORT_NUMBER_7SEG_Update((uint32_t) 10);

   // Set up safe state
   Drum_motor_required_state_G = STOPPED;
   Drum_motor_required_speed_G = 0;
   Water_pump_required_state_G = OFF;
   Water_heater_required_state_G = OFF;
   Water_valve_required_state_G = CLOSED;
   Detergent_hatch_required_state_G = CLOSED;

   // Wait 10 seconds before unlocking the door
   // (to ensure drum stopped, etc)
   if (++Time_in_state_G >= 10)
      {
      // Unlock the door
      Door_lock_required_state_G = UNLOCKED;

      // Now return to Init state (ready for next wash)
      System_state_G = INIT;
      Time_in_state_G = 0;
      }

   break;
   }
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 7 of 8]

```
      default:    // Default state
      case FAULT: // E
         {
         // For demo purposes only
         REPORT_NUMBER_7SEG_Update((uint32_t) 11);

         // Set up safe state
         Drum_motor_required_state_G = STOPPED;
         Water_pump_required_state_G = OFF;
         Water_heater_required_state_G = OFF;
         Water_valve_required_state_G = CLOSED;
         Detergent_hatch_required_state_G = CLOSED;

         // Wait 10 seconds before unlocking the door
         // (Ensure drum stopped.)
         if (++Time_in_state_G >= 10)
            {
            // Unlock the door
            Door_lock_required_state_G = UNLOCKED;
            }

         break;
         }
      }
   }


/*-----------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-----------------------------------------------------------------*/
```

Listing 15: TTRD03a (washer_controller_1769_001-2_c03a.c) [Part 8 of 8]

```
/*------------------------------------------------------------------*-

   start_switch_1769_001-2_c03a.c (Released 2015-01)

  ------------------------------------------------------------------

   Simple switch interface code, with software debounce.

   [Reads SW3 on LPCXpresso baseboard.]

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

// Project header
#include "../main/main.h"

// Task header
#include "start_switch_1769_001-2_c03a.h"

// ------ Public variable definitions ------------------------------

uint32_t Start_switch_pressed_G = 0;

// ------ Private constants ----------------------------------------

// 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)

// ------ Private variable definitions -----------------------------

static uint8_t sw3_input = 0;

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

  SWITCH_SW3_Init()

  Initialisation function for the switch library.

-*------------------------------------------------------------------*/
void SWITCH_SW3_Init(void)
   {
   // Set up "SW3" as an input pin
   // Params: Port : Pin : 1 for o/p, 0 for i/p
   GPIO_SetDir(START_SWITCH_PORT, START_SWITCH_PIN, 0);
   }
```

Listing 16: TTRD03a (start_switch_1769_001-2_c03a.c) [Part 1 of 2]

```
/*------------------------------------------------------------------*-

  SWITCH_SW3_Update()

  This is the main switch function.
  It should usually be scheduled approx. every 10 ms.

-*------------------------------------------------------------------*/
void SWITCH_SW3_Update(void)
   {
   // Duration of switch press
   static uint32_t Duration = 0;

   // Read SW3
   sw3_input = (GPIO_ReadValue(START_SWITCH_PORT) & START_SWITCH_PIN);

   if (sw3_input == SW_PRESSED)
      {
      Duration += 1;

      if (Duration > SW_THRES)
         {
         Duration = SW_THRES;

         Start_switch_pressed_G = 1;
         }
      else
         {
         // Switch pressed, but not yet for long enough
         Start_switch_pressed_G = 0;
         }
      }
   else
      {
      // Switch not pressed - reset the count
      Duration = 0;

      // Update status
      Start_switch_pressed_G = 0;
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 16: TTRD03a (start_switch_1769_001-2_c03a.c) [Part 2 of 2]

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

```
/*-----------------------------------------------------------------------*-

  water_level_1769_001-2_c03a.c (Released 2015-01)

  ----------------------------------------------------------------------

  Placeholder module.

  Part of simple initial case study (washing machine controller)

  See "ERES (LPC1769)" book, Chapter 3.

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

// Project header
#include "../main/main.h"

// Task header
#include "water_level_1769_001-2_c03a.h"

// Fault injection options
#include "../fault_injection/fault_injection_1769_001-2_c03a.h"

// ------ Public variable declarations ------------------------------

extern uint32_t Water_pump_required_state_G;
extern uint32_t Water_valve_required_state_G;

// ------ Public variable definitions -------------------------------

uint32_t Water_level_reading_G;

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

  WATER_LEVEL_Init()

  Prepare for WATER_LEVEL_Update() task - see below.

-*-----------------------------------------------------------------------*/
void WATER_LEVEL_Init(void)
   {
   // Placeholder task
   Water_level_reading_G = 0;
   }
```

Listing 17: TTRD03a (water_level_1769_001-2_c03a.c) [Part 1 of 2]

```
/*------------------------------------------------------------------*-

  WATER_LEVEL_Update().

  Placeholder task.

  In finished system, this task will read the water level in the
  washing-machine drum.

  Here we run a simple model.

-*------------------------------------------------------------------*/
void WATER_LEVEL_Update(void)
   {
   // Placeholder code
   static uint32_t Water_level = 0;

   // Is the drum filling?
   if ((Water_valve_required_state_G == OPEN) && !WATER_VALVE_FAULT)
      {
      // Drum is filling
      Water_level++;

      // Is the drum full of water (ready to wash)?
      if (Water_level >= 50)
         {
         Water_level_reading_G = 1;
         }
      }

   // Is the drum being drained?
   if ((Water_pump_required_state_G == ON) && !WATER_PUMP_FAULT)
      {
      // Drum is being drained
      if (Water_level > 0)
         {
         // Drum is draining
         Water_level--;
         }

      if (Water_level == 0)
         {
         // Drum is empty
         Water_level_reading_G = 0;
         }
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 17: TTRD03a (water_level_1769_001-2_c03a.c) [Part 2 of 2]

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

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

PART TWO: CREATING RELIABLE TTC DESIGNS

"How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

Alan Turing, 1949.

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

CHAPTER 4: Modelling system timing characteristics

As we noted in Chapter 1, the key feature of time-triggered embedded systems is that we can model the system timing behaviour very precisely, early in the development process. The key timing model is the "Tick List": we begun to explore the analysis of such lists in this chapter.

4.1. Introduction

The focus of this book is on real-time systems.

In real-time systems, the key requirement is for *deterministic* processing: to achieve this, we need to be able to model the system behaviour.

We explore the use of "Tick Lists" as a modelling tool in this chapter.

4.2. Basic Tick Lists

In this chapter, we will focus on models of "time-triggered co-operative" (TTC) systems.

As an example (first introduced in Chapter 1), Figure 26 shows a set of tasks (in this case Task A, Task B, Task C and Task D) that might be executed by a TTC scheduler. The arrows show the timer ticks (that is, timer interrupts).

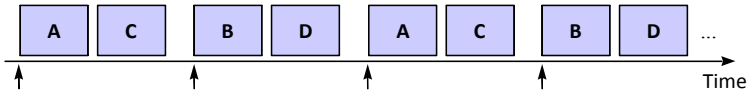


Figure 26: A set of tasks being released according to a pre-determined schedule.

In Figure 26, 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 many times when the system runs.

We can model the above system using a Tick List: such a list lays out which task will run in each system tick, and the order in which these tasks will run.

The Tick List corresponding to the task set shown in Figure 5 could be represented as shown in TL 1.

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

TL 1: The Tick List corresponding to Figure 5.

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

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

In most designs, TT systems will continue executing the same Tick List "for ever", unless: [i] the system changes mode, as a result of a user request; [ii] the system is halted, for example by removing power; or [iii] a fault occurs.

4.3. Determining the required tick interval

In Figure 26, the timer ticks are periodic: this is a common scenario and will be the case for all of the designs presented in this book.

In an aerospace application, the "tick interval" (that is, the time period 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.

If we are starting a new design, how can we determine the required tick interval?

The tick interval can be determined by calculating the Greatest Common Divisor (GCD) of the task periods.

Suppose, for example, that the task set is shown in Table 3.

Table 3: A set of periodic tasks.

Task ID	Period (ms)	WCET (µs)	Offset (ticks)
A	1	200	0
B	2	300	0
C	2	200	1
D	2	150	1
E	3	200	11
F	5	100	13
G	7	50	25

In this case, the Greatest Common Divisor (GCD) is 1ms.

Note that we may be able to employ a tick interval that is less than the GCD value in a TTC design, if we can meet the requirement for "Short Tasks".

4.4. Working with "Short Tasks"

As we will see in Chapter 6, a key to creating an effective TT task is to keep the task duration short.

Rather than providing anodyne advice about keeping task durations "as short as possible", it can be helpful to define a "Short Task" as one that has a worst-case execution time (WCET) that is: [i] less than the period of any task in the system; and [ii] less than the system tick interval.

If we can meet the Short Task requirement for all tasks in the system, then may be able to employ a TTC architecture: if we cannot meet this requirement, then we are likely to have to employ a TTH architecture or TTP architecture.[11]

Please consider again the task set shown in Table 3. We have determined that a tick interval of (at most) 1 ms would be suitable for this task set. As the longest WCET of any task in the system is 300 µs, then this tick interval would allow us to meet the Short Task requirements for this system.

4.5. The hyperperiod

In the examples considered so far in this chapter (and in almost all cases considered in this book), we will be exploring system designs based on sets of periodic tasks.

For any set of periodic tasks, the cycle of task executions will always repeat, after an interval known as the hyperperiod (or "major cycle"). The length of the hyperperiod can be determined by calculating the "lowest common multiple" (LCM) of the periods of all tasks in the set.

For example, suppose that we have a set of two tasks, with periods of 4 ms and 5 ms.

The multiples of 4 are: 4, 8, 12, 16, <u>20</u>, 24…

The multiples of 5 are: 5, 10, 15, <u>20</u>, 25 …

The lowest common multiple (and the length of the hyperperiod) is 20 ms.

As another example, consider again the task set shown in Table 3: here, the length of the hyperperiod is 210 ms (or 210 ticks).

4.6. Performing GCD and LCM calculations

Calculating the required tick interval and the length of the hyperperiod by hand is easy for small task sets, but becomes tedious for larger task sets. Fortunately, there are functions provided in popular spreadsheet packages (such as Microsoft Excel®) that can be used to calculate GCD and LCM values very easily.

4.7. Synchronous and asynchronous task sets

When we start to model systems based on sets of periodic tasks, we need to be clear about the difference between synchronous and asynchronous designs.

---

[11] In Part One of this book, our focus will be on TTC designs: we will consider TTH / TTP designs in Part Two.

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

Where a set of periodic tasks is synchronous, all tasks execute for the first time in the first tick. This arrangement is uncommon in practical systems, not least because the sum of the execution time of all tasks is very likely to exceed the tick interval.

A much more common situation is where a set of periodic tasks is asynchronous. In this case, there is a delay (or offset) – measured in ticks - before one or more tasks runs for the first time. This is the case – for example – in the task set shown in Figure 5. In this figure, Task A and Task B run with an offset of 0, but Task C and Task D both run with an offset of 1 tick. Note that all four tasks have the same period (2 ticks) in this example.

4.8. The importance of task offsets

In a TT system, we typically start with a synchronous set of period tasks. The system designers will then convert this into an asynchronous task set by using changes in the offsets to manipulate the order in which the tasks are released during the system operation. In this way, it is possible to control various important aspects of the system design (such as task precedence constraints, system response times and CPU loading).

Consider a simple example.

Suppose that we have three tasks (Task A, Task B, Task C) that are part of a larger task set. Each of these (three) tasks runs every 10 ms.

We will assume that the tick interval is 1 ms.

We will also assume that we want Task B to run first in the sequence, followed by Task C, then Task A (to meet particular system requirements).

We could add the tasks to the schedule as shown in Code Fragment 15.

```
// Add tasks to schedule.
// Parameters are:
// 1. Task name
// 2. Initial delay / offset (in ticks)
// 3. Task period (in ticks): Must be > 0
// 4. Task WCET (in microseconds)
// 5. Task BCET (in microseconds)

// Add Task A (offset = 2, runs third in sequence)
SCH_Add_Task(Task_A, 2, 10, 100, 50);

// Add Task B (offset = 0, runs first in sequence)
SCH_Add_Task(Task_B, 0, 10, 150, 120);

// Add Task C (offset = 1, runs second in sequence)
SCH_Add_Task(Task_C, 1, 10, 100, 50);
```

Code Fragment 15: Adding tasks to the schedule with different offsets in order to control the execution sequence. See text for details.

The Tick List will then start like this (ignoring any other tasks that may be present in the set):

```
[Tick 0]
Task_B

[Tick 1]
Task_C

[Tick 2]
Task_A
...
```

In this way, we can use the offset values to control the task order. We can then review the Tick List to confirm that the resulting task sequence meets all of our requirements, using the techniques explored during the course of this chapter.

We will say more about techniques for determining the required task offsets in Chapter 5 (Section 5.12).

4.9. The Task Sequence Initialisation Period (TSIP)

When a set of asynchronous periodic tasks runs, there may be an initial period during which the task sequence is not the same as the task sequence when the system is in a "steady state". We refer to this initial period as the Task Sequence Initialisation Period (TSIP).

We say more about the TSIP in Chapter 10 when we consider techniques for monitoring task execution sequences at run time.

4.10. Modelling CPU loading

In Chapter 1, we presented a simple example with three periodic tasks, with worst-case execution time (WCET) values as shown in Table 4.

Table 4: A set of 3 periodic tasks

Task	Period (ms)	WCET (ms)	CPU load (%)
A	5	2	40%
B	10	4	40%
C	20	3	15%
			95%

In this example, the total (average) CPU load is 95%: as this figure is less than 100%, we <u>may</u> be able to schedule the task set.

Such a basic analysis is useful, as an initial "sanity check" during the early phases of a system development, but it doesn't give us any detailed information about the system CPU load.

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

To provide detailed information, we consider the CPU load for every tick over the hyperperiod.

For each tick:
(CPU load) = (Sum of the WCETs for all executing tasks) / (tick interval)

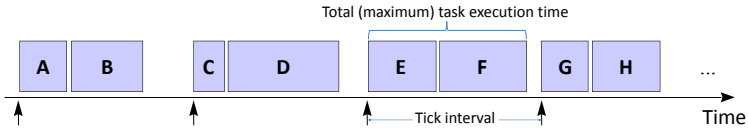


Figure 27: Determining the maximum CPU load for a system.

We need to repeat this calculation for all ticks in the hyperperiod in order to determine what the maximum CPU load will be during the program execution.

This process can be easily automated – once we have a record of the Tick List.

4.11. Worked Example A: Determining the maximum CPU load

Consider the following Tick List:

```
[Tick 0]   [Tick 7]    [Tick 12]
Task A     Task C      Task A
Task B     Task C      Task B
Task C     [Tick 8]    Task C
[Tick 1]   Task A      [Tick 13]
[Tick 2]   Task B      [Tick 14]
Task B     [Tick 9]    Task B
[Tick 3]   Task C      Task C
Task C     [Tick 10]   Task C
[Tick 4]   Task B      [Tick 15]
Task A     Task C      [Tick 16]
Task B     Task C      Task A
[Tick 5]   Task C      Task B
Task C     [Tick 11]   [Tick 17]
[Tick 6]               Task C
Task B                 [Tick 18]
                       Task B
                       [Tick 19]
```

Assume that the scheduler imposes a 2% CPU load.

Assume that the WCET figures are as follows:

Task A = 9 ms; Task B = 3 ms; Task C = 5 ms

Assume that the ticks are periodic, and that the tick interval is 20 ms.

What is the maximum CPU load? What is the average CPU load?

4.12. Worked Example A: Solution

The annotated Tick List below totals up the WCETs for the tasks in each tick.

```
[Tick 0]          [Tick 7]          [Tick 12]
Task A = 9        Task C = 5        Task A = 9
Task B = 3        Task C = 5 [10]   Task B = 3
Task C = 5 [17]   [Tick 8]          Task C = 5 [17]
[Tick 1]          Task A = 9        [Tick 13]
[Tick 2]          Task B = 3 [12]   [Tick 14]
Task B = 3        [Tick 9]          Task B = 3
[Tick 3]          Task C = 5        Task C = 5
Task C = 5        [Tick 10]         Task C = 5 [13]
[Tick 4]          Task B = 3        [Tick 15]
Task A = 9        Task C = 5        [Tick 16]
Task B = 3 [12]   Task C = 5        Task A = 9
[Tick 5]          Task C = 5 [18]   Task B = 3 [12]
Task C = 5        [Tick 11]         [Tick 17]
[Tick 6]                            Task C = 5
Task B = 3                          [Tick 18]
                                    Task B = 3
                                    [Tick 19]
```

Tick 10 has the highest loading: a total of 18 ms out of a tick interval of 20 ms.

This gives us a basic CPU load of 90% (= 18/20 x 100%). To this figure, we have to add the stated scheduler load figure of 2%, giving us a total (maximum) CPU load of approximately 92%.

To calculate the average CPU load, we note that:

- Task A runs every 80 ms, WCET 9ms. 11.25% average CPU load.
- Task B runs every 40 ms, WCET 3 ms. 7.5% average CPU load.
- Task C runs x13 in 400 ms, WCET 5 ms. 16.25% average CPU load.

The total (average) CPU load is therefore 35% (37% including the scheduler load). Note that here – as in many cases. – the average loading figure is <u>much lower</u> than the maximum load.

Note that we can calculate the average CPU load for all tasks in the same way that we used for Task C here: that is, by counting up the number of executions in the hyperperiod: please see Table 5.

Table 5: An alternative method for calculating the average CPU load.

Task	Executions	WCET (ms)	Total exec. time (ms)	Hyperperiod (ms)	Ave CPU loading
A	5	9	45	400	11.25%
B	10	3	30	400	7.50%
C	13	5	65	400	16.25%
					35.00%

4.13. Modelling task jitter

Suppose we have a task that is due to execute every 10 ms. Ideally, it might be released at the following times:

{5, 15, 25, 35, 45, 55, 65, 75, …}

… where these times are measured from the time at which the scheduler began executing.

In reality, we may see some variation in the ideal behaviour presented above. For example, the task might be released at the following times (in milliseconds):

{5, 16, 24, 36, 47, 54, 66, 75, …}.

This departure from the ideal behaviour is known as "task release jitter" (or simply "jitter").

There are different ways in which this task release jitter can be recorded: in this book we will consider the release jitter in a given set of task-release data to be represented as follows:

Task release jitter =
(Maximum interval between task releases in the data set)
- (Minimum interval between task releases in the data set)

Returning to our example above, we have the recorded task release times as follows:

{5, 16, 24, 36, 47, 54, 66, 75, …}

The intervals are therefore as follows:

{11, 8, 12, 11, 7, 12, 9, …}

The minimum interval is 7 ms and the maximum interval is 12 ms: we therefore represent the jitter as 5 ms.

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

The impact of jitter can be very significant in any system involving data sampling, data playback or control. For example, as laid out in Box 4, jitter levels of around 1 µs are enough to result in a loss of data in a system that is sampling using at a low frequency using an 8-bit ADC (and the effects are much more severe at higher sample rates / higher bit rates).

A useful "rule of thumb" is that jitter rates of 10% (measured as a percentage of the required period) are enough to render <u>any</u> system involving sampled data useless.

Given the impact of jitter on system performance (and sometimes on system safety) it is clearly important that we can model the jitter behaviour of the tasks in our system. Fortunately, this is (also) easy to do from the Tick List.

Figure 28 illustrates an example of jitter modelling using a Tick List.

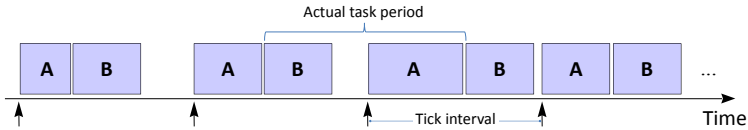


Figure 28: Modelling jitter using a Tick List

In Figure 28, the jitter level of Task B will be determined by three things:

- the jitter in the system ticks;
- the jitter in the scheduler; and,
- the variation in the execution time of Task A.

The jitter in the system ticks will depend on the choice of hardware, and aspects of the scheduler configuration: we explore techniques for measuring this jitter in Chapter 5.[12]

The level of jitter introduced by the scheduler itself is another topic that we will consider in Chapter 5.

The variation in the execution time of Task A is – clearly – also a concern here: in designs with multiple task executions in each tick, we generally aim to release jitter-sensitive tasks at the start of the tick interval. We can also help to reduce levels of jitter in the task set by "balancing" the code in Task A (so that it has approximately the same execution time on every release).

We will discuss techniques for execution-time balancing in Chapter 6.

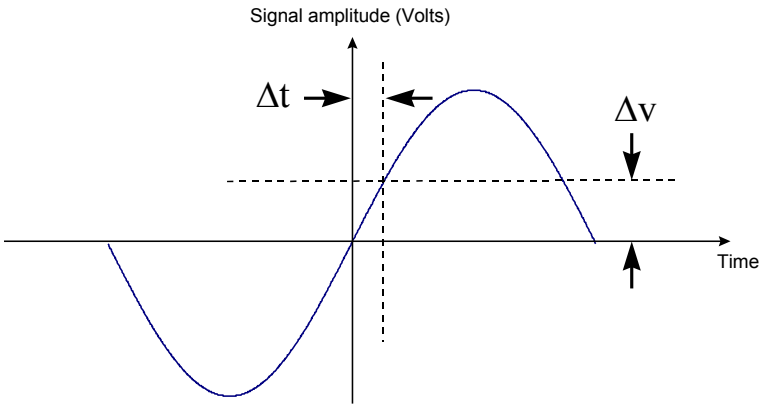
[12] Note that the data from such measurements are needed in order to complete the system model.

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

Understanding the impact of jitter (Part 1 of 2)

Assume that we are trying to sample a signal, represented by a sine wave.

We want to understand the consequences of jitter in the sample time.



We know:

$$v(t) = A\sin(2\pi ft)$$

$$\frac{\Delta V}{\Delta t} \approx \frac{dv}{dt}$$

Also:

$$\frac{dv}{dt} = A2\pi f\cos(2\pi\ ft)$$

At t = 0:

$$\frac{dv}{dt} = A2\pi f$$

Box 4a

Understanding the impact of jitter (Part 2 of 2)

Assume we are sampling a signal at the maximum amplitude (A_{FS}), using an N-bit ADC:

$$A_{FS} = 2^N z$$

…where z is the ADC "step size" (that is, the voltage represented by the least significant bit).

Then:

$$\frac{dv}{dt} = A_{FS} \pi f$$

Thus:

$$\Delta V = A_{FS} \pi f \ \Delta t$$

If the jitter is to have no measurable impact, we need to ensure that it has an amplitude < z / 2.

This implies that Δt must satisfy this condition:

$$\Delta t < \frac{1}{2^N} \cdot \frac{1}{2\pi f}$$

For example, suppose we wish to sample a 500 Hz tone, using an 8-bit ADC (a very modest requirement).

The maximum allowable jitter will be:

$$\frac{1}{2^N} \cdot \frac{1}{2\pi f} \text{ seconds.}$$

That is:

$$\frac{1}{2^8} \cdot \frac{1}{1000\pi} = 1.24 \ \mu\text{s}$$

Box 4b

4.14. Worked Example B: Modelling task release jitter

Consider again the Tick List shown in Example 1.

Why are the jitter levels for Task B are likely to be higher than the jitter levels of Task A?

Will Task A have 0 jitter? If not, why not?

Assuming that Task A has the following characteristics:

Task A WCET = 9 ms
Task A BCET = 7 ms

What is the maximum jitter for Task B?

4.15. Worked Example B: Solution

As Task A always occurs at the start of a tick, the release jitter will depend on the scheduler jitter (only). On the other hand, the release jitter for Task B depends on the scheduler jitter and – particularly – on any variations in the execution time of Task A.

Overall, it's difficult to imagine a practical scenario in which the release jitter for Task B will be lower than the jitter level for Task A.

Jitter for Task A itself is likely to be very low, if the scheduler has been implemented appropriately: this will, of course, depend on the scheduler implementation.

For example, levels of release jitter can sometimes be improved by "balancing" the code in the scheduler: we will say more about this in Chapter 5.

We will now consider the jitter levels for Task B in more detail.

The annotated Tick List is shown on Page 105.

This list shows the scenario in which the jitter for Task B will be at a maximum: this is when Task A alternates between its maximum and minimum execution times on alternate releases. The annotated list shows the WCET for Task A [W = …] and the interval between the releases for Task B [I = …]. Where […] is shown, the data simply repeats maximum or minimum values shown earlier.

The end result is that jitter for Task B is 49 ms – 31 ms = 18 ms.

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

```
[Tick 0]
Task A [W = 07ms]
Task B [I = 47ms]
Task C
[Tick 1]
[Tick 2]
Task B [I = 33ms]
[Tick 3]
Task C
[Tick 4]
Task A [W = 09ms]
Task B [I = 49ms]
[Tick 5]
Task C
[Tick 6]
Task B [I = 31ms]
[Tick 7]
Task C
Task C
[Tick 8]
Task A [W = 07ms]
Task B [I = 47ms]
[Tick 9]
Task C
[Tick 10]
Task B [I = 33ms]
Task C
Task C
Task C
[Tick 11]
[Tick 12]
Task A [W = 09ms]
Task B […]
Task C
[Tick 13]
[Tick 14]
Task B […]
Task C
Task C
[Tick 15]
[Tick 16]
Task A [W = 07ms]
Task B […]
[Tick 17]
Task C
[Tick 18]
Task B […]
[Tick 19]
```

4.16. Modelling response times

All real-time systems (regardless of the internal system architecture) need to respond to events. A key consideration is being able to specify how long a system will take – in the worst case – to respond to a given event.

Here, we will first consider a braking system in a passenger car (Figure 29).

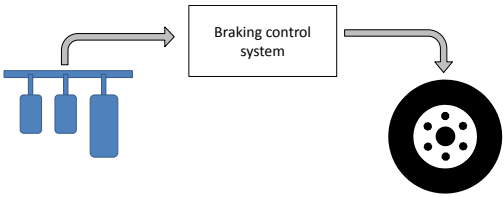


Figure 29: A schematic representation of a braking circuit in a passenger car.

In this example, we will assume that – as part of a larger system – we have two tasks (Task A and Task D) that provide our braking capability (see Figure 30). More specifically, we will assume that Task A is used to monitor the (brake) pedal input and Task D is used to apply the brakes.

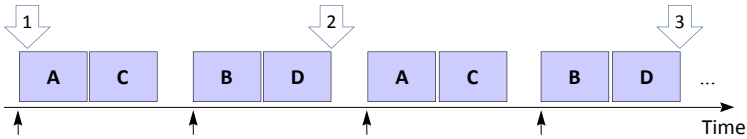


Figure 30: A Tick List corresponding to the simplified braking system shown in Figure 29.

We might start by assuming that if the brake press is detected by Task A at Time 1 (see Figure 30), the brakes will be applied by Task D at Time 2.

However, we cannot generally be sure when in Task A the brake press will be detected, and we must generally assume a "worst case" scenario in which only events that occur before the start of Task A (for example) will be detected by this task. This means that – in Figure 30 – a brake press at Time 1 is likely to result in a brake activation at Time 3.

More generally, the response time can be represented as follows:

Response time
= (Maximum time taken to detect the event)
+ (Maximum time to generate response following detection)

For strictly periodic tasks, the maximum time taken to detect the event can be determined from the task period and the task (worst-case) execution time.

That is, the detection time for Task (A) will be Period (A) + WCET (A).

For a simple activation task (that responds immediately – the next time it runs – after the event has been detected), we need to know the maximum interval (in the Tick List) between the completion of the detection task and the completion of the activation task.

Let us suppose (for example) that Figure 30 represents our entire Tick List.

In this case, the maximum response time is given by:

(One tick interval) – BCET (A) + WCET (B) + WCET (D)

In this case, our total response time will be given by:

Period(A) + WCET (A) + (One tick interval)
– BCET (A) + WCET(B) + WCET (D)

Suppose that our requirement is for a response within 50 ms.

Further suppose that:

The tick interval is 10 ms;
BCET (A) is 2 ms; WCET (A) is 4 ms;
WCET (B) is 4 ms;
WCET (D) is 5 ms.

Our maximum response time is then given by:

Period(A) + WCET (A) + (One tick interval)
– BCET (A) + WCET(B) + WCET (D)

= 20 ms + 4 ms + 10 ms – 2 ms + 4 ms + 5 ms
= 39 ms (which is within the specification).

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

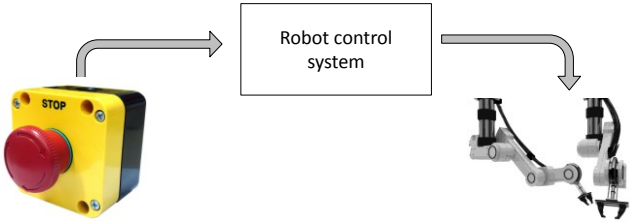


Figure 31: An “emergency stop” facility implemented as part of a control system for an industrial robot.

4.17. Worked Example C: An “emergency stop” interface

In practice, accurate modelling of task and system response times usually needs to be a “white box” activity: that is, we need to understand the internal operation of the tasks, if we are to provide an accurate (and useful) system model.

We will illustrate this issue with another (superficially very simple) example.

Suppose that we have an industrial robot, with an emergency stop switch (Figure 31).

We will assume that the associated Tick List is as shown in TL 2.

```
[Tick 0]                [Tick 3]
SWITCH_Update           SWITCH_Update
EMERGENCY_STOP_Update   EMERGENCY_STOP_Update

[Tick 1]                [Tick 4]
SWITCH_Update           SWITCH_Update
EMERGENCY_STOP_Update   EMERGENCY_STOP_Update
                        TASK_X_Update
```

TL 2: The Tick List associated with the “Emergency Stop” system (Figure 31).

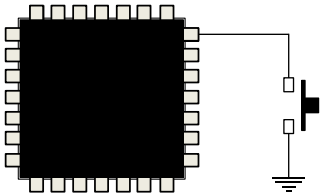


Figure 32: A typical switch connection to a microcontroller. Note that – in practice – the input is likely to be “opto isolated” (or have some equivalent protection), but such an interface would not have an impact on the software architecture that is discussed here.

Following the approach discussed in Section 4.16, we could start to calculate the response time as follows:

Period(SWITCH_Update)
+ WCET (SWITCH_Update)
+ WCET(EMERGENCY_STOP_Update)

In many practical systems, this model would fail to take into account (at least) the behaviour of the switch interface.

For example, let's assume that the switch is connected to a microcontroller as illustrated in Figure 32.

In an ideal world the switch connections shown in Figure 32 would give rise to a waveform (at the port pin) which looks something like that illustrated in Figure 33 (top). In practice, all mechanical switch contacts *bounce* (that is, turn on and off, repeatedly, for a short period of time) after the switch is closed or opened. As a result, the actual input waveform looks more like that shown in Figure 33 (bottom). Usually, switches bounce for less than 20 ms: however large mechanical switches exhibit bounce behaviour for 50 ms or more.

Code Fragment 16 involves checking the switch periodically and reporting a switch press only after a "stable" reading has been obtained over an interval that is at least equal to the bounce period.[13]

We can now return to the question that was raised at the start of this section.

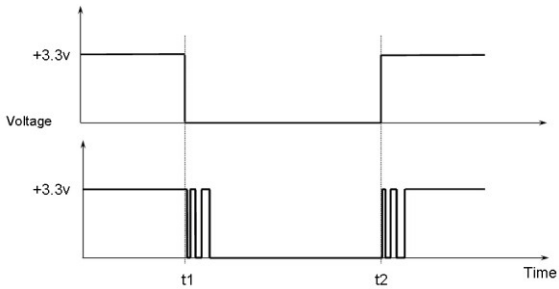


Figure 33: The voltage signal resulting from a mechanical switch. [Top] Idealised waveform resulting from a switch depressed at time t1 and released at time t2 [Bottom] Actual waveform showing leading edge bounce following switch depression and trailing edge bounce following switch release.

[13] A more complete switch-interface task is given in Listing 16, on p.86: this code was used as the interface to the Start switch in the first washing-machine case study. In the present discussion, we'll work with Code Fragment 16, primarily for ease of reference.

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

```
#define SW_THRES (10)

…

void SWITCH_Update(void)
   {
   static uint32_t Duration = 0;

   if (Sw_pin == SW_PRESSED)
      {
      Duration += 1;
      if (Duration > SW_THRES)
         {
         Duration = SW_THRES;
         Sw_pressed_G = 1;  // Switch is pressed...
         return;
         }
      // Switch pressed, but not yet for long enough
      Sw_pressed_G = 0;
      return;
      }

   // Switch not pressed – reset the count
   Duration = 0;
   Sw_pressed_G = 0;  // Switch not pressed...
   }
```

Code Fragment 16: A polled switch interface. Code adapted from "PTTES" (Chapter 19).

If we assume that the code shown in Code Fragment 16 is used to read the emergency stop switch (as shown in Figure 31, assuming the hardware interface shown in Figure 32), and that the code shown in Code Fragment 17 is used to shut down the robot, then how long does it take to begin shutting down the robot?

We will further assume that the WCET figures are as follows:

WCET (SWITCH_Update) = 100 µs.
WCET (EMERGENCY_STOP_Update) = 100 µs.

The tick interval is assumed to be 1 ms.

The switch bounce period is assumed to be 10 ms.

```
void EMERGENCY_STOP_Update(void)
   {

   Robot_shutdown_pin = 1;

   if (Sw_pressed_G == 1)
      {
      Robot_shutdown_pin = 0;
      }
   }
```

Code Fragment 17: Code (simplified) used to shut down the robot shown in Figure 31.

4.18. Worked Example C: Solution

With the code presented, the detection of the switch press may take 21 ms, for the reasons illustrated in Figure 34.

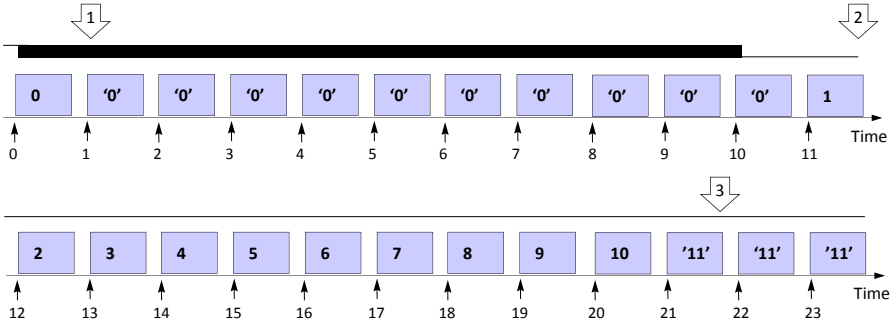


Figure 34: Changes to the internal state of the task SWITCH_Update as a switch is pressed.

Figure 34 shows only the task for reading the switch (Code Fragment 16).

We are interested (of course) in worst-case behaviour. As shown in Figure 34, we assume that the switch may be pressed immediately after Tick 0, and that we may not detect this switch press until the task next runs (Tick 1, indicated by an arrow in the figure). As the switch may then bounce for 10 ms, we assume that the counter variable in the switch-reading task – Duration – may remain at a value of 0 (or be frequently reset back to 0) until the switch has stopped bouncing.

Only after the switch has stopped bouncing will the Duration variable be incremented (once per millisecond in this case), until it exceeds the threshold value (10 in this case).

The end result is that it takes a total time of 21 ms, plus WCET (SWITCH_Update) to confirm that the emergency stop switch has been pressed.

More generally, we may use similar code in the following situation:

- The SWITCH_Update task is called with period SWITCH_UPDATE_PERIOD
- The switch debounce time is SWITCH_DEBOUNCE_TIME: this must be "rounded up" to whole number of SWITCH_UPDATE_PERIODs.
- The threshold is SWITCH_COUNT_THRESHOLD (again expressed in SWITCH_UPDATE_PERIODs)

Of course, this condition must be met:

SWITCH_COUNT_THRESHOLD >=
SWITCH_DEBOUNCE_TIME

The worst-case detection time is then given by:

SWITCH_UPDATE_PERIOD
+ SWITCH_DEBOUNCE_TIME
+ SWITCH_COUNT_THRESHOLD

For the above example, this amount to:

1 ms + 10 ms + 10 ms = 21 ms (as previously modelled).

If we called the SWITCH_Update task every 5 ms, and set the threshold to 2 (periods), the detection time would then become:

5 ms + 10 ms + 10 ms.

We now return to our original example.

As the EMERGENCY_STOP_Update task (Code Fragment 17) runs immediately after SWITCH_Update (in every tick), the total response time will be:

21 ms
+ WCET (SWITCH_Update)
+ WCET(EMERGENCY_STOP_Update)

4.19. Generating Tick Lists

In this chapter, we have considered some techniques for analysing Tick Lists. We will explore techniques for creating such lists in Chapter 5 (Section 5.11).

4.20. Conclusions

As we noted in Chapter 1, the key feature of time-triggered embedded systems is that we can model the system timing behaviour very precisely, early in the development process. The key timing model is the Tick List: we began to explore the development and use of such lists in this chapter. We will consider various other Tick-List models in the remainder of this book.

In Chapter 5, we turn our attention to techniques that can be used to obtain timing data for use in system models.

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

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

CHAPTER 5: Obtaining data for system models

In this chapter, we explore techniques for measuring the execution time of your tasks, and for measuring jitter (both for your tasks and for the scheduler itself). We also consider ways in which you can generate a Tick List.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD05a: Instrumented TTC scheduler (BCET and WCET)
- TTRD05b: Instrumented TTC scheduler with injected task overrun
- TTRD05c: Instrumented TTC scheduler (tick jitter)
- TTRD05d: Instrumented TTC scheduler (task jitter)
- TTRD05e: Creating Tick Lists (TTC scheduler)

5.1. Introduction

The focus of this book is on real-time systems. As we discussed in Chapter 1, the key characteristic for such systems is *deterministic* processing.

Throughout this book, we argue that the key to deterministic processing is the development of a system model: we began to describe how we can analyse a Tick List representation of TT systems in Chapter 4: in the present chapter, we discuss how we can obtain the information that is required to create such a model.

5.2. The importance of WCET / BCET information

In Chapter 4, we required both "worst-case execution time" (WCET) and "best-case execution time" (BCET) for all tasks in order to create the system model: without these data, we cannot model the system behaviour effectively (see Figure 35).

In addition to employing timing data in system models, we often use differences between the WCET and BCET values to identify tasks that may benefit from a design review. More specifically, we typically record the ratio between WCET and BCET for each task and – for tasks with a high ratio (typically around 3 or greater) – we may try to "balance" the task by trying to bring the BCET and WCET figures more in line. We say more about balancing techniques in Chapter 6

At run time, we again require WCET information if we are to be able to detect task overruns (a common requirement in many real-time systems).

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

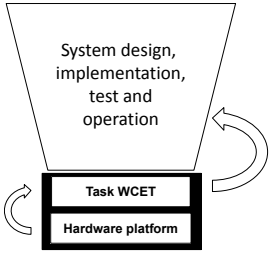


Figure 35: Knowledge of the "worst case execution time" (WCET) is a key requirement during the development, test & verification, and operation of real-time embedded systems.

It may also be useful to know the expected BCET of tasks at run time. This is because – in situations where a task completes more quickly than its predicted BCET (a situation sometimes referred to as a task underrun) – this may be symptomatic of problems with the system. For example, if a task suddenly develops a BCET value of 0, something may be seriously wrong.

Advantages of TTC designs

Information about the upper bound of task execution time is a key factor when designing real-time embedded systems. This "worst case execution time" (WCET) is defined as the longest time taken by the processor to execute a task in the absence of pre-emption.

One advantage of TTC designs is that tasks can be tested in isolation, and the timing measured will generally be the same as that measured when the same task is executed with 100 others in the final system. In any system that involves task pre-emption (even TT designs) timing measurements are more challenging.

Box 5

5.3. Challenges with WCET / BCET measurements

In this chapter, we focus on obtaining timing data using measurements, and it should be stated from the outset that this is not an ideal solution.

To explain this, consider a very simple example (Code Fragment 18).

```
void Flashing_LED_Toggle(void)
   {
   static int Count;

   if (++Count == 100)
      {
      Count = 0;
      led_state = !led_state;
      }
   GPIO_Write(LED_Pin, led_state);
   }
```

Code Fragment 18: A simple code example with a conditional statement.

In Code Fragment 18, there are two main paths through the code: [i] a "best case" path, when the Count value does not reach 100; and, [ii] a "worst-case" path, when the count reaches 100.

Even in this (almost trivial) example, we could easily miss the worst-case situation if we make less than 100 measurements from the code. In any code with multiple or more complicated conditional statements, possibly combined with various different system inputs, making sure that you have actually measured the WCET (and BCET) correctly is a very challenging task (Figure 36).

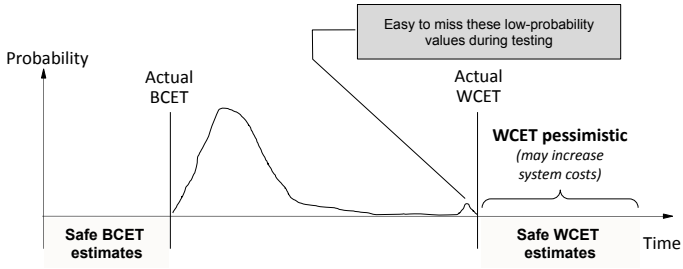


Figure 36: Some of the challenges involved in measurement-based WCET / BCET determination.

Instead – where possible – we would wish to use some form of static timing analysis (STA): this involves automated assessment of the executable code in order to determine the longest and shortest paths through the code (Figure 37), a process that typically starts with the generation of a Control Flow Graph (Figure 38).

Unfortunately, STA requires very detailed hardware models: these are not generally available for COTS processor hardware (on grounds of commercial confidentiality). In addition, some features of some processors (e.g. "superscalar" architectures, complicated cache arrangements, memory "burst" modes) make STA very challenging. Within the present edition of this book, we will therefore restrict our discussions to measurement-based techniques.

```
void Flashing_LED_Toggle(void)
   {
   static int Count;

   if (++Count == 100)
      {
      Count = 0;
      led_state = !led_state;
      }

   GPIO_Write(LED_Pin, led_state);
   }
```

BCET = 1.2 µs
WCET = 2.7 µs

Figure 37: A schematic representation of the process of static timing analysis. In practice, this process usually involves an analysis of the executable code (rather than the source code).

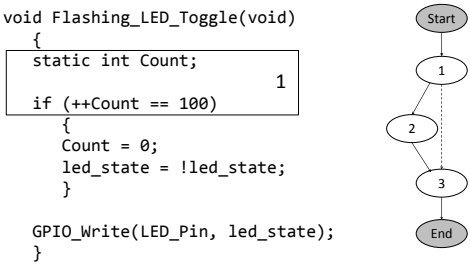


Figure 38: The process of static timing analysis usually involves creation of some form of "control flow graph", that shows the basic program blocks and the flow of execution between them.

5.4. Instrumenting a TTC scheduler: WCET-BCET (TTRD05a)

In the discussions earlier in this chapter, we have focused on detecting task overruns (and related system overloads) at run time.

In order to be able to perform such run-time monitoring (and to support the development of the system in the first place), we need to have accurate information about the WCET (and BCET) of each system task.

An effective way of obtaining detailed information about the execution times of individual tasks is to instrument the scheduler. What this means in practice is that – every time we release a task – we start a timer running: once the task completes, we stop the timer and record the elapsed time (see Code Fragment 19).

```
// Check if there is a task at this location
if (SCH_tasks_G[Index].pTask)
   {
   if (--SCH_tasks_G[Index].Delay == 0)
      {
      // The task is due to run

      SCH_MEASURE_ET_Start_Timer0();

      (*SCH_tasks_G[Index].pTask)();  // Run the task

      SCH_MEASURE_ET_Stop_Timer0();
      SCH_MEASURE_ET_Update_Task_Timing(Index,
                                        SCH_MEASURE_ET_Read_Timer0());
```

Code Fragment 19: Part of the Dispatcher function from a TTC scheduler that has been instrumented in order to record task execution times. From TTRD05a

We then keep track of the maximum and minimum execution time for the task, and provide a means of reporting these values.

TTRD05a provides full support for this process, including a menu-driven interface that uses a USB link to transfer data to a laptop PC (or similar device).

We illustrate the use of this reference design in Appendix 1.

5.5. Example: An injected task overrun (TTRD05b)

TTRD05b again includes an injected (transitory) task overrun and demonstrates that we can measure the WCET of the offending task.

5.6. Obtaining jitter measurements: Tick jitter (TTRD05c)

In addition to knowledge about the execution time of the tasks in the system, we often need to understand how much jitter tasks will experience in their release times.

We explored the implications of jitter in Chapter 4.

In this chapter, we are concerned with making measurements of both jitter in the system tick, and in the task release times.

TTRD05c illustrates how we can obtain (tick) jitter measurements.

Code Fragment 20 shows the instrumented scheduler ISR from this design.

```
void SysTick_Handler(void)
   {
   uint32_t Timer_reading;
   // Measure tick jitter
   SCH_MEASURE_JITTER_Stop_Timer0();
   Timer_reading = SCH_MEASURE_JITTER_Read_Timer0();
   SCH_MEASURE_JITTER_Start_Timer0();

   // Now process the data
   SCH_MEASURE_JITTER_Update_Jitter_Record(Timer_reading);

   // 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 20: The scheduler ISR from TTRD05c: this design has been instrumented to measure tick jitter.

5.7. Example: The impact of idle mode on a TTC scheduler

As we noted in Chapter 2, most TTC scheduler enter “idle” mode at the end of the Dispatcher function: this is typically achieved by means of the SCH_Go_To_Sleep() function (Code Fragment 21).

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

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

Code Fragment 21: 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, we also claimed in Chapter 2 that putting the processor to sleep can help us to control the level of tick jitter.

We can now test the validity of this claim, using TTRD05a. We can do this by running the system twice: once where the processor enters idle mode between ticks, and once when the __wfi() call is commented out.

Table 6 shows the results obtained on a test system.

Table 6: Measuring tick jitter in a scheduler with (and without) idle mode.

Scheduler	Tick jitter (µs)
TTC scheduler with idle mode	0
TTC scheduler without idle mode	0.18

As can be seen from these results, the use of idle mode has a significant impact on the tick jitter (bringing it down to a level that cannot be measured using this approach).

5.8. Obtaining jitter measurements: Task jitter (TTRD05d)

Tick jitter is a useful measure for the timing behaviour of a scheduler: in effect, it forms a "baseline" for the level of jitter that we can expect to see when tasks are released. Such information forms an important part of the modelling process that we described in Chapter 4.

Given knowledge of the task schedule, the levels of tick jitter generated by the scheduler itself, and knowledge of the BCET and WCET of each task, we can use Tick Lists to create complete timing models of the system. From such models, we can then predict (for example) the levels of jitter that we will see in the release times for all system tasks.

It is – of course – crucial that such timing models are correct. By making measurements of task release jitter from the system, we can perform a "sanity check" on the results from our modelling process.

In TTRD05d, we provide support for measurement of task jitter using another instrumented scheduler.

Code Fragment 22 shows part of the scheduler "Dispatcher" function in which the timing measurements are made.

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

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

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

         if (Index == Instrumented_task_id_G)
            {
            // Measure task jitter
            SCH_MEASURE_JITTER_Stop_Timer0();
            Timer_reading = SCH_MEASURE_JITTER_Read_Timer0();
            SCH_MEASURE_JITTER_Start_Timer0();
            }

         (*SCH_tasks_G[Index].pTask)();  // Run the task


         if (Index == Instrumented_task_id_G)
            {
            // Now process the data
            SCH_MEASURE_JITTER_Update_Jitter_Record(Timer_reading);
            }
```

Code Fragment 22: Part of the "Dispatcher" function in TTRD05d in which measurements of task release jitter are recorded.

5.9. Example: The impact of task order on a TTC scheduler

Consider Figure 39.

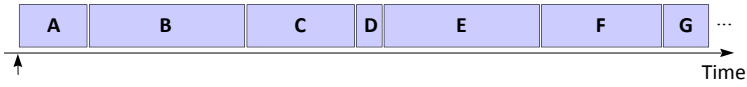


Figure 39: Part of a TTC schedule in which 7 tasks are executed in a single tick interval.

In general (as noted in Chapter 4), we would expect that the levels of task release jitter would be lowest for Task A and highest for Task G in this figure. Using TTRD05d, we can perform a test to see if a demo system conforms to this expected result.

Figure 40 shows results from TTRD05d running on a test system.

```
Values are in nanoseconds (ns)
tick interval is 1000000 ns (= 1 ms)

Task jitter information (ns)
Index  Min time          Max time          Difference (Jitter)
000    000,999.999370    000,999.999370    000,000.000000
001    000,999.999370    000,999.999370    000,000.000000
002    000,009.999370    000,009.999370    000,000.000000
003    000,009.999370    000,009.999370    000,000.000000
004    000,000.983520    000,001.015250    000,000.031730
```

Figure 40: Results from TTRD05d.

The corresponding tasks were as shown in Code Fragment 23.

```
SCH_Add_Task(HEARTBEAT_Update, 0, 1000, 20, 0);
SCH_Add_Task(SPI_7_SEG_Update, 1, 1000, 20, 0);
SCH_Add_Task(SWITCH_SW3_Update, 2, 10, 20, 0);
SCH_Add_Task(REPORT_JITTER_Update, 3, 10, 20, 0);
SCH_Add_Task(USB_SERIAL_Update, 0, 1, 20, 0);
```

Code Fragment 23: The task set used to generate the data shown in Figure 40.

Note that we must record more than 1000 samples here (for each task), to ensure that we record the jitter correctly for the tasks with Index 4 and Index 5. Here we use 1005 samples. Again, please refer to Code Fragment 23 to see why this is necessary.

In this case, the results are as expected, because – with the exception of the USB_SERIAL_Update task, all of the tasks run immediately after the timer tick.

To confirm this behaviour, the task set shown in Code Fragment 24 can be executed.

```
SCH_Add_Task(HEARTBEAT_Update,     0, 1000, 20, 0);
SCH_Add_Task(SPI_7_SEG_Update,     0, 1000, 20, 0);
SCH_Add_Task(SWITCH_SW3_Update,    0, 10, 20, 0);
SCH_Add_Task(REPORT_JITTER_Update, 0, 10, 20, 0);
SCH_Add_Task(USB_SERIAL_Update,    0, 1, 20, 0);
```

Code Fragment 24: A revised task set.

This is now a synchronous task sets (with all tasks running for the first time in Tick 0). The results are shown in Figure 41.

```
Values are in nanoseconds (ns)
tick interval is 1000000 ns (= 1 ms)

Task jitter information (ns)
Index  Min time          Max time          Difference (Jitter)
000    000,999.999370    000,999.999370    000,000.000000
001    000,999.999290    000,999.999450    000,000.000160
002    000,009.978540    000,010.020200    000,000.041660
003    000,009.978540    000,010.020200    000,000.041660
004    000,000.974930    000,001.023810    000,000.048880
005    000,000.975250    000,001.023490    000,000.048240
```

Figure 41: The output generated by the revised task set shown in Code Fragment 24.

5.10. Traditional ways of obtaining task timing information

There are – of course – other ways of obtaining task timing data, in order to confirm the results obtained from the instrumented scheduler. For example, the traditional approach is to control a pin from each task, and use an oscilloscope to measure the timing (Figure 42).

5.11. Generating a Tick List on an embedded platform (TTRD05e)

In Chapter 4, we considered ways of analysing a Tick List.

One way of generating such a list is to create what is sometimes called a "Dry Scheduler": TTRD05e illustrates how this can be achieved.

Code Fragment 25 shows the timer ISR from TTRD05e. In this fragment, we reduce the effective tick rate (in the case by a factor of 10) to simplify the process of displaying the data.

Note that – as we are reporting over a USB connection (as detailed in Appendix 1) – we still need to call the USB update task every millisecond (or this connection will "timeout").

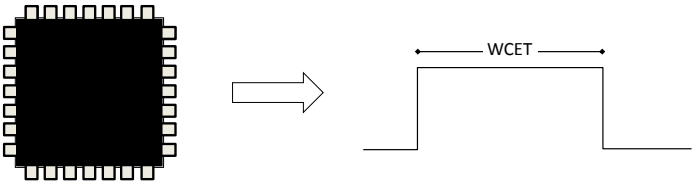


Figure 42: A traditional approach to obtaining task timing information involves controlling a microcontroller pin from each task and using an oscilloscope to measure the timing.

```
void SysTick_Handler(void)
   {
   // For reporting purposes
   static uint32_t Slow_count = 0;
   static uint32_t Tick_list_count = 0;
   static uint32_t Initial_delay = 0;

   // Need to call this to allow data reporting
   USB_SERIAL_Update();

   // Have an initial delay of 15 seconds
   // (to allow time to hook up USB, etc)
   if (Initial_delay++ < 15000)
      {
      return;
      }

   Initial_delay = 15000;

   // We reduce the tick rate
   // (to allow time to report the data)
   if (Slow_count++ < 100)
      {
      return;
      }

   // Increment tick count (used by scheduler)
   Tick_count_G++;

   // Tick_list_count is used to report Tick List
   if (++Tick_list_count == LENGTH_OF_HYPERPERIOD)
      {
      Tick_list_count = 0;
      Passes_G++;
      }

   // Report the current tick
   SCH_CREATE_TICK_LIST_Process_Tick_Number(Tick_list_count, Passes_G);
   }
```

Code Fragment 25: The timer ISR from TTRD05e.

