

REAL-TIME EMBEDDED COMPONENTS AND SYSTEMS *WITH LINUX AND RTOS*



INCLUDES DVD

SAM SIEWERT AND JOHN PRATT

**REAL-TIME
EMBEDDED
COMPONENTS
AND SYSTEMS**
with LINUX and RTOS

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the “Work”), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

MERCURY LEARNING AND INFORMATION (“MLI” or “the Publisher”) and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs (“the software”), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold “as is” without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, and only at the discretion of the Publisher. The use of “implied warranty” and certain “exclusions” vary from state to state, and might not apply to the purchaser of this product.

**REAL-TIME
EMBEDDED
COMPONENTS
AND SYSTEMS**
with LINUX and RTOS

**Sam Siewert
John Pratt**



MERCURY LEARNING AND INFORMATION

Dulles, Virginia
Boston, Massachusetts
New Delhi

Copyright ©2016 by MERCURY LEARNING AND INFORMATION LLC. All rights reserved.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai
MERCURY LEARNING AND INFORMATION
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

S. Siewert and J. Pratt. *Real-Time Embedded Components and Systems with LINUX and RTOS*.
ISBN: 978-1-942270-04-1

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2015944870

151617321 This book is printed on acid-free paper.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc.
For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at authorcloudware.com and other digital vendors. Companion files (figures and code listings) for this title are available by contacting info@merclearning.com. The sole obligation of MERCURY LEARNING AND INFORMATION to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

CONTENTS

<i>Preface</i>	<i>xi</i>
<i>Acknowledgments</i>	<i>xv</i>

Part I: Real-Time Embedded Theory

Chapter 1 Introduction	3
1.1 Introduction	3
1.2 A Brief History of Real-Time Systems	4
1.3 A Brief History of Embedded Systems	7
1.4 Real-Time Services	7
1.5 Real-Time Standards	16
Summary	18
Exercises	18
Chapter References	19
Chapter 2 System Resources	21
2.1 Introduction	21
2.2 Resource Analysis	23
2.3 Real-Time Service Utility	31
2.4 Scheduling Classes	38
2.4.1 Multiprocessor Systems	39
2.5 The Cyclic Executive	39
2.6 Scheduler Concepts	41
2.6.1 Preemptive vs. Non-preemptive Schedulers	44
2.6.2 Preemptive Fixed-Priority Scheduling Policy	47
2.7 Real-Time Operating Systems	50
2.8 Thread-Safe Reentrant Functions	59
Summary	61
Exercises	62
Chapter References	64
Chapter 3 Processing	67
3.1 Introduction	67
3.2 Preemptive Fixed-Priority Policy	68
3.3 Feasibility	72
3.4 Rate-Monotonic Least Upper Bound	73
3.5 Necessary and Sufficient Feasibility	84
3.5.1 Scheduling Point Test	84
3.5.2 Completion Time Test	87

3.6	Deadline-Monotonic Policy	89
3.7	Dynamic-Priority Policies	91
	Summary	95
	Exercises	96
	Chapter References	98
Chapter 4	Resources	99
4.1	Introduction	99
4.2	Worst-Case Execution Time	100
4.3	Intermediate IO	104
4.4	Execution Efficiency	107
4.5	IO Architecture	110
	Summary	111
	Exercises	112
	Chapter References	113
Chapter 5	Memory	115
5.1	Introduction	115
5.2	Physical Hierarchy	116
5.3	Capacity and Allocation	120
5.4	Shared Memory	120
5.5	ECC Memory	121
5.6	Flash File Systems	126
	Summary	127
	Exercises	128
	Chapter References	128
Chapter 6	Multiresource Services	129
6.1	Introduction	129
6.2	Blocking	130
6.3	Deadlock and Livelock	130
6.4	Critical Sections to Protect Shared Resources	132
6.5	Priority Inversion	132
6.5.1	Unbounded Priority Inversion Solutions	134
6.6	Power Management and Processor Clock Modulation	138
	Summary	139
	Exercises	139
	Chapter References	140
Chapter 7	Soft Real-Time Services	141
7.1	Introduction	141
7.2	Missed Deadlines	142
7.3	Quality of Service	143
7.4	Alternatives to Rate-Monotonic Policy	144

7.5	Mixed Hard and Soft Real-Time Services	147
	Summary	148
	Exercises	148
	Chapter References	149

Part II: Designing Real-Time Embedded Components

Chapter 8 Embedded System Components 153

8.1	Introduction	153
8.2	Hardware Components	155
8.2.1	Sensors	155
8.2.2	Actuators	158
8.2.3	IO Interfaces	159
8.2.4	Processor Complex or SoC	163
8.2.5	Processor and IO Interconnection	164
8.2.6	Bus Interconnection	165
8.2.7	High-Speed Serial Interconnection	170
8.2.8	Low-Speed Serial Interconnection	172
8.2.9	Interconnection Systems	173
8.2.10	Memory Subsystems	174
8.3	Firmware Components	176
8.3.1	Boot Code	176
8.3.2	Device Drivers	177
8.3.3	Operating System Services	178
8.4	RTOS System Software	178
8.4.1	Message Queues	179
8.4.2	Binary Semaphores	180
8.4.3	Mutex Semaphores	181
8.4.4	Software Virtual Timers	181
8.4.5	Software Signals	182
8.5	Software Application Components	182
8.5.1	Application Services	182
8.5.2	Reentrant Application Libraries	185
8.5.3	Communicating and Synchronized Services	186
	Summary	188
	Exercises	188
	Chapter References	189

Chapter 9 Traditional Hard Real-Time Operating Systems 191

9.1	Introduction	191
9.2	Evolution of Real-Time Scheduling and Resource Management	192
9.3	AMP (Asymmetric Multi-core Processing)	194
9.4	SMP (Symmetric Multi-core Processing)	199

9.5	Processor Core Affinity	200
9.6	Future Directions for RTOS	216
9.7	SMP Support Models	217
9.8	RTOS Hypervisors	218
	Summary	219
	Exercises	219
	Chapter References	220
Chapter 10	Open Source Real-Time Operating Systems	221
10.1	FreeRTOS Alternative to Proprietary RTOS	221
10.2	FreeRTOS Platform and Tools	225
10.3	FreeRTOS Real-Time Service Programming Fundamentals	228
	Exercises	235
	Chapter References	235
Chapter 11	Integrating Embedded Linux into Real-Time Systems	237
11.1	Introduction	237
11.2	Embedding Mainline Linux: Interactive and Best-Effort	238
11.3	Linux as a Non-Real-Time Management and User Interface Layer	244
11.4	Methods to Patch and Improve Linux for Predictable Response	245
11.5	Linux for Soft Real-Time Systems	249
11.6	Tools for Linux for Soft Real-Time Systems	249
	Summary	250
	Exercises	250
	Chapter References	251
Chapter 12	Debugging Components	253
12.1	Introduction	253
12.2	Exceptions	254
12.3	Assert	262
12.4	Checking Return Codes	263
12.5	Single-Step Debugging	264
12.6	Kernel Scheduler Traces	273
12.7	Test Access Ports	278
12.8	Trace Ports	280
12.9	Power-On Self-Test and Diagnostics	282
12.10	External Test Equipment	287
12.11	Application-Level Debugging	291
	Summary	292
	Exercises	293
	Chapter References	293

Chapter 13	Performance Tuning	295
13.1	Introduction	295
13.2	Basic Concepts of Drill-Down Tuning	296
13.3	Hardware-Supported Profiling and Tracing	300
13.4	Building Performance Monitoring into Software	304
13.5	Path Length, Efficiency, and Calling Frequency	305
13.6	Fundamental Optimizations	317
	Summary	318
	Exercises	318
	Chapter References	318
Chapter 14	High Availability and Reliability Design	319
14.1	Introduction	319
14.2	Reliability and Availability Similarities and Differences	320
14.3	Reliability	321
14.4	Reliable Software	323
14.5	Design Trade-Offs	324
14.6	Hierarchical Approaches for Fail-Safe Design	327
	Summary	327
	Exercises	327
	Chapter References	328
Part III: Putting it All Together		
Chapter 15	System Life Cycle	331
15.1	Introduction	331
15.2	Life Cycle Overview	332
15.3	Requirements	335
15.4	Risk Analysis	336
15.5	High-Level Design	336
15.6	Component Detailed Design	339
15.7	Component Unit Testing	348
15.8	System Integration and Test	357
15.9	Configuration Management and Version Control	357
15.10	Regression Testing	362
	Summary	362
	Exercises	362
	Chapter References	363
Chapter 16	Continuous Media Applications	365
16.1	Introduction	365
16.2	Video	367
16.3	Uncompressed Video Frame Formats	370
16.4	Video Codecs	371

16.5	Video Streaming	373
16.7	Video Stream Analysis and Debug	374
16.8	Audio Codecs and Streaming	378
16.9	Audio Stream Analysis and Debug	379
16.10	Voice-Over Internet Protocol (VoIP)	379
	Summary	381
	Exercises	381
	Chapter References	382
Chapter 17	Robotic Applications	383
17.1	Introduction	383
17.2	Robotic Arm	384
17.3	Actuation	387
17.4	End Effector Path	393
17.5	Sensing	393
17.6	Tasking	398
17.7	Automation and Autonomy	400
	Summary	401
	Exercises	402
	Chapter References	402
	Chapter Web References	402
Chapter 18	Computer Vision Applications	403
18.1	Introduction	403
18.2	Object Tracking	404
18.3	Image Processing for Object Recognition	406
18.4	Characterizing Cameras	409
18.5	Pixel and Servo Coordinates	411
18.6	Stereo-Vision	412
	Summary	413
	Exercises	414
	Chapter References	414
Appendix A	Terminology Glossary	417
Appendix B	About the DVD	455
Appendix C	Wind River Systems University Program for Workbench/VxWorks	459
Appendix D	Real-Time and Embedded Linux Distributions and Resources	461
Bibliography		463
Index		471

PREFACE

This book is intended to provide the practicing engineer with the necessary background to apply real-time theory to the design of embedded components and systems in order to successfully field a real-time embedded system. The book also is intended to provide a senior-year undergraduate or first-year graduate student in electrical engineering, computer science, or related fields of study with a balance of fundamental theory, review of industry practice, and hands-on experience to prepare for a career in the real-time embedded system industries. Typical industries include aerospace, medical diagnostic and therapeutic systems, telecommunications, automotive, robotics, industrial process control, media systems, computer gaming, and electronic entertainment, as well as multimedia applications for general-purpose computing. Real-time systems have traditionally been fielded as hard real-time applications, such as digital flight control systems, antilock braking systems, and missile guidance. More recently, however, intense interest in soft real-time systems has arisen due to the quickly growing market for real-time digital media services and systems.

This updated edition adds three new chapters focused on key technology advancements in embedded systems and with wider coverage of real-time architectures. The overall focus remains the RTOS (Real-Time Operating System), but use of Linux for soft real-time, hybrid FPGA (Field Programmable Gate Array) architectures and advancements in multi-core system-on-chip (SoC), as well as software strategies for asymmetric and symmetric multiprocessing (AMP and SMP) relevant to real-time embedded systems, has been added. Specifically, a new Chapter 9 provides an overview of RTOS advancements, including AMP and SMP configurations, with a discussion of future directions for RTOS use in multi-core architectures, such as SoC. A new Chapter 10 is devoted to open source RTOS, with emphasis on FreeRTOS. A new Chapter 11 is focused on methods to integrate embedded Linux into real-time embedded systems, with emphasis on soft real-time requirements, methods to patch and improve the Linux kernel for predictable response, and finally best practices for implementation of real-time services and applications that make use of POSIX real-time extensions in the 1003.1 2013 standard. The original Chapters 9, 10, and 11 have been preserved and are now Chapters 12 to 14, and Part III remains unchanged other than chapter renumbering to accommodate the insertion of the new chapters.

John Pratt, a new co-author, has contributed extensively to this edition, with specific focus on FreeRTOS, and brings a unique perspective to this updated version with his commercial mobile embedded systems expertise.

The new Linux examples and extended coverage of Linux in this edition are based upon a summer version of the course *Real-Time Embedded Systems* taught at the University of Colorado, Boulder, to offer an alternative to the traditional fall course that has used the Wind River VxWorks RTOS. The summer course has emphasized the same hard and soft real-time theory, but practice has focused on using Linux to achieve predictable response for systems that require real-time, but where occasional missed deadlines are not catastrophic. For example, mobile digital media, augmented reality applications, computer vision and digital entertainment and interactive systems. While hard real-time mission critical systems are still a major concern, many emergent applications require predictable response and simply need to provide high-quality of service. The use of buffering and time stamps to work ahead and provide high-quality presentation of results is, for example, a method used in digital video encode, transport, and decode, where the systems software is not required to provide deterministic proven hard real-time processing, as has been the goal for the RTOS. Likewise, many systems today use best-effort or soft real-time embedded Linux configurations with coprocessors, either FPGA or ASIC (Application Specific Integrated Circuits), that provide guaranteed hard real-time processing with hardware state machines.

Numerous improvements and corrections have been made to the original edition text to improve readability and clarity based on excellent feedback by many undergraduate and graduate students at the University of Colorado who have used the original edition text since August 2006.

While it's impossible to keep up with all the advancements related to real-time embedded systems, we hope the reader will find the new chapters and expanded example material included on the DVD a useful extension to traditional cyclic executive and RTOS real-time system components and systems architecture. The expanded guidelines and strategies are intended to help the practicing engineer and to introduce advanced undergraduate and graduate students in computer and software engineering disciplines to design strategies and methods to tackle many of the challenges found in real-time embedded systems. This challenging engineering area continues to evolve and relies upon the careful validation and verification efforts of practicing engineers to ensure and balance safety, cost, and capabilities of these complex and critical applications on which we all depend.

Companion Files

Companion files (figures and code listings) for this title are also available by contacting info@merclearning.com.

Sam Siewert
December 2015

ACKNOWLEDGMENTS

I would like to thank my wife, Kristy Klein, and family for enduring the toil associated with this updated edition, which really was started the moment the original edition was completed. I would also like to thank the numerous students at the University of Colorado, Boulder who have used this text in the course *Real-Time Embedded Systems* and have contributed many corrections and ideas for improvement and clarification, as well as students and faculty at other universities who have adopted the book and have been kind enough to share their experiences via e-mail. As an author, my main role has been accurately capturing and recording the suggestions for improvement while also working on extensions to keep up with advancing technology—the advancement has been rapid during the nine years since the original edition. As before, I would like to thank the many teaching assistants, too numerous to mention all by name here, but I feel lucky to have had the pleasure to work with all of them, all of whom have suggested improvements to example code, especially the new Linux code, and who have politely pointed out errors and omissions in the original text and supporting material. I would like to thank Professor Ruth Dameron at University of Colorado, Boulder for vastly improving the clarity and correctness of the new chapter drafts and dealing with my writing style, which sometimes diverges into a stream of consciousness or maybe even unconsciousness at times. I would like to thank Jennifer Blaney and the editorial team at Mercury Learning for making the updated edition a reality and the willingness to help publish this somewhat narrowly focused material written by practicing engineers for engineers and future engineers. Finally, I would especially like to thank John Pratt, who has taken time from his industry career to teach a version of the course associated with the text at University of Colorado Boulder and who also contributed significant extensions and improvement to coverage of open source real-time embedded systems aspects of this edition. He has brought a unique and valuable viewpoint to the text from the world of commercial mobile embedded systems.

Sam Siewert
December 2015

REAL-TIME EMBEDDED THEORY

Chapter 1	Real-time Embedded Theory
Chapter 2	System Resources
Chapter 3	Processing
Chapter 4	I/O Resources
Chapter 5	Memory
Chapter 6	Multiresource Services
Chapter 7	Soft Real-time Services

INTRODUCTION

In this chapter

- A Brief History of Real-Time Systems
- A Brief History of Embedded Systems

1.1 Introduction

The concept of *real-time* digital computing systems is an emergent concept compared to most engineering theory and practice. When requested to complete a task or provide a service in real time, the common understanding is that this task must be done upon request and completed while the requester waits for the completion as an output response. If the response to the request is too slow, the requester may consider lack of response a failure. The concept of real-time computing is really no different. Requests for real-time service on a digital computing platform are most often indicated by asynchronous interrupts. More specifically, inputs that constitute a real-time service request indicate a real-world event sensed by the system—for example, a new video frame has been digitized and placed in memory for processing. The computing platform must now process input related to the service request and produce an output response prior to a deadline measured relative to an event sensed earlier. The real-time digital computing system must produce a response upon request while the user and/or system waits. After the deadline established for the response, relative to the request time, the user gives up or the system fails to meet requirements if no response has been produced.

A common way to define *real time* as a noun is the time during which a process takes place or occurs. Used as an adjective, *real-time* relates to computer applications or processes that can respond with low bounded latency to user requests. One of the best and most accurate ways to define real time for computing systems is to clarify what is meant by correct real-time behavior. A correct real-time system must produce a functionally (algorithmically and mathematically) correct output response prior to a well-defined deadline relative to the request for a service.

The concept of *embedded systems* has a similar and related history to real-time systems and is mostly a narrowing of scope to preclude general-purpose desktop computer platforms that might be included in a real-time system. For example, the NASA Johnson Space Center mission control center includes a large number of commercial desktop workstations for processing of near real-time telemetry data. Often desktop real-time systems provide only soft real-time services or near real-time services rather than hard real-time services. Embedded systems typically provide hard real-time services or a mixture of hard and soft real-time services.

Again, a commonsense definition of embedding is helpful for understanding what is meant by a real-time embedded system. *Embedding* means to enclose or implant as essential or characteristic. From the viewpoint of computing systems, an *embedded system* is a special-purpose computer completely contained within the device it controls and not directly observable by the user of the system. An embedded system performs specific predefined services rather than user-specified functions and services as a general-purpose computer does.

The real-time embedded systems industry is full of specialized terminology that has developed as a subset of general computing systems terminology. To help you with that terminology, this book includes a glossary of common industry definitions. Although this book attempts to define specialized terminology in context, on occasion the glossary can help if you want to read the text out of order or when the contextual definition is not immediately clear.

1.2 A Brief History of Real-Time Systems

The origin of real time comes from the recent history of process control using digital computing platforms. In fact, an early definitive text on the concept was published in 1965 [Martin65]. The concept of real time is

also rooted in computer simulation, where a simulation that runs at least as fast as the real-world physical process it models is said to run in real time. Many simulations must make a trade-off between running at or faster than real-time with less or more model fidelity. The same is true for real-time graphical user interfaces (GUI), such as those provided by computer game engines. Not too much later than Martin's 1965 text on real-time systems, a definitive paper was published that set forth the foundation for a mathematical definition of hard real-time—"Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment" [Liu73]. Liu and Layland also defined the concept of soft real-time in 1973; however there is still no universally accepted formal definition of soft real-time. Numerous research papers and work have been completed to define QoS (Quality of Service) systems where systems either are occasionally allowed to miss deadlines or use strategies where start-up latency and buffering are used to allow for elasticity in real-time systems. Expanded coverage of soft real-time concepts and use of best-effort operating systems for soft real-time requirements has been added to the second edition of this book in Chapter 11 for readers working with embedded Linux and requirements not as strict as hard real-time, where deadlines must never be missed since by definition this means total system failure.

The concept of hard real-time systems became better understood based upon experience and problems noticed with fielded systems—one of the most famous examples early on was the Apollo 11 lunar module descent guidance overload. The Apollo 11 system suffered CPU resource overload that threatened to cause descent guidance services to miss deadlines and almost resulted in aborting the first landing on the Moon. During descent of the lunar module and use of the radar system, astronaut Buzz Aldrin notes a computer guidance system alarm. As recounted in the book *Failure Is Not an Option* [Kranz00], Buzz radios, "Program alarm. It's a 1202." Eugene Kranz, the mission operations director for Apollo 11, goes on to explain, "The alarm tells us that the computer is behind in its work. If the alarms continue, the guidance, navigation, and crew display updates will become unreliable. If the alarms are sustained, the computer could grind to a halt, possibly aborting the mission." Ultimately, based upon experience with this overload condition gained in simulation, the decision was to press on and ignore the alarm—as we all know, the Eagle did land and Neil Armstrong did later safely set foot on the Moon. How, in general, do you know that a system is overloaded with respect to CPU, memory, or IO resources?

Clearly, it is beneficial to maintain some resource margin when the cost of failure is too high to be acceptable (as was the case with the lunar lander), but how much margin is enough? When is it safe to continue operation despite resource shortages? In some cases, the resource shortage might just be a temporary overload from which the system can recover and continue to provide service meeting design requirements. The alarm 1202 may not have been the root cause of the overload, and in fact some accounts point to alarm 1201 as a cause, but the key is that the processor overload indicated by the 1202 was the result of more computing than could be handled by required deadlines when the addition of alarm processing was added to the normal workload. Peter Adler gives a more detailed account for those readers interested in the accurate history accounts as given by engineers who were directly involved [Adler 98].

Since Apollo 11, more interesting real-time problems observed in the field have shown real-time systems design to be even more complicated than simply ensuring margins. For example, the Mars Pathfinder spacecraft was nearly lost due to a real-time processing issue. The problem was not due to an overload, but rather a priority inversion causing a deadline to be missed despite having a reasonable CPU margin. The Pathfinder priority inversion scenario is described in detail in Chapter 6, “Multiresource Services.” As you’ll see, ensuring safe, mutually exclusive access to shared memory can cause priority inversion. While safe access is required for functional correctness, meeting response deadlines is also a requirement for real-time systems. A real-time system must produce functionally correct answers on time, before deadlines for overall system correctness. Given some systems development experience, most engineers are familiar with how to design and test a system for correct function. Furthermore, most hardware engineers are familiar with methods to design digital logic timing and verify correctness. When hardware, firmware, and software are combined in a real-time embedded system, response timing must be designed and tested to ensure that the integrated system meets deadline requirements. This requires system-level design and test that go beyond hardware or software methods typically used.

As history has shown, systems that were well tested still failed to provide responses by required deadlines. How do unexpected overloads or inversions happen? To answer these questions, some fundamental hard real-time theory must first be understood—this is the impetus for the “System Resources” chapter. By the end of the first section of this book, you should

be able to explain what happened in scenarios such as the Apollo 11 descent and the Mars Pathfinder deadline overrun and how to avoid such pitfalls.

1.3 A Brief History of Embedded Systems

Embedding is a much older concept than real time. Embedded digital computing systems are an essential part of any real-time embedded system and process that senses input to produce responses as output to actuators. The sensors and actuators are components providing IO and define the interface between an embedded system and the rest of the system or application. Left with this as the definition of an embedded digital computer, you could argue that a general-purpose workstation is an embedded system; after all, a mouse, keyboard, and video display provide sensor/actuator-driven IO between the digital computer and a user. However, to satisfy the definition of an embedded system better, we distinguish the types of services provided.

A general-purpose workstation provides a platform for unspecified, to-be determined sets of services, whereas an embedded system provides a well-defined service or set of services, such as antilock braking control. In general, providing general services is impractical for applications such as computation of π to the n th digit, payroll, or office automation on an embedded system. Finally, the point of an embedded system is to cost-effectively provide a more limited set of services in a larger system, such as an automobile, aircraft, or telecommunications switching center.

1.4 Real-Time Services

The concept of a real-time service is fundamental in real-time embedded systems.

Conceptually, a *real-time service* provides a transformation of inputs to outputs in an embedded system to provide a function. For example, a service might provide thermal control for a subsystem by sensing temperature with thermistors (temperature-sensitive resistors) to cool the subsystem with a fan or to heat it with electric coils. The service provided in this example is thermal management such that the subsystem temperature is maintained within a set range. Many real-time embedded services are digital control functions and are periodic in nature. An example of a real-time service that has a function other than digital control is digital media

processing. A well-known real-time digital media processing application is the encoding of audio for transport over a network with playback (decoding) on a distant network node. When the record and playback services are run on two nodes and duplex transport is provided, the system provides voice communication services. In some sense, all computer applications are services, but real-time services must provide the function within time constraints that are defined by the application. In the case of digital control, the services must provide the function within time constraints required to maintain stability. In the case of voice services, the function must occur within time constraints so that the human ear is reasonably pleased by clear audio. The service itself is not a hardware, firmware, or software entity, but rather a conceptual state machine that transforms an input stream into an output stream with regular and reliable timing.

Listing 1.1 is a pseudo code outline of a basic service that polls an input interface for a specific input vector.

Listing 1.1: Pseudo Code for Basic Real-Time Service

```
void provide_service(void)
{
    if( initialize_service() == ERROR)
        exit(FAILURE_TO_INITIALIZE);
    else
        in_service = TRUE;
    while(in_service)
    {
        if(checkForEvent(EVENT_MASK) == TRUE)
        {
            read_input(input_buffer);
            output_buffer=do_service(input_buffer);
            write_output(output_buffer);
        }
    }
    shutdown_service();
}
```

This implementation is simple and lends itself well to a hardware state machine (shown in Figure 1.1). Implementing a service as a single looping state machine is often not a practical software implementation on a microprocessor when multiple services must share a single CPU. As more services are added, the loop must be maintained and all services are limited

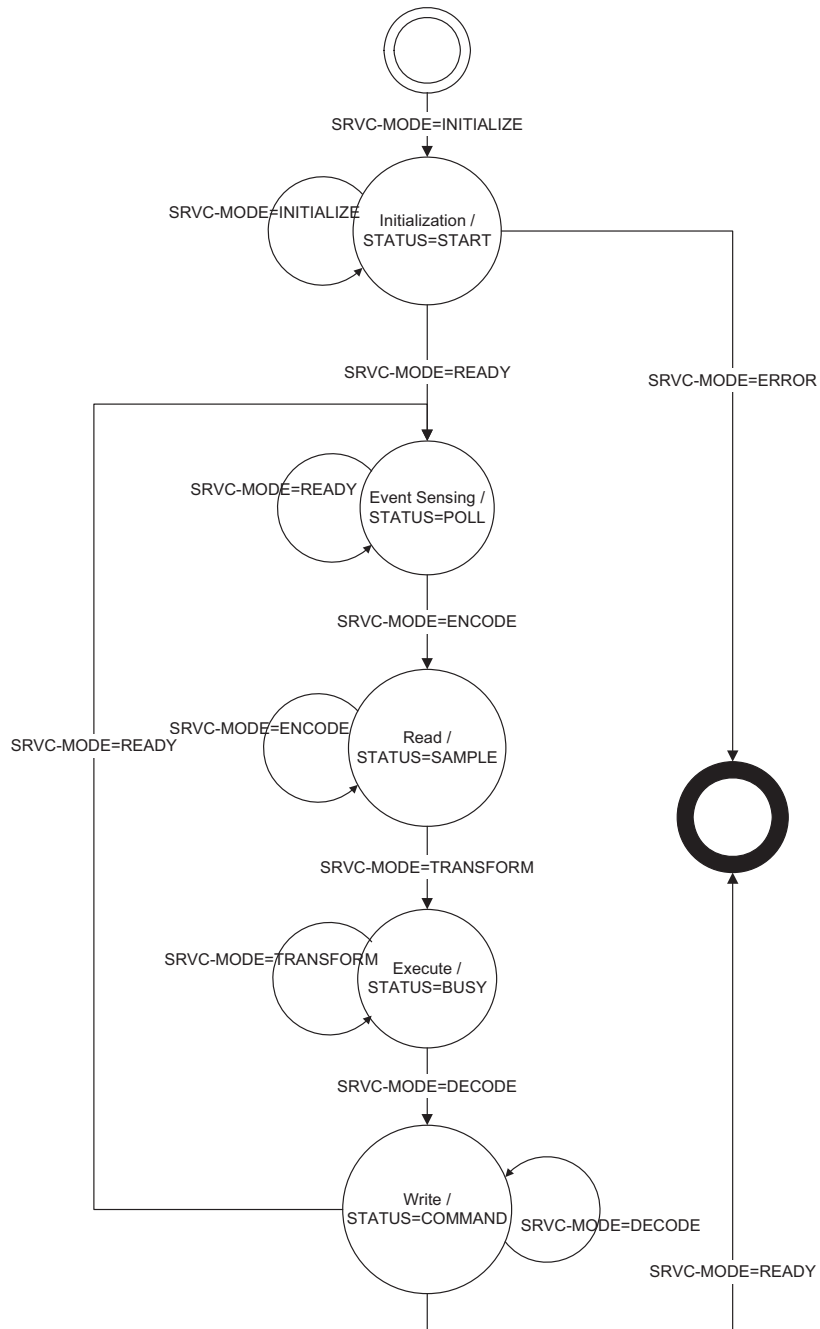


FIGURE 1.1: A Simple Polling State Machine for Real-Time Services

to a maximum rate established by the main loop period. This architecture for real-time services has historically been called the Main+ISR or Cyclic Executive design when applied to software services running on a microcontroller or microprocessor. Chapter 2 will describe the Cyclic Executive and Main+ISR system architectures in more detail. As the Executive approach is scaled, multiple services must check for events in a round-robin fashion. While one service is in the Execute state shown in Figure 1.1, the others are not able to continue polling, which causes significant latency to be added to the sensing of the real-world events. So, Main+ISR works okay for systems with a few simple services that all operate at some basic period or multiple thereof in the main loop. For systems concerned only with throughput, polling services are acceptable and perhaps even preferable, but for real-time services where event detection latency matters, multiple polling services do not scale well. For large numbers of services, this approach is not a good option without significant use of concurrent hardware state machines to assist main loop functions.

When a software implementation is used for multiple services on a single CPU, software polling is often replaced with hardware offload of the event detection and input encoding. The offload is most often done with an ADC (Analog-to-Digital Converter) and DMA (Direct Memory Access) engine that implements the Event Sensing state in Figure 1.1. This hardware state machine then asserts an interrupt input into the CPU, which in turn sets a flag used by a scheduling state machine to indicate that a software data processing service should be dispatched for execution. The following is a pseudo code outline of a basic event-driven software service:

```
void provide_service(void)
{
    if( initialize_service() == ERROR)
        exit(FAILURE_TO_INITIALIZE);
    else
        in_service = TRUE;
    while(in_service)
    {
        if(waitFor(service_request_event, timeout) != TIMEOUT)
        {
            read_input(input_buffer);
            output_buffer=do_service(input_buffer);
            write_output(output_buffer);
        }
    }
}
```

```

        else post_timeout_error();
        post_service_aliveness(serviceIDSelf());
    }
    shutdown_service();
}

```

The preceding **waitFor** function is a state that the service sits in until a real-world event is sensed. When the event the service is tied to occurs, an interrupt service routine releases the software state machine so that it reads input, processes that input to transform it, and then writes the output to an actuation interface. Before entering the **waitFor** state, the software state machine first initializes itself by creating resources that it needs while in service. For example, the service may need to reserve working memory for the input buffer transformation. Assuming that initialization goes well, the service sets a flag indicating that it is entering the service loop, which is executed indefinitely until some outside agent terminates the service by setting the service flag to FALSE, causing the service loop to be exited. In the **waitFor** state, the service is idle until activated by a sensed event or by a timeout.

Timeouts are implemented using hardware interval timers that also assert interrupts to the multiservice CPU. If an interval timer expires, timeout conditions are handled by an error function because the service is normally expected to be released prior to the timeout. The events that activate the state machine are expected to occur on a regular interval or at least within some maximum period. If the service is activated from the **waitFor** state by an event, then the service reads input from the sensor interface (if necessary), processes that input, and produces output to an actuator interface. Regardless of whether the service is activated by a timeout or a real-world event, the service always checks in with the system health and status monitoring service by posting an aliveness indication. A separate service designed to watch all other services to ensure all services continue to operate on a maximum period can then handle cases where any of the services fail to continue operation.

The **waitFor** state is typically implemented by associating hardware interface interrupt assertion with software interrupt handling. If the service is implemented as a digital hardware Mealy/Moore SM (State Machine) the SM would transition from the **waitFor** state to an input state based upon clocked combinational logic and the present input vector driven from

a sensor ADC interface. The hardware state machine would then execute combinational logic and clock and latch flip-flop register inputs and outputs until an output vector is produced and ultimately causes actuation with a DAC (Digital to Analog Converter). Ultimately, the state of the real world is polled by a hardware state machine, by a software state machine, or by a combination of both. Some combination of both is often the best trade-off and is one of the most popular approaches. Implementing service data in software provides flexibility so that modifications and upgrades can be made much more easily as compared to hardware modification.

Because a real-time service is triggered by a real-world event and produces a corresponding system response, how long this transformation of input to output takes is a key design issue. Given the broad definition of *service*, a real-time service may be implemented with hardware, firmware, and/or software components. In general, most services require an integration of components, including at least hardware and software. The real-world events are detected using sensors, often transducers and analog-to-digital converters, and are tied to a microprocessor interrupt with data, control, and status buffers. This sensor interface provides the input needed for service processing. The processing transforms the input data associated with the event into a response output. The response output is most often implemented as a digital-to-analog converter interface to an electromechanical actuator. The computed response is then output to the digital-to-analog interface to control some device situated in the real world. As noted, much of the real-time embedded theory today comes from digital control and process control where computing systems were embedded early on into vehicles and factories to provide automated control. Furthermore, digital control has a distinct advantage over analog because it can be programmed without hardware modification. An analog control system, or analog computer, requires rewiring and modification of inductors, capacitors, and resistors used in control circuitry.

Real-time digital control and process control services are periodic by nature. Either the system polls sensors on a periodic basis, or the sensor components provide digitized data on a known sampling interval with an interrupt generated to the controller. The periodic services in digital control systems implement the control law of a digital control system. When a microprocessor is dedicated to only one service, the design and implementation of services are fairly simple. In this book, we will deal with systems that include many services, many sensor interfaces, many actuator

interfaces, and one or more processors. Before delving into this complexity, let's review the elements of a service as shown in Figure 1.2.

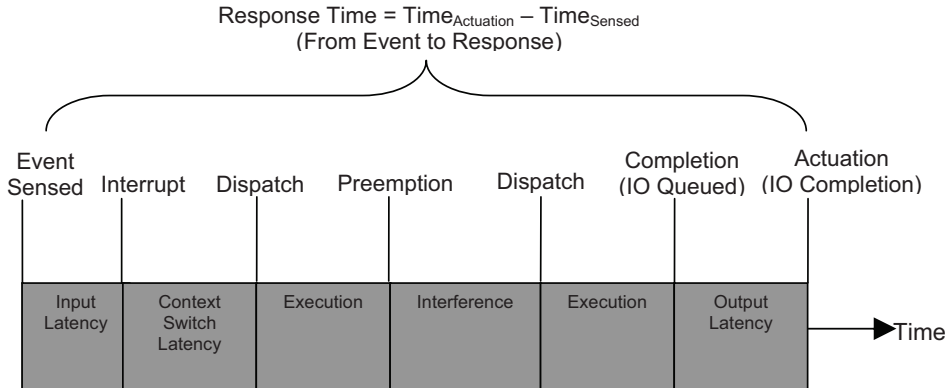


FIGURE 1.2 Real-Time Service Timeline

Figure 1.2 shows a typical service implemented with hardware IO components, including analog-to-digital converter interfaces to sensors (transducers) and digital-to-analog converter interfaces to actuators. The service processing is often implemented with a software component running as a thread of execution on a microprocessor. The service thread of execution may be preempted while executing by the arrival of interrupts from events and other services. You can also implement the service processing without software. The service may be implemented as a hardware state machine with dedicated hardware processing operating in parallel with other service processing. Implementing service processing in a software component has the advantage that the service may be updated and modified more easily. Often, after the processing or protocol related to a service is well known and stable, the processing can be accelerated with hardware state machines that either replace the software component completely in the extreme case or, most often, accelerate specific portions of the processing.

For example, a computer vision system that tracks an object in real time may filter an image, segment it, find the centroid of a target image, and command an actuator to tilt and pan the camera to keep the target object in its field of view. The entire image processing may be completed 30 times per second from an input camera. The filtering step of processing can be as simple as applying a threshold to every pixel in a 640×480 image. However, applying the threshold operation with software can be time-consuming, so it can be accelerated by offloading this step of the

processing to a state machine that applies the threshold before the input interrupt is asserted. The segmentation and centroid finding may be harder to offload because these algorithms are still being refined for the system. The service timeline for the hardware accelerated service processing is shown in Figure 1.3. In Figure 1.3, interference from other services is not shown, but would still be possible.

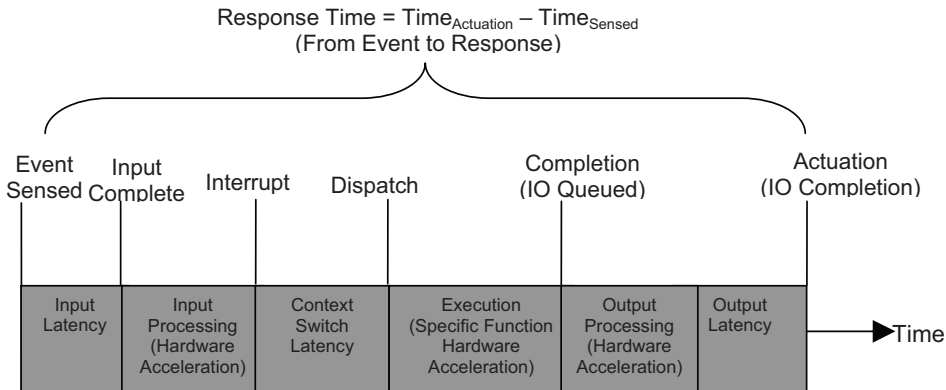


FIGURE 1.3 Real-Time Service Timeline with Hardware Acceleration

Ultimately, all real-time services may be hardware only or a mix of hardware and software processing in order to link events to actuations to monitor and control some aspect of an overall system. Finally, in both Figure 1.2 and Figure 1.3, response time is shown as being limited by the sum of the IO latency, context switch latency, execution time, and potential interference time. *Input latency* comes from the time it takes sensor inputs to be converted into digital form and transferred over an interface into working memory. *Context switch latency* comes from the time it takes code to acknowledge an interrupt indicating data is available, to save register values and stack for whatever program may already be executing (preemption), and to restore state if needed for the service that will process the newly available data. Execution ideally proceeds without interruption, but if the system provides multiple services, then the CPU resources may be shared and interference from other services will increase the response time. Finally, after a service produces digital output, there will be some latency in the transfer from working memory to device memory and potential DAC conversion for actuation output. In some systems, it is possible to overlap IO latency with execution time, especially execution of other services during IO latency for the current service that would otherwise leave the CPU

underused. Initially we will assume no overlap, but in Chapter 4, we'll discuss design and tuning methods to exploit IO-execution overlap.

In some cases, a real-time service might simply provide an IO transformation in real time, such as a video encoder display system for a multimedia application. Nothing is being controlled per se as in a digital control application. However, such systems, referred to as *continuous media real-time applications*, definitely have all the characteristics of a real-time service. Continuous media services, like digital control, require periodic services—in the case of video, most often for frame rates of 30 or 60 frames per second. Similarly, digital audio continuous media systems require encoding, processing, and decoding of audio sound at kilohertz frequencies. In general, a real-time service may be depicted as a processing pipeline between a periodic source and sink, as shown in Figure 1.4. Furthermore, the pipeline may involve several processors and more than one IO interface. The Figure 1.4 application simply provides video encoding, compression, transport of data over a network, decompression, and decoding for display. If the services on each node in the network do not provide real-time services, the overall quality of the video display at the pipeline sink may have undesirable qualities, such as frame jitter and dropouts.

Real-time continuous media services often include significant hardware acceleration. For example, the pipeline depicted in Figure 1.4 might include a compression and decompression state machine rather than performing compression and decompression in the software service on each node. Also, most continuous media processing systems include a data-plane and a control-plane for hardware and software components. The data-plane includes all elements in the real-time service pipeline, whereas the control-plane includes non-real-time management of the pipeline through an API (Application Program Interface). A similar approach can be taken for the architecture of a digital control system that requires occasional management. In the case of the video pipeline shown in Figure 1.4, the control API might allow a user to increase or decrease the frame rate. The source might inherently be able to encode frames at 30 fps (frames per second), but the frames may be decimated and retimed to 24 fps.

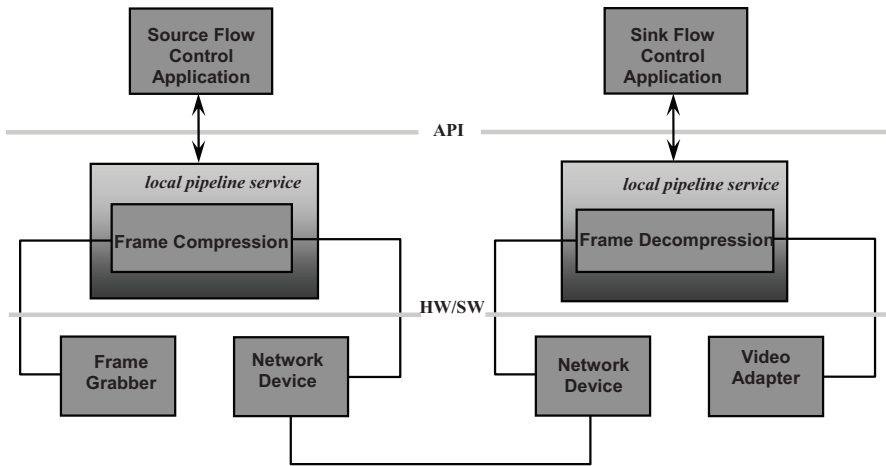


FIGURE 1.4 Distributed Continuous Media Real-Time Service

So far, all the applications and service types considered have been periodic. One of the most common examples of a real-time service that is *not* periodic by nature is error-handling service. Normally a system does not have periodic or regular faults—if it does, the system most likely needs to be replaced! So, fault handling is often asynchronous. The timeline and characteristics of an asynchronous real-time service are, however, no different from those already studied. In Chapter 2, “System Resources,” the characteristics of services will be investigated further so that different types of services can be better understood and designed for more optimal performance.

1.5 Real-Time Standards

The POSIX (Portable Operating Systems Interface) real-time standards have been consolidated into a single updated 1003.1 standard as of 2013 [POSIX 1003.1]. Originally, the real-time standards recognized by IEEE (Institute of Electrical and Electronic Engineers) as well as the Open Group were written as specific extensions to the POSIX base standards, including:

1. IEEE Std 1003.1b-2000: Testing specification for POSIX part 1 including real-time extensions
2. IEEE Std 1003.13-1998: Real-time profile standard to address embedded real-time applications and smaller footprint devices
3. IEEE Std 1003.1b-1993: Real-time extension

4. IEEE Std 1003.1c-1995: Threads
5. IEEE Std 1003.1d-1999: Additional real-time extensions
6. IEEE Std 1003.1j-2000: Advanced real-time extensions
7. IEEE Std 1003.1q-2000: Tracing

The first and one of the most significant updates made to the original base standard for real-time systems was designated 1003.1b, which specifies the API (Application Programmer Interface) that most RTOS (Real-Time Operating System) fully and Linux operating systems implement mostly. The POSIX 1003.1b extensions include definition of real-time operating system mechanisms:

1. Priority Scheduling
2. Real-Time Signals
3. Clocks and Timers
4. Semaphores
5. Message Passing
6. Shared Memory
7. Asynchronous and Synchronous I/O
8. Memory Locking

Some additional interesting real-time standards that go beyond the RTOS or general-purpose operating system support include:

1. DO-178B and updated DO-178C, Software Considerations in Airborne Systems and Equipment Certification
2. JSR-1, the Real-Time Specification for Java, recently updated by NIST (National Institute of Standards and Technology)
3. The TAO Real-Time CORBA (Common Object Request Broker Architecture) implementation
4. The IETF (Internet Engineering Task Force) RTP (Real-Time Transport Protocol) and RTCP (Real-Time Control Protocol) RFC 3550
5. The IETF RTSP (Real-Time Streaming Protocol) RFC 2326

Summary

Now that you are armed with a basic definition of the concept of a real-time embedded system, we can proceed to delve into real-time theory and embedded resource management theory and practice. (A large number of specific terms and terminology are used in real-time embedded systems, so be sure to consult the complete glossary of commonly used terms at the end of this book.) This theory is the best place to start because it is fundamental. A good understanding of the theory is required before you can proceed with the more practical aspects of engineering components for design and implementation of a real-time embedded system.



The Exercises, Labs, and the Example projects included with this text on the DVD are intended to be entertaining as well as an informative and valuable experience—the best way to develop expertise with real-time embedded systems is to build and experiment with real systems. Although the Example systems can be built for a low cost, they provide a meaningful experience that can be transferred to more elaborate projects typical in industry. All the Example projects presented here include a list of components, list of services, and a basic outline for design and implementation. They have all been implemented successfully numerous times by students at the University of Colorado. For more challenge, you may want to combine elements and mix up services from one example with another; for example, it is possible to place an NTSC (National Television Systems Council) camera on the grappler of the robotic arm to recognize targets for pickup with computer vision. A modification such as this one to include a video stream is a nice fusion of continuous media and robotic applications that requires the application of digital control theory as well.

Exercises

1. Provide examples of real-time embedded systems you are familiar with and describe how these systems meet the common definition of real-time and embedded.
2. Find the Liu and Layland paper and read through Section 3. Why do they make the assumption that all requests for services are periodic? Why might this be a problem with a real application?
3. Define hard and soft real-time services and describe why and how they are different.

Chapter References

- [Adler98] <https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html>
- [Kranz00] Gene Kranz, *Failure Is Not an Option*, Berkley Books, New York, 2000.
- [Liu73] C. L. Liu and James W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *Journal of the Association for Computing Machinery*, Vol. 20, No. 1 (January 1973): pp. 46–61.
- [Martin65] James Martin, *Programming Real-Time Computer Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1965.
- [POSIX1003.1] http://www.opengroup.org/austin/papers/posix_faq.html,
<https://www2.opengroup.org/ogsys/catalog/t101>

SYSTEM RESOURCES

In this chapter

- Introduction
- Resource Analysis
- Real-Time Service Utility
- Scheduling Classes
- The Cyclic Executive
- Scheduler Concepts
- Real-Time Operating Systems
- Thread-Safe Reentrant Functions

2.1 Introduction

Real-time embedded systems must provide deterministic behavior and often have more rigorous time- and safety-critical system requirements compared to general-purpose desktop computing systems. For example, a satellite real-time embedded system must survive launch and the space environment, must be very efficient in terms of power and mass, and must meet high reliability standards. Applications that provide a real-time service could in some cases be much simpler if they were not resource-constrained by system requirements typical of an embedded environment.

For example, a desktop multimedia application can provide MPEG (Motion Picture Experts Group) playback services on a high-end PC with

a high degree of quality without significant specialized hardware or software design; this type of scenario is “killing the problem with resources.” The resources of a desktop system often include a high throughput CPU (GHz clock rate and billions of instructions per second), a high bandwidth, low-latency bus (gigabit), a large-capacity memory system (gigabytes), and virtually unlimited public utility power. By comparison, an antilock braking system must run off the automobile’s 12-volt DC (Direct Current) power system, survive the under-hood thermal and vibration environment, and provide braking control at a reasonable consumer cost. So for most real-time embedded systems, simply killing the problem with resources is not a valid option.

The engineer must instead carefully consider resource limitations, including power, mass, size, memory capacity, processing, and IO bandwidth. Furthermore, complications of reliable operation in hazardous environments may require specialized resources, such as error detecting and correcting memory systems. To successfully implement real-time services in a system providing embedded functions, resource analysis must be completed to ensure that these services not only are functionally correct but also produce output on time and with high reliability and availability. Part I of this book, “Real-Time Embedded Theory,” provides a resource view of real-time embedded systems and methods to make optimal use of these resources. In Chapter 3, “Processing,” we will focus on how to analyze CPU resources. In Chapter 4, “IO Resources,” resources will be characterized and methods of analysis presented. Chapter 5, “Memory,” provides memory resource analysis methods. Chapter 6, “Multiresource Services,” provides an overview of how these three basic resources related to workload throughput relate to more fundamental resources, such as power, mass, and size. Chapters 2 through 6 provide the classic hard real-time view of resource sizing, margins, and deterministic system behavior. Chapter 7, “Soft Real-Time Services,” completes the resource view by presenting the latest concepts for soft real-time resource management, where occasional service deadline misses and failure to maintain resource margins are allowed. The second edition provides a new Chapter 11 to provide guidance on how to use embedded Linux as a soft real-time operating system for the soft real-time services described in Chapter 7.

The three fundamental resources, CPU, memory, and IO, are excellent places to start understanding the architecture of real-time embedded systems and how to meet design requirements and objectives. Furthermore,

resource analysis is critical to the hardware, firmware, and software design in a real-time embedded system. Upon completion of the entire book, you will also understand all system resource issues, including cost, performance, power usage, thermal operating ranges, and reliability. Part II, “Designing Real-Time Embedded Components,” provides a detailed look at the design of components with three new chapters in the second edition to expand coverage of FreeRTOS, embedded Linux, and advancements in RTOS to support multi-core. Part III, “Putting It All Together,” provides an overview of how to integrate these components into a system. However, the material in Part I is most critical, because a system designed around the wrong CPU architecture, insufficient memory, or IO bandwidth is sure to fail. From a real-time services perspective, insufficient memory, CPU, or IO can make the entire project infeasible, failing to meet requirements. It is important to not only understand how to look at the three main resources individually, but also consider multiresource issues, trade-offs, and how the main three interplay with other resources, such as power. Multiresource constraints, such as power usage, may result in less memory or a slower CPU clock rate, for example. In this sense, power constraints could in the end cause a problem with a design’s capability to meet real-time deadlines.

2.2 Resource Analysis

Looking more closely at some of the real-time service examples introduced in Chapter 1, there are common resources that must be sized and managed in any real-time embedded system, including the following:

Processing: Any number of microprocessors or micro-controllers networked together

Memory: All storage elements in the system, including volatile and nonvolatile storage

IO: Input and output that encodes sensed data and is used for decoding for actuation

In the upcoming three chapters, we will characterize and derive formal models for each of the key resources: processing, IO, and memory. Here are brief outlines of how each key resource will be examined.

Traditionally the main focus of real-time resource analysis and theory has been centered around processing and how to schedule multiplexed

execution of multiple services on a single processor. Scheduling resource usage requires the system software to make a decision to allocate a resource, such as the CPU, to a specific thread of execution. The mechanics of multiplexing the CPU by preempting a running thread, saving its state, and dispatching a new thread is called a *thread context switch*. Scheduling involves implementing a policy, whereas preemption and dispatch are context-switching mechanisms. When a CPU is multiplexed with an RTOS scheduler and context-switching mechanism, the system architect must determine whether the CPU resources are sufficient given the set of service threads to be executed and whether the services will be able to reliably complete execution prior to system-required deadlines. The global demands upon the CPU must be determined. Furthermore, the reliability of the overall system hinges upon the repeatability of service execution prior to deadlines; ideally, every service request will behave so that meeting deadlines can be guaranteed. If deadlines can be guaranteed, then the system is safe. Because its behavior does not change over time in terms of ability to provide services by well-defined deadlines, the system is also considered deterministic. Before looking more closely at how to implement a deterministic system, a better understanding of system resources and what can make system response vary over time is required.

The main considerations include speed of instruction execution (clock rate), the efficiency of executing instructions (average Cycles Per Instruction [CPI]), algorithm complexity, and frequency of service requests. Today, given that many processors are superscalar pipelines, which provide the ability to process one or more instructions in a clock cycle, the inverse of CPI, IPC (Instructions Per Clock), is also used instead of CPI. Either way, other than a change of denominator and numerator, the ratio has the same ability to indicate execution efficiency, much like MPG (Miles Per Gallon) or GPM (Gallons Per Mile). Chapter 3 provides a detailed examination of processing:

Speed: Clock Rate for Instruction Execution.

Efficiency: CPI or IPC (Instructions Per Clock); processing stalls due to hazards; for example, read data dependency, cache misses, and write buffer overflow stalls.

Algorithm complexity: C_i = instruction count on service longest path for service i and, ideally, is deterministic; if C_i is not known, the worst case should be used—WCET (Worst-Case Execution Time) is the

longest, most inefficiently executed path for service; WCET is one component of response time (as shown in Figures 1.2 and 1.3 in Chapter 1); other contributions to response time come from input latency; dispatch latency; execution; interference by higher-priority services and interrupts; and output latency.

Service Frequency: T_i = Service Release Period.

Chapter 13, “Performance Tuning,” provides tips for resolving execution efficiency issues. Execution efficiency is not a real-time requirement, but inefficient code can lead to large WCETs, missed deadlines, and difficult scheduling. So, understanding methods for tuning software performance can become important.

Input and output channels between processor cores and devices are one of the most important resources in real-time embedded systems and perhaps one of the most often overlooked as far as theory and analysis. In a real-time embedded system, low latency for IO is fundamental. The response time of a service can be highly influenced by IO latency, as is evident in Figures 1.2 and 1.3 of Chapter 1. Many processor cores have the capability to continue processing instructions while IO reads are pending or while IO writes are draining out of buffers to devices. This decoupling helps efficiency tremendously, but when the service processing requires read data to continue or when write buffers become full, processing can stall; furthermore, no response is complete until writes actually drain to output device interfaces. So, key IO parameters are latency, bandwidth, read/write queue depths, and coupling between IO channels and the CPU. The coverage of IO resource management in Chapter 4 includes the following:

- **Latency**
 - Arbitration latency for shared IO interfaces
 - Read latency
 - Time for data transit from device to CPU core
 - Registers, Tightly Coupled Memory (TCM), and L1 cache for zero wait state
 - Single cycle access
 - Bus interface read requests and completions: split transactions and delay

- Write latency
 - Time for data transit from CPU core to device
 - Posted writes prevent CPU stalls
 - Posted writes require bus interface queue
- ***Bandwidth (BW)***
 - Average bytes or words transferred per unit time
 - BW says nothing about latency, so it is not a panacea for real-time systems
- ***Queue depth***
 - Write buffer stalls will decrease efficiency when queues fill up
 - Read buffers—most often stalled by need for data to process
- ***CPU coupling***
 - DMA channels help decouple the CPU from IO
 - Programmed IO strongly couples the CPU to IO
 - Cycle stealing requires occasional interaction between the CPU and DMA engines

Memory resources are designed based upon cost, capacity, and access latency. Ideally all memory would be zero wait state so that the processing elements in the system could access data in a single processing cycle. Due to cost, the memory is most often designed as a hierarchy, with the fastest memory being the smallest due to high cost, and large-capacity memory the largest and lowest cost per unit storage. Nonvolatile memory is most often the slowest access. The management, sizing, and allocation of memory for real-time embedded systems will be covered in detail in Chapter 5, which includes the following:

- Memory hierarchy from least to most latency
 - Level-1 cache
 - Single cycle access
 - Typically Harvard architecture—separate data and instruction caches
 - Locked for use as fast memory, unlocked for set-associative, or direct mapped caches

- Level-2 cache or TCM
 - Few or no wait-states (e.g., two cycle access)
 - Typically unified (contains both data and code)
 - Locked for use as TCM, unlocked to back L1 caches
- MMRs (Memory Mapped Registers)
- Main memory—SRAM, SDRAM, DDR (see Appendix A Glossary)
 - Processor bus interface and controller
 - Multicycle access latency on-chip
 - Many-cycle access latency off-chip
- MMIO (Memory Mapped IO) devices
- Nonvolatile memory like flash, EEPROM, and battery-backed SRAM
 - Slowest read/write access, most often off-chip
 - Requires algorithm for block erase: interrupt upon completion and poll for completion for flash and EEPROM
- Total capacity for code, data, stack, and heap requires careful planning
- Allocation of data, code, stack, and heap to physical hierarchy will significantly affect performance

Traditionally, real-time theory and systems design have focused almost entirely on sharing CPU resources and, to a lesser extent, issues related to shared memory, IO latency, IO scheduling, and synchronization of services. To really understand the performance of a real-time embedded system and to properly size resources for the services to be supported, all three resources must be considered as well as interactions between them. A given system may experience problems meeting service deadlines because it is:

- **CPU bound:** Insufficient execution cycles during release period and due to inefficiency in execution
- **IO bound:** Too much total IO latency during the release period and/or poor scheduling of IO during execution
- **Memory bound:** Insufficient memory capacity or too much memory access latency during the release period

In fact, most modern microprocessors have MMIO (Memory Mapped IO) architectures so that memory access latency and device IO latency contribute together to response latency. Most often, many execution cycles can be overlapped with IO and memory access time for better efficiency, but this requires careful scheduling of IO during a service release. The concept of overlap and IO scheduling is discussed in detail in Chapter 4, “IO Resources.”

This book provides a more balanced characterization of all three major resources.

At a high level, a real-time embedded system can be characterized in terms of CPU, IO, and memory resource margin maintained as depicted in Figure 2.1. The box at the origin in the figure depicts the region where a system would have high CPU, IO, and memory margins—this is ideal, but perhaps not realistic due to cost, mass, power, and size constraints. The box in the top-right corner depicts the region where a system has very little resource margin.

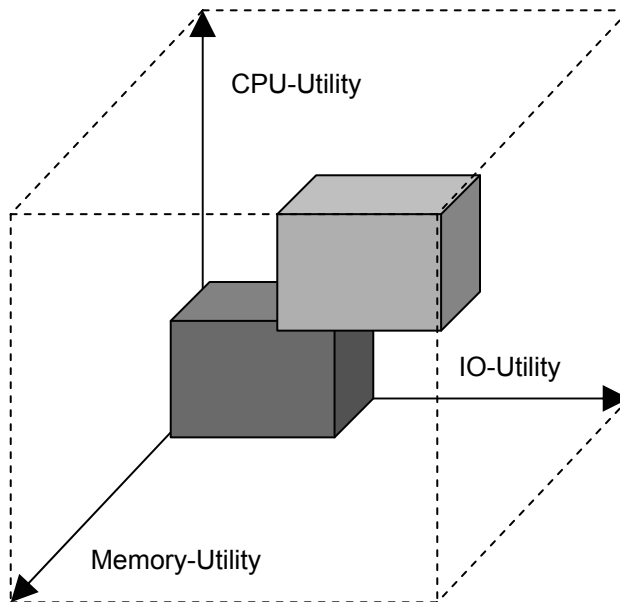


FIGURE 2.1 Real-Time Embedded System Resource Characterization

Often the resource margin that a real-time embedded system is designed to maintain depends upon a number of higher-level design factors, including:

- System cost

- Reliability required (how often is the system allowed to fail if it is a soft real-time system?)
- Availability required (how often the system is expected to be out of service or in service?)
- Risk of oversubscribing resources (how deterministic are resource demands?)
- Impact of oversubscription (if resource margin is insufficient, what are the consequences?)

Unfortunately, prescribing general margins for any system with specific values is difficult. However, here are some basic guidelines for resource sizing and margin maintenance:

CPU: The set of proposed services must be allocated to processors so that each processor in the system meets the Lehoczky, Sha, Ding theorem for feasibility. Normally, the CPU margin required is less than the RM LUB (Rate Monotonic Least Upper Bound) of approximately 30%. You'll see why this is in Chapter 3, "Processing." The amount of margin required depends upon the service parameters—mostly their relative release periods and how harmonic the periods are. Furthermore, for asymmetric MP (MultiProcessor) systems, scaling may be fairly linear if the loads on each processor are, in fact, independent. If services on different processors have dependencies and share resources, such as memory, or require message-based synchronization, however, the scalability is subject to Amdahl's law. Amdahl's law provides estimation for the speedup provided by additional processors when dependencies exist between the processing on each processor. Potential speedup is discussed further in the "Processing" section of this chapter as well.

IO: Total IO latency for a given service should never exceed the response deadline or the service release period (often the deadline and period are the same). You'll see that execution and IO can be overlapped so that the response time is not the simple sum of IO latency and execution time. In the worst case, the response time can be as is suggested by Figures 1.2 and 1.3 in Chapter 1. Overlapping IO time with execution time is therefore a key concept for better performance.

Overlap theory is presented in Chapter 4, “IO Resources.” Scheduling IO so that it overlaps is often called *IO latency hiding*.

Memory: The total memory capacity should be sufficient for the worst-case static and dynamic memory requirements for all services. Furthermore, the memory access latency summed with the IO latency should not exceed the service release period. Memory latency can be hidden by overlapping memory latency with careful instruction scheduling and use of cache to improve performance.

The largest challenge in real-time embedded systems is dealing with the trade-off between determinism and efficiency gained from less deterministic architectural features, such as set-associative caches and overlapped IO and execution. To be completely safe, the system must be shown to have deterministic timing and use of resources so that feasibility can be proven. For hard real-time systems where the consequences of failure are too severe to ever allow, the worst case must always be assumed. For soft real-time systems, a better trade-off can be made to get higher performance for lower cost, but with higher probability of occasional service failures.

In the worst case, the response time equation is

$$\begin{aligned} \forall S_i, T_{\text{response}(i)} &\leq \text{Deadline}_i, S_j \subset S_i, \text{ where } \text{Priority}_j > \text{Priority}_i \\ T_{\text{response}(i)} &= T_{\text{IO_Latency}(i)} + \text{WCET}_i + T_{\text{Memory_Latency}(i)} + T_{\text{interference}(j)} \\ \text{WCET} &\equiv \text{Worst_Case_Execution_Time} \\ T_{\text{interference}(j)} &\equiv T_{\text{response}(i)} - \sum_{j=1}^{i-1} \text{Total_Preempt_Time}_{(j)} \end{aligned}$$

All services S_i in a hard real-time system must have response times less than their required deadline, and the response time must be assumed to be the sum of the total worst-case latency. Worst-case latency is computed as shown in Figure 1.2 and accounted for in the foregoing equations. It is not easy, but WCET, IO latency, and memory latency can most often be measured or modelled to compute response time. The interference time during the response timeline of any given S_i by all instances of S_j , which by definition have higher priority than the S_i of interest, is in fact one of the more challenging components to compute, as we will see in Chapter 3. Not all systems have true hard real-time requirements so that the absolute worst-case response needs to be assumed, but many do. Commercial aircraft flight control systems do need to make worst-case assumptions to be

safe. However, worst-case assumptions need not be made for all services, just for those required to maintain controlled flight.

2.3 Real-Time Service Utility

To more formally describe various types of real-time services, the real-time research community devised the concept of a service utility function. The service utility function for a simple real-time service is depicted in Figure 2.2. The service is said to be released when the service is ready to start execution following a service request, most often initiated by an interrupt.

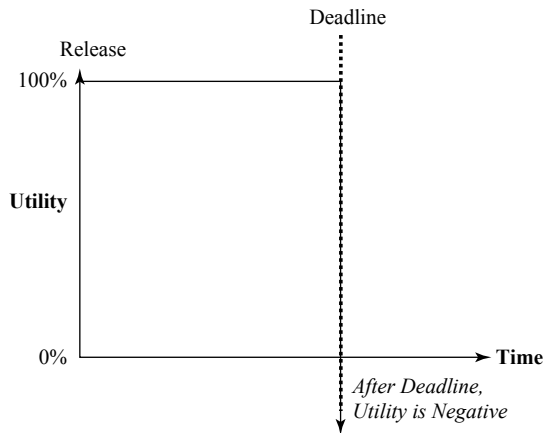


FIGURE 2.2 Hard Real-Time Service Utility

Notice that the utility of the service producing a response any time prior to the deadline relative to the request is full, and at the instant following the deadline, the utility becomes not only zero but actually negative. The implication is that continuing processing of this service request after the deadline not only is futile but also may actually cause more harm to the system than simply discontinuing the service processing. A late response might actually be worse than no response.

More specifically, if an early response is also undesirable, as it would be for an isochronal service, then the utility is negative up to the deadline, full at the deadline, and negative again after the deadline as depicted in Figure 2.3. For an isochronal service, early completion of response processing requires the response to be held or buffered up to the deadline if it is computed early. The services depicted in Figures 2.2 and 2.3 are

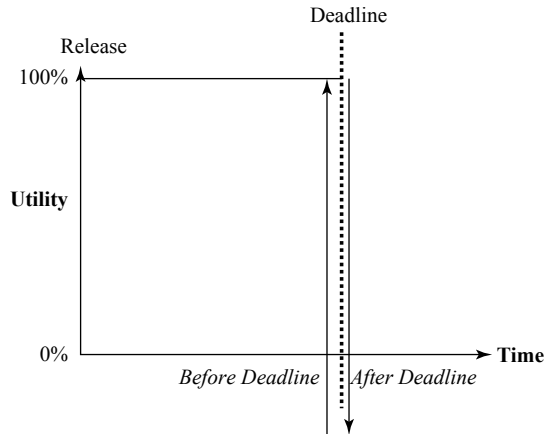


FIGURE 2.3 Isochronal Service Utility

said to be hard real-time because the utility of a late response (or early in the case of isochronal) is not only zero but also negative. A hard real-time system suffers significant harm from improperly timed responses. For example, an aircraft may lose control if the digital autopilot produces a late control surface actuation, or a satellite may be lost if a thrust is applied too long. Hard real-time services, isochronal or simple, require correct timing to avoid loss of life and/or physical assets. For digital control systems, early responses can be just as destabilizing as late responses, so they are typically hard isochronal services, but most often employ buffer-and-hold for responses computed early.

How does a hard real-time service compare to a typical non-real-time application? Figure 2.4 shows a service that is considered to produce a response with best effort. Basically, the non-real-time service has no real deadline because full utility is realized whenever a best-effort application finally produces a result. Most desktop systems and even many embedded computing systems are designed to maximize overall throughput for a workload with no guarantee on response time, but with maximum efficiency in processing the workload. For example, on a desktop system, there is no limit on how much the CPU can be oversubscribed and how much IO backlog may be generated. Memory is typically limited, but as CPU and IO backlog increases, response times become longer. No guarantee on any particular response time can be made for best-effort systems with backlogs. Some embedded systems—for example, a storage system interface—also may have no real-time guarantees. For embedded systems that need no

deadline guarantees, it makes sense that the design attempts to maximize throughput. A high throughput system may, in fact, have low latency in processing requests, but this still does not imply any sort of guarantee of response by a deadline relative to the request. As shown in Figure 2.4, full utility is assumed no matter when the response is generated.

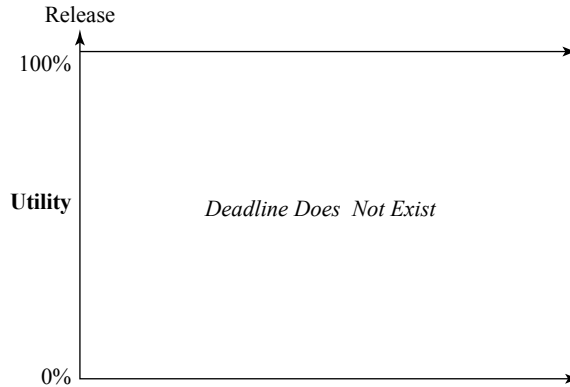


FIGURE 2.4 Best-Effort Service Utility

The real-time research community agrees upon the definitions of hard real-time, isochronal, and best-effort services. The exact definition of soft real-time services is, by contrast, somewhat unclear. One idea for the concept of soft real-time is similar to the idea of receiving partial credit for late homework because a service that produces a late response still provides some utility to the system. An alternative idea for the concept of soft real-time is also similar to a well-known homework policy in which some service dropouts are acceptable. In this case, by analogy, no credit is given for late homework, but the student is allowed to drop his or her lowest score or scores. Either definition of soft real-time clearly falls between the extremes of the hard real-time and the best-effort utility curves. Figure 2.5 depicts the soft real-time concept where some function greater than or equal to zero exists for soft real-time service responses after the response deadline—if the function is identically zero, a well-designed system will simply terminate the service after the deadline, and a service dropout will occur. If some partial utility can be realized for a late response, a well-designed system may want to allow for some fixed amount of service overrun, as shown in Figure 2.5.

For continuous media applications (video and audio), most often there is no reason for producing late responses because they cause frame jitter.

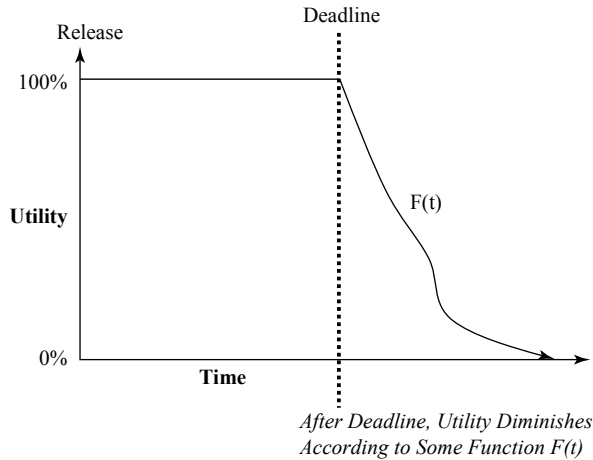


FIGURE 2.5 Soft Real-Time Utility Curve

Often, it is best for these applications if the previous frame or sound-bite output is produced again and the late output dropped when the next frame is delivered on time. Of course, if multiple service dropouts occur in a row, this leads to unacceptable quality of service. The advantage of a dropout policy is that processing can be terminated as soon as it is determined that the current service in progress will miss its deadline, thereby freeing up CPU resources. By analogy, if an instructor gives zero credit for late homework (and no partial credit for partially completed work), wise students will cease work on that homework as soon as they realize they can't finish and reallocate their time to homework for other classes. Other classes may have a different credit policy for accepting late work, but given that work on the futile homework has been dropped, students may have plenty of time to finish other work on time.

A policy known as the *anytime algorithm* is analogous to receiving partial credit for partially completed homework and partial utility for a partially complete service. The concept of an anytime algorithm can be implemented only for services where iterative refinement is possible—that is, the algorithm produces an initial solution long before the deadline, but can produce a better solution (response) if allowed to continue processing up to the deadline for response. If the deadline is reached before the algorithm finds a more optimal solution than the current best, then it simply responds with the best solution found so far. Anytime algorithms have been used most for robotic and AI (Artificial Intelligence) real-time applications where iterative

refinement can be beneficial. For example, a robotic navigation system might include a path-planning search algorithm for the map it is building in memory. When the robot encounters an obstacle, it must decide whether to turn left or right. When not much of the environment is mapped, a simple random selection of left or right might be the best response possible. Later on, after more of the environment is mapped, the algorithm might need to run longer to find a path to its goal or at least one that gets the robot closer to its goal. Anytime algorithms are designed to be terminated at their deadlines and produce the best solution at that time, so by definition anytime services do not overrun deadlines, but rather provide some partial utility solution following their release. The information derived during previous partial solutions may in fact be used in subsequent service releases; for example, in the robot path-planning scenario, paths and partial paths could be saved in memory for reuse when the robot returns to a location occupied previously. This is the concept of iterative refinement. The storage of intermediate results for iterative refinement requires additional memory resources (a dynamic programming method), but can lead to optimal solutions prior to the decision deadline, such as a robot that finds an optimal path more quickly and still avoids collisions and/or lengthy pauses to think! Why wouldn't the example robotic application simply halt when encountering an obstacle, run the search algorithm as long as it takes to find a solution, or determine that it has insufficient mapping or no possible path? This is possible, but may be undesirable if it is better that the robot not stand still too long because sometimes it may be better to do something rather than nothing.

By making a random choice, the robot might get lucky and map out more of the environment more quickly. Anytime algorithms are not always the best approach for services, but clearly they are another option for less deterministic services and avoiding overruns. If the robot simply ran until it determined the optimal answer each time, then this would constitute a best-effort service rather than anytime. Figure 2.6 depicts an anytime real-time utility curve.

Finally, services can be some combination of the previously explained service types: hard, isochronal, best-effort, soft, or anytime. For example, what about a soft isochronal service? An isochronal service can achieve partial utility for responses prior to the deadline, full utility at the deadline, and again partial utility after the deadline. For example, in a continuous media system, it may not be possible to hold early responses until the deadline,

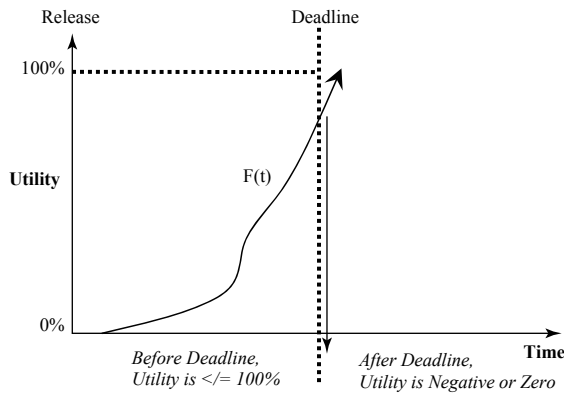


FIGURE 2.6 Anytime Service Utility

and it may also be beneficial to produce a late response—allow an overrun. In some sense, the idea of a hard isochronal service is hard to imagine, yet feedback digital control systems are a very important example of a hard isochronal application—that is, always producing the response exactly at the desired deadline relative to release. Isochronal systems are normally implemented using hold buffers and traditional hard real-time services, early service completions must be buffered, and CPU scheduling must ensure that late responses will never happen. So, a soft isochronal service would be far easier to implement because there is no need for early completion buffering and no need to detect and terminate services that overrun deadlines. Allowing indefinite overrun of any soft real-time service can ultimately be a problem. If overruns continue to occur and service releases start to overlap for the same service, the loading simply climbs higher and higher. How can such a system ever recover? Allowing indefinite overruns would be similar to the scenario where the overly conscientious student continues to work on more and more late homework, ultimately lowering grades in all courses to the point that total failure is the ultimate outcome. The example of a soft isochronal service is depicted in Figure 2.7.

Clearly, an intelligent real-time agent would use a resource-scheduling policy that leads to maximum utility in all responses for all services. For multiple services, the concept of maximizing total utility requires a normalization of utility scales. One possible way to normalize utility scales is to assign an importance factor to each service. Furthermore, in an ideal system, different policies would be applied for overruns based upon the known utility functions for each service and relative importance of each service. Using

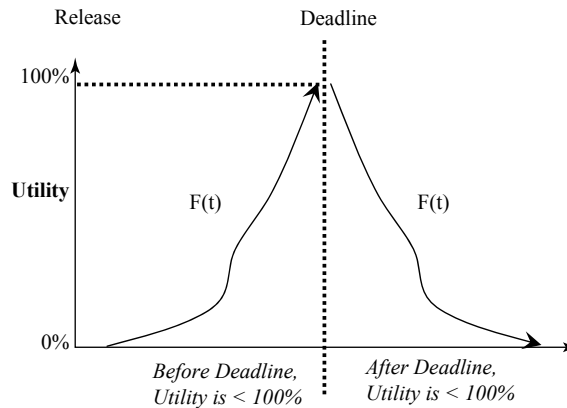


FIGURE 2.7 Soft Isochronal Service Utility

the student example again, the student being the ideal and ultimate scheduler in resource overload is smart enough to do the following:

1. Ensure sufficient margin for hard deadlines—attending classes where attendance is required.
2. Use a best-effort approach for extracurricular activities.
3. Turn in late submissions for reduced credit in classes that allow this.
4. Apply the anytime policy for classes that do not accept late submissions, yet award partial credit.
5. Hold onto assignments completed early in classes where absent-minded professors might misplace an early submission.