Note also that we are displaying the first three hyperperiods in our output: in most cases, the task set will have reached a "steady state" by the time we reach this point (that is, the Task Sequence Initialisation Period will be complete): we discuss the TSIP in more detail in Chapter 10.

Code Fragment 26 shows part of the Dispatcher function from TTRD05e.

```
if (--SCH_tasks_G[Index].Delay == 0)
   {
   // The task is due to run

   // Report the Task ID (DON'T run the task)
   SCH_CREATE_TICK_LIST_Process_Index_Data(Index, Passes_G);
```

Code Fragment 26: Part of the Dispatcher function from TTRD05e.

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

```
*0001
0
3
4
*0002
1
3
4
*0003
2
3
4
*0004
3
4
*0005
3
4
…
```

Figure 43: Typical output from TTRD05e.

As should be clear from Code Fragment 26, no tasks are executed in this system (with the exception of the USB update task): instead, we simply report the task IDs.

Figure 43 shows part of a typical output (copied from a terminal emulator program).

5.12. Creating a Tick List that meets your requirements

There are other ways in which we can employ a form of Dry Scheduler.

Suppose, for example, that you have generated an initial Tick List (as outlined in Section 5.11) and analysed it (as outlined in Chapter 4). Further suppose that – as a result of this analysis - you find that you have not met the system requirements for maximum CPU load (as discussed in Section 4.10). What can you do?

As we discussed in Section 4.8, we can vary the task sequence in our system by changing the offsets. In fact, in designs with 20 or more tasks, we will usually begin the process of configuring the task schedule by trying to identify a combination of task offsets that will allow us to meet the basic CPU loading requirements.

We can achieve this as follows.

- We arrange the tasks in order of priority: in the absence of other considerations, this will typically mean that we follow a "rate monotonic" model and list the tasks in order of their periods (shortest period results in highest priority).
- Tasks will be added to the schedule one at a time, in priority order (highest priority first).

- Each task is initially added to the schedule with an offset of 0.
- We generate the Tick List (for example using the techniques presented in Section 5.11) for the current task set. To be clear: the first time we generate the Tick List there will be just one task in the task set.
- We review the resulting Tick List over the minimum interval: *maximum task offset value + length of the major cycle*
- For each tick in the above interval, we check whether the CPU load exceeds our required threshold (e.g. 80% of the available CPU time).
- If we have met the requirements (as we would expect to do for the first task) then we repeat the above process for the next task in the list. That is, we add the next task to the schedule along with any tasks that have already passed the above schedulability test.
- Once we have added all of the tasks from the set in this way, then our schedule is complete.[14]
- In the likely event that we find that we cannot add a given task to the schedule with offset 0, we remove this task, increase the offset to 1 and try again. We keep repeating this process until we find either [i] that the task can be added with a given offset, or [ii] we find that our offset has increased to a value greater than the hyperperiod: at this point, we have to conclude that we cannot add this task to the schedule successfully.

Overall, this process is both simple and effective. It can be applied manually (for small task sets). For larger task sets, it can be automated, using a Dry Scheduler as the starting point.

Note that the same procedure can be followed in order to ensure that the tasks are added in a manner that meets CPU load requirements and other constraints (such as task jitter).

5.13. Conclusions

In this chapter, we have explored techniques for measuring the execution time of your tasks, and for measuring jitter (both for your tasks and for the scheduler itself). We have also considered ways in which you can generate a Tick List.

In Chapter 6, we consider some important aspects of task design for TT systems.

[14] It – of course – possible that all of the tasks can be added with an offset of 0: we then have a synchronous task set (and it is unlikely that we would have found it necessary to carry out this procedure in the first place …).

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

CHAPTER 6: Timing considerations when designing tasks

In this chapter, we consider the design and implementation of effective tasks for use in a TT system. Our focus is on timing considerations.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD06a: TTC "Super Loop" scheduler with hardware delay
- TTRD06b: Implementing "sandwich delays"

6.1. Introduction

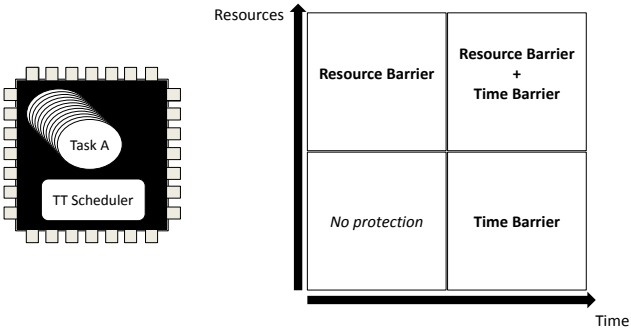


Figure 44: A typical TT platform [left], for which we need to ensure "From From Interference" (FFI), and [right] a schematic representation of barriers that can be used to ensure FFI.

In Chapter 1, we argued that the ability to detect interference between tasks, and between tasks and the scheduler, is a key design challenge for developers of non-trivial embedded systems (Figure 44, left). We summarised this situation by saying that the tasks and scheduler should be able to operate in an environment in which "Freedom From Interference" (FFI) is guaranteed, and we suggested that it can be helpful to break the FFI challenge down into the areas of time and resources (Figure 44, right).

In this chapter, we will begin to consider the issues involved in addressing FFI concerns. Our focus will be on the creation of tasks with deterministic timing behaviour.

When dealing with task timing in an embedded system, two stages need to be addressed: [i] we must design our tasks so that the chances of timing problems are reduced; [ii] we must monitor the system at run time in order to check that our assumptions have been met.

We will consider the monitoring of task execution times in Chapter 9. In the present chapter, our focus will be on the process of task design.

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

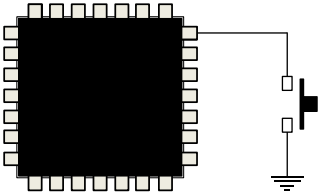


Figure 45: A simple switch connection to a microcontroller (without isolation).

6.2. Design goal: "Short Tasks"

As we discussed in Chapter 2, TTC designs are assembled from Short Tasks. Short Tasks have a worst-case execution time (WCET) that is: [i] less than the period of any task in the system; and [ii] less than the system tick interval.

In general, it is possible to use a TTC architecture if (and only if) if we can meet the Short Task requirement for all tasks in the system.

In this chapter we will consider design approaches that can help us to [i] meet the Short-Task requirement; and [ii] meet the requirements for deterministic timing behaviour that were outlined in the introduction to this chapter. Many of the discussions in this chapter will also be relevant when we begin to consider designs that support task pre-emption (in Part Three).

6.3. The need for multi-stage tasks

In Chapter 4, we discussed an "emergency stop" system that involved reading from a switch interface (see Figure 32).

As discussed in Chapter 4, the code associated with checking the switch state will often need to be able to deal with "switch bounce", resulting in an architecture like that shown in Listing 16 (on p.86).

This type of switch interface is an example of a general architecture that is employed in TT systems: this is known as a "multi-stage task".

This multi-stage architecture typically involves [1] storage of the task state; [2] periodic checking the status input / output line, a hardware component or a data value; [3] updating the task state based on the checks performed in Step 2; and [4] responding appropriately to any change in status of the monitored hardware / software components.

The key to employing a multi-stage architecture effectively is to appreciate that it does **not involve waiting**. That is, rather than (for example) sitting in a delay loop while a hardware action completes – such as a switch bouncing, or a communication peripheral sending part of a message – we store the current task state and go and run another task (or put the processor to sleep):

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

```
void Multi_Stage_Task(void)
   {
   static uint32_t State;

   // Check input / output?
   // Check hardware?
   // Check data?

   // Update State?
   if (State == X)
      {
      // Do something
      }
   }
```

Code Fragment 27: Pseudo code for a "multi-stage task" architecture. See text for details.

we then release the current task again (typically a few milliseconds later), restore the state and we carry on from where we left off.

This multi-state architecture is represented in pseudo-code in Code Fragment 27.

6.4. Example: Measuring liquid flow rates

As an example of a design for a multi-stage task, we will consider a simple sensor interface.

Various forms of sensors generate pulse streams where the pulse rate is used to indicate a physical quantity. For example, suppose we need to measure the rate of liquid flow through a pipe (Figure 46).

Typically, we need to know what the current flow rate is. For example, suppose (in Figure 46) we need to know how many litres of water are passing through the pipes in an industrial cooling system every minute. With some information about the sensor, we can determine this value by counting how many pulse have occurred in (say) 100 ms, and extrapolating from this measure.

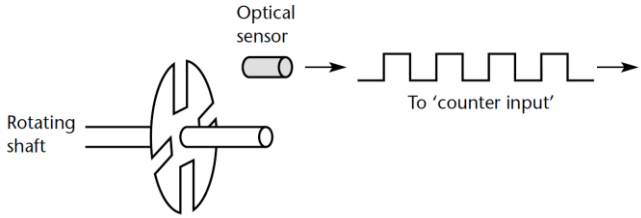


Figure 46: Measuring rotational speed from an optical encoder as a means of measuring liquid flow rates through a pipe.

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

```
void FLOW_RATE_SOFTWARE_Update(void)
   {
   static uint32_t Pulse_count = 0;
   static uint32_t Time_count = 0;
   static uint32_t Previous_input = LO;
   uint32_t Current_input;

   // Read state of input pin
   // Current_input = …

   if ((Previous_input == HI) && (Current_input == LO))
      {
      // Pulse detected …
      Pulse_count++;
      }

   // Assume task called every millisecond
   // Assume that we report (and reset) count every 100 ms
   if (++Time_count == 100)
      {
      // Update global pulse count (used by other tasks)
      Pulse_count_G = Pulse_count;

      // Reset local counters
      Pulse_count = 0;
      Time_count = 0;
      }
   }
```

Code Fragment 28: Pseudo code for flow-rate measurement (software based): an implementation of a multi-stage task.

To achieve this, we could use a software task to read the pulse stream and check for rising (or falling) edges. This can work effectively, if the flow rates are low, or we have the capacity to run very frequent tasks.

This form of software-based pulse counting could be implemented as a multi-stage task, as illustrated in Code Fragment 28.

Where the pulse rates are higher, we would typically use on-chip counter hardware to assist with these measurements. This then becomes a form of "parallel processing" (true multi-tasking), since the counter hardware will take care of the pulse counts while our CPU can do something more useful.

In such a design, our multi-stage task might then be set to read from a counter unit every 100 ms, calculate the flow rate and reset the counter. Values for "maximum", "minimum" and "average" flow rates might be stored, in order to support various checks.

The task would typically be very short, and would not require any delay code.

When using the LPC1769, all four timers (Timer 0 to Timer 3) are designed to count external pulses in this way.

6.5. Example: Buffering output data

As another example of a design for a multi-stage task, we will consider the need to transmit large amounts of data from a TT design.

Suppose – for example – that we wish to transfer data to a PC at a standard 9600 baud over an RS-232 link. Transmitting each byte of data, plus stop / start bits, typically involves the transmission of 10 bits of information. As a result, each byte takes approximately 1 ms to transmit.

Now, suppose we wish to send this information (character string) to the PC:

```
"Current core temperature is 36.678 degrees"
```

If we use a standard function (such as some form of printf()) the task sending these 42 characters will take more than 40 milliseconds to complete. In a system supporting task pre-emption, we may be able to treat this as a low-priority task and let it run as required. This approach is not without difficulties (for example, it may be very wasteful of CPU resources, as we will discuss shortly). However, with appropriate system design we will be able to make this operate correctly under most circumstances.

Now consider the equivalent TTC design. We can't support task pre-emption and a long data-transmission task (around 40 ms) is likely to cause immediate and significant problems. More specifically, the WCET of this task is likely to be much greater than the system tick interval (Figure 47).

Perhaps the most obvious way of addressing this issue is to increase the baud rate; however, this is not always possible, and – even with very high baud rates – long messages or irregular bursts of data can still cause difficulties.

The problem here is not really the speed of transmission (although this is certainly an issue). The real problem it is that the data transfer operation has a duration which depends on the length of the string which we wish to submit. As such, the execution time of the data-transfer task is highly variable (and, in a general case, may depend on conditions at run time). This makes determination of WCET very challenging.

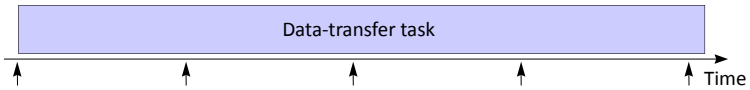


Figure 47: A schematic representation of the problems caused by sending a long character string on an embedded system. In this case, sending the message takes 42 ms while the tick interval is 10 ms.

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

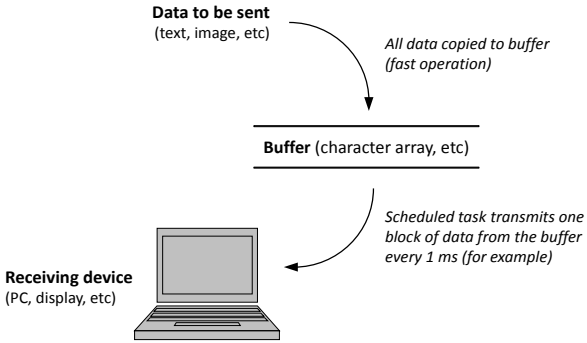


Figure 48: An overview of a "buffered output" architecture.

Note that this problem can arise in <u>any</u> system that involves transfer of comparatively large amounts of data, over (say) CAN links, or to a graphics display, or from an image sensor.

In all of these cases, we want to convert a long data-transfer task (which is called comparatively infrequently and may have a variable duration) into a periodic task (which is called comparatively frequently and which has a short – <u>and known</u> – duration).

The end result is a form of multi-stage task with a software or hardware buffer: it is usually simply called a "buffered output" architecture.

Such a "buffered output" will typically involve the following:

- A buffer (usually just an array, implemented in software).
- A function (or small set of functions) which can be used by the tasks in your system to write data to the array.
- A periodic (scheduled) task which checks the buffer and sends a block of data to the receiving device (when there are data to send).

Figure 48 provides an overview of this system architecture. All of the data to be sent are first moved to a software buffer (a very fast operation). The data is then shifted – one block at a time – to the relevant hardware buffer in the microcontroller (e.g. 1 byte or 16 bytes at a time for a UART, 8 bytes at a time for CAN, etc): this software-to-hardware transfer is carried out every 1ms (for example), using a short periodic task.

Note that the total CPU load for this buffered-output architecture is very low.

The USB interface included in TTRD05a illustrates a form of buffered output. Other examples of this architecture in use are given in "PTTES", Chapter 18 (an RS-232 library) and in Chapter 22 (an LCD library).

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

6.6. Example: DMA-supported outputs

The "buffered output" design described in Section 6.5 involves structuring our tasks in order to "spread out" the processor load. Another way of minimising the software load on the main processor is to transfer as much work as possible to specialised hardware resources.

One effective way in which we can achieve this is through the use of "Direct Memory Access" (DMA) support in our processor platform.

DMA-supported inputs have been commonplace for some time on many platforms: DMA-supported outputs are now also available.

As an example of what can be achieved on the LPC1769 platform, NXP Application Note AN10917 demonstrates how sine waves (pure tones) can be generated via the MCU's digital-to-analogue converter (DAC) without requiring use of the CPU.

6.7. The need for timeout mechanisms

Timeout mechanisms are a core requirement in many tasks.

As an example, of the need for such a mechanism, consider a simple task that has been created to read an analogue value from a potentiometer (Figure 49), as part of a system for controlling the heating in a domestic environment (Figure 50).

Reading the ADC might involve some code like that shown in Code Fragment 29.

Typically the ADC will have been configured in an "Init" function (called before the scheduler was started).

This code structure is very common. It is (usually implicitly) assumed that the ADC is fully operational, and that it has been configured correctly. Having triggered the ADC conversion we wait (for a few microseconds on a modern processor) for this conversion to complete.

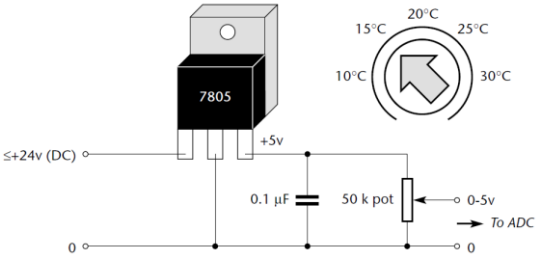


Figure 49: Using an analogue voltage value (read by an analogue-to-digital converter: ADC) to as part of the user interface on a domestic heating system.

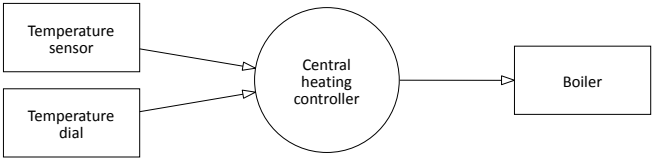
http://www.safetty.net/publications/the-engineering-of-reliable-embedded-systems

```
void ADC_Init()
   {
   // Initialise ADC
   vADC_Init();
   }

void ADC_Update()
   {
   ...

   // Start ADC conversion
   LPC_ADC->ADCR |= (1UL << 24);

   // Bit 31 is the "done" bit, set to 1 when conversion is complete
   while ((LPC_ADC->ADGDR & 0x80000000UL) == 0);
   ...
   }
```

Code Fragment 29: Sample ADC code (without a timeout mechanism). Builds on code example in NXP App Note AN10974.

In many ways, this seems to be an appropriate design. There are, however, some risks here. For example, the ADC may not have been configured correctly (or at all), or it may have been damaged (for example, by an out-of-range voltage input): see Figure 51. In such an event, the associated task may "hang" while waiting for the conversion to complete.

Given the soft timing constraints, this is unlikely to present a serious problem in the central-heating control system, but – if the task (or the approach to reading from the ADC component) - is re-used in a different design, then there may be more serious implications.




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

   while(1) // 'for ever' (Super Loop)
      {
      C_HEAT_Get_Required_Temperature(); // Get input from the user
      C_HEAT_Get_Actual_Temperature();   // Get current room temperature
      C_HEAT_Control_Boiler();           // Adjust the gas burner
      }

   return 1;
   }
```

Figure 50: The framework for a central-heating controller for a domestic environment. Note that – given the "soft" timing constraints – this design uses a very simple "Super Loop" architecture (the simplest practical TTC implementation: TT00).

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

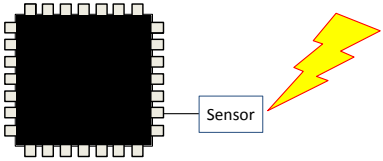


Figure 51: Possible accidental damage that might lead to a sensor failure.

For example, in many designs we need to consider the possibility that a deliberate attempt may be made to disrupt the system operation by interfering with an interface component (Figure 52).

In this (and in many other forms of hardware interface), we need some form of "time out" mechanism, that will abort the connection if the ADC does not complete its conversion within the expected time limit.

Note that it might be argued that any ADC problems are unlikely, and that – to deal with these – we can simply add WDT support (Code Fragment 30), ensuring that the system will be reset in the event that such problems do occur.

Let's assume that we do add WDT support. This can be thought of as a "system timeout" mechanism (as opposed to a "task timeout" mechanism). At best, in the event of a fault in the ADC example, we will move the system into a "fail silent" state (Figure 53).

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

   while(1)
      {
      C_HEAT_Get_Required_Temperature(); // Get input from the user
      C_HEAT_Get_Actual_Temperature();   // Get current room temperature
      C_HEAT_Control_Boiler();           // Adjust the gas burner
      C_HEAT_Feed_WDT ();                 // Feed watchdog
      }
   return 1;
   }
```

Code Fragment 30: A version of the central-heating controller with WDT support.

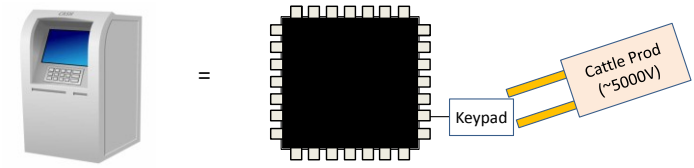


Figure 52: Possible deliberate damage that might lead to a sensor failure.
ATM image from: www.security-faqs.com

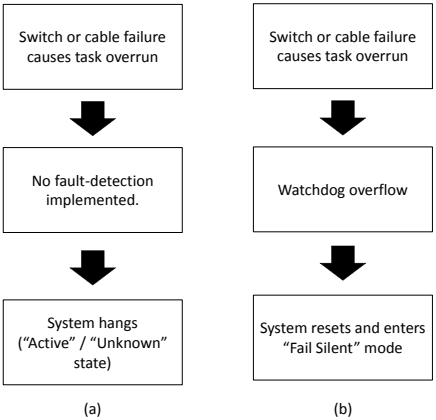


Figure 53: Possible impact of the failure of the ADC interface in designs (a) without WDT support, and (b) with WDT support.

In some circumstances, failing silently in event of problems with the WDT may – indeed – be appropriate behaviour. However, in many designs, we would wish to try to recover from this problem and keep the system running. This might mean (for example) switching to a backup ADC, or using a different kind of sensor to obtain the ADC information. It might mean assuming a "default" ADC value for a few seconds while we try to shut down a system in a controlled manner.

To give us the flexibility to address these issues, we need to "think globally, act locally", and use some form of timeout mechanism around the hardware interface.

6.8. Example: Loop timeouts

A Loop Timeout (as shown in Code Fragment 31) provides a simple timeout mechanism.

```
uint32_t Timeout_loop = 0;
...
// Wait for test condition with built in timeout
while (((Test condition) == 0) && (++Timeout_loop != 0));
```

Code Fragment 31: An example of a "Loop Timeout" mechanism.

With 32-bit integers (as shown), the loop will run a maximum of 4,294,967,295 times before it exits. This value can – of course – be changed by altering the initial value of the variable Timeout_loop.

The timeout duration must – of course – be measured (and it may depend on particular compiler optimisation settings, etc).

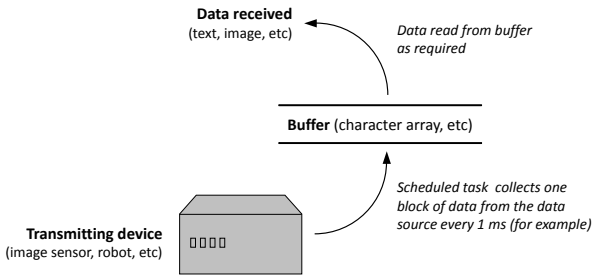


Figure 54: Setting up a "buffered input".

6.9. Example: Hardware timeout

Loop timeouts are effective (and better than having no timeout mechanism), but obtaining precise timing is difficult, and porting the code requires care.

Use of a hardware timeout offers an alternative with much more precise control. The foundation of such a mechanism is a timer-based delay: use of Timer 0 in the LPC1769 to generate such a delay is illustrated in Listing 18 and Listing 19.

We can build on such code to generate a timeout using the approach illustrated in Code Fragment 32.

```
// Set Timer 0 to overflow (and set flag) after 20 µs
// [See Listing 18 for details]

// Wait for test condition with hardware timeout
while (((Test condition) == 0) && ((LPC_TIM0->IR & 0x01) != 0x01));
```

Code Fragment 32: An example of a hardware timeout mechanism.

6.10. Handling large / frequent data inputs

In Section 6.5, we explored one way of transmitting large blocks of data from an embedded system using a "buffered output". If our system is to receive data, we can start by reversing this process (Figure 54).

We consider some possible implementations in the following sections.

6.11. Example: Buffered input

The simplest way of spreading and reducing the load involved in receiving large amounts of input data is to use a hardware buffer.

For example, the LPC1769 supports a 16-byte "First In First Out" (FIFO) buffer on the UART interface: this can significantly reduce the processor load.

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

6.12. Example: DMA input

As with output buffering, DMA support can also reduce the load when receiving input data.

Table 7 lists the various interfaces (inputs) on the LPC1769 that have DMA support.

Table 7: DMA support in the LPC1769 device (adapted from NXP User Manual, Table 544)

Peripheral Function	Request
SSP0 Rx	Dedicated DMA request
SSP1 Rx	Dedicated DMA request
ADC	DMA request or interrupt request
I2S channel 0 - 5	Dedicated DMA request
I2S channel 1 - 6	Dedicated DMA request
UART0 Rx / MAT0.1	Dedicated DMA requests
UART1 Rx / MAT1.1	Dedicated DMA requests
UART2 Rx / MAT2.1	Dedicated DMA requests
UART3 Rx / MAT3.1	Dedicated DMA requests

6.13. Example: Multi-core input ("Smart" buffering)

Our overall goal is to remove as much load as possible from the CPU, in order to speed up the processing and provide deterministic performance. As outlined above, one effective way of achieving this is to make use of specialised hardware on the processor (such as hardware buffers or DMA support).

Another option in these circumstances (and one that is becoming increasingly practical) is to use a dual-core or multi-core processor. In these circumstances, we may choose to have one main processor and a second processor providing support.

For example, the NXP LPC54102 processor has two cores (a Cortex-M4F and a Cortex-M0+ core). Both of these cores are configured as "masters" on the Advanced High Performance Bus (AHPB) bus in the chip: this means that they both have access to all of the available resources (peripherals and memory). The two cores can be easily synchronised, allowing us to create a complete TT architecture on the device (avoiding conflicts over the shared resources).

NXP Application Note AN11609 describes how to use the two cores on this device.

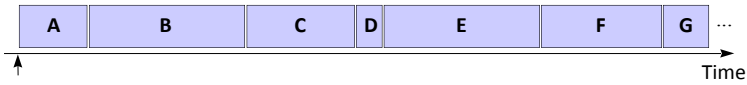


Figure 55: Part of a TTC schedule in which 7 tasks are executed in a single tick interval.

6.14. Example: Customised hardware support

Where specialised hardware is not available, use of a second core is one way of reducing the load on the main processor. Another option is to develop hardware that meets the particular needs of your application.

Specialised hardware can be, for example, be designed using VHDL and implemented using an external Field-Programmable Gate Array (FPGA) or Complex Programmable Logic Device (CPLD). The link to the main processor might then be implemented using an SPI link (for example).

Another alternative is to employ an FPGA devices that incorporates a "soft" or "hard" processor core. There are now many such devices available.

For example, we have used a Cortex-M3 processor core in the examples in this book. Microsemi® produce a SmartFusion2® device that incorporates both a (hard) Cortex-M3 core and additional FPGA logic on a single processor platform. This type of platform opens up a number of new design options.

6.15. Execution-time balancing in TTC designs (task level)

Please consider Figure 55. In general (as noted in Chapter 2), we would expect that the levels of task release jitter would be lowest for Task A and highest for Task G in this figure, because of the variations in the execution time of the various tasks that run after each tick.

Where we need control these jitter levels, we typically record the ratio between WCET and BCET for each task (Figure 56) and – for tasks with a high ratio (typically around 3 or greater) – we may try to "balance" the task by trying to bring the BCET and WCET figures more in line.[15]

$$\frac{\text{WCET}}{\text{BCET}}$$

Figure 56: The ratio between the WCET and BCET of a task can be a useful measure during system development (and cam be used as a means of highlighting tasks that may benefit from "balancing").

[15] Techniques for measuring task execution times and jitter were considered in Chapter 5.

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

6.16. Execution-time balancing in TTC designs (within tasks)

As we have seen already in this chapter, using a TT architecture allows a developer to ensure that some or all of the tasks in the system begin executing at precisely-determined points in time. However - in some real-time systems – care must still be taken when designing and implementing the tasks in order to ensure that all of the system timing requirements are met.

To understand why this is necessary, please consider Code Fragment 33.

```
void Task_A(void)
   {
   /* Task_A has a known WCET of A milliseconds */

   // Read inputs

   // Perform calculations

   /* Starting at t <= A ms */
   // Generate outputs

   /* Task_A completes within A milliseconds */
   }
```

Code Fragment 33: An example of execution-time balancing at the task level.

In this case, the worst-case execution time (WCET) of the task is known. This means that we know (only) that the task will generate certain outputs within A ms from the start of the task (where A is the WCET of the task).

The listing below shows an alternative implementation of the task:

```
void Task_A(void)
   {
   /* Task_A has a known WCET of A milliseconds */
   /* Task_A is balanced */

   // Read inputs (KNOWN AND FIXED DURATION)

   // Perform calculations (KNOWN AND FIXED DURATION)

   /* Starting at t = A1 ms
      (measured from the start of the task),
      for a period of A2 ms, generate outputs */

   /* Task_A completes after (A1+A2) milliseconds */
   }
```

Code Fragment 34: An example of execution-time balancing within a task.

In this alternative implementation, the code within the task has been balanced. Slightly more formally this means that we know both the "worst case" (WCET) and "best case" (BCET) execution times of this task, and that BCET = WCET. We also know the timing of activities of certain activities that occur within the task (that is, we know when these activities will occur at a time measured from when the task begins executing). For example, in Code Fragment 34, the task outputs will be generated in an interval starting A1 ms after the start of the task and finishing A2 ms later.

We'll explore some different techniques for balancing the execution time of tasks in the sections that follow.

6.17. ASIDE: Execution-time balancing in TTH / TTP designs

In this chapter, we have a focus on TTC designs.

In designs that involve task pre-emption, we may also wish to balance the code, in order to reduce (or eliminate) problems caused by "priority inversion": we discuss this process in Chapter 12.

In any system that involves task pre-emption and run-time monitoring, we may also use execution-time balancing to maximise the effectiveness of the monitoring process: again, please see Chapter 12 for further details.

6.18. Example: Execution-time balancing at an architecture level

We have already considered the use of multi-stage tasks in this chapter.

Use of such an architecture can be seen as a way of "smoothing out" the execution times of periodic tasks, so that the workload (and task execution time) is similar every time the task executes (Figure 57).

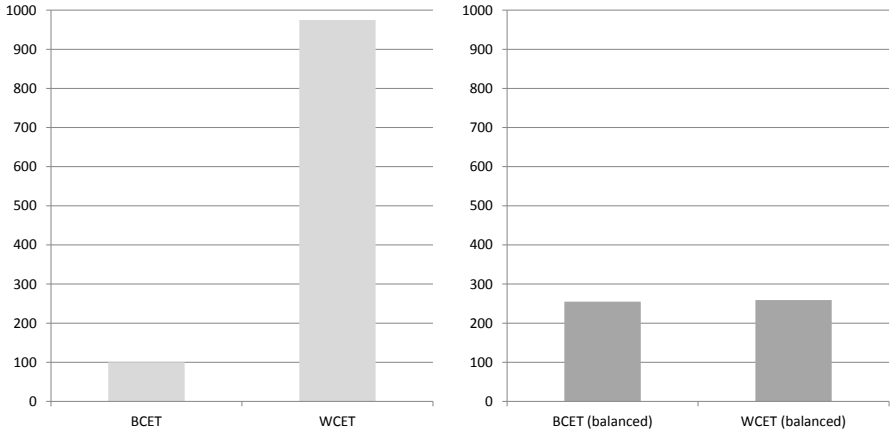


Figure 57: The impact of use of a multi-stage task on the BCET and WCET of a data-transfer task: (Left) no multi-stage architecture; (Right) with a multi-stage architecture.

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

Note that – used in this way – execution-time balancing can greatly improve the system efficiency, because we do not need to "allow space in the schedule" for a task with a long WCET.

Overall, selection of an appropriate task architecture is often the best way of starting the process of execution-time balancing.

6.19. Example: Manual execution-time balancing

Low-level execution-time balancing often starts by looking at conditional statements in the code.

For example, consider Code Fragment 35

```
void PROCESS_DATA_Update(void)
   {
   if (Data_set_1_ready_G)
      {
      Process_Data_Set_1();  // WCET = 0.6 ms
      }

   if (Data_set_2_ready_G)
      {
      Process_Data_Set_2();  // WCET = 0.2 ms
      }
   }
```

Code Fragment 35: An example of unbalanced code.

In Code Fragment 36, we present a more balanced version of this code, this time using delays.

```
void PROCESS_DATA_Update(void)
   {
   if (Data_set_1_ready_G)
      {
      Process_Data_Set_1();  // WCET = 0.6 ms
      }
   else
      {
      Delay_microseconds(600);
      }

   if (Data_set_2_ready_G)
      {
      Process_Data_Set_2();  // WCET = 0.2 ms
      }
   else
      {
      Delay_microseconds(200);
      }
   }
```

Code Fragment 36: A balanced version of the code shown in Code Fragment 35.

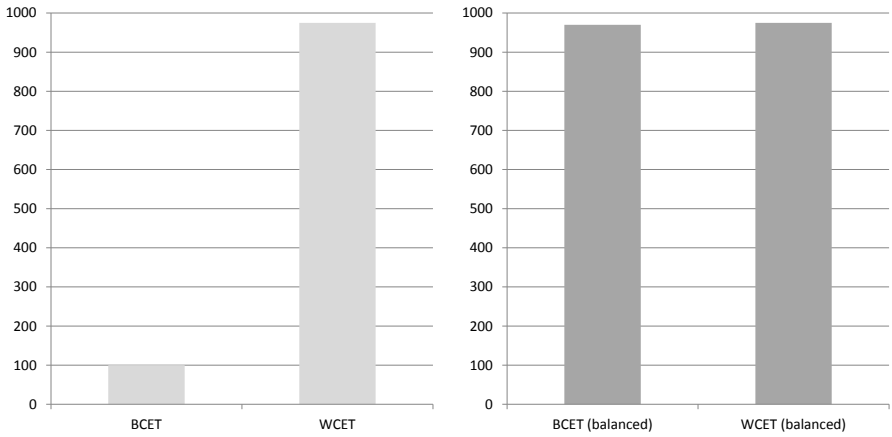


Figure 58: The impact of low-level execution-time balancing.

Note that code-level solutions tend to "balance up" the execution time (Figure 58). This should be compared with architectural solutions, such as the use of multi-stage tasks, which tend to "balance down" the execution time (see, for example, Figure 57).

6.20. Example: Sandwich delays for execution-time balancing

The code shown in Code Fragment 36 may be "more balanced" but – unless the functions Process_Data_Set_1() and Process_Data_Set_2() have themselves been balanced, we may not have helped the situation very much.

One option – therefore – may be to look at the code for the data-processing functions, and balance this. In some circumstances, use of a "Sandwich Delay" may offer another option.

A "Sandwich delay" provides a simple and effective means of ensuring that a particular block of code always takes the same period of time to execute: this is done using two timer operations to "wrap up" the activity you need to perform.

The operation and use of this technique is best illustrated using an example: please refer to Code Fragment 37. In this example, the functions SANDWICH_DELAY_T3_Start() and SANDWICH_DELAY_T3_Wait() are used together to implement the required delay. More specifically, SANDWICH_DELAY_T3_Start() sets up a timer to overflow after an interval of 1 second. The function HEARTBEAT_Update() is then called, after which – by means of SANDWICH_DELAY_T3_Wait() – we wait until timer overflows before continuing. In this way we ensure that the Heartbeat LED changes state approximately once per second.

```
while(1)
   {
   // Delay value in microseconds (1 second)
   SANDWICH_DELAY_T3_Start(1000000);

   HEARTBEAT_Update();

   SANDWICH_DELAY_T3_Wait();
   }
```

Code Fragment 37: Use of a "Sandwich Delay" to flash an LED at 0.5 Hz.

Code for the Sandwich Delay functions is presented in Listing 20.

Note that no interrupts are employed in Sandwich Delays.

6.21. Appropriate use of Sandwich Delays

Sandwich Delays can be an effective solution, but they can present two challenges.

First, Sandwich Delays must be used with care in designs that involve task pre-emption, since – by default - the timer involved will keep running if the task is interrupted. Without due care, this can mean that the time limit is reached while the task concerned has been pre-empted. The preferred way of dealing with this issue is to increase the delays in a given task to take account any pre-emptions by tasks of higher priority: this is generally both easy to model and implement.

The second issue with Sandwich Delays is that they may mask underlying Software Faults or Hardware Faults in the system (including Deliberate Software Changes). For example, it may be possible to run virus code during the Sandwich Delay: if the system is using execution-time measures as a means of detecting such a Deliberate Software Change[16], then the execution of the virus code may not be detected.

Use of manual execution-time balancing can sometimes make it easier to detect both software faults and hardware faults by means of execution-time monitoring.

6.22. Conclusions

In this chapter, we've considered the design and implementation of effective tasks for use in a TT system.

In Chapter 7, we explore ways in which we can implement systems with multiple operating modes.

[16] We discuss such processes in Chapter 9, Chapter 10 and Chapter 12.

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

6.23. Code Listings (TTRD06a)

```
/*------------------------------------------------------------------*-

   main.c (Released 2015-01)

  --------------------------------------------------------------------

   main file for TT project.

   Simple "TTC Super Loop" example.

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

// Project header
#include "main.h"

// Task headers
#include "../tasks/heartbeat_1769_001-2_c02a.h"
#include "../tasks/hardware_delay_lpc1769_t0.h"

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

int main(void)
   {
   // Prepare Heartbeat LED
   HEARTBEAT_Init();

   while(1)
      {
      Hardware_Delay_us(1000000);
      HEARTBEAT_Update();
      }

   return 1;
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 18: TTRD06a (main.c)

```
/*------------------------------------------------------------------*-

  hardware_delay_lpc1769_t0.c (Released 2015-01)

  ------------------------------------------------------------------

  Hardware delay function for LPC1769.

  This version uses Timer 0.

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

// Project header
#include "../main/main.h"

// Links to CMSIS library for Timer 0
#include <lpc17xx_timer.h>
```

Listing 19: TTRD06a (hardware_delay_lpc1769_t0.c) [Part 1 of 2]

```
/*------------------------------------------------------------------*-

  Hardware_Delay()

  Hardware delay (in microseconds), based on Timer 0.

-*------------------------------------------------------------------*/
void Hardware_Delay_us(const uint32_t DELAYmicroseconds)
   {
   // Used to configure Timer 0
   TIM_TIMERCFG_Type TMR0_Cfg;
   TIM_MATCHCFG_Type TMR0_Match;

   // Power-on Timer 0
   LPC_SC->PCONP |= 1 << 1;

   // Initialise Timer 0, prescale counter = 1 µs
   TMR0_Cfg.PrescaleOption = TIM_PRESCALE_USVAL;
   TMR0_Cfg.PrescaleValue = 1;

   // Use channel 0, MR0
   TMR0_Match.MatchChannel = 0;

   // Set flag when MR0 matches the value in TC register
   TMR0_Match.IntOnMatch = ENABLE;

   // Enable reset on MR0: TIMER will reset if MR0 matches it
   TMR0_Match.ResetOnMatch = TRUE;

   // Don't stop on MR0 if MR0 matches it
   TMR0_Match.StopOnMatch = TRUE;

   // Do nothing for external output pin if match
   TMR0_Match.ExtMatchOutputType = TIM_EXTMATCH_NOTHING;

   // Delay value
   // Set Match value, count value in microseconds in this version.
   TMR0_Match.MatchValue = DELAYmicroseconds;

   // Set configuration for Tim_config and Tim_MatchConfig
   TIM_Init(LPC_TIM0, TIM_TIMER_MODE, &TMR0_Cfg);
   TIM_ConfigMatch(LPC_TIM0, &TMR0_Match);

   // Start Timer 0
   TIM_Cmd(LPC_TIM0, ENABLE);

   // Wait for flag to be set
   while ((LPC_TIM0->IR & 0x01) != 0x01);

   // Clear flag
   LPC_TIM0->IR |= 1;
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 19: TTRD06a (hardware_delay_lpc1769_t0.c) [Part 2 of 2]

6.24. Code Listings (TTRD06b)

```
void SANDWICH_DELAY_T3_Start_us(const uint32_t DELAYmicroseconds)
   {
   TIM_TIMERCFG_Type TMR3_Cfg;
   TIM_MATCHCFG_Type TMR3_Match;

   // Power on Timer3
   LPC_SC->PCONP |= 1 << 23;

   // Initialise Timer 3, prescale counter = 1 µs
   TMR3_Cfg.PrescaleOption = TIM_PRESCALE_USVAL;
   TMR3_Cfg.PrescaleValue = 1;

   // Use channel 0, MR0
   TMR3_Match.MatchChannel = 0;

   // Set flag when MR0 matches the value in TC register
   TMR3_Match.IntOnMatch = ENABLE;

   // Disable reset on MR0: TIMER will not reset if MR0 matches it
   TMR3_Match.ResetOnMatch = FALSE;

   // Stop on MR0 if MR0 matches it
   TMR3_Match.StopOnMatch = TRUE;

   // Do nothing for external output pin if match
   TMR3_Match.ExtMatchOutputType = TIM_EXTMATCH_NOTHING;

   // Delay value
   // Set Match value, count value in microseconds in this version.
   TMR3_Match.MatchValue = DELAYmicroseconds;

   // Set configuration for Tim_config and Tim_MatchConfig
   TIM_Init(LPC_TIM3, TIM_TIMER_MODE, &TMR3_Cfg);
   TIM_ConfigMatch(LPC_TIM3, &TMR3_Match);

   // Start Timer 3
   TIM_Cmd(LPC_TIM3, ENABLE);
   }

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

void SANDWICH_DELAY_T3_Wait(void)
   {
   // Wait for timer to overflow
   while ((LPC_TIM3->IR & 0x01) != 0x01);

   // Clear the flag
   LPC_TIM3->IR |= 1;
   }
```

Listing 20: TTRD06b. The functions needed to implement a Sandwich Delay.

CHAPTER 7: Multi-mode systems

In previous chapters, we have considered systems with two operating modes: "Normal" and "Fail silent". In this chapter we consider the design and implementation of systems with multiple operating modes.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD07a: TTC architecture: Nx3 operating modes
- TTRD07b: TTC architecture: Nx3 & "Fail Silent"
- TTRD07c: TTC architecture: Nx3, "Limp Home" & "Fail Silent"

7.1. Introduction

In the designs considered so far in this book, we have supported two system modes: a "Normal" mode and a (very simple) "Fault" or "Fail Silent" mode.

More commonly, a system will have several "normal" system modes: for example, a passenger car may operate in different modes when manoeuvring at low speeds and for motorway driving.

In this chapter, we consider techniques for creating reliable systems with multiple system modes.

We begin by defining more precisely what we mean by a change in system mode.

7.2. What does it mean to change the system mode?

In this book, we say that there is a change in the system mode if the task set is changed. Changing the task set, means that we either:

- change the tasks in the set, or
- keep the same tasks, but change the parameters of one or more of the tasks.

As an example, suppose that we have a set of periodic tasks – Task Set X – as follows:

Task Set X = {A(10 ms), B(15 ms), C(5 ms)}

In this case, we are representing each task with a task name (e.g. "A"), and with a period (e.g. 10 ms). Note that other parameters – e.g. execution time, offset, jitter, and so on – may (of course) be included in a more complete representation of a task set, but this representation will meet our requirements in this example.

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

In the list below, Task Set X, Task Set Y and Task Set Z are all different.

Task Set X = {A(10 ms), B(15 ms), C (5 ms)}
Task Set Y = {A(10 ms), B(15 ms), C (7 ms)}
Task Set Z = {D(10 ms), B(15 ms), C (5 ms)}

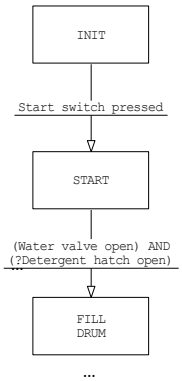
7.3. Mode change or state change?

When we discussed the first version of our controller for a washing machine in Chapter 3, we suggested that there should be a total of ten different system states and two different system modes (Figure 59).

There are other options. For example, we could create a version of the washing machine in which there were ten modes - each corresponding to a state in Figure 59 - and only one state in each mode.

While it is generally possible to implement a state change as a mode change, it will not usually make sense to do this, unless we need to vary the task set in the different modes. In the washer example, the only significant differences in the task set were in the FAIL_SILENT mode: other variations in the system operation were – therefore –probably best represented as state changes.

Overall, the code required to produce a multi-mode system is more complicated than the code required to produce a multi-state system: in the absence of a good reason to increase the system complexity, we would usually wish to employ the simpler solution.




```
// Ten system states
typedef enum
{INIT, START,
FILL_DRUM, HEAT_WATER,
PRE_WASH, MAIN_WASH,
DRAIN_DRUM, SPIN,
FINISHED,
FAULT}
eSystem_state;

// Two system modes
typedef enum
{FAIL_SILENT,
NORMAL}
eSystem_mode;
```

Figure 59: Modes and states (and a partial state-transition diagram) for our simple washing-machine controller.

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

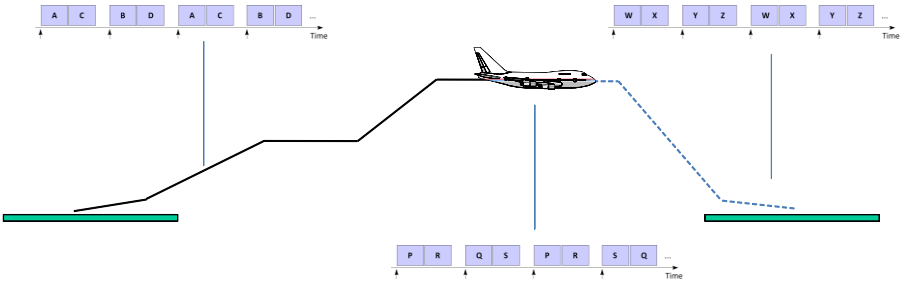


Figure 60: An example of a system with multiple operating modes.

7.4. The timing of mode changes

It should be reiterated that the timing of the transition between system modes is not normally known in advance (because, for example, the time taken for the plane shown in Figure 60 to reach cruising height will vary with weather conditions). That is, the transitions between modes are often event-triggered.

The ability to change modes dynamically in this way makes the design flexible and responsive (in a manner that is sometimes misunderstood by people new to this area, who assume that TT systems are completely static in nature).

Within each mode, of course, we retain all of the advantages of a simple TT design: the tasks are always released according to a schedule that is determined, validated and verified when the system is designed.

7.5. Implementing effective multi-mode designs

If we are to develop reliable multi-mode TT designs, we need to handle the transition between modes with care (see Figure 61).

In Chapter 2, we considered some challenges involved in traditional approaches to mode-changing in TT systems.

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

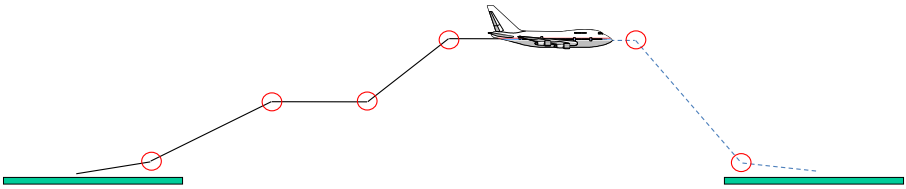


Figure 61: An aircraft flight path, highlighting the points at which the system mode changes.

This helps to ensure 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).

In addition to ensuring that we make a "clean" change between task sets, using a reset-based mode change: [i] allows us to use appropriate watchdog settings for each system mode (see Section 7.8); and [ii] allows us to integrate fault-handling mechanisms into the mode-change process very cleanly (a process that we will begin to explore in more detail in Chapter 8).

7.6. The architecture of a multi-mode system

The systems discussed in previous chapters of this book have all had a basic architecture based on that shown in Figure 62.

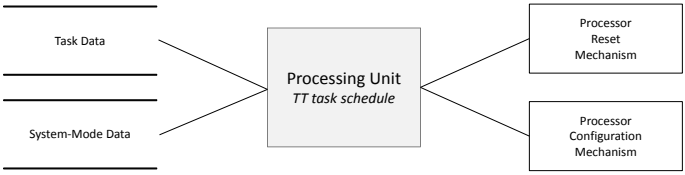


Figure 62: The foundation of the scheduler designs presented in this book.

This architecture (which was first introduced in Chapter 2) provides all of the mechanisms required for changing the system mode safely and reliably.

During the design process, we need to do the following:

- Identify the required system modes
- Identify the triggered required to transition between modes
- Identify the task set required in each mode

If desired, standard modelling tools (for example, state-transition diagrams or statecharts) may be employed to represent the different system modes.

In the absence of faults, the architecture operates as follows:

- The system is powered up in one of N pre-determined "normal" modes: (where N >= 1).
- In response to events that are internal or external to the system (including user inputs), the system may need to change to a different (normal) mode.
- Changing mode involves storing the required (new) system mode in the System-Mode Data store, triggering the Processor Reset Mechanism, then – following the reset – executing the Processor Configuration Mechanism.

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

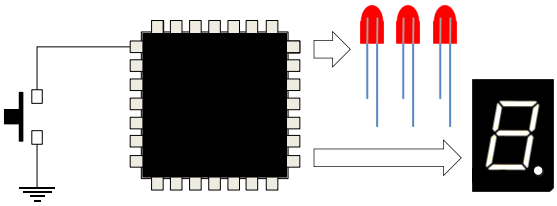


Figure 63: A schematic representation of the simple demo platform that can be used to demonstrate system mode changes using TTRD07a.

7.7. Different system settings in each mode (if required)

As each mode change involves a system reset and a complete (clean) change of mode, we can – if required - use different tick intervals and different WDT settings (as well as different task sets) in each mode.

We illustrate this in TTRD07a.

7.8. Design example with multiple Normal Modes (TTRD07a)

The interfaces involved in TTRD07a are shown schematically in Figure 63.

Example TTRD07a operates as follows:

- The system has three Normal Modes (Normal_01, Normal_02 and Normal_03).
- The 7-segment LED is used to display the current system mode.
- A different Heartbeat LED flashes in each mode.
- In the absence of any intervention, the system moves from Normal_01, to Normal_02 (and so on) every 20 seconds.
- A switch press causes the system to move at once to the next mode in the sequence.

In previous examples, the System-Mode Data has been stored in the WDT unit: this was possible because we only needed to identify watchdog-induced resets when the system started up (all other forms of reset caused the system to enter the Normal Mode).

In multi-mode designs, we need a way of storing (more) information before a system reset, and retrieving this information after the system restarts.

In TTRD07a, the function SYSTEM_Identify_Required_Mode() – see Listing 21 (p.156) – reads information from GPREG0 and GPREG1: these are NVRAM registers in the LPC1769 (see Box 6).

In this design, these registers are used to transfer information about the next required system mode across the "reset boundary".

Transferring data between modes on the LPC1769

In the LPC1769, we have access to several general-purposes registers (GPREGs) that are implemented in non-volatile RAM (NVRAM), as part of the real-time clock component. These GPREGs maintain their contents between resets if power is maintained to the system: this power can, if required, be supported by a small battery (typically a coin cell).

In the multi-mode designs presented in this book, we use two registers (GPREG0 and GPREG1) to transfer data about the required mode between resets. The two registers store the same information, but we "invert" the copy in GPREG1: this reduces the chances that a block of memory that has been erased (or where all memory locations have been set to the same value, such as 0xFFFFFFFF) will be seen as containing valid data.

Note that if a lot of data need to be transferred between modes (or the designs are ported to a processor without NVRAM), then an external battery-backed RAM device can be employed. Various I2C / SPI battery-powered RAMs are available (in both industrial and automotive temperature ranges, if required).

Note also that it may be tempting to consider use of a Flash or EEPROM as an alternative to RAM as a means of storing data between mode changes. Such a solution may avoid the need for a battery, but write times are likely to be longer than for RAM-based solutions, with the result that the time taken to change modes may increase (possibly to several milliseconds). In addition, it is important to remember that Flash and EEPROM devices wear out, typically after some 10,000 or 100,000 write operations with current technology. This may seem unimportant if the mode changes are rare (for example, once an hour): however, there are 8760 hours in a year, and – even if the memory device is replaced during annual service intervals – this is probably too close for comfort for a "10,000 write" component in a reliable embedded system. If mode changes can happen every minute (or less), or the product will not be serviced, then NVRAM becomes a particularly attractive option.

Box 6

Note that we need to store a default value in the GPREG registers (otherwise a fault-related reset may be interpreted as a planned reset): we can do this as shown in Code Fragment 38.

```
// Store "fail silent" mode as default (for *next* mode)
// => used in absence of any specific fault information
LPC_RTC->GPREG0 = FAIL_SILENT;
LPC_RTC->GPREG1 = ~(FAIL_SILENT);  // Inverted copy
```

Code Fragment 38: Setting a default value for the required system mode.

This completes the function SYSTEM_Identify_Required_Mode().

In function SYSTEM_Configure_Required_Mode() – Listing 21 – the information obtained from SYSTEM_Identify_Required_Mode() is used to "bring the system up" in the required mode.

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

As previously noted, it may be useful to be able to use different WDT settings and different tick intervals (as well as appropriate task sets) in each of the system modes. We illustrate some of these options in TTRD07a.

As in the first TTC scheduler examples that we looked at (in Chapter 2), our FAIL_SILENT mode does not use a timer-driven scheduler: instead we have a simple "Super Loop" design, that flashes a warning LED. Other options can – of course – be employed here.

The various Normal Modes are then configured.

The process of changing system modes is demonstrated by means of the function SYSTEM_Mode_Change_Demo(): see Listing 21. Through this function, the mode changes automatically every 10 seconds, unless Switch 3 is pressed on the system baseboard: if this switch is pressed and held down, then the mode will change every second.

7.9. Design example with fault injection (TTRD07b)

TTRD07b is very similar to TTRD07a, but includes an injected task overrun (generated by the HEARTBEAT3_Update() task) a few seconds after this task runs for the first time.

When this overrun occurs, it causes the WDT to overflow: after the resulting processor reset, the system moves into a Fail-Silent Mode.

Please refer to the TTRD for details.

7.10. The process of "graceful degradation"

On Page xv, we provided definitions for the phrases "Uncontrolled System Failure", "Controlled System Failure", "Limp-Home Mode" and "Fail-Silent Mode". You may find it helpful to refer back to these definitions before reading this section.

We can view a Controlled System Failure as a process of what is sometimes called "graceful degradation". As an example of what this means, suppose that we have detected some form of System Fault. We might then reset the system and try to carry on in the same mode, in the expectation that – if the fault was transitory – the system can best meet its requirements by continuing. If another System Fault is then detected within – say – an hour, we might then conclude that we should move the system into a Limp-Home Mode. Once in a Limp-Home Mode, detection of another fault might be cause to make the transition to a Fail-Silent Mode: in the absence of another fault, we may remain in the Limp-Home Mode until power is removed.

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

Figure 64: One scenario that needs to be considered when exploring shut-down modes.
Photograph licensed from: Dreamstime.com. Photographer: Peter Lovás.

If – having entered a Limp-Home Mode or Fail-Silent Mode - the system is restarted, we may force it to remain in an (inactive) Fail-Silent Mode, until it has been investigated by a suitably-qualified individual (either in person, or – in some circumstances – via a remote diagnostics system).

Please note that - if a system detects a fault and enters a Fail-Silent Mode, we always need to consider very carefully what should happen if the power is cycled. In many cases, we would wish to ensure that the system cannot simply be restarted (because it makes little sense to let someone use a system that is known to be faulty). In other circumstances, we may leave the user – or others - at risk if it is not possible to restart the system after a fault occurs. For example, leaving a vehicle stranded on a "level crossing" (Figure 64) or in the fast lane on a motorway are examples of scenarios that need to be considered.

We will consider these matters again in Chapter 13, and in Chapter 15.

7.11. Design example supporting graceful degradation (TTRD07c)

As an example of a system framework that supports graceful degradation, please consider Figure 65. This represents the architecture of TTRD07c.

TTRD07c has three Normal Modes: the behaviour and task sets: in the absence of faults, it operates in these modes just like TTRD07a and TTRD07b.

In TTRD07b, a WDT-induced reset is caused in the event of a task overrun. In TTRD07c, we follow a slightly different approach: we monitor the Fault_code_G variable, to see whether the scheduler has detected an overload situation (using techniques introduced in Chapter 2). If such a problem is detected, we move the system into a Limp-Home Mode.

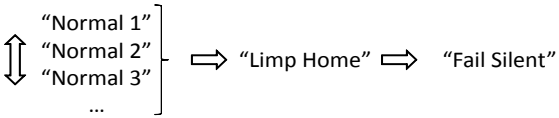


Figure 65: TTRD07c supports 3 Normal Modes plus a Limp-Home Mode and a Fail-Silent Mode.

The Limp-Home Mode is very simple in this example (it essentially flashes another LED). If further overloads are detected, we move into a "fail silent" mode, otherwise we remain in the Limp-Home Mode indefinitely.

Please refer to Listing 22 for further information.

7.12. Mode changes in the presence of faults

When considering mode changes in the presence of faults, "same mode" changes are often worthy of careful consideration. These involve resetting the processor and starting again in the same (normal) mode. If the problem is transitory, this can often be the most effective solution (and it is easily achieved using the frameworks presented in this chapter).

However, where sequences of mode changes are planned in the presence of faults (including more complicated graceful-degradation designs), it may be worth considering the use of an "external supervisor" or "WarranTTor Unit" to control the reset activities. We say more about this option in Chapter 14.

7.13. Conclusions

In this chapter we have explored the design and implementation of systems with multiple operating modes.

In Chapter 8, we move on to consider Task Contracts.

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

7.14. Code listings (TTRD07a)

```
/*-------------------------------------------------------------------*-

  system_mode_change_1769_001-2_c07a.c (Released 2015-01)

  Controls system configuration after processor reset.

  Demonstrates transition between modes (with mode-change reset).

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

// Project header
#include "../main/main.h"

// Task headers
#include "../tasks/heartbeat_basic_x3_1769_001-2_c07a.h"
#include "../tasks/watchdog_1769_001-2_c02a.h"
#include "../tasks/switch_ea-sw3_1769_001-2_c07a.h"
#include "../tasks/spi_7_seg_display_mode_1769_001-2_c07a.h"

// ------ Public variable definitions ------------------------------

// In many designs, System_mode_G will be used in other modules.
// - we therefore make this variable public.
eSystem_mode System_mode_G;

// ------ Public variable declarations -----------------------------

extern uint32_t Sw_pressed_G; // The current switch status (SW3)

extern uint32_t Fault_code_G; // System fault codes

// ------ Private function declarations ----------------------------

void SYSTEM_Identify_Required_Mode(void);
void SYSTEM_Configure_Required_Mode(void);
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 1 of 9]

```
/*------------------------------------------------------------------*-

  SYSTEM_Init()

-*------------------------------------------------------------------*/
void SYSTEM_Init(void)
   {
   SYSTEM_Identify_Required_Mode();
   SYSTEM_Configure_Required_Mode();

   // This design relies on the WDT to change modes
   // => we double check that the WDT is running ...
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      // Watchdog *not* running
      // => we try to shut down as safely as possible
      SYSTEM_Perform_Safe_Shutdown();
      }
   }
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 2 of 9]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Identify_Required_Mode()

  Try to work out the cause of the system reset.

-*-------------------------------------------------------------------*/
void SYSTEM_Identify_Required_Mode(void)
   {
   // System mode information *may* be available
   // If so, we have two copies of same data (GPREG1 is inverted)
   uint32_t Mode_copy1 = LPC_RTC->GPREG0;
   uint32_t Mode_copy2 = ~(LPC_RTC->GPREG1);

   // Store "fail silent" mode as default (for *next* mode)
   // => used in absence of any specific fault information
   LPC_RTC->GPREG0 = FAIL_SILENT;
   LPC_RTC->GPREG1 = ~(FAIL_SILENT);  // Inverted copy

   // If "1", reset was caused by WDT
   uint32_t WDT_flag = (LPC_SC->RSID >> 2) & 1;

   if (WDT_flag != 1)
      {
      // A “normal” system start (not WDT reset)
      // Here we treat all such resets in the same way
      // Set system NORMAL_01
      System_mode_G = NORMAL_01;
      }
   else
      {
      // If we are here, reset was caused by WDT
      // Need to clear the WDT flag
      // => or future resets may be interpreted as WDT
      LPC_SC->RSID &= ~(0x04);

      // Do sanity checks on available mode data
      if ((Mode_copy1 == Mode_copy2) &&
         ((Mode_copy1 == FAIL_SILENT) ||
          (Mode_copy1 == NORMAL_01) ||
          (Mode_copy1 == NORMAL_02) ||
          (Mode_copy1 == NORMAL_03)))
         {
         // Got valid data - set required mode
         System_mode_G = Mode_copy1;
         }
      else
         {
         // Problem with the data?
         // => set the system mode to "Fail Silent"
         System_mode_G = FAIL_SILENT;
         }
      }
   }
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 3 of 9]

```
/*------------------------------------------------------------------*-

  SYSTEM_Configure_Required_Mode()

  Configure the system in the required mode.

-*------------------------------------------------------------------*/
void SYSTEM_Configure_Required_Mode(void)
   {
   // In this design, we use the same tick interval in all normal modes:
   // different intervals can be used, if required.
   SCH_Init(1);

   switch (System_mode_G)
      {
      default:
      case FAIL_SILENT:
         {
         // Fail as silently as possible
         SYSTEM_Perform_Safe_Shutdown();
         break;
         }

      case NORMAL_01:
        {
        // We use different WDT setting in each normal mode
        // (for demo purposes here - can be required in some designs)
        WATCHDOG_Init(1100);

        // Prepare for Heartbeat1 task
        HEARTBEAT1_Init();

        // Flash different LED in different modes
        SCH_Add_Task(HEARTBEAT1_Update, 0, 1000, 100, 0);
        break;
        }

      case NORMAL_02:
        {
        // We use different WDT setting in each normal mode
        WATCHDOG_Init(1120);

        // Prepare for Heartbeat2 task
        HEARTBEAT2_Init();

        SCH_Add_Task(HEARTBEAT2_Update, 0, 1000, 100, 0);
        break;
        }
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 4 of 9]

```
    case NORMAL_03:
      {
      // We use different WDT setting in each normal mode
      WATCHDOG_Init(1150);

      // Prepare for Heartbeat3 task
      HEARTBEAT3_Init();

      SCH_Add_Task(HEARTBEAT3_Update, 0, 1000, 100, 0);
      break;
      }
    }

// Add the tasks below in *ALL* modes in this demo

// Add watchdog task
SCH_Add_Task(WATCHDOG_Update, 0, 1, 10, 0);

// Report current system mode on 7-seg LED display
SPI_7_SEG_Init();
SCH_Add_Task(SPI_7_SEG_Update, 10, 1000, 10, 0);

// Poll SW3 on EA baseboard
SWITCH_SW3_Init();
SCH_Add_Task(SWITCH_SW3_Update, 5, 10, 20, 0);

// We change mode every 10 seconds or when SW3 is pressed
SCH_Add_Task(SYSTEM_Update_Normal, 0, 1000, 100, 50);

WATCHDOG_Update();
}
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 5 of 9]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Update_Normal()

  Periodic task that controls system mode transitions.

  This version is employed in all "normal" modes.

  Call every second in this is demo.

-*-------------------------------------------------------------------*/
void SYSTEM_Update_Normal(void)
   {
   static uint32_t Seconds = 0;
   uint32_t Change_mode = 0;
   eSystem_mode New_mode;


   // Read the switch state
   if (Sw_pressed_G == 1)
      {
      // Switch pressed - change mode immediately
      Change_mode = 1;
      }

   // ... otherwise change mode every 10 seconds
   if (Seconds++ == 10)
      {
      Seconds = 0;
      Change_mode = 1;
      }

   // Handle mode changes
   if (Change_mode == 1)
      {
      switch (System_mode_G)
         {
         default:
         case FAIL_SILENT:
            {
            // Should not reach here ...
            New_mode = FAIL_SILENT;
            break;
            }

         case NORMAL_01:
            {
            New_mode = NORMAL_02;
            break;
            }
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 6 of 9]

```
        case NORMAL_02:
           {
           New_mode = NORMAL_03;
           break;
           }

        case NORMAL_03:
           {
           New_mode = NORMAL_01;
           break;
           }
        }

    // Change mode
    SYSTEM_Change_Mode(New_mode);
    }
}
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 7 of 9]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Change_Mode()

  Force change in the system mode (change to specific mode).

-*-------------------------------------------------------------------*/
void SYSTEM_Change_Mode(const eSystem_mode NEW_MODE)
   {
   // Avoid any potential problems while we store data
   WATCHDOG_Update();

   // Store required new system mode in NVRAM
   LPC_RTC->GPREG0 = (uint32_t) NEW_MODE;
   LPC_RTC->GPREG1 = ~((uint32_t) NEW_MODE);  // Inverted copy

   // In this design, we are using the WDT to force a reset
   // => we check (again) that the WDT is running ...
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      // Watchdog *not* running
      // => we try to shut down as safely as possible
      SYSTEM_Perform_Safe_Shutdown();
      }

   if (NEW_MODE == FAIL_SILENT)
      {
      // We are trying to shut down because of a fault
      // If the problems are serious, we may not manage to reset
      // => attempt to perform safe shutdown in current mode
      // Note: system reset should still take place
      // => this is a "safety net"
      SYSTEM_Perform_Safe_Shutdown();
      }

   // Force system reset (if WDT enabled)
   while(1);
   }
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 8 of 9]

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

```
/*------------------------------------------------------------------*-

  SYSTEM_Perform_Safe_Shutdown()

  Attempt to place the system into a safe state.

  Note: Does not return and may (if watchdog is operational) result
  in a processor reset, after which the function may be called again.

  [The rationale for this behaviour is that - after the reset -
   the system MAY be in a better position to enter a safe state.
   To avoid the possible reset, adapt the code and feed the WDT
   in the loop.]

-*------------------------------------------------------------------*/
void SYSTEM_Perform_Safe_Shutdown(void)
   {
   // Used for simple fault reporting
   uint32_t Delay, j;

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

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

   HEARTBEAT1_Init();

   while(1)
      {
      // Flicker Heartbeat LED to indicate fault
      for (Delay = 0; Delay < 200000; Delay++) j *= 3;
      HEARTBEAT1_Update();
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 21: TTRD07a (system_mode_change_1769_001-2_c07a) [Part 9 of 9]

7.15. Code listings (TTRD07c)

```
/*-------------------------------------------------------------------*-

  system_mode_change_1769_001-2_c0c.c (Released 2015-01)

  Controls system configuration after processor reset.

  Demonstrates transition between modes (with mode-change reset).

  Demonstrates basic use of a "limp home" mode.

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

// Project header
#include "../main/main.h"

// Task headers
#include "../tasks/heartbeat_basic_x4_1769_001-2_c07c.h"
#include "../tasks/watchdog_1769_001-2_c02a.h"
#include "../tasks/switch_ea-sw3_1769_001-2_c07a.h"
#include "../tasks/spi_7_seg_display_mode_1769_001-2_c07a.h"

// ------ Public variable definitions ------------------------------

// In many designs, System_mode_G will be used in other modules.
// - we therefore make this variable public.
eSystem_mode System_mode_G;

// ------ Public variable declarations ------------------------------

extern uint32_t Sw_pressed_G; // The current switch status (SW3)

extern uint32_t Fault_code_G; // System fault codes

// ------ Private function declarations ----------------------------

void SYSTEM_Identify_Required_Mode(void);
void SYSTEM_Configure_Required_Mode(void);
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 1 of 11]

```
/*------------------------------------------------------------------*-

  SYSTEM_Init()

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

void SYSTEM_Init(void)
   {
   SYSTEM_Identify_Required_Mode();
   SYSTEM_Configure_Required_Mode();

   // This design relies on the WDT to change modes
   // => we double check that the WDT is running ...
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      // Watchdog *not* running
      // => we try to shut down as safely as possible
      SYSTEM_Perform_Safe_Shutdown();
      }
   }
```

```
/*-------------------------------------------------------------------*-

  SYSTEM_Identify_Required_Mode()
  Try to work out the cause of the system reset.

-*-------------------------------------------------------------------*/
void SYSTEM_Identify_Required_Mode(void)
   {
   // System mode information *may* be available
   // If so, we have two copies of same data (GPREG1 is inverted)
   uint32_t Mode_copy1 = LPC_RTC->GPREG0;
   uint32_t Mode_copy2 = ~(LPC_RTC->GPREG1);

   // Store "fail silent" mode as default (for *next* mode)
   // => used in absence of any specific fault information
   LPC_RTC->GPREG0 = FAIL_SILENT;
   LPC_RTC->GPREG1 = ~(FAIL_SILENT);  // Inverted copy

   // If "1", reset was caused by WDT
   uint32_t WDT_flag = (LPC_SC->RSID >> 2) & 1;

   if (WDT_flag != 1)
      {
      // A “normal” system start (not WDT reset)
      // Here we treat all such resets in the same way
      // Set system NORMAL_01
      System_mode_G = NORMAL_01;
      }
   else
      {
      // If we are here, reset was caused by WDT
      // Need to clear the WDT flag
      // => or future resets may be interpreted as WDT
      LPC_SC->RSID &= ~(0x04);

      // Do sanity checks on available mode data
      if ((Mode_copy1 == Mode_copy2) &&
         ((Mode_copy1 == FAIL_SILENT) ||
          (Mode_copy1 == LIMP_HOME) ||
          (Mode_copy1 == NORMAL_01) ||
          (Mode_copy1 == NORMAL_02) ||
          (Mode_copy1 == NORMAL_03)))
         {
         // Got valid data - set required mode
         System_mode_G = Mode_copy1;
         }
      else
         {
         // Problem with the data?
         // => set the system mode to "Fail Silent"
         System_mode_G = FAIL_SILENT;
         }
      }
   }
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 2 of 11]

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

```
/*------------------------------------------------------------------*-

  SYSTEM_Configure_Required_Mode()
  Configure the system in the required mode.

-*------------------------------------------------------------------*/
void SYSTEM_Configure_Required_Mode(void)
   {
   // In this design, we use the same tick interval in all normal modes:
   // different intervals can be used, if required.
   SCH_Init(1);

   switch (System_mode_G)
      {
      default:
      case FAIL_SILENT:
         {
         // Fail as silently as possible
         SYSTEM_Perform_Safe_Shutdown();
         break;
         }

     case LIMP_HOME:
        {
        // This is the "limp home" mode
        // - Uses LED on LPCXpresso board (not EA Baseboard)

        // We use different WDT setting in each normal mode
        // (for demo purposes here - can be required in some designs)
        WATCHDOG_Init(4900);

        // Prepare for Heartbeat1 task
        HEARTBEAT_Init();

        // Flash different LED in different modes
        SCH_Add_Task(HEARTBEAT_Update, 0, 1000, 100, 0);

        // Add task to monitor for system overloads
        SCH_Add_Task(SYSTEM_Update_Limp_Home, 0, 1000, 100, 50);

        break;
        }

      case NORMAL_01:
        {
        // We use different WDT setting in each normal mode
        // (for demo purposes here - can be required in some designs)
        WATCHDOG_Init(5000);

        // Prepare for Heartbeat1 task
        HEARTBEAT1_Init();

        // Flash different LED in different modes
        SCH_Add_Task(HEARTBEAT1_Update, 0, 1000, 100, 0);
        break;
        }
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 3 of 11]

```
   case NORMAL_02:
     {
     // We use different WDT setting in each normal mode
     WATCHDOG_Init(5020);

     // Prepare for Heartbeat2 task
     HEARTBEAT2_Init();

     SCH_Add_Task(HEARTBEAT2_Update, 0, 1000, 100, 0);
     break;
     }

   case NORMAL_03:
     {
     // We use different WDT setting in each normal mode
     WATCHDOG_Init(5200);

     // Prepare for Heartbeat3 task
     HEARTBEAT3_Init();

     SCH_Add_Task(HEARTBEAT3_Update, 0, 1000, 100, 0);
     break;
     }
   }

// Add the tasks below in *ALL* modes in this demo

// Add watchdog task
SCH_Add_Task(WATCHDOG_Update, 0, 1, 10, 0);

// Report current system mode on 7-seg LED display
SPI_7_SEG_Init();
SCH_Add_Task(SPI_7_SEG_Update, 10, 1000, 10, 0);

// Add the tasks below only in "normal" modes
if ((System_mode_G == NORMAL_01) ||
    (System_mode_G == NORMAL_02) ||
    (System_mode_G == NORMAL_03))
   {
   // Poll SW3 on EA baseboard
   SWITCH_SW3_Init();
   SCH_Add_Task(SWITCH_SW3_Update, 5, 10, 20, 0);

   // In "normal" modes, we change mode every 10 seconds
   // or when SW3 is pressed
   SCH_Add_Task(SYSTEM_Update_Normal, 0, 1000, 100, 50);
   }

WATCHDOG_Update();
}
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 4 of 11]

```
/*------------------------------------------------------------------*-

  SYSTEM_Update_Normal()

  Periodic task that controls system mode transitions.

  This version is employed in all "normal" modes.

  Call every second in this is demo.

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

void SYSTEM_Update_Normal(void)
   {
   static uint32_t Seconds = 0;
   uint32_t Change_mode = 0;
   eSystem_mode New_mode;

   // Check for system overload
   if (Fault_code_G == FAULT_SCH_SYSTEM_OVERLOAD)
      {
      // In a "normal" mode - move to "limp home"
      SYSTEM_Change_Mode(LIMP_HOME);
      }

   // If there are no overloads, then ...

   // Read the switch state
   if (Sw_pressed_G == 1)
      {
      // Switch pressed
      Change_mode = 1;
      }

   // ... otherwise change mode every 10 seconds
   if (Seconds++ == 10)
      {
      Seconds = 0;
      Change_mode = 1;
      }

   // Handle mode changes
   if (Change_mode == 1)
      {
      switch (System_mode_G)
         {
         default:
         case FAIL_SILENT:
            {
            // Should not reach here ...
            New_mode = FAIL_SILENT;
            break;
            }
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 5 of 11]

```
        case NORMAL_01:
           {
           New_mode = NORMAL_02;
           break;
           }

        case NORMAL_02:
           {
           New_mode = NORMAL_03;
           break;
           }

        case NORMAL_03:
           {
           New_mode = NORMAL_01;
           break;
           }
        }

    // Change mode
    SYSTEM_Change_Mode(New_mode);
    }
}
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 6 of 11]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Update_Limp_Home()

  Periodic task that controls system mode transitions.

  This version is employed in "limp home" mode.

  Call every second in this is demo.

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

void SYSTEM_Update_Limp_Home(void)
   {
   // Checks for (another) system overload
   if (Fault_code_G == FAULT_SCH_SYSTEM_OVERLOAD)
      {
      // Another overrun ...
      SYSTEM_Change_Mode(FAIL_SILENT);
      }

   // In the absence of overloads, we simply remain in "limp home" mode
   // - other behaviour can (of course) be implemented
   }
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 7 of 11]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Change_Mode_Fault()

  Force change in the system mode in the event of a fault

  Note that - in this simple demo - we cannot do very much with
  the fault data in this function.

  In a realistic design, a version of this function is used to
  ensure that some that - for example - some faults are allowed
  to occur up to N times per hour, etc.  That is, we control the
  transition to "fail silent".

  We may also wish to use this function as the basis of a
  "black box" data recorder (and keep a record of any faults).

-*-------------------------------------------------------------------*/
void SYSTEM_Change_Mode_Fault(const uint32_t FAULT)
   {
   // Deal unidentified faults
   if (FAULT == SYS_UNIDENTIFIED_FAULT)
      {
      SYSTEM_Change_Mode(FAIL_SILENT);
      }

   // If we reach here, we are dealing with identified faults:
   // these take us first to "limp home" mode
   if ((System_mode_G == NORMAL_01) ||
       (System_mode_G == NORMAL_02) ||
       (System_mode_G == NORMAL_03))
      {
      // In a normal mode
      // => change to "limp home"
      SYSTEM_Change_Mode(LIMP_HOME);
      }
   else
      {
      // Already in a limp-home mode
      // => change to "fail silent"
      SYSTEM_Change_Mode(FAIL_SILENT);
      }
   }
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 8 of 11]

```
/*------------------------------------------------------------------*-

  SYSTEM_Change_Mode()

  Force change in the system mode (change to specific mode).

-*------------------------------------------------------------------*/
void SYSTEM_Change_Mode(const eSystem_mode NEW_MODE)
   {
   // Avoid any potential problems while we store data
   WATCHDOG_Update();

   // Store required new system mode in NVRAM
   LPC_RTC->GPREG0 = (uint32_t) NEW_MODE;
   LPC_RTC->GPREG1 = ~((uint32_t) NEW_MODE);  // Inverted copy

   // In this design, we are using the WDT to force a reset
   // => we check (again) that the WDT is running ...
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      // Watchdog *not* running
      // => we try to shut down as safely as possible
      SYSTEM_Perform_Safe_Shutdown();
      }

   if (NEW_MODE == FAIL_SILENT)
      {
      // We are trying to shut down because of a fault
      // If the problems are serious, we may not manage to reset
      // => attempt to perform safe shutdown in current mode
      // Note: system reset should still take place
      // => this is a "safety net"
      SYSTEM_Perform_Safe_Shutdown();
      }

   // Force system reset (if WDT enabled)
   while(1);
   }
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 9 of 11]

```
/*------------------------------------------------------------------*-

  SYSTEM_Perform_Safe_Shutdown()

  Attempt to place the system into a safe state.

  Note: Does not return and may (if watchdog is operational) result
  in a processor reset, after which the function may be called again.

  [The rationale for this behaviour is that - after the reset -
   the system MAY be in a better position to enter a safe state.
   To avoid the possible reset, adapt the code and feed the WDT
   in the loop.]

-*------------------------------------------------------------------*/
void SYSTEM_Perform_Safe_Shutdown(void)
   {
   // Used for simple fault reporting
   uint32_t Delay, j;

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

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

   HEARTBEAT1_Init();

   while(1)
      {
      // Flicker Heartbeat LED to indicate fault
      for (Delay = 0; Delay < 200000; Delay++) j *= 3;
      HEARTBEAT1_Update();
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 22: TTRD07c (system_mode_change_1769_001-2_c07c.c) [Part 10 of 11]

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

CHAPTER 8: Task Contracts (Resource Barriers)

In this chapter, we introduce "Task Contracts" and explain how this approach can be used to guide the development of effective tasks for reliable, real-time embedded systems.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD08a: TTC-TC MPU scheduler

8.1. Introduction

As we have stated in previous chapters, our overall goal in this book is to explore techniques which will help us to prevent Uncontrolled System Failures.

As one step towards this goal, we argued in Chapter 6 that code for reading an ADC, such as the following:

```
while ((LPC_ADC->ADGDR & 0x80000000UL) == 0);  // Bit set on completion
```

… should always be accompanied by a "timeout" mechanism, just in case the ADC had not been configured properly, or some part of the analogue input system had suffered damage. Our argument in Chapter 6 was that identifying the cause of a fault at the point of ADC use should give us a greater chance of handling it appropriately than would be possible if (for example) we responded to some of the consequences of such a fault (such as a task overrun, or a system overload).

In the context of our overall goal, use of timeout mechanisms in an ADC task is appropriate, but it is not a complete solution. In particular, as we have identified "incorrect configuration of the ADC" as one potential hazard, it would – surely – make more sense to check that the ADC is configured correctly before we try to use it?

If we check the ADC every time we use it, we then have the potential to address issues caused by both incorrect configuration of this component and by corruption of the configuration settings (such as might be caused if another task reconfigured the ADC during the program execution).

In this chapter, we will explore such issues by building on the concept of "Contracts", first introduced by Bertrand Meyer.

We begin by providing a brief summary of Meyer's work.

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

8.2. Origins of "Contracts" in software development

The concept of "contract-based design" (CBD) was introduced by Meyer in connection with his design of the Eiffel programming language (see, for example, Meyer, 1997).

Meyer views the software system as a set of components, and the specifications that define the interactions between these components as a set of mutual obligations (the "contracts"). Central to his CBD method is the idea that we should embed these contracts in the system code and check for compliance at run time.

This general approach is now widely seen as a means of reducing the number of residual errors in complex software systems.[17]

8.3. What do we mean by a "Task Contract"?

In the remainder of this chapter, we describe a form of CBD – based on what we call Task Contracts – that the author has found to be an effective way of guiding the development of (and review of) effective tasks for TT embedded systems.

As far as this book is concerned, development of a Task Contract (TC) should involve the following processes:

- Drawing up a specification for the task.
- Considering any pre-conditions (PreCs) that must apply before the task will be able to meet this specification, and – where sensible and appropriate – incorporating mechanisms in the task code to check for these PreCs.
- Considering any ways in which the task's environment might change during its operation (after the PreCs have been checked, and in such a way that would be likely to prevent the task from meeting its specification) and – where sensible and appropriate – incorporating mechanisms in the task code to check for any such changes.
- Considering any ways in which it will be possible to confirm that the task has met its required specification when it completes and – where sensible and appropriate – incorporating mechanisms in the task code to check for such a successful completion: these are the Post Conditions (PostCs).

The overall goal with TCs is to support the process of designing, implementing and reviewing the task set.

[17] We provide support for this claim on Page 128.

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

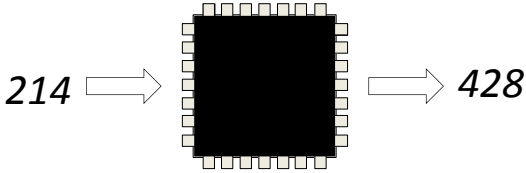


Figure 66: A schematic representation of a task that is required to perform a simple arithmetic calculation (in this case, doubling a data value).

8.4. Numerical example

As a first simple example, suppose that we have a task that is required to double the value of an unsigned integer that it is passed (in some form) as "input data".

Our first goal is to provide a guarantee that the task will complete this operation successfully, provided that certain conditions are met.

In this case (on the assumption that the output data will be represented as an unsigned integer), we have the pre-condition that the input data is in the range 0 – (UINT_MAX / 2), where UINT_MAX is the maximum value that can be represented as an unsigned integer on a particular platform.

Our "pre-condition" for the effective use of our task is – therefore – that the data are in range (otherwise the task simply cannot do what is required). We therefore check this condition before we do anything further in the task. If the condition has not been met, we "abort the task".

If the pre-condition is met, our "guarantee" (or "contract") then applies. After performing the required operation, we may then double check that we have doubled the number successfully – as a "post condition" – before the task completes.

8.5. Control example

As a more representative "embedded" example, suppose that we have a task that is intended to move an actuator to a particular setting (Figure 67).

In this case, the input might be the required setting (in degrees), in the range 0° to 90°.

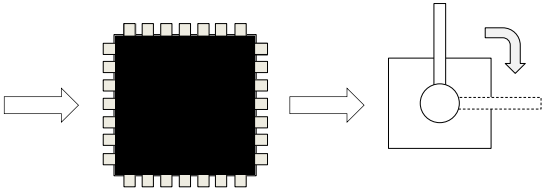


Figure 67: Control of an actuator.

We can – clearly – check that the data passed to the task is in the correct range (an obvious pre-condition).

Note that simply checking that the required input is in range may not be adequate: we may also need to check for rate of change. For example, if the angle of the actuator may be changed at a rate of no more than 2° per minute, and we have a request for a move of 10°, we may need to reject this request (even though it is in range).

In addition, we may – in some safety-related systems – need to use multiple (independent) sources of information – such as three or more different sensors – to confirm the validity of input data.

During the task operation, we may monitor some settings (for example, current drawn by the motor). This may allow us to determine if – for example – if the motor is turning when expected, or whether there is a short circuit in the motor-drive electronics. These represent on-going checks that may be performed during the system operation.

Once we think that the task has completed, we need to know if the task has operated as required. We therefore – again – want to monitor the outputs from the microcontroller and / or use sensors on the actuator to confirm that the actuator has actually moved as required. In this case, for example, there may be "limit switches" that are activated when the actuator is in "right" position (shown with dotted lines in Figure 67).

8.6. Timing is part of the Task Contract

The concept of CBD comes from desktop programming, and is concerned with data values rather than with timing behaviour.

In real-time systems, it can also be useful to think of WCET and BCET as part of the Task Contract in a real-time system.

As an example, we'll revisit the actuator example (from Section 8.5).

As part of this Task Contract, we would also include timing behaviour. To do this, we might say that – provided that the pre-conditions are met – we guarantee that our task will produce the correct output within – say 350 ms (minimum) and 500 ms (maximum).

To summarise the end result.

Our Task Contract will state that the task will move the actuator to a particular position in a period of 350-500 ms, provided that the requested angle [a] is in the range 0° to 90°; [b] does not change by more than 2° per minute, and [c] has been confirmed by a majority decision from at least 2 out of 3 available sensors. Point a to Point c can be checked in the task pre-conditions (timing issues will checked while the task runs).

Assuming that we have met the pre-conditions, we will then check the motor current during the actuator movement, and keep track of the elapsed time.

Once we think that we have moved the actuator, we will confirm that the actuator is now in the required position (probably using external sensors) before exiting from the task.

8.7. Implementing Task Contracts (overview)

The traditional way of implementing some of the key parts of the CBD approach in "desktop" C programs is through use of the assert() macro. This approach is standard, and has the benefit of simplicity, but it is rarely an appropriate solution for developers of real-time embedded systems. This is because the default behaviour of assert() – when the integer expression passed to the macro evaluates to 0 – is to print a message on the display and terminate the program. Since we may well have neither a display nor an operating system to "exit to", we generally need to make some changes to this approach if we are to make use of it in an embedded design.

We may be able to edit the assert() macro, but that isn't really the point. We are looking for an effective means of performing checks on scheduled tasks (and exiting these tasks if the checks fail). Unlike the use of assert(), we want to make this part of our final system code: that is, these checks won't just be used during debugging and then switched off, they will be "shipped", as part of our final system code.

To achieve this, we will use the task return values in a modified scheduler to report any Software Faults or Hardware Faults that were encountered during the task execution.

If any such a fault is detected, the system may enter a Limp-Home or Fail-Silent Mode. Alternatively, it may be possible to replace the task with a "backup" that is able to operate effectively and carry on.

We give a suitable scheduler framework in Listing 24.

8.8. Implementing Task Contracts (timing checks)

We considered some techniques for designing effective tasks in Chapter 6. These included the use of "hardware timeouts" and related solutions that form an important way of meeting task timing requirements.

Beyond such checks, we find it more effective to allocate responsibility for the overall task timing (in terms of WCET and BCET limits) to the scheduler (which will, in turn, employ a MoniTTor unit to do this work).

We will discuss ways of achieving this in Chapter 9.

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

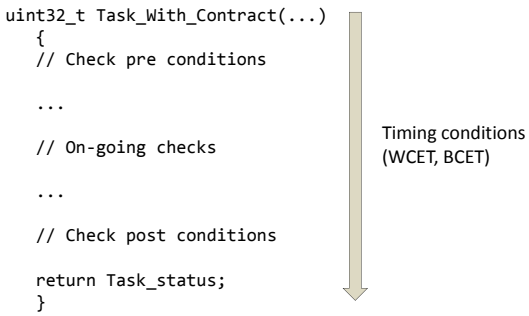


Figure 68: An example of a first implementation of a "task with contract".

8.9. Implementing Task Contracts (checking peripherals)

Please consider again the simple Heartbeat task that we have employed in examples since Chapter 2 (part of which is reproduced in Code Fragment 39 for ease of reference).

Suppose that the HEARTBEAT_Init() function has not been called before the HEARTBEAT_Update() task is added to the array. In these circumstances, the task will not detect this problem: it will run as scheduled, but the LED won't flash.

Alternatively, suppose that another task in the system has been developed and that it reconfigures the port pin that is used by the Heartbeat task. Again, the task will still run and we may not see the LED flashing.

Neither situation may seem to be particularly serious, but they are symptomatic of a potential problem that is much more significant. For example, suppose that an "emergency stop" mechanism is disabled because port pins have been reconfigured by a different task: if we only detect this situation when the emergency stop operation is required, then this may have fatal consequences.

In general, an important pre-condition for any tasks that involve outputs is to check – every time the task is executed – that the outputs have been configured correctly.

```
void HEARTBEAT_Update(void)
   {
   …
   if (LED_state == 1)
      {
      LED_state = 0;
      GPIO_ClearValue(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);
      }
   …
   }
```

Code Fragment 39: Part of the HEARTBEAT_Update() function from Chapter 2.

Similarly, any task involving inputs must also be checked (for similar reasons).

One example of a modified version of the HEARTBEAT_Update() function that meets this requirement is given in Code Fragment 40.

```
uint32_t HEARTBEAT_Update(void)
   {
   static uint32_t LED_state = 0;
   static uint32_t Fault_state = 0;
   uint32_t Precond;

   // Heartbeat pin must be set for O/P
   Precond = GPIO_Check_Output(HEARTBEAT_LED_PORT, HEARTBEAT_LED_PIN);

   if (Precond == RETURN_FAULT)
      {
      return RETURN_FAULT;
      }
   …
```

Code Fragment 40: Part of the HEARTBEAT_Update() function from TTRD08a.

This is not the only option here. For example, rather than checking the state of the port pin, we could set the pin to the required state every time we release the task. This solution may solve the immediate problem. It may even be easier / faster than checking the pin state. However, it may also mask a serious underlying problem with the system: for this reason, we recommend checking the I/O status as a pre-condition in almost all cases.

Note that it is difficult to find a (practical) post condition check for the Heartbeat task. We could add some form of light sensor (Figure 69) to check that the LED is actually in the correct state but – in this example – this would be excessive. In a more realistic example (for example, automatic car headlights, car indicator / turn lights, traffic-light control) such a sensor may well be appropriate.

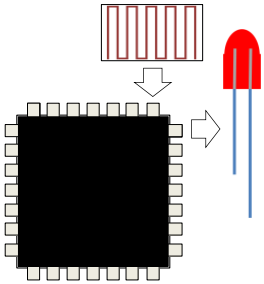


Figure 69: Adding a sensor (e.g. a light-dependent resistor) to check that the LED is operating as required (see text for details).

8.10. Example: Feeding the WDT

The WDT is the final line of defence in many systems. Clearly, this line of defence will only be effective if the WDT has been configured correctly.

When we feed the WDT, we need to check that this is – in fact – the case. Code Fragment 41 shows one way in which we can achieve this.

```
uint32_t WATCHDOG_Update(void)
   {
   // Precondition: is the WDT running?
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      return RETURN_FAULT;
      }

   // Feed the watchdog
   __disable_irq();
   LPC_WDT->WDFEED = 0xAA;
   LPC_WDT->WDFEED = 0x55;
   __enable_irq();

   return RETURN_NORMAL;
   }
```

Code Fragment 41: A modified WATCHDOG_Update task for the LPC1769.

8.11. One task per peripheral

In the examples above, we have argued that we should consider checking the state of peripherals every time we use them.

At a task level, this makes sense. At a system level, we can (and often should) go further. Consider, for example, the design illustrated in Figure 70. In this design we have three tasks (Task A, Task B, Task E) all sharing the same ADC (with each task using a different analogue channel).

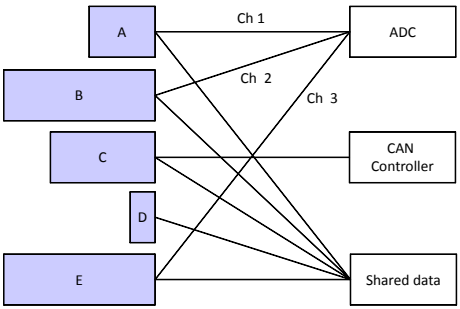


Figure 70: A design in which three tasks share access to a resource (ADC).

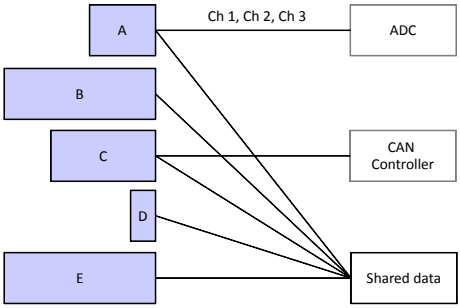


Figure 71: Implementing a "one resource per task" architecture.

In a TTC design (our focus in this part of the book), the risk of direct conflicts over this shared resource is low, because each task will run to completion. However, we have three tasks interacting with the ADC and, therefore, more opportunities for coding errors. In addition, we have the overhead involved in checking the ADC configuration for each of the (three) tasks. Finally, the code may become more complicated to work with if we extend the design to support task pre-emption (as we will do in Part Three).

For all of these reasons, we suggest that you consider allocating "one task per resource" in such designs, and then distributing the data from the ADC to the tasks that require it. For example, in the design shown in Figure 71, we could have Task A read from all three of the ADC channels and pass the information to the other tasks via the "shared data" area in memory.

Note that a side effect of this model is that we are likely to increase the number of data transfers that are required in the system.

8.12. What about shared data?

Up until now, we have carried out data transfers between tasks using global variables. Because we have been dealing with TTC designs, we have no need to protect the data with a "mutex" or similar mechanism.

However, this arrangement gives us no protection against data corruption that may be caused: [i] by a fault in any of the tasks that have access to the shared data area, or [ii] by some form of external interference.

For example, please consider Figure 72.

In Figure 72, neither Task B nor Task E is supposed to have access to Variable X (a global variable): however, as a result of a programming error (for example) it would be possible for one of these tasks to alter the value of this variable. In addition, it may be possible for an "Unknown Source" to alter the value of Variable X (possibly as a result of EMI or a virus, for example).

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

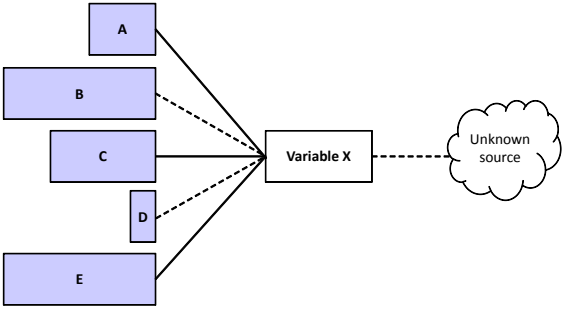


Figure 72: Potential causes of data corruption. See text for details.

As we have discussed previously, this type of "interference" between tasks that share memory is generally very difficult to avoid. However, we can detect it, if we develop our application in an appropriate way.

8.13. Implementing Task Contracts (protecting data transfers)

To reduce the potential for variable corruption, we use another form of Resource Barrier to protect shared data.

These operate as follows:

- We keep shared data "private" and provide access only by means of "Read" and "Write" functions
- We allow only one task to have write access to any shared variable.
- We keep a record of the task that has write access and check the task ID before any write actions are allowed to take place.
- The task with write access keeps a second (local) copy of the data for which it is responsible: the two copies are compared (by the write task) [i] every time it is called; and [ii] every time it writes new data.
- Whenever we use duplicated variables in this way, we run the risk that some faults – for example, setting all memory contents to 0 – will go undetected (because the value of both variables will remain the same after this fault occurs). We can increase our chances of detecting such faults by "inverting" one of the copies when we store it: for example, rather than storing 0b1010 and 0b1010, we store 0b1010 and 0b0101.

As examples, please consider the library code for task data checks (Listing 27 and Listing 28). This is applied in Listing 30 (which resets a counter when a switch is pressed) and Listing 31 (which processes the associated switch input).

8.14. An alternative way of detecting corruption in shared data

There are - of course - different ways in which we can detect corruption in shared data. For example, we can use a "lightweight" form of Resource Barrier (LRB).

What we refer to in this text as LRBs involves using two global variables to store each data value that needs to be transferred between tasks.

More specifically, LRBs for common "unsigned integer" variables involve storing both the data value and an "inverted copy" of each data value, as we did when storing information about the system mode in NVRAM in Chapter 7.

We will illustrate the use of LRBs in TTRD15a.

8.15. How can we detect corruption of the scheduler data?

So far in this chapter, we have considered techniques for detecting corruption of the task data. We also – of course – need to consider the possibility that one of the tasks (or some form of external interference) will change the scheduler data.

Note that we could just let our monitoring system handle this problem, but – as in previous cases considered in this chapter – we would prefer to be able to detect the problem as close to the point of failure as possible, in order to maximize our opportunities for handling the situation correctly.

One effective solution can be to duplicate the scheduler data in memory (and "invert" one of the copies). We can then treat any situation in which the data do not match as a resource-related fault.[18]

We can also consider triplicating the copies and continue to operate the system (possibly in a "Limp Home" mode) if 2 out of 3 copies are found to be the same.

These approaches have the benefit that they can be applied with virtually any microcontroller hardware (subject, of course, to the availability of enough memory).

8.16. Making use of the MPU

As an alternative to duplicating data in memory, we can make use of "memory protection" hardware in the microcontroller.

Such hardware typically allows the programmer to indicate – at specific times during the program execution – that particular memory locations must not be accessed.

---

[18] We will illustrate this approach in TTRD15a.

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

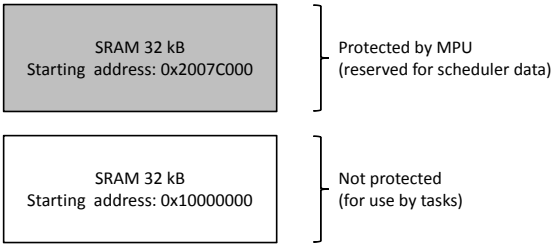


Figure 73: Two areas of SRAM in the LPC1769.

If an attempt is made to access these "protected" locations, the access is prevented and a fault is signaled (usually by triggering an interrupt).

What we can do as a starting point in a system running a TT scheduler is ensure that – before each task is released – the area of memory in which the scheduler data is stored is protected. Once the task completes, the protection is removed, allowing the scheduler to operate normally.

The Cortex-M3 platform provides MPU support, and this support is implemented in the LPC1769 microcontroller. We make use of this protection in TTRD08a (see Listing 25).

To understand the operation of this TTRD, you first need to understand that there are two distinct areas of SRAM in the LPC1769, each 32 kB is size (Figure 73). In TTRD08a we reserve the upper area of memory for use by the scheduler, and use the lower area of memory for the tasks.

Note that other divisions can – of course – be made.

Please refer to Listing 25 for full details.

8.17. Supporting Backup Tasks

In Section 8.7, we suggested that – in circumstances where a problem is detected with a task – it may be possible to replace the code that has detected problems with a backup task and carry on.

TTRD08a supports such an arrangement. However, we will delay a discussion about the development of suitable backup tasks until Chapter 13 (as we view such activities as part of a larger system-level design exercise).

8.18. What do we do if our Resource Barrier detects a fault?

Let's suppose that we release Task A from the scheduler detailed in Listing 24. Let's further suppose that – through the use of the Resource Barriers implemented in this task – we detect a problem: specifically, we'll assume that the two copies of one piece of data employed by Task A are found to be different at run time.

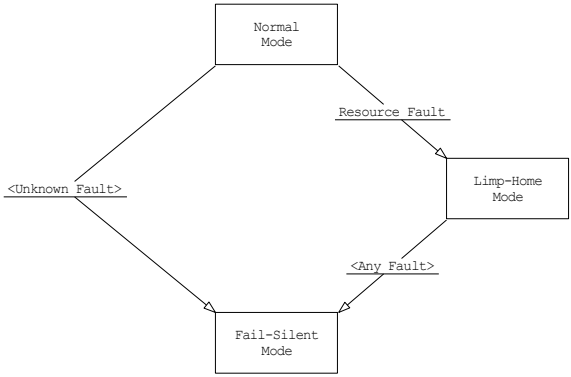


Figure 74: One possible way of handling resource-related faults. Please see text for details and consideration of other alternatives.

One option may be to replace Task A with a backup task. This may be a worthwhile activity for two reasons: [i] it may solve the problem; [ii] if it doesn't solve the problem, it may allow us to narrow down the list of potential issues, since failure of a properly-designed backup task may suggest that there are larger problems with the system.

Overall, it is important to be clear about the limits of our abilities here. If a stable, fully verified system starts detecting resource faults, then it may be very difficult to determine the cause while system is in the field. The problems may, for example, arise as a result of errors in the design or implementation of Task A. Alternatively, the problems may arise with another (apparently unconnected task), or from some form of electrical interference.

In many cases, the best that we can do may be shut the system down, after storing as much information as possible about the fault, to allow for off-line investigation. Where shutting down is not possible, we can try a backup task, or switch to a Limp-Home Mode (Figure 74).

Overall, we must emphasise again that our goal in this text is <u>not</u> to pretend that we can eliminate all faults from any system: instead, our goal is to consider ways in which we can design and implement the system in order to maximise our chances of detecting (and then handling) such faults, in order to minimise the risk of Uncontrolled System Failures.

8.19. Task Contracts and international standards

The use of Task Contracts (as described in this chapter) is compatible with a range of international safety standards.

For example, within IEC 61508-7, the use of pre- and post-conditions in software functions / tasks is referred to as "Failure assertion programming"

(FAP), and is seen as a means of reducing the effort required to complete integration testing (IEC 61508-2).

Similarly, within ISO 26262, use of pre-conditions is referred to as "Plausibility Checks" (PCs), and use of this approach is "Highly Recommended" for ASIL D designs (ISO 26262-6, Table 4).

On top of this, IEC 61508-7 requires that variables should be range checked and (where possible) checked for plausibility. More generally (in the same part of this standard) it is argued that fault-tolerant software should be designed to "expect" failures in its own environment or use outside nominal or expected conditions, and should behave in a predefined manner in these situations: this is exactly what we aim to achieve with Task-Contract pre-conditions.[19]

In terms of post conditions, IEC 61508-7 requires that "the effect of output variables should be checked, preferably by direct observation of associated system state changes".

In terms of pre- and post-conditions, ISO 26262 requires that range checks are performed on all input and output data for systems at all ASIL levels (ISO 26262-6, Section 7.4.14).

In ISO 26262, guidelines are presented in Part 5 (Annex D) that are intended to help the developer choose appropriate safety mechanisms that will allow the detection of the failure of sensors and actuators (for example). The ISO 26262-5 guidelines for the fault-detection in sensors include m-out-of-n redundancy schemes (Section D.2.6.5) and sensor correlation approaches (Section D.2.10.2) that are regarded as providing high levels of diagnostic coverage, and are compatible with the techniques described in this book using Task Contracts and pre-conditions. The ISO 26262-5 guidelines also include techniques for the monitoring of actuators (Section D.2.11.1) that are compatible with the techniques described in this book using Task Contracts and post-conditions.

8.20. Conclusions

In this chapter, we introduced the concept of "Task Contracts" and explained how this approach can be used to guide the development of effective tasks for reliable, real-time embedded systems.

In Chapter 9, we begin to explore other ways of monitoring your system at run time.

[19] These requirements are also relevant to our discussions about System Contracts in Chapter 13.

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

8.21. Code listings (TTRD08a)

```
/*------------------------------------------------------------------*-

  system_multi-mode_1769_001-2_c08a.c (Released 2015-01)
  Controls system configuration after processor reset.
  Demonstrates transition between modes (with mode-change reset).
  Demonstrates basic use of a "limp home" mode.
  Supports Task Contracts.

-*------------------------------------------------------------------*/
// Project header
#include "../main/main.h"

// Task headers
#include "../tasks/heartbeat_basic_x4_1769_001-2_c08a.h"
#include "../tasks/watchdog_1769_001-2_c08a.h"
#include "../tasks/spi_7_seg_display_mode_1769_001-2_c08a.h"
#include "../tasks/switch_ea-sw3_1769_001-2_c08a.h"

// ------ Public variable definitions ------------------------------

// In many designs, System_mode_G will be used in other modules.
// - we therefore make this variable public.
eSystem_mode System_mode_G;

// ------ Public variable declarations -----------------------------

extern uint32_t Sw_pressed_G; // The current switch status (SW3)

extern uint32_t Fault_code_G; // System fault codes

// ------ Private function declarations ----------------------------

void SYSTEM_Identify_Required_Mode(void);
void SYSTEM_Configure_Required_Mode(void);

/*------------------------------------------------------------------*-
  SYSTEM_Init()
-*------------------------------------------------------------------*/

void SYSTEM_Init(void)
   {
   SYSTEM_Identify_Required_Mode();
   SYSTEM_Configure_Required_Mode();

   // This design relies on the WDT to change modes
   // => we double check that the WDT is running ...
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      // Watchdog *not* running
      // => we try to shut down as safely as possible
      SYSTEM_Perform_Safe_Shutdown();
      }
   }
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 1 of 10]

```
/*------------------------------------------------------------------*-

  SYSTEM_Identify_Required_Mode()
  Try to work out the cause of the system reset.