Students are much smarter than most real-time embedded systems. It is likely that implementation of a resource-scheduling algorithm as intelligent and optimal as a student's is not practical for an embedded system. However, most real-time embedded systems do mix policies like the intelligent student did in the example, but most often only two at a time. One of the most frequently used mixes is a set of hard real-time services with some best-effort services. Guarantees for the hard real-time services are proven using methods discussed later in Chapter 3, and best-effort services simply use all the leftover resource by executing in slack time.

The primary focus of this text is on understanding hard real time; however, soft real-time and isochronal services are also covered. An examination of anytime services is not provided in this text. Anytime services are

important to AI and robotic systems where planning algorithms that are NP hard (Nondeterministic Polynomial bound on compute time) must be run in real time. This is a very specialized form of a real-time embedded system. Because soft real-time services are more generalized and less deterministic compared to hard real-time services, soft real time is treated as an open issue in this text—a subject that is an open research area—whereas hard real time is well understood and specific hard real-time policies can be proven optimal albeit with some system constraints and assumptions. The optimality will be demonstrated later in this chapter.

2.4 Scheduling Classes

Scheduling services and their usage of resources can be accomplished by a large variety of methods. In this section, we consider scheduling system processors with work. In general, a system might have more than one processor (CPU), and any given processor might host one or more services. Allocating a CPU to each service provided by the system might be simplest from a scheduling viewpoint, but, clearly, this would also be a costly solution. Furthermore, running services to completion, ignoring all other requests, on a first-come, first-served basis is also simple, but problems such as service starvation and missing deadlines can arise with this approach. To better understand real-time processor scheduling, you first need to review the taxonomy of all major scheduling policies that can be implemented, as shown in Figure 2.8.

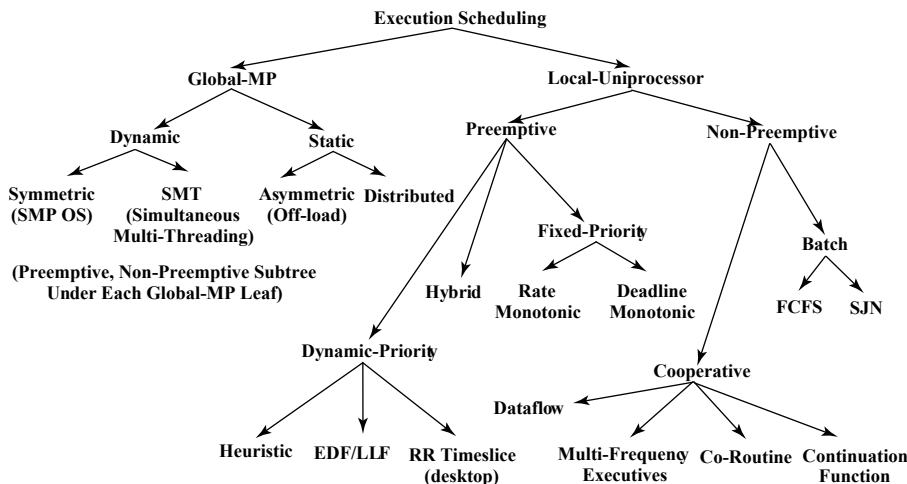


FIGURE 2.8 CPU Resource-Scheduling Taxonomy

As is the case with most policies, no one policy is optimal for all system requirements, not even when all services must be hard real-time. The policies that can be made optimal and have traditionally been used for hard real-time systems are Rate Monotonic, Deadline Monotonic, and, to a lesser extent, EDF (Earliest Deadline First)/LLF (Least Laxity First). The first branch in the taxonomy is based upon hardware design; does the system contain a single CPU or multiple CPUs?

2.4.1 Multiprocessor Systems

For multiprocessor systems, the first resource usage policy decision is whether each CPU will be used for a specific predetermined function (asymmetric, distributed) or whether workload will be assigned dynamically (symmetric). Most general-purpose MP platforms provide SMP (Symmetric MultiProcessing) where the OS determines how to assign work to the set of available processors and most often attempts to balance the workload on all processors. An SMP OS is not simple to implement, and overhead for workload balancing can be high, so many embedded multiprocessor systems are asymmetric or distributed. Asymmetric multiprocessing is used frequently to take a service that was initially provided by software running on a general-purpose CPU and offload it to a hardware state machine or tailored CPU to implement the service, therefore offloading the more general-purpose multiservice CPU. Distributed systems are typically asymmetric and communicate via message passing on a network rather than through shared memory, bus, or cross-bar. Other than the issue of load balancing, multiprocessor systems are most distinguished by their hardware architecture—shared memory, distributed message passing, or some hybrid of the two. The classic taxonomy for such systems includes SISD (Single Instruction, Single Data), SIMD (Single Instruction, Multi-Data), MISD (Multi-Instruction, Single Data), and MIMD (Multi-Instruction, Multi-Data). So, all the combinations of single or multiple instruction processing combined with single or multiple data paths are possible for MP architecture. Most embedded multiprocessor systems are multiple instruction and multiple data path hardware architectures that employ multiple CPUs for speedup.

2.5 The Cyclic Executive

Many real-time systems, including complex, hard real-time, safety-critical systems, provide real-time services using a *cyclic executive* architecture. Cyclic executives do not require an RTOS or generalized scheduling

mechanism. A cyclic executive provides a loop control structure to explicitly interleave execution of more than one periodic process on a single CPU. The cyclic executive is often implemented as a main loop with an invariant loop body known as the cyclic schedule. A *cyclic schedule* includes function calls for each periodic service provided within the major period of the overall loop. The loop may include event polling to determine when to dispatch functions, and functions that need to be called at a higher frequency than the main loop will often be called multiple times within the loop. Likewise, functions implementing periodic services that need to be run at much lower frequency than the main loop may be called only on specific loop counts or only when polled events indicate a service request. For example, the NASA Space Shuttle flight software uses a cyclic executive design for the PASS (Primary Avionics Subsystem) and has provided decades of defect-free operation, providing real-time control of a complex system [Carlow84].

The Space Shuttle PASS flight software includes the hard real-time safety-critical GN&C (Guidance, Navigation, and Control) services that are dispatched by a cyclic executive from a dispatch table configured and selected based upon the current flight stage of the shuttle (ascent, on-orbit, reentry). The highest frequency services maintain shuttle flight control and operate on a 40-millisecond period. As Carlow notes, “The high-frequency executive is scheduled at a relatively high priority to cycle at a 25 Hz rate and initiate all principal function processes directly related to vehicle flight control. Mid-frequency and low-frequency executives are scheduled at lower priorities. They initiate principal function processes, which operate at rates of 6.25 Hz down to 0.25 Hz.” The scheduling of the GN&C executives is provided by the Process Management, which uses a multitasking priority queue structure and schedules the CPU in response to requests made through a service interface that defines event frequency and priority for service requests. This higher-level scheduler handles requests from the GN&C cyclic executive: SM (Systems Management), which monitors systems for faults, and VCO (Vehicle Checkout), which is used for preflight and on-orbit coast avionics systems testing. The GN&C cyclic executive is scheduled to run periodically at 25 Hz (40-millisecond period) and includes its own dispatch table to sequence GN&C services at the high, medium, and low frequencies previously described.

The cyclic executive architecture has been an important and successful approach for hard real-time systems. Carlow explains the overall system: “[the flight software] architectures reflect a synchronous design approach

within which the dispatching of each application process is timed to always occur at a specific point relative to the start of the overall system cycle or loop.” Although the cyclic executive has been successful due to its simplicity and deterministic character, one of its drawbacks is the difficulty required to modify the cyclic schedule. For PASS, Carlow points out that “a major benefit of this approach is repeatability; however, there is only limited flexibility to accommodate change.”

The cyclic executive is often extended to handle asynchronous events with interrupts rather than relying only upon loop-based polling of inputs. This extension of the executive is called the Main+ISR design. As the name implies, this approach involves a main loop cyclic executive with the addition of ISRs (Interrupt Service Routines). The ISRs handle asynchronous events that interrupt the normal execution sequence of an embedded microprocessor. In the Main+ISR approach, the ISRs are best kept short and simple so they relay event data to the Main loop for handling. The Main+ISR approach has some advantage over the pure cyclic executive and polling for event input because it may reduce latency between event occurrence and handling. However, the Main+ISR approach has pitfalls as well. For example, if an input device malfunctions and raises interrupts at a much higher frequency than expected, significant interference to loop processing may be introduced. Although Main+ISR is more responsive to events as they occur, it may be less stable unless a concerted effort is made to protect the system for potential interrupt malfunctions related to interrupt source devices.

2.6 Scheduler Concepts

The design of a generalized RTOS scheduler for processor resources is covered well in most operating system texts. Here, we will briefly review some of the major concepts as they relate to real-time scheduling of the CPU resource. Real-time services may be implemented as threads of execution that have an execution context and are set into execution by a scheduler that determines which thread to dispatch. Dispatch is a basic mechanism to preempt the currently running thread, save its context, and restore the context of the thread to be run along with modification of the instruction pointer or program counter to start or resume execution of the new thread. The scheduler must implement the CPU sharing policy, and the dispatcher must provide the context switch for each thread of

execution. The dispatcher is required to save and restore all the state that each thread of execution uses, including the following:

1. Registers
2. Stack
3. Program counter
4. Thread state

This would be a minimum execution context and is typical of real-time schedulers. By comparison, most general-purpose operating systems, like Linux, execute threads in the context of a process and maintain a process descriptor for each thread. The process context includes much more additional state, such as IO context, shared memory, and dynamic memory allocations. The thread state is one of the best ways to understand how a scheduler works. As illustrated in Table 2.1, thread states are based upon resources needed in the thread execution context.

Table 2.1 State Transition Table for a Thread of Execution

Thread State	Description	Transition	Description
Ready	Thread is queued and ready to run, but has not been dispatched (given CPU)	Running	Thread selected for dispatch based upon scheduling policy
Running	Thread is executing on CPU	Pending	Thread needs another resource in addition to the CPU
		Delayed	Wait requested by thread
		Suspended	Thread raised unhandled exception during execution
		Ready	Thread yields CPU
		Nonexistent	Thread exits
Pending	Thread is waiting on a resource in addition to CPU	Ready	Additional resource has become available
		Suspended	Pending thread is suspended by another thread

Thread State	Description	Transition	Description
Delayed	Thread is waiting for delay period to end	Ready	Delay has expired
		Suspended	Delayed thread is suspended by another thread
Suspended	Thread has raised unhandled exception or has been suspended by command from another thread	Ready	Suspension removed by another thread—thread activated
Nonexistent	Thread has not been created or allocated resources	Ready	Thread creation and activation

Dispatch policy, how the scheduler decides which thread from the set of all those that are ready for dispatch, was the main differentiation in the taxonomy in Figure 2.8. As threads become ready to run, pointers to their context are normally placed on a ready queue by the scheduler for dispatch in the order determined by the scheduling policy. The scheduler must update the ready queue based upon new service request arrivals. The dispatcher will simply loop if the ready queue is empty. Note, however, that in VxWorks, neither the scheduler nor the dispatcher shows up as a task. Also, unlike some operating systems, VxWorks does not include an idle task in the default configuration. In VxWorks, the scheduler and dispatcher are kernel context services rather than task context services. Preemptive schedulers are driven by interrupts and task calls into the kernel API. An interrupt or API call can cause the scheduler to switch context and to potentially dispatch a new thread or allow the same thread to continue execution. In VxWorks, an interrupt or an API call made by the currently running task are the only ways that the currently running task can be preempted. A fixed-priority preemptive scheduler simply dispatches threads from the ready queue based upon a priority they have been assigned at creation unless the application adjusts the priority at runtime. Most often, if two threads have the same priority, they are dispatched on a first-come, first-served basis. Almost all RTOSs include priority preemptive schedulers with support for the basic thread states outlined in Table 2.1.

One of the major drawbacks of a priority preemptive scheduling policy is the cost or overhead of the context switch that occurs on every interrupt. Systems such as Linux and Windows, which use a time-slice preemption scheme where an OS timer tick is generated every so many milliseconds by

a programmable interval timer, have high overhead. By comparison, most RTOSs do not use a time-slice tick, and instead reschedule only when IO generates an interrupt or when a thread makes a system call that results in yielding the CPU—a delay, exit, yield, call for an unavailable resource in addition to the CPU, or suspension due to exception. Otherwise, RTOS threads normally run to completion unless an event releases a thread (makes it ready) at higher priority than the presently executing thread.

Given an MP or uniprocessor hardware architecture, if more than one service can be requested on a given processor, the next branch in the taxonomy concerns whether requests will preempt services already in progress or queue and wait for services in progress to run to completion.

2.6.1 Preemptive vs. Non-preemptive Schedulers

Non-preemptive scheduling policy has existed for general-purpose computing from the beginning of computing systems. In general, this category can be subdivided into batch and cooperative processing. In batch systems, jobs that are submitted to a work queue are dispatched by the system OS and run to completion. The two most well-known dispatch policies are FCFS (First Come First Served) and SJN (Shortest Job Next). The SJN policy has the advantage of completing the largest number of jobs per unit time, but long jobs may never be serviced, and furthermore, SJN requires an estimate of how long each job will require execution. Because real-time services are inherently characterized by producing a response relative to a request and before a deadline, neither non-preemptive policy is of interest in this text.

All real-time services must provide a response relative to requests for the service (a release), and the response is most often constrained by a deadline. At the very least, because real-time systems are request-oriented based upon real-world events, clearly these systems must support preemption or they must poll the real world on a regular basis and provide periodic service processing. Two non-preemptive approaches are sometimes used in real-time systems: data flow and multi-frequency cyclic executives. In a data flow, input interfaces are periodically checked, and when data is available, this data is processed and output is produced for consumption by the next service in the flow or terminated by producing a response. In dataflow processing, the inputs that start a flow are checked in a deterministic order most often, and flows are executed from start to finish or source to sink.

The best property of this type of scheduling policy is that it is fully deterministic—the order of execution in the service and between services (flows) is fully predictable. The disadvantage is that some flows may be known to require execution much more frequently than others—a modification to this scheme leads to a multi-frequency executive (MFE). In the MFE, specific functions (data flows) are executed at a higher frequency than others. For example, a control system data flow may require execution 100 times per second for stable operation, whereas the guidance function may require execution only 1 time per second to direct the system to a target (hopefully without losing control!). Either approach may use asynchronous interrupts or may only poll input status registers; however, context switches beyond simple interrupt servicing are not done. That is, data flows are not switched prior to completion, and executive functions are not switched prior to completion; after a flow or executive function is dispatched (given the CPU), it runs to completion without significant preemption.

Preemptive service releases have an advantage in that the service can be designed much more independently than data flows or executives. Each service can assume that it will execute following release from an interrupt as if it is the only service on the CPU, except each service must be preemptible between release and completion (generation of response). The preemption of threads, ISRs, and the kernel requires an operating system that will save and restore context for each executable service. Furthermore, each service must execute reentrant code if it is shared and must synchronize access to any globally shared resources. These requirements are true of any preemptive multi-programmed system. The services are independent in the sense that no specific cooperation between services is required unless code or data is shared. Shared code or data simply requires reentrant code and shared data access synchronization. The services can be considered to be running asynchronously other than specific synchronization points for shared resource access. Thus, each service can be designed as a separate state machine rather than one state machine composed of many smaller state machines (as is typical of executives). Also, one flow of execution may, in fact, preempt another in cases where one service is more important than another (e.g., maybe one service has a shorter deadline or more negative impact if not completed by its deadline).

Because preemption opens up the possibility that more than one service might be ready to run, and there are fewer CPU resources than services ready to run, a dispatch decision must be made. One of the simplest

dispatch policies is to give the CPU to the service assigned highest priority. These assigned priorities are never changed; they are fixed. To decide how to assign priorities, two common real-time policies are to assign highest priority to the service with the shortest release period (highest request frequency), which is known as RM (Rate Monotonic), and to assign the highest priority to the services with the shortest deadline relative to release, which is known as DM (Deadline Monotonic). Note that RM is identical to DM when the release period equals the deadline. As you will see in this chapter, the RM (and related DM) policy can be shown to provably meet system deadline requirements given deterministic release periods and service execution times. However, one major drawback is that fixed priorities do not guarantee maximum use of the CPU.

To attempt full usage of CPU resources and prove that services can meet deadlines, you are forced to consider preemptive dynamic-priority policies. Two dynamic policies have been shown to have the capability to fully use CPU and guarantee deadlines, assuming service execution times and request intervals are deterministic or bounded. Liu and Layland proposed Earliest Deadline First (EDF), along with RM, and showed them to be optimal—that is, policies that can schedule any set of services that can be scheduled (an exhaustive proof!). However, although RM is simpler, Liu and Layland also showed that RM fundamentally requires margin and less than full CPU use unless service requests are harmonic.

By comparison, in EDF, any time a new service request arrives (indicated by an interrupt asynchronous), the EDF scheduling policy adjusts all priorities so that the service with the earliest deadline is given highest priority. Ignoring the overhead of determining which service has the earliest impending deadline and ignoring overhead of priority reassignment, EDF can be shown to provide full use of the CPU where possible—even when requests intervals are not harmonic. The downside of EDF is that if request rates vary or execution times vary, the effect upon services is hard to predict—which service will miss its deadline in an overload? Variations on EDF have been proposed that intend to improve the determinism of the system given variations, including Least Laxity First (LLF). The LLF policy assigns highest priority to the service that has the least difference between remaining execution time and its deadline—a measure of which service deadline is most pressing. The idea is that laxity may vary based upon execution rates, whereas the EDF assignment is not at all influenced by execution rate.

Given this overview of resource-scheduling policies, you may wonder how these policies actually instantiate the various utility functions presented in the first section of this chapter. They don't, other than the hard real-time utility curve. If you can prove that with a policy and deterministic parameters all services will complete prior to deadlines, the only utility curve this maps to is the hard real-time curve. Recall that the hard real-time utility curve provides full credit for any response delivered before the deadline relative to service request and no credit or negative credit for late response. The isochronal utility function can be implemented as a hard real-time service with early completions held until the response deadline. This approach can be used to implement both hard and soft isochronal services. Policies and scheduling mechanisms for implementation of soft real-time utility and anytime services are open research areas.

2.6.2 Preemptive Fixed-Priority Scheduling Policy

Fixed-priority preemptive policy is most widely used by real-time embedded systems employing an RTOS. Early on, most hard real-time systems used cyclic executives or multi-frequency executives often found in systems such as the Space Shuttle GN&C primary avionics subsystem. The RTOS scheduling framework offers the same deterministic scheduling as the cyclic executive, but with far more flexibility in how services are defined and maintained. The principal reason for the pervasiveness of the RTOS in larger scale configurable real-time embedded systems is the fact that the policy has been proven optimal and a feasibility test exists, along with the ease of defining services as tasks rather than loop or interrupt contexts. The RM feasibility tests provide a method to prove that a given set of services (implemented as RTOS tasks) can be guaranteed to all meet their deadlines if the RM policy is used to assign priorities. For hard real-time systems where proof that services will not miss deadlines is desired, RM is an obvious choice. For example, RM is often used for commercial aircraft, for satellite systems, or any other system where failure to meet all deadlines can result in significant loss of life and/or assets. Finally, as you will see in this section, with an RM policy, you can predict and control exactly which services will miss deadlines in an overload scenario.

First, let's look at why the RM priority policy is optimal. Recall that the RM policy requires a framework where services are released asynchronously (typically via an interrupt) and placed on a ready queue, indicating they need to run. A scheduler then dispatches each service based upon the highest priority task in the ready queue (all services are implemented

as tasks and assumed to have unique priority). The service/task dispatched continues to run to completion unless an interrupt preempts the task presently executing. Following an interrupt, the scheduler reevaluates the ready queue and possibly dispatches a new task (a context switch) if a task of higher priority is added to the ready queue compared to that running prior to the interrupt. When a context switch occurs and a task/service is preempted prior to running to completion, this is called *interference*. In a fixed-priority preemptive system with only one service/task, interference is not possible. In a system with more than one service/task, any given task may be interfered with by all tasks that have been assigned higher priority.

With this fixed-priority preemptive framework, the scheduling problem must be further constrained to derive a formal mathematical model that proves deterministic behavior. Clearly it is impossible to prove deterministic behavior for a system that has nondeterministic inputs. Liu and Layland recognized this and proposed what they believed to be a reasonable set of assumptions and constraints on real systems to formulate a deterministic model. The assumptions and constraints are as follows:

- A1:** All services requested on periodic basis, the period is constant
- A2:** Completion-time < period
- A3:** Service requests are independent (no known phasing)
- A4:** Runtime is known and deterministic (WCET may be used)
- C1:** Deadline = period by definition
- C2:** Fixed-priority, preemptive, run-to-completion scheduling
- A5:** Critical instant—longest response time for a service occurs when all system services are requested simultaneously (maximum interference case for lowest priority service)

As noted earlier, A1 to A_n are assumptions and C1 to C_n are constraints as presented by Liu and Layland in their paper.

Given the fixed-priority preemptive scheduling framework and assumptions described in the preceding list, we can now examine alternatives for assigning priorities and identify a policy that is optimal. Showing that the RM policy is optimal is most easily accomplished by inspecting a system with a small number of services.

An example with two services follows. Given services S₁ and S₂ with periods T₁ and T₂, execution times C₁ and C₂, and release periods

$T_2 > T_1$, take, for example, $T_1 = 2$, $T_2 = 5$, $C_1 = 1$, $C_2 = 2$, and then if $\text{prio}(S_1) > \text{prio}(S_2)$, note Figure 2.9.

S_1 Makes Deadline if $\text{prio}(S_1) > \text{prio}(S_2)$

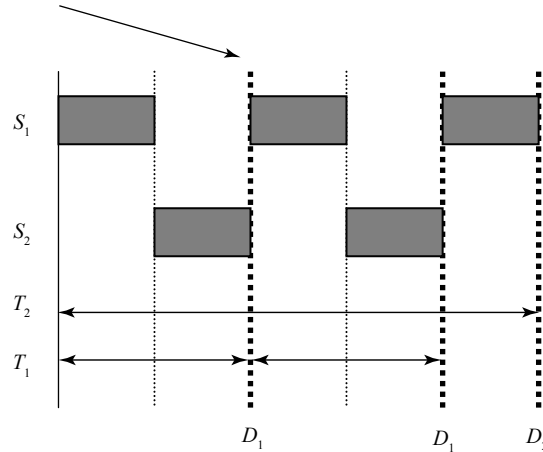


FIGURE 2.9 Example of RM Priority Assignment Policy

In this two-service example, the only other policy (swapping priorities from the preceding example) does not work. Given services S_1 and S_2 with periods T_1 and T_2 and C_1 and C_2 with $T_2 > T_1$, for example, $T_1 = 2$, $T_2 = 5$, $C_1 = 1$, $C_2 = 2$, and then if $\text{prio}(S_2) > \text{prio}(S_1)$, note Figure 2.10.

S_1 Misses Deadline if $\text{prio}(S_2) > \text{prio}(S_1)$

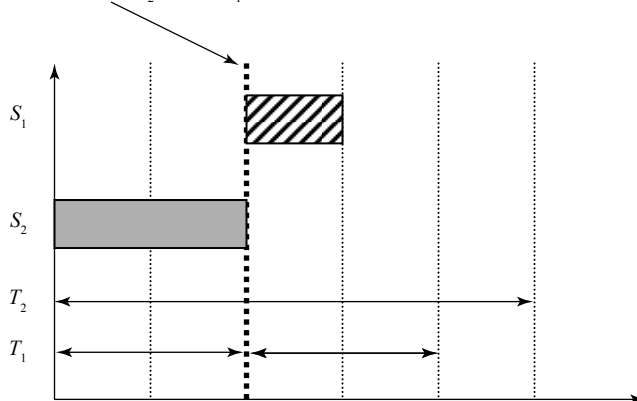


FIGURE 2.10 Example of Non-Optimal Priority Assignment Policy

The conclusion that can be drawn is that for a two-service system, the RM policy is optimal, whereas the only alternative is not optimal because the alternative policy fails when a workable schedule does exist! The same argument can be posed for a three-service system, a four-service system, and finally an N -service system. In all cases, it can be shown that the RM policy is optimal. In Chapter 3, the Rate Monotonic Least Upper Bound (RM LUB) is derived and proven. Chapter 3 also provides system scheduling feasibility tests derived from RM theory that you can use to determine whether system CPU margin will be sufficient for a real-time safe operation.

2.7 Real-Time Operating Systems

Many real-time embedded systems include an RTOS, which provides CPU scheduling, memory management, and driver interfaces for IO in addition to boot or BSP (Board Support Package) firmware. In this text, example code included is based upon either the VxWorks RTOS from Wind River Systems or Linux. The VxWorks RTOS is available for academic licensing from Wind River through the University program free of charge (see Appendix C, “Wind River University Program Information”). Likewise, Linux is freely available in a number of distributions that can be tailored for embedded platforms (see Appendix C, section titled “Real-Time Linux Distributions and Resources”). Key features that an RTOS or an embedded real-time Linux distribution should have include the following:

- A fully preemptible kernel so that an interrupt or real-time task can preempt the kernel scheduler and kernel services with priority.
- Low well-bounded interrupt latency.
- Low well-bounded process, task, or thread context switch latency.
- Capability to fully control all hardware resources and to override any built-in operating system resource management.
- Execution tracing tools.
- Cross-compiling, cross-debugging, and host-to-target interface tools to support code development on an embedded microprocessor.

- Full support for POSIX 1003.1b synchronous and asynchronous inter-task communication, control, and scheduling (Now integrated in IEEE POSIX 1003.1 2013).
- Priority inversion-safe options for mutual exclusion semaphores used with priority-preemptive run-to-completion scheduling (the mutual exclusion semaphore referred to in this text includes features that extend the early concepts for semaphores introduced by Dijkstra). Note that for the Linux CFS (Completely Fair Scheduler) inversion-safe mutual exclusion features are not needed, but if the default scheduling for threads is changed to FIFO (First-In, First-Out) in Linux and real-time priorities assigned, then some method to deal with potential unbounded priority inversion is needed (discussed in more detail in Chapter 6).
- Capability to lock memory address ranges into cache.
- Capability to lock memory address ranges into working memory if virtual.
- Memory with paging is implemented.
- High-precision time-stamping, interval timers, and real-time clocks and virtual timers.

The VxWorks, ThreadX, Nucleus, Micro-C-OS, RTEMs (Real-Time Executive for Military Systems), and many other available real-time operating systems provide the features in the preceding list. This has been the main selling point of the RTOS because it provides time-to-market acceleration compared to designing and coding a real-time executive from scratch, yet provides very direct and efficient interfacing between software applications and hardware platforms. Some RTOS options are proprietary and require licensing, and some, such as RTEMs, FreeRTOS, and Micro-C-OS, require limited or no licensing, especially for academic or personal use only (a more complete list can be found in Appendix D, “RTOS Resources”).

Linux was originally designed to be a multiuser operating system with memory protection, abstracted process domains, significantly automated resource management, and scalability for desktop and large clusters of general-purpose computing platforms. Embedding and adapting Linux for real-time require distribution packaging, kernel patches, and the addition of support tools to provide the eleven key features listed previously. Several companies support real-time Linux distributions, including TimeSys Linux,

Wind River Linux, Concurrent RedHawk Linux, and a few others (a more complete list can be found in Appendix C, section titled “Real-Time Linux Distributions and Resources”). Real-time distributions of Linux may incorporate any of the following patches to improve predictable response:

1. The Robust Mutex patch (<https://www.kernel.org/doc/Documentation/robust-futexes.txt>)
2. Linux preemptible kernel patch (https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch)
3. POSIX clocks and timers patches (http://linux.die.net/man/3/clock_gettime, http://elinux.org/High_Resolution_Timers)
4. Linux KernelShark (<http://rostedt.homelinux.com/kernelshark/>, http://elinux.org/images/6/64/Elc2011_rostedt.pdf)
5. Use of POSIX Pthreads with the FIFO scheduling class and real-time priorities, which are mapped onto Linux kernel tasks by NPTL (Native POSIX Threads Library) [Drepper03], which requires root user privileges as found in examples on the DVD.



Finally, the other option is to write your own resource-management kernel. In Chapter 13, “Performance Tuning,” you’ll see that although an RTOS provides a generic framework for resource-management and multi-service applications, the cost of these generalized features is code footprint and overhead. So, in Chapter 8, “Embedded System Components,” basic concepts for real-time kernel services are summarized. Understanding the basic mechanisms will be helpful to anyone considering development of a custom kernel. Building a custom resource-management kernel is not as daunting as it first may seem, but using a preexisting RTOS may also be an attractive option based upon complexity of services, portability requirements, time to market, and numerous system requirements. Because it is not clear whether use of an RTOS or development of a custom resource kernel is the better option, both approaches are discussed in this text.

In general, an RTOS provides a threading mechanism, in some cases referred to as a *task context*, which is the implementation of a service. A *service* is the theoretical concept of an execution context. The RTOS most often implements this as a thread of execution, with a well-known entry point into a code (text) segment, through a function, and a memory context for this thread of execution, which is called the *thread context*. In

VxWorks, this is referred to as a task and the TCB (Task Control Block). In Linux, this is referred to as a process and a process descriptor. In the case of VxWorks, the context is fairly small, and in the case of a Linux process, the context is relatively complex, including IO status, memory usage context, process state, execution stack, register state, and identifying data. In this book, *thread* will be used to describe the general implementation of a service (at the very least a thread of execution), *task* to describe an RTOS implementation, and *process* to describe the typical Linux implementation. Note that Linux FIFO threads are similar in many ways to a VxWorks task because with NPTL these threads are mapped directly onto Linux kernel tasks.

Typical RTOS CPU scheduling is fixed-priority preemptive, with the capability to modify priorities at runtime by applications, therefore also supporting dynamic-priority preemptive. Real-time response with bounded latency for any number of services requires preemption based upon interrupts. Systems where latency bounds are more relaxed might instead use polling for events and run threads to completion, increasing efficiency by avoiding disruptive asynchronous context switch overhead due to interrupts. Remember, as presented in Chapter 1, we assume that response latency must be deterministic and that this is more important than throughput and overall efficiency. If you are designing an embedded system that does not really have real-time requirements, avoid asynchronous interrupts altogether. Dealing with asynchronous interrupts requires debugging in interrupt context and can add complexity that might be fully avoidable in a non-real-time system. First, what really constitutes a real-time deadline requirement must be understood. By completion of Chapter 3, you should be able to clearly recognize whether a system requires priority preemptive scheduling. An RTOS provides priority preemptive scheduling as a mechanism that allows an application to implement a variety of scheduling policies:

- RM (Rate Monotonic) or DM (Deadline Monotonic), fixed priority
- EDF (Earliest Deadline First) or LLF (Least Laxity First), dynamic priority
- Simple run-to-completion cooperative tasking

These policies are a small subset of the larger taxonomy of scheduling policies presented previously (refer to Figure 2.8). In the case of simple

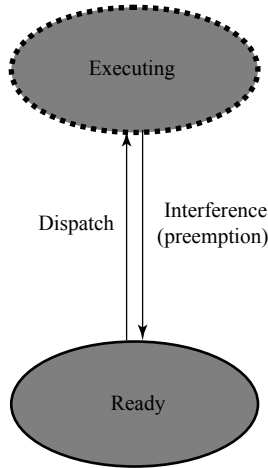


FIGURE 2.11 Basic Dispatch and Preemption States

run-to-completion cooperative tasking, the value of the RTOS is mostly the BSP, boot, driver, and memory management because the application really handles the scheduling. Given that bounded latency is most often a hard requirement in any real-time system, the focus is further limited to RM, EDF, and LLF. Ultimately, a real-time scheduler needs to support dispatch, execution context management, and preemption. In the simplest scenario, where services run to completion, but may be preempted by higher-priority service, the thread states are depicted in Figure 2.11.

In the state transition diagram shown in Figure 2.11, we assume that a thread in execution never has to wait for any resources in addition to the CPU, that it never encounters an unrecoverable error, and that there is never a need to delay execution. If this could be guaranteed, the scheduling for this type of system is fairly simple. All services can be implemented as threads, with a stack, a register state, and a thread state of executing or ready on a priority ordered queue. Most often, threads that implement services operate on memory or on an IO interface. In this case, the memory or IO is a secondary resource, which if shared or if significant latency is associated with use, may require the thread to wait and enter a pending state until this secondary resource becomes available. In Figure 2.12, we add a pending state, which a thread enters when a secondary resource is not immediately available during execution. When this secondary resource becomes available—for example, when a device has data available—that resource can set an atomic test-and-set flag (a semaphore), indicating availability to the scheduler to transition the pending thread back to the ready state.

In addition, if a thread may be arbitrarily delayed by a programmable amount of time, then it will need to enter a delayed state, as shown in Figure 2.13. A delay is simply implemented by a hardware interval timer that provides an interrupt after a programmable number of CPU clock cycles or external interval-timer count that is incremented by dedicated oscillator cycles. When the timer is set, an interrupt handler for the expiration is installed so that the delay timeout results in restoration of the thread from delayed state back to ready state.

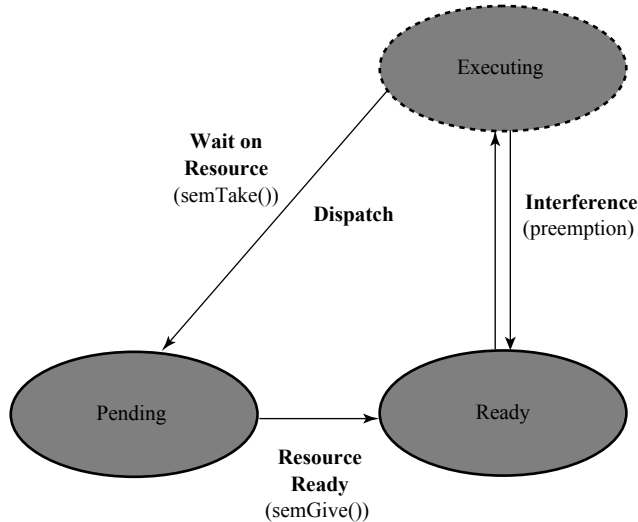


FIGURE 2.12 Basic Service States Showing Pending on Secondary Resource

In addition, if a thread may be arbitrarily delayed by a programmable amount of time, then it will need to enter a delayed state, as shown in Figure 2.13. A delay is simply implemented by a hardware interval timer that provides an interrupt after a programmable number of CPU clock cycles or external oscillator cycles. When the timer is set, an interrupt handler for the expiration is installed so that the delay timeout results in restoration of the thread from delayed state back to ready state.

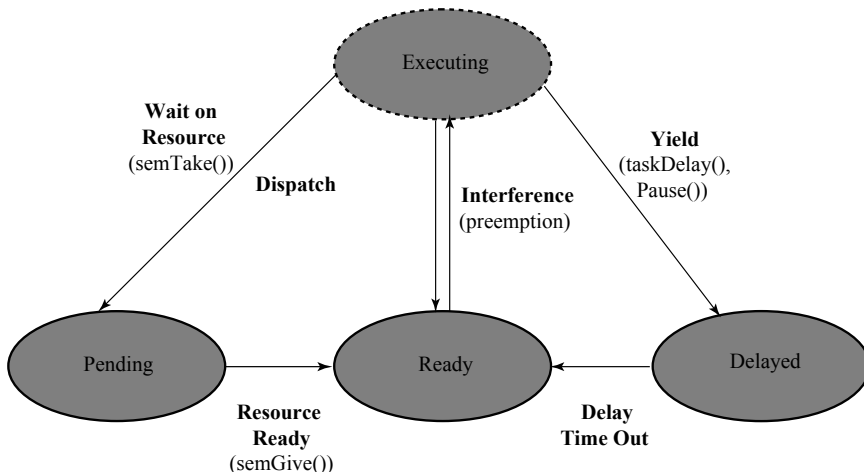


FIGURE 2.13 Basic Service States Including Programmed Delays

Finally, if a thread of execution encounters a non-recoverable error—for example, division by zero in the code—then continuation could lead to significant system endangerment. In the case of division by zero, this will cause an overflow result, which in turn might generate faulty command output to an actuator, such as a satellite thruster, which could cause loss of the asset. If the division by zero is handled by an exception handler that recalculates the result and therefore recovers within the service, continuation might be possible, but often recovery is not possible.

Because the very next instruction might cause total system failure, a non-recoverable exception should result in suspension of that thread. Figure 2.14 shows the addition of a suspended state. If a thread or task that is already in the delayed state can also be suspended by another task, as is the case with VxWorks, then additional states are possible in the suspended state, including delayed+suspended, pending+suspended, and simple suspended.

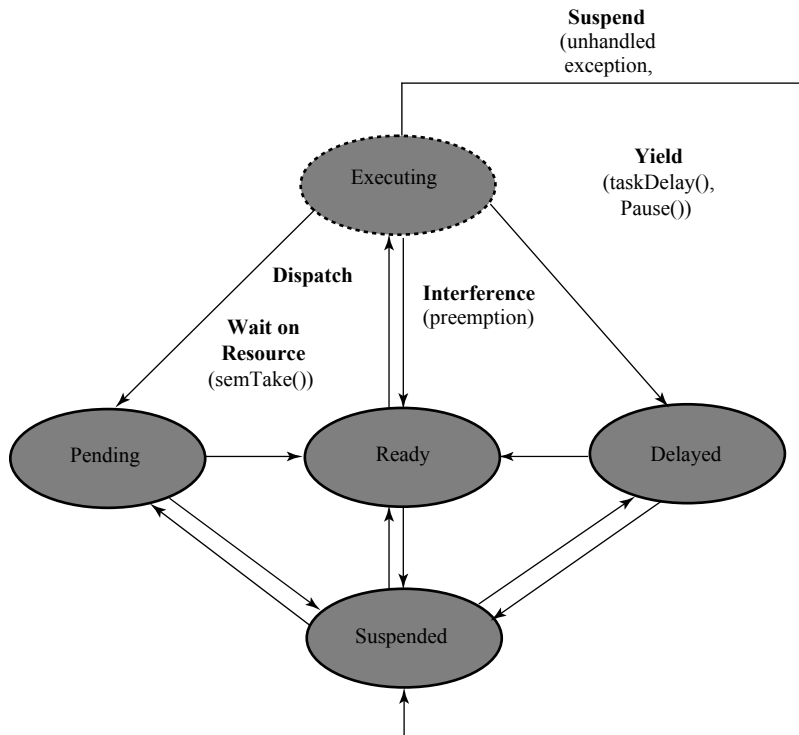


FIGURE 2.14 Service States Including Programmed Suspension or Suspension Due to Exception

Although scheduling the CPU for multiple services with the implementation of threads, tasks, or processes is the main function of an RTOS, the RTOS also provides management of memory and IO along with methods to synchronize and coordinate usage of these secondary resources along with the CPU. Secondary resources lead to the addition of the pending state if their availability at runtime can't be guaranteed in advance. Furthermore, if two threads must synchronize, one thread may have to enter the pending state to wait for another thread to execute up to the rendezvous point in its code. In Chapter 6, you'll see that this seemingly simple requirement imposed by secondary resources and the need for thread synchronization leads to significant complexity.

The RTOS provides IO resource management through a driver interface, which includes common entry points for reading/writing data, opening/closing a session with a device by a thread, and configuring the IO device. The coordination of access to devices by multiple threads and the synchronization of thread execution with device data availability are implemented through the pending state. In the simplest case, an Interrupt Service Routine (ISR) can indicate device data availability by setting a semaphore (flag), which allows the RTOS to transition the thread waiting for data to process from pending back to the ready state. Likewise, when a thread wants to write data to an IO output buffer, if the buffer is currently full, the device can synchronize buffer availability with the thread again through an ISR and a binary semaphore. In Chapter 8, you'll see that all IO and thread synchronization can be handled by ISRs and binary semaphores, but that alternative mechanisms, such as message queues, can also be used.

Memory in the simplest scenarios can be mapped and allocated once during boot of the system and never modified at runtime. This is the ideal scenario because the usage of memory is deterministic in space and time. Memory usage may vary, but the maximum used is predetermined, as is the time to claim, use, and release memory. In general, the use of the C library ***malloc*** is frowned upon in real-time embedded systems because this dynamic memory-management function provides allocation and de-allocation of arbitrary segments of memory. Over time, if the segments truly are of arbitrary size, the allocation segments must be coalesced to avoid external fragmentation of memory, as shown in Figure 2.15.

Likewise, if arbitrarily sized segments are mapped onto minimum size blocks of memory (e.g., 4 KB blocks), then allocation of a 1-byte buffer

will require 4,096 bytes to be allocated, with 4,095 bytes within this block wasted. This internal fragmentation is shown in Figure 2.16.

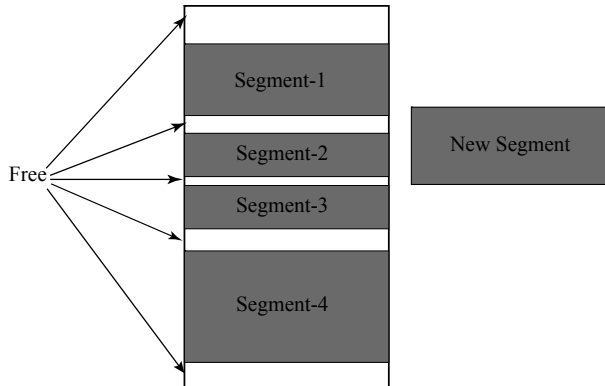


FIGURE 2.15 Memory Fragmentation for Data Segments or Arbitrary Size

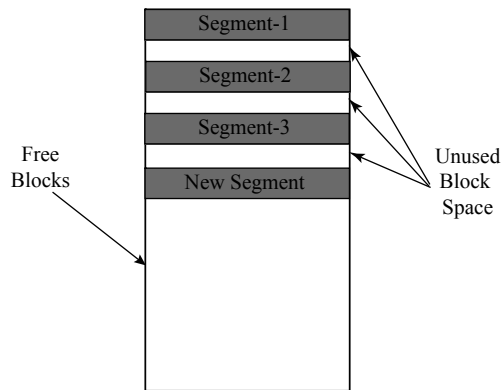


FIGURE 2.16 Internal Block Fragmentation for Fixed-Size Dynamic Allocation

In many real-time embedded systems, a compromise for memory management can be reached whereby most working data segments are predetermined, and specific usage heaps (dynamically allocated blocks of memory) can be defined that have little to no external or internal fragmentation issues.

Overall, because an RTOS provides general mechanisms for CPU, memory, and IO resource management, adopting an off-the-shelf RTOS, such as VxWorks or embedded Linux, is an attractive choice. However, if the resource usage and management requirements are well understood and simple, developing a custom resource kernel may be a good alternative.

2.8 Thread-Safe Reentrant Functions

When an RTOS scheduling mechanism is used, care must be taken to consider whether functions may be called by multiple thread contexts. Many real-time services may use common utility functions, and a single implementation of these common functions will save significant code space memory. However, if one function may be called by more than one thread and those threads are concurrently active, the shared function must be written to provide reentrant function support so that it is thread-safe. Threads are considered concurrently active if more than one thread context is either executing or awaiting execution in the ready state in the dispatch queue. In this scenario, thread A might have called function F and could have been preempted before completing execution of F by thread B, which also calls function F. If F is a pure function that uses no global data and operates only on input parameters through stack, then this concurrent use is safe. However, if F uses any global data, this data may be corrupted and/or the function may produce incorrect results. For example, if F is a function that retrieves the current position of a satellite stored as a global state and returns a copy of that position to the caller, the state information could be corrupted, and inconsistent states could be returned to both callers. For this example, assume function F is defined as follows:

```
typedef struct position {double x, y, z;} POSITION;
POSITION satellite_pos = {0.0, 0.0, 0.0};
POSITION get_position(void)
{
    double alt, lat, long;
    read_altitude(&alt);
    read_latitude(&lat);
    read_longitude(&long);

    /* Multiple function calls are required to convert the
    geodetic navigational sensor state to a state in inertial
    coordinates
    */
    satellite_pos.x = update_x_position(alt, lat, long);
    satellite_pos.y = update_y_position(alt, lat, long);
    satellite_pos.z = update_z_position(alt, lat, long);
    return satellite_pos;
}
```

Now, if thread A has executed up to the point where it has completed its call to `update_x_position`, but not yet to the rest of `get_position`, and thread B preempts A, updating the position fully, thread A will be returned an inconsistent state. The state A returned will include a more recent value for `x` and older values for `y` and `z` because the `alt`, `lat`, and `long` variables are on the stack that is copied for each thread context. Thread B will be returned a consistent copy of the state updated from a set of sensor readings made in one call, but A will have the `x` value from B's call and will compute `y` and `z` from the `alt`, `lat`, `long` values on its stack sampled at an earlier time. The function as written here is not thread-safe and is not reentrant due to the global data, which is updated so that interrupts and preemption can cause partial update of the global data. VxWorks provides several mechanisms that can be used to make functions reentrant. One strategy is to protect the global data from partial updates by preventing preemption for the update critical section using `taskLock()` and `taskUnlock()`. The solution to make this function thread-safe using `taskLock()` and `taskUnlock()` is

```
typedef struct position {double x, y, z;} POSITION;
POSITION satellite_pos = {0.0, 0.0, 0.0};
POSITION get_position(void)
{
    double alt, lat, long;
    POSITION current_satellite_pos;
    read_altitude(&alt);
    read_latitude(&lat);
    read_longitude(&long);

    /* Multiple function calls are required to convert the
       geodetic navigational sensor state to a state in iner-
       tial coordinates. The code between Lock and Unlock is the
       critical section.
    */
    taskLock();
    current_satellite_pos.x = update_x_position(alt, lat, long);
    current_satellite_pos.y = update_y_position(alt, lat, long);
    current_satellite_pos.z = update_z_position(alt, lat, long);
    satellite_pos = current_satellite_pos; /* assumes struc-
       ture assignment */
    taskUnlock();
    return current_satellite_pos;
}
```

The use of Lock and Unlock prevents the return of an inconsistent state to either function because it prevents preemption during the update of the local and global satellite position. The function is now thread-safe, but potentially will cause a higher-priority thread to wait upon a lower-priority thread to complete this critical section of code. The VxWorks RTOS provides alternatives, including task variables (copies of global variables maintained with task context), interrupt level Lock and Unlock, and an inversion-safe mutex. The simplest way to ensure thread safety is to avoid the use of global data and to implement only pure functions that use only local stack data; however, this may be impractical. Chapter 8 provides a more detailed discussion of methods to make common library functions thread-safe.

Summary

A real-time embedded system should be analyzed to understand requirements and how they relate to system resources. The best place to start is with a solid understanding of CPU, memory, and IO requirements so that hardware can be properly sized in terms of CPU clock rate (instructions per second), memory capacity, memory access latency, and IO bandwidth and latency. After you size these basic three resources, determine resource margin requirements, and establish determinism and reliability requirements, then you can further refine the hardware design for the processing platform to determine power, mass, and size. In many cases, an RTOS provides a quick way to provide management of CPU, memory, and IO along with basic firmware to boot the software services for an application. Even if an off-the-shelf RTOS is used, it is important to understand the common implementations of these resource-management features along with theory on resource usage policy. In Chapter 3, the focus is CPU resource management; in Chapter 4, it is IO interface management; in Chapter 5, memory management; and in Chapter 6 multiresource management. In general, for real-time embedded systems, service response latency is a primary consideration, and overall, the utility of the response over time should be well understood for the application being designed. Traditional hard real-time systems require response by deadline or the system is said to have failed. In Chapter 7, we consider soft real-time systems where occasional missed deadlines are allowed and missed deadline handling and service recovery are designed into the system.

Exercises

1. Provide an example of a hard real-time service found in a commonly used embedded system and describe why this service utility fits the hard real-time utility curve.
2. Provide an example of a hard real-time isochronal service found in a commonly used embedded system and describe why this service utility fits the isochronal real-time utility curve.
3. Provide an example of a soft real-time service found in a commonly used embedded system and describe why this service utility fits the soft real-time utility curve.
4. Implement a VxWorks task that is spawned at the lowest priority level possible and that calls `semTake` to enter the pending state, waiting for an event indicated by `semGive`. From the VxWorks shell, start this task and verify that it is in the pending state. Now, call a function that gives the semaphore the task is waiting on and verify that it completes execution and exits.
5. Implement a Linux process that is executed at the default priority for a user-level application and waits on a binary semaphore to be given by another application. Run this process and verify its state using the `ps` command to list its process descriptor. Now, run a separate process to give the semaphore causing the first process to continue execution and exit. Verify completion.
6. Read *Liu and Layland's RMA paper* (don't get hung up on math). Please summarize the paper's main points (at least three or more) in a short paragraph.
7. Write a paragraph comparing the RM and the Deadline Driven or EDF (Earliest Deadline First) policies described in Liu and Layland in your own words and be complete, but keep it to one reasonably concise paragraph. Note that the Deadline Driven scheduling in Liu and Layland is now typically called a dynamic-priority EDF policy. In general a number of policies that are deadline-driven have evolved from the

Deadline Driven scheduling Liu and Layland describe, including EDF and Least Laxity First. Give at least two specific differences.

8. Cross Compiling Code for VxWorks:

You will need to download the file “two_tasks.c” from the DVD example code.



Create a Tornado project that includes this file, and download the file to a lab target. Using the *windshell* use the function *moduleShow* to verify that the object code has been downloaded. Submit evidence that you have run *moduleShow* in your lab report (copy and paste into your report or do a **Ctrl+PrntScrn**).

Next, still in the *windshell*, do an *lkup* “test_task” to verify that the function entry point for two_tasks example has been dynamically linked into the kernel symbol table. Place evidence in your report.

Note that typing “help” on a *windshell* command line will provide a summary of these basic commands. In addition, all commands can be looked up in the VxWorks API manual.

Capture the output from the *moduleShow* command in the Windshell and paste it into your lab write-up to prove that you’ve done this.

9. Notice the two functions test_tasks1() and test_tasks2() in the two_tasks.c file. Describe what each of these functions do. Run each function in the *windshell* by typing the function name at the *windshell* prompt. Explain the difference as to how synchronization of tasks is achieved in two different functions—namely, test_task1 and test_task2.

10. Run the Windview (now called Systemviewer) tool and capture output while the test_tasks program is running. Explain what you see—please identify your tasks, where two tasks are running on the CPU, and where they synchronize with semTake and semGive. Provide a detailed explanation and annotate the trace directly by using “**Ctrl+PrntScrn**” and “Ctrl-v” to paste an image into your write-up and mark up the

graphic directly. Describe the difference between `test_tasks1()` and `test_tasks2()` and how this is shown by the Windview (Systemviewer) output.

11. Now modify the `two_tasks.c` code so that you create a third task named “mytesttask” and write an entry point function that calls “`pause()`” and nothing else. Use the “i” command in the windshell and capture output to prove that you created this task. Does this task show execution on Windview? Why or why not?
12. For the next portion of the lab, you will experiment with creating a bootable project for the simulator. Create a bootable project based upon the Simulator BSP and configure a new kernel image so that it includes POSIX message queues. Build this kernel and launch the simulator with this image instead of the default image. Download an application project with the `posix_mq.c` DVD code, and if you did the first part right, you will get no errors on download. Do `lkup` “send” and you should see the function entry points in the `posix_mq.c` module. Cut and paste this output into your submission to show you did this.
13. Set a break point in function sender with the **windshell** command “b sender.” Now run `mq_demo` from the windshell. When the break point is hit, start a Debug session using the “bug” icon. Use the Debug menu and select “Attach...” and attach to the “Sender” task. You should see a debug window with the cursor sitting at the sender function entry point. Switch the View to “Mixed Source and Disassembly” and now count the number of instructions from the sender entry up to the call to `mq_open`. How many steps are there from a break point set at this entry point until you enter `mq_open` using the single step into? Provide a capture showing the break point output from your **windshell**.



Chapter References

- [Bovet00] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly, 2000.
- [BriRoy99] Loïc Briand and Daniel Roy, *Meeting Deadlines in Hard Real-Time Systems—The Rate Monotonic Approach*. IEEE Computer Society Press, 1999.

- [Burns91] A. Burns, “Scheduling Hard Real-Time Systems: A Review,” *Software Engineering Journal* Volume 6, Issue 3, pp. 116-128, (May 1991).
- [Carlow] Gene D. Carlow, “Architecture of the Space Shuttle Primary Avionics Software System”, *Communications of the ACM*, Volume 27, Number 9, September 1984.
- [Drepper03] Ulrich Drepper and Ingo Molnar, The Native POSIX Thread Library for Linux, February 2003, Red Hat Inc., posted on the website, http://www.cs.utexas.edu/~witchel/372/lectures/POSIX_Linux_Threading.pdf
- [Robbins96] *Practical Unix Programming—A Guide to Concurrency, Communication, and Multithreading*, Prentice Hall, 1996 (ISBN 0-13-443706-3).
- [WRS99] *VxWorks Programmer’s Guide*, Wind River Systems, 1999.

PROCESSING

In this chapter

- Introduction
- Preemptive Fixed-Priority Policy
- Feasibility
- Rate-Monotonic Least Upper Bound
- Necessary and Sufficient Feasibility
- Deadline-Monotonic Policy
- Dynamic-Priority Policies

3.1 Introduction

Processing input data and producing output data for a system response in real time do not necessarily require large CPU resources, but rather careful use of CPU resources. Before considering how to make optimal use of CPU resources in a real-time embedded system, you must first better understand what is meant by processing in real time. The mantra of real-time system correctness is that the system must not only produce the required output response for a given input (functional correctness), but also do so in a timely manner (before a deadline). A *deadline* in a real-time system is a relative time after a service request by which time the system must produce a response. The relative deadline seems to be a simple concept, but a more formal specification of real-time services is helpful due to the many types of applications. For example, the processing in a voice or video real-time

system is considered high-quality if the service continuously provides output neither too early nor too late, without too much latency and without too much jitter between frames. Similarly, in digital control applications, the ideal system has a constant time delay between sensor sampling and actuator outputs. However, by comparison, a real-time service that monitors a satellite's health and status and initiates a safe recovery sequence when the satellite is in danger must enter the recovery as quickly as possible after the dangerous condition is detected. Digital control and continuous media (audio and video) are isochronal real-time services. Responses should not be generated too long after or too early after a service request. System health monitoring, however, is a simple real-time service where the system must produce a response (initiate the recovery sequence) no later than some deadline following the request.

3.2 Preemptive Fixed-Priority Policy

Given that the RM priority assignment policy is optimal, as shown in the previous chapter, we now want to determine whether a proposed set of services is feasible. By feasible, we mean that the proposed set of services can be scheduled given a fixed and known amount of CPU resource. One such test is the RM LUB: Liu and Layland proposed this simple feasibility test they call the RM Least Upper Bound (RM LUB). The RM LUB is defined as

$$U = \sum_{i=1}^m (C_i/T_i) \leq m(2^{1/m} - 1)$$

U: Utility of the CPU resource achievable

C_i: Execution time of Service i

m: Total number of services in the system sharing common CPU resources

T_i: Release period of Service i

Without much closer inspection of how this RM LUB was derived, it is purely magical and must be accepted on faith.

Rather than just accept the RM LUB, we can instead examine the properties of a set of services and their feasibility and generalize this information: can we in general determine a feasibility test for a set of proposed services sharing a CPU according to RM policy?

To answer this question, we can simply diagram the timing for release, preemption, dispatch, and completion of a set of services from the critical instant and later. The critical instant assumes that in the worst case, all services might be requested at the same time. If we do attempt to understand a system by diagramming, for what period of time must the system be analyzed? If we analyze the scheduling with the RM policy for some arbitrary period, we may not observe the release and completion of all services in the proposed set. So, to observe all services, we at least need to diagram the service execution over a period equal to or greater than the largest period in the set of services. You'll see that if we really want to understand the use of the system, we actually must diagram the execution over the least common multiple of all periods for the proposed set of services, or *LCM* time. These concepts are best understood by taking a real example. Let's also see how this real example compares to the RM LUB.

For a system, can all C_s fit in the largest T over LCM (Least Common Multiple) time? Given Services S_1 , S_2 with periods T_1 and T_2 and C_1 and C_2 , assume $T_2 > T_1$, for example, $T_1 = 2$, $T_2 = 5$, $C_1 = 1$, $C_2 = 1$; and then if $\text{prio}(S_1) > \text{prio}(S_2)$, you can see that they can by inspecting a timing diagram, as shown in Figure 3.1.

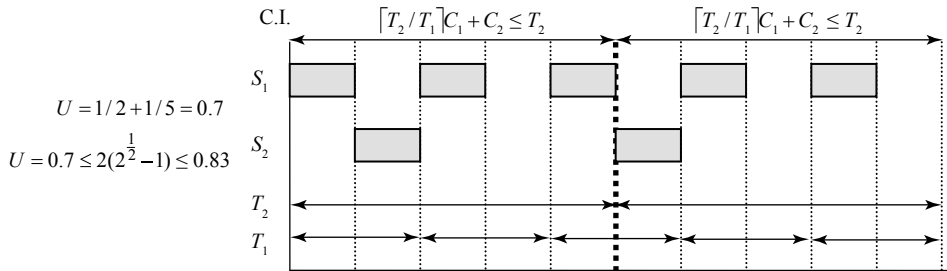


FIGURE 3.1 Example of Two-Service Feasibility Testing by Examination

The actual utilization of 70% is lower than the RM LUB of 83.3%, and the system is feasible by inspection. So, the RM LUB appears to correctly predict feasibility for this case.

Why did Liu and Layland call the RM LUB a least upper bound? Let's inspect this bound a little more closely by looking at a case that increases utility, but remains feasible. Perhaps we can even exceed their RM LUB.

In this example, RM LUB is safely exceeded, given Services S_1 , S_2 with periods T_1 and T_2 and C_1 and C_2 ; and assuming $T_2 = T_1$, for example, $T_1 = 2$, $T_2 = 5$, $C_1 = 1$, $C_2 = 2$; and then if $\text{prio}(S_1) > \text{prio}(S_2)$, note Figure 3.2.

By inspection, this two-service case with 90% utility is feasible, yet the utility exceeds the RM LUB. So, what good is the RM LUB? The RM LUB is a pessimistic feasibility test that will fail some proposed service sets that actually work, but it will never pass a set that doesn't work. A more formal way of describing the RM LUB is that it is a sufficient feasibility test, but not necessary.

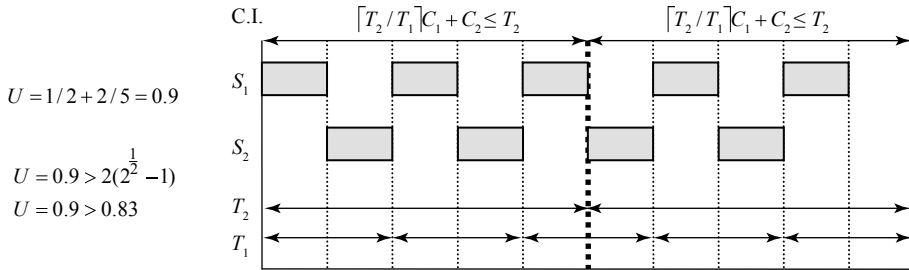


FIGURE 3.2 Example of Two-Service Case Exceeding RM LUB

The formal differences between sufficient, necessary and necessary and sufficient

What exactly is meant by a sufficient condition or a necessary and sufficient condition? *Necessary*, *sufficient*, and *necessary and sufficient* have well-defined meaning in logic. As defined in Wikipedia, the free online encyclopedia, here are the standard definitions of all three:

Necessary condition: To say that A is *necessary* for B is to say that B *cannot* occur without A occurring or that *whenever* (wherever, etc.) B occurs, so does A. Drinking water regularly is necessary for a human to stay alive. If A is a necessary condition for B, then the logical relation between them is expressed as “If B then A” or “B only if A” or “ $B \rightarrow A$.”

Sufficient condition: To say that A is *sufficient* for B is to say precisely the converse: that A cannot occur without B, or whenever A occurs, B occurs. That there is a fire is sufficient for there being smoke. If A is a sufficient condition for B, then the logical relation between them is expressed as “If A then B” or “A only if B” or “ $A \rightarrow B$.”

Necessary and sufficient condition: To say that A is necessary and sufficient for B is to say two things: (1) A is necessary for B and (2) A is sufficient for B. The logical relationship is therefore “A if and only if B.” In general, to prove “P if Q,” it is equivalent to proving both the statements “if P, then Q” and “if Q, then P.”

For real-time scheduling feasibility tests, sufficient therefore means that passing the test guarantees that the proposed service set will not miss deadlines; however, failing a sufficient feasibility test does not imply that the proposed service set will miss deadlines. An N&S (Necessary and Sufficient) feasibility test is exact—if a service set passes the N&S feasibility test, it will not miss deadlines, and if it fails to pass the N&S feasibility test, it is guaranteed to miss deadlines. Therefore, an N&S feasibility test is more exact compared to a sufficient feasibility test. The sufficient test is conservative and, in some scenarios, downright pessimistic. The RM LUB is useful, however, in that it provides a simple way to prove that a proposed service set is feasible. The sufficient RM LUB does not prove that a service set is infeasible—to do this, you must apply an N&S feasibility test. Presently, the RM LUB is the least complex feasibility test to apply. The RM LUB is $O(n)$ order n complexity—requiring summation of service execution times and comparison to a simple expression that is a function of the number of services in the proposed set. By comparison, the only known N&S feasibility tests for the RM policy are $O(n^3)$ —requiring iteration loops bounded by the number of services nested twice so that three nested loops must be executed (as shown in the “Feasibility” section of this chapter). If the feasibility test is performed once during design, then it makes sense to test the proposed service set with an N&S test. However, for quick calculations, “back of the envelope calculations,” the RM LUB is still useful. In some cases, it also may be useful to perform a feasibility test in real time. In this case, the system providing real-time services might receive a request to support an additional service with known RM properties (period, execution time, and deadline) while presently running an existing service set—this is called *online* or *on-demand scheduling*; for this type of system, the feasibility test itself requires resources for evaluation—keeping the requirements of the feasibility test minimal may be advantageous. Either way, for online scheduling, the feasibility test is a service itself and requires resources like any other service.

Now that you understand how the RM LUB is useful, let’s see how the RM LUB is derived. After understanding the RM LUB derivation, N&S feasibility algorithms are easier to understand as well. Finally, much like the demonstration that the RM policy is optimal with two services, it’s easier to derive the RM LUB for two services (if you want to understand the full derivation of the RM LUB for an unlimited number of services, see Liu and Layland’s paper [Liu73]).

3.3 Feasibility

Feasibility tests provide a binary result that indicates whether a set of services (threads or tasks) can be scheduled given their C_i , T_i , and D_i specification. So the input is an array of service identifiers (S_i) and specifications for each, and the output is TRUE if the set can be safely scheduled so that none of the deadlines will be missed and FALSE if any one of the deadlines might be missed. There are two types of feasibility tests:

- Sufficient
- Necessary and Sufficient (N&S)

Sufficient feasibility tests will always fail a service set that is not real-time-safe (i.e., that can miss deadlines). However, a sufficient test will also fail a service set that is real-time-safe occasionally as well. Sufficient feasibility tests are not precise. The sufficient tests are conservative because they will never pass an unsafe set of services. N&S tests are precise. An N&S feasibility test will not pass a service set that is unsafe and likewise will not fail any test that is safe. The RM LUB is a sufficient test and therefore safe, but it will fail service sets that actually can be safely scheduled.

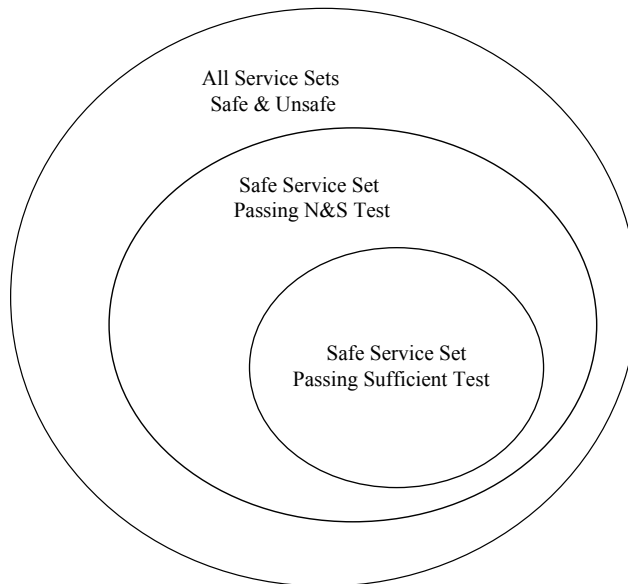


FIGURE 3.3 Relationship between Sufficient and N&S Feasibility Tests

By comparison, the Scheduling Point and Completion tests for the RM policy are N&S and therefore precise. Examples showing the imprecise but safe characteristic of the RM LUB are examined in the following section. It will also become clear that service sets with relatively harmonic periods can easily fail the sufficient RM LUB and be shown to be safe when analyzed—the more precise N&S feasibility tests will correctly predict such harmonic service sets as safe that may not pass the RM LUB. The N&S test will precisely identify the safe service set. The sufficient tests are yet another subset of the N&S safe subset, as depicted in Figure 3.3.

3.4 Rate-Monotonic Least Upper Bound

Taking the same two-service example shown earlier in Figure 3.2, we have the following set of proposed services. Given Services S_1 , S_2 with periods T_1 and T_2 and execution times C_1 and C_2 , assume that the services are released with $T_1 = 2$, $T_2 = 5$, execute deterministically with $C_1 = 1$, $C_2 = 2$, and are scheduled by the RM policy so that $\text{prio}(S_1) > \text{prio}(S_2)$. If this proposed system can be shown to be feasible so that it can be scheduled with the RM policy over the LCM (Least Common Multiple) period derived from all proposed service periods, then the Lehoczky, Sha, and Ding theorem [Briand99] guarantees it real-time-safe. The theorem is based upon the fact that given the periodic releases of each service, the LCM schedule will simply repeat over and over, as shown in Figure 3.4.

Note that there can be up to $\lceil T_2/T_1 \rceil$ releases of S_1 during T_2 as indicated by the #1, #2, and #3 execution traces for S_1 in Figure 3.4. Furthermore, note that in this particular scenario the utilization U is 90%.

The CI (Critical Instant) is a worst-case assumption that the demands upon the system might include simultaneous requests for service by all services in the system! This eliminates the complexity of assuming some sort of known relationship or phasing between service requests and makes the RM LUB a more general result.

Given this motivating two-service example, we can now devise a strategy to derive the RM LUB for any given set of services S for which each service S_i has an arbitrary C_i , T_i . Taking this example, we examine two cases:

Case 1[Figure 3.4]: C_1 short enough to fit all three releases in T_2 (fits S_2 critical time zone)

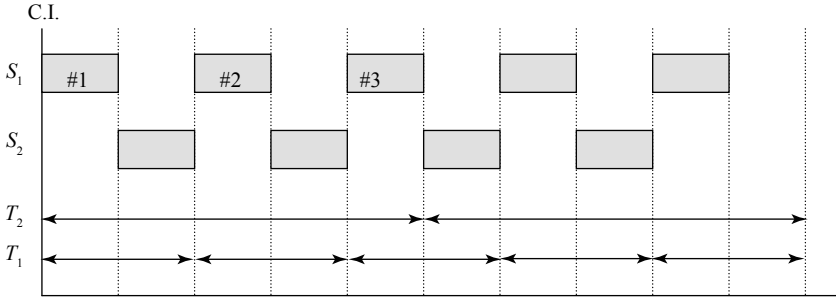


FIGURE 3.4 Two-Service Example Used to Derive RM LUB

Case 2 [Figure 3.7]: C_1 too large to fit last release in T_2 (doesn't fit S_2 critical time zone)

Examine U in both cases to find common U upper bound. The critical time zone is depicted in Figure 3.5.

In Case 1, all three S_1 releases requiring C_1 execution time fit in T_2 as shown in Figure 3.5. This is expressed by

$$C_1 \leq T_2 - T_1 \left\lfloor T_2 / T_1 \right\rfloor \quad (3.1)$$

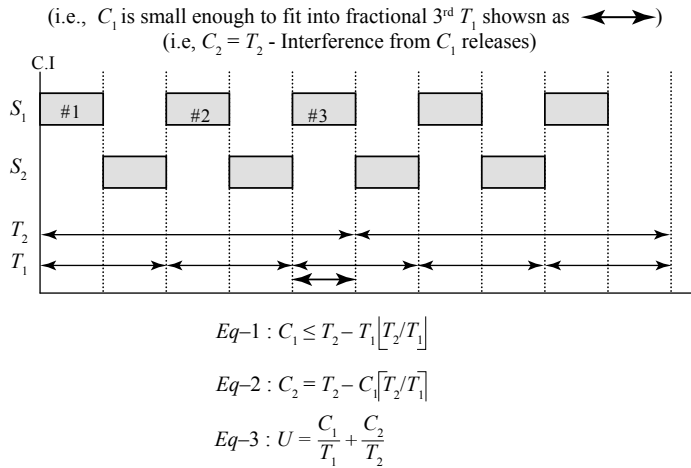


FIGURE 3.5 Example of Critical Time Zone

Note that the expression is the minimum number of times that T_1 occurs fully during T_2 . The expression is the fractional amount of time that the third occurrence of T_1 overlaps with T_2 . Equation 3.2 simply expresses the length of C_2 to be long enough to use all time not used by S_1 —note that

when the third release of S_1 just fits in the critical time zone shown by the arrow beneath release #3 of S_1 in time segment five, then S_1 occurs exactly $\lceil T_2/T_1 \rceil$ times during T_2 .

$$C_2 = T_2 - C_1 \lceil T_2 / T_1 \rceil \quad (3.2)$$

From Figure 3.5 it should be clear that if C_1 was increased and C_2 held constant, the schedule would not be feasible; likewise if C_2 was increased and C_1 held constant, they just fit! It is also interesting to note that, looking over the LCM, we do not have full utility, but rather 90%. Equation 3.3 defines the utility for the two services.

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} \quad (3.3)$$

At this point, let's plug the expression for C_2 in Equation 3.2 into Equation 3.3, which creates Equation 3.4:

$$U = \frac{C_1}{T_1} + \frac{\lceil T_2 - C_1 \lceil T_2 / T_1 \rceil \rceil}{T_2} \quad (3.4)$$

Now, simplify by the following algebraic steps:

$$U = \frac{C_1}{T_1} + \frac{T_2}{T_2} + \frac{\lceil -C_1 \lceil T_2 / T_1 \rceil \rceil}{T_2} \quad (\text{pull out } T_2 \text{ term})$$

$$U = \frac{C_1}{T_1} + 1 + \frac{\lceil -C_1 \lceil T_2 / T_1 \rceil \rceil}{T_2} \quad (\text{note that } T_2 \text{ term is } 1)$$

$$U = 1 + C_1 \left[\left(1 / T_1 \right) - \frac{\lceil T_2 / T_1 \rceil}{T_2} \right] \quad (\text{combine } C_1 \text{ terms})$$

This gives you Equation 3.5:

$$U = 1 + C_1 \left[\left(1 / T_1 \right) - \frac{\lceil T_2 / T_1 \rceil}{T_2} \right] \quad (3.5)$$

What is interesting about Equation 3.5 is that U *monotonically decreases with increasing* C_1 when $(T_2 > T_1)$. Recall that T_2 must be greater than T_1

given the RM priority policy assumed here. The term $\left[(1/T_1) - \frac{\lceil T_2/T_1 \rceil}{T_2} \right]$

is always less than zero because T_2 is greater than T_1 . This may not be immediately obvious, so let's analyze the characteristics of this expression a little more closely. Say that we fix $T_1 = 1$ and because T_2 must be greater than T_1 , we let it be any value from 1 + to ∞ . Note that when $T_2 = 1$, this is a degenerate case where the periods are equal—you'll find out later why this is some-

thing we never allow. If we plot the expression $\frac{\lceil T_2/T_1 \rceil}{T_2}$, you see that it is a periodic function that oscillates between 1 and 2 as we increase T_2 , equaling 1 anytime T_2 is a multiple of T_1 . By comparison, the term $(1/T_1)$ is constant and equal to 1 in this specific example (we could set T_1 to any constant value—try this and verify that the condition still holds true). Figure 3.6 shows the periodic relationship between T_2 and T_1 that guarantees that U monotonically decreases with increasing C_1 when $(T_2 > T_1)$ for Case 1.

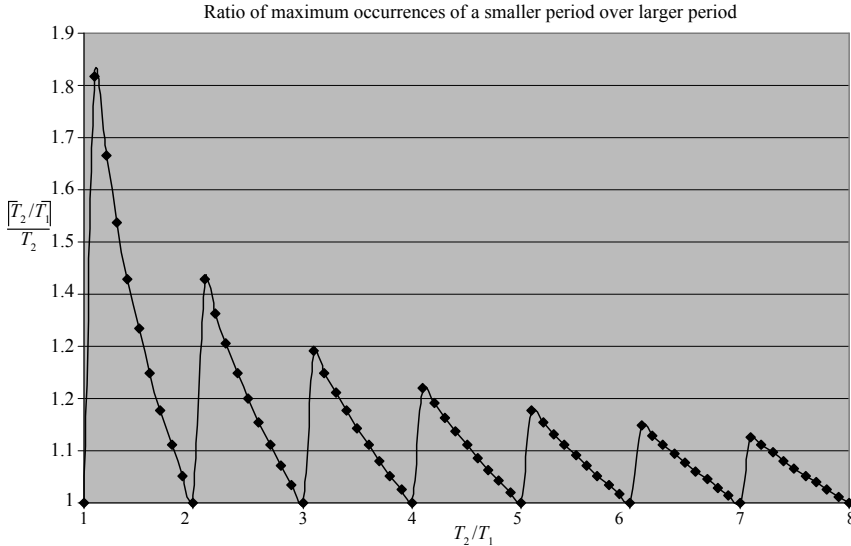


FIGURE 3.6 Case 1 Relationship of T_2 and T_1

So, now that you understand the scenario where C_1 just fits into the critical time zone, let's look at Case 2.

In Case 2, the last release of S_1 does not fit into T_2 —that is, C_1 spills over T_2 boundary on last release, as shown in Figure 3.6. This spillover condition of the last release of S_1 during T_2 is expressed simply as Equation 3.6:

$$C_1 \geq T_2 - T_1 \lfloor T_2 / T_1 \rfloor \quad (3.6)$$

Equation 3.6 is the same as Equation 3.1, but with the inequality flipped—when S_1 's C_1 is just a little too large to fit in the critical time zone shown in Figure 3.7.

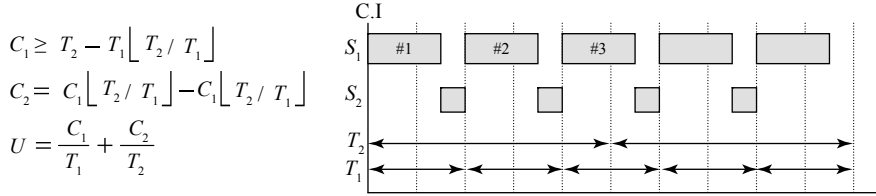


FIGURE 3.7 Case 2 Overrun of Critical Time Zone by S_1

Even though S_1 overruns the critical time zone, time remains for S_2 , and we could find a value of C_2 for S_2 that still allows it to meet its deadline of T_2 . To compute this smaller C_2 , we first note that S_1 release #1 plus #2 in Figure 3.7 along with some fraction of #3 leave some amount of time left over for S_2 . However, if we simply look at the first two occurrences of T_1 during T_2 , leaving out the third release of S_1 during T_2 , then we see that this time is the sum of all full occurrences of T_1 during T_2 , which can be expressed as $T_1 \lfloor T_2 / T_1 \rfloor$. Furthermore, the amount of time that S_1 takes during this $T_1 \lfloor T_2 / T_1 \rfloor$ duration is exactly $C_1 \lfloor T_2 / T_1 \rfloor$. From these observations we derive Equation 3.7:

$$C_2 = T_1 \lfloor T_2 / T_1 \rfloor - C_1 \lfloor T_2 / T_1 \rfloor \quad (3.7)$$

Substituting Equation 3.7 into the utility Equation 3.3 again as before, we get

Equation 3.8:

$$U = \frac{C_1}{T_1} + \frac{\lfloor T_1 \lfloor T_2 / T_1 \rfloor \rfloor - C_1 \lfloor T_2 / T_1 \rfloor}{T_2} \quad (3.8)$$

Now simplifying by the following algebraic steps:

$$U = (T_1 / T_2) \lfloor T_2 / T_1 \rfloor + \frac{C_1}{T_1} + \frac{\lfloor -C_1 \lfloor T_2 / T_1 \rfloor \rfloor}{T_2} \quad (\text{separating terms})$$

$$U = (T_1 / T_2) \lfloor T_2 / T_1 \rfloor + C_1 \lfloor (1 / T_1) - (1 / T_2) \rfloor \lfloor T_2 / T_1 \rfloor \quad (\text{pulling out com-})$$

mon C_1 term)

This gives us Equation 3.9:

$$U = (T_1 / T_2) \lfloor T_2 / T_1 \rfloor + C_1 \left[(1 / T_1) - (1 / T_2) \lfloor T_2 / T_1 \rfloor \right] \quad (3.9)$$

What is interesting about Equation 3.9 is that U *monotonically increases with increasing C_1* when $(T_2 > T_1)$. Recall again that T_2 must be greater than T_1 given the RM priority policy assumed here. The term $(1 / T_2) \lfloor T_2 / T_1 \rfloor$ is always smaller than $(1 / T_2)$ because T_2 is greater than T_1 . As before, this may not be immediately obvious, so let's analyze the characteristics of this expression a little closer—once again, we fix $T_1 = 1$ and because T_2 must be greater than T_1 , we let it be any value from $1+$ to ∞ . Now if we plot this again, you see that $(1 / T_2) \lfloor T_2 / T_1 \rfloor$ is less than 1 in all cases and therefore also less than $(1 / T_1)$, as can be seen in Figure 3.8.

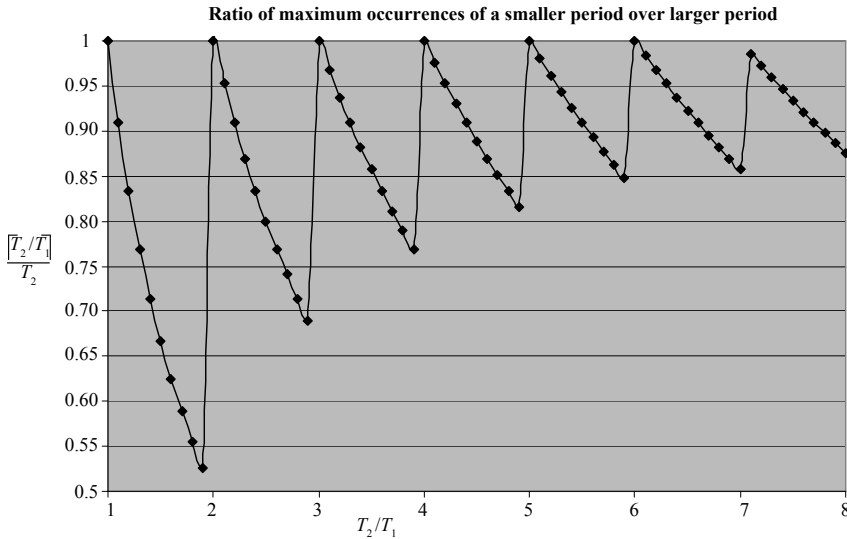


FIGURE 3.8 Case 2 Relationship of T_2 and T_1

So, now we also understand the scenario where C_1 just overruns the critical time zone. The key concept is that in Case 1, we have the maximum number of occurrences of S_1 during T_2 and in Case 2 we have the minimum.

If we now examine the utility functions for both Case 1 and Case 2:

$$U = 1 + C_1 \left[(1 / T_1) - \frac{\lceil T_2 / T_1 \rceil}{T_2} \right] \quad (3.10)$$

$$U = (T_1 / T_2) \lfloor T_2 / T_1 \rfloor + C_1 \left[(1 / T_1) - (1 / T_2) \lfloor T_2 / T_1 \rfloor \right] \quad (3.11)$$

Let's plot the two utility functions on the same graph, setting $T_1=1$, $T_2 = 1+ \text{to } \infty$, and $C_1 = 0 \text{ to } T_1$.

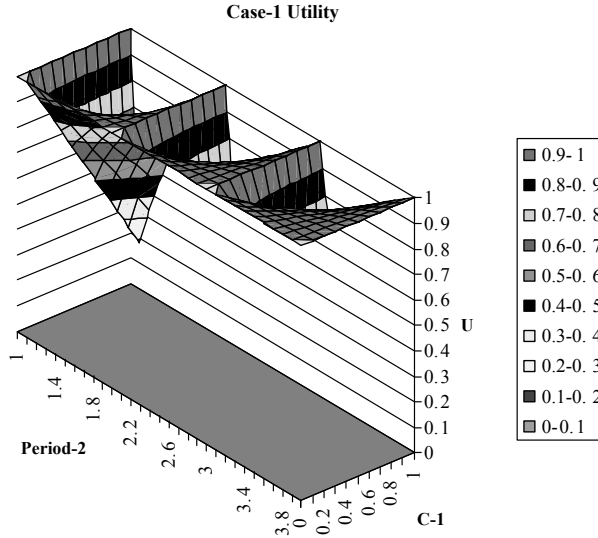


FIGURE 3.9 Case 1 Utility with Varying T_2 and C_1

When $C_1 = 0$, this is not particularly interesting because this means that S_2 is the only service that requires CPU resource; likewise, when $C_1 = T_2$, then this is also not so interesting since it means that S_1 uses all of the CPU resource and never allows S_2 to run. Looking at the utility plot for Equation 3.5 in Figure 3.9 and Equation 3.9 in Figure 3.10, we can clearly see the periodicity of utility where maximum utility is achieved when T_1 and T_2 are harmonic.

What we really want to know is where the utility is equal for both cases so that we can determine utility independent of whether C_1 exceeds or is less than the critical time zone. This is most easily determined by subtracting Figure 3.9 and Figure 3.10 data to find where the two function differences are zero. Figure 3.11 shows that the two function differences are zero on a diagonal when T_2 is varied from 1 times to 2 times T_1 and C_1 is varied from zero to T_1 .

Finally, if we then plot the diagonal of either utility curve (from Equation 3.5 or Equation 3.9) as shown in Figure 3.12, we see identical curves that clearly have a minimum near 83% utility.

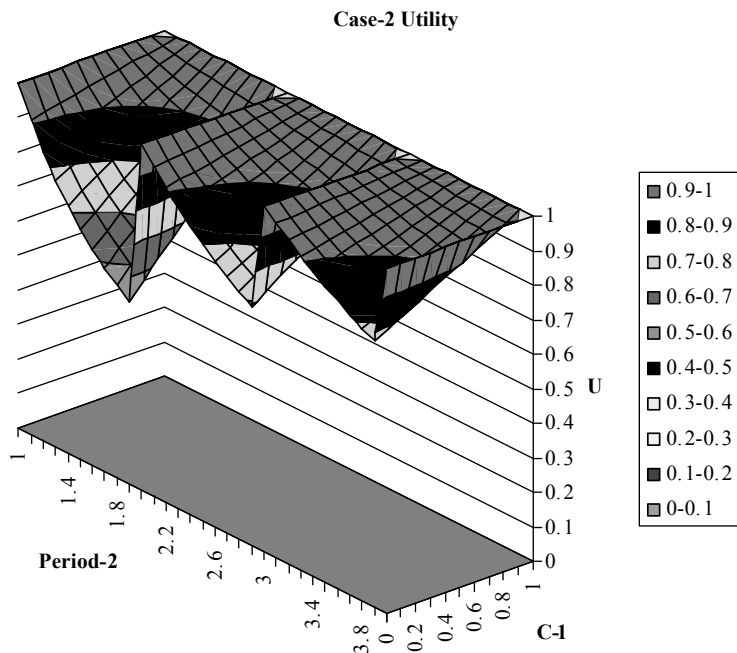


FIGURE 3.10 Case 2 Utility with Varying T_2 and C_1

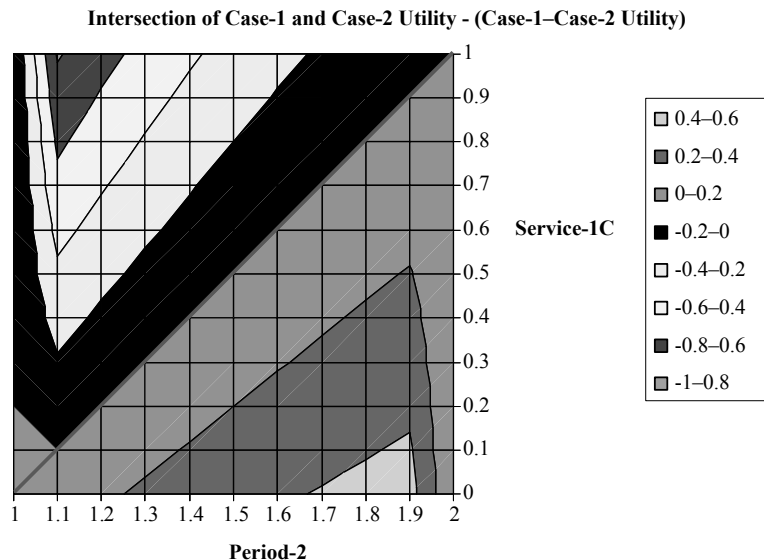


FIGURE 3.11 Intersection of Case 1 and Case 2 Utility Curves

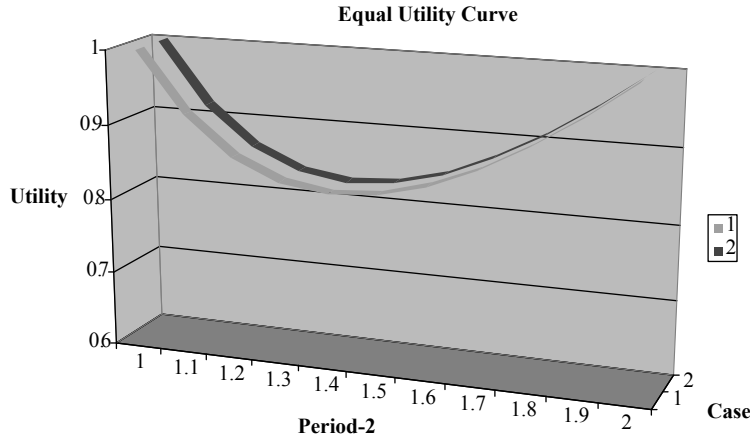


FIGURE 3.12 Two-Service Utility Minimum for Both Cases

Recall that Liu and Layland claim the least upper bound for safe utility given any arbitrary set of services (any relation between periods and any relation between critical time zones) is defined as: $U = \sum_{i=1}^m (C_i / T_i) \leq m(2^{1/m} - 1)$.

For two services we see that $m(2^{1/m} - 1) = 0.83$!

We have now empirically determined that there is a minimum safe bound on utility for any given set of services, but in so doing, we can also clearly see that this bound can be exceeded safely for specific T_1 , T_2 , and C_1 relations.

For completeness, let's now finish the two-service RM least upper bound proof mathematically. We'll argue that the two cases are valid only when they intersect, and given the two sets of equations this can occur only when C_1 is equal for both cases:

$$\begin{aligned} C_1 &= T_2 - T_1 \lfloor T_2 / T_1 \rfloor \\ C_2 &= T_2 - C_1 \lceil T_2 / T_1 \rceil \\ U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} \end{aligned}$$

Now, plug C_1 and C_2 simultaneously into the utility equation to get Equation 3.12:

$$\begin{aligned}
U &= \frac{T_2 - T_1 \lfloor T_2 / T_1 \rfloor}{T_1} + \frac{T_2 - C_1 \lceil T_2 / T_1 \rceil}{T_2} \\
U &= \frac{T_2 - T_1 \lfloor T_2 / T_1 \rfloor}{T_1} + \frac{T_2 - (T_2 - T_1 \lfloor T_2 / T_1 \rfloor) \lceil T_2 / T_1 \rceil}{T_2} \\
U &= \frac{T_2 - T_1 \lfloor T_2 / T_1 \rfloor}{T_1} + \frac{T_2 - T_2 \lceil T_2 / T_1 \rceil + T_1 \lfloor T_2 / T_1 \rfloor \lceil T_2 / T_1 \rceil}{T_2} \\
U &= (T_2 / T_1) - \lfloor T_2 / T_1 \rfloor + 1 - \lceil T_2 / T_1 \rceil + (T_1 / T_2) \lfloor T_2 / T_1 \rfloor \lceil T_2 / T_1 \rceil \\
U &= 1 - \lceil T_2 / T_1 \rceil + (T_1 / T_2) \lfloor T_2 / T_1 \rfloor \lceil T_2 / T_1 \rceil + (T_2 / T_1) - \lfloor T_2 / T_1 \rfloor \\
U &= 1 - (T_1 / T_2) \left((T_2 / T_1) \lceil T_2 / T_1 \rceil - \lfloor T_2 / T_1 \rfloor \lceil T_2 / T_1 \rceil - (T_2 / T_1)^2 + (T_2 / T_1) \lfloor T_2 / T_1 \rfloor \right) \\
U &= 1 - (T_1 / T_2) [\lceil T_2 / T_1 \rceil - (T_2 / T_1)] [(T_2 / T_1) - \lfloor T_2 / T_1 \rfloor] \quad (3.12)
\end{aligned}$$

Now, let whole integer number of interferences of S_1 to S_2 over T_2 be $I = \lfloor T_2 / T_1 \rfloor$ and the fractional interference be $f = (T_2 / T_1) - \lfloor T_2 / T_1 \rfloor$.

From this, we can derive a simple expression for utility:

$$U = 1 - \left(\frac{f(1-f)}{(T_2 / T_1)} \right) \quad (3.13)$$

The derivation for Equation 3.11 is based upon substitution of I and f into Equation 3.12 as follows:

$$\begin{aligned}
U &= 1 - (T_1 / T_2) [\lceil T_2 / T_1 \rceil - (T_2 / T_1)] [(T_2 / T_1) - \lfloor T_2 / T_1 \rfloor] \\
U &= 1 - (T_1 / T_2) [1 + \lfloor T_2 / T_1 \rfloor - (T_2 / T_1)] [(T_2 / T_1) - \lfloor T_2 / T_1 \rfloor] \quad \text{based on} \\
&\quad \text{ceiling(N.d)} = 1 + \text{floor(N.d)}
\end{aligned}$$

$$U = 1 - (T_1 / T_2) [1 - ((T_2 / T_1) - \lfloor T_2 / T_1 \rfloor)] [(T_2 / T_1) - \lfloor T_2 / T_1 \rfloor]$$

$$U = 1 - (T_1 / T_2) (1 - f)(f)$$

$$U = 1 - \left(\frac{f(1-f)}{(T_2 / T_1)} \right)$$

By adding and subtracting the same denominator term to Equation 3.13, we can get:

$$U = 1 - \left(\frac{f(1-f)}{\lfloor T_2 / T_1 \rfloor + (T_2 / T_1) - \lfloor T_2 / T_1 \rfloor} \right)$$

$$U = 1 - \left(\frac{f(1-f)}{(I+f)} \right)$$

The smallest I possible is 1, and the LUB for U occurs when I is minimized, so we substitute 1 for I to get:

$$U = 1 - \left(\frac{(f-f^2)}{(1+f)} \right)$$

Now taking the derivative of U w.r.t. f , and solving for the extreme, we get:

$$\partial U / \partial f = \frac{(1+f)(1-2f) - (f-f^2)(1)}{(1+f)^2} = 0$$

Solving for f , we get:

$$f = (2^{1/2} - 1)$$

And, plugging f back into U , we get:

$$U = 2(2^{1/2} - 1)$$

The RM LUB of $m(2^{1/m} - 1)$ is $2(2^{1/2} - 1)$ for $m=2$, which is true for the two-service case—QED.

Having derived the RM LUB by inspection and by mathematical manipulation, what did we actually learn from this? Most importantly, the pessimism of the RM LUB that leads to low utility for real-time safety is based upon a bound that works for all possible combinations of T_1 and T_2 . Specifically, the RM LUB is pessimistic for cases where T_1 and T_2 are harmonic—in these cases it is possible to safely achieve 100% utility! In many cases, as shown by demonstration in Figure 3.5 and the Lehoczky, Sha, Ding theorem, we can also safely utilize a CPU at levels below 100% but

above the RM LUB. The RM LUB still has value since it is a simple and quick feasibility check that is sufficient. When going through the derivation, it should now be evident that in many cases it is not possible to safely utilize 100% of the CPU resource with fixed-priority preemptive scheduling and the RM policy; in the next section the Lehoczky, Sha, Ding theorem is presented and provides a necessary and sufficient feasibility test for RM policy.

3.5 Necessary and Sufficient Feasibility

Two algorithms for determination of N&S feasibility testing with RM policy are easily employed:

- Scheduling Point Test
- Completion Time Test

To always achieve 100% utility for any given service set, you must use a more complicated policy with dynamic priorities. You can achieve 100% utility for fixed-priority preemptive services, but only if their relative periods are harmonic. Note also that RM theory does not account for IO latency. The implicit assumption is that IO latency is either insignificant or known deterministic values that can be considered separately from execution and interference time. In the upcoming “Deadline-Monotonic Policy” section, you’ll see that it’s straightforward to slightly modify RM policy and feasibility tests to account for a deadline shorter than the release period, therefore allowing for additional output latency. Input latency can be similarly dealt with, if it’s a constant latency, by considering the effective release of the service to occur when the associated interrupt is asserted rather than the real-world event. Because the actual time from effective release to effective release is no different than from event to event, the effective period of the service is unchanged due to input latency. As long as the additional latency shown in Figure 3.13 is acceptable and does not destabilize the service, it can be ignored. However, if the input latency varies, this causes period jitter. In cases where period jitter exists, you simply assume the worst-case frequency or shortest period possible.

3.5.1 Scheduling Point Test

Recall that by the Lehoczky, Sha, Ding theorem, if a set of services can be shown to meet all deadlines from the critical instant up to the longest deadline of all tasks in the set, then the set is feasible. Recall the critical

- $\left\lceil \frac{(l)T_k}{T_j} \right\rceil$ represents the number of times S_j executes within l period of S_k .

$C_j \left\lceil \frac{(l)T_k}{T_j} \right\rceil$ is the time required by S_j to execute within l periods of S_k —

if the sum of these times for the set of tasks is smaller than l period of S_k , then the service set is feasible.



The following is a C code algorithm included with test code on the DVD for the Scheduling Point Test, assuming arrays are sorted according to the RM policy where period[0] is the highest priority and shortest period:

```
#include <math.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define U32_T unsigned int

int scheduling_point_feasibility(U32_T numServices,
                                U32_T period[], U32_T wcet[],
                                U32_T deadline[])
{
    int rc = TRUE, i, j, k, l, status, temp;

    for (i=0; i < numServices; i++) // iterate from highest to
                                    lowest priority
    {
        status=0;

        for (k=0; k<=i; k++)
        {
            for (l=1; l <= (floor((double)period[i]/(double)
                                   period[k])); l++)
            {
                temp=0;

                for (j=0; j<=i; j++) temp += wcet[j] *
                    ceil((double)l*(double)period[k]/(double)
                        period[j]);
            }
        }
    }
}
```

```

        if (temp <= (l*period[k]))
        {
            status=1;
            break;
        }
    }
    if (status) break;
}
if (!status) rc=FALSE;
}
return rc;
}

```

3.5.2 Completion Time Test

The Completion Time Test is presented as an alternative to the Scheduling Point Test [Briand99]:

$$a_n(t) = \sum_{j=1}^n \left\lceil \frac{t}{T_j} \right\rceil C_j$$

- $\left\lceil \frac{t}{T_j} \right\rceil$ is the number of executions of S_j at time t .
- $\left\lceil \frac{t}{T_j} \right\rceil C_j$ is the demand of S_j in time at t .
- $a_n(t)$ is the total cumulative demand from the n tasks up to time t .

Passing this test requires proving that $a_n(t)$ is less than or equal to the deadline for S_n , which proves that S_n is feasible. Proving this same property for all S from S_1 to S_n proves that the service set is feasible.

The following is a C code algorithm for the Completion Time Test, assuming arrays are sorted according to the RM policy where `period[0]` is the highest priority and shortest period:

```

#include <math.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define U32_T unsigned int

```



```

int completion_time_feasibility(U32_T numServices,
                                U32_T period[], U32_T wcet[],
                                U32_T deadline[])
{
    int i, j;
    U32_T an, anext;

    // assume feasible until we find otherwise
    int set_feasible=TRUE;

    //printf("numServices=%d\n", numServices);

    for (i=0; i < numServices; i++)
    {
        an=0; anext=0;

        for (j=0; j <= i; j++)
        {
            an+=wcet[j];
        }

        //printf("i=%d, an=%d\n", i, an);

        while(1)
        {
            anext=wcet[i];

            for (j=0; j < i; j++)
                anext += ceil(((double)an)/((double)period[j]))
                        *wcet[j];

            if (anext == an)
                break;
            else
                an=anext;

            //printf("an=%d, anext=%d\n", an, anext);
        }

        //printf("an=%d, deadline[%d]=%d\n", an, i, deadline[i]);
    }
}

```

```

        if (an > deadline[i])
        {
            set_feasible=FALSE;
        }
    }

    return set_feasible;
}

```

3.6 Deadline-Monotonic Policy

Deadline-monotonic (DM) policy is very similar to RM except that highest priority is assigned to the service with the shortest deadline. The DM policy is a fixed-priority policy (be sure not to confuse this with EDF (Earliest Deadline First), where priority is assigned dynamically with highest priority assigned to the active service with the nearest or earliest deadline at any given point in time). The DM policy eliminates the original RM assumption that service period must equal service deadline and allows RM theory to be applied for scenarios even when deadline is less than the period. This is useful for dealing with significant output latency. The DM policy can be shown to be optimal fixed-priority assignment policy like RM policy because D_i and T_i differ only by a constant value, and $D_i \leq T_i$. The DM policy feasibility tests are most easily implemented as iterative tests like Scheduling Point and the Completion Time Test for RM policy. This policy and associated feasibility tests were first introduced by Alan Burns and Neil Audsley [Audsley93]. The sufficient feasibility test they first introduced is simple and intuitive:

$$\forall i: 1 \leq i \leq n: \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1.0 \quad (3.14)$$

C_i is the execution time for service i , and I_i is the interference time service i must tolerate over its deadline D_i time period since the time of request for service.

Equation 3.14 states that for all services from 1 to n , if the deadline interval is long enough to contain the service execution time interval plus all interfering execution time intervals, then the service is feasible. If all services are feasible, then the system is feasible (real-time-safe).

Interference to Service S_i is due to preemption by all higher-priority services S_1 to S_{i-1} , and the total interference time is the number of releases

of S_j over the deadline interval D_i . The number of S_i interferences is then multiplied by execution time C_j and summed for all S_j . Note that S_j always has higher priority than S_i .

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \quad (3.15)$$

Where $\left\lceil \frac{D_i}{T_j} \right\rceil$ is the worst-case number of releases of S_j over the deadline interval for S_i . Because the interference is the worst-case number of releases, interference is over-accounted for—the last interference may be only partial. So, there will be $\left\lceil \frac{D_i}{T_j} \right\rceil$ full interferences and some partial interference from the last additional interference instance. So, we can better account for the partial interference with

$$I_i = \sum_{j=1}^{i-1} \left[\left\lceil \frac{D_i - D_j}{T_j} \right\rceil + 1 \right] C_j + \left[\left\lceil \frac{D_i}{T_j} \right\rceil - \left[\left\lceil \frac{D_i - D_j}{T_j} \right\rceil + 1 \right] \right] \times \text{Min} \left[C_j, D_i - \left\lceil \frac{D_i}{T_j} \right\rceil T_j \right] \quad (3.16)$$

Where $\left[\left\lceil \frac{D_i - D_j}{T_j} \right\rceil + 1 \right] C_j$ is the full-interference time,

$$\left[\left\lceil \frac{D_i}{T_j} \right\rceil - \left[\left\lceil \frac{D_i - D_j}{T_j} \right\rceil + 1 \right] \right] = 0 \text{ if no partial interference and } 1 \text{ if there is, and}$$

$\text{Min} \left[C_j, D_i - \left\lceil \frac{D_i}{T_j} \right\rceil T_j \right]$ is the partial interference time if it exists.

However, even Equation 3.16 does not exactly account for partial interference. So, both Equations 3.15 and 3.16 when used with Equation 3.14 are only sufficient feasibility tests.

A slightly different approach to dealing with $T_i \neq D_i$ is to simply assume that S_i has a shorter period than it really does until $T_i = D_i$. This approach is a variation of period transform that would affect overall utilization, but allows the RM policy and feasibility approaches to be applied without modification, including the N&S Completion Time Test and/or Scheduling Point

Test. In general, period transform is used to increase the frequency of a periodic service to raise its RM priority, often by dividing the implementation of the service into multiple parts. Because output latencies typically are not a huge portion of response time, period transform is a practical way to force real-world problems into the RM framework—deriving a DM N&S feasibility test would be another option. However, the N&S DM feasibility test is complex, so unless there is a huge output latency, period transform is the best approach due to the large body of well-understood RM theory.

An alternative to RM or DM is the dynamic-priority policies derived from the deadline-driven dynamic-priority approach, first presented by Liu and Layland. In the next section, we'll explore the advantages and disadvantages of dynamic priorities compared to static. Today, for hard real-time systems that must provide deterministic responses to service requests, fixed-priority RM policy remains the most widely used and universally accepted theory.

3.7 Dynamic-Priority Policies

Priority preemptive dynamic-priority systems can be thought of as a more complex class of priority preemptive where priorities are adjusted by the scheduler every time a new service is released and ready to run. The concept was first formally introduced by Liu and Layland [Liu73] with their description of deadline-driven scheduling policy. The policy Liu and Layland specified in their paper later became known as an EDF (Earliest Deadline First) dynamic-priority policy. The policy is called EDF because the scheduler gives highest priority to the service that has the soonest deadline whenever a dispatch decision is made. This also means that anytime an additional thread is placed on the ready queue, the EDF scheduler must reevaluate all dispatch priorities because the newly added thread may not have a deadline later than all the existing threads on the ready queue. Basically, the EDF scheduler must be able to insert the new thread into the queue based upon time to its deadline relative to the time to deadline for all other threads—the insertion has a complexity that is of the order $n - O(n)$, where n is the number of threads on the queue. By comparison a fixed-priority policy scheduler can be implemented with complexity that is $O(1)$, or constant time using priority queues. Liu and Layland proved that the potential utility for EDF is full and that deadlines can be guaranteed with EDF. This is an incredible result when compared to fixed-priority RM policy.

Essentially no margin is required for real-time safety. Is this really true? If so, then why wouldn't EDF be the only policy ever used for real-time systems? Figure 3.14 shows a scenario where the fixed-priority RM policy fails, and EDF succeeds. Furthermore, it shows that a related dynamic priority policy, LLF (Least Laxity First), also succeeds where RM fails.

Like EDF, LLF is a dynamic-priority policy where services on the ready queue are assigned higher priority if their laxity is the least. *Laxity* is the time difference between their deadline and remaining computation time. This requires the scheduler to know all outstanding service request times, their deadlines, the current time, and remaining computation time for all services, and to reassign priorities to all services on every preemption. Estimating remaining computation time for each service can be difficult and typically requires a worst-case approximation. Like EDF, LLF can also schedule 100% of the CPU for schedules that can't be scheduled by the static RM policy.

Example	T1	2	C1	1	U1	0.5	LCM =	70		
	T2	5	C2	1	U2	0.2				
	T3	7	C3	2	U3	0.285714	Utot =	0.985714		
RM Schedule										
S1							????????			
S2						????????				
S3								LATE		
EDF Schedule										
S1										
S2										
S3										
TTD										
S1	2	X	2	X	2	X	2	X	2	X
S2	5	4	X	X	X	5	4	3	X	X
S3	7	6	5	4	3	5	X	7	6	5
LLF Schedule										
S1										
S2										
S3										
Laxity										
S1	1	X	1	X	1	X	1	X	1	X
S2	4	3	X	X	X	4	3	2	X	X
S3	5	4	3	2	2	2	X	2	4	3

FIGURE 3.14 RM Policy Overload Scenario

Intuitively, it's hard to believe that the use of any resource can be safe with no margin at all. Even the slightest miscalculation on resources required can cause an overload (overuse of resources). This is a likely scenario given that determining the actual execution time that all services will require is not easy unless very pessimistic worst-case times are assumed. So, it becomes interesting to consider what happens to threads or services in an overload scenario for a given policy. For EDF, overload

leads to nondeterministic failure—that is, it's very hard to predict exactly which and how many services will miss their deadlines in an overload. It depends upon the state of the relative priorities during the overload, which in turn depends upon the order of time to deadline times for all services ready to run.

By comparison, for fixed-priority policies such as RM, in an overload, all services of lower priority than the service that is overrunning may miss their deadline, yet all services of higher priority are guaranteed not to be affected, as shown in Figure 3.15.

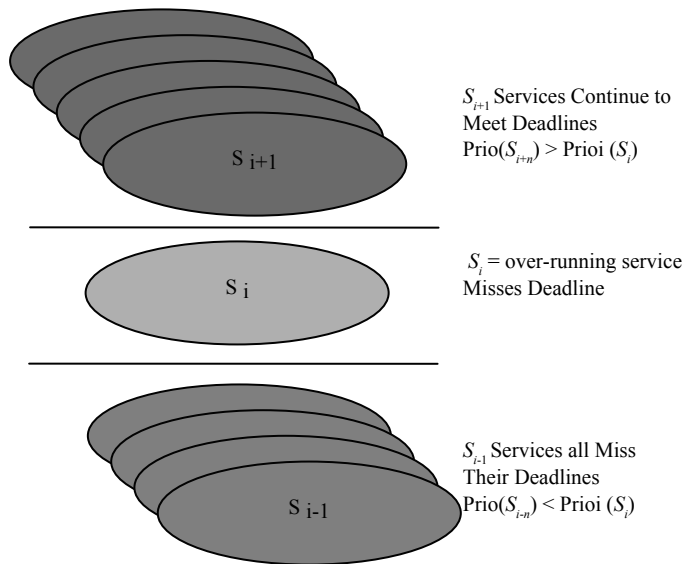


FIGURE 3.15 RM Policy Overload Scenario

For EDF an overrun by any service may cause all other services on the ready queue to miss their deadlines; a new service added to the queue, therefore adjusting priorities for all, will not preempt the overrunning service. The overrunning service has a time to deadline that is negative because it has passed, so it continues to be the highest priority service and continues to cause others to wait and potentially miss deadlines. In an overrun scenario, common policy is to terminate the release of a service that has overrun. This causes a service dropout. However, simply detecting overrun and terminating the overrunning service take some CPU resource, which without any margin means that some other service will miss its deadline—with overrun control EDF becomes much more well behaved in an overload

scenario—the services with the soonest deadlines will then clearly be the ones to lose out. However, determining which services this will be in advance—based upon the dynamics of releases relative to each other—is still difficult. Figure 3.16 graphically depicts the potentially cascading EDF overload failure scenario—all services queued while the overrunning service executes potentially miss their deadlines, and the next service is likely to overrun as well, causing a cascading failure. Probably the best option for an EDF overload is to dump all services in the queue—this at least would be more deterministic.

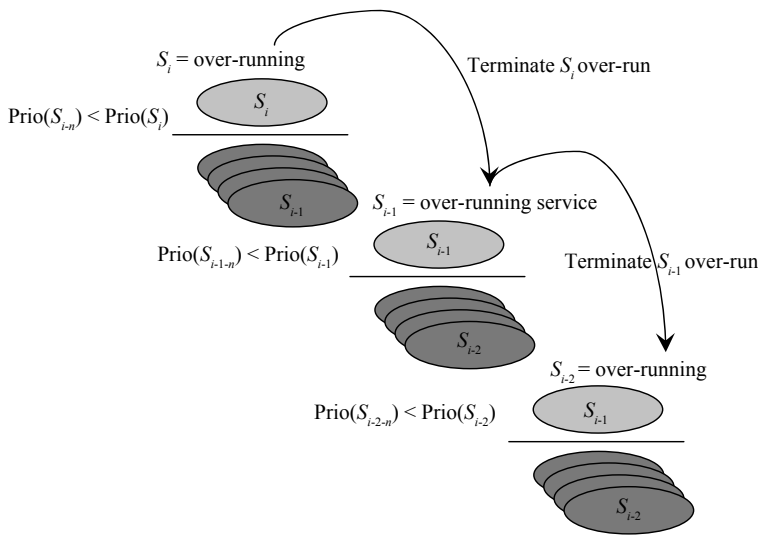


FIGURE 3.16 EDF Policy Cascading Failure Overload Scenario

Variations of EDF exist where a different policy is encoded into the deadline-driven, dynamic-priority framework (defined originally by Liu and Layland). One of the more interesting variations is LLF. In LLF, highest priority is assigned to the service that has the least difference between its remaining execution time and its upcoming deadline. Laxity is the time difference between their deadline and remaining computation time for a service. Determining the least laxity service requires the scheduler to know all outstanding service request times, their deadlines, the current time, and remaining computation time for all services. After all this is known, the scheduler must then reassign priorities to all services on every event that adds another service to the ready queue. Estimating remaining computation time for each service can be difficult and typically requires a worst-case approximation. The LLF policy encodes the concept of imminence, which

intuitively makes sense—every student knows that they should work on the homework where they have the most to do and that is due soonest first—unless of course that particular homework is not worth much credit.

In some sense, all priority encoding policies, dynamic or static, miss the point—what we really want to do is encode which service is most important and make sure that it gets the resources it needs first. We want an intelligent scheduler, like a student who takes into consideration laxity, impact of missing a deadline for a given assignment, cost of dropping one or more, and then intelligently determines how to spend resources for maximum benefit. This concept is an open research area in soft real-time systems and has been investigated by a number of researchers [Brandt99]. In fact, since the landmark Liu and Layland formalization of RM and deadline-driven scheduling, most of the processor resource research has been oriented to one of four things:

- Generalization and reducing constraints for RM application
- Solving problems related to RM application for real systems
- Devising alternative policies for deadline-driven scheduling
- Devising new soft real-time policies to reduce margin required in RM policy

In Chapter 7, “Soft Real-Time Services,” we will explore some of the methods that have been proposed to adapt RM and deadline-driven policies to situations where an occasional service dropout or overrun is acceptable. Soft real-time methods for handling missed deadlines will be more closely examined, along with more in-depth coverage of EDF and LLF scheduling.

Summary

Fixed-priority preemptive scheduling with an RM priority assignment policy (shortest period has highest priority) is most often used and advised for hard real-time systems. Hard real-time systems by definition must have deadline guarantees because the consequences of a missed deadline are total system failure, significant loss of assets, and possible loss of human life. All service sets run as threads of execution should be tested with a sufficient or better yet N&S feasibility test before being fielded for hard real-time

operation. Passing an N&S feasibility test guarantees a service set will meet its deadlines as long as C_i , T_i , and D_i were properly specified and are deterministic or worst-case. For quick analysis, a sufficient test such as the RM LUB may be useful. The potential disharmony in period is the reason that the RM LUB is less than full utility for two or more services. Services that have harmonic periods can be safely scheduled with utility exceeding the RM LUB despite failing to pass this test and will pass an N&S test. In general, dynamic-priority policies, such as EDF and LLF, are not considered safe for hard real-time systems due to their difficult-to-predict deadline-overrun characteristics. Dynamic-priority policies do work well for soft real-time applications, such as game engines, video, audio, and multimedia applications, where an occasional service dropout is acceptable.

Exercises

1. Code the sufficient RM scheduling feasibility test [Liu73, p. 9, Theorem 5] for following ANSI C function prototype:

```
int RM_sufficient(
    int Ntasks,
    int *tid,
    unsigned long int *T,
    unsigned long int *C,
    unsigned long int *D);
```

Where Ntasks is the number of tasks in the task set, tid is an array of unique task Ids, T is an array of the release periods, C is an array of the computation times, and D is an array of the deadlines (T must equal D for each tid in RM). Finally, the function should simply return 1 if the system can be scheduled, and 0 if it can't.

2. Code the Completion Time Test (Chapter 2 of RTECS text, Processing section) for the following ANSI C function prototype:

```
int Sched_completion(
    int Ntasks,
    int *tid,
    unsigned long int *T,
    unsigned long int *C,
    unsigned long int *D);
```

Where parameters are defined again as in #1. Assume that T must equal D in all cases.

3. Describe in your own words what the difference is between a “sufficient” and a “necessary and sufficient” scheduling feasibility test.
4. Why is the sufficient RM least upper bound so pessimistic?
5. If EDF can be shown to meet deadlines and potentially has 100% CPU resource utilization, then why is it not typically the hard real-time policy of choice? That is, what are drawbacks to using EDF compared to RM/DM? In an overload situation, how will EDF fail?
6. Code a function to compute the Fibonacci sequence to any number of terms. The sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, . . . The Fibonacci sequence or Fibonacci numbers begin with 0 and 1. The next term is then the sum of the two previous terms. (Do not be concerned if your Fibonacci number overflows an unsigned 32-bit integer—just let it overflow).
7. Now, determine how many terms in the sequence correspond to 10 milliseconds of computation on a lab target PC (note that the answer may vary depending upon the specific target used). The easiest way to do this is to use WindView to measure the CPU time taken by a task calling your function with a large value—see how long that takes and then scale up or down the number as needed to achieve 10 milliseconds of computation. Repeat this to determine how many terms are required for 20 milliseconds of computation using WindView.
8. Given a task set with two tasks calling your Fibonacci sequence, one with N terms for 10 milliseconds of execution and the other for 20 milliseconds, is the system feasible if the 10 millisecond task is released every 20 milliseconds and the 20 millisecond task every 50 milliseconds (i.e., $T_1=20$ msec, $T_2=50$ msec, $C_1=10$ msec, $C_2=20$ msec and all D_i 's = T_i 's)? Base your answer upon the Lehockzy, Sha, and Ding theorem.
9. Run the foregoing system and either show evidence that it works or explain why it won't.

Chapter References

- [Audsley93] N. Audsley, A. Burns, and A. Wellings, “Deadline Monotonic Scheduling Theory and Application,” *Control Engineering Practice*, Vol. 1, 1993, pp. 71–78.
- [Brandt99] Scott Brandt, “*Soft Real-Time Processing with Dynamic Quality of Service Level Resource Management*,” PhD Dissertation, Department of Computer Science, University of Colorado, 1999.
- [Briand99] L. Briand and D. Roy, “*Meeting Deadlines in Hard Real-Time Systems*,” IEEE Computer Society (1999): pp. 28–31.
- [Buttazzo02] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems—Predictable Scheduling Algorithms and Applications*, Kluwer Academic, 2002.
- [Laplante12] Phillip A. Laplante and Seppo J. Ovaska, *Real-Time Systems Design and Analysis*, 4th ed. IEEE Press, Wiley, 2012.
- [Liu73] C. Liu and J. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of the Association for Computing Machinery*, Vol. 20, No. 1 (January 1973): pp. 46–61.
- [Mall07] Rajib Mall, *Real-Time Systems—Theory and Practice*, Dorling Kindersley (India), 2007.

RESOURCES

In this chapter

- Introduction
- Worst-Case Execution Time
- Intermediate IO
- Execution Efficiency
- IO Architecture

4.1 Introduction

The input and output to a service shown previously in Figure 3.13 require IO to/from a device such as a sensor or actuator (encoder or decoder). This IO is part of the response time, and as shown in Chapter 3, it simply adds to response latency, but does not affect the service execution or interference time during the response. Most services, unless they are trivial, involve some intermediate IO after the initial sensor input and before the final posting of output data to a write buffer. This intermediate IO is most often MMR (Memory-Mapped Register) or memory device IO. If this intermediate IO has single core cycle latency, zero wait-state, then it has no additional impact on the service response time. However, if the intermediate IO stalls the CPU core, then this increases the response time while the CPU processing pipeline is stalled. Rather than considering this intermediate IO as device IO, it is more easily modeled as execution efficiency. Device IO latency is hundreds, thousands, and even millions of core cycles.

By comparison, intermediate IO latency is typically tens or hundreds of core cycles—if more latency than this is possible, then the core hardware design should be reworked.

4.2 Worst-Case Execution Time

Ideally the execution time for a service release would be deterministic. For simple microprocessor architectures, this may be true. The Intel 8088 and the Motorola 68000, for example, have no CPU pipeline and no cache, and given memory that has no wait-states, you can take a block of code and count the instructions from start to finish. Furthermore, for these architectures, the number of CPU cycles required to execute each instruction is known—some instructions may take more cycles than others, but all are known numbers. So, the total number of CPU clock cycles required to execute a block of code can be calculated. To compute deterministic execution time for a service, the following system characteristics are necessary:

- Exact number of instructions executed from service release input up to response output.
- The exact number of CPU clock cycles for each instruction is known.
- The number of CPU clock cycles for a given instruction is constant.

Let's assume that the second and third characteristics are true, providing deterministic hardware. The same set of instructions always requires the same number of CPU clocks to execute. This alone does not guarantee deterministic execution time because an algorithm may be data-driven. The number of loop iterations or the depth of recursion of the algorithm may be a function of the inputs to the algorithm. For data-driven algorithms, the path length, or total instruction count, is a function of the input. Most algorithms are data-driven. Any block of code that contains decision constructs, such as “if” statements or “case” statements, will execute a different path based upon the outcome of the “if” expression or the “case” statement. For simple data-driven algorithms and code blocks, you can simply count instructions in all paths and compare to determine the longest path. This appears simple enough for a small block of code and simple algorithm, but what about an application that performs a complex service? For example, assume a service needs to find the root of a function where the function is not simple. In the case of finding a root for a function that is determined

by integrating sensor rates over time, the function is data-driven and not known a priori. Say you want to predict when a rolling satellite will be brought to rest via deceleration using a thruster—that is, when the thrust function (and therefore acceleration) causes the velocity function to reach zero. The thrust function is often known for a particular type of thruster. One method to find the root of any function is to iterate, bisecting an interval to define x and feeding the bisection value into $F(x) = 0$ to test how close the current guess for x is to zero. If the guess is higher, then a lesser or greater subinterval will be selected for the next iteration. If $F(x)$ is a continuous function, then with successive iterations, the interval will become diminishingly small and the bisection of that interval, or x , will come closer and closer to the true value of x where $F(x)$ is zero. How much iteration will this take? The answer depends upon the following requirements:

- Accuracy of x needed
- Complexity of the function $F(x)$
- The initial interval

Finally, some functions may actually have more than one solution as well. Many numerical methods are similar to finding roots by bisection in that they require a total path length that varies with the input. For such algorithms, you need to place an upper bound on the path length to define WCET (Worst-Case Execution Time).

Assuming an upper bound on the algorithmic path length and deterministic hardware, then the WCET is safe as an input to an RM (Rate-Monotonic) feasibility test. A service release that requires less than the maximum path length simply enjoys more than necessary resource margin. With the evolution of CPU hardware design, most microprocessors have evolved to maximize throughput and provide better overall efficiency by employing acceleration to the most commonly executed instruction sequences and data references. This is typified by the RISC (Reduced Instruction Set Computer) with instruction pipelining and use of memory caches. As CPU core clock rates increased, memory access latency for comparably scaled capacity has generally not kept pace. So, most RISC pipelined architectures make use of cache, a small zero wait-state (single CPU cycle access latency) memory. Unfortunately, cache is too small to hold many applications. So, set associative memories are used to temporarily hold main memory data—when the cache holds data for an address referenced by a program, this is a

hit, and the single-cycle access to data and/or code allows the CPU pipeline to fetch an instruction or load data into a register in a single cycle. A cache miss, however, stalls the pipeline. Furthermore, IO from MMRs (Memory-Mapped Registers) may require more than a single CPU core cycle and will likewise stall the CPU pipeline if the data is needed for the next instruction. Detecting potential stalls and avoiding them is an art that can increase execution efficiency overall for a CPU—for example, instructions that cause an MMR read can be executed out of order so that the instruction requiring the read data is executed as late as possible, delaying the potential pipeline stall.

The point of pipelining, described in detail in the next section, is to increase overall execution efficiency. However, as is evident from the examples of cache misses and MMR latencies that may cause a data-dependency stall, pipelines will stall, and this is a function of the instruction and data stream. The efficiency is therefore data- and code-driven and not deterministic. So, execution efficiency will vary, even for the same block of code, because cache contents may be a function not only of the current thread of execution but also of the previous threads that executed in the past. In summary, WCET is a function of the longest path length and the efficiency in executing that path. Equation 4.1 describes WCET:

$$\text{WCET} = \left[\text{CPI}_{\text{worst_case}} \times \text{Longest_Path_Instruction_Count} \right] \times \text{Clock_Period} \quad (4.1)$$

The CPI is a figure that describes efficiency in pipelined execution as the number of Clocks Per Instruction on average that are required to execute each instruction in a block of code. In the next two sections, “Intermediate IO” and “Execution Efficiency,” we discuss how best- and worst-case CPI can be determined or at least approximated well. The longest path instruction count must be determined by inspection, formal software engineering proof, or actual instruction count traces in a simulator or with the target CPU architecture. Warning—most CPU core documentation states a CPI that is best-case rather than worst-case.

For full determinism in WCET for hard real-time systems, you must guarantee the following:

- All memory access is to known latency memory, including locked cache or main memory with zero or bounded wait-states.

- Unlocked cache hits are not expected in unlocked cache because the hit rate is not deterministic.
- Overlap of CPU and device IO is not expected nor required to meet deadlines.
- Full accounting for all pipeline hazards in addition to cache misses and device IO read/write wait states.
- Stalls are lumped into CPI and taken as worst-case (e.g., branch-density \times branch penalty).
- Longest path is known, and instruction count for it is known.

For soft real-time systems, you can allow occasional service dropouts or limited overruns and therefore use ACET (Average-Case Execution Time). The ACET can be estimated from the following information:

- Expected L1 and L2 cache hit/miss ratio and cache miss penalty.
- Expected overlap of CPU and device IO required for deadlines.
- All other pipeline hazards are typically secondary and can be ignored like branch misprediction.
- Average length path is known, and the instruction count for it is known.

In summary, you have the following two equations which are Worst-Case Execution Time and Average-Case Execution Time (4.2):

$$\text{WCET} = \text{Memory_Latency} + \text{Device_IO_Latency} + \left[\text{Longest_Path_Inst_Count} \times \text{CPI}_{\text{Effective}} \right]$$

$$\text{ACET} = \left[\text{Expected_Cache_Miss_Rate} \times \text{Miss_Penalty} \right] + \left[\text{NOA} \times \text{IO_Latency} \right] + \left[\text{Expected_Path_Inst_Count} \times \text{CPI}_{\text{Effective}} \right]$$

In these equations in 4.2, the effective CPI accounts for secondary pipeline hazards, such as branch mispredictions. The term NOA (Non-Overlap Allowed) is 1.0 if IO time is not overlapped with processing at all.

4.3 Intermediate IO

In a non-preemptive run-to-completion system with a pipelined CPU, six key related equations describe CPU-IO overlap. Note that the IO described here is device IO that occurs during the service execution, rather than the initial IO, which releases the service in the first place. In some sense, the device IO occurring during service execution can be considered micro-IO and usually consists of MMR access rather than block-oriented DMA (Direct Memory Access) IO. Although this intermediate IO is much lower-latency than the initial block IO latency, it reduces the execution efficiency significantly. Ideally, with careful generation of machine code (compiler optimizations), careful hardware design for pipeline instruction reordering, and careful service design, you can minimize the loss of efficiency due to micro-IO. First, you must understand what it means to overlap IO with CPU.

Consider the following execution and IO overlap *definitions*:

- ICT = Instruction Count Time (Time to execute a block of instructions with no stalls = CPU Cycles \times CPU Clock Period)
- IOT = Bus Interface IO Time (Bus IO Cycles \times Bus Clock Period)
- OR = Overlap Required—percentage of CPU cycles that must be concurrent with IO cycles
- NOA = Non-Overlap Allowable for S_i to meet D_i —percentage of CPU cycles that can be in addition to IO cycle time without missing service deadline
- D_i = Deadline for Service S_i relative to release (interrupt initiating execution)
- CPI = Cycles Per Instruction for a block of instructions

The characteristics of overlapping IO cycles with CPU cycles for a service S_i are summarized as follows by the five possible *overlap conditions* for CPU time and IO time relative to S_i deadline D_i :

1. $D_i \geq \text{IOT}$ is required; otherwise if $D_i < \text{IOT}$, S_i is **IO-Bound**.
2. $D_i \geq \text{ICT}$ is required; otherwise if $D_i < \text{ICT}$, S_i is **CPU-Bound**.
3. $D_i \geq (\text{IOT} + \text{ICT})$ requires no overlap of IOT with ICT.
4. If $D_i < (\text{IOT} + \text{ICT})$ where ($D_i \geq \text{IOT}$ and $D_i \geq \text{ICT}$), overlap of IOT with ICT is required.

5. If $D_i < (IOT + ICT)$ where $(D_i < IOT \text{ or } D_i < ICT)$, deadline D_i can't be met regardless of overlap.

For all five overlap conditions listed here, $ICT > 0$ and $IOT > 0$ must be true. If ICT and IOT are zero, no service is required. If ICT or IOT alone is zero, then no overlap is possible. When $IOT = 0$, this is an ideal service with no intermediate IO.

From these observations about overlap in a non-preemptive system, we can deduce the following axioms:

$$CPI_{\text{worst_case}} = (ICT + IOT) / ICT \quad (4.3)$$

$$CPI_{\text{best_case}} = (\max(ICT, IOT)) / ICT \quad (4.4)$$

$$CPI_{\text{required}} = D_i / ICT \quad (4.5)$$

$$OR = 1 - [(D_i - IOT) / ICT] \quad (4.6)$$

$$CPI_{\text{required}} = [ICT(1 - OR) + IOT] / ICT \quad (4.7)$$

$$NOA = (D_i - IOT) / ICT \quad (4.8)$$

$$OR + NOA = 1 \text{ (by definition)} \quad (4.9)$$

Equations 4.7 and 4.8 provide a cross-check. Equation 4.7 should always match Equation 4.5 as long as condition 4 or 3 is true. Equation 4.9 should always be 1.0 by definition—whatever isn't overlapped must be allowable, or it would be required. When no overlapping of core device IO cycles is possible with core CPU cycles, then the following condition must hold for a service to guarantee a deadline:

$$\begin{aligned} & [\text{Bus_IO_Cycles} \times \text{Core_to_Bus_Factor}] + \\ & \text{Core_Cycles} < WCET_i < D_i \end{aligned} \quad (4.10)$$

The WCET must be less than the service's deadline because we have not considered interference in this CPU-IO overlap analysis. Recall that interference time must be added to release IO latency and WCET:

$$\forall S_i, T_{\text{response}(i)} \leq \text{Deadline}_i$$

$$T_{\text{response}(i)} = T_{\text{IO_Latency}(i)} + WCET_i + T_{\text{Memory_Latency}(i)} + \sum_{j=1}^{i-1} T_{\text{interference}(j)} \quad (4.11)$$

The WCET deduced from Equation 4.10 must therefore be an input into the normal RM feasibility analysis that models interference. The Core-to-Bus-Factor term is ideally 1. This is a zero wait-state case where the processor clock rate and bus transaction rate are perfectly matched. Most often, a read or write will require multiple core cycles. The overlap required (OR) is indicative of how critical execution efficiency is to a service's capability to meet deadlines. If OR is high, then the capability to meet deadlines requires high efficiency, and deadline overruns are likely when the pipeline stalls. In a soft real-time system, it may be reasonable to count on an OR of 10% to 30%, which can be achieved through compiler optimizations (code scheduling), hand optimizations (use of prefetching), and hardware pipeline hazard handling. Note that the ICT and IOT in Figure 4.1 are shown in nanoseconds as an example for a typical 100MHz to 1GHz CPU core executing a typical block of code of 100 to 6,000 instructions with a $CPI_{\text{effective}} = 1.0$. It is not possible to have $OR > 1.0$, so the cases where OR is greater than 1.0 are not feasible.

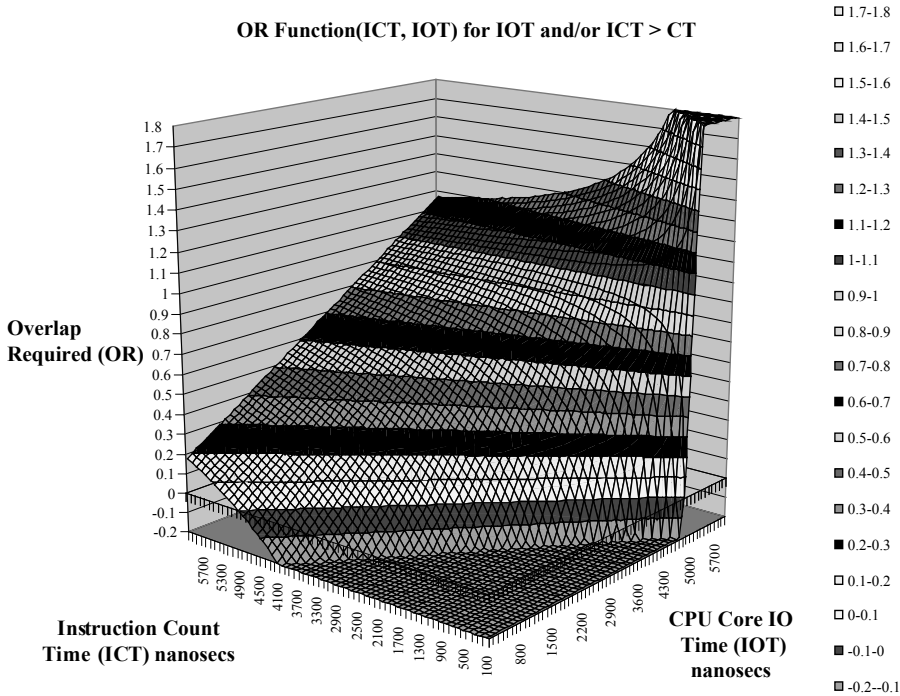


FIGURE 4.1 CPU-IO Overlap Required for Given ICT and IOT

4.4 Execution Efficiency

When WCET is too worst-case, a well-tuned and pipelined CPU architecture increases instruction throughput per unit time and significantly reduces the probability of WCET occurrences. In other words, a pipelined CPU reduces the overall CPI required to execute a block of code. In some cases, IPC (Instructions Per Clock), which is the inverse of CPI, is used as a figure of merit to describe the overall possible throughput of a pipelined CPU. A CPU with better throughput has a lower CPI and a higher IPC. In this text, we will use only CPI, noting that:

$$\text{CPI} = \frac{1}{\text{IPC}}$$

The point of pipelined hardware architecture is to ensure that an instruction is completed every clock for all instructions in the ISA (Instruction Set Architecture). Normally CPI is 1.0 or less overall in modern pipelined systems. Figure 4.2 shows a simple CPU pipeline and its stage overlap such that one instruction is completed (retired) every CPU clock.

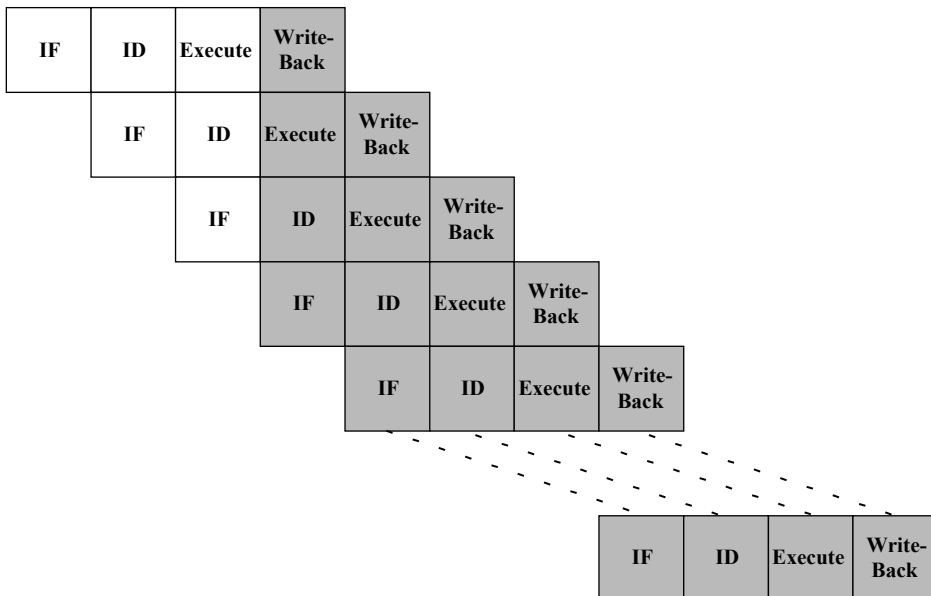


FIGURE 4.2 Simple Pipeline Stage Overlap (Depth=4)

In Figure 4.2, the stages are Instruction Fetch (IF), Instruction Decode (ID), Execute, and register Write-Back. This example pipeline is four stages, so for the pipeline to reach steady-state operation and a CPI of 1.0, it requires four CPU cycles until a Write-Back occurs on every IF. At this point, as long as the stage overlapping can continue, one instruction is completed every CPU clock.

Pipeline design requires minimization of hazards, so the pipeline must stall the one-cycle Write-Backs to produce correct results. The strategies for pipeline design are well described by computer architecture texts [Hennessy03], but are summarized here for convenience. Hazards that may stall the pipeline and increase CPI include the following:

- Instruction and data cache misses, requiring a high-latency cache load from main memory
- High-latency device reads or writes, requiring the pipeline to wait for completion
- Code branches—change in locality of execution and data reference

The instruction and data cache misses can be reduced by increasing cache size, keeping a separate data and instruction cache (Harvard architecture), and allowing the pipeline to execute instructions out of order so that something continues to execute while a cache miss is being handled. The hazard can't be eliminated unless all code and data can be locked into a Level-1 cache (Level-1 cache is single-cycle access to the core by definition).

The high-latency device read/write hazard is very typical in embedded systems where device interfaces to sensors and actuators are controlled and monitored via MMRs. When these devices and their registers are written or read, this can stall the pipeline while the read or write completes. A split-transaction bus interface to device MMRs can greatly reduce the pipeline hazard by allowing reads to be posted and the pipeline to continue until the read completion is really needed—likewise a split-transaction bus allows writes to be posted to a bus interface queue in a single cycle. When a write is posted, the pipeline goes on assuming the MMR write will ultimately complete, but that other instructions in the pipeline do not necessarily need this to complete before they execute.

Finally, code branching hazards can be reduced by branch prediction and speculative execution of both branches—even with branch prediction

and speculative execution, a misprediction typically requires some stall cycles to recover.

By far, the pipeline hazards that contribute most to lowering CPI are cache misses and core bus interface IO latency (e.g., MMR access). Branch mispredictions and other pipeline hazards often result in stalls of much shorter duration (by orders of magnitude) compared to cache misses and device IO. The indeterminism of cache misses can be greatly reduced by locking code in instruction cache and locking data in data cache. A Level-1 instruction and data cache is often too small to lock down all code and all data required, but some architectures include a Level-2 cache, which is much larger and can also be locked. Level-2 cache usually has 2-cycle or more, but less than 10-cycle access time—locking code and data into L2 cache is much like having a 1 or more wait-state memory. Low wait-state memory is often referred to as a TCM (Tightly Coupled Memory). This is ideal for a real-time embedded system because it eliminates much of the non-determinism of cache hit/miss ratios that are data-stream- and instruction-stream-driven. An L2 cache is often unified (holds instructions and data) and 256 KB, 512 KB, or more in size. So, you should lock all real-time service code and data into L1 or L2 caches, most often L2, leaving L1 to increase efficiency with dynamic loading. All best-effort service code and data segments can be kept in main memory because deadlines and determinism are not an issue, and any unlocked L1 or L2 cache increases the execution efficiency of these main-memory-based best-effort services.

Eliminating non-determinism of device IO latency and pipeline hazards is very difficult. When instructions are allowed to execute while a write is draining to a device, this is called *weakly consistent*. This is okay in many circumstances, but not when the write must occur before other instructions not yet executed for correctness. Posting writes is also ultimately limited by the posted write bus interface queue depth—when the queue is full, subsequent writes must stall the CPU until the queue is drained by at least one pending write. Likewise, for split-transaction reads, when an instruction actually uses data from the earlier executed read instruction, then the pipeline must stall until the read completes. Otherwise the dependent instruction would execute with stale data, and the execution would be errant. A stall where the pipeline must wait for a read completion is called a *data-dependency stall*. When split-transaction reads are scheduled with a register as a destination, this can create another hazard called register pressure—the register awaiting read completion is tied up and can't be used at all by other instructions

until the read completes, even though they are not dependent upon the read. You can reduce register pressure by adding a lot of general-purpose registers (most pipelined RISC architectures have dozens and dozens of them) as well as by providing an option to read data from devices into cache. Reading from a memory-mapped device into cache is normally done with a cache pre-fetch instruction. In the worst case, we must assume that all device IO during execution of a service stalls the pipeline so that

$$\text{WCET} = \left[(\text{CPI}_{\text{best_case}} \times \text{Longest_Path_Instruction_Count}) + \text{Stall_Cycles} \right] \times \text{Clock_Period}$$

If you can keep the stall cycles to a deterministic minimum by locking code into L2 cache (or L1 if possible) and by reducing device IO stall cycles, then you can also reduce WCET. Cache locking helps immensely and is fully deterministic.

4.5 IO Architecture

In this chapter, IO has been examined as a resource in terms of latency (time) and bandwidth (bytes/second). The view has been from the perspective of a single processor core and IO between that processor core and peripherals. The emergence of advanced ASIC architectures, such as SoC (System on Chip), has brought about embedded single-chip system designs that integrate multiple processors with many peripherals in more complex interconnections than BIU (Bus Interface Unit) designs. Figure 4.3 provides an overview of the many interconnection networks that can be used on-chip and between multichip or even multi-subsystem designs, including traditional bus architectures.

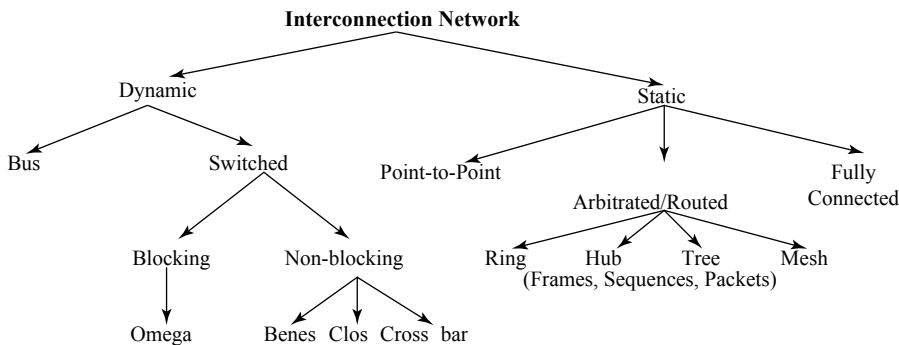


FIGURE 4.3 Taxonomy of Interconnection Networks

The cross-bar interconnection fully connects all processing and IO components without any blocking. The cross-bar is said to be dynamic because a matrix of switches must be set to create a pathway between two end points, as shown in Figure 4.4. The number of switches required is a quadratic function of the number of end points such that N points can be connected by N^2 switches—this is a costly interconnection. Blocking occurs when the connection between two end points prevents the simultaneous connection between two others due to common pathways that can't be used simultaneously. The bus interconnection, like a cross-bar, is dynamic, but is fully blocking because it must be time-multiplexed and allows no more than two end points within the entire system to be connected at once.

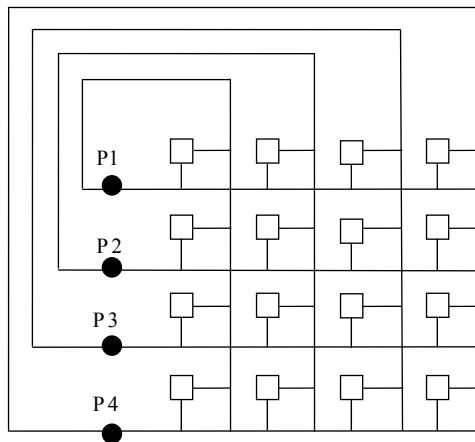


FIGURE 4.4 Cross-Bar Interconnection Network

Summary

The overall response time of a service includes input, intermediate IO, and output latency. The input and output latencies can most often be determined and added to the overall response time. Intermediate IO is more complex because intermediate IO includes register, memory bus, and, in general, interconnection network latencies that will stall an individual processor's execution. The stall time reduces execution efficiency for each processor, and unless these stall cycles can be used for other work, increases WCET for the service.

Exercises

1. If a processor has a cache hit rate of 99.5% and a cache miss penalty of 160 core processor cycles, what will the average CPI be for 1,000 instructions?
2. If a system must complete frame processing so that 100,000 frames are completed per second and the instruction count per frame processed is 2,120 instructions on a 1GHz processor core, what is the CPI required for this system? What is the overlap between instructions and IO time if the intermediate IO time is 4.5 microseconds?
3. Read Sha, Rajkumar, et al.'s paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." [ShaRajkumar90] Write a brief summary, noting at least three key concepts from this paper.
4. Review the DVD code for `heap_mq.c` and `posix_mq.c`. Write a brief paragraph describing how these two message queue applications are similar and how they are different. Make sure you not only read the code but also build it, load it, and execute it to make sure you understand how both applications work.
5. Write VxWorks code that spawns two tasks: A and B. A should initialize and enter a `while(1)` loop in which it does a `semGive` on a global binary semaphore S1 and then does a `semTake` on a different global binary semaphore S2. B should initialize and enter a `while(1)` loop in which it does a `semTake` of S1, delays 1 second, and then does a `semGive` of S2. Test your code on a target or VxSim and turn in all sources with evidence that it works correctly (e.g., show counters that increment in windshell dump).
6. Now run the code from the previous problem and analyze a WindView (now System Viewer) trace for it. Capture WindView trace and add annotations by hand, clearly showing `semGive` and `semTake` events, the execution time, and delay time, and note any unexpected aspects of the trace.
7. Use `taskSwitchHookAdd` and `taskSwitchHookDelete` to add some of your own code to the Wind kernel context switch sequence, and prove it works by increasing a global counter for each preempt/dispatch context switch and time-stamping a global with the last preempt/dispatch time with the x86 PIT (programmable interval timer)—see the DVD



sample PIT code. The VxWorks “tickGet()” call can also be used to sample relative times, but is accurate only to the tick resolution (1 millisecond assuming sysClkRateSet(1000)). Make an On/Off wrapper function for your add and delete so that you can turn on your switch hook easily from a windsh command line and look at the values of your two globals. Turn in your code and windshell output showing that it works.

8. Modify your hook so that it will compute separate preempt and dispatch counts for a specific task ID or set of task IDs (up to 10) and the last preempt and dispatch times for all tasks you are monitoring. Run your code from #4, and monitor it by calling your On/Off wrapper before running your test tasks. What are your counts and last times and how do they compare with WindView analysis?
9. Use your program to analyze the number of tNetTask dispatches/preemptions and modify the code to track the average time between dispatch and preemption. Write a program to gather stats on tNetTask for 30 seconds. What is the number of dispatches/preemptions? What is the average dispatch time?

Chapter References

- [Almasi89] George Almasi and Allan Gottlieb, *Highly Parallel Computing*. Benjamin/Cummings, San Francisco, 1989.
- [Hennessy03] John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, New York, 2003.
- [Patterson94] David Patterson and John Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, New York, 1994.
- [ShaRajkumar90] Lui Sha, Rangunthan Rajkumar, John P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, IEEE Transactions on Computers, Volume 39, Number 9, September 1990.

MEMORY

In this chapter

- Introduction
- Physical Hierarchy
- Capacity and Allocation
- Shared Memory
- ECC Memory
- Flash File Systems

5.1 Introduction

In the previous chapter, memory was analyzed from the perspective of latency, and, in this sense, was treated like most any other IO device. For a real-time embedded system, this is a useful way to view memory, although it's very atypical compared to general-purpose computing. In general, memory is typically viewed as a logical address space for software to use as a temporary store for intermediate results while processing input data to produce output data. The physical address space is a hardware view where memory devices of various type and latency are either mapped into address space through chip selects and buses or hidden as caches for mapped devices. Most often an MMU (Memory Management Unit) provides the logical-to-physical address mapping (often one-to-one for embedded systems) and provides address range and memory access attribute checking. For example, some memory segments may have attributes set so they are read-only. Typically, all code is placed in read-only attribute memory segments. Memory-Mapped IO

(MMIO) address space most often has a non-cacheable attribute set to prevent output data from being cached and never actually written to a device. From a higher-level software viewpoint, where memory is viewed as a global store and an interface to MMIO, it's often useful to set up shared memory segments useable by more than one service. When memory is shared by more than one service, care must be taken to prevent inconsistent memory updates and reads. From a resource perspective, total memory capacity, memory access latency, and memory interface bandwidth must be sufficient to meet requirements.

5.2 Physical Hierarchy

The physical memory hierarchy for an embedded processor can vary significantly based upon hardware architecture. However, most often, the Harvard architecture is used, which has evolved from GPCs (general-purpose computers) and is often employed by embedded systems as well. The typical Harvard architecture with separate L1 (Level-1) instruction and data caches but with unified L2 (Level-2) cache and either on-chip SRAM or external DDR (Dynamic Data RAM) is shown in Figure 5.1.

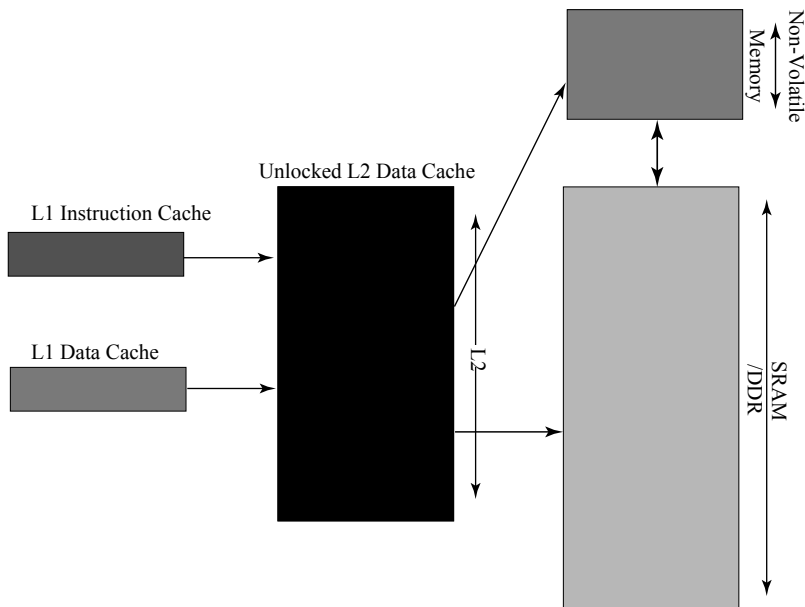


FIGURE 5.1 Harvard Architecture Physical Memory Hierarchy

From the software viewpoint, memory is a global resource in a single address space with all other MMIO devices as shown in Figure 5.2.