-*------------------------------------------------------------------*/
void SYSTEM_Identify_Required_Mode(void)
   {
   // System mode information *may* be available
   // If so, we have two copies of same data (GPREG1 is inverted)
   uint32_t Mode_copy1 = LPC_RTC->GPREG0;
   uint32_t Mode_copy2 = ~(LPC_RTC->GPREG1);

   // Store "fail silent" mode as default (for *next* mode)
   // => used in absence of any specific fault information
   LPC_RTC->GPREG0 = FAIL_SILENT;
   LPC_RTC->GPREG1 = ~(FAIL_SILENT);  // Inverted copy

   // If "1", reset was caused by WDT
   uint32_t WDT_flag = (LPC_SC->RSID >> 2) & 1;

   if (WDT_flag != 1)
      {
      // A "normal" system start (not WDT reset)
      // Here we treat all such resets in the same way
      // Set system NORMAL_01
      System_mode_G = NORMAL_01;
      }
   else
      {
      // If we are here, reset was caused by WDT
      // Need to clear the WDT flag
      // => or future resets may be interpreted as WDT
      LPC_SC->RSID &= ~(0x04);

      // Do sanity checks on available mode data
      if ((Mode_copy1 == Mode_copy2) &&
         ((Mode_copy1 == FAIL_SILENT) ||
          (Mode_copy1 == LIMP_HOME) ||
          (Mode_copy1 == NORMAL_01) ||
          (Mode_copy1 == NORMAL_02) ||
          (Mode_copy1 == NORMAL_03)))
         {
         // Got valid data - set required mode
         System_mode_G = Mode_copy1;
         }
      else
         {
         // Problem with the data?
         // => set the system mode to "Fail Silent"
         System_mode_G = FAIL_SILENT;
         }
      }
   }
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 2 of 10]

```
/*------------------------------------------------------------------*-

  SYSTEM_Configure_Required_Mode()
  Configure the system in the required mode.

-*------------------------------------------------------------------*/
void SYSTEM_Configure_Required_Mode(void)
   {
   uint32_t Task_ID;

   TASK_DATA_Init();  // Set up the shared data

   SCH_Init(1);       // Same tick interval in all "normal" modes

   switch (System_mode_G)
      {
      default:
      case FAIL_SILENT:
         {
         // Fail as silently as possible
         SYSTEM_Perform_Safe_Shutdown();
         break;
         }

     case LIMP_HOME:
        {
        // This is the "limp home" mode
        // - Uses LED on LPCXpresso board (not EA Baseboard)

        // We use different WDT setting in each mode
        // (for demo purposes here - can be required in some designs)
        WATCHDOG_Init(4900);

        // Prepare for Heartbeat task (with state access)
        HEARTBEAT_Init();
        Task_ID = SCH_Add_Task(HEARTBEAT_Update, 0, 1000, 100, 0);
        TASK_DATA_Allow_Access(Task_ID, HEARTBEAT_STATE);

        // Add task to monitor for system overloads
        SCH_Add_Task(SYSTEM_Update_Limp_Home, 0, 1000, 100, 50);
        break;
        }

      case NORMAL_01:
        {
        // We use different WDT setting in each normal mode
        // (for demo purposes here - can be required in some designs)
        WATCHDOG_Init(5000);

        // Prepare for Heartbeat1 task (with state access)
        HEARTBEAT1_Init();
        Task_ID = SCH_Add_Task(HEARTBEAT1_Update, 0, 1000, 100, 0);
        TASK_DATA_Allow_Access(Task_ID, HEARTBEAT1_STATE);
        break;
        }
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 3 of 10]

```
   case NORMAL_02:
     {
     // We use different WDT setting in each normal mode
     WATCHDOG_Init(5020);

     // Prepare for Heartbeat task (with state access)
     HEARTBEAT2_Init();
     Task_ID = SCH_Add_Task(HEARTBEAT2_Update, 0, 1000, 100, 0);
     TASK_DATA_Allow_Access(Task_ID, HEARTBEAT2_STATE);

     break;
     }

   case NORMAL_03:
     {
     // We use different WDT setting in each normal mode
     WATCHDOG_Init(5200);

     // Prepare for Heartbeat3 task (with state access)
     HEARTBEAT3_Init();
     Task_ID = SCH_Add_Task(HEARTBEAT3_Update, 0, 1000, 100, 0);
     TASK_DATA_Allow_Access(Task_ID, HEARTBEAT3_STATE);

     break;
     }
   }

// Add the tasks below in *ALL* modes in this demo

// Add watchdog task
SCH_Add_Task(WATCHDOG_Update, 0, 1, 10, 0);

// Report current system mode on 7-seg LED display
SPI_7_SEG_Init();
Task_ID = SCH_Add_Task(SPI_7_SEG_Update, 2, 1000, 20, 0);
TASK_DATA_Allow_Access(Task_ID, MODE_7_SEG);

// Add the tasks below only in "normal" modes
if ((System_mode_G == NORMAL_01) ||
    (System_mode_G == NORMAL_02) ||
    (System_mode_G == NORMAL_03))
   {
   // Poll SW3 on EA baseboard
   SWITCH_SW3_Init();
   Task_ID = SCH_Add_Task(SWITCH_SW3_Update, 1, 10, 20, 0);
   TASK_DATA_Allow_Access(Task_ID, SW3_STATE);

   // In "normal" modes, we change mode every 10 seconds
   // or when SW3 is pressed
   SCH_Add_Task(SYSTEM_Update_Normal, 0, 1000, 100, 50);
   }

WATCHDOG_Update(0);
}
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 4 of 10]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Update_Normal()

  Periodic task that controls system mode transitions.

  This version is employed in all "normal" modes.

  Call every second in this is demo.

-*-------------------------------------------------------------------*/
uint32_t SYSTEM_Update_Normal(const uint32_t TASK_ID)
   {
   static uint32_t Seconds = 0;
   uint32_t Change_mode = 0;
   eSystem_mode New_mode;

   // Check for system overload
   if (Fault_code_G == FAULT_SCH_SYSTEM_OVERLOAD)
      {
      // In a "normal" mode - move to "limp home"
      SYSTEM_Change_Mode(LIMP_HOME);
      }

   // If there are no overloads, then ...

   // Read the stored switch state ...
   if (TASK_DATA_Read(SW3_STATE))
      {
      // Switch pressed
      Change_mode = 1;
      }

   // ... otherwise change mode every 10 seconds
   if (Seconds++ == 10)
      {
      Seconds = 0;
      Change_mode = 1;
      }

   // Handle mode changes
   if (Change_mode == 1)
      {
      switch (System_mode_G)
         {
         default:
         case FAIL_SILENT:
            {
            // Should not reach here ...
            New_mode = FAIL_SILENT;
            break;
            }
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 5 of 10]

```
      case NORMAL_01:
         {
         New_mode = NORMAL_02;
         break;
         }

      case NORMAL_02:
         {
         New_mode = NORMAL_03;
         break;
         }

      case NORMAL_03:
         {
         New_mode = NORMAL_01;
         break;
         }
      }

   // Change mode
   SYSTEM_Change_Mode(New_mode);
   }

return RETURN_NORMAL;
}
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 6 of 10]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Update_Limp_Home()

  Periodic task that controls system mode transitions.

  This version is employed in "limp home" mode.

  Call every second in this is demo.

-*-------------------------------------------------------------------*/
uint32_t SYSTEM_Update_Limp_Home(const uint32_t TASK_ID)
   {
   // Checks for (another) system overload
   if (Fault_code_G == FAULT_SCH_SYSTEM_OVERLOAD)
      {
      // Another overrun ...
      SYSTEM_Change_Mode(FAIL_SILENT);
      }

   // In the absence of overloads, we simply remain in "limp home" mode
   // - other behaviour can (of course) be implemented

   return RETURN_NORMAL;
   }
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 7 of 10]

```
/*-------------------------------------------------------------------*-

  SYSTEM_Change_Mode_Fault()

  Force change in the system mode in the event of a fault

  Note that - in this simple demo - we cannot do very much with
  the fault data in this function.

  In a realistic design, a version of this function is used to
  ensure that some that - for example - some faults are allowed
  to occur up to N times per hour, etc.  That is, we control the
  transition to "fail silent".

  We may also wish to use this function as the basis of a
  "black box" data recorder (and keep a record of any faults).

-*-------------------------------------------------------------------*/
void SYSTEM_Change_Mode_Fault(const uint32_t FAULT)
   {
   // Deal unidentified faults
   if (FAULT == SYS_UNIDENTIFIED_FAULT)
      {
      SYSTEM_Change_Mode(FAIL_SILENT);
      }

   // If we reach here, we are dealing with identified faults:
   // these take us first to "limp home" mode
   if ((System_mode_G == NORMAL_01) ||
       (System_mode_G == NORMAL_02) ||
       (System_mode_G == NORMAL_03))
      {
      // In a normal mode
      // => change to "limp home"
      SYSTEM_Change_Mode(LIMP_HOME);
      }
   else
      {
      // Already in a limp-home mode
      // => change to "fail silent"
      SYSTEM_Change_Mode(FAIL_SILENT);
      }
   }
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 8 of 10]

```
/*------------------------------------------------------------------*-

  SYSTEM_Change_Mode()

  Force change in the system mode (change to specific mode).

-*------------------------------------------------------------------*/
void SYSTEM_Change_Mode(const eSystem_mode NEW_MODE)
   {
   // Avoid any potential problems while we store data
   WATCHDOG_Update(0);

   // Store required new system mode in NVRAM
   LPC_RTC->GPREG0 = (uint32_t) NEW_MODE;
   LPC_RTC->GPREG1 = ~((uint32_t) NEW_MODE);  // Inverted copy

   // In this design, we are using the WDT to force a reset
   // => we check (again) that the WDT is running ...
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      // Watchdog *not* running
      // => we try to shut down as safely as possible
      SYSTEM_Perform_Safe_Shutdown();
      }

   if (NEW_MODE == FAIL_SILENT)
      {
      // We are trying to shut down because of a fault
      // If the problems are serious, we may not manage to reset
      // => attempt to perform safe shutdown in current mode
      // Note: system reset should still take place
      // => this is a "safety net"
      SYSTEM_Perform_Safe_Shutdown();
      }

   // Force system reset (if WDT enabled)
   while(1);
   }
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 9 of 10]

```
/*------------------------------------------------------------------*-

  SYSTEM_Perform_Safe_Shutdown()

  Attempt to place the system into a safe state.

  Note: Does not return and may (if watchdog is operational) result
  in a processor reset, after which the function may be called again.

  [The rationale for this behaviour is that - after the reset -
   the system MAY be in a better position to enter a safe state.
   To avoid the possible reset, adapt the code and feed the WDT
   in the loop.]

-*------------------------------------------------------------------*/
void SYSTEM_Perform_Safe_Shutdown(void)
   {
   // Used for simple fault reporting
   uint32_t Delay, j;

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

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

   // We *don't* set up the WDT in this mode.
   // (but this can be done if required in your application.)

   // Use here for fault reporting
   HEARTBEAT_Init();

   // Avoid access problems
   TASK_DATA_Allow_Access(0, HEARTBEAT_STATE);

   while(1)
       {
       // Flicker Heartbeat LED to indicate fault
       for (Delay = 0; Delay < 200000; Delay++) j *= 3;
       HEARTBEAT_Update(0);
       }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 23: TTRD08a (system_multi-mode_1769_001-2_c08a.c) [Part 10 of 10]

```
/*------------------------------------------------------------------*-

  ttc_sch_mpu_1769_001-2_c08a.c (Released 2015-01)

  --------------------------------------------------------------------

  Time-Triggered Co-operative (TTC) task scheduler for LPC1769.

  Supports tasks with contracts.
  Supports Resource Barriers.
  Supports backup tasks.
  Supports MPU.

  See "ERES (LPC1769)" book (Chapter 8)
  for further information about this scheduler.

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

// Project header
#include "../main/main.h"

// MPU support
#include "mpu_support_1769_001-2_c08a.h"

// ------ Public variable definitions ------------------------------

// Used to report faults, if required, using Heartbeat LED
// See scheduler header file for details of fault codes
uint32_t Fault_code_G;

// ------ Private variable definitions -----------------------------

// Choosing to store task data in AHB RAM (behind MPU barrier)
// Allowing 4kbytes for this array
sTask* SCH_tasks_G = (sTask*) 0x2007C000;

// Also store Backup tasks in same area
// Again, allowing 4kbytes for this array
sTask* SCH_backup_tasks_G = (sTask*) 0x2007D000;

// The current tick count (also behind MPU barrier)
uint32_t* Tick_count_ptr_G = (uint32_t*) 0x2007E000;

// Flag indicating whether any task is running
static volatile uint32_t Task_running_G = 0;

// Flag indicating whether the scheduler is running
static volatile uint32_t Sch_running_G = 0;

// ------ Private function prototypes ------------------------------

static void SCH_Go_To_Sleep(void);

void SysTick_Handler(void);
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 1 of 10]

```
/*------------------------------------------------------------------*-

  SCH_Init()

  Scheduler initialisation function.  Prepares scheduler
  data structures and sets up timer interrupts every TICKms
  milliseconds.

  You must call this function before using the scheduler.

  [Required_SystemCoreClock frequency can be found in main.h.]

-*------------------------------------------------------------------*/
void SCH_Init(const uint32_t TICKms)
   {
   uint32_t i;

   // Reset the global fault variable
   Fault_code_G = 0;

   // Initialise tick count
   *Tick_count_ptr_G = 0;

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

   // Using CMSIS

   // Must check board oscillator frequency, etc
   // - see "system_lpc17xx.c" (in linked CMSIS project)
   //
   // *If* these values have been set correctly for your hardware
   // SystemCoreClock gives the system operating frequency (in Hz)
   if (SystemCoreClock != Required_SystemCoreClock)
      {
      // Fatal fault
      SYSTEM_Perform_Safe_Shutdown();
      }

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

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 2 of 10]

```
/*------------------------------------------------------------------*-

  SCH_Start()

  Starts the scheduler, by enabling SysTick interrupt.

  NOTES:
  * All tasks must be added before starting scheduler.
  * Any other interrupts MUST be synchronised to this tick.

-*------------------------------------------------------------------*/
void SCH_Start(void)
   {
   // Configure the MPU (full configuration):
   // this leaves the MPU in "scheduler" mode
   lpc1769_mpu_init();

   // Enable SysTick timer
   SysTick->CTRL |= 0x01;

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

   // Set flag to indicate scheduler is running
   Sch_running_G = 1;
   }

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

  SCH_Report_Status()

  Reports whether the scheduler is running (or not)
-*------------------------------------------------------------------*/
uint32_t SCH_Report_Status(void)
   {
   return Sch_running_G;
   }
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 3 of 10]

```
/*------------------------------------------------------------------*-

  SysTick_Handler()

  [Function name determined by CMIS standard.]

  This is the scheduler ISR.  It is called at a rate
  determined by the timer settings in the SCH_Init() function.

-*------------------------------------------------------------------*/
void SysTick_Handler(void)
   {
   // Set up MPU for scheduler use
   // (may be necessary in case of TTH / TTP design or task overrun)
   lpc1769_mpu_update_scheduler_mode();

   // Increment tick count (only)
   (*Tick_count_ptr_G)++;

   // Set up MPU for task execution
   lpc1769_mpu_update_task_mode();

   // 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;
      }
   }

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

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.

-*------------------------------------------------------------------*/
void SCH_Dispatch_Tasks(void)
   {
   uint32_t Index;
   uint32_t Update_required = 0;
   uint32_t Status, Backup_status;

   __disable_irq();  // Protect shared resource (Tick_count_G)
   lpc1769_mpu_update_scheduler_mode();
   if ((*Tick_count_ptr_G) > 0)
      {
      (*Tick_count_ptr_G)--;
      Update_required = 1;
      }
   __enable_irq();
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 4 of 10]

```
while (Update_required)
   {
   // Go through the task array
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      // Check if there is a task at this location
      if (SCH_tasks_G[Index].pTask)
         {
         if (--SCH_tasks_G[Index].Delay == 0)
            {
            // Set "Task_running" flag
            __disable_irq();
            Task_running_G = 1;
            __enable_irq();

            // Set up MPU for task execution
            lpc1769_mpu_update_task_mode();

            // Run the task
            Status = (*SCH_tasks_G[Index].pTask)(Index);

            // Set up MPU again (for scheduler use)
            lpc1769_mpu_update_scheduler_mode();

            // Clear "Task_running" flag
            __disable_irq();
            Task_running_G = 0;
            __enable_irq();

            if (Status == RETURN_FAULT)
               {
               // Fault detected during task execution
               // Try to switch to a Backup Task
               Backup_status = SCH_Replace_Task(Index);

               if (Backup_status == RETURN_FAULT)
                  {
                  // No Backup Task (or it has already been tried)
                  // Here we change mode (as appropriate)
                  SYSTEM_Change_Mode_Fault(SYS_RESOURCE_FAULT);
                  }
               }

            // All tasks are periodic in this design
            // - schedule task to run again
            SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
            }
         }
      }
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 5 of 10]

```
      __disable_irq();
      // Already in scheduler (MPU) mode
      if ((*Tick_count_ptr_G) > 0)
         {
         (*Tick_count_ptr_G)--;
         Update_required = 1;
         }
      else
         {
         Update_required = 0;
         }
      __enable_irq();
      }

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

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

  SCH_Add_Task()

  Causes a task (function) to be executed at regular intervals
  or after a user-defined delay

  pTask  - The name of the task (function) to be scheduled.
           NOTE: All scheduled functions must be
           'uint32_t, const uint32_t'

  DELAY  - The interval (TICKS) before the task is first executed

  PERIOD - Task period (in ticks).  Must be > 0.

  WCET   - Worst-Case Execution Time (microseconds)

  BCET   - Best-Case Execution Time (microseconds)

  RETURN VALUE:

  Returns the position in the task array at which the task has been
  added.  If the return value is SCH_MAX_TASKS then the task could
  not be added to the array (there was insufficient space).  If the
  return value is < SCH_MAX_TASKS, then the task was added
  successfully.

  Note: The return value is used to support Resource Barriers
        and backup tasks.

-*------------------------------------------------------------------*/
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 6 of 10]

```
uint32_t SCH_Add_Task(uint32_t (* pTask)(const uint32_t),
                      const uint32_t DELAY,
                      const uint32_t PERIOD,
                      const uint32_t WCET,
                      const uint32_t BCET
                      )
   {
   uint32_t Index = 0;
   uint32_t Return_value = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
      {
      Index++;
      }

   // Have we reached the end of the list?
   if (Index == SCH_MAX_TASKS)
      {
      // Task list is full
      //
      // Set the global fault variable
      Fault_code_G = FAULT_SCH_TOO_MANY_TASKS;

      // Prepare to return a fault code
      Return_value = SCH_MAX_TASKS;
      }

   // Check for "one shot" tasks
   // - not permitted in this design
   if (PERIOD == 0)
      {
      // Set the global fault variable
      Fault_code_G = FAULT_SCH_ONE_SHOT_TASK;

      // Also return a fault code
      Return_value = SCH_MAX_TASKS;
      }

   if (Return_value != SCH_MAX_TASKS)
      {
      // If we're here, there is a space in the task array
      SCH_tasks_G[Index].pTask  = pTask;

      SCH_tasks_G[Index].Delay  = DELAY + 1;
      SCH_tasks_G[Index].Period = PERIOD;
      SCH_tasks_G[Index].WCET   = WCET;
      SCH_tasks_G[Index].BCET   = BCET;

      Return_value = Index; // return position of task
      }

   // Single return point
   return Return_value;
   }
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 7 of 10]

```
/*------------------------------------------------------------------*-

  SCH_Add_Backup_Task()

  Adds a Backup Task to the schedule.

  The Backup Task will replace the corresponding Main Task if
  a fault is detected in the Main Task.

  pTask  - The name of the task (function) to be scheduled.
           NOTE: All scheduled functions must be
           'uint32_t, const uint32_t'

 [DELAY  - Same as the Main Task]

 [PERIOD - Same as the Main Task.]

  WCET   - Worst-Case Execution Time (microseconds)
           Usually <= the corresponding value
           for the Main Task.

  BCET   - Best-Case Execution Time (microseconds)
           Usually >= the corresponding value
           for the Main Task.

  RETURN VALUE: None.

  Note: Parameters are not checked!

-*------------------------------------------------------------------*/
void SCH_Add_Backup_Task(const uint32_t INDEX,
                         uint32_t (* pFunction)(uint32_t),
                         const uint32_t WCET,
                         const uint32_t BCET)
   {
   SCH_backup_tasks_G[INDEX].pTask  = pFunction;

   SCH_backup_tasks_G[INDEX].Delay  = SCH_tasks_G[INDEX].Delay;
   SCH_backup_tasks_G[INDEX].Period = SCH_tasks_G[INDEX].Period;
   SCH_backup_tasks_G[INDEX].WCET   = WCET;
   SCH_backup_tasks_G[INDEX].BCET   = BCET;
   }
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 8 of 10]

```
/*------------------------------------------------------------------*-

  SCH_Replace_Task()

  Aim to replace Main Tasks with a corresponding Backup Task, if:
  [i]  a Backup Task is available, and
  [ii] any Backup Task is not already in use

  If there is no Backup Task (or this is already in use),
  we return a fault code (and expect the system to Fail Silent).

  RETURN VALUE:  RETURN_FAULT or RETURN_NORMAL

-*------------------------------------------------------------------*/
uint32_t SCH_Replace_Task(const uint32_t TASK_INDEX)
   {
   uint32_t Return_value = RETURN_NORMAL;

   if (SCH_tasks_G[TASK_INDEX].pTask == 0)
      {
      // No task at this location
      // Serious problem!

      // Return a fault code
      Return_value = RETURN_FAULT;
      }

   if (Return_value == RETURN_NORMAL)
      {
      if (SCH_backup_tasks_G[TASK_INDEX].pTask == 0)
         {
         // No Backup Task at this location

         // Here we treat this as a fault
         // (other behaviour may make more sense
         // in your application)
         Return_value = RETURN_FAULT;
         }
      }

   if (Return_value == RETURN_NORMAL)
      {
      // If we are here, there is a Backup Task

      // Check if the Backup Task is already in use
      if (SCH_backup_tasks_G[TASK_INDEX].pTask ==
          SCH_tasks_G[TASK_INDEX].pTask)
         {
         // We are already using the Backup Task
         // Here we treat this as a fault
         // (other behaviour may make more sense
         // in your application)
         Return_value = RETURN_FAULT;
         }
      }
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 9 of 10]

```
   if (Return_value == RETURN_NORMAL)
      {
      // If we are here, there is a Backup Task
      // and it is not currently being used

      // Use the Backup Task
      SCH_tasks_G[TASK_INDEX].pTask
      = SCH_backup_tasks_G[TASK_INDEX].pTask;

      SCH_tasks_G[TASK_INDEX].WCET
      = SCH_backup_tasks_G[TASK_INDEX].WCET;

      SCH_tasks_G[TASK_INDEX].BCET
      = SCH_backup_tasks_G[TASK_INDEX].BCET;

      Return_value = RETURN_NORMAL;
      }

   // Single return point
   return Return_value;
   }

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

  SCH_Go_To_Sleep()

  This scheduler enters 'sleep mode' between clock ticks
  to [i] reduce tick jitter; and [ii] save power.

  The next clock tick will return the processor
  to the normal operating state.

  Note: a slight performance improvement may be achieved
  if this code is pasted into the 'Dispatch' function
  (this may be at the cost of code readability & portability)

  *** May be able to make further improvements to the jitter
  *** behaviour depending on the target hardware architecture

  *** Various power-saving options can be added
  *** (e.g. shut down unused peripherals)

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

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 24: TTRD08a (ttc_sch_mpu_1769_001-2_c08a.c) [Part 10 of 10]

```
/*-----------------------------------------------------------------------*-

  mpu_support_1769_001-2_c08a.c (Released 2015-01)

  ---------------------------------------------------------------------

  Allows scheduler to make use of "memory protection unit" in
  LPC1769 microcontroller.

  Reference was made to example code on the Feabhas blog when
  creating the first version of this software.

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

// Project header
#include "../main/main.h"

// Module header
#include "mpu_support_1769_001-2_c08a.h"

#define REGION_Valid    (0x10)

#define REGION_Enabled  (0x01)

#define REGION_16K      (13 << 1)      // 2**14 == 16k
#define REGION_32K      (14 << 1)      // 2**15 == 32k
#define REGION_512K     (18 << 1)      // 2**19 == 512k
#define REGION_1024K    (19 << 1)      // 2**20 == 1m
#define REGION_2048K    (20 << 1)      // 2**21 == 2m

// Privileged Read Write, Unprivileged Read Write
#define FULL_ACCESS                (0x03 << 24)

// Privileged Read, Unprivileged Read
#define READ_ONLY                  (0x06 << 24)

// TEX:0b001 S:0b0 C:0b0 B:0b0
#define NORMAL                     (8 << 16)

// TEX:0b000 S:0b0 C:0b1 B:0b1 - Periph
#define DEVICE_NON_SHAREABLE       (3 << 16)

// TEX:0b000 S:0b0 C:0b0 B:0b0
#define STRONGLY_ORDERED_SHAREABLE (0 << 16)

// All Instruction fetches abort
#define NOT_EXEC                   (0x01 << 28)
```

Listing 25: TTRD08a (mpu_support_1769_001-2_c08a.c) [Part 1 of 3].

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

```
/*-----------------------------------------------------------------------*/

void lpc1769_mpu_init(void)
   {
   // Full configuration - **Scheduler** mode

   // Disable MPU
   MPU->CTRL = 0;

   // Configure Region 0 to cover 512KB Flash
   // (Normal, Non-Shared, Executable, Read-only)
   MPU->RBAR = 0x00000000 | REGION_Valid | 0;
   MPU->RASR = REGION_Enabled | NORMAL | REGION_512K | READ_ONLY;

   // Configure Region 1 to cover CPU 32KB SRAM
   // (Normal, Non-Shared, Executable, Full Access)
   MPU->RBAR = 0x10000000 | REGION_Valid | 1;
   MPU->RASR = REGION_Enabled | NOT_EXEC | NORMAL | REGION_32K
               | FULL_ACCESS;

   // Configure Region 0 to cover AHB 32KB SRAM
   // (Normal, Non-Shared, Not executable, Full access)
   // REGION SET TO FULL ACCESS
   MPU->RBAR = 0x2007C000 | REGION_Valid | 2;
   MPU->RASR = REGION_Enabled | NOT_EXEC | NORMAL | REGION_32K
               | FULL_ACCESS;

   // Configure Region 3 to cover 16KB GPIO
   // (Device, Non-Shared, Full Access Device, Full Access)
   MPU->RBAR = 0x2009C000 | REGION_Valid | 3;
   MPU->RASR = REGION_Enabled |DEVICE_NON_SHAREABLE | REGION_16K
               | FULL_ACCESS;

   // Configure Region 4 to cover 512KB APB Peripherials
   // (Device, Non-Shared, Full Access Device, Full Access)
   MPU->RBAR = 0x40000000 | REGION_Valid | 4;
   MPU->RASR = REGION_Enabled | DEVICE_NON_SHAREABLE | REGION_512K
               | FULL_ACCESS;

   // Configure Region 5 to cover 2MB AHB Peripherials
   // (Device, Non-Shared, Full Access Device, Full Access)
   MPU->RBAR = 0x50000000 | REGION_Valid | 5;
   MPU->RASR = REGION_Enabled | DEVICE_NON_SHAREABLE | REGION_2048K
               | FULL_ACCESS;

   // Configure Region 6 to cover the 1MB PPB
   // (Privileged, XN, Read-Write)
   MPU->RBAR = 0xE0000000 | REGION_Valid | 6;
   MPU->RASR = REGION_Enabled | STRONGLY_ORDERED_SHAREABLE
               | REGION_1024K | FULL_ACCESS;

   // Enable MPU
   MPU->CTRL = 1;
   __ISB();
   __DSB();
   }
```

Listing 25: TTRD08a (mpu_support_1769_001-2_c08a.c) [Part 2 of 3].

```
/*-------------------------------------------------------------------*/
void lpc1769_mpu_update_task_mode(void)
   {
   /* Disable MPU */
   MPU->CTRL = 0;

   // Configure Region 2 to cover AHB 32KB SRAM
   // (Normal, Non-Shared, Not executable, Read only)
   // This is where scheduler data are stored
   MPU->RBAR = 0x2007C000 | REGION_Valid | 2;
   MPU->RASR = REGION_Enabled | NOT_EXEC | NORMAL | REGION_32K
               | READ_ONLY;

   // Enable MPU
   MPU->CTRL = 1;

   __ISB();
   __DSB();
   }

/*-------------------------------------------------------------------*/
void lpc1769_mpu_update_scheduler_mode(void)
   {
   // Disable MPU
   MPU->CTRL = 0;

   // Configure Region 0 to cover AHB 32KB SRAM
   // (Normal, Non-Shared, Not executable, Full access)
   // REGION SET TO FULL ACCESS
   MPU->RBAR = 0x2007C000 | REGION_Valid | 2;
   MPU->RASR = REGION_Enabled | NOT_EXEC | NORMAL | REGION_32K
               | FULL_ACCESS;

   // Enable MPU
   MPU->CTRL = 1;

   __ISB();
   __DSB();
   }

/*-------------------------------------------------------------------*/
void HardFault_Handler(void)
   {
   SYSTEM_Perform_Safe_Shutdown();
   }

/*-------------------------------------------------------------------*/
void MemManage_Handler(void)
   {
   SYSTEM_Perform_Safe_Shutdown();
   }

/*-------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-------------------------------------------------------------------*/
```

Listing 25: TTRD08a (mpu_support_1769_001-2_c08a.c) [Part 3 of 3].

```
/*-----------------------------------------------------------------------*-

  gpio_checks_1769_001-2_c08a.c (Released 2015-01)

  ---------------------------------------------------------------------

  Check functions for GPIO on LPC1769.

  [Low-level code was adapted from NXP examples.]

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

// Project header
#include "../main/main.h"

// Module header
#include "gpio_checks_1769_001-2_c08a.h"


// ------ Private function prototypes -----------------------------

static LPC_GPIO_TypeDef* GPIO_Get_Port_Ptr(const uint8_t);
```

Listing 26: TTRD08a (gpio_checks_1769_001-2_c08a.c) [Part 1 of 4]

```
/*-----------------------------------------------------------------------*-

  GPIO_Get_Port_Ptr()

  Translate port number into port pointer (for use with GPIO fns).
  Port must in the range 0-4.

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

static LPC_GPIO_TypeDef* GPIO_Get_Port_Ptr(const uint8_t PORT_NUMBER)
   {
   LPC_GPIO_TypeDef* pGPIO = NULL;

   switch (PORT_NUMBER)
      {
      case 0:
         {
         pGPIO = LPC_GPIO0;
         break;
         }
      case 1:
         {
         pGPIO = LPC_GPIO1;
         break;
         }
      case 2:
         {
         pGPIO = LPC_GPIO2;
         break;
         }
      case 3:
         {
         pGPIO = LPC_GPIO3;
         break;
         }
      case 4:
         {
         pGPIO = LPC_GPIO4;
         break;
         }
      default:
      break;
      }

   return pGPIO;
   }
```

Listing 26: TTRD08a (gpio_checks_1769_001-2_c08a.c) [Part 2 of 4]

```
/*-------------------------------------------------------------------*-

  GPIO_Check_Output()

  Check that port pin(s) is/are correctly configured for output.

-*-------------------------------------------------------------------*/
uint32_t GPIO_Check_Output(const uint8_t PORT_NUMBER,
                           const uint32_t REQD_BIT_VALUE)
   {
   uint32_t Current_bit_value;

   LPC_GPIO_TypeDef* pGPIO = GPIO_Get_Port_Ptr(PORT_NUMBER);

   if (pGPIO == NULL)
      {
      // Invalid port
      return RETURN_FAULT;
      }

   // Valid port - check pins are configured for output

   // Read current settings
   Current_bit_value = pGPIO->FIODIR;

   if ((REQD_BIT_VALUE & Current_bit_value) != REQD_BIT_VALUE)
      {
      return RETURN_FAULT;
      }

   return RETURN_NORMAL;
   }
```

Listing 26: TTRD08a (gpio_checks_1769_001-2_c08a.c) [Part 3 of 4]

```
/*-------------------------------------------------------------------*-

  GPIO_Check_Input()

  Check that port pin(s) is/are correctly configured for input.

-*-------------------------------------------------------------------*/
uint32_t GPIO_Check_Input(const uint8_t PORT_NUMBER,
                          const uint32_t REQD_BIT_VALUE)
   {
   uint32_t Current_bit_value;

   LPC_GPIO_TypeDef* pGPIO = GPIO_Get_Port_Ptr(PORT_NUMBER);

   if (pGPIO == NULL)
      {
      // Invalid port
      return RETURN_FAULT;
      }

   // Valid port - check pins are configured for input

   // Read current settings
   Current_bit_value = pGPIO->FIODIR;

   // All reqd bits should set to 0
   if (REQD_BIT_VALUE & Current_bit_value)
      {
      return RETURN_FAULT;
      }

   return RETURN_NORMAL;
   }


/*-------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-------------------------------------------------------------------*/
```

Listing 26: TTRD08a (gpio_checks_1769_001-2_c08a.c) [Part 4 of 4]

```
/*------------------------------------------------------------------*-

  task_data_checks_1769_001-2_c08a.c (Released 2015-01)

  --------------------------------------------------------------------

  Shared data (comms) for tasks.

  Protected by Resource Barriers.

   See "ERES (LPC1769)" Chapter 8 for details.

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

// Project header
#include "../main/main.h"

// ------ Private data -------------------------------------------

static uint32_t Shared_task_data_uint32[TASK_DATA_SIZE];
static uint32_t Task_data_access_uint32[TASK_DATA_SIZE];

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

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

   TASK_DATA_Init()

   Prepare the task data arrays.

-*------------------------------------------------------------------*/
void TASK_DATA_Init(void)
   {
   uint32_t i;

   // All data initialised to 0 (inverted)
   for (i = 0; i < TASK_DATA_SIZE; i++)
      {
      Shared_task_data_uint32[i] = 0xFFFFFFFF;
      }

   // All shared data "blocked" (assigned to invalid task ID)
   for (i = 0; i < TASK_DATA_SIZE; i++)
      {
      Task_data_access_uint32[i] = SCH_MAX_TASKS;
      }
   }
```

Listing 27: TTRD08a (task_data_checks_1769_001-2_c08a.c) [Part 1 of 3]

```
/*-----------------------------------------------------------------------*-

   TASK_DATA_Allow_Access()

   One task (only) is allowed write access to shared data.
   [Other options are possible - that is the implementation here.]

   This function registers the task for access.

   Note:
   Access will only be granted *before* the scheduler starts running.

-*-----------------------------------------------------------------------*/
uint32_t TASK_DATA_Allow_Access(const uint32_t TASK_ID,
                                const uint32_t DATA_INDEX)
   {
   // Only allow access links during init phase
   if (SCH_Report_Status())
      {
      // Scheduler is running - don't allow access
      return RETURN_FAULT;
      }

   __disable_irq();  // Protect shared resource
   Task_data_access_uint32[DATA_INDEX] = TASK_ID;
   __enable_irq();

   return RETURN_NORMAL;
   }

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

   TASK_DATA_Write()

   Provide write access to the task data (to registered tasks)

-*-----------------------------------------------------------------------*/
uint32_t TASK_DATA_Write(const uint32_t TASK_ID,
                         const uint32_t DATA_INDEX,
                         const uint32_t DATA)
   {
   if (Task_data_access_uint32[DATA_INDEX] != TASK_ID)
      {
      return RETURN_FAULT;
      }

   // Write access is permitted
   // => store (inverted) value
   Shared_task_data_uint32[DATA_INDEX] = ~DATA;

   return RETURN_NORMAL;
   }
```

Listing 27: TTRD08a (task_data_checks_1769_001-2_c08a.c) [Part 2 of 3]

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

```
/*-----------------------------------------------------------------*-

   TASK_DATA_Read()

   Provide read access to the task data (no restrictions).

-*-----------------------------------------------------------------*/
uint32_t TASK_DATA_Read(const uint32_t DATA_INDEX)
   {
   // Return inverted value
   return ~(Shared_task_data_uint32[DATA_INDEX]);
   }

/*-----------------------------------------------------------------*-
  ---- END OF FILE ---------------------------------------------
-*-----------------------------------------------------------------*/
```

Listing 27: TTRD08a (task_data_checks_1769_001-2_c08a.c) [Part 3 of 3]

```
/*-----------------------------------------------------------------*-

   task_data_index_1769_001-2_c08a.h (Released 2015-01)

  ------------------------------------------------------------------

   Information about shared task data that is protected using
   Resource Barriers.

   See "ERES (LPC1769)" Chapter 8 for details.

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

#ifndef _TASK_DATA_H
#define _TASK_DATA_H 1

// Length of data array
#define TASK_DATA_SIZE (20)

// Data index of Heartbeat states
#define HEARTBEAT_STATE (0)
#define HEARTBEAT1_STATE (1)
#define HEARTBEAT2_STATE (2)
#define HEARTBEAT3_STATE (3)

// Data index of SW3_STATE
#define SW3_STATE (4)

// Data index of COUNT_7_SEG
#define MODE_7_SEG (5)

#endif

/*-----------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-----------------------------------------------------------------*/
```

Listing 28: TTRD08a (task_data_index_1769_001-2_c08a.c)

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

```
/*------------------------------------------------------------------*-

  heartbeat_basic_x4_1769_001-2_c08a.c (Released 2015-01)

  ------------------------------------------------------------------

  Four simple 'Heartbeat' tasks for LPC1769.

  "Heartbeat 3" has task overrun (~3.6 ms) after 5 seconds.

  Target LPCXpresso board (LED2) plus RGB LED on EA Baseboard.

  No fault reporting.

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

// Project header
#include "../main/main.h"

// Task header
#include "heartbeat_basic_x4_1769_001-2_c08a.h"
```

Listing 29: TTRD08a (heartbeat_basic_x4_1769_001-2_c08a.c) [Part 1 of 3]

```
/*-------------------------------------------------------------------*-

  Initialisation functions

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

void HEARTBEAT_Init(void)
   {
   // Set up LED2 as an output pin
   // Params: Port : Pin : 1 for o/p, 0 for i/p
   GPIO_SetDir(HEARTBEAT_PORT, HEARTBEAT_PIN, 1);
   }

// Other initialisation functions omitted here


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

  Tasks.

  Changes state of associated LED (typically call once per second).

  PRE:  Valid pin.
  POST: No sensible checks.

  State of LED pin is monitored for each task
  (to check for interference).

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

// This version has injected fault
uint32_t HEARTBEAT_Update(const uint32_t TASK_ID)
   {
   static uint32_t Heartbeat_state = 0;

   uint32_t Precond_gpio;
   uint32_t Return_value = RETURN_NORMAL;

   // Used for fault injection
   uint32_t Delay, j;
   static uint32_t Task_overrun_counter = 0;

   // Heartbeat pin must be set for O/P
   Precond_gpio = GPIO_Check_Output(HEARTBEAT_PORT, HEARTBEAT_PIN);

   if (Precond_gpio == RETURN_FAULT)
      {
      Return_value = RETURN_FAULT;
      }

   // Check that stored copy of Heartbeat state matches local copy
   if (TASK_DATA_Read(HEARTBEAT_STATE) != Heartbeat_state)
      {
      // Mismatch ...
      Return_value = RETURN_FAULT;
      }
```

Listing 29: TTRD08a (heartbeat_basic_x4_1769_001-2_c08a.c) [Part 2 of 3]

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

```
   // Force task overrun after 5 seconds (test / demo purposes)
   if (Task_overrun_counter++ == 5)
      {
      Task_overrun_counter = 0;

      // Trigger temporary task overrun (for demo purposes)
      // This gives delay of ~3.6 ms
      for (Delay = 0; Delay < 20000; Delay++)
         {
         j *= 3;
         }
      }

   // If we have met the preconditions, we update the Heartbeat LED state
   if (Return_value == RETURN_NORMAL)
      {
      // Change the LED from OFF to ON (or vice versa)
      if (Heartbeat_state == 1)
         {
         // Update both copies of LED state (and the LED)
         Heartbeat_state = 0;
         Return_value = TASK_DATA_Write(TASK_ID, HEARTBEAT_STATE, 0);

         GPIO_ClearValue(HEARTBEAT_PORT, HEARTBEAT_PIN);
         }
      else
         {
         Heartbeat_state = 1;
         Return_value = TASK_DATA_Write(TASK_ID, HEARTBEAT_STATE, 1);

         GPIO_SetValue(HEARTBEAT_PORT, HEARTBEAT_PIN);
         }
      }

   return Return_value;
   }


// Other tasks omitted here


/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 29: TTRD08a (heartbeat_basic_x4_1769_001-2_c08a.c) [Part 3 of 3]

```
/*-----------------------------------------------------------------------*-

  spi_7_seg_display_mode_1769_001-2_c08a.c (Released 2015-01)

  ---------------------------------------------------------------------

  Display mode on 7-segment LED panel on EA Baseboard.

  (Borrows from NXP example program - thanks due to NXP Vietnam)

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

// Project header
#include "../main/main.h"

// Task header
#include "spi_7_seg_display_mode_1769_001-2_c08a.h"

// ------ Public variable declarations -----------------------------

// See scheduler for definition
extern uint32_t Fault_code_G;

// See system module for definition
extern uint32_t System_mode_G;

// ------ Private constants ----------------------------------------

static const uint8_t segmentLUT[11] =
   {
   // FCPBAGED
   (uint8_t) ~0b11011011, // 0
   (uint8_t) ~0b01010000, // 1
   (uint8_t) ~0b00011111, // 2
   (uint8_t) ~0b01011101, // 3
   (uint8_t) ~0b11010100, // 4
   (uint8_t) ~0b11001101, // 5
   (uint8_t) ~0b11001111, // 6
   (uint8_t) ~0b01011000, // 7
   (uint8_t) ~0b11011111, // 8
   (uint8_t) ~0b11011101, // 9
   (uint8_t) ~0b10001110  // 10 'F' => Fault
   };

// ------ Private variable definitions ------------------------------

static uint8_t rxBuff[5];
static uint8_t txBuff[5];

static SSP_CFG_Type sspChannelConfig;
static SSP_DATA_SETUP_Type sspDataConfig;
```

Listing 30: TTRD08a (spi_7_seg_display_mode_1769_001-2_c08a.c) [Part 1 of 4]

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

```
/*------------------------------------------------------------------*-

  SPI_7_SEG_Init()

  Prepare for SPI_7_SEG_Update() function - see below.

-*------------------------------------------------------------------*/
void SPI_7_SEG_Init(void)
   {
   LPC_PINCON->PINSEL0 |= 0x2<<14; // SCK1
   LPC_PINCON->PINSEL0 |= 0x2<<18; // MOSI1

   // Set up chip select pin for output
   // Params: Port : Pin : 1 for o/p, 0 for i/p
   GPIO_SetDir(LED_DISPLAY_PORT_CS, LED_DISPLAY_PIN_CS, 1);

   sspDataConfig.length = 1;
   sspDataConfig.tx_data = txBuff;
   sspDataConfig.rx_data = rxBuff;

   SSP_ConfigStructInit(&sspChannelConfig);
   SSP_Init(SSP_CHANNEL, &sspChannelConfig);
   SSP_Cmd(SSP_CHANNEL, ENABLE);
   }
```

Listing 30: TTRD08a (spi_7_seg_display_mode_1769_001-2_c08a.c) [Part 2 of 4]

```
/*-------------------------------------------------------------------*-

  SPI_7_SEG_Update()

  Displays current system mode on 7-segment display.

  (SPI interface.)

-*-------------------------------------------------------------------*/
uint32_t SPI_7_SEG_Update(const uint32_t TASK_ID)
   {
   uint32_t Precond_gpio;
   uint32_t Return_value = RETURN_NORMAL;

   // Precondition
   // Check that CS pin is set for O/P
   Precond_gpio
   = GPIO_Check_Output(LED_DISPLAY_PORT_CS, LED_DISPLAY_PIN_CS);

   if (Precond_gpio == RETURN_FAULT)
      {
      Return_value = RETURN_FAULT;
      }

   if (Return_value == RETURN_NORMAL)
      {
      // Clear CS pin
      GPIO_ClearValue(LED_DISPLAY_PORT_CS, LED_DISPLAY_PIN_CS);

      // Display system mode
      txBuff[0] = segmentLUT[(uint32_t) System_mode_G];

      // Transfer to 7-Segment Display Driver
      SSP_ReadWrite(SSP_CHANNEL, &sspDataConfig, SSP_TRANSFER_POLLING);

      // Set CS pin to 1
      GPIO_SetValue(LED_DISPLAY_PORT_CS, LED_DISPLAY_PIN_CS);
      }

   return Return_value;
   }
```

Listing 30: TTRD08a (spi_7_seg_display_mode_1769_001-2_c08a.c) [Part 3 of 4]

```
/*------------------------------------------------------------------*-

  SPI_7_SEG_Display_Fault()

  Display 'F' on 7-segment display

  (SPI interface.)

-*------------------------------------------------------------------*/
void SPI_7_SEG_Display_Fault(void)
   {
   // Clear CS pin
   GPIO_ClearValue(LED_DISPLAY_PORT_CS, LED_DISPLAY_PIN_CS);

   // Add fault data to buffer
   txBuff[0] = segmentLUT[10];

   // Transfer to 7-Segment Display Driver
   SSP_ReadWrite(SSP_CHANNEL, &sspDataConfig, SSP_TRANSFER_POLLING);

   // Set CS pin to 1
   GPIO_SetValue(LED_DISPLAY_PORT_CS, LED_DISPLAY_PIN_CS);
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 30: TTRD08a (spi_7_seg_display_mode_1769_001-2_c08a.c) [Part 4 of 4]

```
/*------------------------------------------------------------------*-

   switch_ea-sw3_1769_001-2_c08a.c (Released 2015-01)

  ---------------------------------------------------------------------

   Simple switch interface code, with software debounce.

   [Reads SW3 on EA Baseboard.]

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

// Project header
#include "../main/main.h"

// Task header
#include "switch_ea-sw3_1769_001-2_c08a.h"

// ------ Private constants ---------------------------------------

// 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)

// ------ Private variable definitions ------------------------------

static uint8_t sw3_input = 0;

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

  SWITCH_SW3_Init()

  Initialisation function for the switch library.

-*------------------------------------------------------------------*/
void SWITCH_SW3_Init(void)
   {
   // Set up "SW3" as an input pin
   // Params: Port : Pin : 1 for o/p, 0 for i/p
   GPIO_SetDir(SW_RST_CNT_PORT, SW_RST_CNT_PIN, 0);

   // Note that 0 is the default (reset) value
   // The code below can be used for test purposes
   //GPIO_SetDir(SW_RST_CNT_PORT, SW_RST_CNT_PIN, 1);
   }
```

Listing 31: TTRD08a (switch_ea-sw3_1769_001-2_c08a.c) [Part 1 of 3]

```
/*-------------------------------------------------------------------*-

  SWITCH_SW3_Update()

  This is the main switch function.

  It should usually be scheduled approx. every 10 ms.

-*-------------------------------------------------------------------*/
uint32_t SWITCH_SW3_Update(const uint32_t TASK_ID)
   {
   // Initial value of local copy must be 0
   // (if standard init operation is used for shared data)
   static uint32_t SW3_STATE_local = 0;

   static uint32_t Duration = 0;
   uint32_t Precond_gpio;
   uint32_t Return_value = RETURN_NORMAL;

   // Precond check
   // Check that Switch pin is set for input
   Precond_gpio = GPIO_Check_Input(SW_RST_CNT_PORT, SW_RST_CNT_PIN);

   if (Precond_gpio == RETURN_FAULT)
      {
      Return_value = RETURN_FAULT;
      }

   // Check that stored copy of switch state matches local copy
   if (TASK_DATA_Read(SW3_STATE) != SW3_STATE_local)
      {
      // Mismatch ...
      Return_value = RETURN_FAULT;
      }
```

Listing 31: TRD08a (switch_ea-sw3_1769_001-2_c08a.c) [Part 2 of 3]

```
   // If we have met the preconditions, we read the switch
   if (Return_value == RETURN_NORMAL)
      {
      // Read "reset count" switch input (SW3)
      sw3_input = (GPIO_ReadValue(SW_RST_CNT_PORT) & SW_RST_CNT_PIN);

      if (sw3_input == SW_PRESSED)
         {
         Duration += 1;

         if (Duration > SW_THRES)
            {
            Duration = SW_THRES;

            // Update local copy of data
            SW3_STATE_local = 1;

            // Write the value
            Return_value = TASK_DATA_Write(TASK_ID, SW3_STATE, 1);
            }
         else
            {
            // Update local copy of data
            SW3_STATE_local = 0;

            // Switch pressed, but not yet for long enough
            Return_value = TASK_DATA_Write(TASK_ID, SW3_STATE, 0);
            }
         }
      else
         {
         // Switch not pressed - reset the count
         Duration = 0;

         // Update local copy of data
         SW3_STATE_local = 0;

         // Record switch status
         Return_value = TASK_DATA_Write(TASK_ID, SW3_STATE, 0);
         }
      }

   // Single return point
   return Return_value;
   }

/*-----------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-----------------------------------------------------------------*/
```

Listing 31: TRD08a (switch_ea-sw3_1769_001-2_c08a.c) [Part 3 of 3]

```
/*------------------------------------------------------------------*-

  watchdog_1769_001-2_c08a.c (Released 2015-01)

  ----------------------------------------------------------------

  'Watchdog' library for LPC1769.

  -- With Task Contracts --

  ** Jumper controlled (see below) **

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

// Project header
#include "../main/main.h"

// Task header
#include "watchdog_1769_001-2_c08a.h"

// ------ Public variable declarations ----------------------------

// See scheduler module for definition
extern uint32_t Fault_code_G;

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

  WATCHDOG_Init()

  Set up watchdog timer on LPC1769.

  ****************************************************************
  * Handle with care - if WDT is running, debug links may be lost *
  * In this design, WDT is enable only when jumper is inserted.   *
  ****************************************************************

  The watchdog timer is driven by the Internal RC Oscillator:
  the minimum available timeout is 256 usec.

-*------------------------------------------------------------------*/
```

Listing 32: TRD08a (watchdog_1769_001-2_c08a.c) [Part 1 of 3]

```
void WATCHDOG_Init(const uint32_t WDT_TIMEOUTus)
   {
   uint32_t wdt_ticks = 0;
   uint32_t Jumper_input;

   // *If* WDT jumper is in place, we start the WDT

   // Read WDT jumper setting
   // - set up jumper pin for input
   // - params: Port : Pin : 1 for o/p, 0 for i/p
   GPIO_SetDir(WDT_JUMPER_PORT, WDT_JUMPER_PIN, 0);

   // Note: we only read the jumper during system init phase
   Jumper_input = (GPIO_ReadValue(WDT_JUMPER_PORT) & WDT_JUMPER_PIN);

   if (Jumper_input == WDT_JUMPER_INSERTED)
      {
      // Jumper is in place
      // Set up the WDT

      // Drive WDT from internal RC timer (IRC)
      LPC_WDT->WDCLKSEL = 0x00;

      // Calculate required tick count for WDT timeout
      wdt_ticks  = (((WDT_PCLK) / WDT_US_INDEX) * (WDT_TIMEOUTus / 4));

      // Check if tick count is within allowed range
      if ((wdt_ticks >= WDT_TIMEOUT_MIN)
           && (wdt_ticks <= WDT_TIMEOUT_MAX))
         {
         LPC_WDT->WDTC = wdt_ticks;
         }
      else
         {
         // We simply "stop" if WDT values are wrong
         // - other solutions may make sense for your application
         // - for example, use closest available timeout.
         while(1);
         }

      // Reset if WDT overflows
      LPC_WDT->WDMOD |= 0x02;

      // Start WDT
      LPC_WDT->WDMOD |= 0x01;

      // Feed watchdog
      WATCHDOG_Update(0);
      }
   }
```

Listing 32: TRD08a (watchdog_1769_001-2_c08a.c) [Part 2 of 3]

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

```
/*-------------------------------------------------------------------*-

  WATCHDOG_Update()

  Feed the watchdog timer.

  See Watchdog_Init() for further information.

-*-------------------------------------------------------------------*/
uint32_t WATCHDOG_Update(const uint32_t TASK_ID)
   {
   uint32_t Return_value = RETURN_NORMAL;

   // Precondition: is the WDT running?
   if ((LPC_WDT->WDMOD & 0x01) == 0)
      {
      // WDT is not running
      Return_value = RETURN_FAULT;
      }

   if (Return_value == RETURN_NORMAL)
      {
      // Feed the watchdog
      __disable_irq();
      LPC_WDT->WDFEED = 0xAA;
      LPC_WDT->WDFEED = 0x55;
      __enable_irq();
      }

   return Return_value;
   }

/*-------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-------------------------------------------------------------------*/
```

Listing 32: TRD08a (watchdog_1769_001-2_c08a.c) [Part 3 of 3]

CHAPTER 9: Task Contracts (Time Barriers)

In this chapter we explore architectures for monitoring task execution times while the system is running. These architectures are used to support the use of Task Contracts by implementing Time Barriers in the system.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD09a: TTC MoniTTor architecture (internal)

9.1. Introduction

In Chapter 8, we explored various ways in which we could detect that resources (such as peripherals or data) that had been allocated to a particular task had suffered from some form of interference.

Our approach to this problem involved the use of what we called "Task Contracts" (Figure 75).

By the end of Chapter 8, we had added various fault-detection mechanisms to our code. In doing so, our aim was to demonstrate that:

- By making appropriate use of the memory-protection unit in our target processor (in this case an LPC1769) we can increase the chances of detecting activity by one or more tasks that might have an impact on the operation of the scheduler.
- By checking the configuration settings of hardware components (e.g. watchdog timers and I/O ports) before we use them in our tasks, we can increase our chances of detecting that the resources required by a given task had been altered, reconfigured or damaged by another source.
- By maintaining two copies of data that are used to record a task state and / or transmitted to another task, we can further increase our chances of detecting situations in which a given task has not been able to operate with the required level of "Freedom From Interference".

Our consideration of Task Contracts (TCs) is not yet quite complete. As Figure 75 shows, implementation of a TC usually also requires us to check for situations in which a task breaches its WCET or BCET conditions.

We can perform these checks by means of "Time Barriers", implemented using some simple techniques that are explored in this chapter.

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

```
uint32_t Task_With_Contract(const uint32_t TASK_ID)
   {
   // Check pre conditions (incl. shared data)

   ...

   // On-going checks

   ...

   // Check post conditions

   return Task_status;
   }
```

Timing conditions
(WCET, BCET)

[Checked by scheduler]

Figure 75: A schematic representation of a "Task Contract" implementation.

9.2. An evolving system architecture

In Chapter 2, we presented the system architecture shown in Figure 76.

This architecture provides basic mechanisms for: [i] performing a processor reset, usually by means of a watchdog timer (WDT) in the systems that we have considered so far in this book; [ii] retrieving the required system mode, based on the values recovered from the WDT register and / or using information from NVRAM (or some other form of memory device) after the reset; and [iii] configuring the system in the required operating mode as laid out in the SYSTEM_Init() function.

To support the techniques described in this chapter, and incorporate the techniques introduced in Chapter 8, we will extend this architecture as illustrated in Figure 77.

The architecture incorporates mechanisms for monitoring the task WCET and BCET of the system tasks. This makes use of information about the task execution times (information that will usually be stored in the scheduler task array), and adding a simple MoniTTor unit to the system design.

We describe the operation of this MoniTTor unit in the remainder of this chapter.

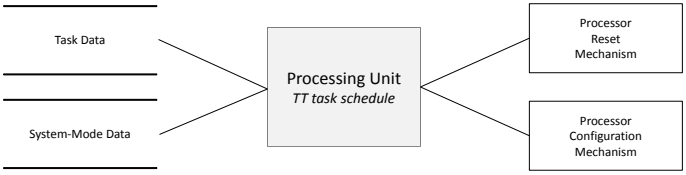


Figure 76: The system platform employed in our discussions in previous chapters.