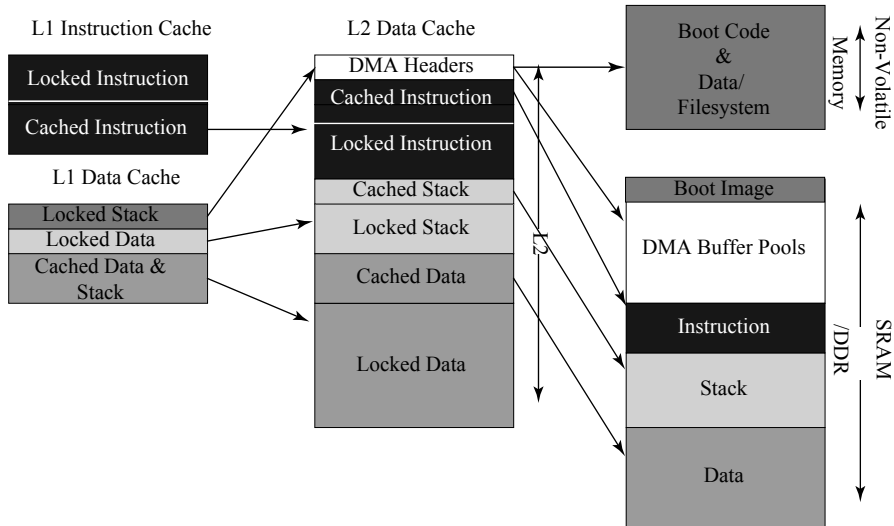


FIGURE 5.2 Logical Partitioning and Segmenting of Physical Memory Hierarchy by Firmware

Memory system design has been most influenced by GPC architecture and goals to maximize throughput, but not necessarily to minimize the latency for any single memory access or operation. The multilevel cached memory hierarchy for GPC platforms now often includes Harvard L1 and unified L2 caches on-chip with off-chip L3 unified cache. The caches for GPCs are most often set-associative with aging bits for each cache set (line) so that the LRU (Least Recently Used) sets are replaced when a cache line must be loaded. An N -way set-associative cache can load an address reference into any N ways in the cache, allowing for the LRU line to be replaced. The LRU replacement policy, or approximation thereof, leads to a high cache hit-to-miss ratio so that a processor most often finds data in cache and does not have to suffer the penalty of a cache miss. The set-associative cache is a compromise between a direct-mapped and a fully associative cache. In a direct-mapped cache, each address can be loaded into one and only one cache line, making the replacement policy simple, yet often causing cache thrashing. *Thrashing* occurs when two addresses are referenced and keep knocking each other out of cache, greatly decreasing cache efficiency. Ideally, a cache memory would be so flexible that the LRU set (line) for the entire cache would be replaced each time, minimizing

the likelihood of thrashing. Cost of fully associative array memory prevents this, as does the cost of so many aging bits, and most caches are four-way or eight-way set-associative with 2 or 3 aging bits for LRU. Figure 5.3 shows a direct-mapped cache where a main memory that is four times the size of the cache memory has memory sets (collections of four or more 32-bit byte addressable words), which can be loaded only into one cache set.

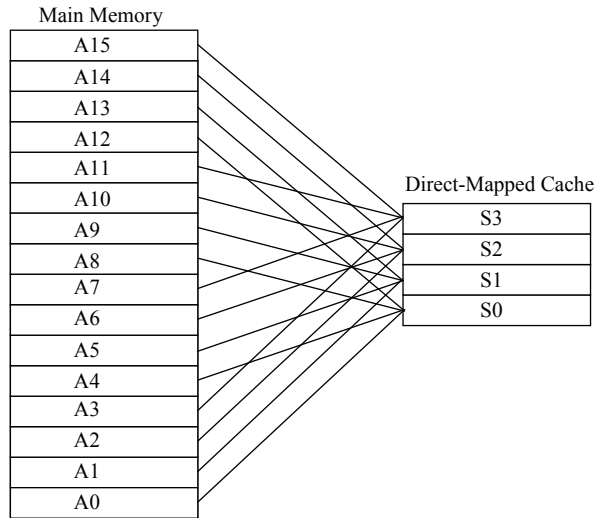


FIGURE 5.3 Direct Mapping of Memory to Cache Lines (Sets)

By comparison, Figure 5.4 shows a two-way set-associative cache for a main memory four times the size of the cache. Each line can be loaded into one of four locations in the cache. The addition of 2 bits to record how recently each line was accessed relative to the other three in the set allows the cache controller to replace lines that are LRU. The LRU policy assumes that lines that have not been accessed recently are less likely to be accessed again anytime soon. This has been shown to be true for most code where execution has locality of reference, where data is used within small address ranges (often in loops) distributed throughout memory for general-purpose programs.

For real-time embedded systems, the unpredictability of cache hits/misses is a problem. It makes it very difficult to estimate WCET (Worst-Case Execution Time). In the extreme case, it's really only safe to assume that every cache access could incur the miss penalty. For this reason, for hard real-time systems, it's perhaps advisable not to use cache. However, this would greatly reduce throughput to obtain deterministic execution. So,

rather than including multilevel caches, many real-time embedded systems instead make investment in TCM (Tightly Coupled Memory), which is single-cycle access for the processor, yet has no cache functionality.

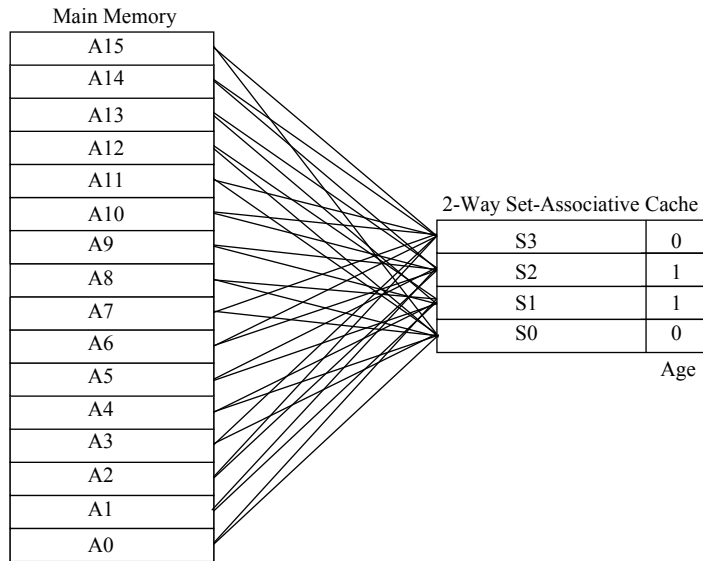


FIGURE 5.4 Two-Way Set-Associative Mapping of Memory to Cache Lines (Sets)

Furthermore, TCM is often dual ported so that service data (or context) can be loaded via DMA and read/written at the same time. Some GPC processors include L1 or L2 caches that have the capability to lock ways so that the cache can be turned into this type of TCM for embedded applications.

The realization that GPC cache architecture is not always beneficial to real-time embedded systems, especially hard real-time systems, has led to the emergence of the software-managed cache. Furthermore, compared to GPCs, where it is very difficult to predict the care with which code will be written, embedded code is often carefully analyzed and optimized so that data access is carefully planned. A software-managed cache uses application-specific logic to schedule loading of a TCM with service execution context by DMA from a larger, higher-latency external memory. Hardware cost of a GPC cache architecture is avoided, and the management of execution context tailored for the real-time services—the scheduling of DMAs and the worst-case delay for completing a TCM load—must, of course, still be carefully considered.

5.3 Capacity and Allocation

The most basic resource concern associated with memory should always be total capacity needed. Many algorithms include space and time trade-offs, and services often need significant data context for processing. Keep in mind that cache does not contribute to total capacity because it stores only copies of data rather than unique data. This is another downside to cache for embedded systems where capacity is often limited. Furthermore, latency for access to memory devices should be considered carefully because high latency access can significantly increase WCET and cause problems meeting real-time deadlines. So, data sets accessed with high frequency should, of course, be stored in the lowest latency memory.

5.4 Shared Memory

Often two services find it useful share a memory data segment or code segment. In the case of a shared data segment, the read/write access to this shared memory must be guaranteed to be consistent so that one service in the middle of a write is not preempted by another, which could then read partially updated data. For example, a satellite system might sample sensors and store the satellite state vector in a shared memory segment. The satellite state vector would typically include 3 double precision numbers with the X, Y, Z location relative to the center of Earth, 3 double precision numbers for the velocity, and 3 more double precision numbers with the acceleration relative to Earth. Overall, this state would likely have 9 double precision numbers that can't be updated in a single CPU cycle. Furthermore, the state vector might also contain the attitude, attitude rate, and attitude acceleration—up to 18 numbers! Clearly it would be possible for a service updating this state to be preempted via an interrupt, causing a context switch to another service that uses the state. Using a partially updated state would likely result in control being issued based upon corrupt state information and might even cause loss of the satellite.

To safely allow for shared memory data, the mutual exclusion semaphore was introduced [Tanenbaum87]. The mutex semaphore protects a critical section of code that is executed to access the shared memory resource. In VxWorks, two functions support this: `semTake(Semid)` and `semGive(Semid)`. The `semTake(Semid)` blocks the calling service if `Semid=0` and allows it to continue execution if `Semid=1`. The `semGive(Semid)` increments `Semid` by 1, leaving it at 1 if it is already 1. When the `semGive(Semid)` causes `Semid` to go from 0 to 1, any services blocked earlier by a call to `semTake(Semid)`

when `Semid` was 0 now unblocks one of the waiting services. Typically the waiting services are unblocked in first-in, first-out order. By surrounding the code that writes and/or reads the shared memory with `semTake(Semid)` and `semGive(Semid)` using a common `Semid`, the updates and reads are guaranteed to be mutually exclusive. The critical sections are shown in Table 5.1.

Table 5.1 Shared Memory Writer and Reader Example

Update-Code	Read-Code
...	...
<code>semTake(Semid);</code>	<code>semTake(Semid);</code>
<code>X = getX();</code>	<code>control(X, Y, Z);</code>
<code>Y = getY();</code>	<code>semGive(Semid);</code>
<code>Z = getZ();</code>	...
<code>semGive(Semid);</code>	...
...	...

Clearly if the Update-Code was interrupted and preempted by the Read-Code at line 4, for example, then the `control(X, Y, Z)` function would be using the new `X` and possibly an incorrect and definitely old `Y` and `Z`. However, the `semTake(Semid)` and `semGive(Semid)` guarantee that the Read-Code can't preempt the Update-Code no matter what the RM policies are. How does it do this? The `semTake(Semid)` is a TSL instruction (Test and Set-Lock). In a single cycle, supported by hardware, the `Semid` memory location is first tested to see if it is 0 or 1. If 1, set to 0, and execution continues; if `Semid` is 0 on the test, the value of `Semid` is unchanged, and the next instruction is a branch to a wait-queue and CPU yield. Whenever a service does a `semGive(Semid)`, the wait-queue is checked and the first waiting service is unblocked. The unblocking is achieved by dispatching the waiting service from the wait-queue and then placing it on the ready-queue for execution inside the critical section at its normal priority.

5.5 ECC Memory

For safety-critical real-time embedded systems it is imperative that data corruption not go undetected and ideally should be corrected in real time if possible. This can be accomplished by using ECC (Error Correcting Circuitry) memory interfaces. An ECC memory interface can detect and correct SBEs (Single Bit Errors) and also can detect MBEs (Multi-Bit Errors), but cannot correct MBEs. One common ECC SBE detection and correction method used is Hamming code, which is most often

implemented as a SECDED (Single Error Correction, Double Error Detection) method as described here. When data is written to memory, parity bits are calculated according to a Hamming encoding and added to a memory word extension (an additional 8 bits for a 32-bit word). When data is read out, check bits are computed by the ECC logic. These check bits, called the syndrome, encode read data errors as follows:

1. If Check-Bits = 0 AND parity-encoded-word = parity-read-word => NO ERRORS
2. If Check-Bits != 0 AND parity-encoded-word != parity-read-word => SBE, CAN CORRECT
3. If Check-Bits != 0 AND parity-encoded-word = parity-read-word => DOUBLE BIT ERROR DETECTED, HALT
4. If Check-Bits = 0 AND parity-encoded-word != parity-read-word => parity-word ERROR

On double bit MBEs, the processor normally halts since the next instruction executed with unknown corrupted data could cause a fatal error. By halting the CPU will go through a hardware watchdog timer reset and safe recovery instead.

For a correctable SBE the CPU raises a non-maskable interrupt, which software should handle by acknowledging and then reading the address of the SBE location, and finally writing the corrected data back to the memory location from the read register. The ECC automatically corrects data as it is read from memory into registers, but most often it is up to the software to write the corrected data in the register back to the corrupted memory location. By halting, the CPU will go through a hardware watchdog timer reset and safe recovery. However, for the most common Hamming code formulation presented here, MBEs beyond double bit can't be reliably detected and will lead to errant behavior. Most often, ECC is used to protect memory from SEUs (Single Event Upsets), where a single bit flips at a time due to particle radiation from the solar wind or background cosmic radiation. So, while space-based systems, high altitude aircraft and very rarely some terrestrial computing systems are impacted by SEUs, the failure mode is such that normally only one or two bits will flip. The Hamming ECC methods are not sufficient to protect against more significant large impact memory failures. Corruption of many bits or loss of memory banks should be protected by mirroring memory devices, or mirroring in addition to use of Hamming ECC.

To understand how Hamming encoding works to compute extended parity and a syndrome capable of SBE detection/correction and double bit MBE detection, it is best to learn by example. Figure 5.5 shows a Hamming encoding for an 8-bit word with 4 parity bits and an overall word parity bit shown as pW.

In the Figure 5.6 example, the syndrome computed is nonzero and the pW is not equal to PW2, which is the case for a correctable SBE. Notice that the syndrome value of binary 1010 encodes the errant bit position for bit-5, which is data bit d02. Figure 5.7 illustrates a double bit flip MBE scenario.

Note that the syndrome is nonzero, but pW1=pW2, which can only happen for 2, 4, 6, or 8 bit flips, meaning that an uncorrectable double bit MBE has occurred. Figure 5.8 shows a scenario where the ED parity is corrupted.

In this case, the syndrome is zero and pW1=pW2, indicating a parity error on the overall encoded data word. This is a correctable SBE. Similarly, it's possible that one of the Hamming extended parity bits could be flipped as a detectable and correctable SBE, as shown in Figure 5.9.

This covers all the basic correctable SBE possibilities as well as showing how an uncorrectable MBE is detected. Overall, the Hamming encoding provides

			0	1	2	3	4	5	6	7	8	9	10	11	12
			pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
	bit	D	X	X	X	1	X	1	0	0	X	0	1	0	0
Data even parity	1	p01		0		1		1		0		0		0	
	2	p02			0	1			0	0			1	0	
Data odd parity (p03=1)	4	p03					1	1	0	0					0
	8	p04									1	0	1	0	0
	16	p05													
	32	p06													
ED odd parity (so pW=1)		ED	1	0	0	1	1	1	0	0	1	0	1	0	0
			0	1	2	3	4	5	6	7	8	9	10	11	12
		SYN	pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
Check bits: No difference	0	ED	1	0	0	1	1	1	0	0	1	0	1	0	0
	c01	0		0		1		1		0		0		0	
	c02	0			0	1			0	0			1	0	
	c03	0					1	1	0	0					0
	c04	0									1	0	1	0	0
	c05	X													
	c06	X													
pW2 == pW (as expected)	1	pW2	1	1	0	0	1	1	1	0	0	1	0	1	0
	0	CD	1	0	0	1	1	1	0	0	1	0	1	0	0
Check-Bits == 0 AND pW == pW2 => NO ERRORS															

FIGURE 5.5 Hamming Encoding for 8-Bit Word with No Error in Computed Syndrome

perfect detection of SBEs and MBEs, unlike a single parity bit for a word, which can detect all SBEs, but can detect MBEs only where an odd number of bits are flipped. In this sense, a single parity bit is an imperfect detector and also offers no correction because it does not encode the location of SBEs.

		0	1	2	3	4	5	6	7	8	9	10	11	12
		pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
bit	D	X	X	X	1	X	1	0	0	X	0	1	0	0
1	p01		0		1		1		0		0		0	
2	p02			0	1		0	0				1	0	
4	p03					1	1	0	0					0
8	p04									1	0	1	0	0
16	p05													
32	p06													
	ED	1	0	0	1	1	1	0	0	1	0	1	0	0
		0	1	2	3	4	5	6	7	8	9	10	11	12
		pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
SBE, bit-flipped (d02 from 1 to 0)	5	ED	1	0	0	1	1	0	0	1	0	1	0	0
SYN encodes bit position (flip ED[5] to restore)	c01	1		1		1		0		0		0		0
	c02	0			0	1		0	0			1	0	
	c03	1				0	0	0	0					0
	c04	0								1	0	1	0	0
	c05	X												
	c06	X												
pW2 != pW (parity error)	pW2	0	1	0	0	1	1	0	0	1	0	1	0	0
	5	CD	1	0	0	1	1	1	0	0	1	0	1	0
Check-Bits != 0 AND pW != pW2 => SBE, CAN CORRECT														

FIGURE 5.6 Hamming Syndrome Catches and Corrects pW SBE

		0	1	2	3	4	5	6	7	8	9	10	11	12
		pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
bit	D	X	X	X	1	X	1	0	0	X	0	1	0	0
1	p01		0		1		1		0		0		0	
2	p02			0	1		0	0				1	0	
4	p03					1	1	0	0					0
8	p04									1	0	1	0	0
16	p05													
32	p06													
	ED	1	0	0	1	1	1	0	0	1	0	1	0	0
		0	1	2	3	4	5	6	7	8	9	10	11	12
		pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
MBE, 2 bits-flipped (d02 and d05 both)	12	ED	1	0	0	1	1	0	0	1	1	1	0	0
SYN encodes MBE (out of range for ECC)	c01	0		0		1		0		0		1		0
	c02	0			0	1		0	0			1	0	
	c03	1				0	0	0	0					0
	c04	1								0	1	1	0	0
	c05	X												
	c06	X												
pW != pW2 (double MBE)	pW2	1	1	0	0	1	1	0	0	1	1	1	0	0
	MBE	CD	1	0	0	1	1	1	0	0	1	0	1	0
Check-Bits != 0 AND pW == pW2 => DOUBLE MBE DETECTED														

FIGURE 5.7 Hamming Syndrome Catches MBE, but Can't Correct the Data

		0	1	2	3	4	5	6	7	8	9	10	11	12
		pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
bit	D	X	X	X	1	X	1	0	0	X	0	1	0	0
1	p01		0		1		1		0		0		0	
2	p02			0	1			0	0			1	0	
4	p03					1	1	0	0					0
8	p04									1	0	1	0	0
16	p05													
32	p06													
	ED	1	0	0	1	1	1	0	0	1	0	1	0	0
		0	1	2	3	4	5	6	7	8	9	10	11	12
	SYN	pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
8	ED	1	0	0	1	1	1	0	0	0	0	1	0	0
c01	0		0		1		1		0		0		0	
c02	0			0	1			0	0			1	0	
c03	0					1	1	0	0					0
c04	1									1	0	1	0	0
c05	X													
c06	X													
pW2	0	1	0	0	1	1	1	0	0	0	0	1	0	0
8	CD	1	0	0	1	1	1	0	0	1	0	1	0	0

Check-Bits != 0 AND pW != pW2 => Parity SBE, CAN CORRECT

FIGURE 5.8 Hamming Syndrome Catches and Corrects pW SBE

		0	1	2	3	4	5	6	7	8	9	10	11	12
		pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
bit	D	X	X	X	1	X	1	0	0	X	0	1	0	0
1	p01		0		1		1		0		0		0	
2	p02			0	1			0	0			1	0	
4	p03					1	1	0	0					0
8	p04									1	0	1	0	0
16	p05													
32	p06													
	ED	1	0	0	1	1	1	0	0	1	0	1	0	0
		0	1	2	3	4	5	6	7	8	9	10	11	12
	SYN	pW	p01	p02	d01	p03	d02	d03	d04	p04	d05	d06	d07	d08
0	ED	0	0	0	1	1	1	0	0	1	0	1	0	0
c01	0		0		1		1		0		0		0	
c02	0			0	1			0	0			1	0	
c03	0					1	1	0	0					0
c04	0									1	0	1	0	0
c05	X													
c06	X													
pW2	1	0	0	0	1	1	1	0	0	1	0	1	0	0
0	CD	1	0	0	1	1	1	0	0	1	0	1	0	0

Check-Bits == 0 AND pW != pW2 => pW ERROR

FIGURE 5.9 Hamming Syndrome Catches and Corrects Corrupt Parity Bit SBE

5.6 Flash File Systems

Flash technology has mostly replaced the use of EEPROM (Electrically Erasable Programmable Read-Only Memory) as the primary updateable and nonvolatile memory used for embedded systems. Usage of EEPROM continues for low-cost, low-capacity NVRAM (NonVolatile RAM) system requirements, but most often Flash memory technology is used for boot code storage and for storage of nonvolatile files. Flash file systems emerged not long after Flash devices. Flash offers in-circuit read, write, erase, lock, and unlock operations on sectors. The erase and lock operations operate on the entire sector, most often 128 KB in size. Read and write operations can be a byte, word, or block. The most prevalent Flash technology is NAND (Not AND) Flash, where the memory device locations are erased to all 1s, and writes are the NAND of the write data and the current memory contents. Data can be written only to erased locations with all 1s unless it is unchanged, and sectors can typically be erased 100,000 times or more before they are expected to fail as read-only memory.

Based upon the characteristics of Flash, it's clear that minimizing sector erases will maximize Flash part lifetime. Furthermore, if sectors are erased at an even pace over the entire capacity, then the full capacity will be available for the overall expected lifetime. This can be done fairly easily for usages such as boot code and boot code updates. Each update simply moves the new write data above the previously written data and sectors are erased as needed. When the top of Flash is encountered, the new data wraps back to the beginning sector. The process results in even wear because all sectors are erased in order over time. File systems are more difficult to implement with wear leveling because arbitrarily sized files and a multiplicity of them lead to fragmentation of the Flash memory. The key to wear leveling for a Flash file system is to map file system blocks to sectors so that the same sector rotation and even number of erases can be maintained as was done for the single file boot code update scheme. A scenario for mapping two files, each with two 512-byte LBAs (Logical Block Addresses), and a Flash device with two sectors, each sector 2,048 bytes in size (for simplicity of the example), shows that 16 LBA updates can be accommodated for 5 sector erases. This scenario is shown in Figure 5.10 for 16 updates and total of 5 sector erases (3 for Sector 0 and 2 for Section 1). Continuing this pattern 32 updates can be completed for 10 sector erases (5 for section 0 and 5 for sector 1). In general this scenario has approximately six times more updates than evenly distributed erases.

With wear leveling, Flash is expected to wear evenly so that the overall lifetime is the sum of the erase limits for all the sectors rather than the limit on one, yielding a lifetime with millions of erases and orders of magnitude more file updates than this. Wear leveling is fundamental to making file system usage with Flash practical.

		#1 - Start	#2	#3	#4	#5	#6	#7
	Sector Erased (S0, S1)	0,0	1,1	1,1	1,1	1,1	2,1	2,1
#1 – All blocks FREE	S1							
#2 – Erase S0 & S1, Write LB 0, 1, 2, 3		PB7	FREE	FREE	FREE	LB3	LB3	LB3
#3 – Read LB 0, 2, Modify, Write LB 0, 2		PB6	FREE	FREE	LB2	LB2	INVLD	INVLD
#4 – Read LB 1, 3, Modify, Write LB 1, 3		PB5	FREE	LB3	LB3	INVLD	INVLD	INVLD
#5 – Read LB 0, 2, Modify and Cache		PB4	FREE	LB2	INVLD	INVLD	INVLD	INVLD
#6 – Buffer LB 0, 1, 2, Erase S0	S0							
#7 – Write-back LB 0, 1, 2 to S0		PB3	FREE	FREE	FREE	LB1	LB1	FREE
11 Writes, 3 Sector Erases		PB2	FREE	FREE	LB0	LB0	INVLD	FREE
Write Amplification = 11 / 10 = 1.1		PB1	FREE	LB1	LB1	INVLD	INVLD	FREE
		PB0	FREE	LB0	INVLD	INVLD	FREE	LB2
	FS LBs Updated		0,1,2,3	0,2	1,3	0,2	0,2	0,2
	FS LBs Cached					0,2	0,2	
	Sector LBs Buffered						1	
	Sectors Erased (S0, S1)	2,1	#8	#9	#10	#11	#12	#13
	S1	2,1	2,1	2,2	2,2	2,2	2,2	3,2
Start State from End State #7 Above		LB3	INVLD	FREE	FREE	LB2	LB2	LB2
#8- Read LB 1, 3, Modify and Cache		INVLD	INVLD	FREE	FREE	LB0	LB0	LB0
#9 – Erase S1		INVLD	INVLD	FREE	LB3	LB3	INVLD	INVLD
#10 – Write-back LB 1, 3 to S1		INVLD	INVLD	FREE	LB1	LB1	INVLD	INVLD
#11 – Read LB 0, 2, Modify, Write LB 0, 2	S0							
#12 – Read LB 1, 3, Modify and Cache		LB1	INVLD	INVLD	INVLD	INVLD	FREE	FREE
#13 – Erase S0		FREE	FREE	FREE	FREE	FREE	FREE	FREE
#14 – Write-back LB 1, 3		LB2	LB2	LB2	LB2	INVLD	INVLD	FREE
6 Writes, 2 Sector Erases		LB0	LB0	LB0	LB0	INVLD	INVLD	FREE
Write Amplification = 17 / 16 = 1.0625	FS LBs Updated	0,2	1,3	1,3	1,3	0,2	1,3	1,3
	FS LBs Cached		1,3	1,3			1,3	1,3
	Sector LBs Buffered							

FIGURE 5.10 Simple Flash Wear Leveling Example

Summary

Memory resource capacity, access latency, and overall access bandwidth should be considered when analyzing or designing a real-time embedded memory system. While most GPC architectures are concerned principally with memory throughput and capacity, real-time embedded systems are comparatively more cost-sensitive and often are designed for less throughput, but with more deterministic and lower worst-case access latency.

Exercises

1. Describe why a multilevel cache architecture might not be ideal for a hard real-time embedded system.
2. Write VxWorks code that starts two tasks, one which writes one of two phrases alternatively to a shared memory buffer, including: (1) “The quick brown fox jumps over the lazy dog” and (2) “All good dogs go to heaven” and when it’s done have it call `TaskDelay(143)`. Now add a second task that reads the phrases and prints them out. Make the reader task higher priority than the writer and have it call `TaskDelay(37)`. How many times does the reader task print out the correct phrases before they become jumbled?
3. Fix the problem in #2 using `semTake()` and `semGive` in VxWorks to protect the readers/writers critical section.
4. Develop an example of a 32-bit Hamming encoded word (39 bits total) and show a correctable SBE scenario.
5. For the foregoing problem, now show an uncorrectable MBE scenario.

Chapter References

- [Mano91] Morris M. Mano, *Digital Design*, 2nd ed., Prentice-Hall, Upper Saddle River New Jersey, 1991.
- [Tanenbaum87] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Upper Saddle River New Jersey, 1987.

MULTIRESOURCE SERVICES

In this chapter

- Introduction
- Blocking
- Deadlock and Livelock
- Critical Sections to Protect Shared Resources
- Priority Inversion
- Power Management and Processor Clock Modulation

6.1 Introduction

Ideally, the service response as illustrated in Figure 3.13 of Chapter 3 is based upon input latency, acquiring the CPU resource, interference, and output latency only. However, many services need more resources than just the CPU to execute. For example, many services may need mutually exclusive access to shared memory resources or a shared intermediate IO resource. If this resource is not in use by another service, this presents no problem. However, if a service is released and preempts another running service based upon RM policy, only to find that it lacks a resource held by another service, then it is blocked. When a service is blocked, it must yield the CPU despite the RM policy. We hope that the preempted service that holds the additional resource will complete its mutually exclusive use of that resource, at which time the high-priority service will unblock, potentially preempt other services, and continue execution. However, as you'll

see in this chapter, blocking may not be temporary if conditions are sufficient for deadlock or priority inversion.

6.2 Blocking

Blocking occurs anytime a service can be dispatched by the CPU, but isn't because it is lacking some other resource, such as access to a shared memory critical section or access to a bus. When blocking has a known latency, it could simply be added into response time, accounted for, and therefore would not adversely affect RM analysis, although it would complicate it. The bigger concern is unbounded blocking, where the amount of time a service will be blocked awaiting a resource is indefinite or at least hard to calculate. Three phenomena related to resource sharing can cause this: deadlock, livelock, and unbounded priority inversion. Deadlock and livelock are always unbounded by definition. Priority inversion can be temporary, but under certain conditions, priority inversion can be indefinite. Blocking can be extremely dangerous because it can cause a very underutilized system to miss deadlines. This is counterintuitive—how can a system with only 5% CPU loading miss deadlines? If a service is blocked for an indefinite time, then the CPU is yielded for an indefinite time, leaving plenty of CPU margin, but the service fails to produce a response by its deadline.

6.3 Deadlock and Livelock

In Figure 6.1, service S1 needs resources A and C, S2 needs A and B, and S3 needs B and C. If S1 acquires A, then S2 acquires B, then S3 acquires C followed by requests by each for their other required resource, a circular wait evolves, as shown in Figure 6.1. *Circular wait*, also known as the deadly embrace, causes indefinite deadlock. No progress can be made by Services 1, 2, or 3 in Figure 6.1 unless resources held by each are released. Deadlock can be prevented by making sure the circular wait scenario is impossible. It can be detected if each service is designed to post a keep-alive, as discussed in Chapter 14, “High Availability and Reliability Design.” When the keep-alive is not posted due to the deadlock, then a supervisory service can restart the deadlocked service. However, it's possible that when deadlock is detected and services are restarted, they could simply reenter the deadlock over and over. This variant is called *livelock* and also prevents progress completely despite detection and breaking of the deadlock.

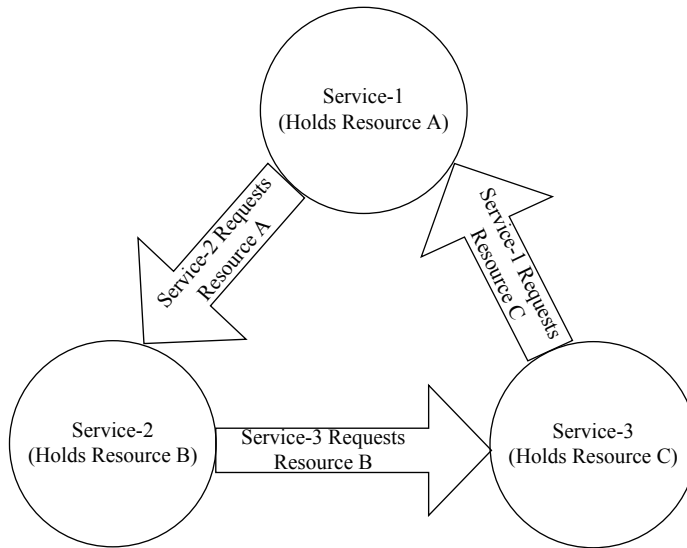


FIGURE 6.1 Shared Resource Deadlock (Circular Wait)

One solution to prevent livelock following deadlock detection and re-starting is to include a random back-off time on restart for each service that was involved—this ensures that one beats the other two to the resource subset needed and completes acquisition, allowing each service the same opportunity in turn.

What if the supervisor service is involved in the deadlock? In this case, a hardware timer known as a watchdog timer is used to require the supervisor service to reset a countdown timer periodically. If the supervisor service becomes deadlocked and can't reset the watchdog timer countdown, then the watchdog timer expires and resets the entire system with a hardware reset. This causes the firmware to reboot the system in hopes that the supervisor deadlock will not evolve again. These strategies will be discussed in more detail in Chapter 11, which covers high-availability and high-reliability design methods.

Even with random back-off, the amount of time that a service will fail to make progress is hard to predict and will likely cause a deadline to be missed, even when the CPU is not highly loaded. The best solution is to eliminate the conditions necessary for circular wait. One method of avoidance is to require a total order on the locking of all resources that can be simultaneously acquired. In general, deadlock conditions should be avoided,

but detection and recovery schemes are advisable as well. Further discussion on this topic of avoidance versus detection and recovery can be found in current research [Minoura82] [Reveliotis00].

6.4 Critical Sections to Protect Shared Resources

Shared memory is often used in embedded systems to share data between two services. The alternative is to pass messages between services, but often even messages are passed by synchronizing access to a shared buffer. Different choices for service-to-service communication will be examined more closely in Chapter 8, “Embedded System Components.” When shared memory is used, because real-time systems allow for event-driven preemption of services by higher-priority service releases at any time, shared resources, such as shared memory, must be protected to ensure mutually exclusive access. So, if one service is updating a shared memory location (writing), it must fully complete the update before another service is allowed to preempt the writer and read the same location, as described already in Chapter 4. If this *mutex* (mutually exclusive access) to the update/read data is not enforced, then the reader might read a partially updated message. If the code for each service that either updates or reads the shared data is surrounded with a `semTake()` and `semGive()` (e.g., in VxWorks), then the update and read will be uninterrupted despite the preemptive nature of the RTOS scheduling. The first caller to `semTake()` will enter the critical update section, but the second caller will be blocked and not allowed to enter the partially updated data, causing the original service in the critical section to always fully update or read the data. When the current user of the critical section calls `semGive()` upon leaving the critical section, the service blocked on the `semTake()` is then allowed to continue safely into the critical section. The need and use of semaphores to protect such shared resources are a well-understood concept in multithreaded operating systems.

6.5 Priority Inversion

Priority inversion is simply defined as any time that a high-priority service has to wait while a lower-priority service runs—this can occur in any blocking scenario. We’re most concerned about unbounded priority inversion. If the inversion is bounded, then this can be lumped into the response latency and accounted for so that the RM analysis is still possible. The use

of any *mutex* (mutual exclusion) semaphore can cause a temporary inversion while a higher-priority service is blocked to allow a lower-priority service to complete a shared memory read or update in its entirety. As long as the lower-priority service executes for a critical section WCET, the inversion is known to last no longer than the lower-priority service's WCET for the critical section.

What causes unbounded priority inversion? Three conditions are necessary for unbounded inversion:

- Three or more services with unique priority in the system—High (**H**), Medium (**M**), Low (**L**) priority sets of services.
- At least two services of different priority share a resource with mutex protection—one or more high **H** and one or more low **L** involved.
- One or more services not involved in the mutex has priority **M** between the two (**H**, **L**) involved in the mutex.

It is important to note that the scheduler must be priority-preemptive run-to-completion. Unbounded inversion is not a problem with fair schedulers that use time slices to ensure that all services make continual progress through periodic timer-based preemption, thus not allowing the **M** interfering task to interfere indefinitely—**L** will complete, albeit slowed down by **M**. As such, fair schedulers may either not address the issue at all or, to prevent increase latency, (bounded, but lengthy delay of **L** by **M**), incorporate simple solutions, such as highest-locker protocol, which we describe in the next section on solutions to lengthy or unbounded inversion. Linux, Solaris, and Java typically provide a highest-locker option for mutex critical sections to hurry up **L** by elevating **L**'s priority to the ceiling priority when an inversion is detected.

To fully describe the problem better before more discussion on solutions, the reader should understand that a member of the **H** priority service set catches an **L** priority service in the critical section and is blocked on the `semTake(semid)`. While the **L** priority service executes in the critical section, one or more **M** priority services interfere with the **L** priority service's progress for an indefinite amount of time—the **H** priority service must continue to wait not only for the **L** priority service to finish the critical section, but for the duration of all interference to the **L** priority service. How long will this interference go on? This would be hard to put an upper bound on—clearly it could be longer than the deadline for the **H** priority service.

Figure 6.2 depicts a shared memory usage scenario for a spacecraft system that has two services using or calculating navigational data providing the vehicle's position and attitude in inertial space—one service is a low-priority thread of execution that periodically points an instrument based upon position and attitude at a target planet to look for a landing site. A second service, running a high priority, is using basic navigational sensor readings and computed trajectory information to update the best estimate of the current navigational state. In Figure 6.2 it can be seen that a set of **M** priority services $\{\mathbf{M}\}$ that are unrelated to the shared memory data critical section can cause **H** to block for as long as **M** services continue to preempt the **L** service stuck in the critical section.

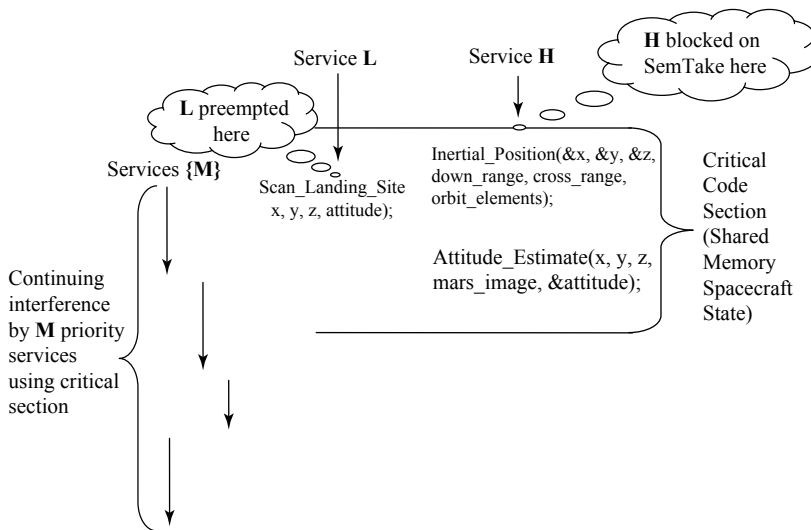


FIGURE 6.2 Unbounded Priority Inversion Scenario

6.5.1 Unbounded Priority Inversion Solutions

One of the first solutions to unbounded priority inversion is to use task or interrupt locking (`VxWorks intLock()` and `intUnlock()` or `task Lock ()` and `task Unlock ()`) to prevent preemptions in critical sections completely, which operates in the same way as a priority ceiling protocol. Priority inheritance was introduced as a more optimal method that limits the amplification of priority in a critical section only to the level required to bound inversions. By comparison, interrupt locking and priority ceiling essentially disable all preemption for the duration of the critical section, but very effectively bound the inversion. To describe better what is meant by priority

inheritance, you must understand the basic strategy to avoid indefinite interference while a low-priority task is in a critical section. Basically, the **H** priority service gives its priority temporarily to the **L** priority service so that it will not be preempted by the **M** priority services while finishing up the critical section—normally the **L** priority service restores the priority loaned to it as soon as it leaves the critical section. The priority of the **L** service is temporarily amplified to the **H** priority to prevent the unbounded inversion. One downside to priority inheritance is that it can chain. When **H** blocks, it is possible that shortly after **H** loans its priority to **L**, another **H**+*n* priority service will block on the same semaphore, therefore requiring a new inheritance of **H**+*n* by **L** so that **H**+*n* is not blocked by services of priority **H**+1 to **H**+*n*−1 interfering with **L**, which has priority **H** [Sha90]. The chaining is complex, so it would be easier to simply give **L** a priority that is so high that chaining is not necessary.

The idea to set a ceiling priority once and for all rather than “loaning” priorities in a dynamic chain became known as the priority ceiling emulation protocol (also known as highest-locker, although there may be detailed differences in how the ceiling is specified and whether it can be changed at runtime or only with recompilation). The ideal priority ceiling protocol would simply set the priority to the highest for all critical section users, assuming this can be determined by the operating system and is more exacting than emulation, where the programmer provides a best guess and was first described in terms of Ada language server and client tasks [Goodenough88]. The Java real-time extensions include a feature that is called priority ceiling emulation protocol [Java RT]. The ceiling priority in priority ceiling emulation protocol is initialized at the time critical sections are constructed (coded) and can’t be changed after that in Java. With priority ceiling emulation protocol, when the inversion occurs, **L** is temporarily loaned the pre-configured ceiling priority, ensuring that it completes the critical section with one priority amplification. However, the ceiling protocol is not ideal because it may either over-amplify the priority of **L** higher than it really needs to be, or through programming error, not amplify it enough to limit the inversion. Over-amplification could cause other problems, such as significant interference to high-frequency, high-priority services. Most real-time systems have gravitated to priority ceiling emulation protocol (highest-locker) for simplicity and with trust that the programmer constructing the critical section will know well the priorities of the users of the critical section. The advantage of the more complex priority inheritance is that the

ceiling does not need to be adjusted when code changes are made or priorities adjusted (it would be easy to forget a ceiling adjustment).

In highest-locker, virtually identical to priority ceiling emulation, the priority of **L** is amplified to the highest priority of all those services that can potentially request access to the critical section based on programmer understanding (and in that sense, assuming the programmer knows the correct ceiling priority to choose while constructing the critical section, highest-locker and priority ceiling emulation protocol are identical). The highest-locker protocol has been used in Unix systems for time-sensitive digital media applications, such as streaming, encoding, and decoding [Vahalia96]. The only downside to highest-locker is that it requires the programmer to indicate to the operating system what the highest-locker priority should be, just like the Java priority ceiling emulation protocol. Any mistake in specification of the highest-locker priority may cause unbounded inversion to persist. In theory, the programmer should know very well what services can enter a given critical section and therefore also know the correct highest-locker priority.

The problem of priority inversion became famous with the Mars Pathfinder spacecraft. The Pathfinder spacecraft was on final approach to Mars and would need to complete a critical engine burn to capture into a Martian orbit within a few days after a cruise trajectory lasting many months. The mission engineers readied the craft by enabling new services. Services such as meteorological processing from instruments were designed to help determine the insertion orbit around Mars because one of the objectives of the mission was to land the Sojourner rover on the surface in a location free of dangerous dust storms. When the new services were activated during this critical final approach, the Pathfinder began to reboot when one of the highest-priority services failed to service the hardware watchdog timer. Discussed in more detail in Chapter 14, “High Availability and Reliability Design,” watchdog timers are used to ensure that software continues to execute sanely. If the countdown watchdog timer is not reset periodically by the highest-priority periodic service, the system is wired to reset and reboot to recover from software failures, such as deadlock or livelock. Furthermore, most systems will reboot in an attempt to recover several times, and if this repeats more than three times, the system will safe itself, expecting operator intervention to fix the recurring software problem.

It was not immediately evident to the mission engineers why Pathfinder was rebooting, other than it was caused by failure to reset the watchdog

timer. Based upon phenomena studied in this chapter alone, reasons for a watchdog timeout could include the following:

- Deadlock or livelock preventing the watchdog reset service from executing
- Loss of software sanity due to programming errors, such as a bad pointer, an improperly handled processor exception, such as divide by zero, or a bus error
- Overload due to miscalculation of WCETs for the final approach service set
- A hardware malfunction of the watchdog timer or associated circuitry
- A multi-bit error in memory due to space radiation causing a bit upset

Most often, the reason for reset is stored in a nonvolatile memory that is persistent through a watchdog reset so that exceptions due to programming errors, memory bit upsets, and hardware malfunctions would be apparent as the reason for reset and/or through anomalies in system health and status telemetry.

After analysis on the ground using a test-bed with identical hardware and data playback along with analysis of code, it was determined that Pathfinder might be suffering from priority inversion. Ideally, a theory like this would be validated first on the ground in the test-bed by recreating conditions to verify that the suspected bug could cause the observed behavior. Priority inversion was suspected because a message-passing method in the VxWorks RTOS used by firmware developers was found to ultimately use shared memory with an option bit for priority, FCFS (First Come First Served), or inversion safe policy for the critical section protecting the shared memory section. In the end, the theory proved correct, and the mission was saved and became a huge success. This story has helped underscore the importance of understanding multiresource interactions in embedded systems as well as design for field debugging. Prior to the Pathfinder incident, the problem of priority inversion was mostly viewed as an esoteric possibility rather than a likely failure scenario. The unbounded inversion on Pathfinder resulted from shared data used by **H**, **M**, and **L** priority services that were made active by mission controllers during final approach.

6.6 Power Management and Processor Clock Modulation

Power and layout considerations for embedded hardware often drive real-time embedded systems to designs with less memory and lower-speed processor clocks. The power consumed by an embedded processor is determined by switching power, short-circuit current, and current leakage within the logic circuit design. The power equations summarizing and used to model the power used by an ASIC (Application Specific Integrated Circuit) design are

$$P_{\text{average}} = P_{\text{switching}} + P_{\text{short_circuit}} + P_{\text{leakage}}$$

$P_{\text{switching}} = (S_{\text{probability}})(C_L)(V_{\text{supply}})^2(f_{\text{clk}})$ — due to capacitor charge/discharge for switching

$P_{\text{short_circuit}} = t(S_{\text{probability}})(V_{\text{supply}})(I_{\text{short}})$ — due to current flow when gates switch

Where the terms in the foregoing equations are defined as follows:

1. P_{leakage} is the power loss based upon threshold voltage.
2. $S_{\text{probability}}$ is the probability that gates will switch, or a fraction of gate switches on average.
3. C_L is load capacitance.
4. I_{short} is short-circuit current.
5. f_{clk} is the CPU clock frequency.

The easiest parameters to control are the V_{supply} and the processor clock frequency in order to reduce power consumption.

Furthermore, the more power put in, the more heat generated. Often real-time embedded systems must operate with high reliability and at low cost so that active cooling is not practical. Consider an embedded satellite control system—a fan is not even feasible for cooling because the electronics will operate in a vacuum. Often passive thermal conduction and radiation are used to control temperatures for embedded systems. More recently, most embedded systems have been designed with processor clock modulation so that V_{supply} can be reduced along with CPU clock rate under the control of firmware when it's entering less busy modes of operation or when the system is overheating.

Summary

Real-time embedded systems are most often designed according to hardware power, mass, layout, and cost constraints, which in turn define the processor, IO, and memory resources available for use by firmware and software. Analysis of a single resource, such as CPU, memory, or IO alone, is well understood for real-time embedded systems—the subject of Chapters 3, 4, and 5. The analysis of multiresource usage, such as CPU and memory, has more recently been examined. The case of multiple services sharing memory and CPU leads to the understanding that deadlock, livelock, and priority inversion could interfere with a service's capability to execute and complete prior to a deadline—even when there is more than sufficient CPU resource. The complexity of multiple resource usage by a set of services can lead to scenarios where a system otherwise determined to be hard real-time safe can still fail.

Exercises

1. Implement the RM LUB (Least Upper Bound) feasibility test presented in an Excel spreadsheet. Verify CPU scheduling feasibility by hand-drawing timing diagrams and using the spreadsheet. Example: $T_1=3$, $T_2=5$, $T_3=15$, $C_1=1$, $C_2=2$, $C_3=3$. What is the LCM of the periods and total utility? Does the example work? Does it pass the RM LUB feasibility test? Why or why not?
2. Given that EDF can provide 100% CPU utilization and provides a deadline guarantee, why isn't EDF always used instead of fixed-priority RMA?
3. Using the same example from #1 ($T_1=3$, $T_2=5$, $T_3=15$, $C_1=1$, $C_2=2$, $C_3=3$), does the system work with the EDF policy? How about LLF? Do EDF and LLF result in the same or a different scheduling?
4. Examine the example code `deadlock.c` for Linux from the DVD that demonstrates deadlock and fix it.
5. Examine the example code `prio_invert.c` from the DVD that demonstrates priority inversion, and use WindView to show that without priority inheritance protocol the inversion occurs, and that with it the problem is corrected.

Chapter References

- [Goodenough88] John Goodenough and Lui Sha, “*The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks*,” CMU/SEI-88-SR-4 Special Report, 1988.
- [Java RT] <http://www.rtsj.org/specjavadoc/javax/realtime/PriorityCeilingEmulation.html>.
- [Minoura82] Toshimi Minoura, “Deadlock Avoidance Revisited,” *Journal of the ACM*, Vol. 29, No. 4 (October 1982): pp. 1032–1048.
- [Reveliotis00] S. A. Reveliotis, “*An Analytical Investigation of the Deadlock Avoidance vs. Detection and Recovery Problem in Buffer-Space Allocation of Flexibly Automated Production Systems*,” Technical Report, Georgia Tech, 2000.
- [Sha90] Lui Sha, Rangunthan Rajkumar, and John P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Transaction on Computers*, Vol. 39, No. 9 pp. 1175–1185, (September 1990).
- [Vahalia96] Uresh Vahalia, *Unix Internals: The New Frontiers*, Prentice-Hall, Upper Saddle River New Jersey, 1996.

SOFT REAL-TIME SERVICES

In this chapter

- Introduction
- Missed Deadlines
- Quality of Service
- Alternatives to Rate-Monotonic Policy
- Mixed Hard and Soft Real-Time Services

7.1 Introduction

Soft real time is a simple concept, defined by the utility curve presented in Chapter 2, “System Resources.” The complexity of soft real-time systems arises from how to handle resource overload scenarios. By definition, soft real-time systems are not designed to guarantee service in worst-case usage scenarios. So, for example, back-to-back cache misses causing a service execution efficiency to be much lower than expected might cause that service’s deadline or another lower-priority service’s deadline to be overrun. How long should any service be allowed to run past its deadline, if at all? How will the quality of the services be impacted by an overrun or by a recovery method that might terminate the release of an overrunning service? This chapter provides some guidance on how to handle these soft real-time scenarios and, in addition, explains why soft real-time methods can work well for some services sets.

7.2 Missed Deadlines

Missed deadlines can be handled in a number of ways:

- Termination of the overrunning service as soon as the deadline is passed
- Allowing an overrunning service to continue running past a deadline for a limited duration
- Allowing an overrunning service to run past a deadline indefinitely

Terminating the overrunning scenario as soon as the deadline is passed is known as a *service dropout*. The outputs from the service are not produced, and the computations completed up to that point are abandoned. For example, an MPEG (Motion Picture Expert's Group) decoder service would discontinue the decoding and not produce an output frame for display. The observable result of this handling is a decrease in quality of service. A frame dropout results in a potentially displeasing video quality for a user. If dropouts rarely occur back to back and rarely in general, this might be acceptable quality. If soft real-time service overruns are handled with termination and dropouts, the expected frequency of dropouts and reduction in quality of service should be computed.

The advantage of service dropouts is that the impact of the overrunning service is isolated to that service alone—other higher-priority and lower-priority services will not be adversely impacted as long as the overrun can be quickly detected and handled. For an RM policy, the failure mode is limited to the single overrunning service (refer to Figure 3.15 of Chapter 3). Quick overrun detection and handling always result in some residual interference to other services and could cause additional services to also miss their deadlines—a cascading failure. If some resource margin is maintained for dropout handling, this impact can still be isolated to the single overrunning service.

Allowing a service to continue an overrun beyond the specified deadline is risky because the overrun causes unaccounted-for interference to other services. Allowing such a service to overrun indefinitely could cause all other services to fail of lesser priority in an RM policy system. For this reason, it's most often advisable to handle overruns with termination and limited service dropouts. Deterministic behavior in a failure scenario is the next best thing compared to deterministic behavior that guarantees success. Dynamic-priority services are more susceptible to cascading failures (refer

to Figure 3.16 in Chapter 3) and therefore also more risky as far as impact of an overrun and time for the system to recover. Cascading failures make the computation of dropout impact on quality of service harder to estimate.

7.3 Quality of Service

Quality of service (QoS) for a real-time system can be quantified based upon the frequency that services produce an incorrect result or a late result compared to how often they function correctly. A real-time system is said to be correct only if it produces correct results on time. In Chapter 14, “High Availability and Reliability Design,” the classic design methods and definitions of availability and reliability will be examined. The QoS concept is certainly related. The traditional definition of availability of a service is defined as follows:

- $\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$
- MTTF = Mean Time to Failure (How long the system is expected to run without failure)
- $\text{MTBF} = \text{MTTF} + \text{MTTR}$ (Mean Time Between Failures, including recovery from earlier failures)
- MTTR = Mean Time to Recovery

The MTBF may be confused with MTTF on occasion and for small MTTR, MTBF and MTTF differ only by MTTR. For many systems the MTTR is seconds or at most minutes and the MTTF is often hundreds of thousands or millions of hours, so in practice, $\text{MTBF} \approx \text{MTTF}$ as long as $\text{MTTR} \ll \text{MTBF}$. This is depicted as follows:

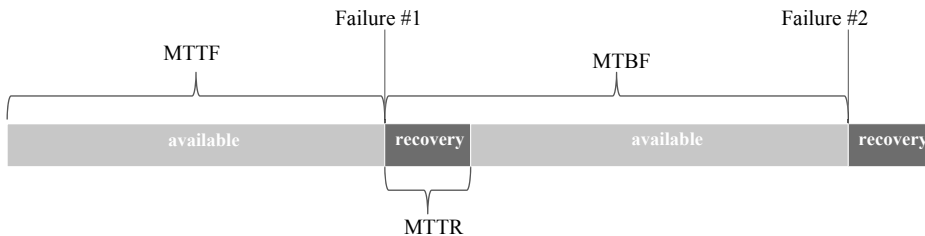


FIGURE 7.1 Difference between MTBF and MTTF

If a service has higher availability, does it in fact also have higher quality? From the viewpoint of service dropouts, measured in terms of frames delivered—for example, for a video decoder—then higher availability does mean fewer service dropouts over a given period of time. This formulation for QoS can be expressed as:

- $\text{QoS} = 1 - \text{Drop_outs} / \text{Deliveries}$
- Where QoS = 1 is full quality and 0 is no quality of service

So, in this example, availability and QoS are directly related to the degree that number of dropouts will be directly proportional to availability. However, delivering decoded frames for display is an isochronal process (defined in Chapter 2). Presenting frames for display too early causes frame jitter and lower QoS with no service dropouts and 100% availability. Systems providing isochronal services and output most often use a DM (Deadline-Monotonic) policy and buffer and hold outputs that are completed prior to the isochronal deadline to avoid jitter. The measure of QoS is application-specific. For example, isochronal networks often define QoS as the degree to which packets transported approximate a constant bit-rate dedicated circuit. To understand QoS well, the specific application domain for a service must be well understood. In the remaining sections of this chapter, soft real-time methods that can be used to establish QoS for an application are reviewed.

7.4 Alternatives to Rate-Monotonic Policy

The RM policy can lead to pessimistic maintenance of high-resource margins for sets of services that lack harmony (described in Chapter 3, “Processing”). Furthermore, RM policy makes restrictive assumptions, such as $T=D$. Because QoS is a bit harder to nail down categorically, designers of soft real-time systems should consider alternatives to RM policy that might better fit their application-specific measures of QoS. For example, in Figure 7.2, the RM policy would cause a deadline overrun and a service dropout, decreasing QoS, but it’s evident that EDF or LLF dynamic-priority policies will result in higher QoS because both avoid the overrun and subsequent service dropout.

The EDF and LLF policies are not always better from a QoS viewpoint. Figure 7.3 shows how EDF, LLF, and RM all perform equally well

Example 2	T1	2	C1	1	U1	0.5	LCM =	910							
	T2	5	C2	1	U2	0.2									
	T3	7	C3	1	U3	0.142857									
	T4	13	C4	2	U4	0.153846	Utot =	0.996703							
RM Schedule															
S1														???????	
S2															
S3															
S4															FAILURE
EDF Schedule															
S1															
S2															
S3															
S4															
TTD															
S1	2	X	2	X	2	X	2	X	2	X	2	X	2	X	X
S2	5	4	X	X	X	5	X	X	X	X	5	4	3	2	2
S3	7	6	5	4	X	X	X	7	6	5	4	3	X	X	X
S4	13	12	11	10	9	8	7	6	5	4	X	X	X	X	X
LLF Schedule															
S1															
S2															
S3															
S4															
Laxity															
S1	1	X	1	X	1	X	1	X	1	X	1	X	1	X	X
S2	4	3	X	X	X	4	X	X	X	X	4	3	2	1	1
S3	6	5	4	3	X	X	X	6	5	4	X	X	X	X	X
S4	11	10	9	8	7	6	5	4	3	2	1	X	X	X	X

FIGURE 7.2 Highly Loaded System—RM Deadline Overrun

Example 3	T1	3	C1	1	U1	0.33	LCM =	15							
	T2	5	C2	2	U2	0.4									
	T3	15	C3	3	U3	0.2	Utot =	0.93							
RM Schedule															
S1															
S2															
S3															
EDF Schedule															
S1															
S2															
S3															
TTD															
S1	3	X	X	3	X	X	3	X	X	3	X	X	3	X	X
S2	5	4	3	X	X	5	4	3	X	X	5	4	X	X	X
S3	15	14	13	12	11	10	9	8	7	6	5	4	3	2	X
LLF Schedule															
S1															
S2															
S3															
Laxity															
S1	2	X	X	2	X	X	2	X	X	2	X	X	2	X	X
S2	3	2	2	X	X	3	3	2	X	X	3	3	X	X	X
S3	12	11	10	9	8	7	6	5	5	4	3	3	2	X	X

FIGURE 7.3 Three Policies and Three Common Schedules

for a given service scenario. From a practical viewpoint, the decision to be made on scheduling policy should be a balance between the impact on QoS by the more adaptive EDF and LLF policies compared to the more predictable failure modes and deterministic behavior of RM in an overload situation. This may be difficult to compute and might be best evaluated by trying all three policies with extensive testing.

In cases where EDF, LLF, and RM perform equally well in a non-overload scenario, RM might be a better choice because the impact of a failure is simpler to contain; that is, there is less likelihood of cascading service dropouts given upper bounds on overrun detection and handling.

As shown in Figure 7.4, it's well worth noting that systems designed to have harmonic service request periods do equally well with EDF, LLF, and RM. Designing systems to be harmonic can greatly simplify real-time scheduling.

Example 4	T1	2	C1	1	U1	0.5	LCM =	16									
	T2	4	C2	1	U2	0.25											
	T3	16	C3	4	U3	0.25	Utot =	1									
RM Schedule																	
S1																	
S2																	
S3																	
EDF Schedule																	
S1																	
S2																	
S3																	
TTD																	
S1	2	X	2	X	2	X	2	X	2	X	2	X	2	X	2	X	X
S2	4	3	X	X	4	3	X	X	4	3	X	X	4	3	X	X	X
S3	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
LLF Schedule																	
S1																	
S2																	
S3																	
Laxity																	
S1	1	X	1	X	1	X	1	X	1	X	1	X	1	X	1	X	X
S2	3	2	X	X	3	2	X	X	3	2	X	X	3	2	X	X	X
S3	12	11	10	9	9	8	7	6	6	5	4	3	3	2	1	0	

FIGURE 7.4 Full Utility from a Harmonic Schedule

Figure 7.5 shows yet another example of a harmonic schedule where policy is inconsequential.

For isochronal services, DM policy can have an advantage by relaxing $T=D$. This allows for analysis of systems where services can complete early to buffer and hold outputs to reduce presentation jitter and thereby increase QoS. Figure 7.6 shows a scenario where the DM policy succeeds when the RM would fail due to requirements where D can be greater or less than the release period T .

Example 5	T1	2	C1	1	U1	0.5	LCM =	10		
	T2	5	C2	2	U2	0.4				
	T3	10	C3	1	U3	0.1	Utot =	1		
RM Schedule										
S1										
S2										
S3										
EDF Schedule										
S1										
S2										
S3										
TTD										
S1	2	X	2	X	2	X	2	X	2	X
S2	5	4	3	2	X	5	4	3	X	X
S3	10	9	8	7	6	5	4	3	2	1
LLF Schedule										
S1										
S2										
S3										
Laxity										
S1	1	X	1	X	1	X	1	X	1	X
S2	3	2	2	1	X	3	3	2	X	X
S3	9	8	7	6	5	4	3	2	1	0

FIGURE 7.5 A Harmonic Service Set

Example 6	T1	2	C1	1	U1	0.5	LCM =	70	For DM	D1	2				
	T2	5	C2	1	U2	0.2				D2	3	EARLIER			
	T3	7	C3	1	U3	0.142857				D3	7				
	T4	13	C4	2	U4	0.153846	Utot =	0.996703		D4	15	LATER			
					RM D2,1					RM D2,2			RM D4,1	RM D2,3	
RM Schedule													????????		
S1															
S2															
S3															
S4													FAILURE		
DM Schedule															
S1															
S2															
S3															
S4															
					DM D2,1					DM D2,2			DM D2,3	DM D4,1	
													DM R4,2	DM R2,4	
													OVERLAP	OVERLAP	

FIGURE 7.6 Deadline-Monotonic Can Work Where RM Fails

7.5 Mixed Hard and Soft Real-Time Services

Many systems include services that are hard real-time, soft real-time, and best-effort. For example, a computer vision system on an assembly line may have hard real-time services, where missing a deadline would cause shutdown of the process being controlled. Likewise, operators may want to occasionally monitor what the computer vision systems “sees.” The video for monitoring should have good QoS so that a human monitor can assess how well the system is working, whether lighting is sufficient, and whether frame rates appear reasonable. Finally, in the same system, operators may occasionally want to dump maintenance data and have no real requirements

for how fast this is done—it can be done in the background whenever spare cycles are available.

The mixing of hard, soft, and best effort can be done by admitting the services into multiple periodic servers for each. The hard real-time services can be scheduled within a time period (epoch) during which the CPU is dedicated to hard real-time services (all others are preempted). Another approach is to transform the period for all the hard real-time services so that they have priorities that encode their importance. Either way, we ensure that the hard real-time services will preempt all soft services and best-effort services on a deterministic and periodic basis.

Best-effort services can always be handled by simply scheduling all these services at the lowest priority and at an equal priority among them. At lowest priority, best-effort services become slack time stealers that execute only when no real-time (hard or soft) services are requesting processor resources.

Summary

Soft real-time services assume that some service releases will fail to meet deadline requirements. Careful consideration should be given to how service deadline overruns will be handled and how this will impact QoS.

Exercises



1. Review `posix_clock.c` and `posix_rt_timers.c` contained on the DVD. Write a brief paragraph describing how these two modules work and what features of the POSIX 1003.1b real-time extensions they demonstrate. Make sure you not only read the code but also build it, load it, and execute it to make sure you understand how both applications work.
2. Build and analyze the DVD `itimer_test.c` module for VxWorks and describe how it works. Specifically what states does the `itimer_test()` task context transition between if it is spawned in a task context using `spitimer_test()`?
3. Build, load, and run the DVD `posix_sw_wd.c` software watchdog monitor code, and describe how it executes, providing evidence for your

description with supporting WindView (now known as System Viewer [WRS06]) traces.

4. Use the Cheddar real-time scheduling analysis tool (found at <http://beru.univ-brest.fr/~singhoff/cheddar/>) and run both simulation and worst-case analysis on the scheduling examples found on the DVD and described in Figures 7.2, 7.3, 7.4, 7.5, and 7.6.



Chapter References

- [WRS99a] *VxWorks Programmer's Guide 5.4, Edition 1*, WRS, Alameda California, 1999.
- [WRS99b] *VxWorks Reference Manual 5.4*, available online as “windman” or hard copy, as Edition 1. WRS, Alameda California, 1999.
- [WRS06] “Wind River Workbench Product Note,” <http://windriver.com/products/product-notes/workbench-product-note.pdf>

DESIGNING REAL-TIME EMBEDDED COMPONENTS

- Chapter 8 Embedded System Components
- Chapter 9 Traditional Hard Real-Time Operating System
- Chapter 10 Open Source Real-Time Operating Systems
- Chapter 11 Integrating Embedded Linux into Real-Time Systems
- Chapter 12 Debugging Components
- Chapter 13 Performance Tuning
- Chapter 14 High Availability and Reliability Design

EMBEDDED SYSTEM COMPONENTS

In this chapter

- Introduction
- Hardware Components
- Firmware Components
- RTOS System Software
- Software Application Components

8.1 Introduction

System design can be approached in a bottom-up or a top-down fashion. The *bottom-up approach* consists of determining the fundamental components that go into a system. The *top-down approach* is a hierarchical breakdown of the system into subsystems and then into components. The top-down approach can be viewed as a concrete breakdown of the system into smaller parts as suggested here, but often an abstract top-down approach is useful where the system is broken down by service and function. This functional top-down approach is described in Chapter 15, “System Life Cycle.” In this chapter, we first examine common components of a real-time embedded system in the concrete sense, going from the overall system down to components. Familiarity of the components can assist the designer in making more optimal system design decisions. The design of a real-time embedded system can be viewed as a hierarchy of subsystems, as depicted in Figure 8.1, showing a real-time stereo-vision tracking system.

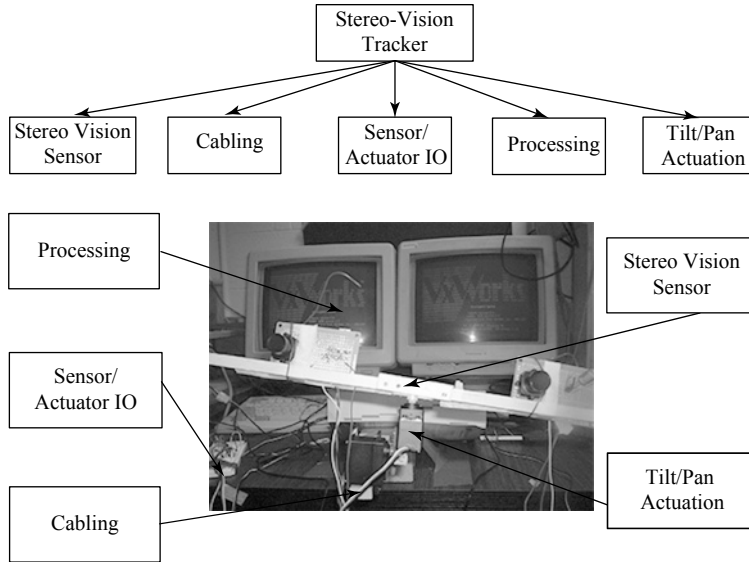


FIGURE 8.1 Subsystems in a Stereo-Vision Tracking System

The stereo-vision tracking system has a simple goal—keep a bright object in the field of view of both cameras even if the object moves and estimate the distance from the camera assembly to the object. A more functional service view of the same stereo-vision tracking system would look much different. This is depicted by the hierarchy for the same system shown in Figure 8.2.

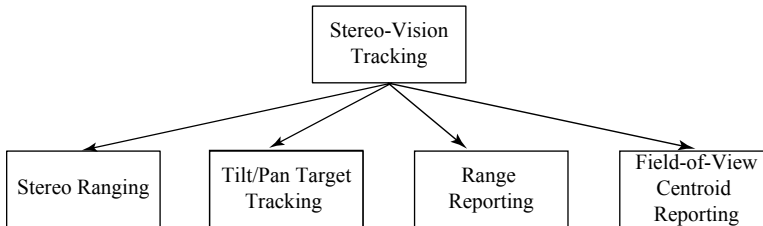


FIGURE 8.2 Services in a Stereo-Vision Tracking System

The stereo-vision tracking system is an example design that is examined in more detail in Chapter 18, “Computer Vision Applications.”

8.2 Hardware Components

The hardware components of a real-time embedded system will include a wide range of components that are mechanical and electrical. For example, in the stereo-vision tracking system we have:

- Structural and mechanical—camera assembly
- Electromechanical actuators—tilt and pan servos
- Electromechanical sensors (transducers)—none, but servo position sensors could be added
- Optical sensors—NTSC cameras
- Cabling—power, NTSC signal, RS232, CAT-5 twisted pairs
- Digital state machines, microcontrollers, and microprocessors—x86 microprocessor, PIC microcontroller
- Analog front-end (sensor) and back-end (actuator) circuits—NTSC, TTL pulse-width modulation
- Networks or bus interfaces—RS232 serial, Ethernet, PCI
- Thermal management—CPU fans

Typically additional test equipment hardware may also require including monitors, development computing environment, oscilloscope, digital multimeter, and a logic analyzer. This, however, is not part of the system, although required to fully implement and verify its proper implementation and operation.

In the following sections, basic hardware components, such as those used in the stereo-vision tracking system, are described.

8.2.1 Sensors

Sensors are devices that respond to physical stimulus (light, heat, pressure, stress/strain, acceleration, magnetism) by transforming the associated energy into electrical energy or by modifying the electrical properties in a circuit. For example, a camera is a sensor that converts photon energy into electrical charge that represents the photon flux for each picture element in an array. A thermistor is a resistor circuit where the resistance of the thermistor changes with temperature, and therefore so does the circuit current at

a given voltage and voltage drop across the load. A sensor assembly may also interface this analog front end to a digital encoding interface. Analog-to-digital converters (ADCs) are used to sample and hold charge, thereby converting the analog circuit current/voltage into a digital value. For example, an 8-bit ADC will encode a sensor circuit operational voltage range into 256 levels. Without encoding, sensors are useful in analog control systems, but for use in digital control systems, encoding is critical. Real-time embedded systems therefore require digital encoding of all sensor inputs, with the exception of subsystems, which are all analog.

The sensor AFE (Analog Front End) involves physics that are very particular to the environmental phenomena being sensed and the method for generating measurable changes in an analog circuit based upon physical stimulus. Many sensors are electromechanical devices where mechanical stimulus, such as stress/strain, the force per unit area and resulting deformation, or motion, causes a change in analog circuit voltage/current. Resistance in many materials is a function of stress, strain, and/or temperature; thus these mechanical properties can be measured using the right material as a resistor in a circuit in the AFE. Motion can be sensed also with a variation of resistance through potentiometers, where resistance is modified by mechanically varying the resistive path in a circuit. A simple example is the use of a multi-turn potentiometer to modify resistance in a circuit with rotation. The sensor couples a physical phenomenon to an electrical one. This coupling may be more erudite, as is the case with an optical encoder that uses periodic interruption of an optical-coupler (LED and photodiode) in a circuit through mechanical mechanisms, such as a filter wheel. In this case the optical-coupler interruptions are counted to estimate rotation.

The sensor always includes the AFE, but may also include the analog-to-digital encoding as well. In the case of an NTSC (National Television Standards Committee) camera, the camera outputs an analog signal that encodes photo-intensity in an image field of view in an analog raster output. The NTSC analog signal can be further encoded from the NTSC signal into a digital image, which is an array of alpha-RGB (Red, Green, Blue) pixels that indicate luminance and chrominance of subareas of the camera's field of view—picture element or pixel alpha-RGB values encoded using an ADC. This is the approach taken in the stereo-vision example. An alternative might employ a CCD (Charge Coupled Device) camera, which provides a more direct encoding of photo-intensity (photon flux) in terms of electrical charge. The range of methods used by sensors to encode the wide

range of physical phenomena and associated energy into electrical energy is too broad to comprehensively discuss in this text. The key concept, however, is that all sensors do convert physical stimulus into electrical outputs that affect an analog circuit, which in turn can be encoded into a digital input using an ADC. The ADC implementation has significant impact on the encoding capability, including the following:

- Sampling frequency
- Sample accuracy
- Input range

The ADC takes an analog input (voltage or current) and converts it into a digital word, most often from 8-bit to 16-bit. The ADC requires a reference voltage, V_{ref} , and normally encodes all inputs into a range of values from zero to: $\frac{V_{\text{ref}}}{(2^n)}$, so that for a 5V reference, a typical 16-bit ADC can

encode the voltage in an AFE into increments representing a change of 0.0763 millivolts, with an input range of 0 to 5V for the AFE signal. The resolution can clearly be increased by reducing the reference voltage V_{ref} ; however, this is at the cost of constraining the input range. This trade-off drives the selection of how many bits the ADC provides in the encoding—to accommodate large input ranges and high resolution, the ADC must have more bits for the encoding. The final question is, how fast can the AFE be sampled? This depends upon the type of ADC:

- Flash—using comparators, one per voltage step, and resistors
- Successive approximation—comparators and counting logic

The flash ADC conversion speed is the sum of the comparator delays and logic delay—typically flash ADCs are the fastest variety. The successive approximation ADC uses comparators to determine first whether the input is greater than half the reference, then whether it's greater than one quarter, and so on until the LSB (Least Significant Bit) comparison is made and the signal level has been approximated successively to the bit accuracy of the ADC—this takes as many clock cycles as the ADC has bits. One more issue with any ADC is how the input signal is sampled—if it changes significantly during the ADC process, then the results will not be accurate, so ADCs must sample and hold the input. The sample and hold time will

add latency and thus reduce the maximum inter-sample frequency. Furthermore, the ADC may automatically sample and provide an interrupt or FIFO input to a state machine or microprocessor, or the ADC may require commands to sample the input and then convert it. For high-rate encoding, such as video, a dedicated hardware state machine typically provides the ADC control and stores encoded data in a FIFO for transfer to a microprocessor via DMA. Methods for transferring encoded data are discussed in more detail in the “Firmware Components” section.

8.2.2 Actuators

Fundamentally, an *actuator* is a transducer that converts electrical energy into some other form, such as sound, motion, heat, or electromagnetism. The simplest form of actuation is switching. The relay provides a mechanism that can be actuated to open or close a switch on command from a digital IO interface. This on/off control does not provide continuous output or simple variation of output amplitude over time. A *servomechanism*, or *servo*, is an actuator that converts electrical energy into mechanical rotation, using a motor and a control interface. Heating elements that are simple resistors can be modulated to provide heat for a system that requires minimum operating temperatures. Likewise, for systems that require active cooling, actuator subsystems can provide cooling using fans, louvers, or some other form of conductive, convective, or radiative cooling. Digital values are decoded into analog signals through an analog back end (ABE) for actuation so that a digitally encoded value drives the voltage in the ABE circuit. This is most often done using either PWM (Pulse-Width Modulation) or a DAC (Digital-to-Analog Converter) so that the amplitude in the ABE can be driven by a stream of digital encoded outputs. With PWM, a periodic digital pulse (e.g., TTL logic level) is driven out with a duty cycle that is proportional to the desired amplitude of the signal at a given point in time. The DAC provides the proportional output automatically based upon the last commanded digital output rather than decoding using a digital duty cycle. Much like ADC sensor interfaces, DAC actuator interfaces should be characterized by the following:

- Type of actuation—on/off or DAC/PWM modulated
- Speed of actuation
- Accuracy of modulation

Most often for accurate high-rate actuators, a DAC is required rather than relays or PWM. For audio output, PCM (Pulse Code Modulation) is used for input sampling and driving an output DAC for duration and at variable output levels.

Actuators can be very unstable and suffer overshoot or failure to settle without careful design and potential feedback from sensors. For example, a scanning mirror can be used to move an optical field of view very accurately by deflecting a pickoff mirror on an optical path through a small angle. An electromechanical mechanism known as a voice-coil flexure can be driven by a DAC so that electromagnetic coils are used to deflect a mirror on a rubber flexure to the left or right; furthermore, the mirror can be restored to a previous position by allowing the flexure to spring back, dampened by the electromagnetic coils. Some of the high-rate feedback control for such an actuator might be implemented as a traditional analog control circuit rather than relying upon the digital real-time embedded system to provide such control.

8.2.3 IO Interfaces

The IO in general to and from a real-time embedded system can be classified first as either analog or digital. In the case of analog IO, as seen in the previous section, an ADC is required to encode analog inputs and a DAC, PWM, or relay interface is required to decode digital outputs when analog IO is interfaced to a real-time embedded system. Many embedded systems may actually be subsystems in a much larger system and therefore may not actually have direct analog IO—instead, many real-time embedded systems have digital IO only or in addition to analog IO. Either way, at some point, all IO becomes digital once encoded or prior to decode. So, prior to the ABE or after the AFE, the embedded system simply sees digital IO. The form of the digital IO, however, can vary significantly and can be characterized as:

- Word-at-a-time IO
- Block IO

Furthermore, the method of interfacing word or block IO can be:

- Memory Mapped IO
- Port IO

In the case of word IO, a simple set of registers defines the interface to the AFE/ABE and provides status, data, and control for encode/decode. A single word is written to an ABE for output to a data register and the output started by setting control bits and output status monitored using the status register. Likewise, for an AFE interface, status can be polled to determine when new encoded data is available and samples commanded through control and monitored via status. The word IO interfaces require significant interaction with the real-time embedded system and are not very efficient, but do provide simple low-rate IO interfaces. These interfaces may require programmed IO where a CPU is involved in each input and output for all phases of the read/write, status monitoring, and control. This is often not desirable for higher-rate interfaces, where even powerful CPUs would spend way too many cycles on programmed IO and not enough on processing to provide services. So, most high-rate interfaces have a block IO interface where a state machine or DMA engine provides command and control of the word encoding/decoding and less frequently interrupts the CPU when significantly large blocks of data have been encoded/decoded—typically 1024, 2048, or 4096 byte blocks.