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

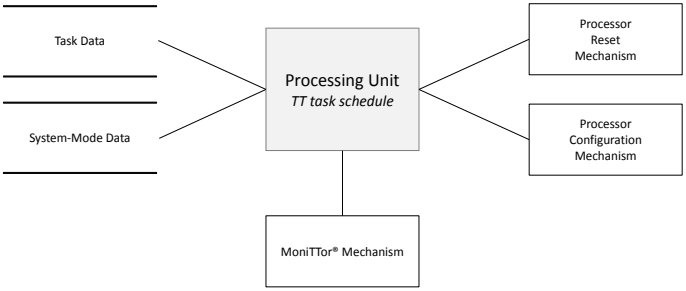


Figure 77: Possible extensions to the basic system architecture to support both Resource Barriers and Time Barriers.

9.3. System operation

The overall operation of the MoniTTor unit is illustrated in Figure 78.

The unit operates as follows.

We start a "MoniTTor timer" just before we release each task: this will usually be implemented using a spare, free-running timer component: for example, we will use Timer 0 in the LPC1769 platform in the examples in this chapter.

The timer will be set to overflow – and trigger an interrupt – after an interval that is a little longer than the WCET of the task that is being monitored. If the interrupt is triggered, then we know that the task has overrun (Figure 79).

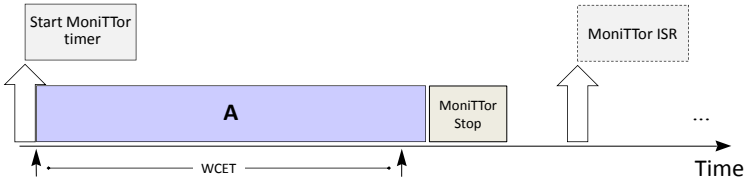


Figure 78: A schematic representation of the operation of the MoniTTor unit that is described in this chapter. The unit generates an interrupt in the event of a task overrun, and – in other circumstances - allows the task-execution time to compared with "best case execution time" limits.

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

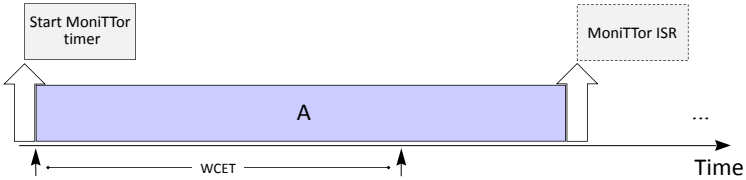


Figure 79: The operation of the MoniTTor when a task overruns.

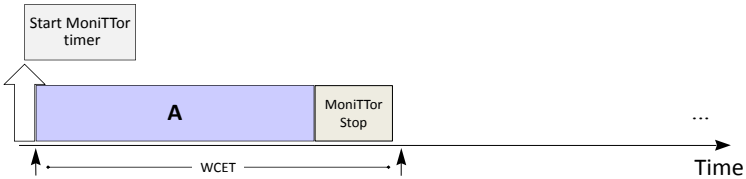


Figure 80: The operation of the MoniTTor when the task completes without overrunning.

Assuming that the task does not overrun, the MoniTTor Timer will be stopped when the task finishes running (Figure 80), and the current timer value will be used to determine the task execution time (ET): if the ET is less than the BCET value stored for the task, then we assume that a fault has occurred.

9.4. Handling execution-time faults

Decisions about how best to handle faults will always – of course – depend on the application: our goal in this text is to present a flexible TT platform that can be adapted to meet the needs of various different applications.

When handling faults that were detected by the Resource Barrier (in Chapter 8), we built on the "graceful degradation" approach outlined in Chapter 7. The resulting architecture involved structuring the system into: [i] a set of one or more Normal Modes; plus [ii] a set of one or more reduced-complexity (and reduced functionality) Limp-Home Modes; plus [iii] a set of one or more static Fail-Silent Modes.

Figure 81 illustrates a use of a similar architecture in designs that incorporate a MoniTToring unit. The figure represents a system that may have multiple Normal Modes. If a resource-related fault or execution-time fault is detected, the system moves into the (or a) Limp-Home Mode. If any other fault is detected, then the system moves into the (or a) Fail-Silent Mode.

Figure 82 illustrates another option. In this case, the system will return from a Limp-Home Mode to a Normal Mode, if no further a fault is detected for a period greater than N hours (where N is a value determined at design time).

There are – of course – numerous other design possibilities here.

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

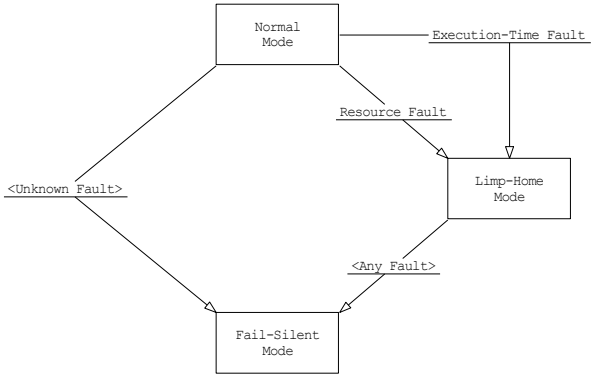


Figure 81: One way of handling execution-time faults (and resource-related faults).

9.5. Example: TTC scheduler with MoniTTor (TTRD09a)

TTRD09a details a TTC design with a monitoring unit (a "MoniTTor").

To understand how this TTRD operates, please start with Code Fragment 42. With the exception of one change to the type of functions that can be scheduled (in Chapter 8) we have used the same data structure throughout the previous examples in this book. In all of these cases, we've been storing information about the WCET and BCET of each task, but – apart from acting as a form of documentation – this information has not served much useful purpose.

We change this situation in TTRD09a.

In this design, we implement the MoniTTor described in Section 9.3 using Timer 0 in the LPC1769. Full details can be found in Listing 33.

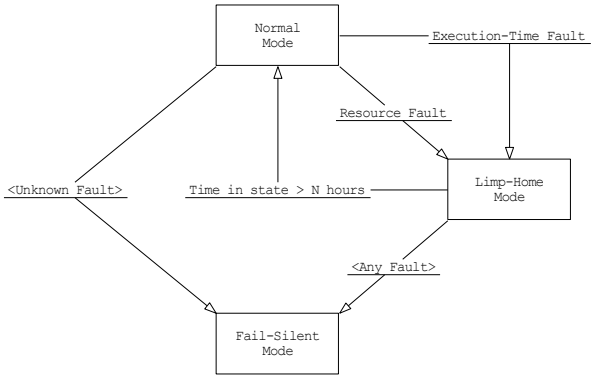


Figure 82: A design in which the system may return from a Limp-Home Mode.

9.6. Working with long tasks

Please note that the architectures presented in this chapter are designed for use with tasks that complete their execution within the tick interval in which they are released. As an example, Figure 83 illustrates a design for which this condition is not met.

```
// User-defined type to store required data for each task
typedef struct
   {
   // Pointer to the task
   // (must be a 'uint32_t (const uint32_t)' function)
   uint32_t (*pTask) (const uint32_t);

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

   // Interval (ticks) between subsequent runs.
   uint32_t Period;

   // Worst-case execution time (microseconds)
   uint32_t WCET;

   // Best-case execution time (microseconds)
   uint32_t BCET;
   } sTask;
```

Code Fragment 42: The user-defined type that is used to store information about the tasks that are controlled using the scheduler detailed in TTRD09a. The structure members used to store information about task execution times are highlighted.

```
// Before running the task, start the internal MoniTTor
MONITTOR_I_Start(Index, SCH_tasks_G[Index].WCET,
                 SCH_tasks_G[Index].BCET, 10);

(*SCH_tasks_G[Index].pTask)();  // Run the task

// Stop the internal MoniTTor
MONITTOR_I_Stop();
```

Code Fragment 43: Monitoring task execution using Timer 0.

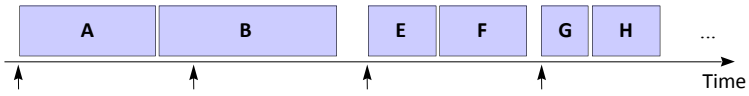


Figure 83: A system design in which Task B completes in the tick interval following the one in which it was released.

The MoniTTor units presented in this chapter may not operate as expected when used in designs like that shown in Figure 83, because the overhead from the timer tick may be enough to cause the task to exceed its expected WCET limit.

We present designs in Chapter 12 that can be used to monitor execution times in systems where one or more tasks complete after a tick interrupt.

9.7. External MoniTTor solutions

The monitoring solutions considered in this chapter are implemented on the same processor as the main system scheduler and the system tasks.

In Chapter 14, we will consider some alternative architectures that employ an external monitoring system. External solutions can reduce the risk that common-cause faults will occur on both the main processor and in the monitoring unit. One implication of this is that – if the main processor fails – the external monitoring system may still be able to ensure that the system as a whole is shut down safely.

9.8. Alternatives to MoniTTor

Developers familiar with traditional fault-handling mechanisms in TT designs may have employed "Task Guardians" in some systems (e.g. see Hughes and Pont, 2008). Task Guardian mechanisms can be very effective, particularly where they are employed in conjunction with appropriate hardware. However, when used in the absence of specialised hardware, traditional Task Guardians tend to require more complex software frameworks than the MoniTTor solutions presented in this chapter.[20]

9.9. Conclusions

In this chapter, we've begun to explore architectures for monitoring task execution times while the system is running. In the next chapter, we explore some related techniques for monitoring task execution sequences.

[20] The MoniTTor solutions described in this chapter do not require specialised hardware: they can be applied with any COTS processor platform.

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

9.10. Code listings (TTRD09a)

```
/*------------------------------------------------------------------*-

  monittor_i_t0_1769_003-0_c09a.c (Released 2015-02)

  --------------------------------------------------------------------

  MoniTTor mechanism (internal) for LPC1769.  Uses Timer 0.

  See "ERES: LPC1769" (Chapter 9) for further information.

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

// Project header
#include "../monittor/monittor_i_t0_1769_003-0_c09a.h"

// LPC1769 - Timer 0
#include <lpc17xx_timer.h>

// ------ Private variable definitions -----------------------------

// Timer 0 match register
static TIM_MATCHCFG_Type TMR0_Match;

// ID of task that is currently being monitored
static uint32_t Task_ID_G;
static uint32_t Task_BCET_G;
static uint32_t Task_t_variation_G;
```

Listing 33: TRD09a (monittor_i_t0_1769_003-0_c09a.c) [Part 1 of 5]

```
/*-----------------------------------------------------------------------*-

  MONITTOR_I_Init()

  Set up Timer 0 as MoniTTor unit.

-*-----------------------------------------------------------------------*/
void MONITTOR_I_Init(void)
   {
   TIM_TIMERCFG_Type TMR0_Cfg;

   // Power-on Timer 0
   LPC_SC->PCONP |= 1 << 1;

   // Initialise Timer 0, prescale counter = 1 µs
   TMR0_Cfg.PrescaleOption = TIM_PRESCALE_USVAL;
   TMR0_Cfg.PrescaleValue = 1;

   // Use channel 0, MR0
   TMR0_Match.MatchChannel = 0;

   // Set timer flag when MR0 matches the value in TC register
   TMR0_Match.IntOnMatch = ENABLE;

   // Enable reset on MR0: timer will reset if MR0 matches it
   TMR0_Match.ResetOnMatch = TRUE;

   // Stop timer if MR0 matches it
   TMR0_Match.StopOnMatch = TRUE;

   // Do nothing for external output pin if match
   TMR0_Match.ExtMatchOutputType = TIM_EXTMATCH_NOTHING;

   // Start timer configuration
   TIM_Init(LPC_TIM0, TIM_TIMER_MODE, &TMR0_Cfg);

   // IMPORTANT:
   // Interrupt priorities must be set in Scheduler init function

   // [Not yet started timer]
   }
```

Listing 33: TRD09a (monittor_i_t0_1769_003-0_c09a.c) [Part 2 of 5]

```
/*-----------------------------------------------------------------------*-

  MONITTOR_I_Start()

  Start monitoring a task (call just before task is released).

  ALLOWED_TIMING_VARIATION_us gives some flexibility in both
  WCET and BCET measurements.

-*-----------------------------------------------------------------------*/
void MONITTOR_I_Start(const uint32_t TASK_ID,
                      const uint32_t TASK_WCET_us,
                      const uint32_t TASK_BCET_us,
                      const uint32_t ALLOWED_TIMING_VARIATION_us)
   {
   Task_ID_G = TASK_ID;
   Task_BCET_G = TASK_BCET_us;
   Task_t_variation_G = ALLOWED_TIMING_VARIATION_us;

   // Timeout value (in microseconds)
   TMR0_Match.MatchValue = (TASK_WCET_us + ALLOWED_TIMING_VARIATION_us);

   // Complete timer configuration
   TIM_ConfigMatch(LPC_TIM0, &TMR0_Match);

   // Enable interrupt for Timer 0
   NVIC_EnableIRQ(TIMER0_IRQn);

   LPC_TIM0->TCR = 0x02;   // Reset timer
   LPC_TIM0->TCR = 0x01;   // Start timer
   }
```

Listing 33: TRD09a (monittor_i_t0_1769_003-0_c09a.c) [Part 3 of 5]

```
/*-----------------------------------------------------------------------*-

  MONITTOR_I_Stop()

  Stop monitoring a task (call just after task completes).

  Task has not overrun - need to check for possible underrun.

-*-----------------------------------------------------------------------*/
void MONITTOR_I_Stop(void)
   {
   uint32_t Execution_time_us;

   // Stop the timer
   LPC_TIM0->TCR = 0;

   // Check BCET (read timer value)
   // This value is in microseconds
   Execution_time_us = LPC_TIM0->TC;

   if ((Execution_time_us + Task_t_variation_G) < Task_BCET_G)
      {
      // Task has completed too quickly
      // => Change mode
      SYSTEM_Change_Mode_Fault(FAULT_TASK_TIMING);

      // Not storing task ID here
      // Not distinguishing BCET / WCET faults
      // => either / both this can be done here, if required
      }
   }

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

  MONITTOR_I_Disable()

  Disable the MoniTTor unit.

  Typically used in the event of an unrelated fault, prior to mode
  change, to avoid MoniTTor ISR being triggered erroneously.

-*-----------------------------------------------------------------------*/
void MONITTOR_I_Disable(void)
   {
   // Stop the timer
   LPC_TIM0->TCR = 0;

   // Disable interrupt for Timer 0
   NVIC_DisableIRQ(TIMER0_IRQn);
   }
```

Listing 33: TRD09a (monittor_i_t0_1769_003-0_c09a.c) [Part 4 of 5]

```
/*-------------------------------------------------------------------*-

  TIMER0_IRQHandler()

  [Function name determined by CMIS standard.]

  This is the task overrun ISR.

  It will be triggered only if the monitored task exceeds its
  allowed WCET (and allowed overrun period)

-*-------------------------------------------------------------------*/
void TIMER0_IRQHandler(void)
   {
   // if MR0 interrupt, proceed
   if ((LPC_TIM0->IR & 0x01) == 0x01)
      {
      // Clear MR0 interrupt flag
      LPC_TIM0->IR |= 1 << 0;

      // Task has completed too slowly
      // => Change mode
      SYSTEM_Change_Mode_Fault(FAULT_TASK_TIMING);

      // Not storing task ID here
      // Not distinguishing BCET / WCET faults
      // => either / both this can be done here, if required
      }
   }

/*-------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-------------------------------------------------------------------*/
```

Listing 33: TRD09a (monittor_i_t0_1769_003-0_c09a.c) [Part 5 of 5]

CHAPTER 10: Monitoring task execution sequences

In this chapter we consider the monitoring of task execution sequences.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD10a: TTC MoniTTor-PredicTTor® (generic)
- TTRD10b: TTC MoniTTor-PredicTTor (generic, asynchronous task set)
- TTRD10c: TTC MoniTTor-PredicTTor (TSR protected by MPU)
- TTRD10d: TTC MoniTTor-PredicTTor (TSR on external EEPROM)

10.1. Introduction

As we first outlined in Chapter 1, a three-stage development process is described in this book. The first stage involves modelling the system (using one or more Tick Lists), as introduced in Chapter 4. The second stage involves building the system (for example, using a TTC scheduler, as detailed in Chapter 2 and Chapter 8). The third stage involves adding support for run-time monitoring.

By the end of Chapter 9, we had described run-time mechanisms for dealing with both resource-related and execution-time faults. In doing so, our aim was to demonstrate that:

- By making appropriate use of the memory-protection unit in our target processor we can increase the chances of detecting activity by one or more tasks that might have an impact on the operation of the scheduler.
- By checking the configuration settings of hardware components every time we use them in our tasks, we can increase our chances of detecting that the resources required by a given task have been altered, reconfigured or damaged by another source.
- By maintaining duplicate copies of variables that are used to record a task state and / or used to pass information to another task, we can further increase our chances of detecting situations in which a given task has not been able to operate with the required level of "Freedom From Interference".
- By monitoring each task when it is released and checking its execution time against pre-determined upper (WCET) and lower (BCET) limits, we can confirm that key assumptions made at design time have been met.

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

Overall, through: [i] use of a TTC architecture; [ii] due care with the task design and verification; and [iii] use of the monitoring solutions presented in Chapter 8 and Chapter 9, we can be highly confident that our tasks will meet their contract conditions.

At this point, as we should do repeatedly during the development of any reliable embedded system, we should take a step back and ask the question: "what can possibly go wrong?"

If we review the techniques that have been presented so far in this book, it is clear that - while we have achieved a high level of diagnostic coverage using the architecture and monitoring systems presented in preceding chapters - there remains one serious gap that has not been addressed by our consideration of task contracts.

To illustrate the problem, please consider Figure 84.

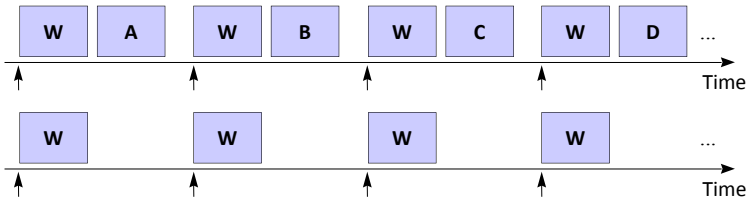


Figure 84: Two task sets the operation of which may be indistinguishable using the techniques that we have considered so far in this chapter: "W" represents a task in which a WDT is "fed".

Suppose that we intend to execute the set of tasks represented by the upper Tick List in Figure 84. Suppose that, instead, only the watchdog task is executed (as per the lower Tick List in the figure).

The monitoring mechanisms explored in Chapter 8 and Chapter 9 were designed to ensure that – once released – a task operates correctly. However, such mechanisms will not generally detect a situation in which the task does not run at all. Can we be sure that we would detect this problem?

If we further suppose that the tasks all execute. Nether of the scenarios illustrated in Figure 85 would necessarily be detected.

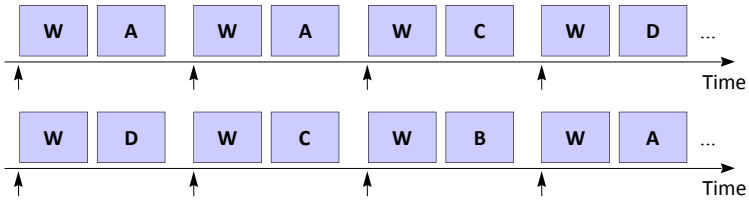


Figure 85: Examples of two incorrect task sequences.

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

Overall, a TT design is intended to execute one or more sets of tasks according to one or more pre-determined sequences: this is its *raison d'être*. However, the monitoring techniques that we have considered so far do not allow us to check that the sequence of tasks executed at run time is actually correct. We'll address this omission in the present chapter.

We call the solution presented here a "PredicTTor" unit (because it is designed to perform predictive monitoring of TT systems).

10.2. Implementing a predictive monitor

An overview of the operation of a PredicTTor unit is shown in Figure 86.

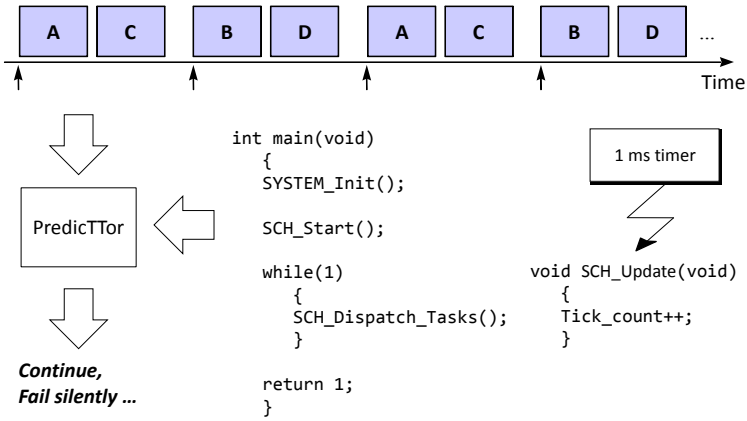


Figure 86: A schematic representation of the operation of a PredicTTor unit.

The approach involves performing the usual scheduler calculation, in order to determine which task to run next. However, before we release this task, we check with a second (independent) representation of the required task schedule in order to be sure that we are about to release the correct task.

Code Fragment 44 shows the point in a scheduler dispatch function at which the PredicTTor is employed. Note that this code fragment (taken from TTRD10a) also employs Resource Barriers and a MoniTTor unit.

The PredicTTor mechanism itself is very simple: please see Listing 34 and Listing 35 for full details. What we have is a representation of the Tick List in an array. We step through this array every time a task is released; we step back to the start of the array when we reach the end of the hyperperiod.

Note that we will consider some different ways in which these data can be stored in Section 10.14.

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

```
if (--SCH_tasks_G[Index].Delay == 0)
   {
   // Set "Task_running" flag
   __disable_irq();
   Task_running_G = 1;
   __enable_irq();

   // Set up MPU for task execution
   lpc1769_mpu_update_task_mode();

   // First check with PredicTTor
   // (avoid running the wrong task)
   PREDICTTOR_I_Check_Task_Sequence(Index);

   // Start the internal MoniTTor
   MONITTOR_I_Start(Index,
                    SCH_tasks_G[Index].WCET,
                    SCH_tasks_G[Index].BCET,
                   100);  // 100 usec leeway

   // Run the task
   Status = (*SCH_tasks_G[Index].pTask)(Index);

   // Stop the MoniTTor: this version triggers
   // a mode change in the event of a fault
   // => no return value.
   MONITTOR_I_Stop();

   // Set up MPU again (for scheduler use)
   lpc1769_mpu_update_scheduler_mode();

   // Clear "Task_running" flag
   __disable_irq();
   Task_running_G = 0;
   __enable_irq();

   if (Status == RETURN_FAULT)
      {
      // Resource-related fault
      // *Not* attemptimg to use backup tasks here
      // Here we force a mode change
      SYSTEM_Change_Mode_Fault(SYS_RESOURCE_FAULT);
      }

   // All tasks are periodic in this design
   // - schedule task to run again
   SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
   }
```

Code Fragment 44: Part of the code from the scheduler used in TTRD10a. This design incorporates a PredicTTor unit, along with a MoniTTor unit and support for Resource Barriers.

10.3. The importance of predictive monitoring

Designers of reliable embedded systems inevitably spend a considerable amount of time reviewing possible fault scenarios and considering how these should be handled.

Despite all of this effort, we do not really want to detect and handle faults at all. What we really want to do is detect incipient faults and react before System Damage can occur. In this way, we might – for example – use vibration measurements to detect that bearings are wearing out in a piece of machinery and call an operator to perform preventative maintenance (at a convenient time), thereby avoiding the need for an emergency plant shut down when the system fails completely.

When we monitor task execution sequences, we are performing a form of predictive monitoring because – unlike the approaches considered in the previous chapters – we can determine in advance that a system is about to run the wrong task. We can then alter the system behaviour before the task is actually executed.

This approach proves to be extremely powerful, and yet it is almost trivially easy to implement in a TT design. Appropriate use of a PredicTTor mechanism can prevent a system from taking actions at the wrong time (such as deploying an airbag in a passenger car, or raising the undercarriage on an aircraft). Without such a mechanism, we can often only react to such events after they have occurred.

10.4. The resulting system architecture

The system architecture that results from the addition of the PredicTTor mechanisms is shown in Figure 87.

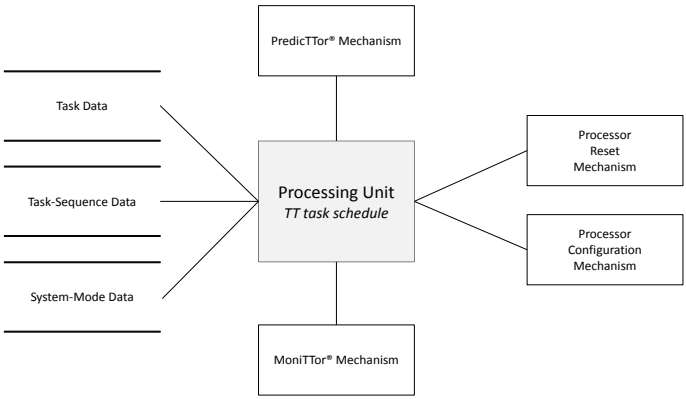


Figure 87: A system platform (TT03) with both a PredicTTor unit and a MoniTTor unit. Various different platforms are compared in Chapter 14.

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

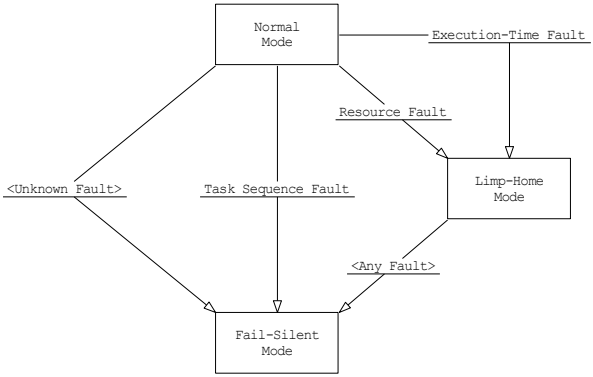


Figure 88: Possible transition between modes in an architecture that performs task-sequence monitoring.

10.5. Handling task-sequence faults

In Chapter 8, we suggested that it may be appropriate to respond to resource faults or execution-time faults by moving into a Limp-Home Mode. We also suggested (in Chapter 9) that it may be appropriate for the system to be able to return from such a mode (to a Normal Mode) in some circumstances.

In most systems, a task-sequence fault will be considered to be of greater severity than either a (single) resource-related fault or BCET / WCET violation (because it reflects a fault with the task scheduling). As a consequence, many designs will react to a task-sequence fault by moving straight to a Fail-Silent Mode.

Of course, decisions about mode transition need to be made very carefully on a system-by-system basis (but the frameworks presented here provide the required degree of flexibility).

10.6. Example: Monitoring a 3-mode system (TTRD10a)

TTRD10a provides a complete example of a system with three Normal Modes, plus a Limp-Home Mode and a Fail-Silent Mode.

The example incorporates a PredicTTor unit, a MoniTTor unit and support for Resource Barriers.

10.7. Creating the Task-Sequence Representation (TSR)

In order to obtain the task-sequence representation that is required for use in a PredicTTor design, some form of Dry Scheduler is usually employed, as we discussed in Section 5.11.

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

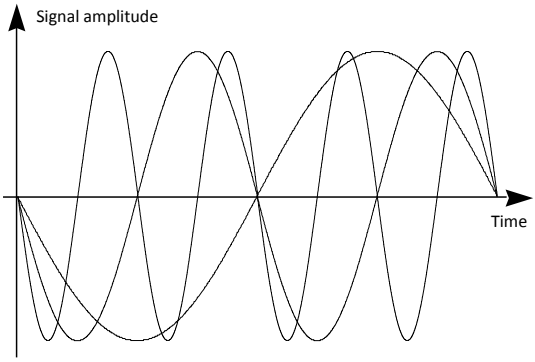


Figure 89: A set of synchronous signals.

10.8. Side effects of the use of a PredicTTor unit

One important "side effect" of the use of a PredicTTor unit is that it forces the development team to develop a full understanding of the operation of their design: without such an understanding, the system will not operate, even on a testbench.

10.9. Synchronous vs. asynchronous task sets revisited

This book is concerned with the design, implementation and monitoring of TT systems that are composed of one or more sets of periodic tasks. As we first noted in Chapter 4 such task sets may be "synchronous" or "asynchronous" in nature.

To summarise:

- If a set of periodic signals (e.g. sine waves) is synchronous, all of signals start – with no phase difference – at time t = 0 (for example, see Figure 89).
- Where a set of periodic tasks is synchronous, all tasks execute for the first time in the first tick (see, for example, Figure 90).
- Where a set of periodic tasks is asynchronous, there is a delay (or offset) – measured in ticks - before at least one task in the set begins executing (see, for example, Figure 91).

In practical systems, synchronous task sets are uncommon, because the sum of the execution time of all tasks is likely to exceed the tick interval in Tick 0 (and in later ticks, at the start of each hyperperiod) in such designs.

When a set of asynchronous periodic tasks runs, there may be an initial period during which the task sequence is not the same as the task sequence when the system is in a "steady state". We refer to this initial period as the Task Sequence Initialisation Period (TSIP).

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

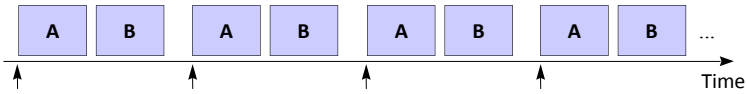


Figure 90: A synchronous task set.

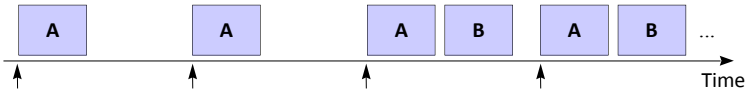


Figure 91: An asynchronous task set.

10.10. The Task Sequence Initialisation Period (TSIP)

The first question we need to ask when we wish to monitor a given task set is whether we need to take into account a TSIP.

This question is easily answered.

If the maximum task offset (that is, the maximum offset, measured in ticks, for any task in the set) is greater than or equal to the minimum task period (again measured in ticks), then we have to consider a Task Sequence Initialisation Period.

For example, suppose that we have to consider the small task set shown in Table 8.

In this example, the maximum offset is 2 ticks and the minimum period is 1 tick, so we need to consider the TSIP.

After the TSIP, the "steady state" task execution sequence begins in the tick in which the task with the largest offset runs for the first time.

For example, in the example shown in Table 8, Task Z has the largest offset (2 ticks) and this task begins in Tick 2.

In many monitoring systems, we start the monitoring process after the TSIP is complete: that is, we often ignore the TSIP when we are monitoring the system activity.

In order to start monitoring at the right time, we clearly need to know the length of the TSIP: the most useful measure of "length" in these circumstances is usually a count of task releases in the TSIP. That is, we usually wish to know how many task releases will occur before we reach the steady-state situation so that we can disregard these samples during the monitoring process.

This TSIP information will – of course - apply both when the system begins executing, and after it changes mode.

Table 8: A set of periodic tasks.

Task ID	Period (ticks)	Offset (ticks)
X	1	0
Y	2	1
Z	3	2

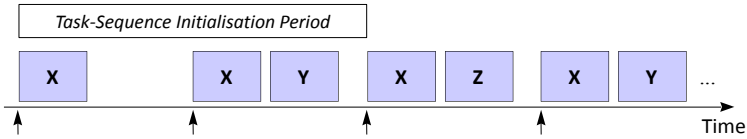


Figure 92: Counting task releases in the TSIP.

In the above example (Table 8), we need to know how many task releases will occur in Tick 0 and Tick 1. In this case, we can determine this easily (by hand): see Figure 92, from which it is clear that there are 3 tasks releases in the TSIP.

In larger designs, we will usually use a form of Dry Scheduler to determine the number of task releases that will occur in the TSIP: please see Section 5.11 for further information.

10.11. Worked example

As an example of the calculations involved in determining the TSIP length, please consider the task list shown in Table 9 (which first introduced in Chapter 4).

As we saw in Chapter 4, the required tick interval is 1 ms, and the length of the hyperperiod is 210 ticks (or 210 ms).

If the ticks are numbered from 0, when (in which tick) will the system begin to execute the "steady state" task sequence?

How long is the TSIP (measured in task releases)?

Table 9: A set of periodic tasks.

Task ID	Period (ms)	WCET (µs)	Offset (ticks)
A	1	200	0
B	2	300	0
C	2	200	1
D	2	150	1
E	3	200	11
F	5	100	13
G	7	50	25

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

10.12. Solution

As noted in Section 10.10, the "steady state" task execution sequence begins in the tick in which the task with the largest offset runs for the first time: in this case, this is Tick 25.

To work out how many task releases will occur in the TSIP, we need to work out how many task calls there will be in Tick 0 to Tick 24.

Please refer to Table 10.

Table 10: Task releases in the TSIP for the tasks in Table 9.

Task ID	Period (ms)	WCET (µs)	Offset (ticks)	TSIP releases (Tick 0 – Tick 24)
A	1	200	0	25
B	2	300	0	13
C	2	200	1	12
D	2	150	1	12
E	3	200	11	5
F	5	100	13	3
G	7	50	25	0

There are a total of 70 task releases in the TSIP.

10.13. Example: Monitoring another 3-mode system (TTRD10b)

TTRD10b shows a complete exam of a design involving two asynchronous task sets.

10.14. Where should we store the TSR?

We need to consider carefully where the TSR should be stored.

Four possibilities are as follows:

1. We can store the TSR in an array in internal memory (as we have done in the examples so far).
2. We can store the TSR in an array that is maintained "behind" the MPU (and therefore protected) in internal memory.
3. We can store the TSR in an external Flash / EEPROM device (ideally one that has been physically "write protected").
4. We can store the TSR in (or in a memory device connected to) another processor unit that is used to monitor the activity of the main processor unit.

We consider Option 4 in Chapter 14: the other options are illustrated in the TTRDs associated with this chapter.

10.15. Links to international standards

If we want to increase our chances of detecting faults, then some level of diversity in the design is generally considered to be a useful contributory factor.

For example, in ISO 26262-5 (Annex D), detailed consideration is given to ways in which the level of diagnostic coverage provided by different mechanisms can be assessed. When considering processing units, it is considered that use of software diversified redundancy can be a way of achieving what is defined in this standard as "High" diagnostic coverage (see ISO 26262-5 Table D.4 and Section D.2.3.4).

The approach followed here for task-sequence monitoring is believed to be compatible with the above requirements for software diversified redundancy.

10.16. Conclusions

In this chapter, we have considered how mechanisms for task-sequence monitoring can be incorporated in our TTC design.

In the next part of the book (Part Three), we turn our attention to TT designs that support task pre-emption.

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

10.17. Code listings (TTRD10a)

```
/*------------------------------------------------------------------*-

  predicttor_i_lpc1769_001-2_c10a.c (Released 2015-01)

  Check task sequences.

  This version assumes internal representations of task sequences
  (one sequence for each mode).

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

// Project header
#include "../main/main.h"

// Module headers
#include "predicttor_i_lpc1769_001-2_c10a.h"
#include "predicttor_i_lpc1769_tick_list_001-2_c10a.h"

// WDT
#include "../tasks/watchdog_1769_001-2_c08a.h"

// ------ Public variable declarations ----------------------------

// The current system mode
extern uint32_t System_mode_G;

// ------ Private variable definitions

// Pointer to the active Tick List
uint32_t* Tick_list_ptr_G;

// Length of current Tick List
uint32_t Length_of_tick_list_G;
```

Listing 34: TRD10a (predicttor_i_lpc1769_001-2_c10a.c) [Part 1 of 3]

```
/*------------------------------------------------------------------*-

  PREDICTTOR_I_Init()

  Prepare internal PredicTTor.

-*------------------------------------------------------------------*/
void PREDICTTOR_I_Init(void)
   {
   // Set up links to Tick List (TSR)

   switch (System_mode_G)
      {
      default:
      case FAIL_SILENT:
         {
         // Nothing to do - PredicTTor not used
         break;
         }

     case LIMP_HOME:
        {
        Tick_list_ptr_G = &TICK_LIST_LIMP_HOME[0];
        Length_of_tick_list_G = LENGTH_OF_TICK_LIST_LIMP_HOME;
        break;
        }

      case NORMAL:
        {
        Tick_list_ptr_G = &TICK_LIST_NORMAL[0];
        Length_of_tick_list_G = LENGTH_OF_TICK_LIST_NORMAL;
        break;
        }
      }
   }
```

Listing 34: TRD10a (predicttor_i_lpc1769_001-2_c10a.c) [Part 2 of 3]

```
/*-----------------------------------------------------------------*-

  PREDICTTOR_I_Check_Task_Sequence()

  Performs the checking.

  This version assumes synchronous task set.

-*-----------------------------------------------------------------*/
void PREDICTTOR_I_Check_Task_Sequence(const uint32_t TASK_ID)
   {
   static uint32_t Tick_list_index = 0;

   if (Tick_list_ptr_G[Tick_list_index] != TASK_ID)
      {
      // Task doesn't match the stored sequence ...
      // => Change mode
      SYSTEM_Change_Mode_Fault(SYS_TASK_SEQUENCE_FAULT);
      }

   // Sequence is correct - update PredicTTor state
   if (++Tick_list_index == Length_of_tick_list_G)
      {
      Tick_list_index = 0;
      }
   }

/*-----------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-----------------------------------------------------------------*/
```

Listing 34: TRD10a (predicttor_i_lpc1769_001-2_c10a.c) [Part 3 of 3]

```
/*-----------------------------------------------------------------*-

   predicttor_i_lpc1769_tick_list_001-2_c10a.h (Released 2015-01)

  -----------------------------------------------------------------

   See main PredicTTor module for details.

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

#ifndef _TICK_LIST_H
#define _TICK_LIST_H 1

// The Tick Lists for each mode
// (No Tick List for Fail Silent mode)
#define LENGTH_OF_TICK_LIST_NORMAL 1001
#define LENGTH_OF_TICK_LIST_LIMP_HOME 501

// Not used here (synchronous task sets)
#define LENGTH_OF_INITIAL_TICK_LIST_NORMAL 0
#define LENGTH_OF_INITIAL_TICK_LIST_FAIL_SILENT 0

// Tick List
static uint32_t TICK_LIST_NORMAL[LENGTH_OF_TICK_LIST_NORMAL] =
{
0,
1,
0,
0,

// Lots of data omitted here …

0,
0,
};

// Tick List
static uint32_t TICK_LIST_LIMP_HOME[LENGTH_OF_TICK_LIST_LIMP_HOME] =
{
0,
1,
0,
0,

// Lots of data omitted here …

0,
0,
};

#endif

/*-----------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*-----------------------------------------------------------------*/
```

Listing 35: TRD10a (predicttor_i_lpc1769_tick_list_001-2_c10a.h)

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

PART THREE: CREATING RELIABLE TTH AND TTP DESIGNS

"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.

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

CHAPTER 11: Supporting task pre-emption

In previous chapters, we have worked with sets of co-operative" ("run to completion") tasks. In this chapter we begin to explore the design and implementation of TT systems that support task pre-emption.

Related TTRDs

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

C programming language (LPC1769 target):

- 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

11.1. Introduction

When developing any embedded system, the author recommends that the design process should start by considering whether it will be possible to use a TTC solution, and that this – as the simplest option – should be the preferred architecture, if it meets the needs of the application.

As we have seen throughout earlier chapters in this book, TTC architectures provides a basis for real-time embedded systems that can be: [i] modelled precisely at design time; [ii] implemented using a platform based on a simple task scheduler; and [iii] monitored effectively at run time to check for timing and / or resource faults.

Unfortunately, a TTC scheduler may not always meet the needs of the application. As we first noted in Chapter 4, one way of determining whether this is the case is to check that the design is composed entirely of "short tasks", where such a task is defined as having a worst-case execution time (WCET) that is: [i] less than the period of any task in the system; and [ii] less than the system tick interval.

Another way of looking at this issue is to consider whether the following test is met for a given task set:

$$WCET \text{ (longest task)} < Period \text{ (most frequent task)}$$

Suppose, for example, that we have two tasks:

- Task A has a worst-case execution time of 10 ms, and
- Task B has a period of 5 ms.

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

We may not be able to schedule both of these tasks using a TTC scheduler, because Task A will prevent Task B from executing as required.

In these circumstances, we may need to consider using a scheduler that supports task pre-emption.

While adapting the scheduler is easy to do, this change has significant implications for many parts of the design, and may – if not carried out correctly – contribute to an unnecessary increase in system complexity and a corresponding reduction in system reliability. It is therefore not a change that should be undertaken without good reason.

One important question to ask in this situation is whether the "short-task guideline" presented above applies to your system (because this is simply a guideline rather than a strict design rule).

Suppose – for example – that Task B in our example has "soft" timing constraints, and / or Task A has a long period, then occasional jitter in the release times of Task B may be perfectly acceptable.

If we do need to adhere to the short-task guideline, then there are various ways of meeting the need for both 'frequent' and 'long' tasks in the same task set while maintaining a TTC architecture. For example, we may be able to use a faster processor and – thereby – shorten the task durations. We may be able to make better use of available hardware features (such as communication buffers or DMA mechanisms) to reduce the execution time of one or more tasks. We may also be able re-design the "long" tasks in a multi-stage form (see Figure 93, and refer to Chapter 6 for further details).

However, while we may be able to extend the number of designs for which a TTC architecture can be applied, this is not a universal solution, and – in practice – many systems involve the use of task pre-emption.

As we noted in Chapter 2, a key characteristic of TTC designs is that we can model the behaviour of the system precisely during the development process. Being able to create such models allows us to have confidence that the system will operate correctly in the field.

When we begin to incorporate task pre-emption in our system designs, our aim is to retain the key advantages of a TTC design (including the ability to model the system behaviour) while obtaining greater design flexibility. The time-triggered hybrid (TTH) and time-triggered pre-emptive (TTP) scheduler designs presented in this chapter allow us to achieve this.

We will begin this chapter by considering TTH designs. We will then go on to explore TTP options.

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

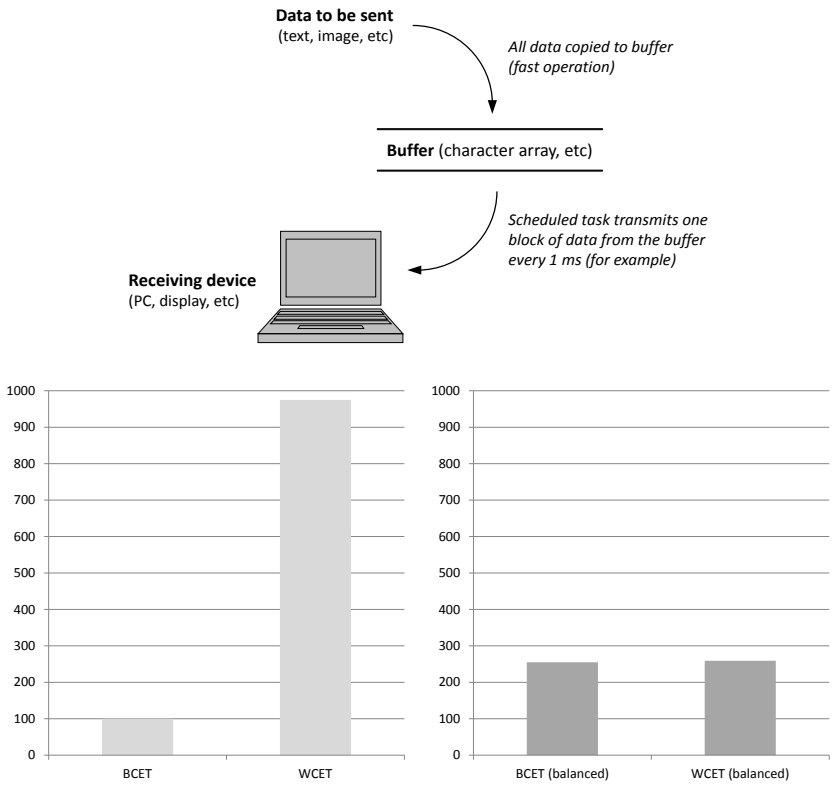


Figure 93: The impact of design decisions (top) on the BCET and WCET of a data-transfer task: (lower left) without the multi-stage architecture; (lower right) with the multi-stage architecture.

11.2. Implementing a TTH scheduler

Converting a TTC scheduler into a TTH scheduler requires very few code changes.

As an example, Code Fragment 45 shows the update function from a typical TTC scheduler (taken from TTRD02a)

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

Code Fragment 45: The 'Update' function (SysTick_Handler()) from TTRD02a.

Code Fragment 46 shows the corresponding update function from a TTH scheduler (taken from TTRD11a).

```
void SysTick_Handler(void)
   {
   // Adjust initial value to adjust offset (if required)
   static uint32_t Heartbeat_counter = 0;

   // Increment tick count
   Tick_count_G++;

   // This is a TTH scheduler - call pre-empting task(s) here

   // Called every tick
   SWITCH_SW3_Update();     // Read switch
   EMERGENCY_STOP_Update(); // Process emergency stop requests

   // Called every 1000 ticks
   if (++Heartbeat_counter == 1000)
      {
      HEARTBEAT_Update();
      Heartbeat_counter = 0;
      }
   }
```

Code Fragment 46: The 'Update' function (SysTick_Handler()) from TTRD11a.

As Code Fragment 46 should make clear, we now – in effect – have two schedulers in the system: these are the original scheduler for the co-operative tasks and an additional (much simpler) scheduler for the higher-priority tasks.

Please note that - for ease of reference - we will refer to tasks released from the task dispatcher (in main()) as "TTC" tasks: we view these as having "Priority 0". We will refer to tasks released in the scheduler ISR as "TTH" tasks: we view these as having "Priority 1" (a higher priority).

Please also note that it is possible to create a more complete "dispatcher" function for the pre-empting tasks (see Pont, 2001, Chapter 17 for a complete example). However, in most cases, we generally prefer to use a very simple framework for two reasons: [i] there are usually only a small number of pre-empting tasks, and the code used to dispatch them rarely becomes complicated; [ii] jitter in the release times of the pre-empting tasks – which can often be an issue of concern – can be controlled more precisely by using a simple architecture.

11.3. Key features of a TTH scheduler

A TTH scheduler differs from a TTC scheduler as follows:

- The assumption that all (TTC) tasks will have a worst-case execution time that is less than the tick interval is relaxed.
- The assumption that all (TTC) tasks released in a given tick interval will complete within the same tick interval is relaxed.

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

- In addition to scheduling any number of TTC tasks, **one** periodic "pre-empting section" (PS) is also supported.
- The PS may be used to release one or more TTH tasks.
- TTH tasks have a higher priority than the TTC tasks and may pre-empt (interrupt) them: we view the TTC tasks as having Priority 0, and the TTH tasks as having Priority 1.
- TTH tasks also run to completion once executed: that is, the TTH tasks cannot themselves be interrupted by any of the other tasks in the system (including other TTH tasks).

An example of a Tick List for a TTH design is shown in Figure 94. In this simple design, we have two co-operative tasks: Task A and Task B: in this case, Task A has an execution time that exceeds the system tick interval. These low-priority tasks may be interrupted by the single, high-priority task (Task P).

In the general case, the fact that there is only one PS in a TTH design, and that tasks released in this block run to completion, greatly simplifies the system architecture compared to a fully pre-emptive solution. In particular: [i] we do not need to implement a complex context-switch mechanism; [ii] arrangements for access to shared resources are greatly simplified (see Section 11.7).

In many cases, the PS will be used to improve the responsiveness of the system, as we will illustrate in our first TTH example.

11.4. TTH example: Emergency stop (TTRD11a)

TTRD11a employs a TTH scheduler to provide an "emergency stop" capability.

For illustrative purposes, the design involves running two (very) long TTC tasks: PLACEHOLDER_TASK_000_T3_Update() has an execution time of approximately 1 second, while PLACEHOLDER_TASK_001_T3_Update() has an execution time of approximately 2 seconds. These tasks are added to the TTC schedule (as in previous examples).

At the same time – as already shown in Code Fragment 46 – the system is reading an emergency stop switch and controlling a standard Heartbeat LED: this is achieved by means of TTH tasks.

Please refer to TTRD11a for full code details.

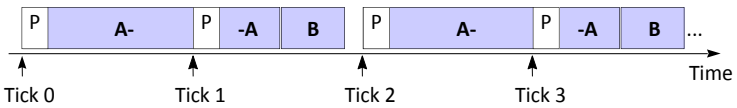


Figure 94: Executing tasks using a TTH scheduler. "A-" indicated the start of Task A: "-A" indicates the continuation of Task A (after pre-emption by Task P).

Table 11: Melodies associated with alerts (IEC 60601-1-8)

Alarm	High priority melody	Mnemonic notes
General	C4-C4-C4-C4-C4	Fixed pitch
Cardiac	C4-E4-G4-G4-C5	Trumpet call; Call to arms; Major chord
Artificial Perfusion	C4-F#4-C4-C4-F#4	Artificial sound; Tri-tone
Ventilation	C4-A4-F4-A4-F4	Inverted major chord; Rise and fall of the lungs
Oxygen	C5-B4-A4-G4-F4	Slowly falling pitches; Top of a major scale; Falling pitch of an oximeter
Temperature	C4-E4-D4-F4-G4	Slowly rising pitches; Bottom of a major scale; Related to slow increase in energy or (usually) temperature
Drug delivery	C5-D4-G4-C5-D4	Jazz chord (inverted 9th); Drops of an infusion falling and "splashing"
Power failure	C5-C4-C4-C5-C4	Falling or dropping down

11.5. TTH example: Medical alarm in compliance with IEC 60601

NXP application note AN10875 describes how to generate medical alarms in compliance with international standard IEC 60601[21] using what is essentially a TTH design.

The IEC 60601 standard proposes that a melody is generated that corresponds with a specific physiological situation or condition: the aim is to ensure that the same alarm signals are used by different manufacturers, to avoid confusing staff. The number of alarms is limited to a total of eight (each with “cautionary” and “emergency” versions). The emergency (high priority) version uses a five note melody that is repeated once. The cautionary (medium priority) alarm uses the first three notes of the high-priority version and does not repeat. Building on research in this area, medium priority alarms are required to have slower rise and fall times and a slower tempo than the high priority alarms. An optional low priority alarm tone is also provided in the standard that involves only two notes.

Please refer to Table 11 for information about the specific melodies.

The melody note C4 in Table 11 refers to middle C, with C5 being one octave above middle C. The standard states that you do not have to use those specific notes: as long as the fundamental note is within the specified frequency range, the alarm melody may be transposed to different keys and still be compliant with the specification.

[21] IEC 60601-1-8 specifies basic safety and essential performance requirements and tests for alarm systems in medical electrical equipment and medical electrical systems, and provides guidance for their application.

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

Table 12: A set of periodic tasks, some of which are jitter sensitive.

Task	Period (s)	WCET (s)	Jitter sensitive?	CPU load	Priority
A	10	2	Y	20%	1
B	10	2	Y	20%	1
C	10	2	Y	20%	1
D	10	2	Y	20%	1
E	20	3	N	15%	0
				95%	

The standard states that the note must consist of the fundamental tone and at least 4 harmonics. The fundamental and 4 harmonics must not differ by more than 15 db in amplitude.

Creating these specific melodies to meet the IEC 60601 standard is clearly not going to be a trivial process. NXP application note AN10875 targets the LPC1769 family, and employs what is essentially a TTH software architecture (with a 40 µs tick interval) to meet these requirements. Full code examples are given, and the resulting signals are shown to be compliant with the standard.

It is stated that the resulting application uses around 8% of the available processor bandwidth and less than 10 kbytes of code space.

Please refer to NXP Application Note AN10875 for further information.

11.6. TTH example: Long pre-empting section (TTRD11b)

Many TTH designs (including the two examples presented in the preceding sections) employ short TTH tasks, but this is not always the case.

For example, consider the set of periodic tasks shown in Table 12.

The average CPU load is 95%: it may therefore be possible to schedule the task set. However, doing so using a TTC scheduler would probably not be possible (because Task A, Task B, Task C and Task D are described as "jitter sensitive", and they would suffer from significant release jitter in a TTC design).

We can schedule this task set successfully, if we treat Task E as a TTC (Priority 0) task, and treat the remaining tasks as TTH (Priority 1) tasks, called from the timer ISR.

Please refer to TTRD11b for full details.

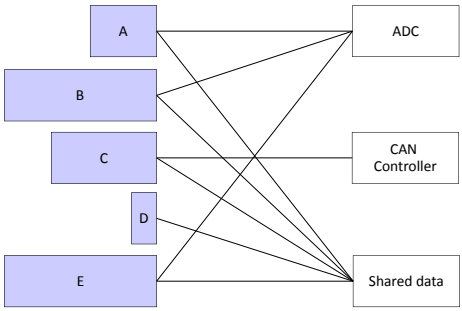


Figure 95: Shared resources in a system.

11.7. Protecting shared resources

Once we introduce the possibility of task pre-emption, we need to consider the issues of shared resources and priority inversion.

In the context of an embedded system, the phrase "shared resource" is usually used to refer to a hardware or software component that is accessed by more than one task.

Suppose that our system has 10 tasks. Some examples of shared resources might be:

- An analogue-to-digital converter (ADC) in which Channel A is used by Task 0, and Channel B is used by Task 3.
- A CAN communication module through which data is sent / read by Task 1, Task 2 and Task 9.
- Areas of memory (e.g. global variables) that are used to exchange data between the various tasks.

In <u>any</u> system that involves task pre-emption (whether it is ET or TT in architecture), we need to control the access to shared resources.

To understand these issues, we will first consider an ET design, illustrated schematically in Figure 99.

(We first explored a TT version of this design in Chapter 8.)

In the present case, we will assume that:

- the tasks have different priorities, with Task A having the highest priority and Task E having the lowest priority.
- the tasks are event triggered (and that they may therefore be released at "random" times).
- the tasks may require access to several shared resources, including an analogue-to-digital converter (ADC), a Controller Area Network (CAN) interface, and some shared data in memory.

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

The challenge we face in any such design is that a task may be accessing a shared resource when a higher-priority task also requires such access.

In order to maintain the integrity of the system in such circumstances, we need to be able to "lock" the resource while it is in use (typically using a "mutex" mechanism or similar: see, for example, Liu, 2000).

We give an example of resource locking in TTRD11c1.

11.8. A TTP scheduler with shared resources (TTRD11c1)

As we noted in Section 11.2, a TTH design is – in effect – "two schedulers in one".

Even with TTH capabilities, we may still face a situation in which:

WCET (longest TTH task) < *Period* (most frequent TTH task)

In this situation, we can "add another scheduler layer", to produce a more general "time-triggered pre-emptive" (TTP) scheduler.[22]

We refer to tasks that are released in the TTP ISR as "TTP tasks", or "Priority 2 tasks" (where TTC tasks have Priority 0 and TTH tasks have Priority 1).

Reference design TTRD11c1 illustrates the basic use of a TTP scheduler.

Building on TTRD11b, TTRD11c1 supports all of the tasks shown in Table 12. It also schedules a "Heartbeat" task once a second. Since the WCET of the Priority 0 (P0) and P1 tasks is up to 3 seconds, the Heartbeat task needs to be a TTP task (P2).

In this design, the P0 and P1 tasks share a resource. A simple "lock" is implemented in order to avoid a conflict over this resource.

11.9. The challenges of priority inversion

Once we have implemented an appropriate protection mechanism, we still need to consider potential challenges caused by the resulting "priority inversion" (PI).

PI arises when – for example Task C (in Figure 99) is using the ADC and Task A requires access. Because Task C will have "locked" the resource, Task A must wait until Task C completes access to the resource before it can continue. Thus, at this time, Task C has an "inverted" priority (it is treated as if it had a priority higher than that of Task A).

---

[22] Where required, we can add further "layers" of timer ISR to our TTP scheduler (to give a "TTP2" or "TTP3" scheduler, for example), but the author has only very rarely found it necessary to do this in a practical system.

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

To address such problems, various "ceiling protocols" have been developed (see, for example, Liu 2000 for an overview). These involve temporary changes to task priorities while resources are accessed. For example, if we consider Figure 99 again, Task C would be given a priority the same as Task A while it is accessing the ADC. This ensures that Task C will complete its use of the ADC as quickly as possible. It can also ensure that we avoid deadlock in the system.

Please note:

- While use of ceiling protocols and related locking mechanisms can ameliorate the impact of PI, they cannot remove it altogether in an ET design. What this means is that – in any ET design where there are multiple levels of task priority and shared resources – it can be very difficult to predict precisely how quickly the system will respond to any event. This can make it very challenging to guarantee real-time performance in any non-trivial ET system.
- By contrast, it is possible (in fact it is often very easy) to eliminate PI problems completely in a TT design, thereby ensuring that TTH and TTP designs can – like TTC designs – provide deterministic timing behaviour.

11.10. Implementing a "ceiling" protocol (TTRD11c2)

We will explore techniques for avoiding PI in TT designs in Chapter 12.

In the absence of such design features, TT designs may still require use of a ceiling protocol.

TTRD11c2 illustrates use of simple ceiling protocol in a TTP design.

11.11. Monitoring task execution times (TTRD11d)

We explored mechanisms for monitoring the execution time of tasks that are released by a TTC scheduler in Chapter 9. We cannot employ exactly the same mechanisms in TTH or TTP designs, but we can do something similar.

As a reminder, Figure 96 shows the MoniTTor mechanism used to check the execution times of tasks in a TTC framework.

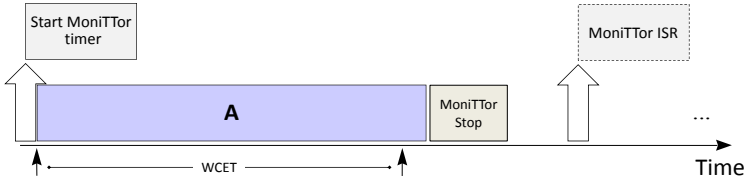


Figure 96: A schematic representation of the operation of a "Task Execution Time Monitoring Unit" (MoniTTor) in a TTC scheduler. The MoniTTor generates an interrupt in the event of a task overrun, and – in other circumstances – checks for task underruns.

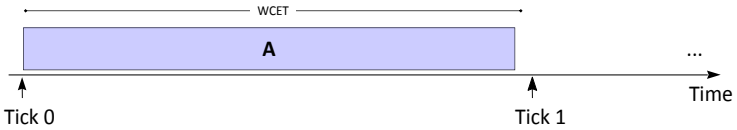


Figure 97: A task released by a TTC scheduler.

As you will recall, we started the "MoniTTor Timer" just before we release each task. The timer was set to overflow – and trigger an interrupt – after an interval that is a little longer than the WCET of the TTC task that was being monitored.

If the task did not overrun, the timer was stopped when the task finished running, and the current timer value was be used to determine the task execution time: if the measured time was less than the BCET value stored for the task, then we assumed that a fault has occurred.

The challenge we face in a TTH or TTP design is that the MoniTTor described in Chapter 9 will not take into account the fact that the TTC tasks may be interrupted during their execution by TTH or TTP tasks. For example, Figure 97 shows Task A released by a TTC scheduler.

Figure 98 illustrates how the same task might be executed by a TTH scheduler (in which we assume there is a higher tick rate).

The task pre-emptions in the TTH design will (inevitably) increase the measured task execution time. If we attempt to use a standard TTC MoniTTor in a TTH design we are likely to trigger erroneous WCET exceptions.

If we wish to continue to monitor the task execution time in TTH designs (for example), then there are two main ways in which we can address this issue:

- We can increase the stored WCET and BCET values of TTC tasks to take into account the pre-emptions that may occur by TTH tasks. This is comparatively easy to do, because we know the period of the TTH interrupts, and we can determine the execution times of the corresponding ISR. Note that we can perform this calculation "off line" or even perform it during the system initialisation phase.
- We can adapt the MoniTTor so that the measurement of the execution time of a TTC task is "paused" whenever a TTH task executes.

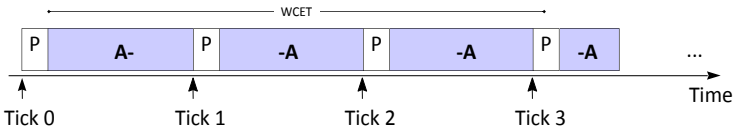


Figure 98: Task A from Figure 97, this time released by a TTH scheduler.

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

Both approaches can be extended easily to deal with TTP designs.

The second approach is more generally useful, because we can use a single timer to monitor the execution of all of the tasks in the system (TTC, TTH and TTP). We do this by "borrowing" the timer every time an interrupt occurs.

To understand how this approach works, please refer to Code Fragment 47. In this example, the MoniTTor will already be in use (in this case, measuring the execution time of Task A) when Tick 1 is generated. The function MONITTOR_I_Pause () will then be responsible for pausing the timer and storing the elapsed- time measurement (relating to Task A).

The same timer is then used to measure the execution time of Task P.

If Task P exceeds its pre-defined WCET then we would expect to take appropriate action (such as shutting down the system, or entering a Limp-Home Mode).

If Task P does not exceed its pre-defined WCET limit then – after Task P completes – the function MONITTOR_I_Stop () will be called. This will be used to check whether Task P has breached its BCET limit. If this limit was breached, then appropriate fault-handling would be carried out, including a change in the system mode.

If Task P completes without breaching either its WCET or BCET limits, then the function MONITTOR_I_Resume () will be called. This function will continue the monitoring of Task A (using the data that were stored earlier).

```
void SCH_Update(void)
   {
   // Pause monitoring for P0 task (if any)
   MONITTOR_I_Pause();

   // Call (and monitor) P1 task
   MONITTOR_I_Start(...);
   Call_P1_task();
   MONITTOR_I_Stop();

   // Resume monitoring P0 task (if any)
   MONITTOR_I_Resume();

   // Increment tick count
   Tick_count_G++;
   }
```

Code Fragment 47: Sharing a single timer to monitor the execution time of both TTC and TTH tasks.

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

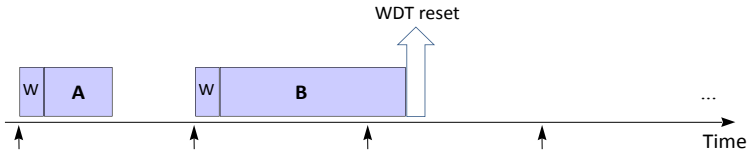


Figure 99: A WDT reset caused by a task overrun in a TTC design.

Where required, this process can be readily expanded for use in other, more general "time triggered pre-emptive" (TTP) designs that may employ other timer interrupts with different levels of priority.

A complete example using this mechanism in a TTP design is presented in TTRD11d.

11.12. Use of watchdog timers in TTH and TTP designs

As we first discussed in Chapter 2, a watchdog timer (WDT) can be used not only for general fault detection, but also as a simple means of detecting task overruns in a TTC design. To do this, we may choose to set the WDT to overflow after an interval slightly longer than the tick interval (Figure 99).

With such settings, a task overrun would delay the WDT refresh and could be used – for example - to initiate a transition to a Fail-Silent Mode.

As we discussed in Chapter 9 (in connection with TTC designs), using a WDT to detect task overruns provides a rather less flexible mechanism than a MoniTTor unit. One consequence of this is that - once we add a MoniTTor unit to the system - we will usually extend the WDT timeout settings, so that the WDT becomes a more general "safety net" mechanism, that will be triggered <u>only</u> if we are unable to handle faults with a more specific mechanism.

The above comments apply to TTC designs. Once we start working with TTH and TTP designs, the use of a WDT must again be considered carefully. In this case, we would <u>generally</u> recommend that WDT refreshes are carried out by a P0 task, or faults with such tasks may be "masked". Please note that this is best viewed a "design guideline" (rather a "design rule"), and may not be the most appropriate solution for cases in which the TTC tasks are very long.

[Where a long WDT period is employed, use of a MoniTTor unit will be particularly important.]

11.13. Conclusions

In this chapter we began to explore the design and implementation of TT systems that support task pre-emption.

In Chapter 12, we consider ways in which we can maximise temporal determinism in such designs: this will in turn allow us to explore techniques for monitoring task execution sequences in TTH and TTP designs.

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

CHAPTER 12: Maximising temporal determinism

Support for task pre-emption adds to the system complexity. In this chapter, we consider techniques that can help us to ensure that the system timing behaviour can be both modelled and monitored effectively.

Related TTRDs

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

C programming language (LPC1769 target):

- 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

12.1. Introduction

A three-stage development process for real-time TTC designs was described in Part Two of this book. The first stage involves modelling the system (using one or more Tick Lists, depending on the number of modes). The second stage involves building the system using a simple TTC scheduler. The third stage involves adding support for run-time monitoring.

In Chapter 11, we began to look at TT systems involving task pre-emption. In the present chapter, we will consider how we can model and monitor systems that employ TTH or TTP architectures.

12.2. Jitter levels in TTH designs (TTRD12a)

As we noted in Chapter 4, most TTC scheduler enter "idle" mode at the end of the Dispatcher function. Using TTRD05c, we explored the impact of idle mode of the scheduler. For ease of reference the results are repeated in Table 13.

As can be seen from these results, the use of idle mode has a measurable impact on the tick jitter (bringing it down to a level that cannot be measured using this approach).

In Chapter 11, we introduced a first TTH scheduler (TTRD11a). It is clear that the processor will not always be in idle mode when the tick is generated in this design: we would therefore expect to see measurable tick jitter.

Table 13: Measuring tick jitter in a TTC scheduler with (and without) idle mode.

Scheduler	Tick jitter (µs)
TTC scheduler without idle mode	0.18
TTC scheduler with idle mode	0

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

Table 14: Jitter measurements from a TTH scheduler.

Scheduler	Tick jitter (µs)
TTC scheduler without idle mode	0.18
TTC scheduler with idle mode	0
TTH scheduler (with idle & long TTC task)	0.46

TTRD12a implements an instrumented TTH scheduler: this allows us to assess the jitter levels that are obtained in a representative system.

Table 14 shows some results obtained from TTRD12a.

As expected, the results demonstrate a significant level of jitter from the TTH scheduler.

12.3. Reducing jitter in TTH designs (TTRD12b)

It is often possible to reduce jitter levels in a TTH design through the use of a "pre tick".

A pre-tick design typically involves generating a second (timer-driven) interrupt that is synchronised to the main scheduler tick, but shifted slightly out of phase. What this means in practice is that the pre-tick will take place just before the main system Tick. All that the pre-tick will then do is place the processor into idle mode. When the main system tick occurs, then – as in a TTC design – the processor will always be in idle mode. The end result is a reduction in tick jitter, at the cost of a small loss in CPU capacity.

The use of a pre-tick is illustrated in Figure 100.

There are different ways of implementing a pre-tick. In TTRD12b, a delayed "idle task" is used. This involves [i] setting up a timer to overrun at an interval less than the tick interval; [ii] starting this timer in the main scheduler ISR; [iii] setting up a second ISR (linked to the pre-tick timer) that will place the processor into idle mode.

Some measurements from TTRD12b are shown in Table 14 (note that the TTC figures in this table are repeated from earlier, for comparison).

As is clear from the table, the use of a pre-tick is able to bring down the level of release jitter to the same level as that seen in the TTC design in this example.

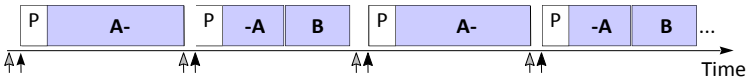


Figure 100: Use of a pre-tick to reduce jitter levels in a TTH scheduler. See text for details.

Table 15: The impact of a pre-tick on jitter levels in a TTH scheduler.

Scheduler	Tick jitter (µs)
TTC scheduler without idle mode	0.18
TTC scheduler with idle mode	0
TTH scheduler (with idle & long C tasks)	0.46
TTH scheduler (with idle & long C tasks) & pre-tick.	0

12.4. How to avoid PI in TT systems (Overview)

In Chapter 11, we argued that it is possible to avoid (not simply ameliorate) the impact of Priority Inversion (PI) in TT designs.

We have already argued (in Chapter 8) that we should try to allocate one task per resource in our TT designs (Figure 101). This approach can help to reduce the number of opportunities for conflicts over shared resources. On its own, however, such a measure will not allow us to completely avoid problems of PI in a TT design.

To achieve such a goal, there are two effective solutions: [i] code balancing; and [ii] a "Timed Resource Access" (TRA) protocol.[23]

Before we look at code balancing in connection with PI, we will first consider some wider advantages of such an approach in TTH/TTP designs.

12.5. Using code balancing with a PredicTTor unit

As we have seen throughout this book, appropriate use of a TT architecture allows a developer to ensure that some or all of the tasks in the system begin executing at precisely-determined points in time. However - in some real-time systems – care must still be taken when designing and implementing the tasks in order to ensure that all of the system timing requirements are met.

For example, we first introduced Code Fragment 48 in Chapter 6.

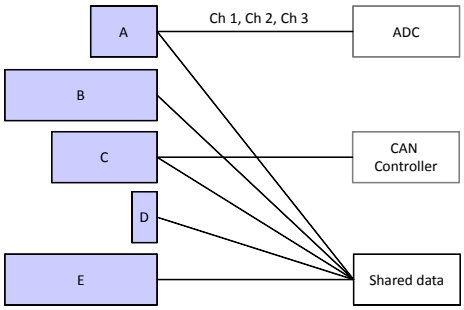


Figure 101: Implementing a "One task per resource" architecture.

[23] TRA protocols were introduced in Maaita (2008).

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

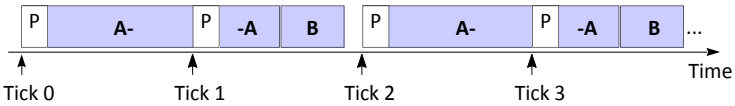


Figure 102: Executing tasks using a TTH scheduler. "A-" indicated the start of Task A: "-A" indicates the continuation of Task A (after pre-emption by Task P).

In this sample, the code within the task has been balanced. This means that we know when certain activities will occur (relative to the start of the task). For example, the task outputs will be generated in an interval starting A1 ms after the start of the task and finishing A2 ms later.

When we first considered Code Fragment 48, our focus was on TTC designs. As we begin to consider the modelling and monitoring of TT systems that support task pre-emption, the use of code- / execution-balancing offers additional (and very significant) benefits.

As an example, please refer to Figure 102.

We first saw a version of this figure in Chapter 11. It shows a set of two TTC tasks where Task A and / or Task B may be pre-empted periodically by the higher-priority (TTH task) Task P.

The complete task sequence for the task set shown in Figure 102 could be listed as follows: Task P, Task A, Task P, Task B, Task P, Task A, Task P, Task B.

If the code for all of these tasks is balanced, then this sequence will always be the same (every time this sequence of tasks runs).

If the code is not balanced, then we may – for example – obtain the task sequence shown in Figure 103 during the system execution.

In this case, the task sequence would be as follows: Task P, Task A, Task B, Task P, Task P, Task A, Task P, Task B.

```
void Task_A(void)
   {
   /* Task_A has a known WCET of A milliseconds */
   /* Task_A is balanced */

   // Read inputs (KNOWN AND FIXED DURATION)

   // Perform calculations (KNOWN AND FIXED DURATION)

   /* Starting at t = A1 ms
      (measured from the start of the task),
      for a period of A2 ms, generate outputs */

   /* Task_A completes after (A1+A2) milliseconds */
   }
```

Code Fragment 48: An example of a task with a balanced execution time.

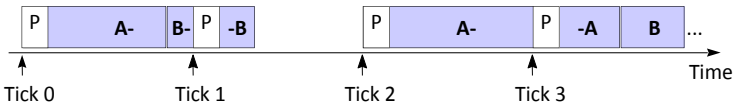


Figure 103: Executing "unbalanced" tasks using a TTH scheduler.

Without code balancing, this could be interpreted as an incorrect task sequence (which may not be what was intended).

The bottom line: it is often desirable to balance some (and possibly all) of the tasks in TT designs that support pre-emption.

12.6. Do you need to balance the code in <u>your</u> system?

It is easy to determine whether you may need to balance the code in your TT design.

To start with, we will assume that you are constructing a TTP system.

If we release only one (TTC) task in a tick, then our concern is whether the <u>difference</u> between the WCET and BCET of this task is greater than the (TTP) tick interval (note that the WCET may be 100 ms, even if the tick interval is – say – 100 µs in a TTP / TTH system).

If we also release one TTH task in this tick, we need to perform the same test for this task.

If we release more than one TTC task in a given tick interval, then we need to sum up the WCETs and BCETs of all of these tasks, and perform the "TTP tick interval" test on this sum.

Similarly, we need to sum up the WCET and BCET of all the TTH tasks and perform the TTP tick interval test on this sum too.

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

12.7. Using code balancing to prevent PI

We will now consider code balancing in the context of PI: to do so, please consider Figure 102 again, and assume that the code for Task A has been balanced.

Suppose that Task P and Task B need to share a resource. Because Task A has been balanced, then there should be no possibility that Task B will block Task P.

Suppose that Task P and Task A need to share a resource. In this case, there is a risk of conflict. However, we may be able to ensure that Task A always accesses this resource at Time 1 and Time 2 (see Figure 103).

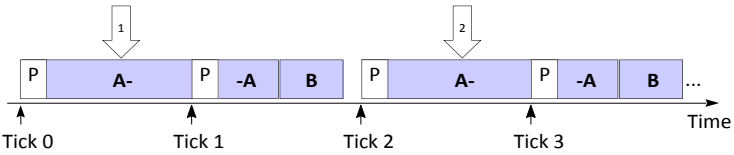


Figure 104: Controlling the time that a task accesses a shared resource may be possible in a TT design: this can ameliorate (or eliminate completely) the problems caused by PI in some designs.

Even if code balancing is employed, we suggest that a flag is set during the resource access (see Code Fragment 49), and checked in the timer ISR (Code Fragment 50): if the flag is found to be set, then this may be interpreted as a fault.

12.8. How to avoid PI in TT systems (TRA protocols)

Another way of dealing with PI challenges in a TT design is by means of a Timed Resource Access (TRA) protocol (Maaita, 2008).

To understand the operation of such a protocol, please consider that – in Figure 104 – we (again) wish to access a shared resource from Task B (a Priority 0 task).

For simplicity, we will assume that this resource access will take up to 5 µs.

Before Task B accesses the resource, it will – at run time - ask the scheduler for permission. The scheduler will either answer "Yes, go ahead" (because it knows that the next timer interrupt will occur more than 5 µs in the future) or it will say "No" (because it knows that the next timer interrupt is imminent).

As we demonstrate in TTRD12d, such a TRA protocol (or simply "TRAP") is very easy to implement. It also directly addresses PI concerns, because it ensures that – as required – the higher-priority task always has access to the resource when required.

```
   __disable_irq();
   ADC_flag_G = 1;  // Set resource flag
   __enable_irq();

   // Access resource

   __disable_irq();
   ADC_flag_G = 0;  // Clear resource flag
   __enable_irq();
```

Code Fragment 49: Setting a resource flag.

```
if (ADC_flag_G == 0)
   {
   // Execute task
   }
else
   {
   // Set fault variable, or
   // Count up number of clashes, or
   // Shut down system, etc
   }
```

Code Fragment 50: Checking a resource flag.

12.9. How to incorporate a TRAP in your design

There are two main ways in which a TRAP can be used effectively.

In some cases, we will not do any code balancing and – instead – simply rely on the protocol to deal with any resource conflicts.

Where precise timing behaviour is of greater concern, we will balance the code, but still use a TRA protocol: in these circumstances, a situation in which the TRAP finds that a requested resource access is refused may be interpreted as a fault, and we may – for example – want to record this fault, or allow only X faults per hour, or trigger a shutdown immediately.

Whatever design option is chosen, we have the ability to address all PI problems.

12.10. A complete TTP design (TTRD12d)

Building on TTRD11d and the techniques presented in this chapter, TTRD12d presents a complete example of a TTP design.

TTRD12d incorporates both MoniTTor and PredicTTor mechanisms and illustrates use of a TRA protocol.

12.11. Conclusions

In this chapter, we've looked more closely at the development of reliable TT designs that support task pre-emption.

In Part Four, we consider techniques for completing our system.

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

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

PART FOUR: COMPLETING THE SYSTEM

"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

Tony Hoare, 1980

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

CHAPTER 13: From Task Contracts to System Contracts

In Chapter 8, we began to explore the use of Task Contracts. In this chapter, we turn our attention to System Contracts.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD13a: TTC scheduler with temperature monitoring

13.1. Introduction

In Chapter 8, we argued that development of a Task Contract (TC) should involve the following processes:

- Drawing up a specification for the task.
- Considering any pre-conditions (PreCs) that must apply before the task will be able to meet this specification, and – where sensible and appropriate – incorporating mechanisms in the task code to check for these PreCs.
- Considering any ways in which the task's environment might change during its operation (after the PreCs have been checked, and in such a way that would be likely to prevent the task from meeting its specification) and – where sensible and appropriate – incorporating mechanisms in the task code to check for any such changes.
- Considering any ways in which it will be possible to confirm that the task has met its required specification when it completes and – where sensible and appropriate – incorporating mechanisms in the task code to check for such a successful completion: these are the Post Conditions (PostCs).

The intention is that TCs should support the process of designing and implementing robust tasks. In practice, this may mean simply that the above notes are used formalised into a "check list" that is employed during design and code reviews.

In this chapter, we argue that it can be helpful to think about the whole system in a similar way, using "System Contracts".

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

13.2. What is a "System Contract"?

As far as we are concerned in the present book, the development of a System Contract involves the following processes:

- Drawing up a specification for the system.
- Considering any conditions that must be met at startup before the system will be able to meet its specification, and – where sensible and appropriate – incorporating "Power-On Self Test" (POST) mechanisms in the system code to confirm that these conditions have been met.
- Considering any ways in which the system's environment might change during its operation (after the POSTs have been applied, and in such a way that may prevent the system from meeting its specification) and – where sensible and appropriate – incorporating "Built-In Self Test" (BIST) mechanisms in the system code to check for any such changes.
- Considering what modes the system will operate in, and the task set that will be employed in each mode. This will include consideration of both normal operating modes and fault-related modes.
- Considering whether any of the above system modes should employ backup tasks, and – if required – designing these tasks.
- Considering ways in which the system will be shut down, both under normal conditions (for example, by a user) and when a fault is detected (for example, under the control of the system itself), and any reporting that needs to be carried out at this time.

We will explore some of the issues that are raised by the above list during the course of this chapter.

13.3. Generic POST operations

The acronym "POST" is often used to describe a suite of low-level tests that need to be performed before the system starts to operate. The acronym is usually expanded as "Power-On Self Tests", or – sometimes – as "Pre Operation Self Tests".

The key question that we are trying to answer through the use of such tests is whether we have an operational computer system to work with. If the answer is "no", then all we can do is try to "fail as safely as possible".

In most cases (as both of the acronym expansions make clear) we generally need to use the computer system itself to try and answer this question.

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

In order to determine whether the computer is operating correctly, we will typically aim to investigate some or all of the following elements during POST tests:

- The CPU registers;
- The Program Counter (PC);
- The RAM;
- The ROM / Flash memory;
- The clock frequency;
- The interrupt operation;

Most of the items in the above list involve testing memory. This is not straightforward, not least because: [i] available memory is growing rapidly; [ii] test durations tend to grow (sometimes exponentially) with memory size, and [iii] as memory feature sizes shrink, devices are becomes more sensitive to some forms of external interference (such a single-event effects).

When working with read-only memory (e.g. Flash) a "signature" is usually calculated from the memory contents, and stored before the system is executed. Before (and during) the system execution, we can compare the correct (sometimes called "golden") signature with the results of a calculation made on the current memory contents. Any differences indicate that something is amiss.

When it comes to volatile memory (e.g. general-purpose RAM and registers), the required testing regime is far from trivial. Tests will typically start by determining whether the memory cell under test is "stuck" at a particular value. Tests will also be made for coupling between memory cells: for example, if we change the value of (only) one bit in memory, this should not have any impact on any other memory cells in the device. Transition faults may also be tested: for example, under some fault conditions the value stored in a cell may be able to change from 1 to 0 but not from 0 to 1.

To test volatile memory, some form of "marching" algorithm is commonly employed: such tests originated with the work of Suk and Reddy (1981). Since the original tests were introduced, a number of variations have been created. The involve tests involve combinations of write operations followed by read operations that are carried out by moving ("marching") up and down the memory area that is under test.

To test that the system clock frequency is correct, we need some form of independent clock against which we can perform a comparison. Where designs have both an external (usually crystal) oscillator and an internal (often RC) oscillator, it may be possible to use these to perform some

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

checks. Alternatively, where designs have access to a mains (AC) power supply running at 50 Hz / 60 Hz, this may be used to check the system oscillator.

Checking that interrupts are operating correctly may be challenging in an ET design (in which there may be multiple interrupts, that are expected to trigger at "random" times). In a TT design, such tests are comparatively straightforward.

13.4. Example: POST operations that meet IEC 60335 requirements

NXP Application Note AN10918 describes in detail how to perform POST operations that are in compliance with IEC 60335 on an LPC1769 microcontroller. The Application Note is accompanied by a complete code library.

Table 16 summarises the tests that are supported by this library.

Table 16: POST operations supported by NXP Application Note AN10918.

Test	Component	Fault/error	Supported?
1.1.	CPU registers	Stuck at	Yes
1.3.	Program counter	Stuck at	Yes
2.	Interrupt handling and execution	No interrupt or too frequent interrupt	Yes
3.	Clock	Wrong frequency (for quartz synchronized clock: harmonics/ subharmonics only)	Yes
4.1.	Invariable memory	All single bit faults	Yes
4.2.	Variable memory	DC Fault	Yes
4.3.	Addressing (relevant to variable and invariable memory)	Stuck at	Yes
5.1.[24]	Internal data path	Stuck at	No
5.2.[19]	Addressing	Wrong address	No
6.	External communications	Hamming distance 3	No
6.3.	Timing	Wrong point in time and sequence	No
7.[25]	Input/output periphery	Fault conditions specified in H.27	No
7.2.1.[20]	A/D and D/A converters	Fault conditions specified in H.27	No
7.2.2.[20]	Analog multiplexer	Wrong addressing	No

[24] Only applicable when using external memory.
[25] A production plausibility check is recommended.

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

Figure 105: An example of a medical infusion pump (an external pump).
Licensed from: Dreamstime.com. Photographer: Abubjsm.

13.5. Checking the system configuration

The low-level POST operations that were considered in Section 13.3 and Section 13.4 are intended to determine whether the processor is operational. In many designs, the next key test involves checking that the system configuration is correct.

Checking the system configuration is a key concern, particularly in safety-related systems. For example, suppose that we have developed some software to control a medical infusion pump (Figure 105).

Such a pump is used to deliver fluids into a patient's body in a controlled manner. Different forms of infusion pump may, for example, be used to deliver food or medication (such as antibiotics or chemotherapy drugs).

It is clearly essential that such a device operates correctly, and appropriate development procedures (in compliance with IEC 62304 and perhaps IEC 60601) would be followed to create the required control software.

Let us suppose that such procedures have been followed during the development process. Let us also assume that, over time, our organisation has developed a range of different pumps, each serving a slightly different purpose. It may now be possible that the software intended for one pump (let's call it Model 2a) might be installed by mistake in another pump (let's call it Model 2b), with the consequence that the latter product does not operate quite as expected.

The problem raised in this example – a potential mismatch between the system software and hardware - is a significant one in many sectors, not least in situations where in-field upgrades are performed on complex devices.

In addition, it should be appreciated that software changes may not always result accidentally. For example, in some parts of the world, "chipping" of passenger cars has become an option. This involves changing the system configuration in the main vehicle ECU, usually in an attempt to improve the car performance. If carried out competently, the changes may do little damage: however, in other circumstances, and it is possible that the vehicle may become unreliable or even dangerous. Most car manufacturers would – therefore – wish to detect if any such changes have been made to one of their vehicles.

We may also need to consider two further possibilities. The first is that attempts might be made to make deliberate changes to the configuration of (say) a military system during or before the system is used. The second is that through the impact of environmental factors (as as radiation or EMI) the software configuration may be changed at some point during the system life.

To address any / all such challenges, we generally wish to check the system configuration: [i] as part of the POST operations, and [ii] as part of the periodic system checks.

In both cases, this will involve confirming that the program / data stored in Flash (and perhaps in other configuration files) is as expected.

As we noted above, the correct contents of any read-only memory can be coded as a "golden signature": this can be stored before the system is executed. Before (and during) the system execution, we can compare the golden signature with the results of a calculation made on the current memory contents.

13.6. Example: Check the system configuration

If we want our system to start quickly, and we want to be able to perform frequent tests of the system configuration at run time, we need to be able to perform checks on the system configuration very quickly.

On the LPC1769, hardware support is provided in order to speed up this process: this takes the form of the Multiple-Input Signature Register (MISR). The MISR can produce a 128-bit signature from the contents of a specified range of Flash-memory addresses. If we create a signature from the memory contents at design time (sometimes called a "golden signature"), we can then: [i] use the MISR hardware to significantly speed up the process of creating signatures at run time based on the current Flash contents; and [ii] compare the run-time and golden signatures in order to determine whether the memory contents have been altered.

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

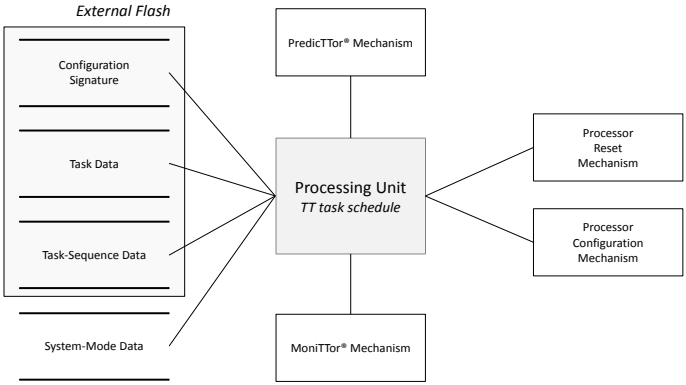


Figure 106: An example of a processor platform (TT04) that employs an external Flash memory device to store a "golden" signature, representing the required program code. See text for details.

Note that the MISR hardware alone may not be enough. We may require an independent (non-volatile) memory device in which we can store the signature. One option in the designs that we are considering in this book is to store task-sequence data and – possibly – task timing data in an external device, such as an external memory device: such a device may also be the logical place to store the configuration signature (see Figure 106). Note that – in some designs – the external memory device is likely to be write protected.

We will say more about this system platform in Chapter 14.

13.7. Generic periodic checks (BISTs)

Assuming that we have passed the POSTs, and confirmed that we are using the correct system configuration, we will generally also need to perform periodic "Built-In Self Test" (BISTs) to ensure that there have been no changes to the system's environment while the device was operating.

It is tempting to assume that the BISTs will be the same as the POSTs, but there are usually some differences. In particular, as we noted in Section 13.4, many of the POSTs involve memory checks. At startup, we can run "destructive" tests at this time for most areas of memory: in other words, we can accept that the tests will alter the memory contents. While the system is running, we usually need to ensure that tests are non-destructive.

13.8. Example: BISTs in compliance with IEC 60335

NXP Application Note AN10918 (and its associated code library) demonstrates how to perform BIST operations that are in compliance with IEC 60335, on an LPC1769 microcontroller.

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

13.9. Additional periodic tests

Various other tests may also need to be carried out in order to ensure that the system can continue to operate correctly.

For example, suppose that the maximum operating temperature of the processor is 125 ^{o}C. We would view this limit as an important part of the System Contract: specifically, we'd have "Processor temperature is <= 125 ^{o}C" as one of the conditions that must be met before the system can be expected to operate correctly, and we would expect to have to monitor this temperature during the system operation.

Another common requirement is that vibration should be monitored.

13.10. Example: Monitoring CPU temperature (TTRD13a)

The operating temperature range for the LPC1769 microcontroller is -40 ^{o}C to 85 ^{o}C.

The EA Baseboard (see Appendix 1) incorporates a MAX6576 temperature sensor. This device produces a square wave with a period proportional to absolute temperature (Kelvin). It has an operating temperature range of -40 ^{o}C to 85 ^{o}C.

TTRD13a demonstrates how the MAX6576 sensor can be used to ensure that the microcontroller is always operated within the required range.[26]

Note: in this example, the other components on the LPCXpresso board and the EA Baseboard would – clearly - need to be checked carefully in order to ensure that they too could operate at the required temperature range.

13.11. System modes

Another important consideration when drawing up the System Contract is to consider the possible system operating modes.

Please refer to Chapter 7 for further discussion about this matter.

13.12. Tasks and backup tasks

In Chapter 8, we explored mechanisms for switching to backup tasks. At the time we did not discuss the design of such tasks: we do so here (in the context of a discussion about System Contracts), for reasons that we explain in this section.

To support this discussion, we will assume that "Task X" in a particular system has failed at run time (that is, while the system is operating).

[26] TTRD15a incorporates an alternative version of this temperature-sensor interface.

We'll also assume – again for the purposes for the purposes of discussion – that we've detected the problem with Task X because it has overrun: that is, its execution time on a particular run has exceeded its predicted "Worst Case Execution Time" (WCET).

Task overruns can occur for a number of reasons. For example, we have previously considered the possibility that a task that has been designed to read from an analogue-to-digital converter (ADC) may "hang" while waiting for a conversion to complete if the ADC has been damaged. This is – of course – the kind of behaviour that can be handled locally (for example, the task should check that the ADC has been configured correctly before using it, and should incorporate a suitable timeout mechanism whenever it takes a reading). We'll assume that Task X has been designed to deal with such problems and has been carefully reviewed: our task overrun problem therefore indicates that a significant (unknown) fault has occurred.

In some designs, the overrun of Task X (in these circumstances) might be reason enough to shut down the system as cleanly as possible. However, not all systems can be shut down immediately in this way: for example, if our task is part of a safety-related automotive system and the vehicle is currently travelling at speed down a motorway, then simply "shutting down" is an option that we would generally wish to avoid.

For similar reasons, we will assume that it is not possible to simply terminate Task X (and have the system carry on without this task).

In these circumstances, we may wish to "switch in" a backup task ("Backup Task X"). This backup task will – we assume – replace Task X in the schedule: indeed, our aim is that – the next time that Task X was due to run, Backup Task X will run in its place. Depending on the system requirements, we may then wish to continue running with Backup Task X indefinitely, or we may wish to signal a fault (and – for example – encourage the driver to stop the vehicle as soon as it is safe to do so).

The design and implementation of Backup Task X must be carried out with care.

Clearly, we'd like to try and ensure that – whatever the underlying faults that gave rise to the failure of Task X – Backup Task X will not suffer from the same problems. The challenge is that we presumably created Task X to be reliable, using appropriate design and implementation approaches. In particular, we will – presumably – have designed Task X in such a way that we didn't expect it to suffer from any faults. So, how can we try to ensure that Backup Task X does not suffer from the same problems as Task X when we don't know what the problem is that we are trying to avoid?

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

One way in which we might try to achieve this is through "N-Version programming". This will involve creating a single specification for the system task, and passing this to two independent teams (for example, we might have one team in Europe and another team in Singapore). Each team is expected to produce a version of the task to the best of their ability.

The underlying assumption is not that either task will be perfect. Instead, what we are trying to do is find a way of creating two tasks that have different faults (that is, statistically uncorrelated faults).

In the mid 1980s, John Knight and Nancy Leveson set out to explore this issue in a much-cited study (Knight and Leveson, 1985; Knight and Leveson, 1986):

> "In order to test for statistical independence, we designed and executed an experiment in which 27 versions of a program were prepared independently from the same requirements specification by graduate and senior undergraduate students at two universities. The students tested the programs themselves, but each program was subjected to an acceptance procedure for the experiment consisting of 200 typical inputs. Operational usage of the programs was simulated by executing them on one million inputs that were generated according to a realistic operational profile for the application. Using a statistical hypothesis test, we concluded that the assumption of independence of failures did not hold for our programs and, therefore, that reliability improvement predictions using models based on this assumption may be unrealistically optimistic."
>
> [Knight and Leveson, 1991]

The Knight and Leveson study has proved influential and it now widely accepted that N-Version programming is not an effective solution to the problem of creating two "equivalent" tasks that can be expected to fail in different ways.

So – if we return to our original problem – how should we produce our backup task?

The answer in the general case is that – as with all of the other tasks in the system – we should start with an appropriate specification and produce a new task to the best of our ability. However, rather than do so in a "Black Box" way – allowing a separate team to create this task – we should work in a "White Box" way, and focus on trying to minimise the opportunities for common-cause failures. This may mean – for example – trying to ensure that we employ different hardware resources and different algorithms in the two tasks. We also know that coding errors are much less likely in short pieces of code and we may – therefore – aim to keep the backup task very simple (in the expectation that this will be likely to improve reliability even

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

if it results in reduced quality or performance). This is sometimes known as a "limp home" task.

Working in this general way may be our only option, if we don't have any knowledge about the faults that have caused our original task to fail. However, if we can obtain information from the original task about the problem that caused it to fail, this may allow us to execute a more appropriate backup option. For example if – as suggested above – our task fails because of an ADC problem (and we are aware of this), we may be able to run a replacement task that uses a different ADC.

13.13. Example: Design of a backup task for analogue outputs

When designing backup tasks, the suggested starting point is to begin by ensuring that the hardware used in the two task designs is "as different as possible".

For example, suppose that we have to produce a backup task for a design that generates analogue outputs. If our original task employed a digital-to-analogue converter, we might decide to base the backup tasks on a Pulse-Width-Modulation (PWM) output device.

Note that it is not always possible to find replacement hardware that meets these requirements. In these circumstances, it may be more appropriate to consider alternative design options. This might mean changing the system mode (rather than replacing a task). In this situation, we would probably

13.14. Shutting the system down

When we are working with Task Contracts, we have to consider how we can confirm that a given task has met its required specification when it completes and (where possible) try to incorporate mechanisms in the code to check for such a successful completion: these are the Post Conditions.

When we are considering System Contracts, the equivalent of a task completion is the process of shutting the system down.

In the first instance, we need to think about a "normal" shut down, such as when the user presses the "Off" switch on a machine or removes an ignition key from a vehicle.

We also need to consider reasonably foreseeable misuse. For example, we need to consider what the implications might be for a piece of industrial equipment if the user avoids a lengthy shut-down procedure at the end of a shift by simply switching off the power.

In addition – as we have noted in many previous examples - the system is likely to have at least one "fault" mode. This needs to be handled appropriately and – probably – reported to the user.

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

Figure 107: An example of a simulation framework used to perform initial tests on an automotive control system. Photo by Peter Vidler.

In the presence of some faults, it may be felt to be necessary to move the system into Limp-Home Mode, and maintain it there until the device is switched off. At this point, it may be felt to be appropriate to prevent the system from being started / activated again, until it has been reviewed by a suitably-qualified individual (either in person, or – in some circumstances – via a remote diagnostics system). We give examples of such a design in Chapter 15 (TTRD15a).

You may also recall that we considered some of the challenges presented by such a scenario in Chapter 7. As we discussed, we may leave the user – or others - at significant risk if it is not possible to restart the system after a fault occurs. For example, leaving a vehicle stranded on a "level crossing" or in the fast lane on a motorway are examples of scenarios that need to be addressed.

One effective solution to such problems is to provide a Limp-Home Mode in which it is still possible to move the vehicle, but only up to a speed of (say) 10 miles per hour (~15 kilometres per hour). This will allow the user to move the vehicle to a safe location, but will not allow "normal" operation. Please note that it would again be important to ensure that the user could not reset any error codes (and therefore move out of the Limp-Home mode) simply by cycling the power: TTRD15a illustrates one way of achieving this.

13.15. Performing initial system tests

The architectures of many safety-related embedded systems are complicated. When engineering such systems, we need to consider the possible system modes and states. It often seems that much of the processing involves: [i] checking the system condition (for example, engine vibration, CPU temperature, the time that the system has been in a particular

state and so on), and [ii] implementing appropriate state- or mode-transitions in the event that faults are detected.

Suppose that we are developing an automotive control system, to be used as part of an autonomous vehicle. We cannot simply "wire up a car" and take a prototype onto a public road (or even onto a test track) until we have ensured that the core architecture and implementation is correct, because: [i] to do so may risk the life of a test driver; and [ii] the inevitable iterations in the development and test process would be both slow and expensive.

Before most automotive systems are ready for testing on a track, some form of simulation will usually be employed. For example, Figure 107 shows a test framework that was created in order to evaluate an adaptive cruise-control design for a passenger car. The framework simulates an infinitely-long motorway, and allows various faults to be injected in the different processor nodes (and the communication network) in order to test the system behavior. In such a framework different software architectures and code implementations can be tried and compared quickly and safely.

Overall, it is difficult to test any form of control or monitoring system unless we have a "target" available (that is, something to control or monitor). As a result, a "hardware in the loop" test framework is often a key development tool for organisations that create safety-related systems in sectors such as aerospace, medical instruments and elevator control.

Even in sectors where the risks are comparatively low, initial simulations are often employed in order to speed up the development process and support initial system testing: we will give an example of such a framework in Chapter 15.

13.16. International standards

The techniques explored in this chapter can be used to support the development of devices that are compliant with various international standards and guidelines.

For example, IEC 61508-7 requires that the system will need to check its own configuration when it is started up (IEC 61508-7 C2.5), and then carry out periodic checks of safety functions (what the standard calls "proof tests": 61508-4, Section 3.8.5) while it is operating.

IEC 61508-7 (A.10.1) suggests that a temperature should be monitored at critical points in the system, in order to ensure that the system is not expected to operate outside its specified temperature range.

Similarly, ISO 26262-5, Section 7.4.2.3, requires that the operating conditions of the hardware complies with the specification of their environmental and operational limits.

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

13.17. Conclusions

In this chapter, we've introduced the idea of System Contracts.

In Chapter 14, we will consider three recommended platforms for reliable TT embedded systems.

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

CHAPTER 14: TT platforms

In this chapter, we build on the material introduced throughout the book and present a suite of three recommended system platforms for reliable embedded systems that employ a TT architecture.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD14a: System Platform TT03
- TTRD14b: System Platform TT04
- TTRD14c: System Platform TT05

14.1. Introduction

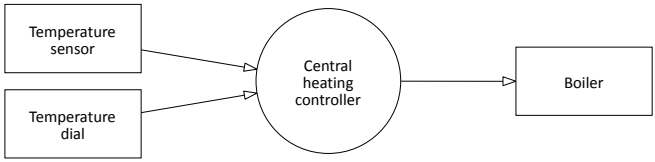
Building on the material presented during the course of this book, three flexible hardware / software platforms for TT embedded systems can be identified. We will describe these platforms in this chapter.

To place these discussions in context, we also consider three further TT architectures, beginning with Platform TT00.

14.2. Platform TT00

Platform TT00 takes the form of a "Super Loop" (Figure 108).

Unlike the platforms employed throughout most of this book, a Super Loop requires no interrupts. It consumes minimal resources.