Processor cores traditionally have provided IO through dedicated pins from the CPU to other devices called IO ports. An alternative is to save on off-chip interface pins by memory mapping IO onto existing address and data lines in/out of the CPU so that IO causes devices to be read or written in the same address space as memory devices. Many processors, in fact, provide both port IO and memory mapped IO, such as the Intel x86. When devices are memory mapped, care must be taken to ensure that the MMU (Memory Management Unit) is aware that particular address ranges are being used for device IO so that output data is not cached, so that writes are fully drained to the device rather than buffered when needed, and so that the address range is allowed to be accessed without exception. Memory locations can be cached, and often it is not necessary for the CPU to wait for writes to be updated in actual memory devices once writes have been queued—for device IO, normally all writes should be fully drained and not cached so that actuation is reliable. If, for example, an output to a DAC was cached for later write-back and this output was driving isochronal speaker output, this would cause an actuation dropout.

Figure 8.3 shows the components of the stereo-vision sensor subsystem for the stereo-vision tracker. This sensor device consists of two NTSC (National Television Standards Committee) cameras and two PCI frame

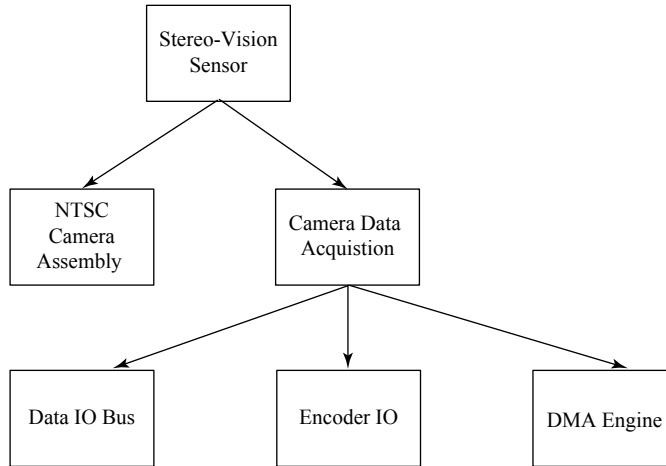


FIGURE 8.3 NTSC Vision Subsystem in Stereo-Vision Tracking System

grabbers, which acquire and encode the NTSC camera output. The data acquisition is composed of an NTSC signal encoding interface, a PCI bus data IO DMA channel, and a programmable DMA engine. The encoding is performed at 30 frames per second for a selection of video-encoding formats, including the maximum resolution of 640x480 32-bit pixels, where each pixel is composed of an 8-bit intensity, alpha, and three 8-bit fields encoding RGB (Red, Green, Blue). The AFE for an NTSC encoder uses a PLL (Phase Lock Loop) to synchronize with the NTSC signal in order to sample and digitize the signal to form a YCrCb (Y=luminance, CrCb=red and blue chrominance). The color NTSC signal format was an enhancement to the original grayscale television signal format. The basic NTSC signal format includes 525 horizontal traces to illuminate a phosphor screen, with 262.5 even scan lines and 262.5 odd with blanking time for signal re-tracing. The intensity of the tracing beam is modulated during the even/odd interlaced line tracing such that each pixel is illuminated for 125 nanoseconds for 427 pixels/line and 10 microseconds of blanking between lines, yielding a scan line time of 63.6 microseconds. The NTSC camera produces a signal conforming to this NTSC standard for direct input into a standard television monitor. The interlacing of horizontal scan lines—that is, tracing odd lines followed by even lines tracing each frame—reduces flicker at the NTSC frame rate of approximately 30 fps (29.97 actual). The AFE PLL synchronizes with the scan line by detecting the NTSC sync and blanking levels and then programs the ADCs to sample the signal for each pixel to

encode YCrCb. The YCrCb data is latched into an internal FIFO memory, and a synchronized DMA engine drains the FIFO with PCI bus transfers from the encoder to host system memory. The YCrCb format for NTSC was chosen so that grayscale televisions can display color NTSC signals by simply using the luminance portion of the signal alone. Most encoders support automatic conversion from YCrCb into alpha-RGB using linear scaling formulae based upon characteristics of human vision:

$$R = Y + Cr$$

$$G = Y - 0.51 \times Cr - 0.186 \times Cb$$

$$B = Y + Cb$$

The relationship between the YCrCb and RGB signals are:

$$Y = 0.3R + 0.59G + 0.11B$$

$$Cr = R - Y = R - (0.3R + 0.59G + 0.11B)$$

$$Cb = B - Y = B - (0.3R + 0.59G + 0.11B)$$

The DMA engine is a simple processor with an RISC instruction set that provides control over the encoding as well as the PCI DMA transfer and generation of host interrupts. So, for example, microcode can be written to encode the 525 NTSC input even and odd lines with 427 pixels, each into a range of formats (e.g., 320x240 alpha-RGB or 80x60 grayscale) based upon the ADC sampling rates with instructions to transfer the encoded data and to generate an interrupt at the completion of each frame encoded and transferred over PCI. The 320x240 alpha-RGB frames (307,200 bytes/frame) are transferred by the DMA engine using multiple PCI bus bursts, typically 512 bytes to 4K each. This is typical of a high-rate block transfer IO interface for a high data rate sensor. In the case of this example, two encoders are bursting video data on the PCI bus simultaneously to two different DMA buffers in two different host memory address ranges.

The stereo-vision tracker also incorporates a low-rate actuation IO interface to enable the system to tilt and pan the stereo-vision sensor (cameras and baseline mount) to follow a bright target that may be moving in order to keep the target in both camera fields of view. Figure 8.4 describes the components making up this low-rate actuation subsystem.

The tilt/pan actuation subsystem uses two servos to provide the tilt and pan rotational degrees of freedom. The servos are commanded with a TTL PWM signal (Servo PWM IO) generated by a microcontroller (Servo Controller). The specific signal generated, and thus position of the servo, can

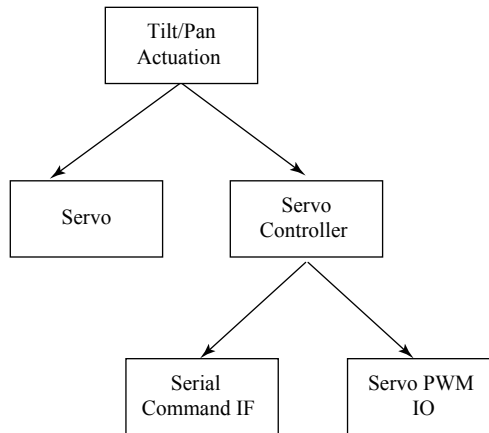


FIGURE 8.4 Tilt/Pan Servo Subsystem in Stereo-Vision Tracking System

be commanded by a microprocessor through a multi-drop serial interface (Serial Command IF). The serial command interface is simple and allows the microprocessor to write out command data only 1 byte at a time. This is a typical low-rate interface.

8.2.4 Processor Complex or SoC

Almost all modern real-time embedded systems include a general-purpose CPU to process firmware/software to provide updateable and flexible services by processing and linking sensor inputs to actuator outputs. If the services that a real-time embedded system must provide are so well known that they can be fully committed to a hardware state machine, then perhaps a processor complex (or set of interconnected CPUs or CPU cores) is not needed. Most often services are expected to change over time or are not well enough specified initially or way too complex to consider hardware-only implementations. The processor complex may be composed of the following:

- A single CPU with port IO and bus interface MMIO
- Multiple CPUs on an internal bus with port/MMIO
- Multiple CPUs with an interconnection network and port/MMIO
- An SoC (System on a Chip) with multiple CPU cores interconnected on-chip with memory, IO, flash, and any number of peripherals making it a single-chip solution

In the case of our working example, the stereo-vision system, a main x86 CPU provides an image processing platform to compute the centroid of the target object as seen by the left and right cameras and encoded using the PCI-bus NTSC encoder subsystem. The servo control is achieved using the Servo Controller, a Microchip PIC that commands multiple servos to tilt/pan the camera assembly using TTL logic-level PWM based upon a serial byte stream command to the controller. Figure 8.5 shows the subsystems (Servo Control and Image Processing) that compose the overall stereo-vision system processing to provide the tracking and ranging services. The Servo Control subsystem uses a digital control law based upon calculated centroid inputs to tilt/pan the stereo-vision sensors in real time to keep the target in the field of view and produces a series of servo commands as output. The Image Processing subsystem uses alpha-RGB video frames at a maximum rate of 30 fps to compute the centroid of the target as seen by each camera and the range to the target based upon a triangulation calculation.

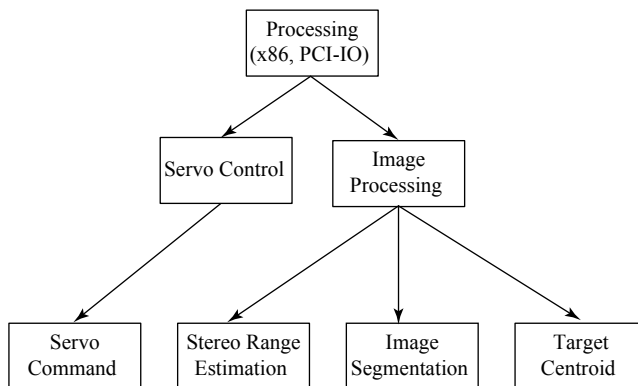


FIGURE 8.5 Processing Subsystem in Stereo-Vision Tracking System

8.2.5 Processor and IO Interconnection

For multi-CPU real-time embedded systems, an interconnection network is required to enable IO and processing to be distributed. The interconnection network can be

- Simple bus or back-plane (e.g., PCI or VME)
- On-chip local bus with bus interface unit to back-plane IO bus
- A crossbar on-chip interconnection between CPUs
- An off-board network—for example, firewire, USB, Ethernet

Figure 8.6 shows taxonomy for interconnection strategies that can be used to integrate CPUs and IO interfaces in an embedded system.

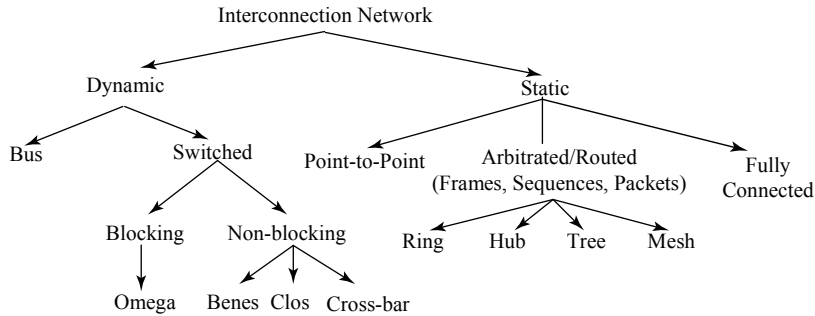


FIGURE 8.6 Taxonomy of Processor-IO Interconnection Strategies

Most embedded systems are integrated with a scalable bus architecture, point-to-point serial links, or networks.

8.2.6 Bus Interconnection

Many different bus architectures have been used and are being used for embedded systems. To better understand integration of processing and IO with a bus interconnection, this chapter will examine the VME (Versa Module Extension) bus and the PCI (Peripheral Component Interconnect) bus. The VME bus has historically been a popular and simple bus architecture used with embedded systems, and, by contrast, PCI is an emergent bus architecture that offers traditional parallel bus integration as well as high-speed serial interconnection with PCI Express.

The PCI bus, introduced as a replacement for the ISA (Industry Standard Architecture) bus prevalent in the desktop PC domain, has evolved and become a popular IO to processor complex integration method in embedded systems. A goal of the first PCI standard, 2.1, was to provide a bus where IO adapters could be interfaced to processor complexes with plug-and-play integration. Plug and play provides a standard for PCI controllers (masters) to probe the bus and find devices after they have been added without any modification to the master interface. The PCI bus was designed to integrate with the legacy ISA bus through an interface called the South Bus. The main PCI controller interface was called the North Bus. At the time that PCI was first introduced, many embedded systems were integrated using the VME bus (Versa Local Bus Module Expansion). Building a real-time embedded system based upon bus integration allows system designers to decompose

the system into subsystems with interface and processor boards that can be designed, built, and tested as units and later integrated on a standard interconnection. Both VME and PCI provide this modularity compared to custom back-plane or onboard integration. The bus integration also provides a fast signal interface compared to packet or frame transmission networks (this has started to change recently, as we'll see later). Table 8.1 briefly summarizes both VME-32 and PCI 2.x, comparing features.

TABLE 8.1 Comparison of PCI and VME Buses

Feature	VME bus	PCI 2.x bus
Bus transfers	Asynchronous 20 MHz	Synch clock 33/66 MHz
Target addressing	32, 24, or 16 bit address (A32, A24, A16)	Multiplexed 32/64 bit address/data bus
Data transfer	32, 24, or 16 bit separate data bus	Multiplexed with 32 bit address bus
Data transfer types	Word or block transfer limited to a specific block size (e.g., 512 bytes)	Burst transfer always with min and max length
Device interrupt mechanism	Daisy-chained priority interrupts	4 shared interrupt lines: A-D routed to programmable interrupt controller
Interrupt vectoring	Interrupt data cycle following interrupt level	Map A-D onto processor vector—e.g., onto IRQ 0...15 on x86 with direct IRQ to vector mapping
Bus access arbitration for multiple initiators (masters)	No arbitration, firmware or custom controller must ensure mutex access	Built-in hidden arbitration
Device addressing	Custom-designed MMIO	Plug 'n' play configuration space allows firmware to set an MMIO or IO base address at run time
Expansion and form factors	Custom bus integration on 6U boards with 3U/6U D-shell form factor	No custom expansion, but many standard form factors: compact PCI, PC/104+, PMC, and standard PC 2.x
Faster options	VME-64+	PCI-X 1.0a, 2.0, and PCI Express

The features of PCI that have made it successful and a popular integration bus are burst transfer (most IO has become high-rate and is most efficient with block transfer), built-in arbitration, and plug-and-play configuration. The two most commonly used form factors for real-time embedded systems are Compact PCI (D-shell back-plane connector) and PC/104+ (a stackable small board form factor). The PCI bus has also been very popular as a chipset interconnection on single-board embedded systems. One reason for the popularity of PCI as a chip interconnection on board is the definition of PCI bridges, which allow for bus-to-bus PCI integration. The PCI 2.x standard provides 32-bit 33-MHz bus cycles and up to 64-bit at 66 MHz for updated 2.x, yielding bandwidth of 128 million bytes per second to 512 million bytes per second. The effective bandwidth given arbitration, addressing, and device response latency overhead will be significantly reduced by bus transaction protocol overhead, but still on the order of 100 to 500 million bytes per second. For the stereo-vision example, which transfers two streams of 32-bit alpha-RGB 320×240 frames 30 times per second, it requires 18,432,000 bytes per second, or approximately 20% of the available PCI 2.1 effective bandwidth (assuming effective bandwidth is 100 million bytes per second). A single, full-resolution encoding (525×427) video stream would require 26,901,000 bytes per second, or about 27% of PCI 2.1 effective bandwidth.

The chipset used for video encoding in the stereo-vision example is the Bt878, now also updated as the Cirrus Stream Machine, and works by fetching DMA RISC engine code from host memory, so some additional PCI transfers are initiated by the chip to fetch code as well as transfer of frame data to the host memory. The stereo-vision systems implemented at the University of Colorado using PCI 2.1 have had no problem making use of PCI 2.1 for this application with a 320×240 30 fps alpha-RGB encoding. Many embedded applications have more than enough bandwidth available from PCI 2.x. One final important feature of PCI is that initiators can configure targets for a maximum and minimum burst length. The minimum serves as a method to reduce overhead so that targets can't transfer small blocks that would incur high overhead for each bus-arbitration and address cycle compared to fewer larger block transfers. The maximum prevents a target from overusing the bus and provides some fairness in bus arbitration for multi-target systems.

Since the first edition, many embedded camera systems now use USB 2.0 for standard definition or lower-resolution high-definition (e.g., 720p)

for raw uncompressed video. At about 1080p at 30 Hz and above, if frames are not compressed, USB 2.0 has insufficient bandwidth to transport the frames. So, many USB 2.0 web cameras and embedded cameras have built-in MPEG-2 or MPEG-4 encoders. For machine and computer vision applications, raw uncompressed frames are ideal because the image processing algorithms generally can't work on compressed data directly. In fact, most cameras today have built-in MPEG-4 encoders, including embedded SPI (Serial Peripheral Interface), MIPI (Media Independent Peripheral Interface), and even gigabit Ethernet cameras. Developers and researchers in machine and computer vision therefore sometimes still use NTSC analog cameras or use Camera Link, direct parallel LVDS (Low-Voltage Differential Signal) bus, or specialized gigabit Vision Ethernet links. Most mobile phones use SPI or MIPI and provide lower-resolution digital video or high-definition snapshots. This too is changing rapidly as of the time of publication of this second edition, with the emergence of USB 3.0 cameras, gigabit Vision Ethernet, and continued use of NTSC and Camera Link. The author most often uses NTSC-to-USB-2.0 frame grabbers, Camera Link to USB 3.0 bridges, or standard definition with USB 2.0 to avoid the overhead of decoding digital video frames in computer and machine vision applications. Finally, to come full circle to PCI, PCI Express, which is largely compatible with PCI in general from a software and driver viewpoint, also continues to be a great transport for digital video [Siewert14].

Since the publication of the first edition of this book, the trend toward high-speed differential serial IO has continued to evolve with the emergence of USB-3.0, PCI Express 3.0, and the emergence of Thunderbolt and announcements of new PCI Express 4.0 and USB-3.1 standards. PCI Express 4.0 is capable of 16 billion transactions per second with 128b/130b encoding (130 bits to encoded 128 data bits) and an effective data rate of 15.754 billion bits per second at the link layer. The new USB-3.1 will be capable of data transfer rates close to 10 billion bits per second and uses a low-overhead 128b/132b encoding. Most of the early differential serial buses used 8b/10b encoding to control running disparity (the number of repeated 1's or 0's—logic high or low) to improve signal integrity. Overall, the use of differential serial continues to be the pervasive technology for high data rate IO for both embedded and general-purpose computing. The trend can be traced back to the original development of gigabit Ethernet, USB, and PCI Express, which have for the most part completely replaced traditional serial and parallel bus protocols in the new millennium and are

likely to continue to be the pervasive interconnection technologies. The following chapters, unchanged from the first edition, trace the emergence of differential serial links, buses, and transport protocols and the transition to high-speed differential serial links from analog serial and parallel buses. The development of encoding methods to improve signal integrity (8b/10b) and the use of link layer and higher-layer protocols to improve error handling for all methods of interconnection can be traced back to the development of Infiniband in 1999, which has become a high-performance computing interconnection standard, but has influenced many other related standards, such as USB, Ethernet, and PCI Express. An understanding of the history that led to the emergence of PCI Express, USB, and gigabit Ethernet will assist the reader with strategies to scale embedded solutions, methods to interface high data rate devices, such as cameras, and provide a long-term vision for IO.

The commercial computing market, specifically high-speed networking, graphics, and databases, pushed PCI and other parallel bus technologies at the turn of the millennium in the year 2000 to evolve into a very high-bandwidth interconnection. At that time, the AGP (Accelerated Graphics Port) standard was developed as a specific single-target expansion for PCI to accommodate high-bandwidth RAMDAC (RAM Digital-to-Analog Converters) used to drive monitors with high-fidelity graphics. Following PCI 2.x, the PCI-X 1.0a and PCI-X 2.0 standards were developed for networking and database host bus adapters and provide 64-bit 133 MHz and up to 64-bit 266/533 MHz bandwidth. The theoretical limit of the PCI bus signaling is 533 MHz. At this speed, the problem of skew between the address/data lines is significant and requires careful layout of the bus traces and advanced signaling techniques that make PCI-X 2.0 expensive and difficult to implement, especially for buses that accommodate more than one target and initiator.

The development of gigabit Ethernet in the early years of the new millennium (2000) at 1 G and 10 G rates (where G = gigabit/sec) helped drive the demand for high-rate PCI bus development for host interfaces to this new high-speed network interconnection. The PCI-X 1.0a standard, at 64-bit 133 MHz (just over 1 GB/sec), became popular for gigabit Ethernet network interfaces at this time; however, this high-end parallel bus was still not sufficient to support 10 G Ethernet. To support 10 G Ethernet, PCI Express was developed, which has a drastically different signaling and physical layer than PCI or PCI-X. PCI Express provides high-speed 2.5 G serial byte

lanes that can be ganged up. With the introduction of PCI-X, the standard introduced an important new concept—split transactions.

In PCI 2.x, when a device has high response latency, the bus delays until the target device responds. With the delay policy, having even one slow target on the bus decreases performance for all targets and initiators on the bus. Split transactions eliminate this delay by providing buffer queues for writes to the bus so that they can be posted by an initiator and drained to a slow target over the bus when it's ready. The initiator is not delayed in this case as long as the write buffer queue is not exhausted before the data is drained to the target. Likewise, on reads, split transaction allows the initiator to post a read request to the bus, which initiates the target read, and if the target is slow, allows the target to negotiate for completion in a later transaction—the bus is freed in the meantime for other transactions. Both PCI-X and PCI Express are split-transaction bus standards—this greatly improves the effective bandwidth because it does not allow the bus to be held for arbitrarily long delay periods. However, there is, of course, still arbitration and addressing overhead.

8.2.7 High-Speed Serial Interconnection

As traditional back-plane buses have become problematic as far as laying out signal traces and dealing with high-speed signaling and skew (rates above 100 MHz), several new high-speed serial interconnection standards were introduced, including:

- Universal Serial Bus
- Firewire
- PCI Express
- Gigabit Ethernet

All four serial/network interconnections can be used for real-time embedded systems and provide an attractive alternative to bus integration. A full discussion of all the high-rate serial protocols is beyond the scope of this text.

The wide adoption of PCI for real-time embedded systems in the past and key features of PCI Express make it an interconnection for scaling existing systems that became popular in the new millennium, and it continues to grow and has become pervasive. PCI Express can be routed on a board, on a back-plane, and even out of a box on a cable for short distances.

Furthermore, it's composed of serial byte lanes operating at 2.5 G for each lane, with the capability to gang up lanes in x1, x2, x4, x8, and x16 configurations. The stated design goal of PCI-E (PCI Express) is to maximize the bandwidth per pin on the interconnection. Each PCI-E byte lane is full duplex, allowing concurrent transmit and receive at 2.5 G. Given these characteristics, we can compare the PCI 2.x, PCI-X, and PCI-E standards as far as bandwidth per pin:

1. PCI-E: $[(2.5 \text{ Gb/s/direction} \times 8\text{b/direction}) \times (1\text{B}/8\text{b})]/40 \text{ pins} = 100 \text{ MB/s/pin}$
2. PCI 2.x: $[(32\text{b} \times 33 \text{ MhZ}) \times (1\text{B}/8\text{b})]/84 \text{ pins} = 1.58 \text{ MB/s/pin}$
3. PCI-X 2.0 266: $[(64\text{b} \times 266 \text{ MhZ}) \times (1\text{B}/8\text{b})]/150 \text{ pins} = 7.09 \text{ MB/s/pin}$

The trend for PCI-E continues to improve from this initial 2.5 billion transactions per second and high effective bandwidth per pin with rates in PCI-E 4.0 expected to top out at 16 billion transactions per second per byte lane; this latest version of PCI-E, PCI-E 4.0, was announced in 2011 and the final specification has been targeted for release in 2017. The link layer encoding and the higher-level transport layers used in PCI-E add overhead, but the effective data transfer rates are still very high and the pin efficiency is still much higher than any previous parallel bus IO. The ability to gang up serial byte lanes from x1 to x4, x8, x16, and even x32 PCI-E will also keep PCI-E relevant for some time to come.

PCI-E clearly has the advantage from the perspective of interconnection layout and cabling over earlier parallel bus technologies, such as PCI 2.x and PCI-X. The complication is that serial byte lane transmission requires significant digital signal processing on each byte lane. Like fiber channel and gigabit Ethernet, PCI-E uses an 8b/10b encoding scheme with a link layer and network layered architecture to achieve 2.5 G transfer rates. Given the demands of gigabit transport, the cost of this digital signal processing and network stack implementation has actually become more feasible than the cost of traditional bus high-speed layout. Furthermore, the ability to use high-speed serial interconnection on-chip, onboard, and off-board makes these standards more attractive than traditional bus architectures. Finally, PCI-E has been designed to be compatible with PCI 2.x and PCI-X from the firmware viewpoint, despite a radically different data transport method. The PCI-E standard supports the same plug-and-play configuration, burst transfers, interrupts, and all basic features of PCI 2.x.

The PCI-E interconnection provides byte lane interconnection with a network layered architecture, as shown in Figure 8.7.

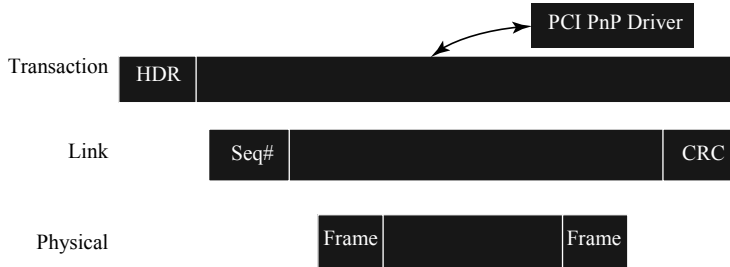


FIGURE 8.7 PCI Express Byte Lane Network Architecture

The PCI-E standard provides not only significantly more bandwidth compared to PCI 2.x and PCI-X but also some features to support real-time continuous media with isochronal channels. The isochronal channels provide bandwidth and latency performance guarantees for transport. The advent of PCI-E, USB, Firewire, and gigabit Ethernet has provided an alternative interconnection architecture for real-time embedded systems. It is likely that these new high-speed serial interconnections will be designed into many future real-time embedded systems.

8.2.8 Low-Speed Serial Interconnection

Many real-time embedded systems include not only high-rate IO for services such as video or network transport but also low-rate command/response or monitoring interfaces. For example, in our stereo-vision system, the servos are commanded through a low-rate multi-drop RS232 interface. The Microchip PIC (Programmable Integrated Circuit) has a TTL logic-level digital serial interface that can be interfaced to the higher-voltage RS232 serial interface. The servos in the stereo-vision application can tilt/pan through wide angles quickly, but have an accuracy of only several degrees, so the command rate required for tracking a quickly moving object at a distance of 10 feet or more is on the order of 1 to hundreds of milliseconds. The command protocol used on the PIC is a simple byte stream command format with opcode bytes and operand bytes. To command a given servo to a new position simply requires sending a PIC address (because the serial interconnection is multi-drop), a servo address, and opcode byte, followed by a servo position operand. A command therefore requires 4 bytes, and at a 1 millisecond rate, this is only 4,000 bytes/second. Clearly PCI,

USB, Firewire, or any of the previously presented high-rate interconnection architectures are not warranted for this type of interface.

The RS232, common serial, point-to-point data transmission has been used in real-time embedded systems since the advent of the industry and remains a common low-rate and debug interface. The RS232 link normally tops out around 115,200 bits/second (about 12 KB/sec) and is not capable of long-distance transmission due to line noise at the 12-volt signaling levels it uses. Other options have evolved that provide similar low- to medium-rate transmission with longer distance, multi-drop, and higher bit rates. These options are widely used in real-time embedded systems:

- RS422—a differential +/- 5v serial link capable of 1 megabit/sec and distances up to 1 km
- Multi-drop RS232, RS422—adding a protocol to address targets on a common link with capability to forward
- I2C—a medium-speed digital interconnection typically used onboard to interconnect chips such as EEPROM to a processor
- SPI (Serial Peripheral Interface)—a digital serial protocol capable of medium rates

A full discussion of all the low- to medium-rate serial protocols is beyond the scope of this text and continues to evolve rapidly over time, but the introduction provided here is a good starting point.

8.2.9 Interconnection Systems

Having discussed the components that can be used to interconnect devices with processors in a real-time embedded system, let's briefly discuss how these components might be arranged in an interconnection architecture. Real-time embedded system architectures and design will be discussed more fully in Chapter 12, but an overview will help summarize the possibilities. Two architectures are most common. The first is the hierarchical network interconnecting a main processor complex with a number of microcontrollers. This architecture has been most popular in robotics and also for aerospace applications, where many sensors and actuators are distributed in a large system, yet processing and services provide system-level functions. For example, a robotic arm that has five degrees of freedom (base, shoulder, elbow, wrist, and claw) with torque control so that

it can handle massive objects might include a microcontroller to provide control for each joint motor. The five microcontrollers, one at each joint, can interface locally to the actuation motor with a DAC and provide closed-loop feedback control based upon local position and stress/strain sensors to provide smooth rotation, even when the arm is handling objects with significant mass. Providing the local control reduces the amount of cabling back to the main processor. The DAC and the sensor interface cabling are routed to the microcontroller, which is physically integrated local to each joint (or control point). The microcontroller has more than sufficient capability to provide the torque control and closed-loop monitoring. Now, the five microcontrollers can be interfaced via a low- to medium-rate serial interconnection back to the main processor complex for commands and to provide status—the main processor complex runs complex services, such as path planning, possibly camera-based object recognition, a user interface, and system health and status monitoring. In fact, the robotics community likens this hierarchical approach to the human body, which includes local reflex control as well as centralized processing in the brain—for example, Rodney Brooks's subsumption architecture [Brooks86].

The alternative to the hierarchical interconnection is a centralized processor complex integrated on a high-rate interconnection, such as PCI, with many IO device interfaces also integrated on PCI. This architecture has an advantage in that all processing can be done in a single processor complex; the distributed processing of the hierarchical architecture requires different development, debug, and test methods compared to the processor complex. Often the processor complex is a microprocessor with an RTOS, and the microcontrollers are simple Main+ISR applications. The downside to the centralized processing is that most often all IO cabling must come from the common central processing enclosure and be routed to sensors and actuators distributed throughout the system. Deciding which type of architecture makes most sense is often driven by the system requirements for actuators and sensors—the number, how distributed they are, how much latency in sensor/actuator activation is allowable, and, of course, cost and complexity.

8.2.10 Memory Subsystems

Real-time embedded systems require nonvolatile data storage to boot the system and to start services. After a power-on reset, the processors in the processor complex each vector to a hardware-defined starting address to execute code. This starting address, typically a high address, such

as 0xFFE0_0000, is designed to map a nonvolatile storage device, such as EEPROM or Flash memory, so that boot code can be stored permanently at this address and executed following a reset to initialize the system. The boot code initializes all basic interfaces and normally loads a basic RTOS so that application services can be loaded and run. The code (often called text segment), data (initialized, uninitialized, and read-only), heap, and stack segments must be created in a working memory by firmware. Figure 8.8 shows a typical memory map for an embedded system.

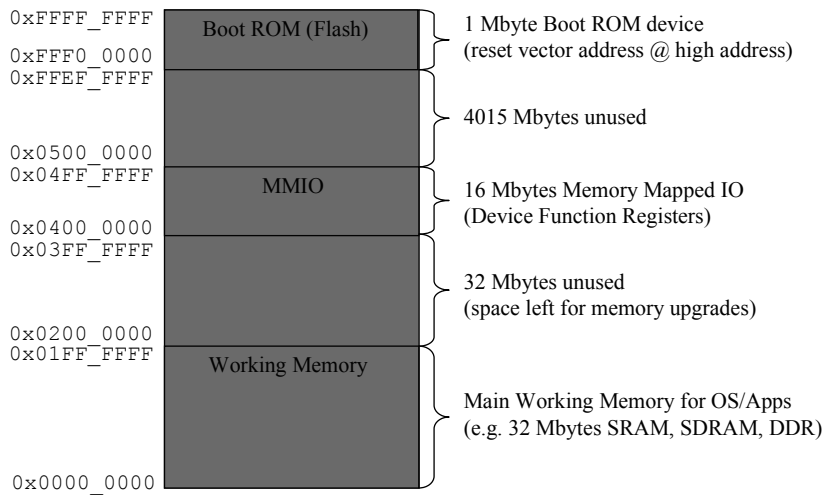


FIGURE 8.8 Common Memory Map for an Embedded System

The memory map is really a logical view of memory from the viewpoint of address space through which firmware and software can access devices. From a hardware viewpoint, memory is better described as a hierarchy of storage devices, including:

- Registers (CPU and memory mapped for device control)
- Cache
- Working Memory
- Extended Memory

Figure 8.9 shows a typical memory hierarchy for an embedded system.

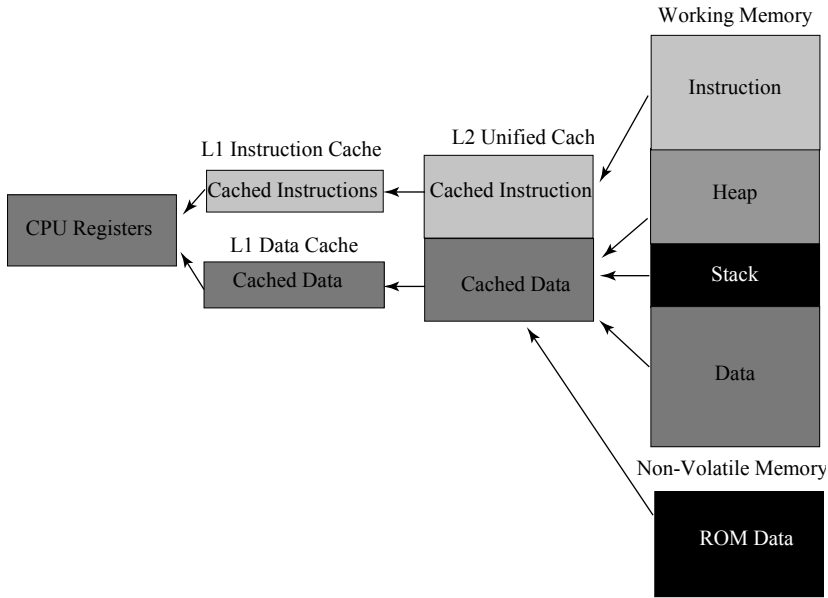


FIGURE 8.9 Common Physical Memory Hierarchy for an Embedded System

8.3 Firmware Components

Some components can be realized only in hardware, but many can be implemented with software or firmware (noting that software interfacing directly to hardware is typically called firmware). Furthermore, if the real-time embedded system has any software-based services or even just management, firmware is needed to interface hardware resources to software applications.

8.3.1 Boot Code

The universal definition of *firmware* is code or software that runs out of a nonvolatile device to make hardware resources available for the rest of the application software. Firmware providing this function is normally referred to in general as *board support package* (BSP) firmware because traditionally this firmware has initialized and made available all onboard resources for a processor complex to software applications. Before the resources have been fully initialized, the firmware boots the board by executing code out of a nonvolatile device so that one or more basic interfaces are made operable and the system can now download additional application software. For example, the BSP boot firmware might initialize an Ethernet interface and provide TFTP download of application code for execution.

8.3.2 Device Drivers

Device interface drivers are most often considered firmware because they directly interface to hardware resources and make those resources available to higher-level software applications. The architecture of a device driver interface is depicted in Figure 8.10 and includes both an HW device interface and an SW application interface.

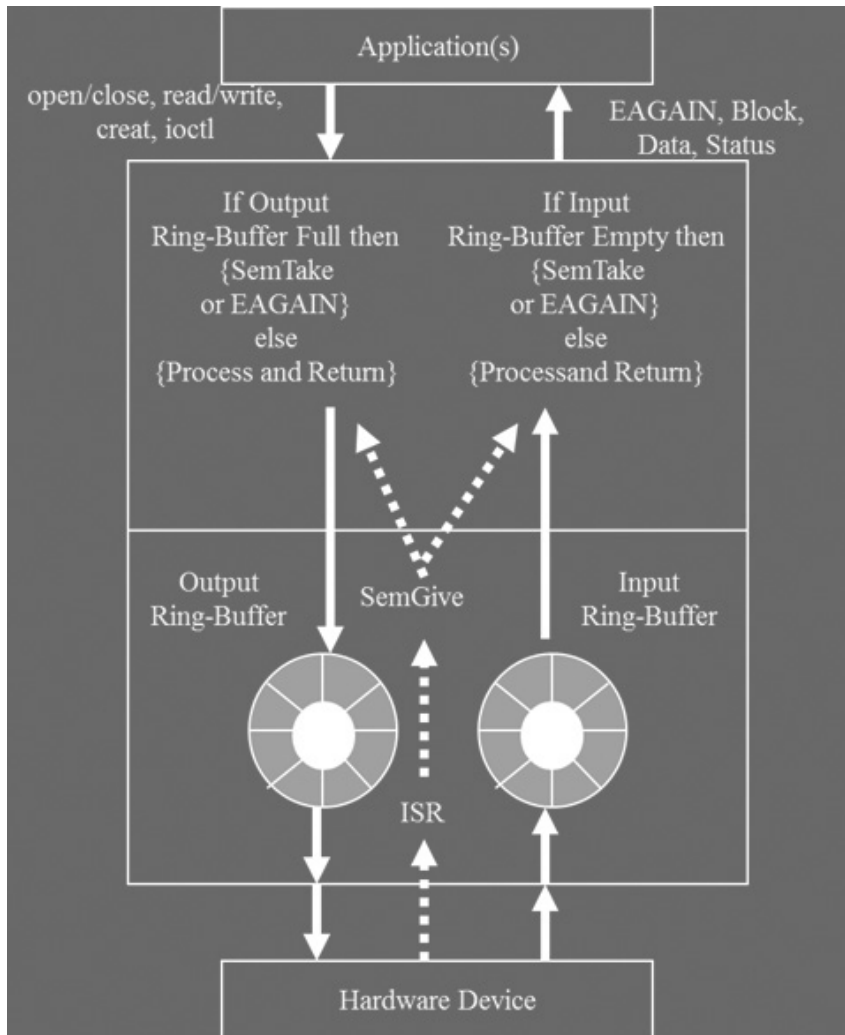


FIGURE 8.10 Device Driver Firmware Interface

8.3.3 Operating System Services

Not all real-time embedded systems require an operating system, as discussed in Chapter 3. Most RTOS implementations do, however, provide a layer of software that acts a single interface for all applications to gain access to system resources. Furthermore, most real-time systems incorporate an RTOS, which provides a framework for resource management and for scheduling processor resources with an RM policy. The RTOS also provides commonly needed services and libraries used by application services. The most fundamental services and mechanisms provided include the following:

- Priority preemptive scheduler for threads
- Thread control block management
- Inter-thread synchronization and communication (e.g., semaphores and message queues)
- Basic IO for system debug and bring-up (e.g., serial, Ethernet, LED)
- Interrupt service routine installation on interrupt vectors
- Transition from boot to operational state
- Timers for delays and blocked thread timeouts
- Drivers for basic hardware devices (serial, Ethernet, timers, nonvolatile memory)

Extended services beyond these may be provided to assist development and debug of a system:

- Cross debug agent
- Interactive shell to view control blocks and system context
- Ability to dynamically load and execute code object files
- Interface to resource analysis tools (e.g., WindView, now known as System Viewer)

8.4 RTOS System Software

An RTOS does not need to provide the same wealth of system services as a full multiuser operating system, such as Linux. The understanding is

that the RTOS user prefers simpler mechanisms and is willing to take on more responsibility in the application to explicitly manage resources. For this reason, it is important that programmers work with an RTOS code with extra precaution to avoid timewasting debug sessions. Some good coding practices that are even more important with an RTOS compared to Windows or Linux include the following:

- Always check return codes for an RTOS API function call.
- Be aware of stack sizes for each task created, and stack usage by local C variables and parameters passed to ensure that stack overruns are not causing problems.
- Use tools such as the Tornado browser and WindView to ensure that you have not overloaded the CPU.
- VxWorks compiled with the MMU Basic has memory protection only for the kernel code, so check array bounds extra carefully to ensure that wild writes are not destabilizing code.
- Do not use ***printf*** in time-critical code because it introduces blocking that will significantly change timing (instead use `logMsg()` calls).
- Be careful to use only API calls that are documented as okay to use in interrupt handlers in your kernel and ISR code (e.g., you can't receive a message in an ISR, but you can send one).
- Set priorities of tasks according to RM theory and demote standard VxWorks services if required.
- Write shutdown functions for tasks that release all resources that tasks create and use.

Following the preceding precautions will make writing RTOS code easier. The standard RTOS mechanisms are designed to make multithreaded real-time applications simpler. Multiple threads need methods for sharing resources, sharing data, synchronizing, keeping track of time, and receiving notification of asynchronous events. The DVD includes numerous examples of using basic mechanisms in the VxWorks RTOS.



8.4.1 Message Queues

The VxWorks RTOS supports both POSIX and native message queue mechanisms. Message queues are used to synchronize tasks and to pass

data between them. The enqueue and dequeue operations are thread-safe (a partial write or read is not possible).

Both types of message queues can be read and written in blocking or non-blocking modes. For blocking modes, when a task does a read on an empty message queue, it blocks until a message is enqueued by another task, allowing it to read and continue. Likewise when a task tries to write a full message queue in blocking mode, it is blocked until the queue has room for the new message. It is wise to set a timeout upper bound rather than using `WAIT_FOREVER` so that indefinite blocking will not occur and errors in synchronization and resource management will be detected through timeouts rather than failure to make progress.

For non-blocking message queues, when a task does a read on an empty queue, it will be returned the error code `EAGAIN`, indicating that no messages were available to read and that the task should try again later. Likewise, for a non-blocking message queue, a task will get the same `EAGAIN` error code if it tries to write to a full queue. This indicates that the writer should try again later, perhaps allowing a read of that queue to create space.

The POSIX message queues include a feature to enqueue messages with a priority level and to read the priority when messages are dequeued. Higher-priority messages are always dequeued first. This essentially allows a sender to put an important message onto the head of the queue. Example code for usage of message queues in VxWorks can be found on the DVD in the `VxWorks-Examples` directory. The `posix_mq.c` shows basic features of POSIX message queues, including priorities. The `heap_mq.c` file shows how pointers to heap allocated buffers can be used with message queues for zero copy buffers. For the heap message queue, the sender must always allocate and set the pointer before sending it, and the receiver should determine when it is safe to de-allocate the buffer. Message queue sends can be called in ISR context, but receive can never be called in ISR context, only in task context.



8.4.2 Binary Semaphores

The binary semaphore is the simplest and most often used mechanism in the RTOS. The `semGive()` function is often used in ISR context to unblock a service handling task when data becomes available as indicated by a hardware interrupt. The `semTake()` call is most often used by tasks to wait for a server request (new data available) or to synchronize with another

task. The `two_tasks.c` code on the DVD provides an example of tasks that synchronize each other using a binary semaphore. Care should be taken to set the binary semaphore initial state (FULL or EMPTY), and the protocol for unblocking must be selected. Protocols for unblocking include `SEM_Q_FIFO` and `SEM_Q_PRIORITY`. For `SEM_Q_FIFO`, if multiple tasks block on the same semaphore, then they are unblocked in the order that they originally arrived and blocked. If instead `SEM_Q_PRIORITY` is used, then the highest priority task will be unblocked first. The FIFO protocol ensures fairness, and the `PRIORITY` protocol helps minimize potential priority inversion. Finally in cases where multiple tasks may be blocked and if all blocking tasks should be released at the same time, the `semFlush()` function provides this feature.



8.4.3 Mutex Semaphores

Mutex semaphores are tracked by task and include protocol for unblocking tasks waiting to enter critical sections protected by the mutual exclusion `semTake()` and `semGive()`. A mutual exclusion semaphore uses the same take and give calls, but is created with `semMCreate()` in VxWorks. For mutex semaphores, three unblocking protocols can be specified:

PRIORITY: Unblocks highest priority task first.

FIFO: Unblocks in the same order tasks arrived in for fairness.

INVERSION_SAFE: Implements priority inheritance described in Chapter 6.

Mutex semaphores should be used to implement reentrant functions when these functions need to access global resources. Initial conditions are important, and most often mutex semaphores are initially set full to allow initial access to critical sections. The `prio_invert.c` code shows an example of using an inversion-safe mutex semaphore in VxWorks.

8.4.4 Software Virtual Timers

Most embedded hardware systems include several hardware interval timers. A PIT (Programmable Interval Timer) can be set so that it will generate an interrupt on a periodic basis, often 1, 10, or 100 milliseconds. The ISR handling the hardware PIT interrupt (IRQ0 on the x86 architecture) updates virtual timers, which keep track of the interrupt count (called *ticks* in VxWorks). All `taskDelay()` calls, timeouts specified, and calls to `tickGet()` are driven by the PIT interrupt rate. If the basic rate is set to 1 millisecond

with `sysClkRateSet(1000)`, then timing accuracy for delays and timeouts will be within 2 milliseconds. It is possible to have just missed a tick expiration when a timer is first set and also possible to overrun at least one tick before a timeout handler is invoked. The PIT can be set to raise timer interrupts more frequently than 1000x per second, but this starts to require significant CPU time to maintain virtual time at high rates. The DVD includes three examples to demonstrate the use of virtual time, including `itimer_test.c`, `posix_clock.c`, and `posix_rt_timers.c`.



8.4.5 Software Signals

Software signals can be thought of as the software equivalent of an interrupt. They are the main mechanism providing asynchronous handling of events in task context. A task can set itself up to catch signals thrown by other tasks or by ISRs. The DVD includes `rt_signal_test.c`, which demonstrates the use of POSIX real-time signals. The POSIX signals queue, unlike most signals, which prevents loss of signals if a signal is thrown while the catching task is in the process of handling a previously thrown signal.



8.5 Software Application Components

Software components represent the most easily updated and flexible implementation of services in a real-time embedded system. The service state machine can be coded in an RTOS framework readily using tasks and task synchronization mechanisms, including binary semaphores, mutex semaphores, message queues, ring buffers, and ISRs.

8.5.1 Application Services

Application services are software images loaded after boot and after some form of RTOS is functional to provide specific services. These services are simply software implementations of service state machines and execute code within the context of a task.

Services must often be synchronized with each other or ISRs. This is normally accomplished with a binary semaphore. The binary semaphore blocks the calling task until the semaphore is given by an ISR or another task. So, if a task calls `semTake(S)` where `S=0`, the task is blocked and enters a pending state until another task or ISR uses `semGive(S)` to set `S=1`. When `S` is set, then all tasks blocked (in a queue) are unblocked in queue

order one at a time based upon creation with the SEM_Q_FIFO option. Some RTOS frameworks, such as VxWorks, provide a semFlush(S), which unblocks all tasks presently blocked on S, no matter how many have queued on S. The following example, “two tasks,” provides a simple instructive example of the use of a binary semaphore by two service tasks in the VxWorks RTOS framework. The two tasks C code is

```
#include "vxWorks.h"
#include "semLib.h"
#include "sysLib.h"

SEM_ID synch_sem;
int abort_test = FALSE;
int take_cnt = 0;
int give_cnt = 0;

void task_a(void)
{
    int cnt = 0;
    while(!abort_test)
    {
        taskDelay(1000);
        for(cnt=0;cnt < 10000000;cnt++);
        semGive(synch_sem);
        give_cnt++;
    }
}

void task_b(void)
{
    int cnt = 0;
    while(!abort_test)
    {
        for(cnt=0;cnt < 10000000;cnt++);
        take_cnt++;
        semTake(synch_sem, WAIT_FOREVER);
        taskDelay(1000);
    }
}

void test_tasks(void)
{

```



```

sysClkRateSet(1000);
synch_sem = semBCreate(SEM_Q_FIFO, SEM_FULL);
/* receiver runs at a higher priority than the sender */
if(taskSpawn("task_a", 10, 0, 4000, task_a, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0) == ERROR)
{
    printf("Task A task spawn failed\n");
}
else
    printf("Task A task spawned\n");
if(taskSpawn("task_b", 11, 0, 4000, task_b, 0, 0, 0, 0, 0,
             0, 0, 0, 0, 0) == ERROR)
{
    printf("Task B task spawn failed\n");
}
else
    printf("Task B task spawned\n");
}

```

The `test_tasks` function spawns Task A and Task B with entry points `task_a()` and `task_b()`. Task A is assigned priority 10, which is higher than Task B priority 11. The semaphore `synch_sem` is initially set full (=1). When Task A is spawned, it will preempt Task B, but will delay for 1 second (1,000 ticks), and then execute a loop and give `synch_sem`. Because Task A yields the CPU initially, Task B will execute despite being lower priority and will execute its loop and take `synch_sem`. On the first execution of Task B, the take will be successful, and Task B will then delay for 1 second—the state of `synch_sem` will be empty (=0) at this point. Following the take by Task B, Task A will have come out of delay and will preempt B regardless of whether it's done with its delay and busy—at this point Task A will give `synch_sem`, and this strict alternation will continue. The initial condition of `synch_sem` (full=1 or empty=0) is critical as well as the relative priorities of Task A and Task B. It is possible for the two tasks to deadlock if the initial conditions are different. The example will always alternate as it's provided here, but if Task A, the giver, did not yield the CPU with a task delay call and was assigned higher priority, it's possible that Task B would never run because it can't preempt Task A. This system as presented here will not deadlock either because circular wait is not possible given the priorities and the initial conditions.

8.5.2 Reentrant Application Libraries

Code shared by multiple threads of execution, as is often the case with application code, must be reentrant. Reentrant code is able to be interrupted and preempted in the execution context of one thread and then executed in the context of a new thread without side effects that would cause either thread to suffer functional bugs. So, reentrant code must carefully handle global resources and protect them so that they are mutually exclusively used by multiple threads. The following are the four main methods to ensure that global data is either protected or converted into task-specific context data:

- Protection of data with use of `intLock()` and `intUnlock()` to ensure that preemption around global data accesses is impossible at the ISR and task level.
- Protection of data with use of `taskLock()` and `taskUnlock()` to ensure that preemption around global data accesses is impossible at the task level.
- Elimination of global data with task variables so that data is no longer shared but owned by a task context and stored in the TCB (Task Control Block).
- Protection of global data with use of `semMCreate()` to establish a mutex semaphore and `semTake()` and `semGive()` to wrap the critical sections where global data is manipulated with multiple instructions that could otherwise be interrupted or preempted.
- Use of stack data only (C parameters and function locals) so that each calling task has its own copy of the data.

Any of these global data elimination or protection methods will make functions thread-safe so that they are reentrant and can be used by multiple concurrently active threads. One of the best and simplest ways to make a function thread-safe is to recall that global data can be eliminated by making use of stack-only. In C, local variables and parameters are maintained in stack memory. Every VxWorks task must specify stack space when `taskSpawn()` is called. Insufficient stack space and declaration of large arrays as C locals can introduce bugs. However, stack-only variables in functions implement a pure function that is thread-safe.

8.5.3 Communicating and Synchronized Services

Application code normally requires multiple services to synchronize and to share data or global resources. As discussed in the previous chapter, a region of memory can be shared by two tasks and updated or read in a critical section using a mutex semaphore—the mutex semaphore guarantees mutually exclusive access to the shared memory by one task at a time in spite of the possibility of preemption of one task by another. A higher-level abstraction of this is the message queue (see Figure 8.11). The message queue provides a buffer shared by two tasks and allows each task to atomically enqueue or dequeue a message buffer. The atomic enqueue and dequeue operations are implemented using a critical section.

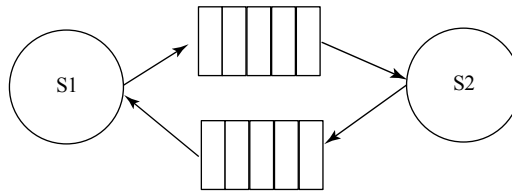


FIGURE 8.11 Message Queue Communication between Tasks

A critical section simply requires that other tasks (services) are not able to preempt S_1 or S_2 while they are in the middle of copying a message from their local memory into the global message queue buffer. The underlying implementation can use mutex semaphores to prevent more than one task from entering the critical section. Other methods that can be used include `taskLock`, `taskUnlock`, whereby the scheduler is actually disabled around the critical section (no preemption is possible at all!). Finally, it's also possible to use `intLock`, `intUnlock` to mask out interrupts entirely in a critical section—recall that preemption occurs through an interrupt or an RTOS system call (a yield inside a critical section is an error). The `taskLock` approach works, but prevents all scheduling during the critical section and therefore has negative impact on the RM policy. Likewise, the `intLock` approach works as well because it disables scheduling changes in addition to masking interrupts and potentially missing events associated with those interrupts. The user of the message queue, however, does not need to be concerned with the implementation, but rather that the enqueue and dequeue are atomic (non-preemptible). The implementation of the message queue should ideally not disable scheduling and should prevent unbounded priority inversion.

The message queue, once created, has simple semantics for the enqueue and dequeue that can be either **BLOCKING** or **NONBLOCKING**. The message queue is created with a fixed message size, a maximum queue length, and **BLOCKING** or **NONBLOCKING** semantics. When the queue is created **BLOCKING**, writers to a full queue will be blocked when they try to enqueue a message. Likewise, readers will be blocked when they try to dequeue a message from an empty queue. With the alternative **NONBLOCKING** semantics, the writer to a full queue will simply be returned an error code, **EAGAIN**, indicating that the enqueue was not successful. Likewise, the reader of an empty queue will be returned **EAGAIN**, indicating that there was nothing to dequeue. A service using the message queue in a **NONBLOCKING** mode may want to do other work and attempt to enqueue or dequeue again at a later time. The **BLOCKING** semantics are used when the service has nothing else worthwhile to do if it can't enqueue or dequeue successfully.

One downside of message queues is that they require a copy of the S_1 local buffer in the global message queue buffer—this is not so efficient. A variant use of message queues that improves efficiency is the heap message queue. In this case, pointers are sent as messages rather than data. The pointers are set to point to a buffer allocated by the sender, and the pointer received is used to access and process the buffer—the receiver normally frees the buffer. It is critical that the sender allocate the buffer and the receiver de-allocate to avoid exhaustion of the associated buffer heap (a pool of reusable buffers). Figure 8.12 shows a heap message queue. The heap message queue avoids the copy otherwise required, which takes considerable CPU time and wastes memory due to double-buffering of the same data.

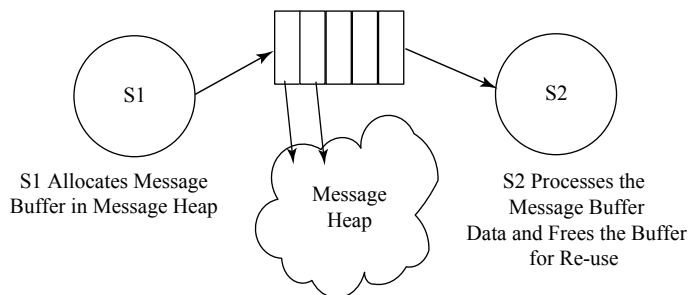


FIGURE 8.12 Heap-Based Message Queue Communication between Tasks

Because the buffer heap associated with the message queue is typically much larger than the queue depth (size), normally the queue will become filled with pointers and block writers before the heap is exhausted. As long as the sender always allocates heap and the receiver always de-allocates, the heap message queue works safely and much more efficiently.

Summary

A real-time embedded system is composed of hardware, firmware, and software components. Services can be implemented in hardware, firmware, or software, or some combination of the three. Component design should be completed so that components can be tested as individual units and then integrated into a larger system design.

Exercises

1. Read the following chapters in *PCI System Architecture* by Shanley and Anderson: 1, 2, 17, 18, and 19. This should give you a good overview of the PCI design and how to write code to probe for PCI devices and configure them. You will also find Sections 3.1–3.3 and Section 3.9 of Chapter 3 from the *VxWorks Programmer's Guide* useful.
2. Write a VxWorks ISR that calls `tickAnnounce` every tick and does a `semGive` on a global binary semaphore every `N` ticks of the system clock, where `N` is a global variable such that the `semGive` frequency is adjustable via the shell. Now, write a VxWorks task that does a `semTake` and immediately updates a global counter to record virtual ticks. Write a driver program to test both the ISR and task and provide output that provides evidence that you got it working. So, for example, if you adjust `N` to 1000, on our system your count should increase by one every second. Please note that you are replacing the VxWorks virtual timer ISR, so you must call `tickAnnounce` so the kernel can still keep track of time!
3. Now write an abstracted top-half for your driver that includes all of the standard driver entry points (`open`, `read`, `write`, and `close`) and blocks a calling task on a `read` until `N` ticks has elapsed, at which time it stuffs the read buffer with a time structure, including seconds and milliseconds. Write a test driver that opens the abstracted device and reads from it in a loop, printing the time in seconds and milliseconds—provide evidence

this is working. A write to your driver should allow for a reset of the virtual time value maintained by your driver (i.e., writing anything resets it to zero).

4. Write a PCI device probing function that can be used to find devices on PCI bus 0 and determines vendor ID. Use your code to find the Cirrus logic PCI video adapters and the Intel North-Bridge chipset in the lab and provide evidence that your code works (these devices will be found on every target—some targets may have additional devices as well).
5. Write a PCI probing function to determine the configuration of the North bridge including: latency timing and the arbitration control, and finally determine if the NB will allow memory access by masters other than the main CPU. Demonstrate that your probe works and describe your probe output. Finally, describe how the NB/SB interface in PCI allows for shared interrupts (PCI and IRQ).

Chapter References

- [Brooks86] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot,” *IEEE Journal of Robotics and Automation*, Vol. RA-2 (April 1986): pp. 14–23.
- [Shanley99] Tom Shanley and Don Anderson, *PCI System Architecture*, 4th ed., Addison-Wesley, New York, 1999.
- [Siewert14] Siewert, Sam B., Shihadeh, Jerries, Myers, Randall, Khandhar, Jay, Ivanov, Vitaly “*Low Cost, High Performance and Efficiency Computational Photometer Design*”, SPIE Technology and Applications, Baltimore, MD, May 2014.
- [WRS99a] *VxWorks Programmer’s Guide 5.4*, Edition 1, WRS, Alameda California, 1999.

TRADITIONAL HARD REAL-TIME OPERATING SYSTEMS

In this chapter

- Introduction
- AMP (Asymmetric Multi-core Processing)
- SMP (Symmetric Multi-core Processing)
- Future Directions for RTOS

9.1 Introduction

Hard real-time operating systems have historically been an alternative to cyclic executives, which are normally built as a main loop with the addition of ISR (Interrupt Service Routines), and must provide a predictable response so that rate-monotonic policies for priority-preemptive scheduling can guarantee a predictable response. The RTOS (Real-Time Operating System) has advantages over the cyclic executive in that it provides a framework for services with a large body of code to reuse when building embedded applications along with a scheduler for priority-preemptive scheduling so that RMA (Rate-Monotonic Analysis)-designed services can be directly implemented in this framework. However, the RTOS has traditionally remained a single-processor core framework, designed for AMP (Asymmetric Multi-core Processing), where system designs that integrate multiple cores must simply run multiple instances of the RTOS on each core. This has changed. Like general-purpose operating systems, such as Linux, many RTOS now also can be configured and integrated for SMP (Symmetric

Multi-core Processing). This makes the RTOS solution more scalable and more efficient compared to AMP, but does introduce new complexities and challenges. In this chapter, we review the fundamental concepts of AMP RTOS, introduce the concepts key to SMP operating systems, discuss new support added to popular RTOS, such as VxWorks, and finally provide a discussion of future directions for RTOS, where many of the features will be potentially inspired by popular commercial operating systems, such as Linux. The chapter should assist the reader with fundamental design decisions, such as when to leverage open source embedded Linux in a system compared to open source FreeRTOS or a commercial proprietary RTOS, like VxWorks.

9.2 Evolution of Real-Time Scheduling and Resource Management

Before diving into AMP, SMP, and future directions for both RTOS and traditional OS support for soft or hard real-time as extensions to best-effort support, it is helpful to quickly review the evolution of hard real-time systems. As shown in Figure 9.1, HRT (Hard Real-Time) has evolved from simple cyclic executives (composed of several main loops that operate at a deterministic frequency) with asynchronous ISRs or simple polling of interfaces to the AMP RTOS (main topic of this text), and now toward mixed support of non-real-time best-effort services alongside HRT and SRT on one platform. At present, there is no consensus on how to mix support on one multi-core platform; however, we will review a number of viable options and discuss advantages, disadvantages, and pitfalls in each. The historical evolution depicted in Figure 9.1 is approximate, but provides a representation of how systems focused on throughput and multiuser interaction have co-evolved alongside systems more focused on predictable or deterministic response for real-time applications, such as digital control and process control systems.

Figure 9.1 begs the question of whether the two lines of system evolution and architecture will ever merge. Many researchers and practicing engineers believe they should remain separate, but a growing contingent would like to see merging of the two lines, at least in terms of configuration of one system framework for one or the other types of applications and system solutions. The authors do not advocate either view, but rather attempt to provide the reader some guidelines on how to build systems that support

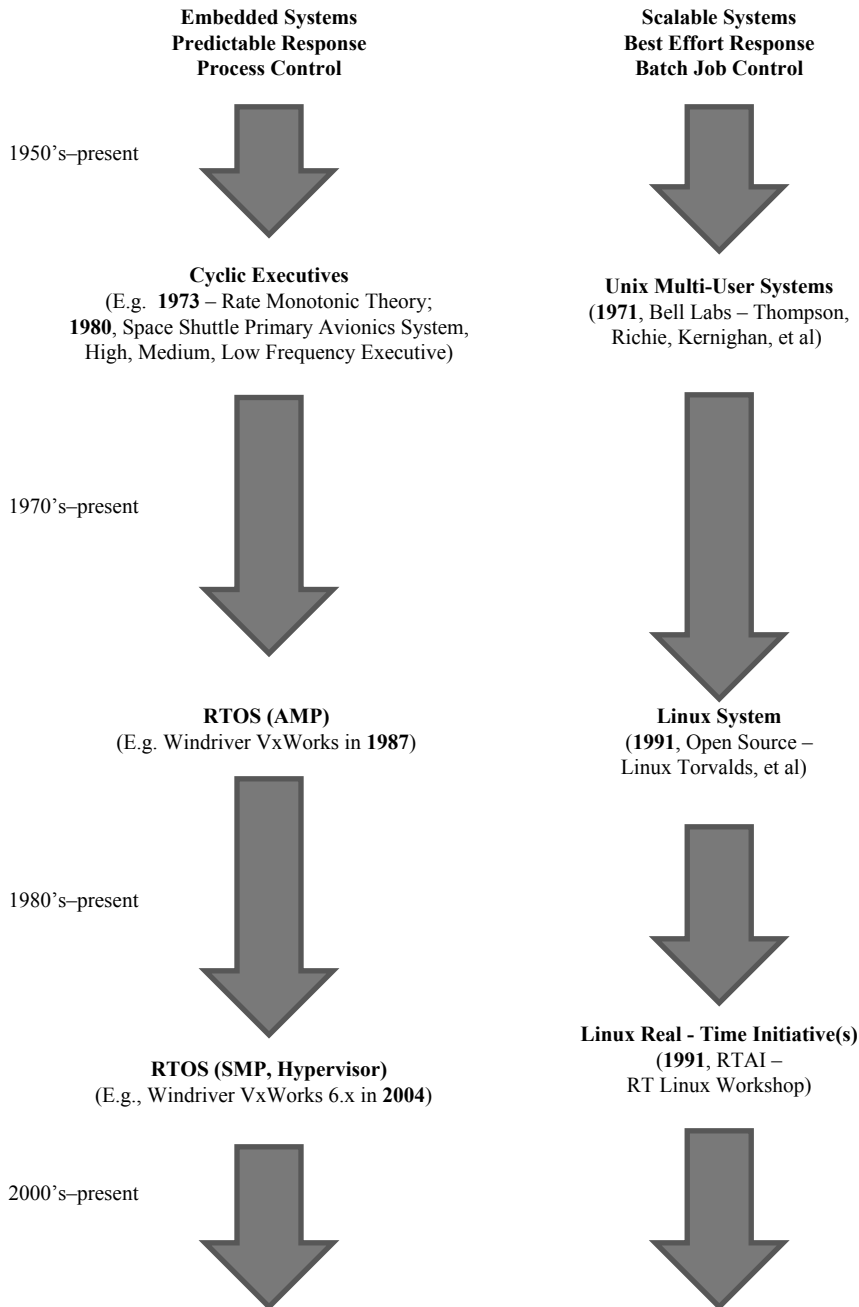


FIGURE 9.1 History of the Co-evolution of Real-Time and Throughput-Oriented Systems

a range of solutions, from hard real-time to best-effort, as well as systems that include mixed support.

9.3 AMP (Asymmetric Multi-core Processing)

AMP (Asymmetric Multi-core Processing) is fundamental to the RTOS and is the assumed architecture in a hard real-time system. It has proven to be just as reliable as a cyclic executive, with some distinct advantages from the software engineering viewpoint in terms of helping to organize larger embedded projects and to decouple scheduling and synchronization from the application itself. AMP has been the main topic of this text, and it is likely that AMP RTOS will continue to be the fundamental real-time embedded system architecture for some time to come. The AMP RTOS, as we have seen with VxWorks examples provided in this text, provides a framework on a single-processor (single CPU and core) system whereby the cyclic executive is replaced by a scheduler with task control, synchronization primitives for tasks, and a library of system code that applications can use to coordinate processing, specify real-time RMA task priorities, and meet deadlines relative to service requests.

The RTOS scheduler framework and tasking have a distinct advantage over the cyclic executive in that they provide a standard approach to coding services, ISRs, and synchronizing them so that multiple developers and larger software engineering efforts have more uniformity. The main disadvantage of the AMP RTOS, just like the cyclic executive, is that scaling to more than one CPU core means replication of the whole RTOS kernel on each processor core (just like the cyclic executive would be replicated on each processor core for scaling). To synchronize two CPUs, each running an RTOS and set of services, we now require message passing over a bus or network. The complexity of embedded applications today, such as the computer vision applications depicted in Figure 9.2, has motivated the use of device interface drivers and modularization of software components with tasking used to implement concurrent processing.

Commercial RTOS, such as VxWorks, have long supported message passing over cluster and bus interconnection networks (e.g., TIPC [Transparent Interprocess Communication] and Wind River's MIPC [Multi-core InterProcess Communication]). For RMA and mission critical applications, where ability to meet deadlines reliably must be formally validated and verified, the AMP approach, as depicted in Figure 9.2, is a simple extension

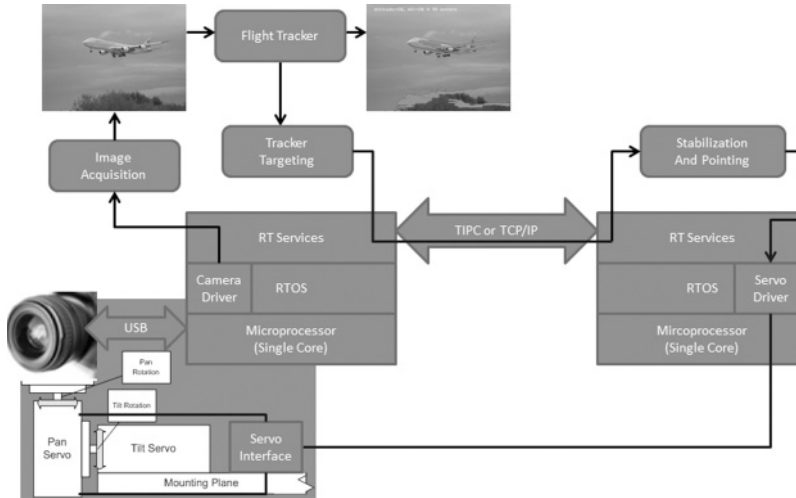


FIGURE 9.2 Example AMP Architecture for Computer Vision

for scaling. We see two instances of the entire RTOS in the solution shown in Figure 9.2. If the design came from a single processor and ran slow, perhaps missing deadlines and lacking RMA margin, then we simply divide up the services according to a software engineering design partitioning (this is subjective), but also often divide interfaces, as shown in Figure 9.2, where the camera interface is now on the first processor and the servo interface is on the second. The partitioning of software interfaces and device driver interfaces reduces loading on each of the two AMP processors, but, of course, we add back some overhead to form and pass messages between them. To summarize, the dual-core solution may be required because of the computational complexity of the image analysis for the Flight Tracker, which likely involves multiple services on the first processor simply to characterize the target of interest (the landing plane). Often continuous image processing (computer vision) requires all of the safely available CPU capability of an AMP computing node because of the resolution of the images, the frame rate, and/or the complexity of the algorithm applied. The stabilization and pointing can simply accept commands over TIPC or simple TCP/IP (Transmission Control Protocol/Internet Protocol) from the first processor and dedicate itself to digital control to point and stabilize the camera at the tilt, pan, and zoom specified.

As seen in the previous example, a wide range of real-time systems that require scaling of computer resources can be solved with AMP. The one downside is that running multiple instances of an RTOS (e.g., VxWorks or

FreeRTOS) adds overhead to the design. If we could have also hosted the service for stabilization and pointing on the first processor, we would have had lower cost and no need for the interconnection network (e.g., Ethernet), and would have saved space, power, and mass in our design. Prior to the new millennium, the idea of multiple processor cores on a single CPU die was not feasible, but today SoC (System on Chip) processors are plentiful and provide dual-core, quad-core, or better on a single chip, with multiple IO interfaces so that the two distinct CPU systems shown in Figure 9.2 could be re-integrated onto a single-chip SoC solution. This does not necessarily mean that AMP will not work. It is possible that two distinct embedded compute nodes could become one SoC that still runs two RTOS instances with services dedicated to each, communicating via a message-passing protocol like TIPC.

The advantage of AMP, whether the design involves two distinct computing nodes or an SoC, is that we can analyze the service loading with RMA on each RTOS, using methods as presented in this text, with no real added complexity. The only new complexity is the message passing between the two RTOS instances and their services, but this is really just like any other form of inter-task communication and synchronization, as previously presented. Furthermore, the porting of a working solution from a legacy system with multiple processor boards to an SoC might be more straightforward and require less re-verification. However, the downside of multiple RTOS instances is a major detractor since it involves significantly more code to configure and maintain even if verification is essentially the same for the otherwise identical RTOS instances. The RTOS, of course, uses memory, processing, and IO resources as well, which could represent substantial overhead.

In an effort to reduce the scaling overhead, many non-real-time systems have made use of a newer architecture known as SMP. For example, Linux has SMP support and normally installs in an SMP configuration since most tablets, laptops, and servers today are multi-core. This is typically true of any modern general-purpose operating system. Based on simplicity, efficient scaling, and lower overhead of a single operating system instance managing services running on multiple cores, the attraction of SMP to embedded solutions is likewise compelling. One final consideration before deciding to jump into SMP rather than AMP is the underlying multi-core hardware architecture.

Multi-core SoCs can and do come in many different hardware architectures. For example, the NVIDIA Jetson quad-core SoC with 192-core GPU (Graphics Processing Unit) coprocessor is an interesting option available at the time this edition was written (shown in Figure 9.3). One of the authors is using the Jetson for embedded computer vision applications for passive stereo vision and visible-plus-infrared image fusion in real time. Likewise, Intel has some interesting new SoCs leveraging the “x86” instruction-set and providing quad-core or better on embedded Intel Atom solutions. The key consideration is how these new SoCs provide concurrent processing in terms of how the CPU cores are interconnected on the chip and whether all cores are general-purpose or whether some are special-purpose coprocessors, like a GPU. In the case of computer vision, much of the processing associated with image processing can be offloaded and accelerated by the coprocessor, therefore freeing up the main SoC cores for other purposes.

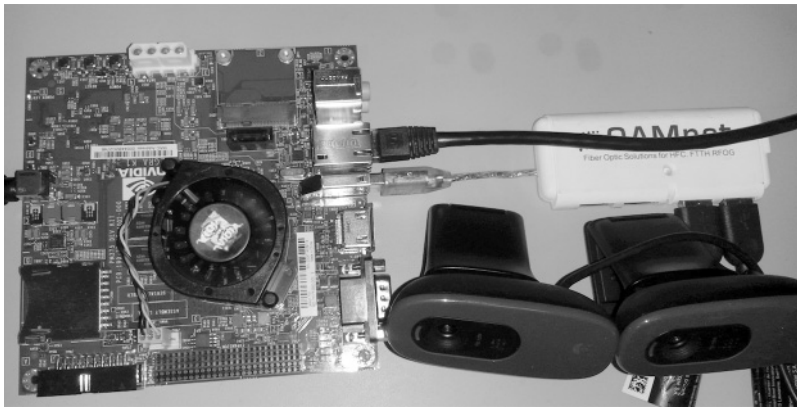


FIGURE 9.3 NVIDIA Jetson ARM SoC with Ubuntu Linux Platform

One way to classify and describe multi-core architectures and the scaling of processing in general is Flynn’s taxonomy, as shown in Figure 9.3. Taking the cross product of the rows and columns gives us SISD (Single Instruction, Single Data), MISD (Multiple Instruction Single Data), SIMD (Single Instruction, Multiple Data), and MIMD (Multiple Instruction, Multiple Data).

Many processors include instruction-set extensions to provide multi-word instructions for applications, like image processing, including Intel (SSE instructions), ARM (NEON instructions), and PowerPC (AltiVec instructions). For example, an operation like XOR can apply to more than just

	Single Instruction	Multiple Instruction
Single Data	SISD (single processor)	MISD (Voting schemes and active-active controllers)
Multiple Data	SIMD (SSE 4.2, ARM NEON, PowerPC AltiVec)	MIMD (e.g., Clusters with TIPC)

FIGURE 9.4 Flynn’s Taxonomy of Concurrent/Parallel System Architectures

a single 32-bit word and can provide XOR of two 128-bit operands. Finally, AMP is an MIMD architecture, but a specific type of MIMD that uses message passing between two RTOS and application software stacks.

For today’s hardware architecture, we really need to extend Flynn’s taxonomy to include a wider variety of vector coprocessors (SIMD)—for example, SPMD (Single Program Multiple Data) processors, like GPUs. An updated Flynn taxonomy that notes this is shown in Figure 9.5.