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

   while(1) // 'for ever' (Super Loop)
      {
      C_HEAT_Get_Required_Temperature(); // Get input from the user
      C_HEAT_Get_Actual_Temperature();   // Get current room temperature
      C_HEAT_Control_Boiler();           // Adjust the gas burner
      }

   return 1;
   }
```

Figure 108: Platform TT00 (example, based on a "Super Loop").

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

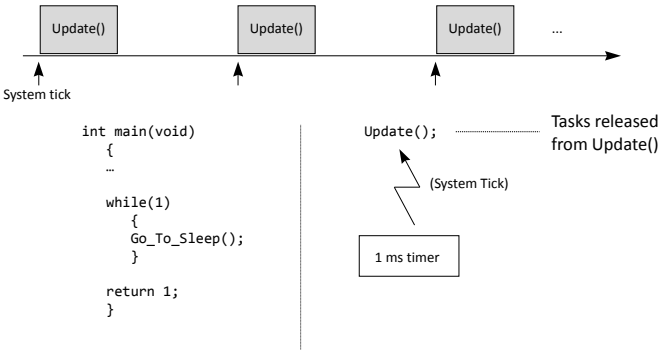


Figure 109: Platform TT01 (example, based on a timer ISR).

Timing behaviour can be controlled in T00 (to an extent) through the use of Sandwich Delays. Even with such delays in place, the design tends to be rather "fragile", in that changes to one task may have significant side effects across the system as a whole.

When employed in the main system processor(s), TT00 is only generally suitable for rather simple designs (typically involving fixed sequences of a small number of tasks): we gave an example of such use in Chapter 6. However, TT00 can provide an effective foundation for WarranTTor Units (see, for example, TTRD14c2).

14.3. Platform TT01

Platform TT01 employs an "interrupt service routine" (ISR) that is triggered by periodic timer "ticks" (Figure 109). Unlike the designs discussed elsewhere in this book, there is no separate "Dispatcher" function: instead, all tasks are called directly from the timer ISR.

Many existing designs are based on a form of TT01. However, unless some form of MoniTTor unit is incorporated, this platform is not generally suitable for use in high-reliability or safety-related systems, because faults (such as those that result in "task overruns") can significantly disrupt the system operation.

Our main use of TT01 is as an introductory / teaching architecture: the platform is explored in detail in "Embedded C" (Pont, 2002).

For new designs, we recommend that TT02 or (preferably) TT03 is considered as an alternative to TT01.

14.4. Platform TT02

In Platform TT02, the process of keeping track of time is separated from the process of dispatching tasks (Figure 110).

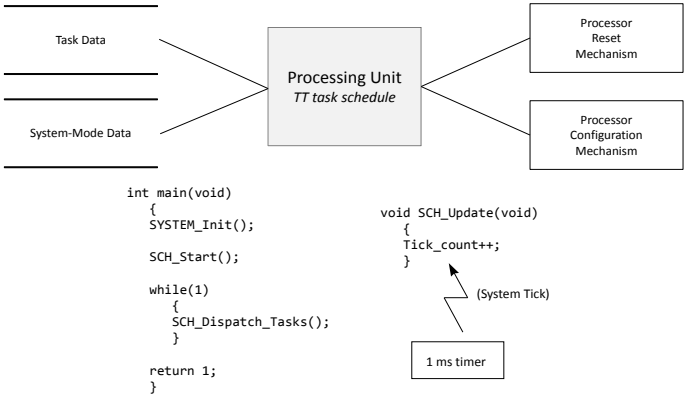


Figure 110: Platform TT02 (example of TTC implementation).

In TT02, the system will employ at least two operating modes (a Normal Mode and a Fail-Silent Mode), and will usually have further modes.

We will reset the processor between modes. The System-Mode Data (typically implemented using NVRAM) and Processor Configuration Mechanism will be used to set up each mode.

The processor will execute a TT scheduler (TTC, TTH or TTP). This means that - in each mode – the system will execute a sequence of tasks according to a pre-determined schedule. Note that the task "sequence" may be very simple in fault modes: for example, it is common to run a single task in a "Super Loop" in a Fail-Silent Mode.

Platform TT02 was explored in detail in Chapter 2. The later platforms all build on this foundation.

14.5. Platform TT03

Our first recommended system platform was originally introduced at the end of Part Two: we refer to this as Platform TT03 (Figure 111).

As with TT01 and TT02, Platform TT03 is based on a single processor unit.

We assume by this stage in the book that Platform TT03 is being incorporated in a three-stage development process. The first stage will involve modelling the system (using one or more Tick Lists, depending on the number of modes), as introduced in Chapter 4. The second stage will involves building the core system (for example, using a TTC scheduler, as detailed in Chapter 2 and Chapter 8, or a TTH or TTP scheduler, as detailed in Chapter 11 and Chapter 12). The third stage will involve adding support for run-time monitoring: in many safety-related designs, we find that some 50% of the system code is devoted to such monitoring activities.

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

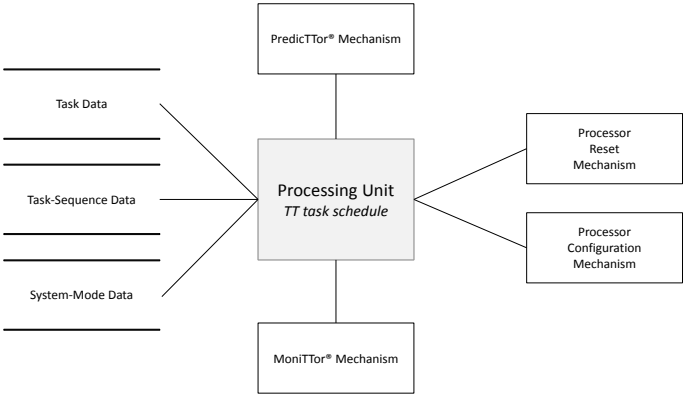


Figure 111: Our first complete platform for TT embedded systems: TT03 (overview).

During the course of this book we have described run-time mechanisms for detecting and handling both resource-related and execution-time faults (see Figure 112). In doing so, our aim was to demonstrate that:

- By means of a PredicTTor mechanism, we can ensure that the tasks are released in the correct sequence at run time. This is a particularly powerful approach because we can check each task before it is released: in many cases, this will – for example – allow us to shut the system down before it enters an unsafe state.
- By means of a MoniTTor mechanism, we can measure the execution time of each task (every time it is released) and compare the measured value with pre-determined upper (WCET) and lower (BCET) limits: if a task breaches a limit, we can take appropriate action (which may – for example – include replacing the task with a backup take, or shutting down the system).
- By checking the configuration settings of hardware components every time we use them in our tasks, we can detect that the resources required by a given task have been altered, reconfigured or damaged by another source.
- By maintaining "inverted copies" of variables that are used to record a task state and / or used to pass information to another task, we can further increase our chances of detecting situations in which a given task has not been able to operate with the required level of "Freedom From Interference".
- By making appropriate use of the memory-protection unit (MPU) in our target processor – where available - we can increase the chances of detecting activity by one or more tasks that might have an impact on the operation of the scheduler.

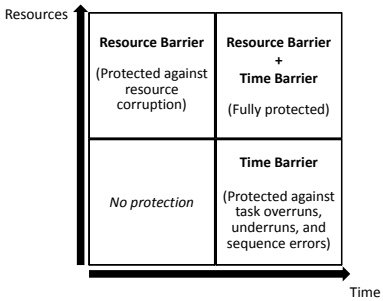


Figure 112: Run-time monitoring is employed to guard against Uncontrolled System Failures in all three of the platforms described in this chapter. We check for faults that are evidenced by incorrect CPU usage (task overruns, task underruns and task sequencing) as well as faults related to other hardware resources (such as CAN controllers, analogue-to-digital convertors, port pins and memory).

Overall, through: [i] use of a TTC, TTH or TTP scheduler; [ii] due care with the task design and verification; and [iii] use of Resource Barriers plus MoniTTor and PredicTTor mechanisms at run time, we can be confident that a system based on TT03 will operate according to its specification.

14.6. Platform TT04

In the system designs presented in this chapter, we assume that the implementation will be based on one or more microcontrollers (MCUs). In the case of TT03, we further assume that the memory employed in the design will be "on chip". In some cases, it can be useful to work with off-chip memory: we have found it useful to place such designs in a different category, referred to here as "Platform TT04".

There are a number of reasons for using external memory:

- We may require an area of nonvolatile memory to store information during system mode changes.
- We may wish to use an external memory device can be used to store the task-sequence data, to reduce the risk of corruption in designs where the MCU employed does not have a Memory Protection Unit.
- We may wish to use an external memory device to store the configuration signature for the system (allowing the system to check that the program code has not been altered at run time, as discussed in Chapter 13).
- We may also wish to use the external memory area to store other information about the tasks (such as execution time data and resource configuration data, as discussed in Chapter 13).
- In some designs, we may wish to have memory available as a "black box" (in order to retrieve key data in the event of an accident).

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

When we wish to store information – such as task-sequence data – before the system executes, Flash or EEPROM may be an effective choice. Such memory devices can usually be physically "write protected" once they are in place, making it very difficult for the processor to alter the contents.

When we need to store information during the system run, Flash or EEPROM memory can still be considered but – where data will be stored frequently – we may exceed the maximum number of write cycles for this technology rather quickly. The traditional alternative to Flash / EEPROM in such circumstances is a battery-backed RAM device with a serial (e.g. SPI) interface. Such a solution can be effective (not least during system prototyping): however, as it relies on a battery, this solution may give rise to reliability concerns if it is to be used in <u>some</u> products.

One effective modern alternative is "MRAM". This provides non-volatile storage without requiring a battery, and without a practical limit on the number of write cycles. MRAM is available for use at high temperatures. This technology is also (inherently) radiation tolerant (which also makes it attractive in space, aerospace and civil-nuclear applications).

Two possible implementations of Platform TT04 is illustrated in Figure 113 and Figure 114.

Please note that it is not – of course – necessary to store all data in the same memory device (indeed, in some designs, we may wish to duplicate data across more than one device).

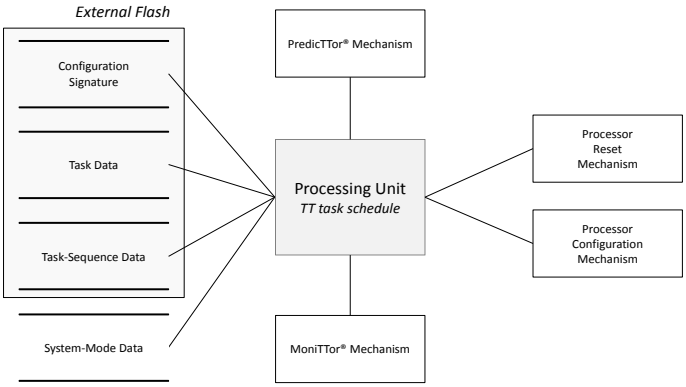


Figure 113: An example of an implementation of our second complete platform for TT embedded systems (TT04).

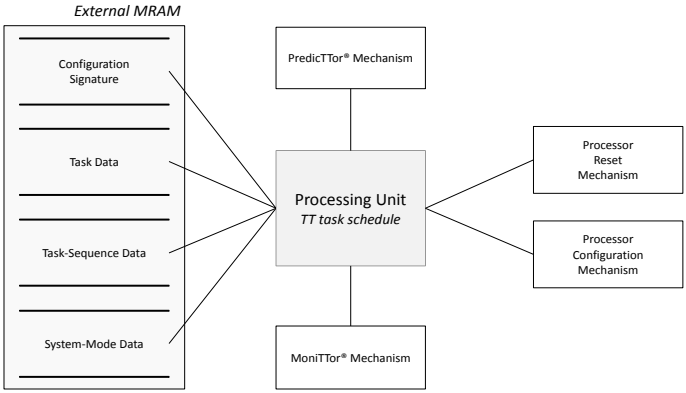


Figure 114: Another example of an implementation of Platform TT04.

14.7. Platform TT05

All of the designs that we have considered so far in this chapter have involved a single processing unit. Using TT03 or TT04, we can make this into a very robust platform, but it will still not be suitable for all applications.

One way of thinking about this issue is to consider what would happen to your system if this single processor failed suddenly, in an active (rather than fail-silent) state.

In some cases, the user may be able to solve the problem by cycling the power, or pressing a "Reset" switch. However - even if the underlying problem is transitory - a manual reset may not a practical option for systems that are remote (such as autonomous vehicles, deep-sea systems and satellites) or those with rapid response requirements (such as many industrial systems and other forms of control / monitoring designs). More generally, it is becoming increasingly recognised that relying on human operators to intervene in the event of system failure may not be appropriate, not least in safety-critical systems.

If the implications of a possible processor failure are serious, then we usually require some form of additional processing device in the system: that is, some form of backup processor that can "step in" if the main processor fails to operate (or fails to operate correctly).

We can achieve this is with Platform TT05: Figure 115 shows a typical implementation.

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

When considering Platform TT05, please note the following:

- The Control Unit is based on TT03 / TT04.
- The system also incorporates a WarranTTor Unit.
- Both the Control Unit and WarranTTor Unit contain MCUs (or other processing elements): see Section 14.10 – Section 14.12 for further discussion about these devices.
- The WarranTTor Unit is intended to deal with the impact of serious failures in the Control Unit, in situations where the Control Unit cannot deal with these failures itself.
- In a TT05 design, the WarranTTor Unit is not usually intended to reproduce or replace the normal operations of the Control Unit: instead, it is designed to implement "safe system shutdown" behaviour (only).
- It is only when the WarranTTor Unit detects that the Control Unit is not operating according to the specification that it will act. In these circumstances, the WarranTTor Unit may (for example): [i] trigger a safe shut-down sequence on the system that is being controlled; and / or [ii] reset - or remove power from - the Control Unit; and / or [iii] issue a "cry for help" (by sounding an alarm or sending appropriate radio messages); and / or [iv] record information in a "black box".
- In order to operate correctly, the WarranTTor Unit requires information about the system task sequence and system mode. In practice, this means that the Control Unit will usually send a message to the WarranTTor Unit before it executes each task and when it changes mode: if the WarranTTor Unit is measuring task execution times, then the Control Unit may also send a message when each task completes.
- Because it receives the above messages, the WarranTTor Unit always knows precisely what state the Control Unit is in: this allows it to take appropriate action in the event that a fault is detected during the system operation.
- Two-way communication between the Control Unit and the WarranTTor Unit is not always required: instead, the Control Unit may simply send a message ("About to run Task 35"), and then carry on, without expecting or requiring a reply. If the behaviour is correct, the WarranTTor Unit will do nothing: if the task sequence is wrong, action may be taken (Figure 116).

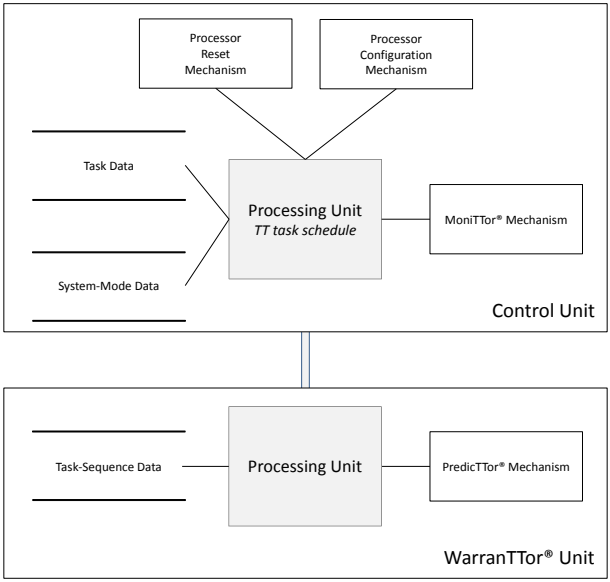


Figure 115: An example of our third complete platform for TT embedded systems (TT05).

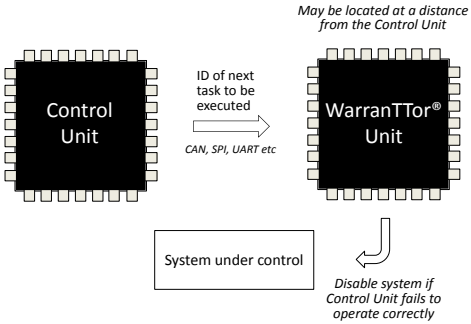


Figure 116: If it detects a problem, the WarranTTor Unit may take direct action to move the target system into a safe state.

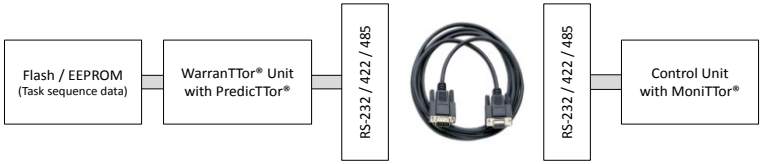


Figure 117: One possible configuration for TT05, where the two processor units are physically separated (possibly by several metres or more) in order to reduce the opportunities for common-cause failures.

14.8. Design example: Avoiding common-cause failures

Whenever we have any form of redundancy, we need to consider common-cause failures.

In the case of TT05, there is little to be gained through the addition of a second processor unit if both units are likely to be damaged by the same event (such as water ingress, EMI, radiation, etc).

Fortunately, the Control Unit and the WarranTTor Unit can – if required - be located some distance apart in a TT05 design (Figure 117).

For example, using a CAN or RS-422 / RS-485 link between the units allows us to place the Control Unit in the engine bay of a vehicle and the WarranTTor Unit in the much more benign environment at the edge of the passenger area.

14.9. Design example: Implementing a "1oo2p" architecture

A "1oo2p" architecture is often employed in safety-related systems.

In a 1oo2p architecture, we have two processing units, each controlling a power-switching device (PSD). The PSDs are connected in series, and are used to control the power to one or more devices that are under the control of the processing units. If either (or both) of the processing units disables the PSD that is under their control, power is removed from the controlled device(s).

As an example of a 1oo2p design, Figure 118 shows (in outline) part of the control system for an electric vehicle.

One key requirement in such a system is that we must prevent unintended acceleration of the vehicle in the event of a failure of the main processor. In this case, both the Control Unit and the WarranTTor Unit must both actively close a power switch or the vehicle will stop. If either the Control Unit or WarranTTor Unit detects a problem then they will open the switch that is under their control.

The use of a 1oo2p architecture is common in industrial and transportation systems (including designs that must be developed in compliance with IEC 615008 or ISO 26262): Platform TT05 provides a highly-effective way of meeting such requirements.

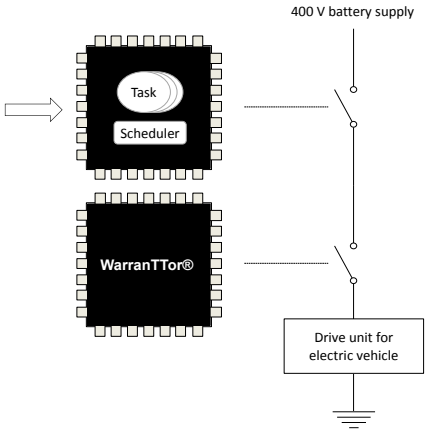


Figure 118: An overview of part of the control system for an electric vehicle.
This has a 1oo2p architecture and could be implemented using TT05.

14.10. Selecting an MCU: General considerations

When selecting an MCU that will be employed as the main processor in any of the platforms outlined in this chapter, we need to start by considering the system environment.

The first concern is usually the temperature at which the processor will be required to operate. For example, in an automotive design, processors located in the engine compartment are likely to have to operate at much higher temperatures than devices that are placed within the passenger area.

When purchasing electronic components, the following labels for operating temperature are typical[27]:

- Full military (-55°C to +125°C)
- Automotive (-40°C to +125°C)
- AEC-Q100 Grade 2 (-40°C to +105°C)
- Extended industrial (-40°C to +85°C)
- Industrial (-20°C to +85°C)
- Commercial (0°C to +70°C)

Please also note that some quoted temperature ranges refer to the ambient temperature (in the environment in which your device will be used) and others refer to the junction temperature (inside the chip itself). Translating between ambient temperature and junction temperature – if required – is rarely straightforward (and techniques for doing this are well beyond the scope of the present book).

[27] Different industries and different organisations use different labels for each temperature range: please check carefully before assuming that the generic labels presented here will apply to your system.

In addition to temperature, you may also need to consider the potential impact of electromagnetic interference (EMI) and radiation on your system. For example, in a satellite design "rad hard" processors are sometimes considered necessary (for example, in the attitude control system).

More generally, EMI can present significant challenges (for example, in automotive, industrial and medical systems). Here, use of a dual-core (or N-core) "lockstep" processor may offer some advantages. Such devices execute the same software on two processor cores – both of which are inside the same semiconductor package – at the same time (or, often, very slightly out of phase). The state of the two processors is compared as the software executes and, if differences are found, it is assumed that a fault has been detected (possibly one caused by EMI). Note that a lockstep processor (in this form) is best viewed as a single-core design with fault-detection mechanisms (sometimes called a "1oo1d" architecture): it provides <u>no protection</u> against any software faults that may be introduced by the development team.

When selecting a processor, it may also be appropriate to consider library support, particularly where the design is being developed in compliance with one or more international safety standards. For example, several manufacturers provide library support with some of their processors for organisations that are developing systems in compliance with IEC 60730. Similarly, many lockstep processors are targeted at developers of ISO 26262 and IEC 61508 applications, and the manufacturers may provide suitable library support with their devices.

Finally, it should be noted that some industries have specific requirements. For example, automotive designs may need to employ AEC-Q100 parts.

14.11. Selecting an MCU: Supporting a TT scheduler

We now turn our attention to some more specific considerations that will apply when considering a processor target that will be used to support one of the system platforms presented in this chapter.

Clearly, we require an appropriate timer (to drive the scheduler ISR): most microcontrollers provide such a timer.

In addition to this timer, we also require [i] a means of resetting the processor when we wish to change mode, and [ii] a means of storing the data about system modes between resets.

In the designs presented in this book we use a watchdog timer (WDT) to reset the processor during mode changes. The vast majority of modern processors provide WDT support, but it is <u>essential</u> that the chosen target processor provides a means of distinguishing between watchdog-induced resets and other forms of processor reset (e.g. power-on resets).

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

It is also preferable that the processor has some form of on-chip oscillator available (in additional to the external crystal oscillator that will be used to drive the CPU in most designs). Not only can the internal oscillator be used to provide an independent oscillator source for the watchdog, it may also be possible to run the system entirely from the internal oscillator in the event of a failure of the external oscillator. As we discussed in Chapter 13, it may also be possible to use the on-chip oscillator to test the behaviour of the main oscillator.

The main system platforms presented in this chapter assume that information can be transferred between system modes. This will typically require some form of non-volatile RAM (that will retain its contents when the processor is reset, and won't be initialised when the processor starts up). Only a few bytes of memory are typically required.

In many modern processors, some form of battery-backed RAM is available, often as part of a real-time clock circuit. This can be an effective means of supporting the required data transfers. Note that the battery backup may be useful when recovering from power-supply failures.

Where internal NVRAM is not available, an external memory device can be connected, as we discussed in connection with TT04. For example, the Microchip 23LCV512 NVRAM may be suitable for some applications. As an alternative, the Everspin Technologies MR25H40 MRAM is available with industrial (-40° to +85 °C) and AEC-Q100 Grade 1 (-40°C to +125 °C) operating temperature ranges.

Note that – as mentioned earlier in this chapter - use of Flash memory, EEPROM or similar technology is not generally suitable for storing system-mode data (because these memory devices usually support a limited number of write cycles before they wear out – and this limit may be reached very quickly).

A final consideration is the availability of a memory protection unit (of some form): this can be a particularly useful feature in TT03 and TT05 designs.

14.12. Selecting an MCU: WarranTTor platform

The WarranTTor platform is usually required to implement a very simple software framework. An effective platform is often an 8-bit microcontroller in a high-temperature, automotive grade package. Such devices are often inexpensive (around US $1 or less).

Depending on the memory capacity of the MCU, we may also need to add an external Flash / EEPROM or similar device to store the task-sequence

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

representation. As noted in Section 14.11, high-temperature devices are inexpensive.

14.13. Conclusions

In this chapter, we have presented three TT platforms that can be used as the basis of reliable embedded systems. Use of each platform is illustrated in a TTRD.

In Chapter 15, we revisit the case study from Chapter 3 and demonstrate how – through the use of Platform TT03 – we can create a framework for a much more robust version of this system.

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

CHAPTER 15: Revisiting the case study

In this chapter, we revisit the case study from Chapter 3 and present a new version of this design based on Platform TT03.

Related TTRDs

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

C programming language (LPC1769 target):

- TTRD15a: Framework for washing-machine controller (TT03)
- TTRD15b: Creates Tick List for TTRD15a (Normal Mode)
- TTRD15c: Creates Tick List for TTRD15a (Limp-Home Mode)

15.1. Introduction

In Chapter 3, we presented a simple initial TT design that could be developed into a controller for a domestic washing machine (Figure 119).

In the present chapter, we will present a more complete framework for the washer controller, this time in a form that would allow us to meet the relevant international safety standards.

We will begin by presenting an overview of the development process that will be followed in this chapter.

15.2. An overview of the development process

In the initial stages of the development of our washing-machine controller we need to: [i] consider the core system requirements; [ii] identify potential threats and hazards that may have an impact on the appliance and / or those using (or in the vicinity of) the appliance; [iii] identify and review any relevant safety standards; [iv] identify a potential system platform (hardware and software architecture); [v] determine whether the proposed team members have the knowledge and experience needed to produce the system.

The above steps are often used to determine whether to proceed with a project. Assuming that we are to proceed, we then need to consider (and record) detailed requirements, develop a high-level design and – crucially – be in a position to provide clear evidence that our design will meet the system requirements (both under "normal" and "fault" conditions). This will involve consideration of: [i] the system modes; [ii] the system states; [iii] the system contract; [iv] the task contracts; [v] the system Tick Lists; and [v] the required fault-detection and fault-handling mechanisms.

The above steps will also allow us to complete a first system prototype: this will be our final goal in the present chapter.

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

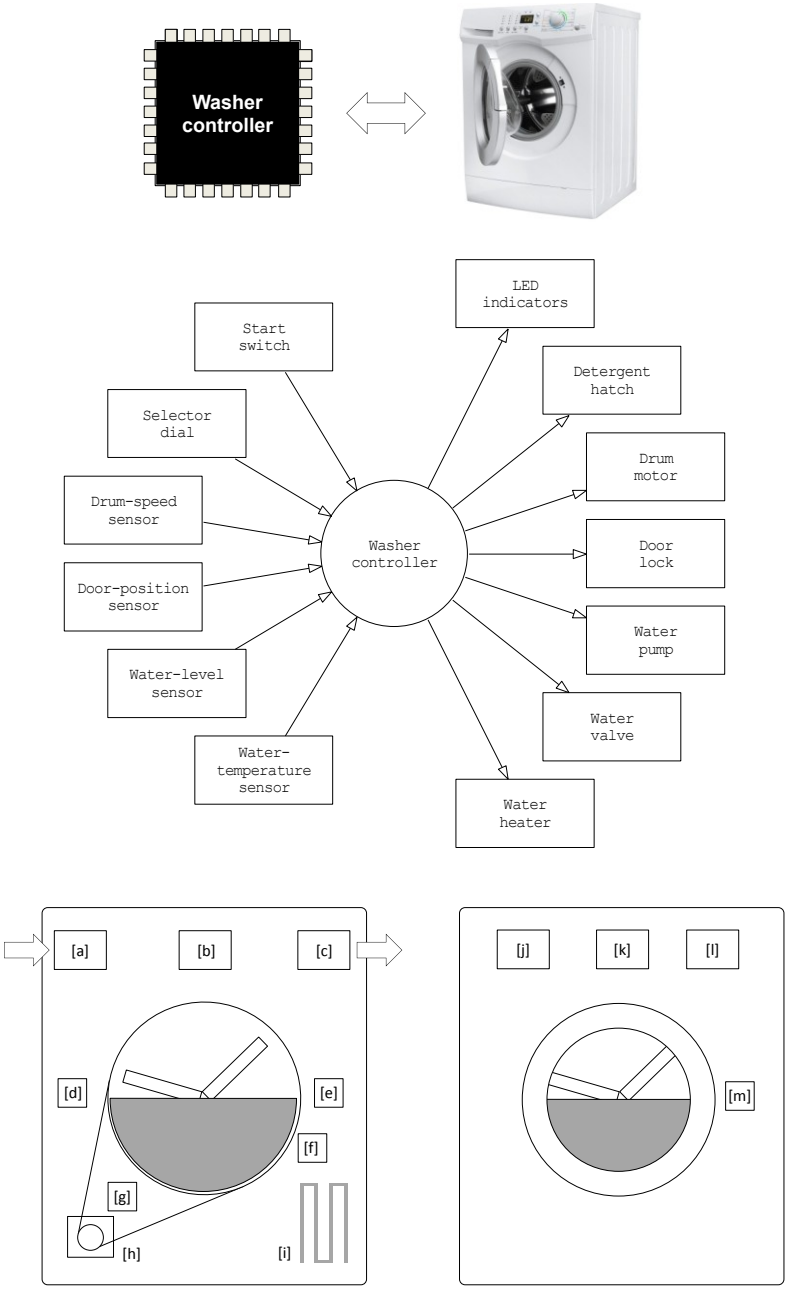


Figure 119: An overview of the components in the washing machine (TTRD15a). Internal view of the system (bottom left): [a] water valve; [b] detergent hatch; [c] water pump; [d] water-level sensor; [e] door-position sensor; [f] water-temperature sensor; [g] drum-speed sensor; [h] drum motor; [i] water heater. External view of the system (bottom right): [j] selector dial; [k] LED indicators; [l] start switch; [m] door lock.

15.3. The system requirements

We provided an informal list of requirements for our washer controller in Chapter 3 (Section 3.4). Our design process in the present chapter will begin with the same list.

Please note that we will introduce a more appropriate representation of the system requirements in Section 15.21.

15.4. Considering potential threats and hazards

Early in the development cycle for any safety-related embedded system, we need to consider potential threats and hazards. This will involve assessment of the risks posed to users of the system or to those in the vicinity.

From this perspective, a washing machine consists of powerful electric motor enclosed in a metal casing. As a normal part of the device operation, the electric motor is used to rotate a heavy metal drum at high speed. Access to this potentially-dangerous mechanism is controlled by a door with an electronic locking mechanism.

The device is used in a domestic environment. There is a risk of injury if access is obtained to the drum while it is rotating. Such injuries could potentially be severe (including loss of a limb), or even life-threatening, particularly for a small child.

The device is connected to a pressurised water supply. The drum is filled with water as a normal part of its operation. There is a risk of flooding if the door is opened at the wrong time: we will assume that this is a "nuisance issue" (rather than a safety issue). However, a combination of water and an electrical supply must always be treated with caution.

In summary: a key threat to users that can be identified is failure of the door lock while the drum is rotating. We will focus on this threat during the discussions that follow.

15.5. Considering international safety standards

Manufacturers of washing machines (and those supplying components for use in such devices) need to comply with various international safety standards, including in this case IEC 60335-1 and IEC 60730-1.

A key challenge is presented by Clause 19 in IEC 60335-1. This clause requires that electronic circuits must be designed and applied in such a way that a fault condition will not render the appliance unsafe with regard to electric shock, fire hazard, mechanical hazard or dangerous malfunction.

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

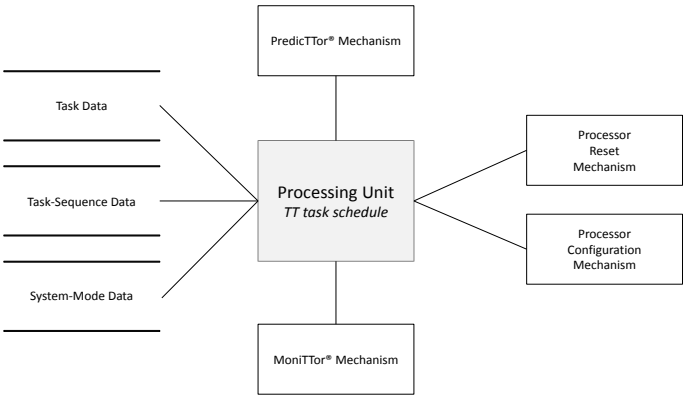


Figure 120: We will use an implementation of Platform TT03 to meet the requirements of our "Class B" washing-machine controller in this chapter.

The effort required to demonstrate compliance with this core clause (and the standard as a whole) depends on the class of equipment being developed: the options are Class A, Class B or Class C.

- Class A control functions are not intended to be relied upon for the safety of the application (IEC 60730, H.2.22.1).
- Class B control functions are intended to prevent an appliance from entering an unsafe state; however, failure of the control function will not lead directly to a hazardous situation (IEC 60730, H.2.22.2).
- Class C control functions are intended to prevent special hazards such as explosion; failure of such functions could directly cause a hazard in the appliance (IEC 60730, H.2.22.3).

In this case, our washer controller will fall into Class B, because failure of the door lock (one of the most serious potential failures) will not lead directly to any injury.[28]

15.6. Potential system platform

One of the permitted architectures for a Class B control system is a single-channel design with periodic self-test (IEC 60730, H.2.16.6).

Our initial assessment is that an LPC1769 MCU combined with Platform TT03 (Figure 120) will provide an effective system foundation.

Please refer back to Chapter 14 for further information about the TT03 platform.

[28] By contrast, a burner control unit for domestic gas-fired water heater would probably be placed in "Class C" since failure of this system may lead directly to an explosion. We will say a little more about Class C designs in Section 15.24.

15.7. Does the team have the required skills and experience?

Development of high-integrity embedded systems involves use of appropriate development processes, software tools and hardware platforms — but it also requires that the people involved have relevant experience and qualifications (see Box 7).

In many ways, this may seem like common sense, but modern standards now make this requirement explicit, requiring that organisations can provide evidence of compliance.

For example, ISO 26262 requires that there should be a "safety culture" within the organisation:

> *"The organisation shall create, foster, and sustain a safety culture that supports and encourages the effective achievement of functional safety."*
>
> [ISO 26262-2, Section 5.4.2.1.]

This standard provides suggestions for various ways in which an organisation can demonstrate that it has created such a safety culture: these include being able to demonstrate that people doing a particular piece of work have "the competence commensurate with the activity assigned".

The requirements of ISO 26262 are having an enormous impact on work in the automotive sector, but concerns about the reliability of embedded systems extend much more widely.

For example, our study in this chapter involves development of a household appliance in compliance with IEC 60335. The first line of the introduction to this standard states:

> *"It has been assumed in the drafting of this International Standard that the execution of its provisions is entrusted to appropriately qualified and experienced persons."*
>
> [IEC 60335-1: 2010, Introduction.]

15.8. Shutting the system down

Before embarking on the development of any safety-related or safety-critical embedded system, it is clearly important to ensure that the team involved has the necessary technical skills and experience (as outlined in the previous section).

Assuming that it has been determined that the team developing our washing-machine controller is appropriately qualified, the second phase of the project will begin by considering the System Contract (as discussed in Chapter 13).

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

Is your team ready to develop safety-critical embedded systems?

In the experience of the author, the techniques presented in the book can form an effective foundation for the development of safety-related and safety-critical systems, in sectors ranging from household goods to cars, aircraft and satellite systems.

Of course, simply reading this book will not make you an instant expert. You'll need to experiment with a few system designs using the techniques presented here, in order to gain familiarity with the techniques that have been presented.

Suppose that your development team has (say) 10 members, and they have all gained experience with the techniques presented in this book on a number of projects: will this then be enough?

The answer to this question will very much depend on the general background and experience of your team. In the author's experience, teams that are most successful in applying the techniques in this book to the development of safety-critical systems will usually also require: [i] one or more domain experts who are familiar with the field in question, whether this is factory automation or satellite design; and [ii] one or more standards experts, who have experience working with the particular safety standards or guidelines with which the system must comply.

Box 7

This includes consideration of the procedures for powering-up the system and shutting it down.

As the process of shutting the system down determines the state that it will be in when powered up, it often makes sense to consider the shut-down process first: this is the approach we will follow here.

There are three main shut-down options: [i] the user performs a "clean" shutdown (we will explain what this means below); [ii] we lose system power, either as the result of a supply failure or because the user has disconnected the device from the supply; or [iii] a fault has been detected.

In the first case, the washer is assumed to have a power on / off button. It is intended that the user will shut the machine down by means of this switch when the system is idle, having completed a wash cycle: this is considered to be a clean shutdown.

A power loss (whatever the cause) may occur at any time. This means that there may be water (and washing) in the machine when power is restored. If power is lost only briefly (or the user cycles the power), the drum may still be turning when the system starts up.

In the event of a fault that renders the machine unusable (e.g. failure of the door lock or water pump), we do not want the user to be able to attempt to use the machine again. Instead, we want to display a fault code and remain in an appropriate mode.

[It must not be possible to clear the fault codes by cycling the power.]

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

15.9. Powering the system up

We will next consider how the system will be powered up.

Clearly, some low-level Power-On Self Tests (POSTs) will be required. As discussed in Chapter 13, NXP Application Note AN10918 describes in detail how to perform POST operations that are in compliance with IEC 60335 on an LPC1769 microcontroller. The Application Note is accompanied by a complete code library. We will assume that this library would be employed in a final version of this system.

As noted in Chapter 13 (Section 13.5 and Section 13.6), we would often expect to check the system configuration at startup. In the case of our washer controller, IEC 60335 specifically requires (Clause R.2.2.8) that developers protect the software in the system from user alteration. We will therefore support configuration tests at startup in our system framework.[29]

If we fail either the power-on or configuration tests when the system is powered up then we will expect the system to enter a Fail-Silent mode (because we cannot safely continue in these circumstances).

If we pass the above tests, then we need to consider any fault codes that were generated during any previous wash cycles. Again, we cannot continue if (for example) there is a fault with the door lock.

Assuming that we have passed the basic POSTs and configuration checks, and have no active fault codes, we then need to try and determine what state the washing machine is in. As we noted in Section 15.8, it is possible that the system will be powered up having completed only part of a previous wash cycle (for example, because of a temporary power failure). To determine whether this is the case, the system will – at least - need to measure the water level in the drum during its initialisation process. Appropriate action will then need to be taken if water is detected: we say more about this in later sections.

Assuming that all of the above checks have been completed successfully, we are ready to configure the system in an appropriate operating mode.

15.10. Periodic system checks

As we (again) discussed in Chapter 13, NXP Application Note AN10918 describes how to perform periodic BIST operations that are in compliance with IEC 60335 on an LPC1769 microcontroller. We assume that the associated code library would be employed in a final version of this system.

---

[29] We may also wish to carry out additional checks on the system configuration during the system operation: we say more about this in Section 15.10.

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

As also discussed in Chapter 13, we would often expect to have to monitor the MCU temperature during the system operation (since the hardware cannot be operated outside its specified range). As TTRD15a may be adapted for use in different systems (for example, those in compliance with IEC 61508), we include support for periodic (MCU) temperature monitoring in this design. Please note that such support would probably not be required in a domestic appliance.

We also provide support in the code framework for periodic tests of the system configuration. Again, such tests may be required in a higher-end design but may not be considered to be necessary in this domestic appliance: these tests are included for completeness in the framework presented here.

15.11. The system modes

We now turn our attention to the system modes.

In TTRD3a, we employed two modes: "Normal" and "Fail-Silent".

In TTRD15a, we have added a Limp-Home Mode.

As you would expect, the Limp-Home Mode is intended as a "half-way house" between the two other modes, and is used to support fault-recovery activities: in particular, it is intended to allow the user to deal with a scenario in which the wash process is aborted while there is still water in the drum. The Limp-Home Mode is primarily intended to provide a safe way of draining the water from the appliance (where it is possible to do so). [30]

Note that no other operations are supported in this mode, and that neither the drum motor nor the water valve is activated: this means that all of the allowed operations can be expected to be safe.

15.12. The system states

As we discussed in Chapter 7, configuring a TT system in a particular mode means scheduling a particular set of tasks to run at pre-determined times.

Within each mode, we also need to consider the possible system states.

Table 17 lists the different states that are employed in TTRD15a in NORMAL mode.

Table 18 lists the different states that are employed in TTRD15a in LIMP_HOME mode.

[30] We cannot, of course, guarantee that the user will be able to drain the drum, because the fault may be that the pump has failed (for example).

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

Table 17: System states for the washing-machine controller (TTRD15a, NORMAL mode)

State	Description
INIT [0]	If it passes the suite of power-on tests (see Section 15.9), the system enters INIT state when it is powered up. In this state, it checks to see if [i] the drum is rotating or [ii] there is water in the drum. In the event of rotation or water, the system moves to LIMP_HOME mode. In the absence of rotation or water, the system remains in INIT state until the Start switch is pressed. If the Start switch is pressed, the system reads the selector dial: the system then moves into START state.
START [1]	In START state, the door is locked. We wait (indefinitely) until the door is closed. Once the door is closed then - if detergent is required (depending on the reading from the selector dial) - this is released into the drum. The system then enters FILL_DRUM state.
FILL_DRUM [2]	In FILL_DRUM state, the water valve is opened to begin filling the drum with cold water. The system checks the water-level sensor periodically, to see if the drum is full. If it takes longer than MAX_DRUM_FILL_DURATION to fill the drum, then something is wrong and the system moves to LIMP_HOME mode. If the drum has filled in time, then the water valve is closed. If hot water is required in the selected wash routine, the system moves into HEAT_WATER state; otherwise, the system moves directly into PRE_WASH state.
HEAT_WATER [3]	In HEAT_WATER state, the water heater is switched on. The system checks the water-temperature sensor periodically, to see if the water has reached the required temperature. If it takes longer than the MAX_WATER_HEAT_DURATION to heat the water, then something is wrong and the system enters LIMP_HOME mode. If the water reaches the required temperature within the time limit then the system moves into PRE_WASH state.
PRE_WASH [4]	In PRE_WASH state, the drum motor is activated in order to rotate the drum. In our simplified design, this simply involves running the motor (constantly) at "medium speed". The system remains in the pre-wash state for a period equal to the PRE_WASH_DURATION; it then deactivates the drum motor and moves into MAIN_WASH state. [*Note that in practice we would drain the drum at the end of the pre-wash and refill it for the main wash, but our simplified design is adequate for our purposes here*.]
MAIN_WASH [5]	In MAIN_WASH state, the drum motor is activated in order to rotate the drum at a medium-high speed (again this is simplified). The system remains in this state for a period equal to the MAIN_WASH_DURATION; it then deactivates the drum motor and moves into DRAIN_DRUM state.
DRAIN_DRUM [6]	In DRAIN_DRUM state, the water pump is activated. If draining the drum takes longer than DRAIN_DRUM_DURATION, the system moves into LIMP_HOME mode, otherwise – after the drum has been drained and the water pump switched off – the system moves into SPIN state.
SPIN [7]	In SPIN state, the drum is rotated at full speed. We remain in this state for a period equal to the SPIN_DURATION. The system then switches off the drum motor and moves into FINISHED state.
FINISHED [-]	In FINISHED state, we double-check that the various system components are switched off. To avoid any risk of injury when the door is opened (just in case the drum is still rotating), there is a delay (of MAX_DRUM_STOP_DURATION seconds) before the door lock is released. The system then returns to INIT state, ready for the next wash.

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

Table 18: System states for the washing-machine controller (TTRD15a2, LIMP_HOME mode)

State	Description
INIT_LH [0]	The system enters INIT_LH state when it is powered up in LIMP_HOME mode. In this state, the system locks the door and sets the other actuators to "OFF" (with the exception of the detergent hatch, which is ignored). The system then checks to see if the drum is rotating: if rotation is detected, the system moves into WAIT_FOR_DRUM_TO_STOP_LH state. If drum rotation is not detected, the system checks to see if there is any water in the drum: if it detects water in the drum, the system moves into DRAIN_DRUM_LH state. If neither drum rotation nor water is detected, the system moves into NORMAL mode.
WAIT_FOR_DRUM_TO_STOP_LH [1]	In WAIT_FOR_DRUM_TO_STOP_LH state, the system checks the door sensor: if the door is open, the system moves into a FAIL_SILENT mode. If the door is not open, the system waits for up to MAX_DRUM_STOP_DURATION seconds for the drum to stop. If the drum has stopped rotating within the time limit, the system moves to INIT_LH mode; otherwise the system moves into a FAIL_SILENT mode.
DRAIN_DRUM_LH [2]	In WAIT_FOR_DRUM_TO_STOP_LH state, the water pump is activated: this begins the process of removing water from the drum. If draining the drum takes longer than DRAIN_DRUM_DURATION, the system moves into a FAIL_SILENT mode, otherwise – after the drum has been drained – the system turns off the water pump and moves into INIT_LH state.

Table 19: The core task sets used in the two main system modes.

Task	Normal Mode	Limp-Home Mode
Detergent_Hatch	X	
Door_Lock	X	
Door_Sensor	X	X
Drum_Motor	X	
Drum_Sensor	X	X
Selector_Dial	X	
Start_Switch	X	
Water_Heater	X	
Water_Level	X	X
Water_Pump	X	
Water_Temperature	X	
Water_Valve	X	

Please note that the states listed in Table 17 (from TTRD15a) have a lot in common with the states from TTRD3a (Table 2, on Page 67). However, there are also a number of differences between the two tables. For example, there is no Fault state in TTRD15a. Instead, faults cause a direct transition to a "fault" mode. This significantly reduces the time taken to respond to detected faults.

We begin to say more about fault detection and fault-handling mechanisms in TTRD15a in Section 15.15.

15.13. The task sets

Based on the discussions in the previous sections, TTRD15a has three modes: "Normal", "Limp Home" and "Fail Silent".

Our full set of tasks will be almost the same as in the initial design (in Chapter 3), but not all tasks will run in all modes: please see Table 19 for details.

Please note that we run only a single function in Fail-Silent Mode (as in previous examples in this book).

15.14. Modelling the task set and adjusting the task offsets

Code Fragment 51 shows how the tasks are added to the schedule in NORMAL mode.

It is important that the "system" task runs first in the sequence, since this performs the initialisation of various shared variables. Following the techniques presented in Section 4.8 and Section 5.12, we use appropriate task offsets to achieve this.

In this case, the other task constraints are limited, and we can simply adjust the offset of the remaining tasks in such a way that we spread out the CPU load (and avoid the total task execution time in any given tick exceeding the tick interval): again, we do this using the techniques introduced in Section 4.8.

Using TTRD15b, we can generate the complete tick list for this mode: this list can then be reviewed in order to confirm that we have met all of the system requirements.[31]

This process will then be repeated for the simpler LIMP_HOME mode.

In this case Code Fragment 52 shows how the tasks are added to the schedule.

Using TTRD15c, we can generate the complete tick list for this mode.

---

[31] The same Tick List will then be used as the basis of our PredicTTor design, as outlined in Section 15.18.

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

```
// Add watchdog task first
SCH_Add_Task(WATCHDOG_Update, 0, 1, 50, 0);

// Add main washer task (system-state task)
SCH_Add_Task(SYSTEM_STATE_N_Update, 0, 1000, 300, 0);

// Door lock / door sensor released frequently
SCH_Add_Task(DOOR_LOCK_Update, 1, 10, 100, 0);
SCH_Add_Task(DOOR_SENSOR_Update, 1, 10, 100, 0);

// Drum motor / drum sensor released frequently
SCH_Add_Task(DRUM_MOTOR_Update, 2, 10, 100, 0);
SCH_Add_Task(DRUM_SENSOR_Update, 2, 10, 100, 0);

// Task to poll SW3 on EA Baseboard (Start switch)
SCH_Add_Task(SWITCH_SW3_Update, 2, 10, 100, 0);

// Remaining tasks are released less frequently
SCH_Add_Task(WATER_HEATER_Update, 3, 100, 100, 0);
SCH_Add_Task(WATER_TEMP_SENSOR_Update, 3, 100, 100, 0);

SCH_Add_Task(WATER_PUMP_Update, 4, 100, 100, 0);
SCH_Add_Task(WATER_VALVE_Update, 4, 100, 100, 0);
SCH_Add_Task(WATER_LEVEL_SENSOR_Update, 4, 100, 100, 0);

SCH_Add_Task(SELECTOR_DIAL_Update, 5, 1000, 100, 0);
SCH_Add_Task(DETERGENT_HATCH_Update, 5, 1000, 100, 0);

// Perform periodic system checks
SCH_Add_Task(SYSTEM_SELF_TEST_Check_MCU_temp, 6, 1, 50, 0);
SCH_Add_Task(SYSTEM_SELF_TEST_Perform_BISTs, 6, 1000, 500, 0);

// Add Heartbeat task
SCH_Add_Task(HEARTBEAT_Update, 7, 1000, 50, 0);
```

Code Fragment 51: Adding tasks to the schedule (NORMAL mode).

```
// Add watchdog task first
SCH_Add_Task(WATCHDOG_Update, 0, 1, 50, 0);

// Add main washer task (system-state task)
SCH_Add_Task(SYSTEM_STATE_LH_Update, 0, 1000, 500, 0);

// Add low-level washer tasks
SCH_Add_Task(DOOR_SENSOR_Update, 1, 100, 100, 0);
SCH_Add_Task(DRUM_SENSOR_Update, 1, 100, 100, 0);
SCH_Add_Task(WATER_LEVEL_SENSOR_Update, 1, 100, 100, 0);

// Perform periodic system checks
SCH_Add_Task(SYSTEM_SELF_TEST_Check_MCU_temp, 2, 1, 50, 0);
SCH_Add_Task(SYSTEM_SELF_TEST_Perform_BISTs, 2, 1000, 500, 0);

// Add Heartbeat task (fast)
SCH_Add_Task(HEARTBEAT_Update, 2, 250, 20, 0);
```

Code Fragment 52: Adding tasks to the schedule (LIMP_HOME mode).

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

15.15. Fault-detection and fault handling (overview)

In TTRD3a, we employed two modes: "Normal" and "Fail-Silent". In the resulting prototype, the fault-detection / fault-handling mechanisms were somewhat limited:

- We employed high-level "state timeout" mechanisms to detect problems with components such as the water heater and pump. If problems were detected by these mechanisms, this triggered a state change (rather than a mode change), on a timescale measured in seconds.[32]
- We relied upon the WDT to detect any timing-related faults (caused, for example, by task overruns), and – implicitly – assumed that this mechanism would detect any other low-level faults that were not picked up by the state-timeout mechanisms.

In TTRD15a, we continue to employ state timeouts: these are still effective – high-level – mechanisms. In addition, we employ both POSTs (as outlined in Section 15.9) and periodic BISTs (as outlined in Section 15.10).

To complete the design, we employ a form of Resource Barrier, a MoniTTor unit and a PredicTTor unit to detect run-time faults. We describe these mechanisms in the sections that follow.

15.16. Using Lightweight Resource Barriers

When implementing Lightweight Resource Barriers (LRBs), we use two global variables to store each data value that needs to be transferred between tasks. More specifically, LRBs for common "unsigned integer" variables involve storing both the data value and an "inverted copy" of each data value, as we did when storing information about the system mode in NVRAM in Chapter 7.

Used in this way, we can very easily perform checks (periodically and / or when data are used) to ensure that the two copies of each variable "match up". Unlike the full Resource Barriers (detailed in Chapter 8), we have no function-call overhead in this checking process.

Code Fragment 53 illustrates use of LRBs in one task from TTRD15a.

Please note that compliance with IEC 60335 requires that – where redundant memory is used to store data - the two copies of the data should use different formats (Section R.2.2.1): LRBs meet this requirement.

[32] We will discuss the system states in TTRD3a and TTRD15a further in Section 15.12.

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

```
uint32_t SWITCH_SW3_Update(void)
   {
   static uint32_t Duration = 0;
   uint32_t Return_value = RETURN_NORMAL;
   uint32_t Precond_gpio;
   uint32_t sw3_input;

   // Precond check: Check that Switch pin is set for input
   Precond_gpio = GPIO_Check_Input(START_SWITCH_PORT, START_SWITCH_PIN);

   if (Precond_gpio == RETURN_FAULT)
      {
      Return_value = RETURN_FAULT;
      }

   // Check data integrity (Lightweight Resource Barriers)
   if (Start_switch_pressed_G != ~Start_switch_pressed_iG)
      {
      Return_value = RETURN_FAULT;
      }

   // ---

   // If we have met the preconditions, we read the switch
   if (Return_value == RETURN_NORMAL)
      {
      // Read "reset count" switch input (SW3)
      sw3_input = (GPIO_ReadValue(START_SWITCH_PORT) & START_SWITCH_PIN);
      if (sw3_input == SW_PRESSED)
         {
         Duration += 1;

         if (Duration > SW_THRES)
            {
            Duration = SW_THRES;
            Start_switch_pressed_G = 1;
            }
         else
            {
            Start_switch_pressed_G = 0;
            }
         }
      else
         {
         // Switch not pressed - reset the count
         Duration = 0;
         Start_switch_pressed_G = 0;
         }
      }

   // Set inverted copy
   Start_switch_pressed_iG = ~Start_switch_pressed_G;

   return Return_value;
   }
```

Code Fragment 53: The interface to the “Start” switch in TTRD15a.

15.17. A MoniTTor unit

As you would expect, TTRD15a employs a MoniTTor unit to ensure that tasks complete within their WCET and BCET limits.

The MoniTTor unit follows the design presented in Chapter 9.

15.18. A PredicTTor unit

As you would expect, TTRD15a also employs a PredicTTor unit to ensure that the tasks execute in the correct sequence at run time.

The PredicTTor unit follows the design presented in Chapter 10.

Two task-sequence representations (TSRs) were required for use with the PredicTTor unit (one for NORMAL mode, one for LIMP_HOME mode). These TSRs were created using "Dry Schedulers" (as explained in Section 5.11).

The Dry Schedulers are implemented in TTRD15b (NORMAL mode) and TTRD15c (LIMP_HOME mode).

15.19. A simple system model and fault-injection facility

As we noted in Section 13.15, it is difficult to begin to test any form of embedded control or monitoring system unless we have a "target" available (that is, something to control or monitor).

In the case of our washing machine, we might start the development using a second microcontroller to simulate our washing machine. This would allow us to carry out initial checks on the system architecture. We might then move on to use a basic hardware-in-the-loop test bed (on a bench top), with some of the key system components (e.g. the drum motor and driver) in place. Once these tests were complete, we would be in a position to carry out the first trials on the complete system hardware.

In TTRD15a, we have included a simple simulation and fault-injection framework within the design itself, in order to allow demonstration and testing of the design <u>without</u> requiring a second processor.

The heart of the simple model is a representation of the water level in the drum: this is stored in an NVRAM register (GPREG4). This means that if the system is reset from a state in which there would be water in the drum, this will be modelled correctly when the system starts again (provided of course that power is maintained to the NVRAM). Code Fragment 54 and Code Fragment 56 illustrate use of this register.

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

```
void WATER_LEVEL_SENSOR_Init(void)
   {
   // Provide an initial value (updated on first task release)
   Water_level_reading_G = 0;
   Water_level_reading_iG = ~0;

   // Simple model of water in drum (value stored in NVRAM
   // => still water in drum after reset.

   // Model is reset if CFC jumper is inserted
   // See Port Header and System Mode module
   if (Model_reset_G)
      {
      // GP reg in NVRAM used to represent the water level
      LPC_RTC->GPREG4 = 0;
      }
   else
      {
      // Do simple sanity checks on GPREG4 data
      // (will be "random" on first run)
      uint32_t Test_level = (uint32_t) LPC_RTC->GPREG4;

      if ((Test_level < 0) || (Test_level > 50))
         {
         // Data out of range => reset
         LPC_RTC->GPREG4 = 0;
         }
      }
   }
```

Code Fragment 54: Part of the TTRD15a simulation: Setting up the water-level sensor.

```
uint32_t WATER_LEVEL_SENSOR_Update(void)
   {
   uint32_t Return_value = RETURN_NORMAL;
   uint32_t Water_level;

   // Check data integrity (Lightweight Resource Barriers)
   if (Water_level_reading_G != ~Water_level_reading_iG)
      {
      Return_value = RETURN_FAULT;
      }

   // Note: we don't check the values used for modelling
   // (because these won't appear in the final system)
   // ---

   // If we have met the pre-conditions, we proceed
   if (Return_value == RETURN_NORMAL)
      {
      // Simple model used here (in place of sensor reading)

      // Read current water level from GPREG4
      // This NVRAM location allows us to maintain the record of the
      // water level between resets (a simple model).
      Water_level = LPC_RTC->GPREG4;
```

Code Fragment 55: Part of the TTRD15a simulation: Measuring water level. [Part 1 of 2]

```
      // Is the drum filling?
      if ((Water_valve_required_state_G == OPEN)
           && !INJ_WATER_VALVE_FAULT)
         {
         // Drum is filling
         Water_level++;

         // Is the drum full of water (ready to wash)?
         if (Water_level >= 50)
            {
            Water_level = 50;
            }
         }

      // Is the drum being drained?
      if ((Water_pump_required_state_G == ON) && !INJ_WATER_PUMP_FAULT)
         {
         // Drum is being drained
         if (Water_level > 0)
            {
            // Drum is draining
            Water_level--;
            }
         }

      // Set default value for Water_level_reading_G
      Water_level_reading_G = 0;
      Water_level_reading_iG = ~0;

      // Check if drum is empty
      if (Water_level > 0)
         {
         // Drum has water (not necessarily enough for washing)
         Water_level_reading_G = 1;
         Water_level_reading_iG = ~1;
         }

      // Check if drum is full enough for washing
      if (Water_level > 25)
         {
         // Drum has enough water for washing
         Water_level_reading_G = 2;
         Water_level_reading_iG = ~2;
         }

      // Update GPREG4
      LPC_RTC->GPREG4 = Water_level;
      }

   return Return_value;
   }
```

Code Fragment 55: Part of the TTRD15a simulation: Measuring water level. [Part 2 of 2]

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

```
/*-----------------------------------------------------------------------*

  Refer to the relevant tasks for info about these faults.

  If multiple faults are selected, the first one detected
  will be reported.

  Other simple fault-injection options (not controlled via this file):

  1. To "inject" a simple watchdog fault, remove the WDT jumper
     (before powering up the board)

  2. To simulate a power failure mid wash / user cycling power mid wash
     use the reset switch on the EA Baseboard.  If there is water
     in the drum when the reset is performed, the system will respond
     to this when it starts up (unless the CFC_JUMPER is inserted).