	Single Instruction	Multiple Instruction
Single Data	SISD (Traditional Uni-processor)	MISD (Voting schemes and active-active controllers)
Multiple Data	SIMD (SSE 4.2, Vector Processing) SPMD (Single Program Multiple Data), GO-GPU	MIMD (Distributed systems (MPMD), Clusters with MPI/PVM (SPMD), AMP/SMP)

FIGURE 9.5 Updates to Flynn’s Taxonomy to Show SPMD (GPU Vector Coprocessors)

The hardware architecture is critical since SMP makes the most sense for SoCs that provide MIMD capability and ideally an interconnection that is uniform. When the interconnection is uniform, we call this UMA (Uniform Memory Access), and this means that all processors can access any address with same cost (latency) for that access. Many MIMD hardware architectures are NUMA (Non-Uniform Memory Access) where latency for memory access depends upon the processor core and address being accessed in memory, or they are true message-passing architectures where the CPUs and their memories are interfaced in a true AMP fashion, with a TIPC, TCP/IP, or other type of message transport network. The NUMA MIMD hardware architectures that use on-chip interconnection that is point-to-point or require minimal forwarding of memory access transactions work best for SMP. Ideally SMP solutions would all run on UMA MIMD hardware architectures, but they do not. This just means that SMP

on NUMA requires a bit of care in software to ensure efficiency, as we will see in the next section.

In Chapter 11, we will discuss use of embedded Linux as an alternative to FreeRTOS or VxWorks. We will see that in some cases, much if not all of the real-time processing can be done by a special-purpose coprocessor or an FPGA (Field Programmable Gate Array), in which case, if all real-time services are offloaded from the processor that hosts the RTOS or general-purpose operating system, then perhaps an RTOS is not needed at all. Hold that thought for Chapter 11, for a more in-depth exploration.

The main takeaway from this section on AMP is that it is still a very valid option for embedded systems due to simplicity and ease of testing during development (each AMP node can stand alone), and many legacy multiprocessor systems were built this way and will be encountered. Now, if time permits, you have significant potential gain in efficiency by going to SMP, and if you are willing to consider more complexity in partitioning hard real-time, soft, and non-real-time services, SMP can offer cost savings and performance advantages.

9.4 SMP (Symmetric Multi-core Processing)

In the previous section we discussed the motivation for SMP compared to AMP, but what constitutes SMP was left a mystery. SMP is best understood by first understanding AMP well, much like an RTOS is best understood by first really understanding a cyclic executive. SMP adds operating system scheduling and resource management complexity compared to the more obvious AMP approach of replicating the entire RTOS and dividing the services up by off-line analysis of resource demands and by interface(s) used, as is most often done in AMP. Recall that one of the main reasons we are driven to multiple processors (and cores) is that one core just can't keep up with the algorithms or data rates required of it for the service we want to provide. Computer vision is a great application example since it is process-, data-, and IO-intensive and finding its way into many interesting embedded applications. So, what is SMP?

SMP runs the RTOS on just one of the processor cores on an SMP (UMA or NUMA) system, and the RTOS then maps tasks onto any one of the “n” cores at runtime during dispatch. Furthermore, for memory management, it must decide exactly where to allocate heap buffers at runtime—for example, when a C program calls malloc, it must decide on which core

ISRs run, where driver code runs, and from which core memory-mapped IO is done. This is all dynamic. This breaks the RMA rules we have studied so far. The collision of SMP (largely developed on non-real-time systems for throughput scaling) and hard real-time systems in particular requires careful and cautious consideration. The combination of real-time services with SMP has been the subject of numerous operating systems and computer architecture research. Outcomes of the research have varied from simply “don’t do this” to “proceed with caution.” We will not attempt to review all of the research, but will prescribe some simple guidelines to help.

First, it is always safe to use SMP on any best-effort system. By definition, we have no deadlines. An SMP RTOS or OS can migrate a task, thread, and process at any time from one CPU core to another, thus causing an unexpected preemption from the RM viewpoint. Furthermore, much like we saw that swapping or page faults in a general OS could cause significant harm to real-time services, this migration of tasks from one processor core to another will add significant interference to the progress of any service that is disrupted as such between release and completion. In a best-effort system, the migration should not cause a problem, but will delay responses on occasion. The SMP system migrates tasks because it needs to balance the load on the processor cores it manages over time. The migration is triggered by noted imbalances as demands for CPU vary on each core over time. One of the simplest algorithms is periodic migration and balancing. If a service starts on core-1, for example, and runs concurrently with two other services, we can do RMA on the three services for that core, but what happens when it migrates? Reduced loading on core-1 is not an issue, but what about increased loading on core-2? In theory, we could apply dynamic admission policies during migration, but that also requires significant resources. However, the OS itself may or may not migrate kernel tasks (recall that in VxWorks, in early editions, all tasks were kernel tasks). Rather than solve what is still a potentially unresolved research challenge, a simpler solution is to use processor core affinity.

9.5 Processor Core Affinity

Even on a non-real-time OS that is SMP, it is sometimes inconvenient to have threads, tasks, or processes migrate. One simple reason is that resources like clocks and timers are sometimes tied to a specific core. What happens if a service reads a clock on core-1 with the purpose of timing an event, the dynamic SMP load balancer migrates tasks on a periodic basis,

and the service then wakes up but now on core-2 and reads the clock again? Maybe nothing fails, as long as the two clocks are synchronized and do not have rollovers, and both have the same base date, but often an interval timer will not meet all of these conditions. So, even in best-effort systems, to ensure determinism, it is a standard feature to set affinity for a task so that it always runs on one and only one processor core.

Let's look at how process core affinity is done in Linux with POSIX threads (similar to a VxWorks kernel task). For this example, we will consider some simple threaded code that can be compiled and run on any Linux SMP system (also available on the DVD).



```
#define _GNU_SOURCE
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

#define NUM_THREADS 64

typedef struct
{
    int threadIdx;
} threadParams_t;

// POSIX thread declarations and scheduling attributes
pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];

void *counterThread(void *threadp)
{
    int sum=0, i, rc;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    for(i=1; i < (threadParams->threadIdx)+1; i++)
        sum=sum+i;

    printf("\nThread idx=%d, sum[0...%d]=%d, running on CPU=%d",
        threadParams->threadIdx,
        threadParams->threadIdx, sum, sched_getcpu());
}
```

```

int main (int argc, char *argv[])
{
    int rc;
    int i;
    cpu_set_t cpuset;
    pthread_t mythread;

    mythread = pthread_self();

    rc = pthread_getaffinity_np(mythread, sizeof(cpu_set_t),
                                &cpuset);

    if (rc != 0)
        perror("pthread_getaffinity_np");

    printf("CPU mask includes:");
    for (i = 0; i < CPU_SETSIZE; i++)
        if (CPU_ISSET(i, &cpuset))
            printf(" %d", i);
    printf("\n");

    for(i=0; i < NUM_THREADS; i++)
    {
        threadParams[i].threadIdx=i;

        pthread_create(&threads[i],    // pointer to thread
                        descriptor
                        (void *)0,      // use default attributes
                        counterThread, // thread function entry
                        point
                        (void *)&(threadParams[i]) // parameters
                                                to pass
                        );
    }

    for(i=0;i<NUM_THREADS;i++)
        pthread_join(threads[i], NULL);

    printf("\nTEST COMPLETE\n");
}

```

If you run “make” on this code on a Linux system and run it, it will simply compute a sum of the digits based on the thread number it has been assigned. Generally, this runs in expected order other than differences due to interrupts, interference, and thread migrations if the system has a high background load that interferes and competes for resources. If you run it on your system with a high background load, you will occasionally see ordering differences, but in general, with SMP load balancing, the Linux OS chooses the least-loaded CPU core for each thread and the simple threads in this example run to completion, with best effort, on the CPU core that Linux assigns. For example, when running on a general-purpose Linux system shared with numerous students, here’s what one of the authors observed for 64 threads that could be scheduled to one of eight CPU cores—that is, four actual, eight when including VCPUs (Virtual CPUs) provided by Intel hyperthreading:

```
%make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
%./pthread
CPU mask includes: 0 1 2 3 4 5 6 7

Thread idx=0, sum[0...0]=0, running on CPU=5
Thread idx=1, sum[0...1]=1, running on CPU=2
Thread idx=2, sum[0...2]=3, running on CPU=6
Thread idx=3, sum[0...3]=6, running on CPU=2
Thread idx=4, sum[0...4]=10, running on CPU=5
Thread idx=5, sum[0...5]=15, running on CPU=5
Thread idx=6, sum[0...6]=21, running on CPU=6
Thread idx=7, sum[0...7]=28, running on CPU=5
Thread idx=8, sum[0...8]=36, running on CPU=6
Thread idx=9, sum[0...9]=45, running on CPU=5
Thread idx=10, sum[0...10]=55, running on CPU=6
Thread idx=11, sum[0...11]=66, running on CPU=5
Thread idx=12, sum[0...12]=78, running on CPU=5
Thread idx=13, sum[0...13]=91, running on CPU=2
Thread idx=14, sum[0...14]=105, running on CPU=5
Thread idx=15, sum[0...15]=120, running on CPU=2
Thread idx=16, sum[0...16]=136, running on CPU=2
Thread idx=17, sum[0...17]=153, running on CPU=5
Thread idx=18, sum[0...18]=171, running on CPU=2
Thread idx=19, sum[0...19]=190, running on CPU=5
Thread idx=20, sum[0...20]=210, running on CPU=2
```

```
Thread idx=21, sum[0...21]=231, running on CPU=5
Thread idx=22, sum[0...22]=253, running on CPU=2
Thread idx=23, sum[0...23]=276, running on CPU=5
Thread idx=24, sum[0...24]=300, running on CPU=2
Thread idx=25, sum[0...25]=325, running on CPU=5
Thread idx=26, sum[0...26]=351, running on CPU=2
Thread idx=27, sum[0...27]=378, running on CPU=5
Thread idx=28, sum[0...28]=406, running on CPU=6
Thread idx=29, sum[0...29]=435, running on CPU=5
Thread idx=30, sum[0...30]=465, running on CPU=5
Thread idx=31, sum[0...31]=496, running on CPU=6
Thread idx=32, sum[0...32]=528, running on CPU=5
Thread idx=33, sum[0...33]=561, running on CPU=6
Thread idx=34, sum[0...34]=595, running on CPU=5
Thread idx=35, sum[0...35]=630, running on CPU=6
Thread idx=36, sum[0...36]=666, running on CPU=5
Thread idx=37, sum[0...37]=703, running on CPU=5
Thread idx=38, sum[0...38]=741, running on CPU=2
Thread idx=39, sum[0...39]=780, running on CPU=5
Thread idx=40, sum[0...40]=820, running on CPU=5
Thread idx=41, sum[0...41]=861, running on CPU=2
Thread idx=42, sum[0...42]=903, running on CPU=5
Thread idx=43, sum[0...43]=946, running on CPU=2
Thread idx=44, sum[0...44]=990, running on CPU=5
Thread idx=45, sum[0...45]=1035, running on CPU=2
Thread idx=46, sum[0...46]=1081, running on CPU=5
Thread idx=47, sum[0...47]=1128, running on CPU=6
Thread idx=48, sum[0...48]=1176, running on CPU=5
Thread idx=49, sum[0...49]=1225, running on CPU=6
Thread idx=50, sum[0...50]=1275, running on CPU=5
Thread idx=51, sum[0...51]=1326, running on CPU=2
Thread idx=52, sum[0...52]=1378, running on CPU=5
Thread idx=53, sum[0...53]=1431, running on CPU=2
Thread idx=54, sum[0...54]=1485, running on CPU=5
Thread idx=55, sum[0...55]=1540, running on CPU=2
Thread idx=56, sum[0...56]=1596, running on CPU=5
Thread idx=57, sum[0...57]=1653, running on CPU=5
Thread idx=58, sum[0...58]=1711, running on CPU=2
Thread idx=59, sum[0...59]=1770, running on CPU=5
Thread idx=60, sum[0...60]=1830, running on CPU=2
Thread idx=61, sum[0...61]=1891, running on CPU=6
Thread idx=62, sum[0...62]=1953, running on CPU=5
```

```
Thread idx=63, sum[0...63]=2016, running on CPU=5
TEST COMPLETE
%
```

Examining the foregoing output, we see that all of the threads are allocated to CPU 2, 5, or 6, with none allocated to CPU 0, 1, 3, 4, or 7. The system in this case just selected the least-loaded CPU cores to handle the new processing requested as it is made. This makes it difficult to predict where a thread will be allocated, and each run will produce a new and different mapping. The mapping is not deterministic or repeatable for the purpose of verification. Without special configuration of Linux for real-time extensions and careful use of POSIX extensions for real-time (discussed in Chapter 11), this varying allocation of threads to CPU cores is to be expected. We do not really care when using a best-effort non-RT OS how the work gets done exactly, but rather that it all gets done as fast as possible on average and that a new load is distributed to the least busy CPU core. Note that Linux provides a nice utility “lscpu” or “cat /proc/cpuinfo,” which allows users to interactively examine the number of SMP CPU cores managed. On the author’s system there are four cores on a single CPU, each with hyperthread support, which Linux presents as eight VCPUs. Hyperthreading is a form of sub-core-level parallelism within the ALU that allows for the issue of instructions at the same time (multiple issue of instructions at the same time is often referred to as “superscalar”) as well as providing support at this level for thread concurrency and context management. This type of system in an ideal scenario is as good as having eight cores, but pipeline hazards and resource contention can cause it to degrade to the equivalent of four cores. Hyperthreading likewise introduces non-determinism to the system, but does increase average throughput for thread pools waiting for CPU core resources.

If we want thread-to-CPU-core-mapping determinism, we can provide this on most SMP systems by using affinity specification, which tells the SMP operating system to dispatch a thread to a specific core or set of cores. Although this is likely to reduce throughput on average and defeat the load balancing, it essentially creates services that are mapped to cores much like AMP would provide. If we run the foregoing code, now modified for CPU core affinity on the same system, here is the result:

```
%./pthread
running on CPU=5, CPUs = 0 1 2 3 4 5 6 7
```

```

Thread idx=0, sum[0...0]=0, running on CPU=0, CPUs = 0
Thread idx=2, sum[0...2]=3, running on CPU=2, CPUs = 2
Thread idx=1, sum[0...1]=1, running on CPU=1, CPUs = 1
Thread idx=3, sum[0...3]=6, running on CPU=3, CPUs = 3
Thread idx=4, sum[0...4]=10, running on CPU=4, CPUs = 4
Thread idx=6, sum[0...6]=21, running on CPU=6, CPUs = 6
Thread idx=7, sum[0...7]=28, running on CPU=7, CPUs = 7
Thread idx=8, sum[0...8]=36, running on CPU=0, CPUs = 0
Thread idx=9, sum[0...9]=45, running on CPU=1, CPUs = 1
Thread idx=10, sum[0...10]=55, running on CPU=2, CPUs = 2
Thread idx=11, sum[0...11]=66, running on CPU=3, CPUs = 3
Thread idx=12, sum[0...12]=78, running on CPU=4, CPUs = 4
Thread idx=5, sum[0...5]=15, running on CPU=5, CPUs = 5
Thread idx=13, sum[0...13]=91, running on CPU=5, CPUs = 5
Thread idx=15, sum[0...15]=120, running on CPU=7, CPUs = 7
Thread idx=16, sum[0...16]=136, running on CPU=0, CPUs = 0
Thread idx=17, sum[0...17]=153, running on CPU=1, CPUs = 1
Thread idx=18, sum[0...18]=171, running on CPU=2, CPUs = 2
Thread idx=19, sum[0...19]=190, running on CPU=3, CPUs = 3
Thread idx=14, sum[0...14]=105, running on CPU=6, CPUs = 6
Thread idx=20, sum[0...20]=210, running on CPU=4, CPUs = 4
Thread idx=22, sum[0...22]=253, running on CPU=6, CPUs = 6
Thread idx=23, sum[0...23]=276, running on CPU=7, CPUs = 7
Thread idx=24, sum[0...24]=300, running on CPU=0, CPUs = 0
Thread idx=25, sum[0...25]=325, running on CPU=1, CPUs = 1
Thread idx=26, sum[0...26]=351, running on CPU=2, CPUs = 2
Thread idx=27, sum[0...27]=378, running on CPU=3, CPUs = 3
Thread idx=28, sum[0...28]=406, running on CPU=4, CPUs = 4
Thread idx=30, sum[0...30]=465, running on CPU=6, CPUs = 6
Thread idx=31, sum[0...31]=496, running on CPU=7, CPUs = 7
Thread idx=21, sum[0...21]=231, running on CPU=5, CPUs = 5
Thread idx=29, sum[0...29]=435, running on CPU=5, CPUs = 5
Thread idx=33, sum[0...33]=561, running on CPU=1, CPUs = 1
Thread idx=34, sum[0...34]=595, running on CPU=2, CPUs = 2
Thread idx=32, sum[0...32]=528, running on CPU=0, CPUs = 0
Thread idx=35, sum[0...35]=630, running on CPU=3, CPUs = 3
Thread idx=37, sum[0...37]=703, running on CPU=5, CPUs = 5
Thread idx=38, sum[0...38]=741, running on CPU=6, CPUs = 6
Thread idx=39, sum[0...39]=780, running on CPU=7, CPUs = 7
Thread idx=36, sum[0...36]=666, running on CPU=4, CPUs = 4
Thread idx=40, sum[0...40]=820, running on CPU=0, CPUs = 0
Thread idx=41, sum[0...41]=861, running on CPU=1, CPUs = 1

```

```

Thread idx=42, sum[0...42]=903, running on CPU=2, CPUs = 2
Thread idx=43, sum[0...43]=946, running on CPU=3, CPUs = 3
Thread idx=44, sum[0...44]=990, running on CPU=4, CPUs = 4
Thread idx=46, sum[0...46]=1081, running on CPU=6, CPUs = 6
Thread idx=47, sum[0...47]=1128, running on CPU=7, CPUs = 7
Thread idx=48, sum[0...48]=1176, running on CPU=0, CPUs = 0
Thread idx=49, sum[0...49]=1225, running on CPU=1, CPUs = 1
Thread idx=50, sum[0...50]=1275, running on CPU=2, CPUs = 2
Thread idx=45, sum[0...45]=1035, running on CPU=5, CPUs = 5
Thread idx=51, sum[0...51]=1326, running on CPU=3, CPUs = 3
Thread idx=53, sum[0...53]=1431, running on CPU=5, CPUs = 5
Thread idx=54, sum[0...54]=1485, running on CPU=6, CPUs = 6
Thread idx=52, sum[0...52]=1378, running on CPU=4, CPUs = 4
Thread idx=55, sum[0...55]=1540, running on CPU=7, CPUs = 7
Thread idx=56, sum[0...56]=1596, running on CPU=0, CPUs = 0
Thread idx=57, sum[0...57]=1653, running on CPU=1, CPUs = 1
Thread idx=58, sum[0...58]=1711, running on CPU=2, CPUs = 2
Thread idx=59, sum[0...59]=1770, running on CPU=3, CPUs = 3
Thread idx=60, sum[0...60]=1830, running on CPU=4, CPUs = 4
Thread idx=62, sum[0...62]=1953, running on CPU=6, CPUs = 6
Thread idx=63, sum[0...63]=2016, running on CPU=7, CPUs = 7
Thread idx=61, sum[0...61]=1891, running on CPU=5, CPUs = 5
TEST COMPLETE
%
```

The fact that thread ordering of 0,2,1,3,4,6 ... up to 11,12,5,13 was out of expected order is annoying, but the results are still correct since we have no deadline or ordering required for the computations. We could use semaphores or other standard synchronization methods if the printed order bothers us or causes an issue with expected results. The author ran this code on a loaded four-CPU core Linux SMP system, and on an SMP system with background load (student processes). When we map to specific cores, they may already be pretty busy with unknown arbitrary workload. On the other hand, the mapping is deterministic. So, how would we fix this issue of unknown interference? The answer is to also make the threads map onto a Linux real-time scheduling class to preempt threads run by the CFS (Completely Fair Scheduler) and therefore ensure that real-time threads always win this competition for specific cores. Updating the code to use both affinity and the POSIX FIFO (First In, First Out) scheduling class (which we must run with root privilege in Linux), which preempts all CFQ mapped threads from user space, we get deterministic core mapping and much less interference in this real-time class. Here is the updated code:


```

#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sched.h>

#define NUM_THREADS 64
#define NUM_CPUS 8

typedef struct
{
    int threadIdx;
} threadParams_t;

// POSIX thread declarations and scheduling attributes
//
pthread_t threads[NUM_THREADS];
pthread_t mainthread;
pthread_t startthread;
threadParams_t threadParams[NUM_THREADS];

pthread_attr_t fifo_sched_attr;
pthread_attr_t orig_sched_attr;
struct sched_param fifo_param;

#define SCHED_POLICY SCHED_FIFO
#define MAX_ITERATIONS (1000000)

void print_scheduler(void)
{
    int schedType = sched_getscheduler(getpid());

    switch(schedType)
    {
        case SCHED_FIFO:
            printf("Pthread policy is SCHED_FIFO\n");
            break;
    }
}

```

```

        case SCHED_OTHER:
            printf("Pthread policy is SCHED_OTHER\n");
            break;
        case SCHED_RR:
            printf("Pthread policy is SCHED_RR\n");
            break;
        default:
            printf("Pthread policy is UNKNOWN\n");
    }
}

void set_scheduler(void)
{
    int max_prio, scope, rc, cpuidx;
    cpu_set_t cpuset;

    printf("INITIAL "); print_scheduler();

    pthread_attr_init(&fifo_sched_attr);
    pthread_attr_setinheritsched(&fifo_sched_attr,
                                PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&fifo_sched_attr,
                                SCHED_POLICY);

    CPU_ZERO(&cpuset);
    cpuidx=3;
    CPU_SET(cpuidx, &cpuset);
    pthread_attr_setaffinity_np(&fifo_sched_attr,
                                sizeof(cpu_set_t), &cpuset);

    max_prio=sched_get_priority_max(SCHED_POLICY);
    fifo_param.sched_priority=max_prio;

    if((rc=sched_setscheduler(getpid(), SCHED_POLICY,
                              &fifo_param)) < 0)
        perror("sched_setscheduler");

    pthread_attr_setschedparam(&fifo_sched_attr, &fifo_param);

    printf("ADJUSTED "); print_scheduler();
}

```

```

void *counterThread(void *threadp)
{
    int sum=0, i, rc, iterations;
    threadParams_t *threadParams = (threadParams_t *)threadp;
    pthread_t mythread;
    double start=0.0, stop=0.0;
    struct timeval startTime, stopTime;

    gettimeofday(&startTime, 0);
    start = ((startTime.tv_sec * 1000000.0) +
             startTime.tv_usec)/1000000.0;

    for(iterations=0; iterations < MAX_ITERATIONS; iterations++)
    {
        sum=0;
        for(i=1; i < (threadParams->threadIdx)+1; i++)
            sum=sum+i;
    }

    gettimeofday(&stopTime, 0);
    stop = ((stopTime.tv_sec * 1000000.0) +
            stopTime.tv_usec)/1000000.0;

    printf("\nThread idx=%d, sum[0...%d]=%d, running on CPU=%d,
           start=%lf, stop=%lf",
           threadIdx,
           threadIdx, sum, sched_getcpu(),
           start, stop);
}

void *starterThread(void *threadp)
{
    int i, rc;

    printf("starter thread running on CPU=%d\n", sched_getcpu());

    for(i=0; i < NUM_THREADS; i++)
    {
        threadParams[i].threadIdx=i;
    }
}

```

```

        pthread_create(&threads[i],    // pointer to thread
                        descriptor
                        &fifo_sched_attr, // use FIFO RT max
                                           priority attributes
                        counterThread, // thread function entry
                                           point
                        (void *)&(threadParams[i]) // parameters
                                                    to pass in
        );

    }

    for(i=0;i<NUM_THREADS;i++)
        pthread_join(threads[i], NULL);

}

int main (int argc, char *argv[])
{
    int rc;
    int i, j;
    cpu_set_t cpuset;

    set_scheduler();

    CPU_ZERO(&cpuset);

    // get affinity set for main thread
    mainthread = pthread_self();

    // Check the affinity mask assigned to the thread
    rc = pthread_getaffinity_np(mainthread, sizeof(cpu_set_t),
                                &cpuset);

    if (rc != 0)
        perror("pthread_getaffinity_np");
    else
    {
        printf("main thread running on CPU=%d, CPUs =",
               sched_getcpu());

        for (j = 0; j < CPU_SETSIZE; j++)

```

```

        if (CPU_ISSET(j, &cpuset))
            printf(" %d", j);

    printf("\n");
}

pthread_create(&startthread,    // pointer to thread descriptor
               &fifo_sched_attr, // use FIFO RT max
                                   priority attributes
               starterThread, // thread function entry point
               (void *)0 // parameters to pass in
               );

pthread_join(startthread, NULL);

printf("\nTEST COMPLETE\n");
}

```

Here are the deterministically mapped and much better ordered results (without resorting to semaphores). The demonstration simply shows that both deterministic mapping of each thread to a single core (or specific set of cores) and priority-preemptive run-to-completion scheduling are needed in conjunction for predictable response.

```

siewerts@asgard:~/se300/simplethread-affinity$ sudo ./pthread
INITIAL Pthread policy is SCHED_OTHER
ADJUSTED Pthread policy is SCHED_FIFO
main thread running on CPU=5, CPUs = 0 1 2 3 4 5 6 7
starter thread running on CPU=3

```

```

Thread idx=0, sum[0...0]=0, running on CPU=3,
start=1421223584.501278, stop=1421223584.507752
Thread idx=1, sum[0...1]=1, running on CPU=3,
start=1421223584.507829, stop=1421223584.513027
Thread idx=2, sum[0...2]=3, running on CPU=3,
start=1421223584.513228, stop=1421223584.518455
Thread idx=3, sum[0...3]=6, running on CPU=3,
start=1421223584.518468, stop=1421223584.525631
Thread idx=4, sum[0...4]=10, running on CPU=3,
start=1421223584.525645, stop=1421223584.534846
Thread idx=5, sum[0...5]=15, running on CPU=3,
start=1421223584.534862, stop=1421223584.545885
Thread idx=6, sum[0...6]=21, running on CPU=3,

```

```

start=1421223584.545901, stop=1421223584.558476
Thread idx=7, sum[0...7]=28, running on CPU=3,
start=1421223584.558489, stop=1421223584.572696
Thread idx=8, sum[0...8]=36, running on CPU=3,
start=1421223584.572712, stop=1421223584.588384
Thread idx=9, sum[0...9]=45, running on CPU=3,
start=1421223584.588395, stop=1421223584.605403
Thread idx=10, sum[0...10]=55, running on CPU=3,
start=1421223584.605414, stop=1421223584.623617
Thread idx=11, sum[0...11]=66, running on CPU=3,
start=1421223584.623627, stop=1421223584.642985
Thread idx=12, sum[0...12]=78, running on CPU=3,
start=1421223584.642996, stop=1421223584.663554
Thread idx=13, sum[0...13]=91, running on CPU=3,
start=1421223584.663563, stop=1421223584.685920
Thread idx=14, sum[0...14]=105, running on CPU=3,
start=1421223584.685929, stop=1421223584.710600
Thread idx=15, sum[0...15]=120, running on CPU=3,
start=1421223584.710610, stop=1421223584.737067
Thread idx=16, sum[0...16]=136, running on CPU=3,
start=1421223584.737077, stop=1421223584.764973
Thread idx=17, sum[0...17]=153, running on CPU=3,
start=1421223584.764984, stop=1421223584.795950
Thread idx=18, sum[0...18]=171, running on CPU=3,
start=1421223584.795959, stop=1421223584.828314
Thread idx=19, sum[0...19]=190, running on CPU=3,
start=1421223584.828324, stop=1421223584.862941
Thread idx=20, sum[0...20]=210, running on CPU=3,
start=1421223584.862950, stop=1421223584.899920
Thread idx=21, sum[0...21]=231, running on CPU=3,
start=1421223584.899929, stop=1421223584.938815
Thread idx=22, sum[0...22]=253, running on CPU=3,
start=1421223584.938823, stop=1421223584.979676
Thread idx=23, sum[0...23]=276, running on CPU=3,
start=1421223584.979685, stop=1421223585.023486
Thread idx=24, sum[0...24]=300, running on CPU=3,
start=1421223585.023496, stop=1421223585.068794
Thread idx=25, sum[0...25]=325, running on CPU=3,
start=1421223585.068804, stop=1421223585.116124
Thread idx=26, sum[0...26]=351, running on CPU=3,
start=1421223585.116154, stop=1421223585.164839
Thread idx=27, sum[0...27]=378, running on CPU=3,

```

```

start=1421223585.164849, stop=1421223585.216194
Thread idx=28, sum[0...28]=406, running on CPU=3,
start=1421223585.216204, stop=1421223585.269694
Thread idx=29, sum[0...29]=435, running on CPU=3,
start=1421223585.269705, stop=1421223585.325550
Thread idx=30, sum[0...30]=465, running on CPU=3,
start=1421223585.325559, stop=1421223585.383871
Thread idx=31, sum[0...31]=496, running on CPU=3,
start=1421223585.383880, stop=1421223585.444075
Thread idx=32, sum[0...32]=528, running on CPU=3,
start=1421223585.444084, stop=1421223585.506471
Thread idx=33, sum[0...33]=561, running on CPU=3,
start=1421223585.506480, stop=1421223585.584472
Thread idx=34, sum[0...34]=595, running on CPU=3,
start=1421223585.584484, stop=1421223585.651034
Thread idx=35, sum[0...35]=630, running on CPU=3,
start=1421223585.651045, stop=1421223585.719421
Thread idx=36, sum[0...36]=666, running on CPU=3,
start=1421223585.719433, stop=1421223585.791845
Thread idx=37, sum[0...37]=703, running on CPU=3,
start=1421223585.791855, stop=1421223585.866582
Thread idx=38, sum[0...38]=741, running on CPU=3,
start=1421223585.866591, stop=1421223585.956924
Thread idx=39, sum[0...39]=780, running on CPU=3,
start=1421223585.956933, stop=1421223586.051432
Thread idx=40, sum[0...40]=820, running on CPU=3,
start=1421223586.051442, stop=1421223586.147357
Thread idx=41, sum[0...41]=861, running on CPU=3,
start=1421223586.147366, stop=1421223586.243549
Thread idx=42, sum[0...42]=903, running on CPU=3,
start=1421223586.243558, stop=1421223586.342285
Thread idx=43, sum[0...43]=946, running on CPU=3,
start=1421223586.342295, stop=1421223586.443378
Thread idx=44, sum[0...44]=990, running on CPU=3,
start=1421223586.443389, stop=1421223586.544770
Thread idx=45, sum[0...45]=1035, running on CPU=3,
start=1421223586.544779, stop=1421223586.646062
Thread idx=46, sum[0...46]=1081, running on CPU=3,
start=1421223586.646071, stop=1421223586.752033
Thread idx=47, sum[0...47]=1128, running on CPU=3,
start=1421223586.752043, stop=1421223586.859080
Thread idx=48, sum[0...48]=1176, running on CPU=3,

```

```

start=1421223586.859089, stop=1421223586.967936
Thread idx=49, sum[0...49]=1225, running on CPU=3,
start=1421223586.967947, stop=1421223587.079380
Thread idx=50, sum[0...50]=1275, running on CPU=3,
start=1421223587.079392, stop=1421223587.194301
Thread idx=51, sum[0...51]=1326, running on CPU=3,
start=1421223587.194311, stop=1421223587.309774
Thread idx=52, sum[0...52]=1378, running on CPU=3,
start=1421223587.309785, stop=1421223587.427783
Thread idx=53, sum[0...53]=1431, running on CPU=3,
start=1421223587.427792, stop=1421223587.545300
Thread idx=54, sum[0...54]=1485, running on CPU=3,
start=1421223587.545310, stop=1421223587.665436
Thread idx=55, sum[0...55]=1540, running on CPU=3,
start=1421223587.665446, stop=1421223587.789663
Thread idx=56, sum[0...56]=1596, running on CPU=3,
start=1421223587.789673, stop=1421223587.914456
Thread idx=57, sum[0...57]=1653, running on CPU=3,
start=1421223587.914465, stop=1421223588.041989
Thread idx=58, sum[0...58]=1711, running on CPU=3,
start=1421223588.041998, stop=1421223588.171214
Thread idx=59, sum[0...59]=1770, running on CPU=3,
start=1421223588.171224, stop=1421223588.302179
Thread idx=60, sum[0...60]=1830, running on CPU=3,
start=1421223588.302189, stop=1421223588.436144
Thread idx=61, sum[0...61]=1891, running on CPU=3,
start=1421223588.436153, stop=1421223588.570589
Thread idx=62, sum[0...62]=1953, running on CPU=3,
start=1421223588.570599, stop=1421223588.708680
Thread idx=63, sum[0...63]=2016, running on CPU=3,
start=1421223588.708691, stop=1421223588.848359
TEST COMPLETE

```

In Chapter 11, we further attempt to get a predictable response out of mainline Linux (official Linux Foundation releases of the kernel found on kernel.org) and investigate the time jitter of service requests for the FIFO scheduling class mapped to specific processor resources. The remaining issues we have to deal with in the mainline Linux SMP system include: (a) sections of the Linux kernel that are not preemptible by the FIFO real-time threads, (b) interference from interrupt servicing, (c) issues with NUMA asymmetry in memory access, and (d) blocking of threads needing resources other than the CPU. Linux has recently had significant kernel upgrades

to make the kernel itself, kernel tasks, ISR kernel support, and the system in general more preemptible by user threads. More detailed discussion on how to configure Linux for the best real-time response is covered in Chapter 11. In Chapter 11, we also look at more interesting and compelling examples using computer vision applications that produce significant CPU loading on multi-core systems, like the NVIDIA Jetson multi-core ARM and Intel multi-core x86.

9.6 Future Directions for RTOS

Hard real-time SMP is an open research project; however, two trends are developing. First, as we will discuss more in Chapter 11, extensions to and variations on the mainline Linux kernel are supporting both SMP and real-time scheduling [RedHawk], [ARTiS], [RTAI]. The position of the Linux Foundation, the ultimate gatekeepers of the official Linux kernel, is that Linux supports soft real-time, which fits with both authors' experience as well, based on numerous comparisons of RTOS and Linux with or without various patches [FoundRTI]. The viability of Linux for soft real-time service implementation, of course, has changed over time since Linux has undergone significant kernel-level changes, including the scheduler and to support SMP. At the same time, while Linux is being improved by both the Linux Foundation and third-party distributors of Linux tailored for real-time systems, RTOS vendors have started to integrate and support SMP [VxWorksSMP], [RTEMS]. Either way, the authors recommend that developers carefully verify ability to meet deadlines with predictable response for soft real-time and deterministic response for hard real-time. It should likewise be noted that RMA is complicated by task migration for load balancing, ISR mapping, and general resource asymmetry, especially in NUMA SMP configurations. Whether patches and configurations are made to increase the predictability of Linux for real-time services or an RTOS is configured to support SMP, the developer should exercise caution to ensure that deadlines are met. Often, this means leveraging features such as processor core affinity for a subset of real-time services such that the result is really a mixed real-time and non-real-time system where the true benefits of SMP are really only used for the non-real-time services. Support for mixed real-time and non-real-time is a significant future direction for both RTOS and extensions to Linux.

Another approach to mixing both real-time and non-real-time services on one system is to make use of a hypervisor. Again, the authors caution

the reader to carefully verify ability to meet deadlines with predictable or deterministic response as required in this type of configuration since the hypervisor can add overhead and complexity to scheduling and resource allocation analysis. Two types of hypervisors exist: Type 1 runs directly on the CPU and supports multiple OS instances; Type 2 runs on a base OS that, in turn, can run any number of guest operating systems. A well-known example of a Type 2 hypervisor is Oracle's Virtual Box [VB]. A Type 2 hypervisor multiplexes resource use for each guest OS, and, while very convenient for testing and supporting multiple operating systems on one platform, would not work at all for predictable response. Therefore, RTOS vendors who suggest the use of a hypervisor are suggesting use of a Type 1 hypervisor, where an RTOS runs side-by-side with a traditional OS like Linux [Vx-WorksHV]. In this model, the mapping of each guest OS to resources on a multi-core system is perhaps not much different than AMP where one core runs the RTOS and another core runs the non-real-time OS, like Linux. However, the hypervisor provides some flexibility in this mapping, which is done at runtime rather than one time by design.

9.7 SMP Support Models

To summarize, SMP involves active runtime mapping of processor resources (CPU cores, memory extents, device interfaces, interrupt handling), which has the advantage of dynamically balancing over time as well as eliminating the overhead from multiple instances of the OS itself. If an application has no real-time requirements, SMP is clearly advantageous. If an application has mixed requirements for best-effort (non-real-time) services alongside soft real-time (predictable response) and hard real-time (deterministic response), then it may still be possible to configure an SMP system to support all three classes of service. The safe alternative, proven over time, is AMP, where a subset of CPU nodes are dedicated to real-time service and run a traditional RTOS. Another subset of CPU nodes on the same system could then run an OS like Linux and support soft real-time and best-effort services. Message passing can be used if a real-time service needs to communicate with soft or best-effort services on the other AMP node. This is a safe approach where RMA can be used to verify the AMP RTOS node(s). The ideal solution has one SMP kernel that can support all three classes of service. Such an SMP system must support RMA for hard real-time services and provide real-time synchronization features, such as priority inversion protection for priority-preemptive run-to-

completion tasks. Therefore an SMP system that supports real-time must essentially provide control, such as the processor affinity settings reviewed in this chapter, ISR mapping (available in Linux), and memory access management for NUMA. In theory, if sufficient control and features are provided in an SMP system, it could support multiple classes of service, but the authors recommend caution and rigorous verification (as was true for AMP hard real-time) to ensure that RM policy is upheld in the SMP system for the hard real-time services. Generally, RTOS solutions are adding support for SMP, and traditional SMP solutions, like Linux, are starting to add support for predictable-response service levels, which is explored in more depth in Chapter 11.

9.8 RTOS Hypervisors

The RTOS hypervisor concept makes use of a Type 1 hypervisor that runs directly on the multi-core SoC hardware and, in turn, hosts an RTOS, such as VxWorks, alongside best-effort operating systems, such as Linux, as shown in Figure 9.6.

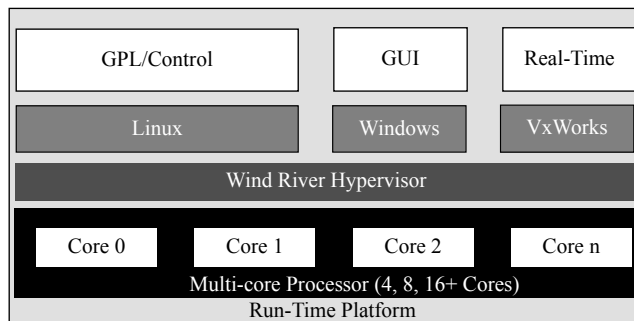


FIGURE 9.6 Wind River RTOS with Hypervisor - <http://www.windriver.com/products/hypervisor/>

In some sense, this configuration is similar to AMP where different operating systems are run on each core, but the hypervisor does a virtual mapping to cores and resources such that Linux, for example, can span multiple cores in an SMP configuration and VxWorks can be mapped to a single core for determinism. Communication between the VxWorks guest RTOS and the Linux best-effort guest OS could again be TIPC or TCP/IP or perhaps could use a hypervisor mechanism. This approach appears simpler than running VxWorks as an SMP kernel, but the authors have not tested this configuration. It appears well worth testing.

Summary

A good mantra in embedded systems is to keep things simple always. As such, if the complexity of SMP or virtual machine RTOS is not required and does not offer significant benefits that far outweigh the added complexity, additional verification, and potential risk, then the best option is to stick with AMP. Multiple CPU cores and message passing are reasonably complex and can lead to serious issues, like deadlock, discussed earlier in this text, along with other forms of unbounded blocking. It does not make sense to incorporate new architectural features just because they are new. SMP used in scalable Linux systems has significant payoff for high-performance computing, but most embedded systems do not share the same need to scale to the same degree. As we will see in Chapter 11, if an embedded system is not hard real-time, but rather best-effort, and embedded Linux provides a favorable approach for reasons we will discuss more in Chapter 11, then SMP might be a good choice, but beware of “feature creep” and simply adding SMP to a design because it is new.

Exercises

1. Download, build, and run the Linux pthread example.
2. Read white papers and documentation you can find on the Internet for “VxWorks AMP.” Describe how, when, and why Wind River recommends using the AMP configuration of their RTOS.
3. Read white papers and documentation you can find on the Internet for “VxWorks SMP”. Describe how, when and why Wind River recommends using the SMP configuration of its RTOS.
4. Read white papers and documentation you can find on the Internet for “VxWorks hypervisor.” Describe how, when, and why Wind River recommends using its hypervisor with VxWorks and other real-time or non-real-time operating systems.
5. Download, build and explain code found at http://mercury.pr.erau.edu/~siewerts/cec450/code/rt_simplethread/

Chapter References

- [ARTiS] <http://www.lifl.fr/west/publi/PMSD05.pdf>, <http://pieleric.free.fr/MSP-D04rr04.pdf>
- [Flynn72] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Transactions Computing*, Vol. C–21, No. 9 (September 1972): pp. 948–960.
- [FoundRTI] <http://www.linuxfoundation.org/programs/rt-linux-initiative>
- [FreeRTOS] <http://www.freertos.org/>
- [LLNL] <https://computing.llnl.gov/tutorials/pthreads/>
- [RCU] <http://www.rdrop.com/~paulmck/RCU/realtimeRCU.2005.04.23a.pdf>
- [RedHawk] <http://real-time.ccur.com/home/products/redhawk-linux>
- [RTAI] <https://www.rtai.org/>
- [RTEMS] <https://www.rtems.org/>
- [SMPRTOS] https://moodle.polymtl.ca/pluginfile.php/81015/course/section/16828/SMP_for_MPSOC.pdf
- [VB] <https://www.virtualbox.org/>
- [VxWorksHV] <http://www.windriver.com/products/product-overviews/wind-river-hypervisor-product-overview.pdf>
- [VxWorksSMP] <http://www.windriver.com/products/vxworks/multi-core.html>

OPEN SOURCE REAL-TIME OPERATING SYSTEMS

In this chapter

- FreeRTOS Alternative to Proprietary RTOS
- FreeRTOS Platform and Tools
- FreeRTOS Real-Time Service Programming Fundamentals

10.1 FreeRTOS Alternative to Proprietary RTOS

This chapter explores the use of an open source RTOS and includes analysis of trade-offs between use of open source and proprietary RTOS such as VxWorks. The chapter looks at cost, debug, features and extensions for open source. Methods of evaluating RTOS platforms are discussed including activity level, code quality, platform alternatives, features and primitives and finally real-time scheduling performance. A multitask experiment used to evaluate VxWorks has been ported to FreeRTOS and results are compared to provide insight into how well an open source RTOS can compare to proprietary. The code tested can be found on the DVD included with the text.



There are a number of options available for a real-time operating system. Up to this point, the focus has been on one of the best proprietary options, VxWorks. This chapter explores the trade-offs associated with using an open source RTOS and provides some guidelines for evaluating open source RTOS platforms. This includes an in-depth analysis of one of the most promising choices available, the FreeRTOS offering.

When trying to select a platform, there are a number of factors to consider. The first consideration is generally cost. Typical proprietary licenses include an up-front license cost, a per-unit cost, or a mix of the two. This cost pays for the engineering that goes into developing the OS and support associated with issues or requests encountered while developing the application. In some cases, the vendor also provides training to help get the project started.

When using an open source operating system, these costs can be replaced by engineering time from the group or individual developing the application. Essentially, the costs can be transferred from direct (out-of-pocket) costs to indirect costs in time from developers to understand, support, and troubleshoot operating system issues. This can make it very important to assess the initial quality of the code and operating system before selecting it and allocating extra developer time for debugging. It is also important to explore whether external support options are available as contract services. This opens the possibility of relying on developers familiar with the platform later if the tasks get larger than expected.

There are some distinct advantages associated with open source platforms, particularly once a development team has built up experience with the code base. The ability to see into the OS functions and scheduler can uncover a number of issues and simplify debugging. With visibility into the kernel and OS functions, the developer can set breakpoints or trace the interactions between the application and OS. It is also a good approach to completely understand the behavior of an API, particularly when called within a real-time task.

New or custom features pose especially tough challenges with proprietary operating systems and may increase the interest in using an open source platform. Most operating system vendors are willing to discuss new features or custom APIs for a particular application. The company supporting the OS may include the development costs in the license or require an additional fee. In some cases—for example, low-volume applications requesting custom changes that are not useful in other contexts—the vendor may choose not to implement the functionality at all or request a higher fee. In these cases, an open source platform provides additional options where the development team can implement the required extensions.

When deciding to pursue an open source implementation of an RTOS, it is important to understand the overall quality and capabilities. There are

some general topics that apply to any open source project and others that are specific to a quality RTOS implementation. Both aspects are important and need to be considered since changes anywhere in the RTOS may cause deadline or timing issues.

To understand the quality of an open source software project, the strongest test is how actively changes are being made and what the quality controls are on those changes. This is started simply enough with a code review and inspection of the overall quality of the project. This extends to understand the project's environment and community. If a project has a good deal of active development, it is more likely to support the features desired for the application. When problems are encountered or the development team has questions, it is much more likely that an active community will provide help and respond to questions. There are likely to be more embedded platforms supported, and the process of adding new targets is usually better understood and documented.

The level of activity needs to be balanced with tight controls if true support for real-time behavior is to be maintained. While a wide variety of developers is beneficial from a support and features perspective, uncontrolled submissions or changes could very easily break the real-time nature of the system. Many common software engineering concepts lead to non-deterministic execution time. All changes need to be carefully evaluated for where these mechanisms are introduced and how critical sections or interrupt locks are employed. In the best arrangements, any changes go through a code review process to ensure these protections are not violated and run through a regression suite to validate the changes.

The most important evaluation of an RTOS quality is the scheduling methodology and the actions taken when OS primitives are invoked. This aspect of the inspection can start with the documentation or project wiki pages; an advantage of the open source nature is that the developers can inspect the functions directly. Starting with the scheduler, there are several red flags to look for in the code. Any "fairness" mechanisms that do not involve tasks at the same priority are a definite problem, though these are less likely to be present in anything described as an RTOS.

There are many other aspects that require closer investigation. One critical aspect to explore is how often and how interrupt locks are used in the code. Locking interrupts prevents the scheduler from running on almost all platforms. This creates important issues if interrupts are ever

locked during a nondeterministic operation, such as linked lists, sorting, or other variable length operations. These are not necessarily unbounded operations; even $O(n)$ or $O(\log(n))$ traversals can create substantial variability in scheduling. These can occur anywhere since the scheduler is generally triggered in an interrupt; any accesses of shared data structures in the kernel generally require interrupt locks. Tracking down all examples of interrupt locks may require some time and effort, but any data structures in the kernel that are shared with the scheduler are a very good place to start.

The evaluation of the FreeRTOS implementation encompasses all of these aspects. The scheduling methodology is a strict priority-preemptive algorithm. To save space in the implementation, the priorities of the ready tasks are set in a mask with the highest bit set for the highest-priority task. The list of ready tasks is kept in an array indexed by priority. Since each priority level may have multiple ready tasks, the tasks at each level are added to a FIFO. This also creates a round-robin behavior among tasks at the same level. Each time slice, the scheduler selects the ready task with the highest priority. If more than one task is ready at a given priority, the next task at that priority is selected and the current task returns to its ready queue.

All of the operations used for task management and status are implemented so that the operations require a deterministic time. Using the mask gives a fast lookup of the next priority level. In some instruction sets, this is even accomplished with a single instruction using a “count leading zeros” operation. With the priority level selected, the task is selected by grabbing the task at the head of the FIFO. Similarly, adding a new task to the ready list requires only setting the appropriate bit and adding that task’s TCB to the end of the FIFO. It is important to understand both sides of the operation, since the act of setting a new task to be ready will require access to the scheduler’s core data structures and will be performed with interrupts disabled.

An empirical validation of the RTOS scheduling performance can also serve as a regression test for any changes. This requires a simple application that starts with a fixed duration function. This fixed processing increment is run in different tasks on periodic timers with the correct rate-monotonic priorities assigned. If any task finds its timer-released semaphore filled when it completes the processing, it has exceeded its deadline. This framework should even include several different primitives supported by the OS.

Several mixes of the task periodicities and processing durations can be run. Each combination should have extremely limited margin available,

however, to create the strictest test possible. Once the task set runs to completion for the least common multiple of the task periods, it should be able to continue indefinitely. By running the task set over a very long duration relative to the task periods, it will uncover any variability that accumulates over time or is introduced by asynchronous operations. This makes it very well-suited to run in conjunction with a coverage test or other mechanisms that exercise the RTOS code.

Based on one of the author's analysis of the FreeRTOS platform, in this author's opinion, FreeRTOS shows that all of the key criteria for a quality RTOS are met in this platform and provides comparable capability of proprietary RTOS's, but the reader is invited to compare for themselves and draw their own conclusions. The FreeRTOS code uses a different style and model than many other projects, which can be slightly difficult to follow at first. The code clearly adheres to this style, however, and has tight controls on the coding standard and how application code is incorporated. Part of this comes from the submission mechanism used, where any updates must be published by a dedicated team that maintains the FreeRTOS code. This does limit the opportunities for additional functionality to be added but provides a clear safeguard to code quality and preservation of real-time performance. Based upon preliminary empirical validation using FreeRTOS examples found on the included DVD, FreeRTOS appears to provide a predictable response scheduler with a variety of programming primitives available to applications.



10.2 FreeRTOS Platform and Tools

This section covers the following topics as outlined here:

Supported Boards

- *How does compilation work with FreeRTOS?*
 - *Comparison of separately loading applications vs. "Giant Blob"*
- *Finding supported boards, ordering*
- *Recognizing differences between boards*
- *Configuring FreeRTOS*
- *Windows Simulator*

Built-In Functionality

- *Debug tools*
 - *Trace*
 - *Debug environments*
- *Networking and connectivity*
 - *LWIP*
 - *Serial*

FreeRTOS+ and Extension Capabilities

- *Nabto, UDP, TCP, CLI, Trace, SSL, IO*

The first step in understanding the FreeRTOS platform is to explore the code organization. Any application that uses FreeRTOS should be segmented into three parts, two from the OS and one from the application. Figure 10.1 shows this hierarchy, with the RTOS implementation above the device-specific porting layer. The application code interacts with the RTOS layer and is completely independent of the specifics of the device porting. The device porting layer is specific to both the board and compiler used since some of the required functionality is generally implemented in assembly code and/or compiler-specific mechanisms. This layering is important to understand since FreeRTOS does not support dynamic application loading. Instead all of the code is compiled in a single project and the application, RTOS, and device layers are all statically linked.

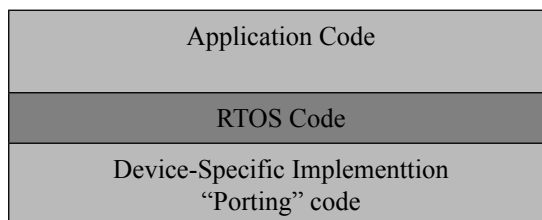


Figure 10.1 FreeRTOS Layers

This layering is also important for configuring the build environment and interpreting the FreeRTOS code as it is downloaded. The distribution of FreeRTOS includes all of the current device- and compiler-specific code required for all of the supported boards. This code can be found in the *portable* directory of the download. The RTOS code itself is in the root of the

Source directory, with all of the headers needed for the application in the *include* directory [FreeRTOS Organization].

The primary means for selecting a board and loading applications is through the compiler project settings. Include the code for the correct platform with the general OS implementation code in the build project, along with the application source to create a FreeRTOS application. Depending on the board and the tools required, this may take on a number of different forms. Each board-specific wiki includes details on which compilers are supported and how the project is configured. The wiki for each supported board also includes ordering information for development boards used with the example application. When ordering the development board, take care to ensure the selected toolchain matches the tools used with the port of FreeRTOS.

When evaluating a candidate board it is important to understand the functionality available in the port for that board. The official ports include the base functionality of the OS primitives and configurations. There are a number of additional capabilities that may or may not be present for any particular board. There are some power-saving features discussed later in the chapter, for example, that have varying levels of support in the different boards. Some of this variation is driven by the hardware functions provided by the processors or development boards. There are some limitations, such as code size, imposed by the tools selected for the ports. Carefully review the descriptions provided in the port and example application to understand the capabilities and limitations of each board [FreeRTOS Ports].

Once a board is selected, FreeRTOS provides several configuration options that should be explored and set to the behavior required by the application. The settings are included in the *FreeRTOSConfig.h* provided with the port to the board selected. This will require some understanding of the board and the software design of the application, so there may be changes later in the process if the design has not fully stabilized. Some of the more important settings to consider are listed here.

- `configUSE_PREEMPTION`: This flag should be set and will be required for the majority of real-time applications. This enables task preemption once a higher-priority task is ready to run. The alternative is cooperative multitasking.

- `configUSE_IDLE_HOOK` and `configUSE_TICK_HOOK`: These are settings that enable application tailoring for the use of slack time (idle) and software clocks.
- `configTICK_RATE_HZ`: This determines how often the timer that triggers the scheduler will run.
- `configMAX_PRIORITIES`: Determines how many unique priorities can be allocated to the tasks in the application.
- `configUSE_TRACE_FACILITY` and `configCHECK_FOR_STACK_OVERFLOW`: These enable the debug and protection functionality.
- `configUSE_MUTEXES`, `configUSE_RECURSIVE_MUTEXES`, and `configUSE_COUNTING_SEMAPHORES`: These enable the different OS primitives.
- There are several APIs that can be used within the application configurable to save space. If these are not required and space is an issue, these can be used to disable the APIs. Many of them are enabled by default.

10.3 FreeRTOS Real-Time Service Programming Fundamentals

This section covers the following FreeRTOS core features as outlined here:

Tasks

- *Ignore the co-routines*

OS Primitives

- *FromISR versions and what they do different*
- *Types of Control/protection and when to use them*
 - *Binary semaphores, Counting semaphores, Mutex*

Important aspects to consider for task and operating system primitives in any RTOS include: using a timeout on the give can cause a number of side effects. Certainly the task will have some additional delay for the timeout. Failing to give (from a full queue or a set semaphore) will cause the task to go into more critical sections, trying over and over again to set the semaphore. This will also cause the scheduler to suspend and resume, potentially adding a small amount of additional overhead on the scheduler.

Since the resume checks if tasks were readied, it is only the duration of the check for empty.

Similarly, using binary semaphores will cause the additional checks and behaviors if the semaphore is already full. For semaphores that may be set more than once before servicing, the implementation is much cleaner using a counting semaphore.

Additional features of FreeRTOS considered include:

Timers

- *Effects of timer wake-up compared to delays*

Memory

Advanced topics

- *Idle hook*
- *Low power modes*
- *Deferred interrupts*
- *Memory and stack protections*
 - *Malloc fail hook*
 - *Stack overflow hook*
 - *MPU and restricted tasks*

In every operating system, there is a difference between code that is well written for the resources available and code that is less tailored and has unintended effects. This section provides some guidelines for how to use the primitives provided by FreeRTOS. If there is any ambiguity or new technique required for a particular application, the best advice is always to thoroughly understand the inner workings of the functions available in the API. The following analyses can serve as examples for problems and pitfalls to evaluate when selecting the appropriate resource for any particular application.

There are two mechanisms that can be used in FreeRTOS to create contexts for processing, tasks, and co-routines. The concepts related to pre-emptive multitasking discussed in this text apply to the tasks. Since the co-routines do not follow the same rules as tasks and are primarily designed to allow for incredibly limited memory footprints, they are not recommend-

ed for use. This is also reflected by the developers supporting FreeRTOS who are no longer adding new functionality to co-routines. Ignoring the co-routines simplifies the application as well as the API investigation since different OS structures are required for co-routines and tasks [FreeRTOS Tasks].

FreeRTOS offers a number of different OS primitives that can be used to signal tasks, synchronize processing activities, and distribute processing load between tasks. These primitives include:

- Queues
- Binary Semaphores
- Counting Semaphores
- Mutexes
- Recursive Mutexes

Each primitive has a function that it performs well and may have unintended consequences when used improperly. Think of these as the tools provided by the OS to build a new application. Selecting the correct tool is an important part of building a stable and reliable application.

The first and most important distinction between types of primitives is the calling context. Each mechanism provided by the operating system has two variations, the one called by a task and one from interrupt context. It is important to use these appropriately since the assumptions made by each are very different. The source of the differences comes from the different requirements for interactions from an ISR and from a traditional task. The execution of an ISR will block other interrupts in a pending state until it completes, so a critical section is not needed. Another key difference is ordering of the context switch if one is required. When called from within a task, the context switch is triggered through a yield if a higher-priority task was activated by the transaction. From an ISR, there is a slight different mechanism where an asynchronous scheduler interrupt is triggered after the current ISR returns. The net effect is the same in either case, but the code required to achieve the result is very different.

The other general variation of the primitives used by tasks is the *Alternative API*. The distinction between the common versions and the *Alternative API* is the approach to critical sections. Within the Alt definitions of the functions, the entire function is encapsulated within the critical section. In

the regular implementation, more limited portions of the function are included in the critical section. The trade-off between these two mechanisms is runtime versus interrupt variability. While the net increase in the time spent in critical sections can be small, the *Alternative API* versions of the functions are not typically recommended. The potential exception would be the highest-priority task in the system since in that case execution time is likely the driving factor. As a general approach, increasing the variability of the interrupt and scheduler timing creates a system with which it is much more difficult to guarantee that deadlines are met.

The core structure of all of the OS primitives is the queue implementation. The binary and counting semaphores directly use a queue with predefined settings. These each use a queue with elements of size 0. The binary semaphore further reduces the size of the queue to a single element. Otherwise, all of the descriptions of the queue operations and behaviors directly apply. There is some difference in how mutexes behave and utilize the queue structures, so those are addressed after the queue itself is explored in more detail.

Creating and deleting queues dynamically allocates and frees the memory required for the queue. This includes the variables that track the queue state, such as number of entries currently in the queue and the tasks that are blocked on reading. The allocation provides space for the number of elements specified times the size of each element. Given the potential for heap fragmentation and long latency calls, it is highly recommended that queues, semaphores, and mutexes are statically configured on boot or during major state transitions. The queues themselves do not support dynamic re-sizing, making the accesses to them much more deterministic.

The limited size of the queue also creates some additional effects when it is accessed to send a message or give a semaphore. Since the number of entries in the queue is configured at creation, it is possible for a task attempting to send something on the queue to encounter a full queue. It should be immediately obvious to a programmer what happens when a task attempts to read from an empty queue—the task will go into the pending state for up to the duration specified during the call to the read function. The same approach is used for a send attempt on a full queue. If a timeout is specified on the send call, the sending task may be placed in the pending state for up to that duration. Binary semaphores do not include a delay value in the wrapped calls to the queue APIs, so any attempt to set an already set semaphore will result in a failure return code. This does ensure that

the give for the semaphore does not incur additional delays [FreeRTOS Queues].

When task is delayed on an access to a queue, the list of tasks waiting to either send or receive is included in a priority-sorted list. These queues and semaphores are expected to be used as synchronization or signaling mechanisms, and the highest-priority task should be activated to either send or receive when the queue is available. It is also possible for a task to “peek” at the contents of the queue without removing the item from the queue. This can be useful in some situations and is supported by the API provided by the queue implementation.

The base queue implementation is extended to provide a mutex or recursive mutex capability. Similar to the counting semaphore, the mutex uses the queue structures to provide the base capability. The mutex adds another dimension not present in the normal queue or semaphore interactions. Where the queues and semaphores are sorted by the priority of the pending tasks, the mutex implementations allow for priority inheritance of the highest pending task to the current holder [FreeRTOS Mutexes].

Recursive mutexes add a simple extension to the normal mutexes where the same task can hold the mutex multiple times. The mutex is not released until the last lock is released by that task. This is an especially useful feature in mutexes where the calling task may invoke multiple functions on the same data set. After each operation completes, the one instance of the lock can be released until all activities have finished and the final unlock occurs. These mutexes are created and treated slightly differently, so it is important to use the GiveRecursive or TakeRecursive variations of the APIs when accessing recursive mutexes.

Beyond the communication and protection mechanisms, FreeRTOS provides two mechanisms for time-based activities. These are the delay API function calls and the software timers. Both of these mechanisms have similar resolutions but different trade-offs.

The TaskDelay function relies on the scheduler to create the timing on the delay. When a task calls the delay API, it is added to a list task with a number of ticks to count down before expiring. This is a static time that relies on only the scheduler, and the task is released directly in the schedule tick when the requested duration expires.

The software timers are handled by a separate task. The timer function calls send messages to the task on its command queue. The software timer task is released by a hardware timer that is scheduled based on the delay times that other tasks have requested. This flexibility allows times to be reset or changed. This mechanism also allows for auto-reloading of the timers. These features make the software timers especially well-suited for periodic actions or events that may have their execution time varied by outside signals. Since the timer expiration event handler is called in the context of the timer task, it is important to limit the operations in that callback. Long callbacks may compromise the timing of other software timers [FreeRTOS Timers].

FreeRTOS is primarily designed for embedded systems with limited resources. The OS allows the developer to configure the stack sizes of tasks and can dramatically limit the overall memory footprint. This is a big advantage in most cases as the overall device cost is tied to the resources—including memory—used by the platform. It does open additional opportunities for issues to arise, so FreeRTOS also provides mechanisms to help address those problems.

The first potential issue that could happen when stack sizes are left to the defaults or improperly set is a stack overflow. This results in the stack spilling over into other memory that is likely used for other purposes. The issue's cause can be difficult to debug since the effect of this contention may vary with time or loading or may appear in an unrelated area different than the source of the problem. The authors recommend enabling the memory stack check protections to combat this. It does have implications for stack size and some accesses but is a proactive approach that prevents these issues before they happen.

Another memory problem that may arise stems from the dynamic allocations from the heap using malloc. There are a number of ways that dynamic allocations could fail either from heap fragmentation or from lack of available resources. It is especially important for portable code where the memory sizes could change between platforms to include some protection. Within FreeRTOS there is an option for a global hook invoked when a malloc attempt fails. This is much more reliable than expecting all of the code allocating dynamic memory to properly check the return values and instigate a local error handling mechanism. Even with the protection, it is not recommended to use dynamic memory for any operation other than start-up or state transitions. The calls are still variable, and the potential for fragmentation could cause the application to fail.

The best protection available is the hardware-based memory management systems. They are not available on many platforms that support FreeRTOS as of version 8.2 since the targets used tend to be lower-cost and lower-complexity devices. Where it is available, the use of the restricted tasks and memory protections is highly recommended. The restricted tasks do require additional software engineering since they do not have a common memory space, but it results in good practice, better debug ability, and better security. To ensure that any effects of the protection do not cause deadline issues, it is also a good idea to start and develop with restricted tasks. Adding memory protection late in development creates a collection of issues, and the last ones to be tracked are the ones affecting real-time reliability and behavior.

Since many embedded systems are also sensitive to power consumption, FreeRTOS provides a way for the developer to enable a low power state when the system is idle. This tool is known as the Idle task hook and can change the behavior of the system. In its normal mode, FreeRTOS includes a lowest-priority task that does no processing but checks for new task switching. This ensures the lowest possible response time when nothing is running but continues to use processor cycles and power even when there is no work to do [FreeRTOS Idle Power].

From that description, it sounds like enabling a low power mode during idle periods would be a highly attractive concept. In practice it adds a layer of complexity that must be understood, accounted for, and debugged using special test scenarios. It still may be highly advisable since the power consumed may be an important factor. It is especially tricky for real-time systems since the low power modes also include a wake-up delay for the system. When an event occurs in a low power mode, the processor typically has to re-enable some of the hardware functionality. This results in some amount of delay that needs to be included in any timeline that may be subject to a low power state. This requires additional analysis and testing in the low power mode to ensure the system is still able to meet all of its deadlines.

Exercises

1. Create a user-defined interrupt handler for the timer ISR and a task for processing. The timer should be scheduled on a regular basis, and the interrupt handler should signal the processing task. To ensure that the

timer is being triggered with the correct periodicity, pass the interrupt timing to the processing task.

2. Create a pair of tasks that signal each other. The first task performs some computation, signals the other task, and waits for a signal from that task. The second task repeats the same pattern so that they alternate. Each task should complete a defined amount of work, such as computing a specified number of Fibonacci values. Profile each task so that one task is executing for 10 ms and the other for 40 ms.
3. Modify the timer ISR to signal two tasks with different frequencies: one task every 30 ms and the other every 80 ms. Use your processing load from #2 to run 10 ms of processing on the 30-ms task and 40 ms of processing on the 80-ms task.

Chapter References

- [DE1-SoC FreeRTOS] http://www.freertos.org/RTOS_Altera_SoC_ARM_Cortex-A9.html
- [FreeRTOS Idle Power] <http://www.freertos.org/low-power-tickless-rtos.html>
- [FreeRTOS Mutexes] <http://www.freertos.org/Real-time-embedded-RTOS-mutexes.html>
- [FreeRTOS Organization] <http://www.freertos.org/a00017.html>
- [FreeRTOS Ports] <http://www.freertos.org/a00090.html>
- [FreeRTOS Queues] <http://www.freertos.org/Embedded-RTOS-Queues.html>
- [FreeRTOS Tasks] <http://www.freertos.org/taskandcr.html>
- [FreeRTOS Timers] <http://www.freertos.org/RTOS-software-timer.html>

INTEGRATING EMBEDDED LINUX INTO REAL-TIME SYSTEMS

In this chapter

- Introduction
- Embedding Mainline Linux: Interactive and Best-Effort
- Linux as a Non-Real-Time Management and User Interface Layer
- Methods to Patch and Improve Linux for Predictable Response
- Linux for Soft Real-Time Systems
- Tools for Linux Soft Real-Time Systems

“Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT.”

—Linus Torvalds as documented by yquotes.com

11.1 Introduction

The Real-Time Linux initiative was kicked off in 1999 at the first RT Linux workshop and is now supported through OASDL (Open Source Automation Development Lab). As indicated by Linus Torvalds and the Linux Foundation, the use of Linux for real-time systems does not fit the original vision for Linux, but with care, Linux can be configured to provide predictable response in embedded systems. The question of whether it makes

sense to adapt Linux for mission-critical hard real-time systems remains open. So, RT Linux is most often used for soft real-time for predictable response and in enterprise solutions where the Linux kernel most often acts as an interface to lower-level mission-critical firmware and hardware services. The challenge for RT Linux to gain broader use in real-time systems is to gain acceptance and provide a competitive solution to the RTOS, but at the same time, preserve all of the features of Linux that make it a great alternative to the RTOS. Linux can be configured for a wide range of systems, from high-performance computing, to interactive desktop or laptop systems, to datacenter servers. The goal for an RTOS, cyclic executive, or any embedded mission-critical system has always been to keep the design and configuration simple, to simplify verification, and to minimize the resources required in terms of space, mass, and power. Many of the early features of the Linux kernel simply made it unacceptable for real-time systems (e.g., the early BKL [Big Kernel Lock], making the kernel non-preemptible), but many of these limiting features have also now been removed or improved [BKL]. In this chapter, we review best-effort, soft, and hard real-time requirements, how to adapt Linux for more predictable response, and how to integrate Linux into a range of hard and soft real-time architectures as a component. Further, it should be noted that embedding Linux is not synonymous with real-time Linux—not all real-time systems are embedded and not all embedded systems require real-time response.

11.2 Embedding Mainline Linux: Interactive and Best-Effort

Back in 1994 when one of the authors first thought about embedding Linux for a NASA Space Shuttle demonstration project, this concept was not so straightforward. The requirements called for a range of services, from best-effort to soft real-time, in one embedded system to demonstrate instrument operations and automated planning and re-planning of observations from the payload bay of the Space Shuttle. The RTEMS RTOS was ultimately selected to run on a Motorola 68K processor board based on its small memory footprint and clear real-time features, but we did run Linux on the ground at Goddard Space Flight Center (the payload operations ground entry point for telemetry and commanding) and a mixture of Sun (Oracle) Solaris workstations in our POCC (Payload Operations Control Center) at the University of Colorado Boulder [Shepperd98]. RTEMS was an excellent selection as was Linux as an alternative to another Solaris

Unix workstation; both saved cost on the project compared to proprietary operating systems and speeded up the project by providing non-proprietary full-source options compared to VxWorks and Solaris Unix [RTEMS]. The overall DATA (Distributed Automation Technology Advancement) system with embedded segment on the Hitchhiker class payload (running RTEMS), the ground entry point (running Linux), and the POCC running Solaris is shown in Figure 11.1. The system was flown on STS-85 in the summer of 1997 and was one of the first university-operated Shuttle payloads (Hitchhiker class) to be fully operated remotely from a campus outside of a NASA facility. At that time, this author had the thought, “Wouldn’t it be great if we could just run Linux on the embedded segment, the near real-time ground entry point and POCC systems, and on the engineering analysis workstations?” Today, almost 20 years later, this is much more feasible, given improvements made to the Linux kernel and the advancement in microprocessor capability with embedded SoCs.

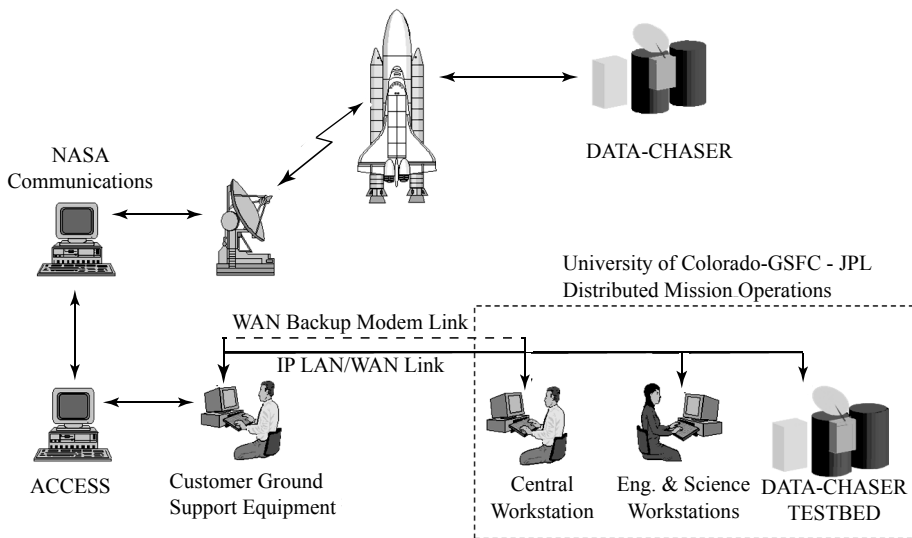


FIGURE 11.1 The University of Colorado Distributed Automation Technology Advancement Mission Operations System

Many systems are in fact not HRT (Hard Real-Time), which by definition means that failure to meet a deadline results in loss of life or property, but are rather soft real-time with requirements for near real-time interaction with users and the environment in which they operate. The DATA-CHASER (Colorado Hitchhiker and Student Experiment of solar Radiation)

system was just such a system. Students interacted with the payload through a NASA communication link for telemetry and commanding of three recycle instruments that had flown on previous missions borrowed from Colorado's LASP (Laboratory for Atmospheric and Space Physics). The fundamental engineering goal of the mission was to demonstrate a shared-control architecture where advanced planning, re-planning, and rule-based automation worked cooperatively with ground operators. This type of system needs to be interactive and meet soft real-time requirements where latency is on average within acceptable bounds.

Today, building the embedded segment with an embedded Linux system as well as ground segment Linux systems would be far simpler based on excellent and growing support for Linux in commercial embedded systems. Fueled by commercial applications like Android tablets and mobile smart phones, today one can simply purchase a development kit that comes preloaded with embedded Linux. For example, as shown in Figure 11.2, Texas Instruments ships an ARM-based SoC called OMAP (Open Multimedia Applications Platform) that runs a Debian/Ubuntu Linux distribution or Angstrom Linux (a more minimal distribution of Linux maintained by Texas Instruments). These types of systems can be powered from off-the-shelf lithium polymer batteries for many hours (up to 12 in the authors' experience with 5,000 mAh or more) in embedded applications.



FIGURE 11.2 TI-OMAP Beagle xM Running Debian/Ubuntu or Angstrom Linux

The available SoCs that boot embedded Linux (typically either Debian/Ubuntu Linux or Yocto Linux) are growing rapidly and now include many multi-core SoCs with vector coprocessors or options to interface custom

hardware for IO using FPGAs. The NVIDIA Jetson is a quad-core system with a 192-core GPU, as shown in Figure 11.3, that comes pre-loaded with Debian/Ubuntu Linux and OpenCV (Open Computer Vision), which is readily used in instrumentation and machine vision applications—for example, on UAVs (Unmanned Aerial Vehicles) or robotics applications where MV/CV (Machine Vision/Computer Vision) is useful for the application.

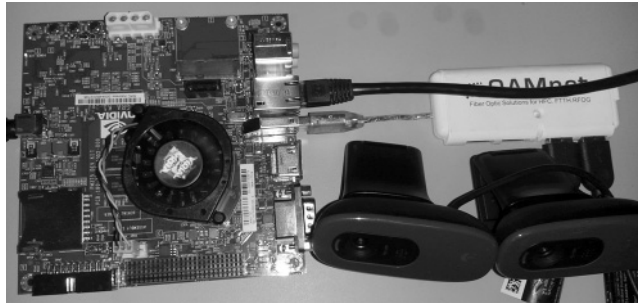


FIGURE 11.3 NVIDIA Jetson ARM SoC Ubuntu Linux Platform

The authors have been experimenting with both the NVIDIA Jetson and a semi-custom-built FPGA interface board for MV/CV using the Altera DE0 (Development and Education—Model 0) FPGA board, as shown in Figure 11.4. The point of this work is to compare the many-core vector coprocessing provided by a GPU for MV/CV to purpose-built FPGA state machines for image transformation in terms of power efficiency and processing capability. Either way, the availability of Linux on embedded systems is revolutionizing interactive, soft real-time, and best-effort embedded systems. Whether it will also overtake the RTOS for mission-critical systems remains to be seen. The authors caution that mission-critical HRT systems are best constructed using proven, verifiable RMA methods and AMP RTOS. This may never change since the market-driven aspects of embedding Linux (interactive Android systems, game consoles, set-top boxes, smart televisions) do not exist for HRT mission-critical systems. As such, it is not so likely that the Linux Foundation or Linux developers in general will be highly motivated to make changes to Linux to support HRT.