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

// Fault detected during system configuration checks
#define INJ_CONFIGURATION_FAULT (0)

// Fault detected during POSTs
#define INJ_POST_FAULT (0)

// Fault detected during BISTs
#define INJ_BIST_FAULT (0)

// Fault with door sensor *or* door lock
#define INJ_DOOR_FAULT (0)

// Fault with drum motor or drum sensor
#define INJ_DRUM_MOTOR_FAULT (0)

// Fault with water heater or water temp sensor
#define INJ_WATER_HEATER_FAULT (0)

// Fault with water pump or water level sensor
#define INJ_WATER_PUMP_FAULT (0)

// Fault with water valve or water level sensor
#define INJ_WATER_VALVE_FAULT (1)

// Setting this flag causes Heartbeat task to overrun after ~5 seconds
// => this fault should be detected by the MoniTTor unit
#define INJ_TASK_OVERRUN_FAULT (0)

// Changes the task sequence
// => this fault should be detected by the PredicTTor unit
// => see "Heartbeat" task for details
#define INJ_TASK_SEQUENCE_FAULT (0)

// Injects resource fault
// => this fault should be detected by the Resource Barriers
// => see "Heartbeat" task for details.
#define INJ_RESOURCE_FAULT (0)
```

Code Fragment 56: Controlling the injection of faults in TTRD15a.

Code Fragment 56 illustrates how different faults can be injected in the design. For example, we can simulate a fault with the water pump as follows:

```
#define INJ_WATER_PUMP_FAULT (1)
```

This will then be reflected in the behavior of the water-level sensor task (see Code Fragment 55).

Note that this model is far from perfect, but it provides a means of exploring the operation and architecture of TTRD15a: it is adequate for our purposes here.

15.20. Fault codes and fault reporting

In the event of the failure of an embedded system, we usually wish to determine the cause. In many cases, we wish to use the internal monitoring system to obtain and report (or store) accurate and detailed fault codes.

In the case of TTRD15a, we maintain an extensive list of 3-digit fault codes. Note that facilities for reporting faults are often very limited in embedded designs. In this case, we report these as a sequence of digits via the single 7-segment LED unit on the EA Baseboard: in this way fault code "213" is displayed as "F" then "2" then "-" then "1" then "-" then "3": each digit is displayed for around 1 second at a time, then we repeat the sequence.

A full list of the fault codes supported in this design is given in Table 20.

15.21. Revisiting the system requirements

In Section 15.3, we stated that our design in this chapter would be based on the informal set of system requirements that we first presented in Section 3.4. In practice, the list presented in Section 3.4 is suitable only as very early draft.

In this chapter (and throughout this book), we are concerned with the creation of a reliable, real-time embedded system that must be [i] fully tested and verified during development; and [ii] monitored for faults when in use.

During the development, we can only conduct an effective test and verification (T&V) process for any system if we have a complete requirements specification to work from (since the requirements specification is the only "benchmark" against which the "correctness" – or otherwise – of the system may be assessed).

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

Table 20: Fault codes reported by TTRD15a.

Code	Fault label	Description
200	FAULT_UNIDENTIFIED	No identified fault. May be generated by unplanned WDT overflow
201	FAULT_NO_WDT	WDT not enabled.
202	FAULT_CONFIGURATION	Failed configuration check
203	FAULT_SCH_RUNNING	Fault when attempting to add task to the schedule: the scheduler is already running.
204	FAULT_SCH_TOO_MANY_TASKS	Fault when attempting to add task to the schedule: no space available in the scheduler array
205	FAULT_SCH_ONE_SHOT_TASK	Attempt to add a "one shot" task to the schedule (only periodic tasks are supported)
206	FAULT_POST	Failed POST
207	FAULT_BIST	Failed BIST
208	FAULT_MCU_TEMPERATURE	MCU temperature is out of range
209	FAULT_RESOURCE	Resource-related fault
210	FAULT_TASK_TIMING	Timing-related fault (WCET or BCET)
211	FAULT_TASK_SEQUENCE	Task-sequence fault
212	FAULT_WATER_IN_DRUM	Water found in drum in initial state (Normal mode).
213	FAULT_DRUM_TURNING	Drum found to be rotating in initial state (Normal mode)
214	FAULT_DRUM_STOP_OPEN_DOOR_LH	Drum is rotating (LH mode). Waiting for it to stop. Door appears to be open (dangerous combination).
215	FAULT_DRUM_STOP_TIMEOUT_LH	Drum is rotating (LH mode). Timeout while waiting for it to stop
216	FAULT_DOOR	A fault with the door lock or door sensor?
217	FAULT_FILL_DRUM	A fault with the water supply or water valve?
218	FAULT_HEAT_WATER	A fault with the water heater?
219	FAULT_DRAIN_DRUM	A fault with the pump?
220	FAULT_DRAIN_DRUM_TIMEOUT_LH	Timeout while trying to drain drum in LH mode

When drawing up such a specification we need to produce a clearly-defined list of requirements. The software modules (and / or hardware modules) then need to be designed and implemented to meet these requirements. The link between Requirement X and the related software / hardware Module Y needs to be clearly documented and justified (we say a little more about this below).

Overall, the T&V process must provide a "paper trail", explaining how it was confirmed that the completed system does – indeed – meet all of the requirements.

As a minimum way of supporting such a process, we suggest the use of [i] a numbered list of system requirements; [ii] a numbered list of software files; [iii] a numbered list of hardware schematics; [iv] a numbered list of review documents; [iv] a set of tables in a word-processor or spreadsheet that are used to link the above documents together.

We provide some examples to illustrate the required links between the system requirements and the source-code listings in Table 21 to Table 23.

Table 21: System requirements (extract)

Requirement	Description
R1	The Washing-Machine Controller (WMC) shall support the operating modes and states detailed in this section.
R1.1	The Washing-Machine Controller (WMC) shall be in one of the following operating modes: NORMAL, LIMP_HOME, FAIL_SILENT.
R1.2	Within mode NORMAL, the WMC will operate in one of the following states: INIT, START, FILL_DRUM, HEAT_WATER, PRE_WASH, MAIN_WASH, DRAIN_DRUM, SPIN, FINISHED.
R1.3	Within mode LIMP_HOME, the WMC will operate in one of the following states: INIT_LH, WAIT_FOR_DRUM_TO_STOP_LH, DRAIN_DRUM_LH.
R1.4	Within mode FAIL_SILENT, the WMC will operate in a single (unnamed) state. After an initialisation period, this will be a static state.
R2	When powered on, the WMC shall perform the startup self tests described in this section.
R2.1	Power-On Self Tests (POSTs) in compliance with IEC 60335, as documented in NXP Application Note NXP Application Note AN10918.
R2.2	Tests of its own configuration files, as documented in NXP document UM10360 (2 April 2014), Section 32.10.

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

Table 22: List of source-code files (extract)

ID	File name
L0032	system_1769_002-0_c15a.c
L0033	system_1769_002-0_c15a.h
L0034	system_mode_1769_002-0_c15a.c
L0035	system_mode_1769_002-0_c15a.h
L0038	system_self_test_configuration_1769_002-0_c15a.c
L0039	system_self_test_configuration_1769_002-0_c15a.h
L0042	system_self_test_post_1769_002-0_c15a.c
L0043	system_self_test_post_1769_002-0_c15a.h
L0044	system_state_lh_1769_002-0_c15a.c
L0045	system_state_lh_1769_002-0_c15a.h
L0046	system_state_n_1769_002-0_c15a.c
L0047	system_state_n_1769_002-0_c15a.h

Table 23: Links between system requirements and source-code listings for a safety-related or safety-critical design, or one that has certification requirements (extract)

Requirement	Related listing(s)	Related review documents
R1.1	L0034, L0035	RD001_1
R1.2	L0046, L0047	RD001_2
R1.3	L0044, L0045	RD001_3
R1.4	L0032, L0033	RD001_4
R2.1	L0042, L0043	RD002_1
R2.2	L0038, L0039	RD002_2

Table 24: Links between system requirements and source-code listings for a design that has no safety or certification requirements (extract)

Requirement	Related listing(s)	Notes
R1.1	L0034, L0035	Code review carried out by MJP [23 January 2015]. No problems noted.

We have not attempted to include the related document "RDxxx" here. Such documents should include enough information to justify the claim that Requirement X is met by Listing Y. For example, design and code reviews may be carried out and documented by an independent team (that is, one that was involved in neither the design nor the implementation). Bear in mind that – in the worst case scenario – such review documents may be examined by an "Expert Witness" in a court case: they need to be produced accordingly.

Note that if there are no safety or certification requirements, then it may be sufficient to perform a code review and note this fact (see Table 24).

Table 25: The directory structure used in TTRD15a.

Directory	Summary of contents
fault_injection	The fault-injection model. Intended for demo purposes only.
heartbeat	The Heartbeat LED module.
main	Function main() and the Project Header (Main.H).
monittor	The MoniTTor module.
port	The Port Header.
predicttor	The PredicTTor module.
resource_barriers_light	The Resource Barriers.
scheduler	The task scheduler.
system	Supports system initialisation and “fail silent” shutdown.
system_mode	Support for the different system modes.
system_state	Support for the different system states.
system_test	Support for the various low-level (self) tests.
task_support_functions	Support functions used by the tasks.
tasks	The tasks.
watchdog	The watchdog module.

15.22. Directory structure

The directory structure employed in TTRD15a is described in Table 25.

15.23. Running the prototype

To support the prototype, both the LPCxpresso board and the EA Baseboard are required: please refer to Appendix 1 for details.

You will require the watchdog jumper: please refer to Appendix 1 for details. Note that without this jumper in place, you will immediately see Fault Code 201.

If a fault code is generated and power is maintained to the NVRAM, this fault code will not be cleared by a system reset. To reset the fault code, the CFC jumper will be required: please refer to the Port Header file for details.

To explore the operation of the example, we suggest that you start without any fault codes and try the normal operation. SW3 (on the EA Baseboard) represents the Start switch and will trigger the “wash” operation. The system state will be displayed on the 7-segment display as the cycle progresses.

If you then use the fault-injection framework, you can explore the system architecture in more detail. Note that if you don't have a battery installed on the Baseboard (to power the NVRAM), you can still simulate a power failure during the wash cycle by pressing the reset switch (on the Baseboard).

15.24. International standards revisited

This book has described various ways of developing TT systems. As we conclude the TT design of our washing-machine controller, it is worth considering how an ET solution might differ from TTRD15a.

When compared with TTRD15a, it would not be nearly as straightforward to either model or monitor the ET design: a longer period of testing would probably therefore be required for the ET system. As it is very rarely possible to determine whether enough testing has been carried out on an ET design (because of the large number of possible system states), it is very likely – in the view of the author – that the TT design would be found to be more reliable in the field. It is perhaps, therefore, not surprising that compliance with IEC 60335 requires limited use of interrupts (Section R.3.2.3.2).

Finally, please note that in the type of (Class B) design explored in this chapter, we would not normally expect to employ a WarranTTor Unit. However, if the design was categorised as Class C, then we would usually expect to require such a unit. Class C control functions: [i] are intended to prevent "special hazards", such as an explosion; and [ii] may directly cause a hazard if they fail (IEC 60730-1, H.2.22.3).

15.25. Conclusions

This concludes our overview of the revised version of the washing-machine controller.

Please refer to TTRD15a to see the complete framework for this system.

PART FIVE: CONCLUSIONS

"Safety and security are intertwined through communication. ... A safety-critical system is one in which the program must be correct, otherwise it might wrongly change some external device such as an aircraft flap or a railroad signal. A security-critical system is one in which it must not be possible for some incorrect or malicious input from the outside to violate the integrity of the system by, for example, corrupting a password-checking mechanism."

John Barnes, 2007

in "Avionics: Elements, Software and Functions"
Edited by C.R. Spitzer
CRC Press

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

CHAPTER 16: Conclusions

In this chapter we present our conclusions.

16.1. The aim of this book

Throughout this book, we have argued that – if we wish to develop embedded systems for which it is possible to avoid Uncontrolled System Failures (as defined on p.xv) – we should start each new project by considering whether it will be possible to employ a TT architecture.

If a TT architecture will meet the needs of the application, then we have further argued that the project should follow a "Model-Build-Monitor" methodology (as laid out in previous chapters).

To support the development of such projects, we have described three platforms for reliable systems (TT03, TT04 and TT05).

16.2. The LPC1769 microcontroller

This book is part of a series. In this edition, our focus has been on designs based on an LPC1769 microcontroller.

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

The LPC1769 was selected as the hardware platform for this book because it forms an effective hardware target for TT03-based systems.

In keeping with the chosen microcontroller target, the two case studies in this book have had a focus on white goods (specifically, a domestic washing machine).

As we have discussed in the case studies, a washing machine incorporates a powerful electric motor and is used in an environment in which we can expect there to be young, unsupervised children. Any controller for such a device must – clearly – be able to avoid Uncontrolled System Failures.

The washing machine controller is an IEC 60335 "Class B" application: the needs of this application can be met through use of a TT03 platform.

We have also suggested (in Chapter 15), that a combination of the LPC1769 microcontroller and a TT05 platform may also form a suitable foundation for more hazardous IEC 60335 "Class C" applications.

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

16.3. From safety to security

Increasingly, as we move into a world with an "Internet of Things" (IoT), we also need to consider security concerns. As we noted in Chapter 15, developers of a washing machine are require (IEC 61335, R.2.2.8) to protect the software in the system from user alteration. We considered this responsibility in the context of the design of a door lock for our washing machine in Chapter 15. Protecting against such "attacks" on consumer devices will become more challenging as more designs are opened up to WWW access.

As we have noted throughout this book, we cannot ever expect to be able to be able rule out the possibility that one task will interfere with one another when they share a computer platform. Fortunately, however, we can – using the techniques explored in this book – provide evidence of our ability to detect such interference. The same mechanisms will also allow us to detect security-related changes to the system behaviour.

Note that as such breaches become more likely, we may need to consider using TT05 platforms in designs that are currently operating TT03 or TT04 platforms.

16.4. From processor to distributed system

In the discussions in this book, we have assumed that you are working with a TT design based on a single processor, with the possible addition of a separate processor used for monitoring / control purposes (the WarranTTor processor).

Many designs incorporate more than one processor, perhaps operating in some form of bus arrangement. For example, a Steer-by-Wire application for a vehicle will involve steering sensors (in the passenger area) and steering actuators (adjacent to the wheels), connected by a suitable computer network.

In such circumstances, we need to consider both the scheduling of tasks on the individual processors (at each end of the network), and the scheduling of messages on the network itself. We may also need to be sure (using "Bus Guardians", for example) that failure of one processor in the system cannot have an impact on the other processors. We may require a duplicated (or triplicated) bus arrangement, or a different network topology.

You'll find guidance on the development of reliable, distributed TT systems using "Shared-Clock" schedulers in "PTTES" (Pont, 2001).

10.5. Conclusions

In this chapter, we've presented our conclusions.

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

APPENDIX

"The problems of the real world are primarily those you are left with when you refuse to apply their effective solutions."

Edsger W. Dijkstra, 1988.

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

APPENDIX 1: LPC1769 test platform

In this appendix we provide some information about the test platform that was used to develop the LPC1769 TTRDs.

A1.1. Introduction

In this edition of the book, the main processor target is an NXP® LPC1769 microcontroller.

In almost all cases, the code examples that are associated with the book (the various "Time-Triggered Reference Designs", TTRDs) can be executed on a low-cost and readily-available evaluation platform: we describe the hardware platform that was used to create the examples associated with this book in this appendix.

A1.2. The LPC1769 microcontroller

As noted in the Preface, 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.

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

A1.3. LPCXpresso toolset

The LPC1769 TTRDs were created using the LPCXpresso toolkit (Figure 121).

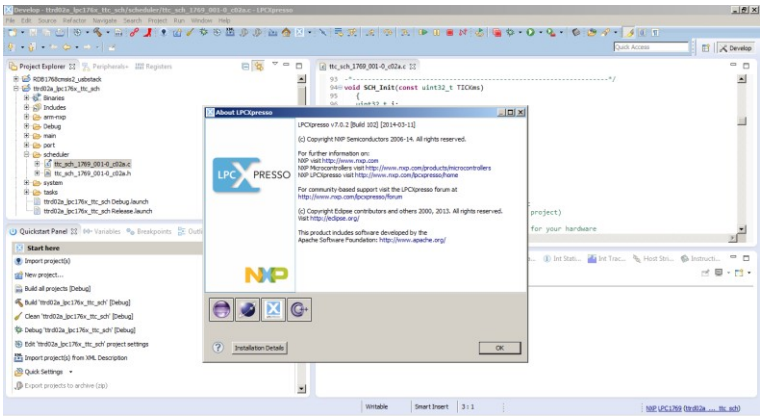


Figure 121: The LPCXpresso IDE.

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

The version of this toolkit that was used to create these designs can be downloaded (free of charge) from the NXP WWW site:

http://www.lpcware.com/lpcxpresso

From the NXP WWW site:

> *"Designed for simplicity and ease of use, the LPCXpresso IDE is a highly-integrated software development environment for NXP's LPC microcontrollers. It includes all the tools necessary to develop high-quality software solutions in less time and at a lower cost. LPCXpresso builds on its Eclipse foundation by including many enhancements that simplify development with NXP LPC microcontrollers. It also features the industry-standard GNU toolchain, with the choice of a proprietary optimized C library or the standard Newlib library. The LPCXpresso IDE lets developers build an executable of any size with full code optimization. LPCXpresso 6 and 7 are a major updates to NXP's MCU tools ecosystem. It reflects the NXP acquisition of Code Red Technologies, the makers of Red Suite."*

A1.4. LPCXpresso board

The examples in this book all target the LPCXpresso LPC1769 board (Figure 122).

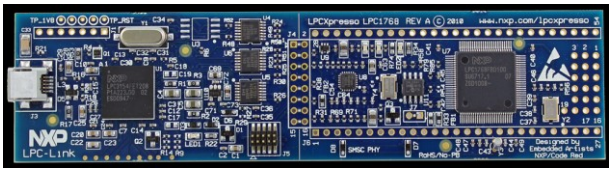


Figure 122: The LPC1769 board.
Photo from NXP.

This board is widely available, at very low cost.

A1.5. The EA Baseboard

Most of the examples in this book assume that the LPCXpresso board is connected to the Baseboard from Embedded Artists (the "EA Baseboard").

This board includes the features listed in Table 26.

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

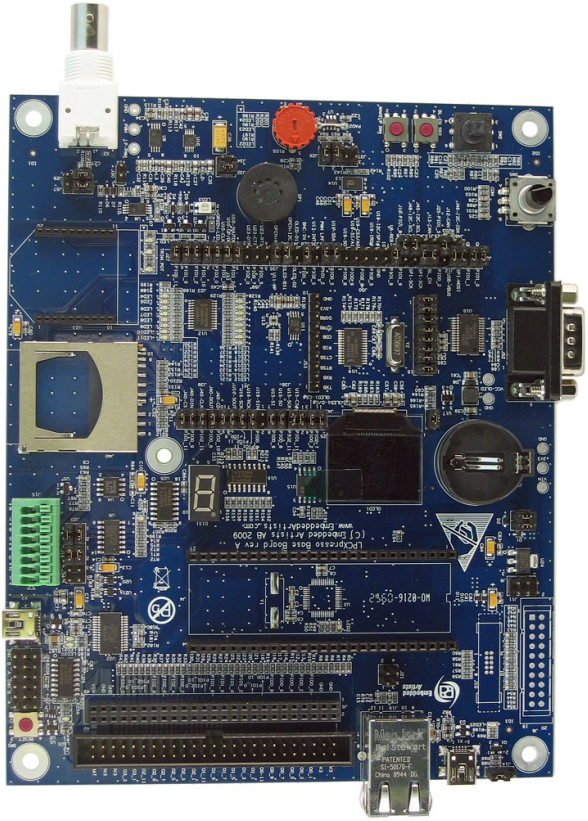


Figure 123: The EA Baseboard.
Photo from Embedded Artists.

Table 26: Features of the EA Baseboard.
[Information obtained from the EA WWW site.]

General features	Socket for LPCXpresso module 50-pin expansion dual row pin list connector (male, 100 mil) for connection to external designs / logic analyser 50-pin expansion dual row header connector (female, 100 mil) for connection to a breadboard Battery power (small coin battery) for NVRAM USB interface Reset pushbutton
Digital IO	RGB-LED (can be PWM controlled) 5-key joystick switch 2 pushbuttons, one for activating bootloader Rotary switch with quadrature encoding (timer capture) Temperature sensor with PWM output (timer capture)
Analog IO	Trimming potentiometer input (analog input) PWM to analog LP-filtering (PWM output and analog input) Speaker output (PWM output) Oscilloscope probe in/out stage
Serial bus - SPI	Shift register driving 7-segment LED SD/MMC memory card interface Dataflash SPI-NOR Flash
Serial bus - I2C	PCA9532 port expander connected to 16 LEDs 8kbit E2PROM MMA7455L accelerometer with I2C interface Light sensor
Serial bus - I2C/SPI shared	SC16IS752 - I2C/SPI to 2xUART bridge; connected to RS232 full-modem interface and one expansion UART 96x64 pixel white OLED (alternative I2C/SPI interface)
Serial bus - UART	USB-to-serial bridge, with automatic ISP activation RS422/485 interface Interface socket for XBee RF-module
Other LPC1769 support	CAN bus interface Ethernet RJ45 connector
Power	Powered via USB (+5V)

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

A1.6. Running TTRD02a

Figure 124 shows the combination of LPCXpresso board and EA Baseboard that can be used to run most of the TTRDs in this book, starting from TTRD02a.

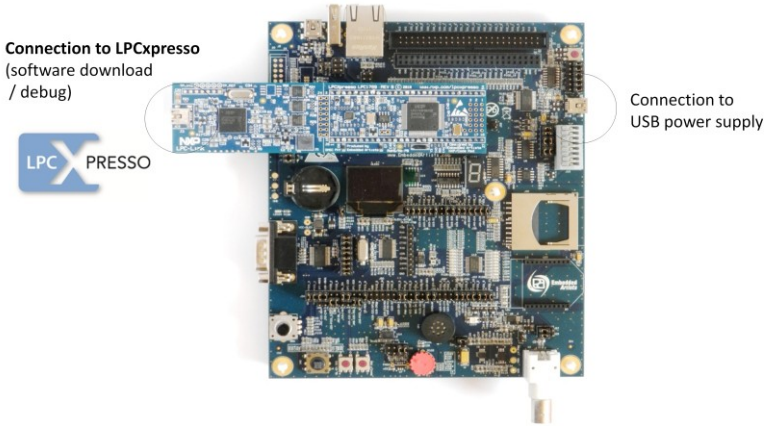


Figure 124: The LPCXpresso board and EA Baseboard connected together.

Figure 125 shows the connections for the jumper (wire) required to enabled the WDT, as discussed in Chapter 2.

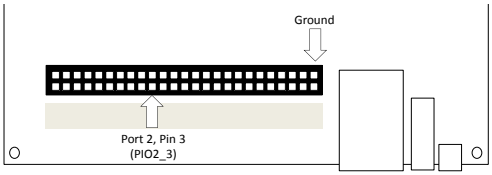


Figure 125: Jumper connections on the EA Baseboard that can be used to enable WDT support in most of the TTRDs.

A1.7. Running TTRD05a

Please note that to run TTRD05a (and other reference designs with a USB interface) you will need to make three USB connections to the boards, as shown in Figure 126.

You will also need to run a terminal emulator (such as "CoolTerm") on a laptop or desktop PC (or equivalent) running Windows®: see Figure 127.

In order to connect to the board from CoolTerm (or similar):

1. Copy the usbser.inf file (included with TTRD05a) to a temporary location.
2. Plug the USB cable into the USB device port on the baseboard
3. When requested (after downloading and starting execution of the example) direct Windows to the temporary directory containing the usbser.inf file. Windows then creates an extra COMx port that you can open in your terminal emulator.

When configuring your terminal program, you should set it to append line feeds to incoming line ends, and to echo typed characters locally. You will then be able to see the character you type, as well as the characters returned by the example running on the board.

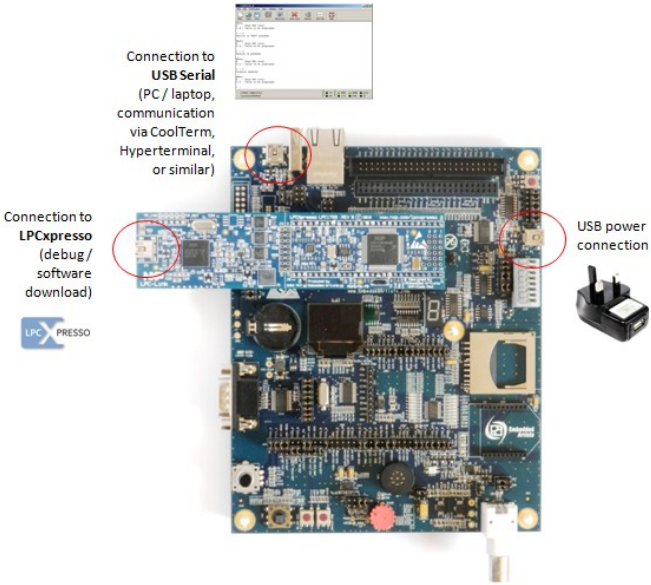


Figure 126: USB connections required to run TTRD05a.

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

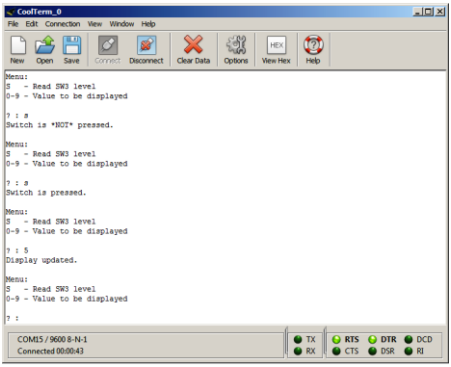


Figure 127: Use of CoolTerm with TTRD05a.

A1.8. Conclusions

This concludes our review of the LPC1769 hardware platforms used to support the various TTRDs that are described in this book.

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

Full list of references and related publications

Ahmad, N. and Pont, M.J. (2009) "Remote debugging of embedded systems which employ a time-triggered architecture", Proceedings of the 5th UK Embedded Forum, Leicester, UK, 23-24 September, 2009, pp.97-106. Published by Newcastle University. ISBN 978-0-7017-0222-9.

Ahmad, N. and Pont, M.J. (2010) "Debugging remote embedded systems: The impact of system software architecture", Proceedings of the 2010 UK Electronics Forum, Newcastle, UK, 30 June-1 July, 2010, pp.17-23. Published by Newcastle University. ISBN 978-0-7017-0232-8.

Amir, M. and Pont, M.J. (2008) "Synchronising tasks in wireless multi-processor environments using a shared-clock architecture: A pilot study", Proceedings of the 4th UK Embedded Forum, Southampton, UK, 9-10 September, 2008, pp.30-39. Published by IET. ISBN 978-0-8634-1949-2.

Amir, M. and Pont, M.J. (2009) "A time-triggered communication protocol for CAN-based networks with a star topology", Proceedings of the 5th UK Embedded Forum, Leicester, UK, 23-24 September, 2009, pp.30-44. Published by Newcastle University. ISBN 978-0-7017-0222-9.

Amir, M. and Pont, M.J. (2010) "A novel shared-clock scheduling protocol for fault-confinement in CAN-based distributed systems", Proceedings of the 5th IEEE International Conference on System of Systems, IEEE SoSE 2010, Loughborough University, UK, 22nd-24th, June 2010.

Amir, M. and Pont, M.J. (2010) "A time-triggered communication protocol for CAN-based networks with a fault-tolerant star topology ", Proceedings of the International Symposium on Advanced Topics in Embedded Systems and Applications under the 7th IEEE International Conference on Embedded Software and Systems, Bradford, West Yorkshire, UK, 29th June-1st July, 2010. ISBN 978-0-7695-4108-2.

Amir, M. and Pont, M.J. (2010) "Integration of TTC-SC5 and TTC-SC6 shared-clock protocols", Proceedings of the 1st UK Electronics Forum, Newcastle, UK, 30th June-1st July, 2010. Published by Newcastle University. ISBN 978-0-7017-0232-8.

Amir, M. and Pont, M.J. (2011) "Comments on: Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems", *Microprocessors and Microsystems*, Vol. 35, pp.81-82.

Amir, M. and Pont, M.J. (2013) "Improving flexibility and fault-management in CAN-based 'Shared-Clock' architectures", *Microprocessors and Microsystems – Embedded Hardware Design* 37(1): 9-23.

Athaide, K., Hughes, Z.M. and Pont, M.J. (2007) "Towards a time-triggered processor", Proceedings of the 3rd UK Embedded Forum, Durham, UK, 203 April, 2007, pp.166. Published by IET. ISBN 9780863418037. ISSN 0537-9989.

Athaide, K.F., Pont, M.J. and Ayavoo, D. (2008) "Deploying a time-triggered shared-clock architecture in a multiprocessor system-on-chip design", in Proceedings of the 4th UK Embedded Forum (September 2008, Southampton, UK).

Athaide, K.F., Pont, M.J. and Ayavoo, D. (2008) "Shared-clock methodology for time-triggered multi-cores", in Susan Stepney, Fiona Polack, Alistair McEwan, Peter Welch, and Wilson Ifill (Eds.), *"Communicating Process Architectures 2008",* IOS Press.

Ayavoo, D., Pont, M.J. and Parker, S. (2004) "Using simulation to support the design of distributed embedded control systems: A case study". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.54-65. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Ayavoo, D., Pont, M.J. and Parker, S. (2005) "Observing the development of a reliable embedded system". In Vardanega, T and Wellings, A. (Eds.) "Proceedings of the 10th Ada-Europe International Conference on Reliable Software Technologies, York, UK, June 20-24 2005", pp. 167-179. Lecture Notes in Computer Science, Vol. 3555. Published by Springer-Verlag [ISBN: 3-540-26286-5]

Ayavoo, D., Pont, M.J. and Parker, S. (2006) "Does a 'simulation first' approach reduce the effort involved in the development of distributed embedded control systems?". Proceedings of the 6th UKACC International Control Conference, Glasgow, Scotland, 2006.

Ayavoo, D., Pont, M.J., Fang, J., Short, M. and Parker, S. (2005) "A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.60-90. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Ayavoo, D., Pont, M.J., Short, M. and Parker, S. (2005) "Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.246-261. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Ayavoo, D., Pont, M.J., Short, M. and Parker, S. (2007) "Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems", *Microprocessors and Microsystems*, **31**(5): 326-334.

Bautista, R., Pont, M.J. and Edwards, T. (2005) "Comparing the performance and resource requirements of 'PID' and 'LQR' algorithms when used in a practical embedded control system: A pilot study". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.262-289. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Bautista-Quintero, R. and Pont, M.J. (2006) "Is fuzzy logic a practical choice in resource-constrained embedded control systems implemented using general-purpose microcontrollers?", Proceedings of the 9th IEEE International Workshop on Advanced Motion Control (Istanbul, March 27-29, 2006), Volume 2, pp.692-697. IEEE catalog number 06TH8850. ISBN 0-7803-9511-5.

Bautista-Quintero, R. and Pont, M.J. (2008) "Implementation of H-infinity control algorithms for sensor-constrained mechatronic systems using low-cost microcontrollers", *IEEE Transactions on Industrial Informatics*, 16(4): 175-184.

Chan, K.L. and Pont, M.J. (2010) "Real-time non-invasive detection of timing-constraint violations in time-triggered embedded systems", Proceedings of the 7th IEEE International Conference on Embedded Software and Systems, Bradford, UK, 2010, pp.1978-1986. Published by IEEE Computer Society. ISBN 978-0-7695-4108-2.

Edwards, T., Pont, M.J., Scotson, P. and Crumpler, S. (2004) "A test-bed for evaluating and comparing designs for embedded control systems". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.106-126. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Edwards, T., Pont, M.J., Short, M.J., Scotson, P. and Crumpler, S. (2005) "An initial comparison of synchronous and asynchronous network architectures for use in embedded control systems with duplicate processor nodes". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.290-303. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Gendy, A.K. and Pont, M.J. (2007) "Towards a generic 'single-path programming' solution with reduced power consumption", Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), September 4-7, 2007, Las Vegas, Nevada, USA.

Gendy, A.K. and Pont, M.J. (2008) "Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems", *IEEE Transactions on Industrial Informatics*, **4**(1): 37-46.

Gendy, A.K. and Pont, M.J. (2008) "Automating the processes of selecting an appropriate scheduling algorithm and configuring the scheduler implementation for time-triggered embedded systems", Proceedings of The 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP08), 22-25 September 2008, Newcastle upon Tyne, UK

Gendy, A.K., Dong, L. and Pont, M.J. (2007) "Improving the performance of time-triggered embedded systems by means of a scheduler agent", Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), September 4-7, 2007, Las Vegas, Nevada, USA.

Goble, W.M. and Cheddie, H. (2005) "Safety Instrumented Systems Verification: Practical probabilistic calculations", The Instrumentation, Systems and Automation Society. ISBN: 978-1-55617-909-9.

Hanif, M., Pont, M.J. and Ayavoo, D. (2008) "Implementing a simple but flexible time-triggered architecture for practical deeply-embedded applications", in Proceedings of the 4th UK Embedded Forum (September 2008, Southampton, UK).

Hughes, Z.M. and Pont, M.J. (2004) "Design and test of a task guardian for use in TTCS embedded systems". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.16-25. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Hughes, Z.M. and Pont, M.J. (2008) "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed", Transactions of the Institute of Measurement and Control, Vol. 30: pp.427-450.

Hughes, Z.M., Pont, M.J. and Ong, H.L.R. (2005) "Design and evaluation of a "time-triggered" microcontroller". Poster presentation at DATE 2005 (PhD Forum), Munich, Germany, March 2005.

Hughes, Z.M., Pont, M.J. and Ong, H.L.R. (2005) "The PH Processor: A soft embedded core for use in university research and teaching". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.224-245. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Imran, S., Short, M. and Pont, M.J. (2008) "Hardware implementation of a shared-clock scheduling protocol for CAN: A pilot study", in Proceedings of the 4th UK Embedded Forum (September 2008, Southampton, UK).

Key, S. and Pont, M.J. (2004) "Implementing PID control systems using resource-limited embedded processors". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.76-92. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Key, S.A., Pont, M.J. and Edwards, S. (2004) "Implementing low-cost TTCS systems using assembly language". In: Henney, K. and Schutz, D. (Eds) Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 8), Germany, June 2003: pp.667-690. Published by Universitätsverlag Konstanz. ISBN 3-87940-788-6.

Knight, J.C. and Leveson, N.G. (1985) "A Large Scale Experiment In N-Version Programming", Digest of Papers FTCS-15: Fifteenth International Symposium on Fault-Tolerant Computing, June 1985, Ann Arbor,MI. pp. 135-139.

Knight, J.C. and Leveson, N.G. (1986) "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming", IEEE Transactions on SoftwareEngineering,Vol. SE-12, No. 1 (January 1986), pp. 96-109.

Knight, J.C. and Leveson, N.G. (1990) "A reply to the criticisms of the Knight & Leveson experiment", ACM SIGSOFT Software Engineering Notes 15 (1), pp. 24-35.

Koelmans, A., Bystrov, A. and Pont, M.J. (2004)(Eds.) "Proceedings of the First UK Embedded Forum" (Birmingham, UK, October 2004). Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (2005)(Eds.) "Proceedings of the Second UK Embedded Forum" (Birmingham, UK, October 2005). Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Kurian, S. and Pont, M.J. (2005) "Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.36-59. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Kurian, S. and Pont, M.J. (2005) "Mining for pattern implementation examples". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.194-201. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Kurian, S. and Pont, M.J. (2006) "Evaluating and improving pattern-based software designs for resource-constrained embedded systems". In: C. Guedes Soares & E. Zio (Eds), "Safety and Reliability for Managing Risk: Proceedings of the 15th European Safety and Reliabilty Conference (ESREL 2006), Estoril, Portugal, 18-22 September 2006", Vol. 2, pp.1417-1423. Published by Taylor and Francis, London. ISBN: 0-415-41620-5 (for complete 3-volume set of proceedings). ISBN: 978-0-415-42314-4 (for Volume 2).

Kurian, S. and Pont, M.J. (2007) "Maintenance and evolution of resource-constrained embedded systems created using design patterns", *Journal of Systems and Software*, **80**(1): 32-41.

Lakhani, F. and Pont, M.J. (2010) "Using design patterns to support migration between different system architectures", Proceedings of the 5th IEEE International Conference on Systems of Systems Engineering, June 2010, Loughborough, UK.

Lakhani, F., and Pont, M.J. (2010) "Code balancing as a philosophy for change: Helping developers to migrate from event-triggered to time-triggered architectures" Proceedings of First UK Electronics Forum, 30June- 1 July 2010, Newcastle, UK , Published by Newcastle University. ISBN: 978-0-7017-0232-8

Lakhani, F., and Pont, M.J. (2012) "Applying design patterns to improve the reliability of embedded systems through a process of architecture migration", HPCC-ICESS 2012: 1563-1570.

Lakhani, F., Pont, M.J. and Das, A. (2009) "Can we support the migration from event triggered to time triggered architectures using design patterns?" Proceedings of 5th UK Embedded Forum, Leicester, UK , pp. 62-67. Published by Newcastle University. ISBN: 978-0-7017-0222-9

Lakhani, F., Pont, M.J. and Das, A. (2009) "Towards a pattern language which supports the migration of systems from an event-triggered pre-emptive to a time-triggered co-operative software architecture" Proceedings of 14th Annual European Conference on Pattern Languages of Programming", Irsee, Germany, 8-12 July 2009. published by CEUR vol. 566 . ISSN : 1613-0073

Lakhani, F., Pont, M.J. and Das, A. (2010) "Creating embedded systems with predictable patterns of behaviour, supporting the migration between event-triggered and time-triggered software architectures" Proceedings of 15th Annual European Conference on Pattern Languages of Programming" , Irsee, Germany, 7 -11 July 2010.

Li, Y. and Pont, M.J. (2002) "On selecting pre-processing techniques for fault classification using neural networks: A pilot study", *International Journal of Knowledge-Based Intelligent Engineering Systems,* **6**(2): 80-87.

Li, Y., Pont, M.J. and Jones, N.B. (2002) "Improving the performance of radial basis function classifiers in condition monitoring and fault diagnosis applications where 'unknown' faults may occur", *Pattern Recognition Letters*, **23**: 569-577.

Li, Y., Pont, M.J., and Jones, N.B. (1999) "A comparison of the performance of radial basis function and multi-layer Perceptron networks in a practical condition monitoring application", Proceedings of Condition Monitoring 1999 [Swansea, UK, April 12-15, 1999] pp.577-592.

Li, Y., Pont, M.J., Jones, N.B. and Twiddle, J.A. (2001) "Using MLP and RBF classifiers in embedded condition monitoring and fault diagnosis applications", *Transactions of the Institute of Measurement & Control,* **23**(3): 313-339.

Li, Y., Pont, M.J., Parikh, C.R. and Jones, N.B. (2000) "Comparing the performance of three neural classifiers for use in embedded applications", in: John, R. and Birkenhead, R. (Eds.) *Advances in Soft Computing: Soft Computing Techniques and Applications*, Springer-Verlag, Heidelberg, pp.34-39, [ISBN 3-7908-1257-9]

Li, Y.H., Jones, N.B. and Pont, M.J. (1998) "Applying neural networks and fuzzy logic to fault diagnosis: a review". in John, R.I. (1998), Editor, "Proceedings of: 'Recent Advances in Soft Computing '98" [Leicester, July 1998] pp.104-119. Published by DeMontfort Expertise, Leicester, UK [ISBN 185 721 2592].

Liu, JWS (2000) "Real-time systems", Prentice Hall, Upper Saddle River, New Jersey. ISBN 0-013- 0996513

Maaita, A. (2008) "Techniques for enhancing the temporal predictability of real-time embedded systems employing a time-triggered software architecture", PhD thesis, University of Leicester (UK).

Maaita, A. and Pont, M.J. (2005) "Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.18-35. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Mearns, D.D.U., Pont, M.J. and Ayavoo, D. (2008) "Towards Ctt (a programming language for time-triggered embedded systems)", in Proceedings of the 4th UK Embedded Forum (September 2008, Southampton, UK).

Meyer, B. (1997) "Object-Oriented Software Construction", 2nd Edition. Prentice-Hall. ISBN:0-13-629155-4.

MISRA (2012) "Guidelines for the Use of the C Language in Critical Systems", ISBN 978-1-906400-10-1.

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.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D. (2007) "Rapid software development for reliable embedded systems using a pattern-based code generation tool". *SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems)*, **115**(7): 795-803.

Mwelwa, C., Pont, M.J. and Ward, D. (2004) "Code generation supported by a pattern-based design methodology". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.36-55. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Mwelwa, C., Pont, M.J. and Ward, D. (2004) "Using patterns to support the development and maintenance of software for reliable embedded systems: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-Chip Design, Test and Technology", Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989), pp. 15-20.

Mwelwa, C., Pont, M.J. and Ward, D. (2005) "Developing reliable embedded systems using a pattern-based code generation tool: A case study". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.177-193. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

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

Nahas, M., Pont, M.J. and Jain, A. (2004) "Reducing task jitter in shared-clock embedded systems using CAN". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK,

October 2004), pp.184-194. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Nahas, M., Pont, M.J. and Short, M.J. (2009) "Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol", Journal of Systems Architecture Vol. 55: pp.344–354.

Nahas, M., Short, M.J. and Pont, M.J. (2005) "Exploring the impact of software bit stuffing on the behaviour of a distributed embedded control system implemented using CAN", in Proceedings of the 10th international CAN Conference, held in Rome, 8-10 March 2005, pp. 10-1 to 10-7.

Ong, H.L.R and Pont, M.J. (2001) "Empirical comparison of software-based error detection and correction techniques for embedded systems", Proceedings of the 9th International Symposium on Hardware / Software Codesign, April 25-27 2001, Copenhagen, Denmark. Pp.230-235. Published by ACM Press, New York. ISBN: 1-58113-364-2.

Ong, H.L.R and Pont, M.J. (2002) "The impact of instruction pointer corruption on program flow: a computational modelling study", *Microprocessors and Microsystems*, **25**: 409-419

Ong, H.L.R, Pont, M.J. and Peasgood, W. (2001) "A comparison of software-based techniques intended to increase the reliability of embedded applications in the presence of EMI" *Microprocessors and Microsystems*, **24**(10): 481-491.

Ong, H.L.R., Pont, M.J., and Peasgood, W. (2000) "Hardware-software tradeoffs when designing microcontroller-based applications for high-EMI environments"**,** IEE Colloquium on Hardware-Software Co-Design, Savoy Place, London, 8 December, 2000. IEE Colloquium Digests #111.

Parikh, C.R., Pont, M.J., Li, Y. and Jones, N.B. (1999) "Neural networks for condition monitoring and fault diagnosis: The effect of training data on classifier performance", Proceedings of Condition Monitoring 1999 [Swansea, UK, April 1999] pp.237-244.

Parikh, C.R., Pont, M.J., Li, Y.H., Jones, N.B. and Twiddle, J.A. (1998) "Towards a flexible application framework for data fusion using real-time design patterns," Proceedings of 6th European Congress on Intelligent Techniques & Soft Computing (EUFIT), Aachen, Germany, September 7-10, 1998. pp.1131-1135.

Parikh C.R., M.J. Pont and N.B. Jones (2001) "Application of Dempster-Shafer theory in condition monitoring systems", *Pattern Recognition Letters*, **22**(6-7): 777-785.

Parikh C.R., Pont, M.J., Jones, N.B. and Schlindwein, F.S. (2003) "Improving the performance of CMFD applications using multiple classifiers and a fusion framework", *Transactions of the Institute of Measurement and Control*, **25**(2): 123-144.

Phatrapornnant, T. and Pont, M.J. (2004) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture: A case study", Proceedings of the IEE / ACM Postgraduate Seminar on "System-On-

Chip Design, Test and Technology", Loughborough, UK, 15 September 2004. Published by IEE. ISBN: 0 86341 460 5 (ISSN: 0537-9989), pp. 3-8.

Phatrapornnant, T. and Pont, M.J. (2004) "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.127-143. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Phatrapornnant, T. and Pont, M.J. (2006) "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling", *IEEE Transactions on Computers*, **55**(2): 113-124.

Pont, M.J. (1996) *"Software Engineering with C++ and CASE Tools"*, Addison-Wesley [964 pages]. ISBN: 0-201-87718-X.

Pont, M.J. (1998) "Control system design using real-time design patterns", Proceedings of Control '98 (Swansea, UK), September 1998, pp.1078-1083.

Pont, M.J. (2000) "Can patterns increase the reliability of embedded hardware-software co-designs?", IEE Colloquium on Hardware-Software Co-Design, Savoy Place, London, 8 December, 2000. IEE Colloquium Digests #111.

Pont, M.J. (2000) "Designing and implementing reliable embedded systems using patterns", in, Dyson, P. and Devos, Martine (Eds.) *"EuroPLoP '99: Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999"*. ISBN 3-87940-774-6, Universitätsverlag Konstanz.

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. (2002) *"Embedded C"*, Addison-Wesley. ISBN: 0-201-79523-X.

Pont, M.J. (2003) "An object-oriented approach to software development for embedded systems implemented using C", *Transactions of the Institute of Measurement and Control* **25**(3): 217-238.

Pont, M.J. (2003) "Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns", *Informatica*, **27**: 81-88.

Pont, M.J. (2008) "Applying time-triggered architectures in reliable embedded systems: Challenges and solutions", Elektrotechnik & Informationstechnik, Vol. 125(11): 401-405.

Pont, M.J. (2015) *"The Engineering of Reliable Embedded Systems: TMS570 edition"*, SafeTTy Systems Ltd. ISBN: 978-0-9930355-1-7. *In press*.

Pont, M.J. and Banner, M.P. (2004) "Designing embedded systems using patterns: A case study", *Journal of Systems and Software*, **71**(3): 201-213.

Pont, M.J. and Mwelwa, C. (2003) "Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language" Paper presented at the Second *Nordic Conference on Pattern Languages of Programs,* ("VikingPLoP 2003"), Bergen, Norway, September 2003.

Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", in Hruby, P. and Soressen, K. E. [Eds.] *Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002 ("VikingPloP 2002")*, pp.159-200. Published by Micrsoft Business Solutions. ISBN: 87-7849-769-8.

Pont, M.J., Kureemun, R., Ong, H.L.R. and Peasgood, W. (1999) "Increasing the reliability of embedded automotive applications in the presence of EMI: A pilot study", IEE Colloquium on Automotive EMC, Birmingham, September 28, 1999.

Pont, M.J., Kurian, S. and Bautista-Quintero, R. (2007) "Meeting real-time constraints using 'Sandwich Delays'". In: Zdun, U. and Hvatum, L. (Eds) Proceedings of the Eleventh European conference on Pattern Languages of Programs (EuroPLoP '06), Germany, July 2006: pp.67-77. Published by Universitätsverlag Konstanz. ISBN 978-3-87940-813-9.

Pont, M.J., Kurian, S., Wang, H. and Phatrapornnant, T. (2007) "Selecting an appropriate scheduler for use with time-triggered embedded systems" Paper presented at the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007).

Pont, M.J., Li, Y., Parikh, C.R. and Wong, C.P. (1999) "The design of embedded systems using software patterns", Proceedings of Condition Monitoring 1999 [Swansea, UK, April 12-15, 1999] pp.221-236.

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T. (2004) "Prototyping time-triggered embedded systems using PC hardware". In: Henney, K. and Schutz, D. (Eds) Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 8), Germany, June 2003: pp.691-716. Published by Universitätsverlag Konstanz. ISBN 3-87940-788-6.

Puschner, P. and Burns, A., 2002, "Writing Temporally Predictable Code", In Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, Jan. 2002, pp. 85–91.

Rizvi, S.A.I. and Pont, M.J. (2009) "Hardware support for deadlock-free resource sharing in an embedded system with a TT architecture", Proceedings of 5th UK Embedded Forum 2009, University of Leicester, UK, 23-24 September, pp. 1-9. Published by Newcastle University. ISBN: 978-0-7017-0222-9

Short, M. and Pont, M.J. (2005) "Hardware in the loop simulation of embedded automotive control systems", in Proceedings of the 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005) held in Vienna, Austria, 13-16 September 2005, pp. 226-231.

Short, M. and Pont, M.J. (2006) "Predicting the impact of hardware redundancy on the performance of embedded control systems". Proceedings of the 6th UKACC International Control Conference, Glasgow, Scotland, 30 August to 1 September, 2006.

Short, M. and Pont, M.J. (2007) "Fault-tolerant time-triggered communication using CAN", *IEEE Transactions on Industrial Informatics*, **3**(2): 131-142.

Short, M. and Pont, M.J. (2008) "Assessment of high-integrity embedded automotive control systems using Hardware-in-the-Loop simulation", *Journal of Systems and Software*, **81**(7): 1163-1183.

Short, M., Fang, J., Pont, M.J. and Rajabzadeh, A. (2007) "Assessing the impact of redundancy on the performance of a brake-by-wire system". *SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems)*, **115**(7): 331-338.

Short, M., Pont, M.J. and Fang, J. (2008) "Assessment of performance and dependability in embedded control systems: Methodology and case study", *Control Engineering Practice.* Vol. **16**, pp.1293– 1307

Short, M., Pont, M.J. and Fang, J. (2008) "Exploring the impact of task pre-emption on dependability in time-triggered embedded systems: A pilot study", Proceedings of the 20th EUROMICRO Conference on Real-Time Systems (ECRTS 08), Prague, Czech Republic, July 2nd – 4th, 2008

Suk, D.S. and Reddy, S.M. "A March Test for Functional Faults in Semiconductor Random-Access Memories", lEEE Trans. Computers, Vol. C-30, No. 12, 1981, pp. 982-985

Vidler, P.J. and Pont, M.J. (2005) "Automatic conversion from 'single processor' to 'multi-processor' software architectures for embedded control systems". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.209-223. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Vidler, P.J. and Pont, M.J. (2006) "Computer assisted source-code parallelisation". In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganà, A., Mun, Y. and Choo, H. (eds.) Proceedings of the IEE International Conference on Computational Science and its Applications (Glasgow, May 8-11, 2006), Part V. Lecture Notes in Computer Science (LNCS), Vol. 3984, pp.22-31.

Vladimirova, T, Bannister, N., Fothergill, J, Fraser, G, Lester, M., Wright, D, Pont, M.J., Barnhart, D.J. anf Emam, O., "CubeSat mission for space weather monitoring", Proceedings of 11th Australian Space Science Conference, ASSC'11, 26 – 29 September 2011, Canberra, Australia.

Wang, H. and Pont, M.J. (2008) "Design and implementation of a static pre-emptive scheduler with highly predictable behaviour", in Proceedings of the 4th UK Embedded Forum (September 2008, Southampton, UK).

Wang, H., Pont, M.J. and Kurian, S. (2007) "Patterns which help to avoid conflicts over shared resources in time-triggered embedded systems which employ a pre-emptive scheduler" Paper presented at the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007).

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

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

Index

1oo1d 314

1oo2p 312

asynchronous task set 95, 253, 254, 256

backup task 188, 189, 290, 296, 297, 298, 299

Best-Case Execution Time 22, 113, 137, 283

buffered input 135

buffered output 129

Built In Self Test 290, 295, 323

Bus Guardian **344**

cattle prod 133

chipping cars 294

code balancing 281, 283, 284

common-cause failures 312

Controlled System Failure **xv**, 153

controller - central heating 133

controller - washing machine 63, 317

Controller Area Network 272, 312

Delete Task 26

Deliberate Hardware Change **xvi**

Deliberate Software Change **xvi**

Direct Memory Access 131, 136, 266

distributed system 344

DO-178C xxi

Dry Scheduler 121, 252, 255

EA Baseboard 28, 66, 296, 348, 349, 350, 351

emergency stop 107, 126, 182, 269

execution-time balancing 101, 138, 139, 140, 141, 142

Fail-Silent Mode **xv**

GPREG 151

graceful degradation 153, 154, 238

Greatest Common Divisor 94, 95

Hardware Error **xvi**

Hardware Fault **xvi**

hardware timeout 181

hyperperiod 20, 95, 249, 253, 255

IEC 60335 xxi, 292, 295, 319, 320

IEC 60601 xxi, 270, 271, 293

IEC 60601-1-8 270

IEC 60730 xxi, 314, 319, 320, 340

IEC 61508 xxi, 14, 189, 190, 301, 314

IEC 62304 xxi, 293

infusion pump 293

Internet of Things 344

inverted copy 152

Inverted copy 152

ISO 26262 xxi, 14, 190, 257, 301, 314

jitter 9, 26, 100, 101, 102, 113, 117, 137, 147, 266, 268, 271, 280

Limp-Home Mode **xv**

Lowest Common Multiple 95

LPC1769 xxvi, 271, 289, 292, 294, 295, 296, 303, 343, 347, 348, 353

marching algorithm 291

MAX6576 (temperature sensor) 296

maximum CPU load 98

medical alarm 270

Memory Protection Unit xxvi, 187, 256, 306, 347

MISRA C xxi, xxv, 7, 68

Model-Build-Monitor 12, 343

modelling CPU loading 97

modes vs. states 148

MoniTTor 181, **236**, 237, 238, 239, 241, 249, 250, 251, 252, 274, 275, 276, 277

multi-core 136

Multi-core input 136

multi-mode system - architecture 150

Multiple-Input Signature Register 294

N-Version programming 298

NVRAM 151, 152

Platform TT00 132, **303**

Platform TT01 **304**

Platform TT02 **304**

Platform TT03 251, **305**, 320

Platform TT04 295, **307**

Platform TT05 **309**, 311, 344

Port Header 36

Power-On Self Test 290, 291, 292, 293, 294, 323

predictive monitor 14, 249, 251

PredicTTor **249**, 250, 251, 252, 253, 310

Priority 0 task **268**, 269, 273

Priority 1 task **268**, 269, 273

Priority 2 task **273**

priority inversion 13, 139, 272, 273, 281, 284

Project Header 35

PTTES xxvi, 17, 109, 130, 344

Resource Barrier 13, 177, 186, 188, 237, 238, 249, 250, 251, 252

Resource Barrier (lightweight) 187

response time 9, 105, 106

RS-422 312

RS-485 312

Sandwich Delay 141, 142

SCH_MAX_TASKS 21

shared resource 13, 26, 136, 185, 269, 272, 273, 274, 281, 284

Shared-Clock scheduler 344

Short Task 94, 126

Software Error **xvi**

Software Fault **xvi**

Steer-by-Wire application 344

steering-column lock 12

synchronous task set 120, 253, 254

system configuration 293, 294, 295

System Contract 190, 289, 290, 296, 299, 302

System Damage **xvi**

System Fault **xvi**

System Hardware **xvi**

System Software **xvi**

system under control 311

Task Contract 13, 155, 177, 178, 180, 182, 186, 189, 190, 235, 236, 289, 299

Task Guardian 29, **241**

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

task offset 96

task offset - adjusting 123

task overrun 20, 29

task release jitter 104

Task Sequence Initialisation Period 97, 122, 253, 254, 255, 256

Task Sequence Representation 252, 256

Tick jitter 117

Tick List 8, 9, 93, 111, 269, 305

Tick List - how to create 121

Time Barrier 235, 237

timeout 297

timeout - hardware 135

timeout - loop 134

timeout mechanisms 131

TT00 **303**

TT01 **304**

TT02 **304**

TT03 **305**

TT04 **307**

TT05 309

TTC designs - advantages of 114

TTC scheduler 9, 17, 21, 93, 266, 267, 271, 274, 275, 289, 305

TTH scheduler 10, 33, 267, 268, 269, 275, 280, 282, 283

TTP scheduler 33, 305

UART 135

Uncontrolled System Failure **xv**, 12, 14, 63, 153, 177, 189, 343

WarranTTor 155, 304, **310**, 312, 315, 340, 344

watchdog timer 18, 28, 29, 66, 277, 314, 351

watchdog timer - checking the configuration 184

watchdog timer in TTC design 28

watchdog timer in TTH design 277

watchdog timer in TTP design 277

WCET measurement - challenges 114

Worst-Case Execution Time 22, 113, 137, 138, 283, 297

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