One option for system designers who want to use embedded Linux but have a mixture of HRT, SRT, and best-effort requirements for services is to use a coprocessor for the HRT services that either runs an AMP RTOS or a coprocessor that provides hardware-based processing with an FPGA or purpose-built ASIC. In general, the AMP RTOS solution hosts the HRT

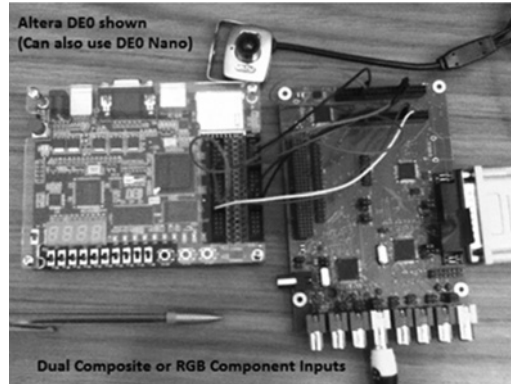


FIGURE 11.4 FPGA Coprocessor Board for Use with TI-OMAP Linux Systems for MV/CV

services with the SRT and best-effort services coordinated on a separate node through message-passing interfaces where Linux runs concurrently with the AMP RTOS.

Many more options for turn-key embedded Linux exist, such as the Wandboard shown in Figure 11.5, and the list continues to grow. Today the fun of Linux bring-up, development of the LSP (Linux Support Package), which is equivalent in function to an RTOS BSP (Board Support Package), is almost no longer an issue for the systems and applications developer who is happy using one of the many off-the-shelf embedded solutions.

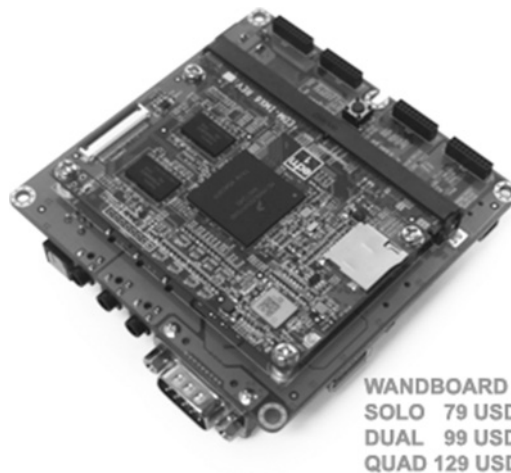


FIGURE 11.5 Freescale Wandboard (Photo: <http://www.wandboard.org/>)

However, as the Linux Foundation has noted, the ability to tailor Linux configurations, the Linux kernel, and the LSP is still critical for many embedded products and projects. So, the Linux Foundation has supported the Yocto project, which provides an approach to create simpler custom Linux configurations and kernel builds to assist the integration of Linux with new hardware and for more efficient smaller-size kernel images. Figure 11.6 shows the Intel Galileo Board, which can run either a Yocto-built Linux kernel or a more turn-key Debian/Ubuntu Linux kernel.

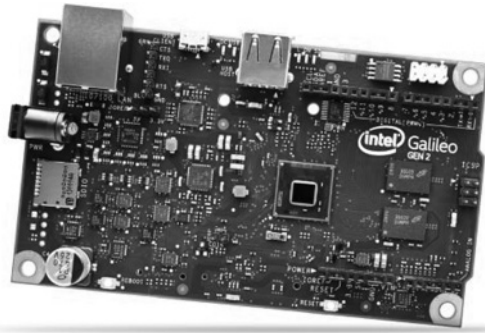


FIGURE 11.6 Intel Atom Galileo Board Running Ubuntu Linux

(Photo: <http://www.intel.com/content/www/us/en/do-it-yourself/galileo-maker-quark-board.html>)

Many of the Yocto-supported systems are designed for scaling of embedded systems and include advanced coprocessing solutions with FPGAs and PCI Express (Peripheral Component Interconnect), such as the DE2i board that boots a reference Yocto Linux image out of the box. Yocto is well supported by the Linux Foundation [Yocto]. One of the author's has used all of the example off-the-shelf boards with Linux as described in this chapter for teaching and research. These systems likely make good evaluation and reference design starting points for practitioners developing new Internet of Things appliances and devices, as well as practicing engineers working on Android mobile systems, game consoles, set-top boxes, and other commercial and consumer devices. The Intel DE2i shown in Figure 11.7 is the largest, most scalable and configurable embedded Linux system this author has worked with. Along with smaller-scale TI-OMAP, NVIDIA Jetson, the DE2i provides an excellent embedded Linux platform for research, teaching, and evaluation for applications that require significant digital media or image processing capability for SRT and interactive best-effort applications.

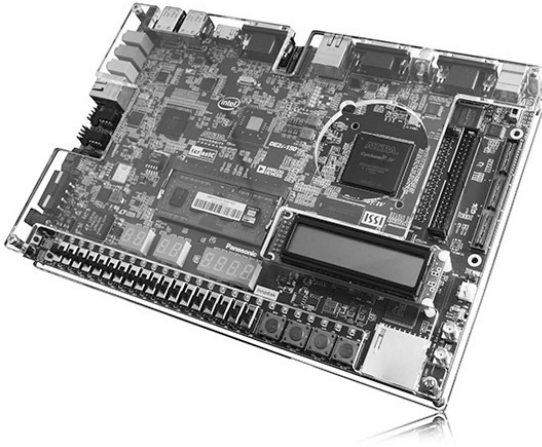


FIGURE 11.7 Intel Quad-Core Atom with Altera Cyclone IV FPGA Running Yocto Linux (Photo: <http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/de2i-150-fpga-dev-kit.pdf>)

11.3 Linux as a Non-Real-Time Management and User Interface Layer

Many industries and systems vendors provide support for HRT and SRT with interactive interfaces for users provided by Linux or Windows by segmenting the HRT/SRT services and supporting them in an embedded cyclic executive or RTOS subsystems or FPGA/ASIC hardware solution on-chip, with an interface to a best-effort operating system to provide interfaces for users. The idea is that the user interface is primarily for configuration and control of the HRT processing done by the CE, RTOS, or ASIC subsystem. This is a well-proven approach. For example, National Instruments often uses this approach for laboratory automation and data acquisition solutions so that use of its solutions provides well-known and friendly interfaces for scientific and engineering users, but at the same time provides predictable or deterministic responses for mission-critical digital control and data acquisition processes.

At the time of publication, the DE1 SoC, which incorporates a Cyclone-V SoC with FGPA fabric and multi-core ARM processors, has been made available through Altera's University Program. Altera provides a FreeRTOS port, as described on its support page [Altera RTOS].

Many embedded Linux systems run a Debian distribution kernel and often provide a full Ubuntu distribution as well, such that the embedded system includes native build tools and a graphical user interface just like Ubuntu installed on a Virtual Box VM.



FIGURE 11.8 Altera DE1-SoC FreeRTOS or Embedded Linux System
(Photo: <https://www.altera.com/support/training/university/de1-soc.html>)

The DE1 SoC and SoCs like it with hard processor cores integrated with an FPGA fabric provide an excellent option for running Linux best-effort or for soft real-time predictable-response applications with all hard real-time services implemented as FPGA state machines, providing deterministic response with coprocessing. This is an excellent alternative to an RTOS AMP approach for systems that must mix hard real-time services with the convenience of a general-purpose operating system, such as Linux.

11.4 Methods to Patch and Improve Linux for Predictable Response

The primary issue with trying to use Linux in HRT systems is the approach used for kernel data and preemption. In this section, we evaluate a number of the factors that make the default configuration of Linux problematic for deterministic scheduling and how the developers are attempting to address them. These changes are available from a variety of sources, and while they do not completely provide guaranteed scheduling performance, they greatly improve the situation.

Any operating system must face a number of challenges around protecting kernel and scheduler data that may be accessed by both threads and interrupts. This may come from different threads in the kernel performing operations, or it may be through API calls by user threads. Similarly, the interrupts may be signaling some activity to wake up a thread or a timer that releases a thread that was previously sleeping.

When Linux solely supported single-core processors, protecting these resources was relatively straightforward—use calls to disable interrupts. This ensures that the kernel data that may be updated when a task unblocks another task is not also accessed by an API that is called when an interrupt triggers a similar event. Disabling interrupts also prevents the scheduler from running and potentially replacing the kernel thread that was accessing the data structures.

As Linux began to support multiple processor cores, this situation became more complex. The initial approach was to include a single shared mutex that protects all the kernel data. This mechanism, called the Big Kernel Lock (BKL), was introduced shortly after the support for multiple processors. This allowed a couple of advantages, most importantly protecting kernel resources from simultaneous accesses on different processors. It also strongly mimics the original behavior of locking interrupts, making it very easy to overlay on the existing code.

There are a number of inherent difficulties with this type of approach. Using a large granularity (like “any kernel structures”) leads to inefficient protection. The use of the BKL presented other issues. While it did allow recursive locks to simplify nested calls, it could not be called within interrupt context. It is also possible to sleep a thread while holding the BKL, and it is automatically released when the thread sleeps. This makes coding especially tricky since there is the potential to release the lock and compromise the benefits of the kernel lock.

So what are the alternatives? Clearly, the Linux kernel benefits from more fine-grained locking compared to the earlier BKL. How else can Linux create a mechanism to lock data structures and protect access without also creating issues with interrupt context accesses? The answer for multiprocessor systems is an approach known as spin locks. The spin lock performs the same function as a binary semaphore used as a mutex, but it does it without sleeping the thread that attempts to take the lock. This idea is a very different approach from the BKL. It allows much finer-grain locking since spin locks allow each data structure or hardware resource to be protected individually. By preventing sleeping when the lock is currently held, it can even be used in interrupt context, creating a complete solution to the problem of protecting kernel resources.

So how does it work? To take the spin lock the processor continually polls on a lock, using an atomic check-and-set operation to attempt to

capture the lock. If the lock is captured, the code continues processing and completes whatever actions are needed in the critical section. Once the critical section exits, the code releases the spin lock and continues normal thread operation. This all works since in a multiprocessor system, there is another hardware context that is running the thread that currently holds the spin lock. This allows the critical section that currently holds the lock to complete and prevents the spinning from continuing indefinitely.

This implementation brings several limitations with it, creating constraints on the code that is using spin locks and changing the overall behavior of the scheduler. The first important restriction is that code that is currently holding a spin lock must not sleep. Similar to not sleeping when interrupts are disabled, it is very important that code that holds a spin lock completes the critical section as quickly as possible. Any thread or interrupt that is attempting to access the spin lock will tie up hardware resources and limit the number of cores available for other activities. Another obvious limitation is that spin locks cannot be used recursively. Once the lock is held, all future attempts to claim it will fail, even from the same thread.

The other effects are more subtle. While an effective mechanism for protecting data and ensuring that even interrupts have coherent access, the timing of this synchronization is nearly impossible to predict. Multiple kernel threads and interrupts all interacting to hold data structures with open-ended durations can prevent the scheduler and other key threads from running. Each thread that accesses a spin lock is—by necessity—un-preemptible within those critical sections. Making this problem more extreme, many architectures will prevent additional interrupts from firing while one is currently active. The consequence is that the spinning done in interrupt contexts further increases the variability of interrupt timing. With the scheduler tick driven off a timer interrupt, this directly translated into addition variability in the task scheduling.

The approach taken by the PREEMPT_RT patch looks to address these issues. The primary approach is to convert the spin locks into semaphores that allow the blocked process to sleep so that any kernel process that is holding a spin lock can sleep and allow the scheduler to run. This also requires that interrupts be run in high-priority threads instead of directly in interrupt contexts. With the interrupts in a process context, they can sleep as well while still preserving the protection of data between interrupts and kernel threads.

The only remaining sections that cannot be preempted are the areas that deal directly with scheduler resources. These sections get a new designation function call, and the number of these new non-preemptible sections is kept to a minimum. The more flexible approach reduces the complexity since there are fewer areas that require great care to not sleep the thread.

This does come with an impact from increased overhead. If a section holding a spin lock is brief—as it is expected to be—the window of time that the thread is sleeping is small. Especially when the overhead cost of saving and restoring contexts is large, this can slow down the response in many cases. The benefit is that the worst-case time is more tightly bounded, providing much more reliable scheduling.

To help mitigate some of the additional overhead, there is a version of the spin locks called “sleeping spin locks”—these poll briefly and then sleep if the lock is still not available.

For real-time services, an RTOS normally provides a binary semaphore, a mutual exclusion semaphore (MUTEX), and counting semaphores for producer and consumer threads. As has been well documented and discussed by the Linux community, a fair scheduling system, such as the default in Linux, does not suffer from priority inversion. This was hotly debated and discussed when Linux was first used in soft real-time system solutions. The reason a fair scheduler like Completely Fair Scheduler (CFS) in Linux eliminates priority inversion is because it is in fact fair. In the unbounded inversion scenario, the middle-priority thread (or task) must be able to interfere with the low-priority indefinitely as is the case with priority-preemptive run-to-completion schedulers in an RTOS. In a fair scheduler, the low-priority thread will continue to get some CPU time and will therefore not block the high-priority task waiting on the MUTEX indefinitely. However, Linux does provide the FIFO scheduling class with real-time priorities, which, unlike the default, is not fair. So, for FIFO threads in Linux, unbounded priority inversion can still be an issue. For this scenario when FIFO threads are used, the Linux kernel can be patched with the MUTEX / FUTEX patch for inversion-safe semaphores for use with priority-preemptive run-to-completion threads used with the FIFO scheduling option (compared to CFS time-sliced threads or round-robin).

11.5 Linux for Soft Real-Time Systems

Soft Real-Time Linux requires methods of analysis and tools to ensure that response times for SRT services are within acceptable bounds on average. This is true for a range of SRT applications run on Linux, including game consoles, set-top boxes, smart televisions, scientific laboratory data acquisition systems, and many other applications where predictable response is needed, but occasional variation in the latency and response is acceptable. Generally speaking, not only should the Linux kernel be configured and patched as described in the previous section, but also the service applications should be written with care using the POSIX real-time extensions presented in Chapter 9. The POSIX real-time extensions include many operating system features to improve predictable response for real-time services and applications. One of the best starting points to learn these extensions is still Bill Gallmeister's summary of the standards [Gallmeister95], but the reader should also carefully review the extensions to POSIX [POSIX1003.1] themselves, with emphasis on the real-time index [POSIX RT], and read the corresponding manual pages for Linux (or RTOS) being used to make sure that the standards are, in fact, supported. As of the time of the publication, it is possible to download an entire POSIX 1003.1 2013 Edition from the Open Group as a set of HTML files that can be browsed. Once this is done, the T101 directory will contain a Realtime Index [POSIX RT].

11.6 Tools for Linux for Soft Real-Time Systems

Many tools are available for Linux to profile and trace systems, including Wireshark, Kernelshark, system logs, the GNU profiler, Intel's VTune, and simple instrumentation with time stamps in application code as provided by example in this text. Generally speaking, profiling is useful to determine where the majority of time is spent in an application or by an operating system kernel. Tracing, which is often more useful for soft real-time verification, analysis, and debug, provides instrumentation showing events related to code execution with time stamps. Both profiling and tracing tools are valuable for Linux soft real-time systems, but tracing is invaluable.

Summary

Not everyone agrees that RT Linux is a great idea; however, it is gaining support, and is certainly feasible and useful for systems and applications that mix soft real-time with best-effort services. For example, Linux is already used in digital cable and digital media set-top boxes where video is decoded and presented in real time, but with relatively soft deadlines and with significant hardware acceleration provided by decoder coprocessors. Very few applications are truly hard real-time, where missing an occasional deadline by milliseconds or tens of milliseconds will really cause loss of life or property. So, even if RT Linux only supports soft real-time better than it does today, this is still significant and will broaden the use of Linux and threaten the RTOS market share. The cyclic executive has the key advantage of a very small resource footprint in addition to determinism, so it's unlikely that RT Linux will compete with it, but RT Linux is likely to be a formidable challenge to the traditional RTOS for all applications except HRT. So, HRT applications such as commercial aircraft flight control, digital control in general, and process control are likely to continue to use the RTOS for some time. The hypervisor RTOS running with a Linux configuration on the same platform is noteworthy and could compete with real-time Unix (e.g., LynxOS and RedHawk Linux). Based on a history of patching Linux to support predictable-response applications and real-time versions of Unix, it is unlikely that a modified version of VxWorks, for example, will replace Linux or slow down the erosion of RTOS market share for soft real-time systems. The fact that Wind River supports its own Linux distribution is an interesting indication that perhaps it feels the same way.

Exercises

1. Install Oracle Virtual Box on a Windows or Macintosh OS-X personal computer, download and install the Ubuntu LTS (Long-Term Support) version of Linux, and install it on a VM (Virtual Machine). Configure and boot Linux on the VM, download Linux examples from the DVD, and run and test the “simplethread” code.
2. Download, build, and run the Linux threaded image processing example from the DVD, and analyze, describe, and characterize timing for the thread grid used for both best-effort SCHED_OTHER and priority-preemptive run-to-completion SCHED_FIFO.



3. Apply Linux RT patches to the Linux kernel installed on Virtual Box from #1 or on any embedded Linux system, and run jitter and latency build-up tests before and after patching, similar to the analysis found on the DVD.
4. On a Virtual Box Linux installation from #1 or any embedded Linux system, download the DVD `posix_mq.c` code, build it, and show that you can send messages between `pmq_send` and `pmq_receive`. Describe where the messages are buffered.
5. On a Virtual Box Linux installation from #1 or any embedded Linux system, download the “twoproc” code, build it, run it, and compare it to “simplethread”. Which approach provides the simplest method to make use of multi-core systems and provide concurrency? Describe two or more advantages and disadvantages of Linux processes compared to threads in general.
6. On a Virtual Box Linux installation from #1 or any embedded Linux system, download the “blocking-examples” code, build it, run it, and describe what happens in `deadlock.c` as well as `pthread3.c` and `pthread3ok.c`.



Chapter References

- [Altera RTOS] http://www.freertos.org/RTOS_Altera_SoC_ARM_Cortex-A9.html
- [BKL] <http://kernelnewbies.org/BigKernelLock>
- [eLinux] http://elinux.org/Main_Page
- [E Linux] <http://elinux.org>
- [Gallmeister95] Gallmeister, B. O., *POSIX.4 Programmer's Guide: Programming for the Real World*. O'Reilly Media, 1st ed., January 1995
- [Linux Found] <http://www.linuxfoundation.org>
- [Oracle VB] <https://www.virtualbox.org/>
- [POSIX1003.1] http://www.opengroup.org/austin/papers/posix_faq.html, <https://www2.opengroup.org/ogsys/catalog/t101>
- [POSIX RT] From downloaded T101.zip, select T101/susv4tc1/index.html

[RTEMS] <https://www.rtems.org/>

[RT Linux Wiki] <https://rt.wiki.kernel.org>

[Savador14] Salvador, O., and D. Angolini, *Embedded Linux Development with Yocto Project*. Packt, Birmingham UK, 2014.

[Shepperd98] Shepperd, R., J. Willis, E. Hansen, J. Faber, and S. Siewert, “DATA-CHASER: A Demonstration of Advanced Mission Operations Technologies,” IEEE Aerospace Conference, 1998.

[Ubuntu LTS] <http://www.ubuntu.com/>

[Yocto] <https://www.yoctoproject.org/>

DEBUGGING COMPONENTS

In this chapter

- Introduction
- Exceptions
- Assert
- Checking Return Codes
- Single-Step Debugging
- Kernel Scheduler Traces
- Test Access Ports
- Trace Ports
- Power-On Self-Test and Diagnostics
- External Test Equipment
- Application-Level Debugging

12.1 Introduction

Debug ideally includes hardware and software support in the form of monitoring and control so that errant conditions can be reproduced and analyzed and corrective action taken to eliminate bugs. Corrections can be made with software or hardware modification in order to eliminate the errant behavior detected by a debug monitor. A debug monitor is any code or hardware state machine or interface that allows the user to observe errant, unexpected deviations from designed behavior. To detect and define a bug,

the system designer must first have a clear concept of correct behavior from a design specification. In this chapter common debug software and hardware mechanisms, both built into the system and external, are reviewed. It would be impossible to include a comprehensive review in one chapter, so the goal of this chapter is to arm the reader with knowledge so that at the very least, the reader will know what questions to ask and how to further research debug methods.

Since the first edition of this book, both Linux and Wind River's Vx-Works tools have dramatically improved, and the FreeRTOS project also became available and an option in addition to RTEMs and other open source RTOS (Real-Time Operating System). Overall, single-step debug, profile, and trace tools used for software verification have advanced significantly in terms of ease of use and capability, but the fundamental uses and skills remain the same. The Wind River tools are integrated into the Workbench IDE (Integrated Development Environment) [WRS06], which serves the same function as Tornado, but provides tighter integration of edit, build, debug, and verification tools, even more so than Tornado did. The IDE has always been an advantage of the RTOS along with small-footprint configurable kernels, low-latency context switches, and interrupt handling, but today, Linux now has all the same tools available. The difference is that it's up to developers to create their own IDE, using a wide range of editors, build methods, debuggers, tracing tools, and profilers. To stay completely up to date, it is likely that this chapter would have to be rewritten every six months, so here, the focus is the methods and value rather than specifics of how to run the tools. For those new to Linux, the DVD provides a number of quick-start how-to guides that demonstrate current use of Linux development, debug, profile tools, and trace tools.



12.2 Exceptions

Code is often developed as a set of functions to be called by other code modules or applications in a larger system. When code is designed for use by others, it's important to program defensively so that a function does not assume that it will be passed only expected arguments. Likewise, for applications calling library functions that are perhaps linked in as object code and not verified at the source level, it's possible that this code might perform an illegal instruction, attempt to decode a bad address, overflow its stack, or any number of other errant behaviors. In general, mature code

bases will not suffer from such shortcomings, but during development, integration, and test, it may be useful to use exception handling and program asserts to isolate errant functions and code blocks.

One of the simplest and most common mistakes that can cause almost every microprocessor to generate an exception is to divide by zero. Division by zero causes an overflow and an undefined arithmetic result. In VxWorks, the kernel includes default exception handling, which suspends the task that caused the exception and prints out debug information to assist with locating the cause of the exception.

For example, the following code will cause the divide-by-zero exception on the VxWorks simulator and if compiled and run on Linux. Nobody would ever write code this obviously wrong; however, if the denominator is computed and data-driven, division by zero might not be so obvious. For the purpose of understanding how exceptions work, it's also one of the easiest to force on all processors.

```
#include "stdio.h"
```

```
int diverror(void)
{
    return 1/0;
}
```

Run on the VxWorks simulator, the output to the windshell indicates that the task just spawned caused an exception and the suspended task ID noted along with the program counter and the target status register.

```
-> sp diverror
task spawned: id = 10f0f60, name = slul
value = 17764192 = 0x10f0f60
->
Exception number 0: Task: 0x10f0f60 (slul)
```

```
Divide Error
```

```
Program Counter:          0x00f45368
Status Register:          0x00010246
```

```
408a0b  _vxTaskEntry  +47 : _diverror (0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0)
```


Probing a little deeper with the Tornado tools, dumping the task states in the windshell reveals that the created task from the `sp` command has been put into the suspended state by the VxWorks scheduler:

```
-> i
NAME          ENTRY          TID      PRI    STATUS      PC      SP
-----
tExcTask      _excTask      1108de0   0  PEND          408358  1108ce0
tLogTask      _logTask      11032b0   0  PEND          408358  11031b0
tWdbTask      _wdbTask      10fe668   3  READY         408358  10fe518
slu1          _diverror     10f0f60  100  SUSPEND       f45368  10f0ed8
value = 0 = 0x0
```

Dumping exception debug information to the windshell or console along with suspension of the offending task is the default handling of a microprocessor exception for VxWorks. This is often sufficient during development; however, an application might want to supply specific exception handling. The VxWorks kernel therefore includes a callback that can be registered with the kernel so that the application can provide custom handling or recovery from exceptions. The divide-by-zero code is now modified to include an exception hook:

```
#include "stdio.h"
#include "excLib.h"

void myExcHook(int taskID, int excVec, void *excStackFrame)
{
    logMsg("Exception Trap for task 0x%x, excVec=0x%x\n",
taskID, excVec);
}
int diverror(void)
{
    excHookAdd(myExcHook);
    return 1/0;
}
```

With the added application-specific handler the application-specific handler is called in addition to the default handling. The ***setout*** utility is used to ensure that the handler ***logMsg*** is output to the windshell.

```
-> setout
Original setup: sin=3, sout=3, serr=3
All being remapped to your virtual terminal...
```

```

You should see this message now!!!
0x10f92b8 (t1): You should also see this logMsg
value = 32 = 0x20 = ' ' = __major_os_version__ + 0x1c
-> testExc
choose and exception [b=bus error, d=divide by zero]:
Generating divide by zero exception

```

```
Exception number 0: Task: 0x10f92b8 (t2)
```

```
Divide Error
```

```
Program Counter:      0x00f4510e
Status Register:      0x00010206
```

```

408a0b  _vxTaskEntry    +47 : 423df8 (110e450, 0, 0, 0, 0, 0, 0,
                                0, 0, 0)
423e51  _wdbFuncCallLibInit+ad : _testExc (0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0)
f452eb  _testExc          +f3 : _diverror (0)
0x10f92b8 (t2): Trapped an Exception for task 0x10f92b8
value = 0 = 0x0

```

```
-> i
```

	NAME	ENTRY	TID	PRI	STATUS	PC	SP
tExcTask	_excTask		1108de0	0	PEND	408358	1108ce0
tLogTask	_logTask		11032b0	0	PEND	408358	11031b0
tWdbTask	_wdbTask		10fe668	3	READY	408358	10fe518
t2	0x423df8		10f92b8	4	SUSPEND	f4510e	10f91c0

```

value = 0 = 0x0

```

Similarly in Linux, if the same code with divide by zero is executed, the process is terminated and a core is dumped to the file system for debug. Before looking into Linux, first run the same example code and now generate a bus error. After this second run, the task listing using the “i” command shows that the task for the divide-by-zero run and the task for the bus error are now both suspended.

```

-> testExc
choose and exception [b=bus error, d=divide by zero]:
Generating bus error segfault exception

```

```
Exception number 0: Task: 0x10f0f60 (t3)
```

```
General Protection Fault
Program Counter:      0x00f4512d
Status Register:      0x00010206

408a0b  _vxTaskEntry   +47 : 423df8 (110d948, 0, 0, 0, 0, 0, 0,
                                0, 0, 0)
423e51  _wdbFuncCallLibInit+ad : _testExc (0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0)
f452cf  _testExc       +d7 : _buserror (ffffffff)
0x10f0f60 (t3): Trapped an Exception for task 0x10f0f60
value = 0 = 0x0
-> i

```

NAME	ENTRY	TID	PRI	STATUS	PC	SP
tExcTask	_excTask	1108de0	0	PEND	408358	1108ce0
tLogTask	_logTask	11032b0	0	PEND	408358	11031b0
tWdbTask	_wdbTask	10fe668	3	READY	408358	10fe518
t2	0x423df8	10f92b8	4	SUSPEND	f4510e	10f91c0
t3	0x423df8	10f0f60	4	SUSPEND	f4512d	10f0e88

```
value = 0 = 0x0
```

The default handling in Linux or VxWorks is essentially the same if an exception is raised by errant code executing in a task or Linux process context. The default exception handling is much more drastic for code executing in VxWorks kernel or ISR context. In VxWorks, the default handling reboots the target. Looking now at output on the simulator console, the default VxWorks exception handler also provides indication of the exception here.

VxWorks

Copyright 1984-2002 Wind River Systems, Inc.

```
CPU: VxSim for Windows
Runtime Name: VxWorks
Runtime Version: 5.5
BSP version: 1.2/1
Created: Jul 20 2002, 19:23:59
WDB Comm Type: WDB_COMM_PIPE
WDB: Ready.
```

```
Exception !
Vector 0 : Divide Error
    Program Counter : 0x00f4510e
    Status Register : 0x00010206
Exception !
Vector 13 : General Protection Fault
    Program Counter : 0x00f4512d
    Status Register : 0x00010206
```

On Linux, the same code causes a core dump when the shell is configured to allow this.

To ensure that core dumps are allowed, use the built-in shell command called “unlimit” after compiling the example exception-generating code provided on the DVD.



```
[siewerts@localhost ex]$ tcsh
[siewerts@localhost ~/ex]$ gcc -g gen_exception.c -o genexc
[siewerts@localhost ~/ex]$ unlimit
```

Now, run the genexc executable to generate a segmentation fault by de-referencing a bad pointer.

```
[siewerts@localhost ~/ex]$ ./genexc
choose and exception [b=bus error, d=divide by zero]:b
Generating bus error segfault exception
Segmentation fault (core dumped)
```

On a Linux system, the core file dumped can be loaded and debugged with gdb. This allows for examination of the stack trace and identifies the offending line of C code.

```
[siewerts@localhost ~/ex]$ gdb genexc core.13472
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
Core was generated by `./genexc'.
Program terminated with signal 11, Segmentation fault.
```

```

Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x0804837f in buserror (badPtr=0xffffffff) at
                                                    gen_exception.c:38
38          someData = *badPtr;
(gdb) bt
#0  0x0804837f in buserror (badPtr=0xffffffff) at
                                                    gen_exception.c:38
#1  0x080483d8 in main () at gen_exception.c:56
#2  0x420158d4 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)

```

Run the code again and generate the divide-by-zero exception. Now load the core file dumped for the divide-by-zero exception generator, and, once again, the stack trace and offending line of code can be examined with ***gdb***.

```

[siewerts@localhost ~/ex]$ ./genexc
choose and exception [b=bus error, d=divide by zero]:d
Generating divide by zero exception
Floating exception (core dumped)
[siewerts@localhost ~/ex]$

siewerts@localhost ~/ex$ gdb genexc core.13473
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
Core was generated by `./genexc'.
Program terminated with signal 8, Arithmetic exception.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x08048368 in diverror (arg=0) at gen_exception.c:31

```

```

31          someNum = 1/arg;
(gdb)

```

So, both Linux and VxWorks provide stack trace information and the location in code where the exception is raised. When an exception occurs, it's easiest to single-step debug the code to determine why the code is causing the exception. In the Tornado environment, this is most easily done using the Cross Wind debug tool, as shown in Figure 12.1. The “Single-Step Debugging” section of this chapter will cover exactly how to start and use the single-step debugger in Tornado (or Workbench for readers using the latest IDE from Wind River).

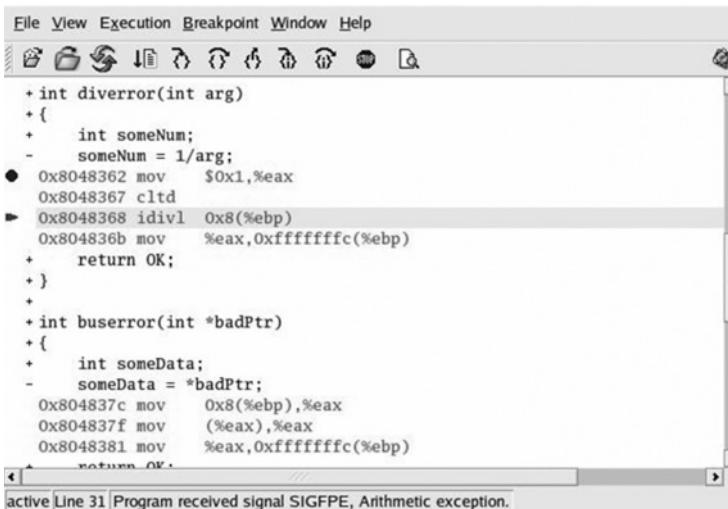


FIGURE 12.1 Using KDE Development Kdbg to Analyze and Exception

Likewise, in Linux, a graphical single-step debugger will make it easier to determine why code is raising an exception. Numerous graphical debuggers are available for Linux, and most run on top of the gdb command-line debugger. Figure 12.2 shows usage of the KDE environment Kdbg graphical debugger.

Again, the “Single-Step Debugging” section of this chapter covers exactly how to start and use the single-step debugger in the Linux KDE environment.

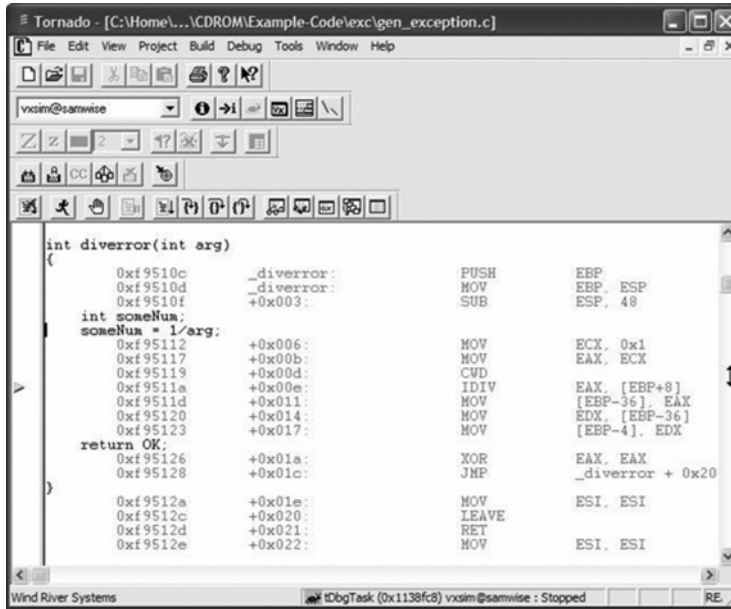


FIGURE 12.2 Using Tornado Cross Wind to Analyze Exception

12.3 Assert

Preventing exceptions rather than handling them is more proactive. Certainly code can check for conditions that would cause an exception and verify arguments to avoid the possibility of task suspension or target reboot. The standard method for this type of defensive programming is to include assert checks in code to isolate errant conditions for debug. In C code, pointers are often passed to functions for efficiency. A caller of a function might not pass a valid pointer, and this can cause an exception or errant behavior in the called function that can be difficult to trace back to the bad pointer. The following code demonstrates this with a very simple example that first passes printAddr a valid pointer and then passes a NULL pointer:

```

#include "stdio.h"
#include "stdlib.h"
#include "assert.h"

char validPtr[] = "some string";
char *invalidPtr = (char *)0;

```

```

void printAddr(void *ptr)
{
    /* will assert and exit program here if pointer is NULL */
    assert((int)ptr);

    printf("ptr = 0x%08x\n", ptr);
}

int main(void)
{
    printAddr(validPtr);
    printAddr(invalidPtr);

    return OK;
}

```

Running the preceding code on a VxWorks target or VxSim produces the following output:

```

ptr = 0x00f453ac
Assertion failed: (int)ptr, file C:/Home/Sam/Book/CDROM/Example-Code/assert.c, line 11

```

Use of `assert` checking in code makes the error in the calling arguments obvious and avoids confusion. Without the `assert` check on the pointer parameter, it might seem that there is an error in the function called when it finally attempts to de-reference or otherwise use the pointer, which would most likely cause an exception. The `assert` check is also supported in Linux.

12.4 Checking Return Codes

Any RTOS like VxWorks provides a significant API with mechanisms for task control, inter-task communication, synchronization, memory management, and device interfacing. Calls into the API can fail for numerous reasons, including the following:

- Failure to meet preconditions by the application (e.g., `semTake` when semaphore has not been created).
- Kernel resources have been exhausted.
- A bad pointer or argument is passed by the application.
- A timeout occurs on a blocking call.

For this reason, application code should always check return codes and handle failures with warnings logged or sent to console or assert. Not checking return codes often leads to difficult-to-figure-out failures beyond the initial obvious API call failure.

In VxWorks the task variable `errno` always indicates the last error encountered during an API call. A call to `perror()` in VxWorks will also print more useful debug information when an API call returns a bad code.

12.5 Single-Step Debugging

Single-step debugging is often the most powerful way to analyze and understand both software algorithmic errors, hardware/software interfaces errors, and sometimes even hardware design flaws. Single-step debugging can be done at three different levels in most embedded systems:

- Task- or process-level debugging
- System- or kernel-level debugging
- Processor-level debugging

Most application developers are accustomed to task- or process-level debugging. In this case, a process or task is started in VxWorks or Linux, and most often a break point is set for the entry point of the task or process. This method allows the user to debug only one thread of execution at a time. Often, this is sufficient control because either the application being debugged is signally threaded, or if multithreaded, then other threads in the overall application will most often block awaiting synchronizing events (e.g., a semaphore or message) before proceeding. Debugging asynchronous multithreaded applications is much more difficult and requires either system- or kernel-level debugging or processor-level using TAP (Test Access Port) hardware tools.

Task-level debugging in VxWorks is simple. Command-line debugging can be performed directly within the windshell. Graphical debugging can be performed using the Tornado tool known as Cross Wind, accessed and controlled through a source viewer that displays C code, assembly, or a mixed mode. For embedded systems, debugging is described as cross-debugging because the host system on which the debug interface runs does not have to be the same architecture as the target system being debugged.

On the other hand, it certainly can be the same, which might be the case for an embedded Linux system or an embedded VxWorks system running on Intel architecture. Furthermore, the embedded system may not have sufficient IO interfaces for effective debug. A cross-debugging system runs a debug agent on the embedded target, and debug source viewing and command and control are done on a host system. For Tornado, the host tool is Cross Wind and the debug agent is WDB (Wind Debug). The debug agent accepts commands and replies with current target state information when requested.

One of the most basic features of any debugger is the ability to set break points and to run or single-step between them. There are two ways that break points are most often implemented:

- Hardware break points
- Software break points

Hardware break points require that the processor include a break point address register, a comparator that determines whether the IP (Instruction Pointer) or PC (Program Counter) matches the requested break address, and a mechanism to raise a debug exception. The debug exception causes a halt in the normal thread of execution, and the debug agent installs a handler so that the user can examine the state of the target at the point of this exception. One important and notable characteristic of hardware break points is that the number is limited to the number of comparator registers provided by the specific processor architecture. Often the limit is only two or at most a half dozen, but it's definitely limited by hardware supporting resources. A significant advantage of hardware breakpoints is that they do not modify the code being debugged at all and are reliable even if memory is modified or errantly corrupted.

A processor reset or power cycle is most often the only way they can be cleared. Software break points are unlimited in number. They are implemented by inserting an instruction into the code segment of the thread of execution being debugged. They modify the code, but only by inserting a single instruction to raise a debug exception at each requested break point. After the debug exception is raised, the debug agent handling of the exception is identical. When the debug agent steps beyond the current break point, it must restore the original code it replaced with the debug exception instruction. Software break points have the disadvantage that they can and

mostly likely will be lost every time code is reloaded, if memory is errantly corrupted, and when a processor is reset.

Most software debug agents, such as WDB and the GNU debug agent, use software break points due to their flexibility and unlimited number. For task-level debug, the host Cross Wind debug tool requests the WDB debug agent to set a break point in a code segment for a specific task. This allows the debug agent to handle the debug exception, to compare the current task context to the task being debugged, and to suspend that task. Run the example code sequencer that creates two tasks that run and can be calibrated to run for 10 to 20 milliseconds on any target system, and the sequencer releases serviceF10 ever 20 milliseconds and serviceF20 every 50 milliseconds. This uses 90% of the processor cycles while it runs. After the two tasks are running, their state can be observed with the “i” command to dump all task control blocks.

```
-> i
```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO
tExcTask	_excTask	1158de0	0	PEND	408358	1158ce0	0
tLogTask	_logTask	11532b0	0	PEND	408358	11531b0	0
tWdbTask	_wdbTask	114e668	3	READY	408358	114e518	0
t1	0x423df8	11492b8	4	DELAY	408358	11491d8	0
serviceF10	_fib10	1140f60	21	PEND	408358	1140e74	0
serviceF20	_fib20	113bc08	22	PEND	408358	113bb1c	0

```
value = 0 = 0x0
```

The t1 task is the sequencer and most often will be observed in delay between releases of serviceF10 and serviceF20. The two services will most often be observed as pending while they wait for release. The **i** command is implemented by a request to the tWdbTask (the target agent), so it will be observed as ready (running in VxWorks) because it has to be running to make the observation. Now, start the Cross Wind debugger (usually by clicking on a bug icon), select Attach, and then select serviceF10 and use the **i** command to dump the TCBs (Task Control Blocks) again.

Figure 12.3 shows the Cross Wind debugger now attached to serviceF10.

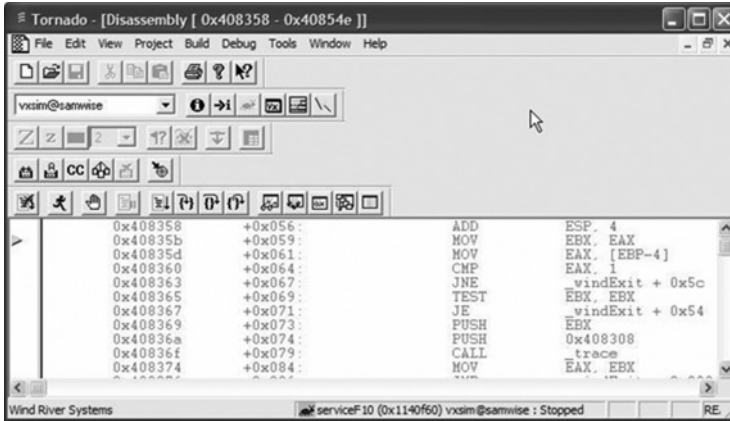


FIGURE 12.3 Cross Wind Debugger Attached Asynchronously to Running Task

```
-> i
```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO
tExcTask	_excTask	1158de0	0	PEND	408358	1158ce0	0
tLogTask	_logTask	11532b0	0	PEND	408358	11531b0	0
tWdbTask	_wdbTask	114e668	3	READY	408358	114e518	0
t1	0x423df8	11492b8	4	DELAY	408358	11491d8	0
serviceF10	_fib10	1140f60	21	SUSPEND	408358	1140e74	0
serviceF20	_fib20	113bc08	22	PEND	408358	113bb1c	0

```
value = 0 = 0x0
```

The serviceF10 task has been suspended by the WDB debug agent. Now the debug agent can be queried for any information on the state of this task and code executing in the context of this task that the user wants to see. It can be single-stepped from this point as well. When this is done, most often the running task is caught by the attach somewhere in kernel code, and if the user does not have full kernel source, a disassembly is displayed along with a single-step prompt.

Asynchronously attaching and single-stepping a running task are often not helpful to the user debugging the code being scheduled by the kernel and that calls into the kernel API. Instead of attaching to a running task, now start the same sequencer from the debugger (first stop the debugger and restart the target or simulator to shut down the running tasks). Now, restart the debugger after the target or simulator is running again, and from the Debug menu, select Run. Start Sequencer from the Run dialog box, and select Break at Entry. Note that arguments to the function entry point

can also be passed in from the debugger tool if needed. At the windshell, output will indicate that the task has been started and that it has hit a break point immediately on entry.

```
->
Break at 0x00f940e7: _Sequencer + 0x3          Task: 0x11492b8 (tDbgTask)

-> i

```

	NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO
tExcTask	_excTask	1158de0	0	PEND	408358	1158ce0	0	
tLogTask	_logTask	11532b0	0	PEND	408358	11531b0	0	
tWdbTask	_wdbTask	114e668	3	READY	408358	114e518	0	
tDbgTask	_Sequencer	11492b8	100	SUSPEND	f940e7	1149244	0	

```
value = 0 = 0x0
```

Note the debug agent wrapper task tDbgTask with the entry point started from the debugger is in the suspend state. At the same time, a viewer window with a source-level debug prompt appears in Tornado, as shown in Figure 12.4.

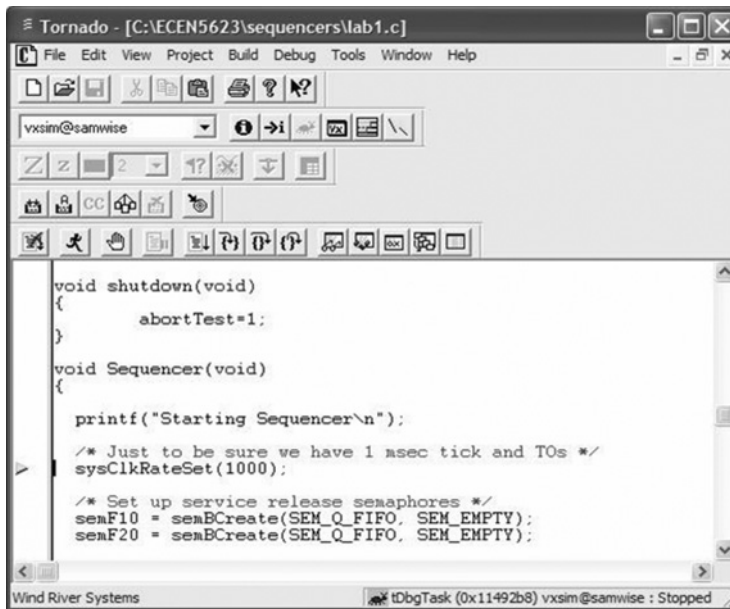


FIGURE 12.4 Cross Wind Source-Level Debugging

In general, a C compiler generates numerous machine code instructions for each line of C code, and the source-level debugger can view the current IP (Instruction Pointer) location in a mixed C source and disassembled view, as shown in Figure 12.5. This view can be very useful if the C compiler is generating bad code (not likely, but possible) or if it's generating inefficient code (more likely).

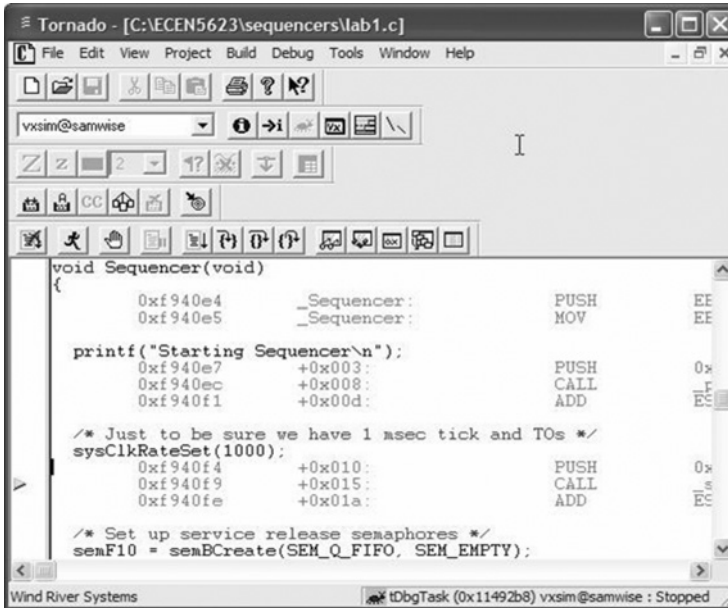


FIGURE 12.5 Cross Wind Mixed C and Assembly-Level Debugging

Process-level debug in Linux is often the equivalent of task-level debug in VxWorks. A Linux process can be multithreaded with POSIX threads, for example, in which case the pthread is the equivalent of the VxWorks task. One advantage of embedding Linux is that many of the application algorithms can be developed and debugged on the host Linux system without cross-debugging. Cross-debugging is required only when the bug is related to how the code specifically executes on the target system. Figure 12.6 shows the use of Kdbg to debug the multithreaded priority inversion demo code found on the DVD.



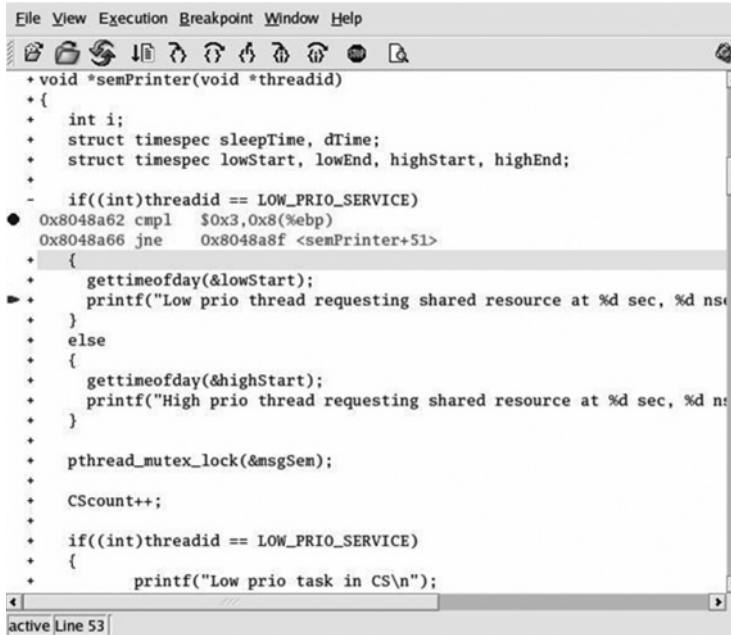


FIGURE 12.6 Kdbg Mixed C and Assembly Source-Level Debugging

The example pthread code can be built using gcc linking in the pthread library and using the -g option to include debug symbols.

```
[siewerts@localhost ~/ex]$ gcc -g pthread.c -lpthread -o pthread
```

The Kdbg application can load the pthread executable and will find the C source based upon the ELF (Executable and Linking Format) information, which includes all symbols as well as the source code path.

If an embedded Linux application needs to be debugged, it can be remotely debugged with gdb by connecting to gdb remotely over a serial or TCP connection over Ethernet. To use gdb remotely, you must link debugging stubs with your embedded application and then use gdb target remote /dev/ttyS0, for example, to connect over serial to the embedded application for debug.

System- or kernel-level debugging rather than attaching to a single task or process is tricky because both the Linux and the VxWorks systems rely upon basic services (or daemons in Linux) to maintain communication with the user. For example, in VxWorks, the WDB and shell tasks must run along with the net task to maintain communication between the Tornado host tools and the target. Similarly, Linux communicates through shells or a windowing

system. In VxWorks, system-level debug requires a special kernel build and the use of a polling driver interface for communication. Likewise, Linux requires use of kdb the kernel debugger. The advantage of system-level debug is that it allows for single-stepping of the entire kernel and all applications scheduled by it. System- or kernel-level debugging is most often used by Linux driver, kernel module, and kernel feature developers.

Instead of system-level debug, a hardware debug method can be used to single-step the entire processor rather than using the system-debug software method. This has an advantage in that no services are needed from the software because communication is provided by hardware. The JTAG (Joint Test Application Group) IEEE standard has evolved to include debug functionality through the TAP (Test Access Port). A *JTAG* is a device that interfaces to a host via parallel port, USB, or Ethernet to command and control the TAP hardware that allows for processor single-stepping, register loads and dumps, memory loads and dumps, and even download of programs for services such as flashing images into nonvolatile memory.

Figure 12.7 shows the Wind River VisionCLICK JTAG debugger running on a laptop connected to a PowerPC evaluation board with a parallel port host interface and a JTAG BDM (Background Debug Mode) connection to the PowerPC TAP.



FIGURE 12.7 Typical JTAG Debugger Setup

The JTAG standard includes external clocking of the device under test (most often the processor for debug), the capability to clock test data in

or out of the TAP, and reset control. The TAP provides basic decoding of address, data, and control so that any addressable memory within the processor can be read or written. A JTAG debugger can be used, including extended versions of gdb, so that code segments loaded via JTAG into memory of nonvolatile memory can be single-stepped. Although JTAG debugging requires additional hardware tools, it has powerful capability. For example, code can be boot-strapped onto a system that has no current boot code. Bringing up new hardware and downloading, testing, and flashing initial boot code that runs from a system reset vector is a firmware development task. By definition, firmware is software that runs out of nonvolatile memory, most often to boot a device and make it useable by higher-level software, often loaded by the firmware. The JTAG requires no target software support and provides an interface so that initial boot-strap code can be loaded, debugged, and eventually programmed into a nonvolatile boot memory device. After the platform has been boot-strapped with JTAG, a more traditional debug agent, such as WDB, can be used instead for system-level or task-level debug.

Since publication of the first edition of this book, many systems now have a USB-to-JTAG bridge chip (or module on an SoC) so that a simple USB cable from your laptop is all that you need for JTAG debugging (e.g., the DE1-SoC featured in Chapter 11 and with resources found on the DVD). The fundamentals of JTAG and TAP, however, have remained the same.



Ability to develop code on embedded systems with native development tools has also grown tremendously. For example, the Beagle boards and the Jetson featured as examples in this book come pre-loaded with boot images for Linux. The DE1-SoC (System on a Chip) with FPGA (Field Programmable Gate Array) for reconfigurable computing has readily found pre-built Linux images that can be written to a flash device (micro-SD card) using open source utilities to create an embedded Linux boot and root file system. So, the days of laboring over board bring-up are long gone; however, any work on custom hardware will still require the use of a JTAG for BSP (Board Support Package) or LSP (Linux Support Package) development. The BSP or LSP is the firmware that is board-specific and enables an RTOS or Linux to boot (start execution out of a power-on reset) and manage basic interfaces, like serial, USB, and Ethernet. In some ways these reference boot images and kernels are similar to the old PROM monitors (plug-in EEPROMs that had basic pre-built shells that ran over serial), but of course much more capable. Another reason that JTAG still has significant

value is that developers may want to build a simpler cyclic executive or their own custom operating environment. This is a fair amount of work, but for more purpose-built systems, this can lead to much less overhead and a more efficient set of real-time services and features. Finally, the JTAG is most often used for reconfigurable SoC systems, like the DE1-SoC, to download FPGA bit-streams, the state machine and combinational logic designs used to define coprocessors. So, JTAG will continue to be an invaluable development and debug tool; USB-to-JTAG bridges on-chip and at lower cost have made this critical firmware tool more accessible than ever.

12.6 Kernel Scheduler Traces

Kernel scheduler tracing is a critical debug method for real-time systems. Real-time services run either as ISRs or as priority-preemptive scheduled tasks. In VxWorks, this is the default scheduling mechanism for all tasks. In Linux, the POSIX thread FIFO scheduling policy must be used for system scope threads. Process scope POSIX threads are run only when the process that owns them is run. System scope POSIX threads, such as VxWorks tasks, are run according to preemptive priorities by the kernel. The kernel itself must also be preemptible for predictable-response real-time scheduling. The Linux kernel can be configured and patched to make it more preemptible, like the VxWorks Wind kernel. Given a preemptible priority-driven multi-service system as just described, RM theory and policy can be used to assign priorities to services and to determine whether the system is feasible. A system is feasible if none of the services will miss their required deadlines. Theory is an excellent starting point, but the theoretical feasibility of a system should be verified and any deadline misses observed must be debugged.

The Tornado/VxWorks development framework (now called Workbench) includes a tool called WindView (now called System Viewer) that provides an event trace of all tasks (services) and ISRs (services) in the system along with synchronous and asynchronous events. A semaphore give and take is shown as a synchronous event on the context trace along with asynchronous events, such as interrupts, exceptions, and timeouts. Figure 12.8 shows a WindView trace for an NTSC video capture driver, which transports frames to an external viewing application over TCP. System Viewer adds some new post-capture (after the trace is downloaded from a target) analysis features, but largely is the same otherwise. Often the simplest level of even collection showing task execution and context switches along with interrupts is in fact the most useful view.

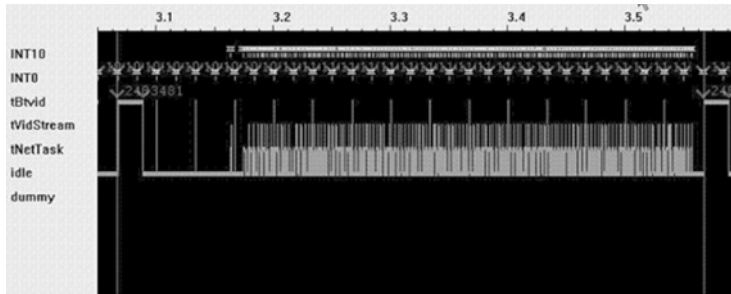


FIGURE 12.8 WindView Trace for Video Capture and Transport

In Figure 12.8, the 1-millisecond interval timer interrupt can be seen as INT0.

Exactly 33 of these occur between the two starts of the tBtvid task releases. The video encoder frame rate for this trace was 30 fps. The INT10 interrupt is associated with the tNetTask and is the Network Interface Card (NIC) DMA-completion interrupt. The bulk of the service activity between frame acquisitions is releases of the streaming task and the TCP/IP network task. Idle time indicates that the processor is not fully utilized, and in a WindView trace, this is time that the Wind kernel dispatcher was spinning and waiting for something to dispatch from the ready queue by priority. Figure 12.9 shows two synthetic load services (each service computes the Fibonacci sequence): S_1 runs for 10 milliseconds every 20 milliseconds, and S_2 runs for 20 milliseconds every 50 milliseconds.

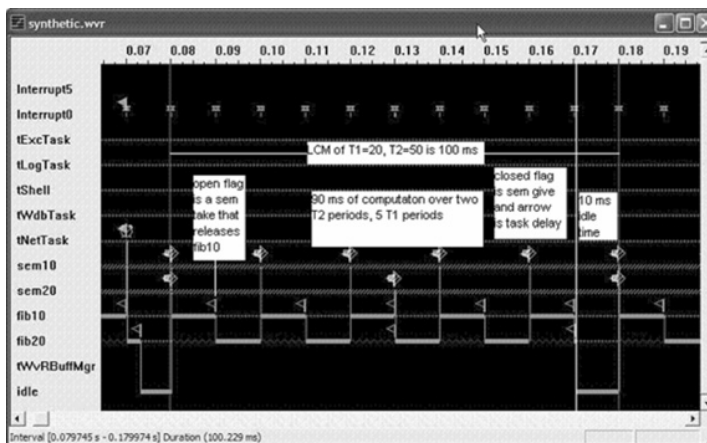


FIGURE 12.9 WindView Trace for Synthetic Service Loading

On this trace, the semaphores used to sequence and release the two services, fib10 and fib20, can be seen along with two sequencers, sem10 and sem20, which simply delay and give semaphores to release fib10 and fib20. The task delays are indicated by the right-pointing arrows and the hashed trace. The solid traces for fib10 and fib20 are processor execution time, and the dotted trace indicates time where some other task or interrupt is running, or idle time. This particular two-service scenario uses 90% of the available processor cycles over a 100-millisecond period. The 100-millisecond period is the LCM (Least Common Multiple) of the two release periods $T_1=20$ and $T_2=50$ milliseconds. The expected execution times are $C_1=10$ milliseconds and $C_2=20$ milliseconds for S_1 and S_2 . From this trace, it's clear that the releases of the services are well synchronized, and the execution times are accurate and as expected. Time-stamping on the trace is done using architecture-specific hardware timers, such as the interval timer, or on the Intel x86, the TSC (Time Stamp Counter), which is as accurate as the processor cycle rate. In general, time-stamping is accurate to a microsecond or less when supported by a hardware time stamp driver.

Scheduler traces of tasks and kernel events, such as WindView, are clearly very illuminating when debugging real-time systems. Situations like services overrunning deadlines are made obvious and allow the debugger to zero in on problematic sequences and perhaps better synchronize services. WindView can also be very helpful when services simply run too long and must be tuned to improve efficiency. The runtime of each release can be clearly observed. Using a function call, `wvEvent`, in application code, the trace can be further annotated with the chevron indicators and counts shown previously in Figure 12.8. These application event indicators can be used for events such as frame counts or anything that is useful for tracing application-level events. Although WindView is clearly useful, a frequent concern is how intrusive all this tracing is to the execution of application services.

WindView (System Viewer) is a tracing mechanism that involves software in-circuit rather than hardware in-circuit to collect the trace data. SWIC (SoftWare In-Circuit) methods are advantageous because they don't require electrical probing like logic analyzers (LAs) or built-in logic on processor chips, and can be added to systems in the field and even remotely accessed. One of the authors worked on a project where operators of a space telescope have the option of turning on WindView collection and dumping a trace back to Earth over the NASA Deep Space Network—hopefully this

will never be needed. To understand how intrusive SWIC methods such as WindView are, you need to understand the event instrumentation and the trace output. The least intrusive way to collect trace data is to buffer the trace records in memory (typically bit-encoded 32-bit words or cache-line-sized records). Most processors have posted write-buffer interfaces to memory that have significant throughput bandwidth. Looking more closely at the traces in Figures 12.8 and 12.9 earlier in the chapter, it's clear that events occur at the rate of milliseconds on the 200 MHz processor the traces were taken on. In general, because there are trace events every 1 million or more nanosecond scale processor cycles for event rates measured in milliseconds and the trace writes to memory take one write-back cycle, the trace output loading on the processor is very low. The instrumentation to detect events in the Wind kernel requires built-in logic in the kernel itself. The instrumentation requires logical tests. Overall, each event trace point might take 10 to 100 processor cycles every 1 million cycles—a very low additional loading to the processor for tracing. For low-load tracing, it's critical that the trace be buffered to memory. After collecting trace data, the full trace should be dumped only after all collection is done and during non-real-time operation or slowly during slack time. Dumping a trace buffer can be very intrusive, and significant care should be taken regarding how and when this is done. In all traces shown in this chapter, dumping was done after collection by request using the WindView host tools.

Building trace instrumentation into the Wind kernel is simple. The Workbench (or Tornado) configuration tool can be used to specify how the WindView instrumentation is built into the VxWorks kernel, where and how much trace data is buffered, and how time-stamping is done. Figure 12.10 shows usage of the Tornado (now Workbench) kernel configuration tool and specifically WindView instrumentation, download, and time-stamping options for the VxWorks build. The latest versions of Wind River Workbench with System Viewer work essentially the same way as Tornado and WindView, but of course the graphical user interface has an updated look and is perhaps more user-friendly.

Linux can be traced using a tool called the Linux Trace Toolkit, which is very similar to WindView. The operation is a bit different, but the end result is the same: a SWIC trace of kernel and system events, including application services. Like VxWorks, to use LTT, you must patch and build a custom Linux kernel with LTT instrumentation. The details of LTT usage are well covered by the maintainers of LTT and by *Linux Device Drivers*

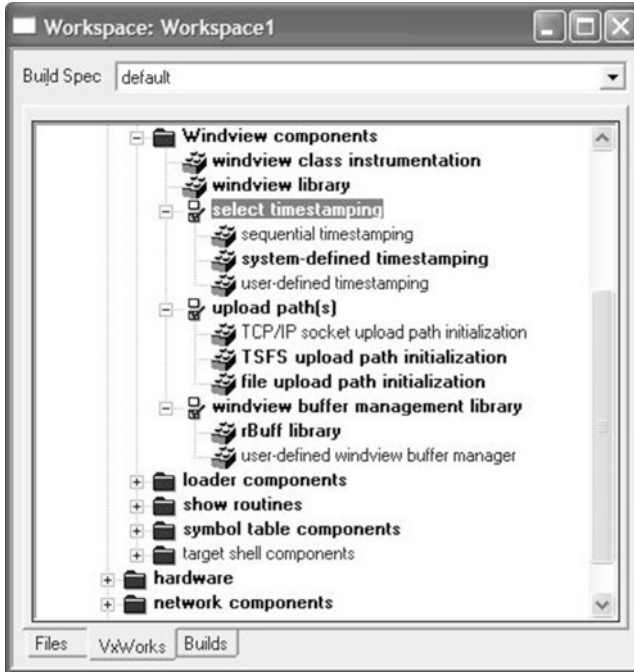


FIGURE 12.10 Adding WindView Kernel Instrumentation

[Corbet05] and *Building Embedded Linux Systems* [Yaghmour03]. Note that SWIC tracing can be easily built into most any system by placing memory writes with efficient encoded trace records into the kernel scheduler. Much of the work is related to dumping the trace buffer and post-processing it for analysis. Often placing the instrumentation in the kernel is very straightforward.

Since the first edition of this book was written, a number of additional system tracing tools have become available for Linux, including “Ftrace” [Ftrace], Kernelshark [Kernelshark], and Systemtap [Stap]. At the time of the first edition, LTT was really the only option for event trace verification and debug similar to the Wind River System Viewer, but today, there are numerous options for tracing Linux kernel events and function calls and viewing a graphical trace. The online documentation for all three tools is excellent. Likewise the installation of LTT next-generation [LTTng] is much simpler than the previous versions, which required kernel patches. With a few apt-get installations you can be running LTTng and looking at ***babel-trace*** output listings pretty quickly. Systemtap is also simple, but requires

writing scripts, although a large library of existing scripts exists that can be used as is or adapted. Finally ftrace and Kernelshark are perhaps most like WindView (System Viewer) for VxWorks. None of the Linux tracing tools seem as simple as System Viewer, but keep in mind that the Linux kernel is much more complex than the Wind kernel and has many more events.

12.7 Test Access Ports

The IEEE 1149.1 standard for JTAG was originally designed for boundary scan to ensure that devices on a digital electronic board assembly were all properly connected. Designed primarily for factory verification, the interface includes test data in and test data out signals so that bit sequences can be shifted into and out of a chain of devices under test. An expected test data output sequence can be verified for a test data input sequence. The interface also includes a clock input and reset. Eventually, microprocessor and firmware developers determined that JTAG could be extended with the TAP interface. The TAP allows a JTAG to send bit patterns through the scan chain and also allows the JTAG user to command a processor, single-step it, load registers and memory, download code, and dump registers and memory out so that commands and data can be sent to the device under test. A processor with a TAP can be fully clocked and controlled by a JTAG device, which allows the user to boot-strap it into operation right out of the hardware reset logic.

From the basic JTAG functionality, external control of microprocessors for firmware development has continued to evolve. Multiprocessor systems on a single chip can also be controlled by a single JTAG when they are placed on a common boundary scan chain. The TAP allows for selection of one of the processors with bypass logic in the chain so that processors not under JTAG control can forward commands and data. More on-chip features to support JTAG have evolved and are often referred to as on-chip debug, yet the basic command and control of these additional features are still through the basic JTAG signal interface. Before JTAG, firmware was developed for new hardware systems with a “burn and learn” approach, where EEPROM devices were externally programmed and socketed into the new system to provide nonvolatile boot code to execute from the processor reset vector. Typically, the EEPROM was programmed to first initialize a serial port to write out some confirming “hello world” data, blink an LED, and initialize basic components, such as memory. If the EEPROM

program didn't work, the firmware programmer would try again, repeating the process until progress was made.

After a PROM program was developed for a new processor board, most often the program was provided as a PROM monitor to make things easier for software developers. PROM monitors would provide basic code download features, memory dumps, disassembly of code segments, and diagnostic tests for hardware components on the system board. The PROM monitor was a common starting point for developing more sophisticated boot code and firmware to provide platform services for software.

Today, most systems are boot-strapped with JTAG rather than “burn and learn” or PROM monitors. The capability of JTAG and on-chip debug has progressed so much that, for example, the entire Tornado/VxWorks tool system (Workbench/VxWorks) can be run over JTAG, including WindView (System Viewer). This allows firmware and software developers to bring up new hardware systems rapidly, most often the same day that hardware arrives back from fabrication. The history of this evolution from burn and learn to PROM monitors to use of JTAG and more advanced on-chip debug is summarized well by Craig A. Haller of Macraigor Systems [Zen]:

First there was the “crash and burn” method of debugging. . . . After some time, the idea of a hardware single step was implemented. . . . At some point in history, someone (and after reading this I will get lots of email claiming credit) had the idea of a debugger monitor (aka ROM monitor). . . . Similar in concept to the ROM emulator, the next major breakthrough in debugging was the user friendly in-circuit emulator (ICE). . . . The latest addition to the debugger arsenal is on-chip debugging (OCD). . . . Some processors enhance their OCD with other resources truly creating complete on-chip debuggers. IBM's 4xx PowerPC family of embedded processors have a seven wire interface (“RISCTrace”) in addition to the OCD (“RISCWatch”) that allow for a complete trace of the processor's execution.

The OCD that Craig Haller describes is a combination of JTAG with a trace port, which approximates the capability of an ICE. The ICE traditionally included a chip bond-out interface so that the ICE could monitor all pin IO signals to the DUT (Device Under Test). This was more valuable when most processors interfaced to memory to fetch code and update data without cache, on-chip memory, on-chip buses, and in the case of SoC, even multiple processor cores on-chip. Monitoring external signals when

all of the action is on-chip does not make sense. So, the ICE has become less widely used and methods of OCD (On-Chip Debug) are expanding to include not only JTAG and trace but also on-chip ILAs (Internal Logic Analyzers) and signal probe multiplexors, such as the Xilinx Chip Scope monitor. Given the trends with larger and larger levels of integration on a single chip with SoC designs, OCD will continue to expand. Along with simplification of the cabling (typically only a USB cable needed today since the USB-to-JTAG bridge is often on-chip now too) the capabilities of OCD continue to improve. Likewise, many embedded systems today ship with pre-built Linux images or RTOS images (almost serving the same purpose as a PROM monitor, but with much more sophistication). However, JTAG and OCD will always have values since someone must always write bring-up firmware for new printed circuit boards and new SoCs.

12.8 Trace Ports

Simpler microcontrollers and older microprocessors often were integrated into a system using an ICE (In-Circuit Emulator). An ICE includes an interface that is interposed between a processor and the system board and can monitor and control all input and output signals to and from the device. The ICE can therefore track the state of the device under test as long as no internal memory devices other than registers are loaded and stored to an externally visible memory. When on-chip cache emerged as a processor feature, this made a true ICE difficult to implement.

Then the ICE had no idea what was being loaded from cache and written back to cache on-chip and could easily lose track of the internal state of the processor being emulated. The emulation provided not only all the features of JTAG TAP but also full-speed state tracing so that bugs that were observable only running at full speed, and not seen when single-stepping, could be understood. This type of hardware or software bug, often due to timing issues or race conditions in code that is not well synchronized, is known as a *Heisenbug*. The name indicates that, like the Heisenberg Uncertainty Principle, where a particle's position and momentum can't be simultaneously observed, a Heisenbug can't be single-stepped and the faulty logic observed simultaneously. This type of bug is observable only under full-speed conditions and is difficult to isolate and reproduce in a single-step debug mode. An ICE can be invaluable for understanding and correcting a Heisenbug.

Trace ports have emerged as a solution for the cost and complexity of ICE implementation and the ability to debug full-speed execution of software with no intrusion at the software level. WindView (System Viewer) is also an approach, but does involve intrusion at the software level. Trace ports use internal hardware monitors to output internal state information on a limited number of dedicated IO pins from a processor to an external trace acquisition device. For example, the IP can be output from the processor core onto 32 output pins every cycle so that the thread of execution can be traced while a system runs full-speed. The trace analysis is done off-line after the fact, but collection must occur at the same speed as the execution, and therefore external trace acquisition requires test equipment such as a logic analyzer or specialized digital signal capture device. For a full picture of what code is doing in a full-speed trace, the IP must be captured along with data address and data value vectors on the interface unit between the processor core and the memory system, including cache. This is a significant number of output pins. Most often, trace ports abbreviate information to reduce trace port pin count. For example, a trace port might include the IP only (no data or address information) and furthermore compress the IP output to 8 bits. The 32-bit IP can be compressed down to 8 bits by including built-in trace logic that outputs only relative offsets from an initial four-cycle output of the IP at the start of the trace. Most often, code branches locally in loops or in decision logic rather than making absolute 32-bit address branches. If the code does take a 32-bit address branch, then the internal logic can indicate this so that the external trace capture system can obtain the new IP over five or more core cycles.

Trace ports can be invaluable for difficult software defects (bugs) observable only at full-speed operation. However, a trace port is expensive, complicated, and not easy to decode and use because it requires external logic analysis equipment. As a result, internal trace buffers are being designed into most processor cores today. The internal trace buffer stores a limited number of internal state vectors, including IP, address, and data vectors, by a hardware collection state machine into a buffer that can be dumped. Trace buffers or trace registers (often the buffer is seen as a single register that can be read repeatedly to dump the full contents) are typically set up to continuously trace and stop trace on exception or on a software assert. This allows the debugger to capture data up to a crash point for a bug that is observable only during extended runtimes at full speed. This post-mortem debug tool allows a system that crashed to be analyzed to determine

what led to the crash. For extensive debug, some trace buffers are essentially built-in LAs that allow software to program them to collect a range of internally traceable data, including: (a) bus cycle data, (b) the instruction pointer, (c) address, (d) data vector, and (e) register data. The point of full-state capture is to emulate the capability that hardware/software debuggers had when all these signals could be externally probed with an LA.

12.9 Power-On Self-Test and Diagnostics

An important part of the boot firmware is the ability to test all hardware interfaces to the processor. Boot code can implement a series of diagnostic tests after a power-on reset based on a nonvolatile configuration and indicate how far it has advanced in the testing since reset through LEDs, tones, or a record in nonvolatile memory. This process is called POST (Power-On Self Tests) and provides a method for debugging boot failures. If the POST codes are well-known, then a hardware failure can be diagnosed and fixed easily. For example, if the POST code indicates that all interfaces are okay, but that memory tests failed, then replacing memory is likely to fix the problem. POST codes are also often output on a bus to memory or an external device, so probing an external bus allows you to capture POST codes and diagnose problems even when the external devices on that bus are not operating correctly. The x86 PC BIOS (Basic Input Output System) has a rich history of POST and POST code output to well-known devices and interfaces so that configuration and external device failures can be readily diagnosed and fixed [POST]. A career can be made writing and understanding BIOS firmware and, for that matter, firmware on any system. A significant part of a firmware engineer's job is simply getting a processor up and cycling monitor and diagnostic code safely so that the system can be more easily used by application programmers.

Describing how diagnostics can be written in general is difficult because the range of peripheral devices a processor might interface to is endless. However, all processors interface to memory, either internal on-chip or external off-chip memory. So, every firmware engineer should be familiar with memory testing. Memory tests include the following:

- Walking 1s test to verify processor to memory data bus interface
- Memory address bus verification

- ECC (Error Correction Circuitry) initialization and test
- Device pattern testing

The walking 1s test ensures that memory devices have been properly wired to the processor bus interface. Byte lanes for wide memory buses could be swapped, and shorts, open circuits, noise, or signal skew could all cause data to be corrupted on the memory interface. A walking 1s test should simply write data to memory with words or arrays of words that match the width of the memory data bus. A memory bus is often 128 bits or wider. Most architectures support load/store multiple or load/store string instructions that allow multiple registers to be written to the bus or read from the bus for aligned data structures. On the PowerPC G4, load/store string instructions can be used to write 128 bits to memory on a 128-bit memory bus with walking 1s to ensure that the interface is fully functional. Memory data buses wider than a single 32 bit will have a bus interface that may coalesce multiple writes to adjacent word addresses, but using the multi-word instructions helps ensure that the test uses the full bus width. Since the publication of the first edition, most all modern processors and SoCs now include vector instructions that can modify, read, and write multiple words at a time, known generally as SIMD (Single Instruction, Multiple Data)—for example, the SSE (Streaming SIMD) instructions for x86, AltiVec for PowerPC, ARM Advanced SIMD known as “NEON,” and MIPS-3D for the MIPS instruction set architecture.

The following code included on the DVD uses structure assignment in C to coax the compiler into generating load/store string instructions with the proper gcc C compiler directives.



```
#include "stdio.h"
#include "assert.h"

/* 4 x 32-bit unsigned words = 128-bit */
typedef struct multiword_s
{
    unsigned int word[4];
} multiword_t;

/* these declarations should be aligned on 128-bit boundary */
const multiword_t test_zero = {{0x0,0x0,0x0,0x0}};
volatile multiword_t test_pattern = {{0x0,0x0,0x0,0x0}};
volatile multiword_t test_location = {{0x0,0x0,0x0,0x0}};
```

```

void assign_multi(void)
{
    test_location = test_pattern;
}

void test_multi(void)
{
    register int i, j;

    for(i=0;i<4;i++)
    {
        test_pattern = test_zero;

        for(j=0;j<32;j++)
        {
            /* walk the 1 up the bits in 128-bit aligned structure
            */
            test_pattern.word[i] = (0x1 << j);

            /* structure assignment */
            assign_multi();

            /* assert if stored data does not have a bit set */
            assert(test_location.word[i]);

        }
    }
}

int main(void)
{
    test_multi();
}

```

Compiling this code on a Darwin OS PowerPC G4 Macintosh with no particular compiler directives, the `assign_multi` function code does not use load/store string instructions.

```

Sam-Siewerts-Computer:~ samsiewert$ gcc mw.c -o mw
Sam-Siewerts-Computer:~ samsiewert$ otool -v -t mw > mw.out

```

The ***otool*** is a binary utility for Darwin OS that will disassemble object code. The disassembled code for `mw.c` reveals that the compiler does not normally generate load/store string instructions. The load/store multiple instructions (`stmw` and `lmw`) are used to push and pop the stack. The 128-bit structure assignment load/store instructions are indicated in bold type.

```
_assign_multi:
00002a70 stmw   r30,0xffff8(r1)
00002a74 stwu   r1,0xffd0(r1)
00002a78 or     r30,r1,r1
00002a7c mfspr   r0,lr
00002a80 bcl     20,31,0x2a84
00002a84 mfspr   r8,lr
00002a88 mtspr   lr,r0
00002a8c addis   r9,r8,0x0
00002a90 addi    r9,r9,0x5ac
00002a94 addis   r2,r8,0x0
00002a98 addi    r2,r2,0x59c
00002a9c lwz     r0,0x0(r2)
00002aa0 lwz     r11,0x4(r2)
00002aa4 lwz     r10,0x8(r2)
00002aa8 lwz     r2,0xc(r2)
00002aac stw     r0,0x0(r9)
00002ab0 stw     r11,0x4(r9)
00002ab4 stw     r10,0x8(r9)
00002ab8 stw     r2,0xc(r9)
00002abc lwz     r1,0x0(r1)
00002ac0 lmw     r30,0xffff8(r1)
00002ac4 blr
```

Modifying the compile line a bit to request level 2 optimization and to allow load/store string code generation and disassembling again the same structure assignment in C now is done with one `lswi` and one `stswi` in place of the four `lwz` (load word and zero) and four `stw` (store word) instructions above.

```
Sam-Siewerts-Computer:~ samsiewert$ gcc -O2 -mstring mw.c -o
mw
Sam-Siewerts-Computer:~ samsiewert$ otool -v -t mw > mw.out
```

```
_assign_multi:
00002b24 mfspr   r0,lr
```

```

00002b28  bcl      20,31,0x2b2c
00002b2c  mfspr    r10,lr
00002b30  mtspr    lr,r0
00002b34  addis    r9,r10,0x0
00002b38  addis    r2,r10,0x0
00002b3c  addi     r9,r9,0x504
00002b40  addi     r2,r2,0x4f4
00002b44  lswi     r5,r2,16
00002b48  stswi    r5,r9,16
00002b4c  blr

```

Instead of coaxing the compiler into the load/store string instructions, the assembly could be written and called from the C. The most important point, however, is that the memory test should test the full bus width and not just one word at a time. The `mw.c` code has debug output that can be turned on by passing `-DDEBUG` on the compile line so that the walking 1s test patterns can be observed and so that the 128-bit alignment of the multi-word structures can be verified. Note that the addresses of the structures are all multiples of 0x10, 16 bytes, or 128 bits. By default most compilers will align structures unless specifically directed not to do so, but it is still a good idea to verify this. Some of the walking 1s output has been abbreviated here.

```

addr(test_zero) = 0x00002ff0
addr(test_pattern) = 0x00003020
addr(test_location) = 0x00003030

i=0, j=0  mword = 0x00000000000000000000000000000001
i=0, j=1  mword = 0x00000000000000000000000000000002
i=0, j=2  mword = 0x00000000000000000000000000000004
i=0, j=3  mword = 0x00000000000000000000000000000008
...
i=0, j=31 mword = 0x0000000000000000000000000008000000

i=1, j=0  mword = 0x0000000000000000000000000100000000
i=1, j=1  mword = 0x0000000000000000000000000200000000
i=1, j=2  mword = 0x0000000000000000000000000400000000
i=1, j=3  mword = 0x0000000000000000000000000800000000
...
i=1, j=31 mword = 0x00000000000000000800000000000000

```

```

i=2, j=0  mword = 0x00000000000000001000000000000000
i=2, j=1  mword = 0x00000000000000002000000000000000
i=2, j=2  mword = 0x00000000000000004000000000000000
i=2, j=3  mword = 0x00000000000000008000000000000000
...
i=2, j=31 mword = 0x00000000800000000000000000000000

i=3, j=0  mword = 0x00000001000000000000000000000000
i=3, j=1  mword = 0x00000002000000000000000000000000
i=3, j=2  mword = 0x00000004000000000000000000000000
i=3, j=3  mword = 0x00000008000000000000000000000000
...
i=3, j=31 mword = 0x80000000000000000000000000000000

```

The memory address bus can be tested by storing and retrieving patterns to addresses that are powers of two [Barr99]. By addressing with powers of two, the address tested walks 1s on the address lines. Furthermore, addresses might be aliased either due to a mistake or on purpose if not fully decoded. For example, the same 32-MB memory can be addressed in the range from 0x00000000 to 0x001FFFFFF (0x002000000 bytes) and then again from 0x002000000 to 0x003FFFFFF, and so on. This could be an address decoding error if it's intended that the memory be mapped only once in the first 32 MB of the address space and not aliased at other multiples of 0x002000000. This would happen if the address decoding included only 26 bits rather than 32 bits. Most often a pattern of alternating 1s and 0s is used (0xAA or 0x55) for each byte to ensure that bits can be set to 0 and 1 at all locations. The pattern and anti-patterns are operated on bitwise and combined to quickly verify data written and read back. This pattern and anti-pattern test can be written and read back over the entire memory range for a full device test.

12.10 External Test Equipment

Use of external test equipment for debug can be expensive, but also can save countless hours localizing a hardware/software interface bug that might be very difficult to isolate otherwise. Historically, the most common external test equipment used for verifying hardware/software systems included an oscilloscope and a logic analyzer. The oscilloscope is used mostly to isolate signaling issues with the hardware, to verify signals, and to tune

output signals. The oscilloscope is most useful to the hardware engineer, but from a systems viewpoint, it's valuable in embedded systems for verifying the outermost extents of the system from sensor inputs (analog) to actuator outputs (analog or PWM signals). Figure 12.11 shows the interface between a VxWorks embedded processor system, which captures frames from an NTSC camera, runs an image-processing service that determines where a visual target is in the camera's FOV (Field of View), and then commands tilt/pan servos to center the object in the camera's FOV. The MSO (Mixed Signal Oscilloscope) shown can probe 2 analog and 16 digital sources. Here the MSO is used to examine an NTSC output and verify the 30 fps output rate of the camera.

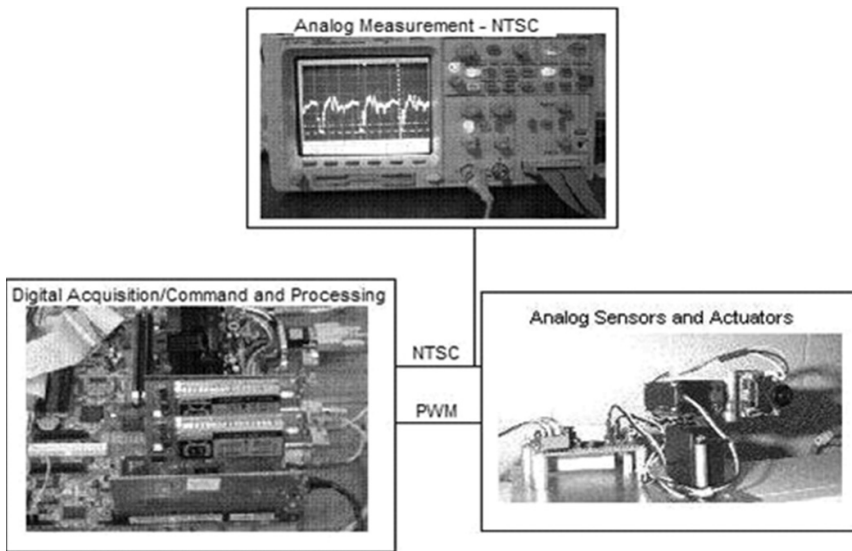


FIGURE 12.11 System Interfaces for an Embedded System

Examining the NTSC Signal for Thumbnail Camera

(Analog/Digital Interface between Processor and Sensor/Actuator Subsystem)

Likewise, a relatively inexpensive MSO can be used for low-speed limited-width digital logic analysis as well. Logic analyzers have an advantage over oscilloscopes in that they can observe 16, 32, or more signals at the same time, but only at well-defined logic levels like TTL ($V_{\text{threshold}} = 1.4\text{V}$) and CMOS ($V_{\text{threshold}} = 2.5\text{V}$). The oscilloscope can be used to look at the same channels and to see their analog nature, including noise, rise time, fall

time, and overshoot. The MSO has an advantage in that it can display logic analyzer output and one or more oscilloscope changes on the same view, allowing for logic analysis with probing of the same signals to examine analog signal issues that might be affecting digital logic.

Using the MSO to analyze an NTSC signal at milliseconds of resolution, the odd and even line raster periods can be seen at half the frame period (frame rate is 30 fps). Figure 12.12 shows the odd and even raster output signals measured with the MSO from a composite output CCTV (Closed Circuit Television) camera. NTSC is interlaced 2:1 so the field rate is 60 fps. The composite signal output from the NTSC camera combines luminance and chrominance. The signal changes based upon the scene that the camera views, but the basic period does not.

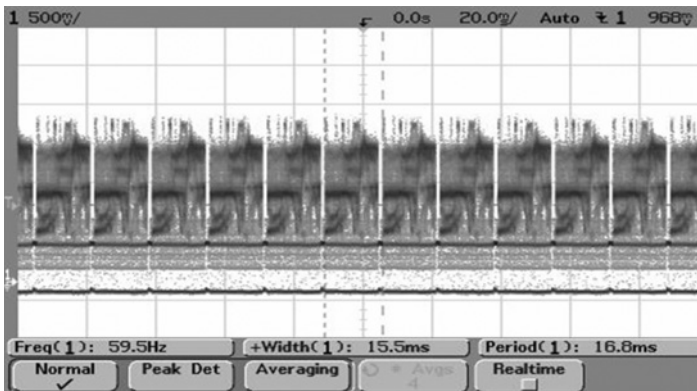


FIGURE 12.12 NTSC Composite Signal Odd and Even Scan Lines

If we use the MSO to go to a microsecond resolution, now the individual scan lines within an odd/even raster can be detected. The CCTV cameras have 510 horizontal pixels and 492 vertical pixels, so 246 odd lines are digitized and then 246 even lines. Each line therefore has a period of 67.75 microseconds in theory based upon the frame rate. Figure 12.13 shows each scan line as having a period of 61.7 microseconds. However, the simple calculation does not take into account vertical blank lines (where closed caption is inserted into the NTSC signal) or the latency between the odd and even scans. The measured period for a single scan line is less than the theoretical upper bound on the scan line period and therefore makes sense.

The MSO can also be used to analyze the tilt/pan servo control channels. A hobby servo is not rigorously standardized, but in general the servo

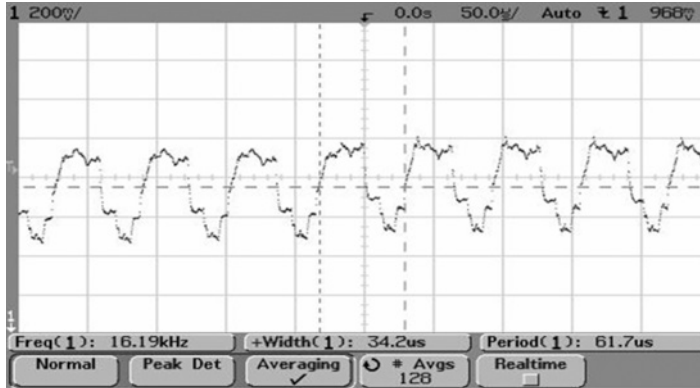


FIGURE 12.13 NTSC Composite Signal Individual Scan Lines

circuit and gearing are designed to be controlled by a PWM output with a separation between peaks of 20 to 80 milliseconds and a pulse width that varies between 2 milliseconds and 1 millisecond with servo center near 1.5 milliseconds. Servo characteristics can vary, so each new hobby servo should be individually characterized and the PWM output signal tuned to it for the best results. The spacing between the pulses allows hobby radios to multiplex multiple servo signals into one PWM output, so the receiver can de-multiplex to allow for four or more channels on a single RC (Radio Control) frequency. For the applications of hobby servos presented in this book, only one signal is carried on each PWM signal, so the spacing between pulses can vary significantly and has no effect. The pulse width sets the servo position. Figure 12.14 shows an MSO measurement of the PWM signal generated by the NCD 209 two-channel servo control chip.

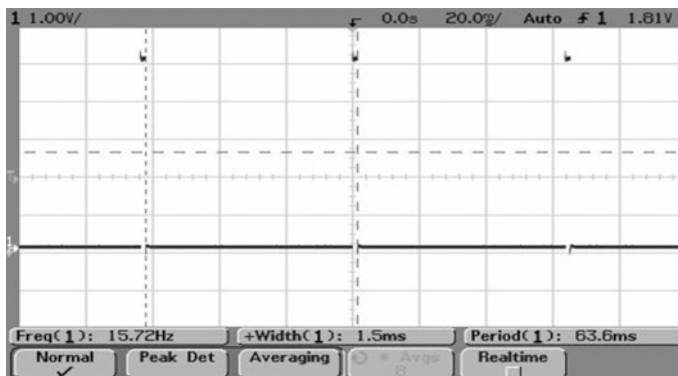


FIGURE 12.14 NCD 209 Servo Controller Default PWM Output

Note that the pulse width for the servo centering (the default output) is 1.5 milliseconds as expected and that the spacing between pulses is 63 milliseconds.

For the most part, oscilloscopes are used to verify signals, to tune generated output signals, and to look for noise issues, cross talk, or other signal integrity problems in embedded system interfaces to sensors and actuators. Logic analyzers, on the other hand, are most often used in the digital domain depicted earlier in Figure 12.11. An LA (Logic Analyzer) can be used to debug logic after analog signals have been digitized or before digital signals are output to drive DACs or PWM actuators like servos. Likewise, an LA can be used to debug internal digital logic on a processor board or even to trace events on digital software/hardware interfaces, such as buses or GPIO (General Purpose IO) where the signal level is a common logic level. Probing high-speed logic signals such as those found on a DDR memory bus can be difficult due to speed, skew issues, and complexity in decoding the logic. Probing low-speed logic, such as GPIO, or simpler memory interfaces, such as SRAM, is much simpler. High-speed specialized buses, such as PCI, are often most easily traced using a bus analyzer that is specifically designed to capture and analyze logic signals only on one bus interface. Because the trend in embedded systems is to move external memory on-chip, the most useful probing is GPIO or using external bus analyzers, such as a PCI analyzer.

With a bus analyzer or LA used to capture GPIO output, software can be instrumented to emit trace information to the external bus or GPIO pins for analysis. However, given enough memory for a software trace buffer, it's not clear how this is more advantageous than the SWIC WindView (System Viewer) style of tracing. In fact, Wind River often describes WindView as a “software logic analyzer.” A write to an external bus or GPIO memory-mapped address can be more intrusive than posting a write to on-chip or external memory. However, oscilloscopes and LAs continue to be useful in general for diagnosing hardware problems and for tracing software that is not instrumented and software/hardware interactions.

12.11 Application-Level Debugging

Application-level debugging can be done by single-stepping a thread or using WindView (System Viewer). However, for multithread applications, single-stepping is sometimes not as useful when the interaction

between threads is the problem being examined. WindView is helpful, but shows only kernel events unless it's further instrumented by the addition of `wvEvent()` calls in the application being debugged. However, `wvEvent()` has only limited information associated with it, so often programmers resort to `printf` calls so they can output state information to a console in string format. This can sometimes work okay, but often causes new bugs and can be misleading because `printf` calls significantly change execution timing. This occurs mostly because `printf` requires a large amount of code from the C library to execute and also requires potential blocking while data is output to console devices. In VxWorks, `printf` should also not be called from ISR or kernel context. The better way to debug with output to a console is using `logMsg`. Logging simply causes a message to be queued that points to a message buffer. The actual string output is completed by a logging service rather than in the calling context. This allows the console IO to be decoupled from the calling thread of execution, and the caller is slowed down only for the duration of an in-memory write and message enqueue. Furthermore, the priority of the logging service (task) can be controlled and therefore interference to real-time services by `logMsg` debug output is also controlled better than inline `printf` calls. Similarly, in Linux, `syslog` can and should be used. However, for Linux, the `printf` is fairly efficient and highly buffered, so it's not quite as intrusive as it is in VxWorks and many RTOS, which have simpler IO libraries. Likewise, for Linux kernel development, the `printk` (kernel print) is logged to a kernel log buffer that can be accessed in user space through the "dmesg" command.

Summary

Debugging the software/hardware interface, applications, and complex multi-service interactions can be difficult. Poorly synchronized services can suffer from race conditions. Likewise, poorly designed hardware/software interfaces that do not include synchronizing mechanisms can also have race conditions. In general, separate services should synchronize through semaphores or message queues if they need to interact, and likewise software should synchronize with hardware through interrupts, polling status registers, or control interfaces. Simple functional software errors can be easily caught through the use of parameter checking and assert calls. Hardware diagnostics and POST run by boot firmware can make hardware failures easier to isolate. Good practices can prevent many bugs before they be-

come difficult to diagnose. However, many bugs related to software/hardware integration will still arise; when they do, knowledge of debug monitors, trace, and external test equipment is invaluable.

Exercises

1. Use an oscilloscope to characterize a common sensor or actuator interface. Specifically, use a hobby servo controller like the NCD 209 and characterize the PWM for a given hobby servo and its limits of operation.
2. Use WindView to analyze the synthetic loading example included on the DVD. By default it includes two services that use 90% of the processor cycles. Modify this example to create and overload by increasing S_2 run time to 30 milliseconds. Use a WindView trace to prove that this overloads the processor.
3. Download and use Kernelshark on a Linux system for system tracing (as documented by http://elinux.org/images/6/64/Elc2011_rostedt.pdf, <http://man7.org/linux/man-pages/man1/kernelshark.1.html>, <http://linux.die.net/man/1/trace-cmd>). Take a trace and explain what you observe.

Chapter References

- [Barr99] M. Barr, *Programming Embedded Systems in C and C++*, O'Reilly, Sebastopol California, 1999.
- [Corbet05] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, O'Reilly, Sebastopol California, 2005.
- [Ftrace] <http://elinux.org/Ftrace>
- [GDB] <http://www.gnu.org/software/gdb/documentation/>
- [Kernelshark] <http://rostedt.homelinux.com/kernelshark/>
- [LTT] <http://www.opersys.com/LTT/>
- [LTTng] <http://lttng.org/>
- [POST] <http://bioscentral.com/>
- [PowerPC] "PowerPC Microprocessor Family: The Programming Environments for 32-bit Microprocessors," Motorola MPCFPE32B/AD Rev. 1, 1997.

[Stap] <https://sourceware.org/systemtap/>

[WRS06] “Wind River Workbench Product Note,” <http://windriver.com/products/product-notes/workbench-product-note.pdf>

[Yaghmour03] K. Yaghmour, *Building Embedded Linux Systems*, O’Reilly, Sebastopol California, 2003.

[Zen] <http://www.macraigor.com/zenofbdm.pdf>

PERFORMANCE TUNING

In this chapter

- Introduction
- Basic Concepts of Drill-Down Tuning
- Hardware-Supported Profiling and Tracing
- Building Performance Monitoring into Software
- Path Length, Efficiency, and Calling Frequency
- Fundamental Optimizations

13.1 Introduction

The art of performance tuning a system or application is a huge topic that can't be covered completely in this chapter. The chapter covers basic methods and provides enough knowledge for you to pursue a more in-depth understanding of performance tuning on your own with further study. Performance tuning is critical to real-time embedded systems when services are missing deadlines. Otherwise, efficient, high-performance execution of services is not necessary, although it can help the designer to save cost, reduce power, and simplify thermal cooling for systems. Fundamentally, a real-time system is operating correctly when it produces the required output (functions) and produces it by a specific deadline relative to request. Early in a project, system designers must size processing based upon understanding of the complexity of service algorithms. After algorithms are implemented and the system is running, services may take longer to run than expected and overrun deadlines. When this happens, there are several options:

- Reduce the complexity of the algorithm
- Reduce the frequency of services
- Increase the execution efficiency of the algorithms with tuning
- Increase the processor throughput by upgrading hardware to run with a faster clock, more bandwidth, more memory, and less latency

From the standpoint of real-time correctness, any of these options is acceptable. Reducing the complexity of the algorithms for services may be an option, but might degrade the overall system quality. For example, software codecs (compression and decompression protocol for a data stream) can provide high rates of compression at the cost of higher algorithmic complexity. If the compression or decompression can't keep up with the desired frame rate, the options include a lower-performance codec that is simpler and can be executed at the desired frame rate, or a lower frame rate. The simpler codec that provides less compression will require transport with higher bandwidth.

Often the options of reducing performance (e.g., lower frame rate) or of obtaining more resources (e.g., more bandwidth or increasing processor cycle rate) are not feasible or will not meet the system requirements. An alternative to more resources or performance reduction is performance tuning to optimize the efficiency of service execution. This is easily said and hard to do. Ideally, a system design should not count heavily upon performance tuning to make it feasible, but tuning can save a project that would otherwise not work. From an RM perspective, tuning is just reducing WCET for a software service. Another option is to offload a service or specific functions that the service performs to hardware state machines. Offloading functions can increase overall system concurrency and provide significant speedup. ("Big Iron Lessons, Part 3: Performance Monitoring and Tuning" by Sam Siewert. First published by IBM at IBM developerWorks Rational. (www.ibm.com/developerWorks/rational). All rights retained by IBM and the author(s).)

13.2 Basic Concepts of Drill-Down Tuning

The performance of firmware and software must be tuned to a workload. A *workload* is a sequence of service requests, commands, IOs, or other quantifiable transactions that exercise the software. Workloads most often are produced by workload generators rather than real-world service provi-

sion. Good workloads capture the essential features of real-world workloads and allow for replay or generation of the service requests at a maximum rate so that bottlenecks can be identified in systems. Most workload-generation tools are application- or service domain-specific—for example, compiler benchmarks or IO subsystem workloads generated by IOmeter, a commonly used disk IO workload generator.

Characteristics of code segments that most affect performance include the following:

- Segment path length (instruction count)
- Segment execution efficiency (cycle count for a given path)
- Segment calling frequency (in Hertz)
- Execution context (critical path, or error path)

The critical path includes code executed at high frequency for performance evaluation workloads and benchmarks. It is often a small portion of the overall code implemented and can also often fit into Level 1 instruction or L2 unified cache.

Interrupt-based profiling is the best place to start analysis of software services and deadline overruns. Interrupt profiling requires that the system be run with a well-defined test workload, ideally a workload that stresses the services and is causing deadline overruns. The interrupt profiler samples the IP (Instruction Pointer) or LR (Link Register) to determine location in code on a periodic basis. The LR is widely used in processor architectures because it contains a return from interrupt address and the samples of where the code was in execution are taken by interrupt. Observing the IP in the sample interrupt routine is of little use for profiling. Often a 1-millisecond sample rate is used. The workload should be repetitive at some frequency—for example, a video frame rate of a specific resolution, a data request rate for a specific transfer size, a digital control loop sensor sample and control law output rate, or any workload that generates uniform periodic service requests. Multiple workloads can be analyzed, but while profiling, one workload should be run at a time. Mixed workloads are much harder to profile and understand. While a uniform workload is being run, the profiler will sample the execution locations and with asynchronous sampling relative to the workload period (this is important) the profile will begin to stabilize and show where in the code most of the time is spent for the given workload.

Where time is spent, which service and function the IP or LR is observed in most often are a function of three characteristics of the services:

1. Frequency of service execution (or function execution)
2. Instruction count or path length for each service release (or function)
3. Efficiency of code execution in each service release (or function)

Note that the profiling can be mapped to services (tasks), to functions, to lines of C code, or even to individual machine code instructions. Figure 13.1 depicts how a profile collected at the level of hits observed at a 32-bit code address can be mined to understand where time is being spent from a very high level down to a machine code instruction.

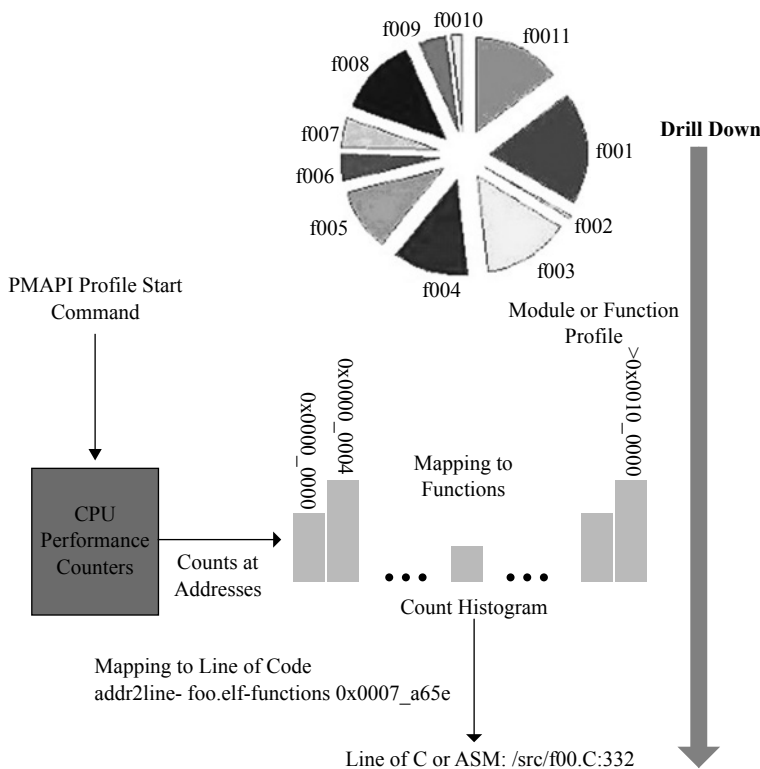


FIGURE 13.1 Performance Profile Data Mining (Hot Spot Drill-Down) Concept

As mentioned in the introduction, frequency and instruction count reduction can solve the performance problem, but at the cost of overall system performance. Ideally the goal is to identify inefficient code, or hot

spots as they are often called, and to make modifications to make those code blocks execute more efficiently. A profile that is cycle-based, where periodic sampling of the IP or LR is used, however, does identify hot spots in general and sets up more detailed analysis for why those code blocks are hot—frequency of execution, number of instructions in the block, or poor execution efficiency. Poor execution efficiency can be the result of stalling a processor pipeline, missing cache, blocking reads from slow devices, misaligned data structures, non-optimal code generation from a compiler, bad coding practices, inefficient bus usage, and unnecessary computation.

One approach to tuning is to go through code from start to finish and to look for efficiency issues, correct them, and try running again. This is tedious. It can work, but the time required, the slow payoff, and the large number of code changes required all contribute to risk with this approach. A better approach is to identify the hot spots, narrowing the scope of code changes significantly, and then examine this subset of the code for potential optimization. Combined with profiling and hot spot identification, optimization now has a feedback loop so that the value of optimizations can be measured as code is modified. The feedback approach reduces risk (modifying any working code always runs the risk of introducing a bug or destabilizing the code base) and provides focus for the effort. For any tedious effort it is also affirming to be able to measure progress, and rerunning the profile and observing that less time is spent in a routine after an optimization are welcome feedback. It should be noted that high-level performance measures often show not apparent improvement for localized optimizations. An optimization may lead to a hurry-up-and-wait scenario for that particular function or service, where a bottleneck elsewhere in the system masks the improvement.

The overall process being proposed is called drill-down. First, hot spots are identified without regard to why they are hot, and then closer analysis is used to determine why specific sections of service code or the system are rate-limiting. This initial level of analysis is important since it can help drive focus on the truly rate-limiting software blocks. If the software is not the rate-limiting bottleneck in the system, then profiling will still help, but will result in arriving at blocking locations more quickly. If the bottlenecks are related to IO latencies, this will still show up as a hot spot, but the optimization required may require code redesign or system redesign. For example, status registers that are read and respond slowly to the read request will stall execution if the read data is used right away. Reading hardware status into a cached state long before the data must be processed will reduce stall

time. Ideally the hardware/software interface would be modified to reduce the read latency in addition. Many system-level design improvements can be made to avoid IO bottlenecks—for example, the use of fast access tightly coupled memory so that the impact of cache misses (the miss penalty) is minimized or, in some cases, for single-cycle access memory, eliminated.

Adjusting hardware resources may often be impossible, especially during the hardware/software integration phase of a project. If you need more single-cycle on-chip memory, a huge cost and time investment is necessary to adjust this resource. Instead, if you can modify firmware and software to minimize the loss of performance through a bottleneck, you might reach improvement without hardware changes. System architects should ideally determine what their bottleneck will be and then plan for scalability. Firmware and software engineers with tuning tools and some hardware support built into their systems can ensure that maximum performance is achieved for any given set of hardware resource constraints. The rest of this chapter discusses common built-in CPU performance measurement hardware features and how you can use them to tune firmware and software for optimal performance. Simple interrupt-based profiling will help, but the ability to separate hot spots out based upon frequency, instruction count, and execution efficiency requires some built-in hardware assist. The good news is that almost all modern processor cores now include PMUs (Performance-Monitoring Units) or performance counters that can provide even better vision into performance issues than interrupt-based profiles.

Ideally, after a workload set is identified and performance optimizations are being considered for inclusion into a code base, ongoing performance regression testing should be in place. Performance regression testing should provide simple high-level metrics to evaluate current performance in terms of transactions or IOs per second. Also, some level of profiling should ideally be included, perhaps supported by performance counters. The generation of this data should be automatic and should be able to be correlated to specific code changes over time. In the worst case, this can allow for backing out code optimizations that did not work and for quick identification of code changes or features added that adversely affect performance.

13.3 Hardware-Supported Profiling and Tracing

Some basic methods for building performance-monitoring capability into the hardware include the following:

- Performance event counters (PMU)
- Execution trace port (branch-encoded)
- Trace register buffer (branch-encoded)
- Cycle and interval counters for time-stamping

This chapter section focuses on built-in event counters and how they can be used to profile firmware and software to isolate execution-efficiency hot spots. Tracing is advised once hot spots are located. In general, *tracing* provides shorter-duration visibility into the function of the CPU and firmware, but can be cycle-accurate and provide a view of the exact order of events within a CPU pipeline. Trace is often invaluable for dealing with esoteric timing issues and hardware or software interaction bugs, and you can also use it to better understand performance bottlenecks. A profile provides information that is much like an initial medical diagnosis. The profile answers the question, where does it hurt? By comparison, while a profile can't tell the tuner anything about latency, a trace will provide a precise measure of latency. Profiling supported by performance counters can indicate not only where time is spent but also hot spots where cache is most often missed or code locations where the processor pipeline is most often stalled. A trace is needed to determine stall duration and to analyze the timing at a stall hot spot.

Understanding latency for IO and memory access can enable better overlap of processing with IO. One of the most common optimizations is to queue work and start IO early so that the CPU is kept as busy as possible. So, while profiling is the most popular approach to tuning, tracing features should still be designed into the hardware for debug and performance tuning as well.

Many (though not all) common CPU architectures include trace ports or the option to include a trace port. The IBM and AMCC PowerPC 4xx CPUs, Xilinx Virtex-II Pro, ARM Embedded Trace Macrocell, and Tensilica are among the chips that fall into this category. The 4xx series of processors has included a branch-encoded trace port for many years. The branch-encoded trace port makes a nice compromise between visibility into the core and pin-count coming off-chip to enable tracing. Because the trace is encoded, it is not cycle-accurate, but is accurate enough for hardware or software debugging and for performance optimization. Given modern EDA (Electronic Design Automation) tools, ASIC design verification

has advanced to the point where all significant hardware issues can be discovered during simulation and synthesis. SoC (System on Chip) designs make post-silicon verification difficult if not impossible, since buses and CPU cores may have few or no signals routed off-chip. For hardware issues that might arise with internal on-chip interfaces, the only option is to use built-in logic analyzer functions, such as those provided by the Virtex-II Pro ChipScope. This type of cycle-accurate trace is most often far more accurate than is necessary for tuning the performance of firmware or software.

Encoded-branch trace, such as that provided by the IBM and AMCC PowerPC 4xx, typically requires only eight IO pins per traceable core. The encoding is based upon trace output that on a core-cycle basis provides information on interrupt vectors, relative branches taken, and the occasional absolute branch taken. Most branches and vectors that the PC (Program Counter) follows are easily encoded into 8 bits. Typical processors might have at most 16 interrupt vectors, requiring only 4 bits to encode the interrupt source. Furthermore, relative branches for loops, C switch blocks, and if blocks typically span short address ranges that might require 1 or 2 bytes to encode. Finally, the occasional long-branch to an absolute 32-bit address will require 4 bytes. So, overall, encoded output is typically 1 to 5 bytes for each branch point in code. Given that most code has a branch density of 1 in 10 instructions or less, the encoded output, which can take up to five cycles to output an absolute branch, is still very accurate from a software execution order and timing perspective.

The program counter is assumed to linearly progress between branch points. So, the branch trace is easily correlated to C or assembly source code after it is acquired through a logic analyzer or acquisition tool such as RISCTrace (see references for more information on this tool). Due to the high rate of acquisition, which is nearly the cycle rate of the CPU, the trace window for a trace port will be limited. For example, even a 64MB trace buffer would capture approximately 16 million branch points, or about 200 million instructions. At a clock rate of 1 GHz, that's only one fifth of a second of code execution. This information is invaluable for hardware and software debugging and timing issues, as well as direct measurement of latencies. However, most applications and services provide a large number of software operations per second over long periods of time. For visibility into higher-level software performance, profiling provides information that is much easier to use than the information tracing provides. After a profile is understood, trace can provide an invaluable level of drill-down to understand poorly performing sections of code.

Performance counters first appeared in the IBM PowerPC architecture in a patent approved in 1994. Since then, the manufacturers of almost all other processor architecture have licensed or invented similar features. The Intel PMU (Performance-Monitoring Unit) is a well-known example in wide use, perhaps most often used by PC game developers. The basic idea is simple. Instead of directly tracing code execution on a CPU, a built-in state machine is programmed to assert an interrupt periodically so that an ISR (Interrupt Service Routine) can sample the state of the CPU, including the current address of the PC (Program Counter). Sampling the PC address is the simplest form of performance monitoring and produces a histogram of the addresses where the PC was found when sampled. This histogram can be mapped to C functions and therefore provides a profile indicating the functions in which the PC is found most often.

What does this really tell you? It's an indication of calling frequency, of the size of a function (larger functions have larger address bins), and of the number of cycles that are spent in each function. With 32-bit-word-sized address bins, the profile can provide this information down to the instruction level and therefore by line of C code. Most performance counter state machines also include event detection for CPU core events, including cache misses, data dependency pipeline stalls, branch mispredictions, write-back queue stalls, and instruction and cycle counts.

These events, like periodic cycle-based sampling, can also be programmed to assert an interrupt for ISR sampling of current PC and related event counts. This event profile can indicate address ranges (modules, functions, lines of code) that have *hot spots*. A hot spot is a code location where significant time is spent or where code is executing poorly. For example, if a particular function causes a cache miss counter to fire the sampling interrupt frequently, this indicates that the function should be examined for data and instruction cache inefficiencies. Finally, from these same event counts it is also possible to compute metrics, such as CPI (Clocks Per Instruction), for each function with simple instrumentation added to entry and exit points. This use of counters with inline code to sample those counters (instrumentation) is a hybrid approach between tracing and profiling, often referred to as *event tracing*. The performance counters require a hardware cell built into the CPU core, but also require some firmware and software support to produce a profile or event trace. If the cost, complexity, or schedule prevents inclusion of hardware support for performance monitoring, pure software methods can be used instead, as you'll see next.

13.4 Building Performance Monitoring into Software

Most of the basic methods for building performance-monitoring capability into a system require firmware and software support:

- Performance counter API (hardware-supported)
- Direct code instrumentation to sample cycle and event counters for trace (hardware-supported)
- Steady-state asynchronous sampling of counters through interrupts (hardware-supported)
- Software event logging to memory buffer (software only, but hardware time stamp improves quality)
- Function or block entry/exit tracing to a memory buffer (software only, with post-build modification of binary executables)

Software event logging is a pure software in-circuit approach for performance monitoring that requires no special hardware support. Most embedded operating system kernels provide an event logging method and analysis tools, such as the WindView and Linux Trace Toolkit event analyzers. At first, this approach might seem really intrusive compared to the hardware-supported methods. However, modern architectures, such as the PowerPC, provide features that make event logging traces very efficient. Architectures such as the PowerPC include posted-write buffers for memory writes, so that occasional trace instructions writing bit-encoded event codes and time stamps to a memory buffer take no more than a single instruction. Given that most functions are on the order of hundreds to thousands of instructions in length (typically a line of C code generates multiple assembly instructions), a couple of additional instructions added at function entry and exit will contribute little overhead to normal operation.

Event trace logs are invaluable for understanding operating system kernel and application code event timing. While almost no hardware support is needed, an accurate time stamp clock will make the trace timing more useful. Without a hardware clock, it is still possible to provide an event trace showing only the order of events, but the inclusion of microsecond or better accuracy timestamps vastly improves the trace. System architects should carefully consider hardware support for these well-known and well-tested tracing and profiling methods, as well as scheduling time for software development.

When you implement performance-monitoring hardware and software on an embedded system, the ability to collect huge amounts of data is almost overwhelming. Effort put into data collection is of questionable value in the absence of a plan to analyze that data. Trace information is typically the easiest to analyze and is taken as needed for sequences of interest and mapped to code or to a timeline of operating system events. When traces are mapped to code, engineers can replay and step through code as it runs at full speed, noting branches taken and overall execution flow. For event traces mapped to operating system events, engineers can analyze multithread context switches made by the scheduler, thread synchronizing events, such as semaphore takes and gives, and application-specific events.

Profiles that collect event data and PC location down to the level of a 32-bit address take more analysis than simple mapping to be useful. Performance tuning effort can be focused well through the process of drilling down from the code module level, to the function level, and finally to the level of a specific line of code. Drilling down provides the engineer analyst with hot spots where more detailed information, such as traces, should be acquired. The drill-down process ensures that you won't spend time optimizing code that will have little impact on bottom-line performance improvement.

13.5 Path Length, Efficiency, and Calling Frequency

Armed with nothing but hardware support to time stamp events, it is still possible to determine code segment path length and execution efficiency. Ideally, performance counters would be used to automate the acquisition of these metrics. However, when performance counters aren't available, you can measure path length by hand in two different ways. First, by having the C compiler generate assembly code, you can then count the instructions by hand or by a word count utility. Second, if a single-step debugger is available (e.g., a cross-debug agent or JTAG, Joint Test Applications Group), then you can count instructions by stepping through assembly by hand. Though this is laborious, it is possible, as you'll see by looking at some example code.

The code in Listing 13.1 generates numbers in the Fibonacci sequence.

LISTING 13.1 Simple C Code to Compute the Fibonacci Sequence

```

typedef unsigned int UINT32;
#define FIB_LIMIT_FOR_32_BIT 47

UINT32 idx = 0, jdx = 1;
UINT32 seqCnt = FIB_LIMIT_FOR_32_BIT, iterCnt = 1;
UINT32 fib = 0, fib0 = 0, fib1 = 1;

void fib_wrapper(void)
{
    for(idx=0; idx < iterCnt; idx++)
    {
        fib = fib0 + fib1;
        while(jdx %lt; seqCnt)
        {
            fib0 = fib1; fib1 = fib; fib = fib0 + fib1;
            jdx++;
        }
    }
}

```

The easiest way to get an instruction count for a block of code such as the Fibonacci sequence generating function in Listing 13.1 is to compile the C code into assembly. With the GCC C compiler, this is easily done with the following command line:

```
$ gcc fib.c -S -o fib.s
```

The resulting assembly is illustrated in Listing 13.2. Even with the automatic generation of the assembly from C, it's still no easy task to count instructions by hand. For a simple code block such as this example, hand counting can work, but the approach becomes time-consuming for real-world code blocks.

If you're looking for a less tedious approach than counting instructions by hand from assembly source, you can use a single-step debugger and walk through a code segment in disassembled format. An interesting side effect of this approach for counting instructions is that the counter often learns quite a bit about the flow of the code in the process. Many single-step debugging tools, including JTAG (Joint Test Applications Group) hardware debuggers, can be extended or automated so that they can automatically

step from one address to another, keeping count of instructions in between. Watching the debugger auto-step and count instructions between an entry point and exit point for code can even be an instructive experience that may spark ideas for path optimization.

LISTING 13.2 GCC-Generated ASM for Fibonacci C Code

```
.globl _fib_wrapper
.section __TEXT,__text,regular,pure_instructions
.align 2
_fib_wrapper:
    stmw r30,-8(r1)
    stwu r1,-48(r1)
    mr r30,r1
    mflr r0
    bcl 20,31,"L000000000001$pb"
"L000000000001$pb":
    mflr r10
    mtlr r0
    addis r2,r10,ha16(_idx-"L000000000001$pb")
    la r2,lo16(_idx-"L000000000001$pb") (r2)
    li r0,0
    stw r0,0(r2)
L2:
    addis r9,r10,ha16(_idx-"L000000000001$pb")
    la r9,lo16(_idx-"L000000000001$pb") (r9)
    addis r2,r10,ha16(_Iterations-"L000000000001$pb")
    la r2,lo16(_Iterations-"L000000000001$pb") (r2)
    lwz r9,0(r9)
    lwz r0,0(r2)
    cmplw cr7,r9,r0
    blt cr7,L5
    b L1
L5:
    addis r11,r10,ha16(_fib-"L000000000001$pb")
    la r11,lo16(_fib-"L000000000001$pb") (r11)
    addis r9,r10,ha16(_fib0-"L000000000001$pb")
    la r9,lo16(_fib0-"L000000000001$pb") (r9)
    addis r2,r10,ha16(_fib1-"L000000000001$pb")
    la r2,lo16(_fib1-"L000000000001$pb") (r2)
    lwz r9,0(r9)
    lwz r0,0(r2)
```

```

    add r0,r9,r0
    stw r0,0(r11)
L6:
    addis r9,r10,ha16(_jdx-"L00000000001$pb")
    la r9,lo16(_jdx-"L00000000001$pb")(r9)
    addis r2,r10,ha16(_seqIterations-"L00000000001$pb")
    la r2,lo16(_seqIterations-"L00000000001$pb")(r2)
    lwz r9,0(r9)
    lwz r0,0(r2)
    cmplw cr7,r9,r0
    blt cr7,L8
    b L4
L8:
    addis r9,r10,ha16(_fib0-"L00000000001$pb")
    la r9,lo16(_fib0-"L00000000001$pb")(r9)
    addis r2,r10,ha16(_fib1-"L00000000001$pb")
    la r2,lo16(_fib1-"L00000000001$pb")(r2)
    lwz r0,0(r2)
    stw r0,0(r9)
    addis r9,r10,ha16(_fib1-"L00000000001$pb")
    la r9,lo16(_fib1-"L00000000001$pb")(r9)
    addis r2,r10,ha16(_fib-"L00000000001$pb")
    la r2,lo16(_fib-"L00000000001$pb")(r2)
    lwz r0,0(r2)
    stw r0,0(r9)
    addis r11,r10,ha16(_fib-"L00000000001$pb")
    la r11,lo16(_fib-"L00000000001$pb")(r11)
    addis r9,r10,ha16(_fib0-"L00000000001$pb")
    la r9,lo16(_fib0-"L00000000001$pb")(r9)
    addis r2,r10,ha16(_fib1-"L00000000001$pb")
    la r2,lo16(_fib1-"L00000000001$pb")(r2)
    lwz r9,0(r9)
    lwz r0,0(r2)
    add r0,r9,r0
    stw r0,0(r11)
    addis r9,r10,ha16(_jdx-"L00000000001$pb")
    la r9,lo16(_jdx-"L00000000001$pb")(r9)
    addis r2,r10,ha16(_jdx-"L00000000001$pb")
    la r2,lo16(_jdx-"L00000000001$pb")(r2)
    lwz r2,0(r2)
    addi r0,r2,1
    stw r0,0(r9)

```

```

        b L6
L4:
        addis r9,r10,ha16(_idx-"L00000000001$pb")
        la r9,lo16(_idx-"L00000000001$pb") (r9)
        addis r2,r10,ha16(_idx-"L00000000001$pb")
        la r2,lo16(_idx-"L00000000001$pb") (r2)
        lwz r2,0(r2)
        addi r0,r2,1
        stw r0,0(r9)
        b L2
L1:
        lwz r1,0(r1)
        lmw r30,-8(r1)
        blr

```

Listing 13.3 provides a main program that can be compiled for G4- or G5-based Macintosh computers running Mac OS X. You can download the CHUD toolset for the Mac and use it with the MONster instrumentation included in Listing 13.3 to measure the cycle and instruction counts using the hardware performance counters. The example code in Listing 13.3 takes the Fibonacci code from Listing 13.1 and adds sampling of the PowerPC G4 performance counters through the CHUD interface to the MONster analysis tool. The same types of cycle and event counters are found in the PowerPC performance counters in embedded IBM PowerPC cores.

LISTING 13.3: Simple C Code for PowerPC to Compute CPI for Fibonacci Code Block

```

#include "stdio.h"
#include "unistd.h"
// This code will work on the Macintosh G4 PowerPC with the Mon-
ster
// PowerPC Performance Counter acquisition and analysis tool.
// Simply pass in -DMONSTER_ANALYSIS when compiling the example.
// MONSTER provides a fully featured set of PMAPI counters and
// analysis along with the full suite of CHUD tools for the Macintosh
// G series of PowerPC processors.
//
// Alternatively on x86 Pentium machines that implement the Time Stamp
// Counter in the x86 version of Performance Counters called the PMU,

```

```

// pass in -DPMU_ANALYSIS. For the Pentium, only CPU cycles will be
// measured and CPI estimated based upon known instruction count.
//
// For the Macintosh G4, simply launch the main program from a Mac OS
// shell with the MONSTER analyzer set up for remote monitoring
// to follow along with the examples in the article.
//
// Leave the #define LONG_LONG_OK if your compiler and architecture
// support 64-bit unsigned integers, declared as unsigned long long in
// ANSI C.

//
// If not, please remove the #define below for 32-bit unsigned
// long declarations.
//

#define LONG_LONG_OK
#define FIB_LIMIT_FOR_32_BIT 47

typedef unsigned int UINT32;

#ifdef MONSTER_ANALYSIS
#include "CHUD/chud.h"
#include "mach/boolean.h"

#else

#ifdef LONG_LONG_OK
typedef unsigned long long int UINT64;

UINT64 startTSC = 0;
UINT64 stopTSC = 0;
UINT64 cycleCnt = 0;

UINT64 readTSC(void)
{
    UINT64 ts;

    __asm__ volatile(".byte 0x0f,0x31" : "=A" (ts));
    return ts;
}

```

```

UINT64 cyclesElapsed(UINT64 stopTS, UINT64 startTS)
{
    return (stopTS - startTS);
}

#else
typedef struct
{
    UINT32 low;
    UINT32 high;
} TS64;

TS64 startTSC = {0, 0};
TS64 stopTSC = {0, 0};
TS64 cycleCnt = 0;

TS64 readTSC(void)
{
    TS64 ts;
    __asm__ volatile(".byte 0x0f,0x31" : "=a" (ts.low), "=d" (ts.high));
    return ts;
}

TS64 cyclesElapsed(TS64 stopTS, TS64 startTS)
{
    UINT32 overFlowCnt;
    UINT32 cycleCnt;
    TS64 elapsedT;

    overFlowCnt = (stopTSC.high - startTSC.high);

    if(overFlowCnt && (stopTSC.low < startTSC.low))
    {
        overFlowCnt--;
        cycleCnt = (0xffffffff - startTSC.low) + stopTSC.low;
    }
    else
    {
        cycleCnt = stopTSC.low - startTSC.low;
    }

    elapsedT.low = cycleCnt;

```



```

    elapsedT.high = overFlowCnt;

    return elapsedT;
}
#endif
#endif

UINT32 idx = 0, jdx = 1;
UINT32 seqIterations = FIB_LIMIT_FOR_32_BIT;
UINT32 reqIterations = 1, Iterations = 1;
UINT32 fib = 0, fib0 = 0, fib1 = 1;

#define FIB_TEST(seqCnt, iterCnt) \
    for(idx=0; idx < $!t; iterCnt; idx++) \
    { \
        fib = fib0 + fib1; \
        while(jdx < seqCnt) \
        { \
            fib0 = fib1; \
            fib1 = fib; \
            fib = fib0 + fib1; \
            jdx++; \
        } \
    } \

void fib_wrapper(void)
{
    FIB_TEST(seqIterations, Iterations);
}

#ifdef MONSTER_ANALYSIS
char label[]="Fibonacci Series";

int main( int argc, char *argv[])
{
    double tbegin, telapse;

    if(argc == 2)
    {
        sscanf(argv[1], "%ld", &reqIterations);

        seqIterations = reqIterations % FIB_LIMIT_FOR_32_BIT;
        Iterations = reqIterations / seqIterations;
    }
}

```

```

    }
    else if(argc == 1)
        printf("Using defaults\n");
    else
        printf("Usage: fibtest [Num iterations]\n");

    chudInitialize();
    chudUmarkPID(getpid(), TRUE);
    chudAcquireRemoteAccess();
    tbegin=chudReadTimeBase(chudMicroSeconds);

    chudStartRemotePerfMonitor(label);
    FIB_TEST(seqIterations, Iterations);
    chudStopRemotePerfMonitor();

    telapse=chudReadTimeBase(chudMicroSeconds);

    printf("\nFibonacci(%lu)=%lu (0x%08lx) for %f usec\n",
        seqIterations, fib, fib, (telapse-tbegin));

    chudReleaseRemoteAccess();
    return 0;
}

#else
#define INST_CNT_FIB_INNER 15
#define INST_CNT_FIB_OUTTER 6

int main( int argc, char *argv[] )
{
    double clkRate = 0.0, fibCPI = 0.0;
    UINT32 instCnt = 0;

    if(argc == 2)
    {
        sscanf(argv[1], "%ld", &reqIterations);

        seqIterations = reqIterations % FIB_LIMIT_FOR_32_BIT;
        Iterations = reqIterations / seqIterations;
    }
    else if(argc == 1)
        printf("Using defaults\n");

```

```

else
    printf("Usage: fibtest [Num iterations]\n");

    instCnt = (INST_CNT_FIB_INNER * seqIterations) +
        (INST_CNT_FIB_OUTTER * Iterations) + 1;

    // Estimate CPU clock rate
    startTSC = readTSC();
    usleep(1000000);
    stopTSC = readTSC();
    cycleCnt = cyclesElapsed(stopTSC, startTSC);

#ifdef LONG_LONG_OK
    printf("Cycle Count=%llu\n", cycleCnt);
    clkRate = ((double)cycleCnt)/1000000.0;
    printf("Based on usleep accuracy, CPU clk rate = %lu clks/
sec,",
        cycleCnt);
    printf(" %7.1f Mhz\n", clkRate);
#else
    printf("Cycle Count=%lu\n", cycleCnt.low);
    printf("OverFlow Count=%lu\n", cycleCnt.high);
    clkRate = ((double)cycleCnt.low)/1000000.0;
    printf("Based on usleep accuracy, CPU clk rate = %lu clks/
sec,",
        cycleCnt.low);
    printf(" %7.1f Mhz\n", clkRate);
#endif

    printf("\nRunning Fibonacci(%d) Test for %ld iterations\n",
        seqIterations, Iterations);

    // START Timed Fibonacci Test
    startTSC = readTSC();
    FIB_TEST(seqIterations, Iterations);
    stopTSC = readTSC();
    // END Timed Fibonacci Test

#ifdef LONG_LONG_OK
    printf("startTSC =0x%016x\n", startTSC);
    printf("stopTSC =0x%016x\n", stopTSC);

```

```

    cycleCnt = cyclesElapsed(stopTSC, startTSC);
    printf("\nFibonacci(%lu)=%lu (0x%08lx)\n", seqIterations, fib,
fib);
    printf("\nCycle Count=%llu\n", cycleCnt);
    printf("\nInst Count=%lu\n", instCnt);
    fibCPI = ((double)cycleCnt) / ((double)instCnt);
    printf("\nCPI=%4.2f\n", fibCPI);

#else
    printf("startTSC high=0x%08x, startTSC low=0x%08x\n", startT-
SC.high, startTSC.low);
    printf("stopTSC high=0x%08x, stopTSC low=0x%08x\n", stopTSC.
high, stopTSC.low);

    cycleCnt = cyclesElapsed(stopTSC, startTSC);
    printf("\nFibonacci(%lu)=%lu (0x%08lx)\n", seqIterations, fib,
fib);
    printf("\nCycle Count=%lu\n", cycleCnt.low);
    printf("OverFlow Count=%lu\n", cycleCnt.high);
    fibCPI = ((double)cycleCnt.low) / ((double)instCnt);
    printf("\nCPI=%4.2f\n", fibCPI);
#endif

}
#endif

```

Running the code from Listing 13.3 built for the Macintosh G4, the MONster analysis tool determines the path length and execution efficiency for the Fibonacci sequence code block, as summarized in Listing 13.4.

The example analysis in Listing 13.4 was collected using the MONster configuration and source code easily compiled with GCC and downloadable from the DVD.



LISTING 13.4 Sample Macintosh G4 MONster Analysis for Example Code

```

Processor 1: 1250 MHz PPC 7447A, 166 MHz CPU Bus, Branch Folding:
enabled, Threshold:
0, Multiplier: 2x
(tb1) P1 - Timebase results
(plc1) PMC 1:   1 - CPU Cycles
(plc2) PMC 2:   2 - Instr Completed

```

Config: 5 - CPI-simple

1 - CPI (completed)

P1 Timebase (cpu cycles)	P1 pmc 1	P1 pmc 2	SC res 1
Tbl: (cpu cycles)	(P1) 1-CPU Cycles:	(P1) 2-Instr Completed:	CPI (completed)
8445302.308861008	8413160	1618307	5.19874164790735
Tbl: (cpu cycles)	(P1) 1-CPU Cycles:	(P1) 2-Instr Completed:	CPI (completed)
7374481.710107035	7346540	1360873	5.398402349080333
Tbl: (cpu cycles)	(P1) 1-CPU Cycles:	(P1) 2-Instr Completed:	CPI (completed)
7207497.309806291	7180243	1668938	4.302282649205663
Tbl: (cpu cycles)	(P1) 1-CPU Cycles:	(P1) 2-Instr Completed:	CPI (completed)
44985850.44768739	44808388	38595522	1.160973752343601
Tbl: (cpu cycles)	(P1) 1-CPU Cycles:	(P1) 2-Instr Completed:	CPI (completed)
472475463.2072901	470597098	458561558	1.026246290797887
Tbl: (cpu cycles)	(P1) 1-CPU Cycles:	(P1) 2-Instr Completed:	CPI (completed)
2149357027.379779	2140806560	2108619999	1.015264277591631

```
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 100
Fibonacci(6)=13 (0x0000000d) for 7545.213589 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 10000
Fibonacci(36)=24157817 (0x01709e79) for 5883.610250 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 1000000
Fibonacci(28)=514229 (0x0007d8b5) for 6008.113638 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 100000000
Fibonacci(27)=317811 (0x0004d973) for 36554.381943 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 10000000000
Fibonacci(31)=2178309 (0x00213d05) for 378554.531618 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 1000000000000
Fibonacci(17)=2584 (0x00000a18) for 1720178.701376 usec
Sam-Siewerts-Computer:~/TSC samsiewert$
```

Note that in the example runs, for larger numbers of iterations, the Fibonacci sequence code becomes cached and the CPI drops dramatically. If the Fibonacci code were considered to be critical-path code for performance, you might want to consider locking this code block into Level 1 instruction cache.



If you don't have access to a PowerPC platform, an alternative Pentium TSC (time stamp counter) build, which counts CPU core cycles and uses a hand-counted instruction count to derive CPI, is included on the DVD. You can download and analyze this code on any Macintosh Mac OS X PC or Windows PC running Cygwin tools. On the Macintosh, using MONster with the "Remote" feature, the application will automatically upload counter data to the MONster analysis tool, including the cycle and instruction counts for the Fibonacci code block.

13.6 Fundamental Optimizations

Given analysis of code such as the example Fibonacci sequence code block, how do you take the information and start optimizing the code? The best approach is to make gains by following the “low-hanging fruit” model of optimization—apply optimizations that require the least effort and change to the code, but provide the biggest improvements first. Most often this means simply ensuring that compiler optimizations that can be used are being used. After compiler optimizations are exhausted, then simple code restructuring might be considered to eliminate cache miss hot spots. The optimization process is often architecture-specific. Most instruction set architectures include application programming notes with ideas for code optimization. While the methods for optimizing go beyond the scope of this introductory chapter, some of the most common are summarized here.

Here are some basic methods for optimizing code segments:

- Use compiler basic block optimizations (inline, loop unrolling, and so on).
- Simplify algorithm complexity and unnecessary instructions in code.
- Compute commonly used expressions up front.
- Lock critical-path code into L1 instruction cache.
- Lock critical-path, high-frequency reference data into L1 data cache.
- Take advantage of memory prefetch—prefetch data to be read and modified into L1 or L2 cache.
- Use MMIO prefetch—start IO for data used in algorithms early to avoid data dependency pipeline stalls.

After you’ve exhausted some of these optimization methods using profiling to support identification of the blocks most in need of optimization, you need to consider more complex optimization methods for additional improvement. Good functional regression testing should be included in the optimization process to ensure that functionality is not broken by optimizing code changes. This is especially true for more complex architecture-specific optimizations. It should also be noted that architecture-specific optimizations make code much less portable.

Here are some more advanced methods for optimizing code segments:

- Use compiler feedback optimization—profile provided as input to compiler.
- Hand-tune assembly to exploit features such as conditional execution.

Summary

After the current marriage of software, firmware, and hardware is optimized, how does the architect improve the performance of the next generation of the system? At some point, for every product, the cost of further code optimization will outweigh the gain of improved performance and marketability of the product. However, you can still use performance analysis features, such as performance counters, to characterize potential modifications that should go into next-generation products. For example, if a function is highly optimized, but remains as a high consumer of CPU resources, and the system is CPU-bound such that the CPU is the bottleneck, then hardware re-architecting to address this would be beneficial. That particular function might be considered for implementation in a hardware-accelerating state machine. Or perhaps a second processor core could be dedicated to that one function. The point is that performance counters are useful not only for coaxing maximum performance out of an existing hardware design but also for guiding future hardware design.

Exercises

1. Use the Pentium TSC code to count the cycles for a given code block. Then, in the single-step debugger, count the number of instructions for the same code block to compute the CPI for this code on your system. Provide evidence that your CPI calculation is accurate, and explain why it is as high or low as you find it to be.
2. Compare the accuracy for timing the execution of a common code block, first using the DVD time_stamp.c code and then using the Pentium TSC. Is there a significant difference? Why?
3. Download the Brink and Abyss Pentium-4 PMU profiling tools, and profile the Fibonacci code running on a Linux system. Explain your results.



Chapter References

[Siewert] S. Siewert, “Big Iron Lessons, Part 3: Performance Monitoring and Tuning”, IBM developerWorks, April 2005.

HIGH AVAILABILITY AND RELIABILITY DESIGN

In this chapter

- Introduction
- Reliability and Availability Similarities and Differences
- Reliability
- Reliable Software
- Design Trade-Offs
- Hierarchical Approaches for Fail-Safe Design

14.1 Introduction

High availability and high reliability (HA/HR) are measured differently, but both provide some indication of system robustness and quality expectations. Furthermore, because system failures can be dangerous, both also relate to system safety. Increasing availability and reliability of a system requires time and monetary investment, increasing cost, and increasing time-to-market for products. The worst thing that can come of engineering efforts to increase system availability and reliability is an unintentional decrease in overall availability and reliability. When design for HA/HR is not well tested or adds complexity, this unintentional reduction in HA/HR can often be the result. In this chapter, design methods for increasing HA/HR are introduced along with a review of exactly what HA and HR really mean. (“title” by <author’s name>. First published by IBM at IBM developerWorks Rational. (www.ibm.com/developerWorks/rational). All rights retained by IBM and the author(s).)

14.2 Reliability and Availability Similarities and Differences

Availability is simply defined as the percentage of time over a well-defined period that a system or service is available for users. So, for example, if a system is said to have 99.999%, or *five nines*, availability, this system must not be unavailable more than five minutes over the course of a year. Quick recovery and restoration of service after a fault greatly increase availability. The quicker the recovery, the more often the system or service can go down and still meet the five nines criteria. Five nines is a *high availability* or HA metric.

In contrast, high reliability (HR) is perhaps best described by the old adage that a chain is only as strong as its weakest link. A system built from components that have very low probability of failure leads to high system reliability. The overall expected system reliability is simply the product of all subsystem reliabilities, and the subsystem reliability is a product of all component reliabilities. Based upon this mathematical fact, components are required to have very low probability of failure if the subsystems and system are to also have reasonably low probability of failure. For example, a system composed of 10 components, each with 99.999% reliability, is $(0.99999)^{10}$, or 99.99%, reliable. Any decrease in the reliability of a single component in this type of single-string design can greatly reduce overall reliability. For example adding just one 95% reliable component in the previous example that was 99.99% reliable would drop the overall reliability to 94.99%. Highly reliable components are often constructed of higher-quality raw materials, subjected to more rigorous testing, and often have more complex fabrication processes, all increasing component cost. There may be notable exceptions and advancements where lower-cost components also have higher reliability, making component choice obvious, but this is not typical.

Once simple mathematical relationship between HA and HR is:

$$\text{Availability} = \text{MTBF} / (\text{MTBF} + \text{MTTR})$$

MTBF = Mean Time Between Failures, and $\text{MTBF} \approx \text{MTTF}$

MTTR = Mean Time to Recovery

MTTF = Mean Time to Failure (How long the system is expected to run without failure)

Alternately, if MTTR is large such that MTBF is significantly larger than MTTF, then the mathematical relationship should be:

$$\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$

So, while an HR system has large MTTF, it can tolerate a longer MTTR and still yield decent availability when $\text{MTBF} \approx \text{MTTF}$. Likewise a system with low reliability can provide decent availability if it has very fast recovery times. From this relation we can see that one system with five nines' HA might have low reliability and a very fast recovery (say 1 second) and therefore go down and recover 3,000 or more times in a year (10 times a day!). A highly reliable system might go down once every two years and wait for operator intervention to safely and carefully recover operational state, taking on average 10 minutes every two years for the exact same HA metric of five nines. Clearly these systems would not be considered to have the same quality by most operators.

As noted in Chapter 7, "Soft Real-Time Services," it is more accurate to use MTTF instead of MTBF. The MTBF may be confused with MTTF on occasion, and for small MTTR, MTBF and MTTF differ only by MTTR. For many systems the MTTR is seconds or at most minutes and the MTTF is often hundreds of thousands or millions of hours, so in practice, $\text{MTBF} \approx \text{MTTF}$ as long as $\text{MTTR} \ll \text{MTBF}$. Often to compute five nines' availability MTBF can be used when MTTR is small for quick system recovery [Siewert05], and in these cases, MTTF can be approximated as MTBF. Care should be taken to ensure that the failure rates and time between failures provided by manufacturers of components are well understood. Either way, the key to five nines' high availability is short MTTR and infrequent failures over time.

14.3 Reliability

It is theoretically possible to build a system with low-quality, not-so-reliable components and subsystems, and still achieve HA. This type of system would have to include massive redundancy and complex switching logic to isolate frequently failing components and to bring spares online very quickly in place of those components that failed to prevent interruption to service. Most often, it is better to strike a balance and invest in more reliable components to minimize the interconnection and switching requirements. If you take a very simple example of a system designed with redundant components that can be isolated or activated, it becomes clear

that the interconnection and switching logic does not scale well to high levels of redundancy and sparing. Consider the simple, single-spare dual-string system shown in Figure 14.1.

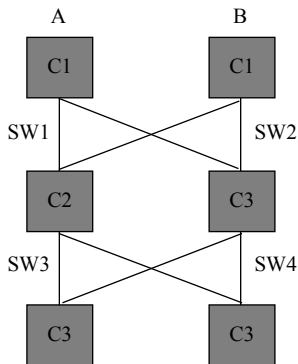


FIGURE 14.1 Dual-String, Cross-Strapped Subsystem Interconnection

The example system shown in Figure 14.1 has eight possible configurations that can be formed by activating and isolating components C1, C2, or C3 from side A or side B. The system must be able to positively detect failed components and track these failures to reconfigure with an operable switch state and new interconnection of activated components. Table 14.1 describes the eight possible configurations for this small-scale HA system example.

TABLE 14.1 Enumeration of Configurations for Three-Subsystem Dual-String Cross-Strapped System

Configuration	Component C1	Component C2	Component C3
1	A	A	A
2	A	A	B
3	A	B	A
4	A	B	B
5	B	A	A
6	B	A	B
7	B	B	A
8	B	B	B

From the simple example described by Figure 14.1, it is evident that a trade-off can be made between the complexity of interconnecting compo-

nents and redundancy management with the cost of including highly reliable components. The cost of hardware components with high reliability is fairly well known and can be estimated based upon component testing, expected failure rates, MTBF (mean time between failures), operational characteristics, packaging, and the overall physical features of the component.

System architects should also consider three simple parameters before investing heavily in HA or HR for a system component or subsystem:

1. Likelihood of unit failure
2. Impact of failure on the system
3. Cost of recovery versus cost of fail-safe isolation

Conceptually, architects should consider how levels of recovery are handled with varying degrees of automation, as depicted in Figure 14.2.

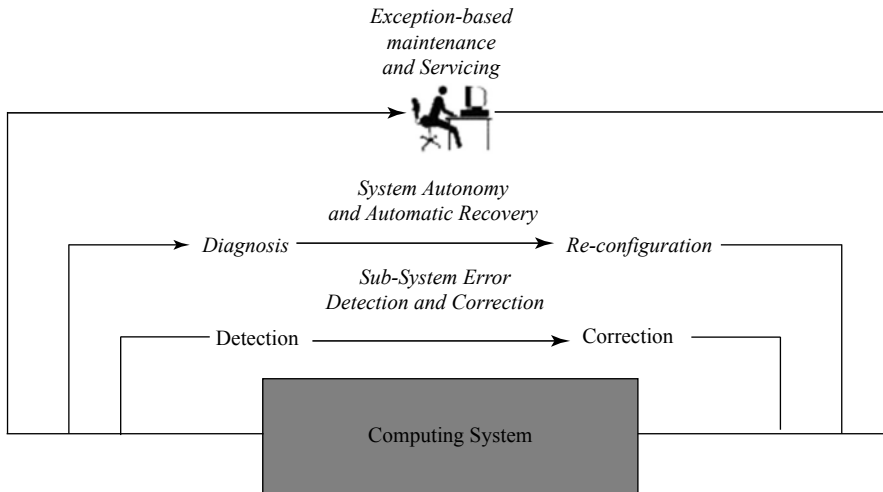


FIGURE 14.2 Supporting Multiple Levels of Recovery Autonomy

14.4 Reliable Software

Perhaps much harder to estimate is the cost of highly reliable software. Clearly, reliable hardware running unreliable software will result in failure modes that are likely to cause service interruption. It is well accepted that complex software is often less reliable, and that the best way to increase

reliability is with testing. Testing takes time and ultimately adds to cost and time to market.

System architects have traditionally focused on designing HA and HR hardware with firmware to manage redundancy and to automate recovery. So, for example, firmware would reconfigure the components in the example in Table 14.1 to recover from a component failure. Traditionally, rigorous testing and verification have ensured that firmware has no flaws, but history has shown that defects can still wind up in the field and emerge due to subtle differences in timing or execution of data-driven algorithms.

Designing firmware and software for HR can be costly. The FAA requires rigorous documentation and testing to ensure that flight software on commercial aircraft is highly reliable. The DO-178B class A standard (now replaced by DO-178C) requires software developers to maintain copious design, test, and process documentation. The updated DO-178C is just as rigorous and in fact adds more objectives to levels A, B, and C. The new standard also includes clarity updates. The point of the standards is to ensure that design, test, and process used in software development for aviation meet minimum criteria. Furthermore, testing must include formal proof that code has been well tested with criteria such as multiple condition decision coverage (MCDC). This criterion ensures that all paths and all statements in the code have been exercised and shown to work. It is very laborious and therefore greatly increases the cost of software components.

14.5 Design Trade-Offs

Designing for HR alone can be cost-prohibitive, so most often a balance of design for HA and HR is better. HA at the hardware level is most often achieved through redundancy (sparing) and switching. A trade-off can be made between the cost of duplication and simply engineering higher reliability into components to reduce the MTBF. Over time, designers have found a balance between HR and HA features to optimize cost and availability. Fundamental to duplication schemes is the recovery latency. When considering component or subsystem duplication for HA, architects must carefully consider the complexity and latency of the recovery scheme and how this will affect firmware and software layers. Trade-offs between working to simply increase reliability instead of increasing availability through sparing should be analyzed. A simple well-proven methodology that is often employed by systems engineers is to consider the trade-off of probability of

failure, impact of failure, and cost to mitigate impact or reduce likelihood of failure. This method is most often referred to as FMEA (Failure Modes and Effects Analysis).

Another, less formal process that system engineers often use is referred to as the “low-hanging fruit” process. This process simply involves ranking system design features under consideration by cost to implement, reliability improvement, availability improvement, and complexity. The point of low-hanging fruit analysis is to pick features that improve HA/HR the most for least cost and with least risk. Without existing products and field-testing, the hardest part of FMEA or low-hanging fruit analysis is estimating the probability of failure for components and estimating improvement to HA/HR for specific features. For hardware, the tried and true method for estimating reliability is based upon component testing, system testing, environmental testing, accelerated testing, and field-testing. The trade-offs between engineering reliability and availability into hardware are fairly obvious, but how does this work with firmware and software?

Designing and implementing HR firmware and software can be very costly. The main approach for ensuring that firmware/software is highly reliable is verification with formal coverage criteria, along with unit tests, integration tests, system tests, and regression testing. Test coverage criteria include feature points; but for HR systems, much more rigorous criteria are necessary, including statement, path, and multiple condition decision coverage (MCDC).

One might wonder why simply designing test cases for path and statement coverage is not sufficient for HR software. The simple snippet of C code in Listing 14-1 shows why MCDC is required. The if test in `main()` has two expressions logically ordered together. Most C compilers will generate code such that the second expression is short-circuited if the first evaluates to true. As a result, both paths in `main()` for the if blocks can be driven by a test without ever executing (testing) the `OutsideLimits` code. So, a test driver must drive the same path with the condition where `recoveryRequired` is false and where `recoveryRequired` is true and `OutsideLimits` is either true or false in combination with this. So, for simple path coverage in `main` there are only two paths noted by coverage of code on the line numbers: Path-A) 28, 30, 32 and Path-B) 28, 30, 13, 20, 36. However, by MCDC you must ensure that `main` Path-A is covered in combination with the paths of the function `OutsideLimits`, which defines Path-C) 28, 30, 13, 15, 16, 32.

LISTING 14.1 Simple C Code with Two Paths and MCDC Testing Requirements

```

01: #define UPPER_LIMIT 100
02: #define TRUE 1
03: #define FALSE 0
04:
05: extern void logMessage(char *msg);
06: extern unsigned int MMIORead(unsigned int addr);
07: extern void StartRecovery(void);
08: extern void ContinueOperation(void);
09: extern int RecoveryRequired;
10:
11: int LimitTest(unsigned int val)
12: {
13:     if(val > UPPER_LIMIT)
14:     {
15:         logMessage("Limit Exceeded\n");
16:         return TRUE;
17:     }
18:     else
19:     {
20:         return FALSE;
21:     }
22: }
23:
24: main()
25: {
26:     unsigned int IOValue = 0;
27:
28:     IOValue = MMIORead(0xF0000100);
29:
30:     if(RecoveryRequired || OutsideLimits(IOValue))
31:     {
32:         StartRecovery();
33:     }
34:     else
35:     {
36:         ContinueOperation();
37:     }
38: }

```

14.6 Hierarchical Approaches for Fail-Safe Design

Ideally, all system, subsystem, and component errors can be detected and corrected in a hierarchy so that component errors are detected and corrected without any action required by the containing subsystem. This hierarchical approach for fault detection and fault protection/correction can greatly simplify verification of an RAS design. An ECC memory component provides for single-bit error detection and automatic correction. The incorporation of ECC memory provides a component level of RAS, which can increase RAS performance and reduce the complexity of supporting RAS at higher levels. HR systems often include design elements that ensure that non-recoverable failures result in the system going out of service, along with safing to reduce risk of losing the asset, damaging property, or causing loss of life.

Summary

Systems designed for high availability may not necessarily be designed for high reliability. Highly reliable systems tend to be highly available, although they often fail-safe and wait for human intervention for recovery rather than risking automatic recovery. So, there is no clear correlation between the HA and HR other than the equation for availability in terms of MTBF and MTTR. As discussed, given that high MTBF and long MTTR compared to low MTBF and rapid MTTR could yield the same availability, it appears that availability alone does not characterize the safeness or quality of a system very well.

Exercises

1. How many configurations must a fault recovery system attempt for a dual string fully cross-strapped system with four subsystems?
2. Describe what you think a good fail-safe mode would be for an earth orbiting satellite that encounters a non-recoverable multi-bit error when it passes through the high radiation zone known as the South Atlantic Anomaly. What should the satellite do? How should it eventually be recovered for continued operation?

3. If it takes 30 seconds for Windows to boot, how many times can this OS fail due to deadlock or loss of software sanity in a year for five nines HA, ignoring detection time for each failure?
4. If the launch vehicle for satellites has an overall system reliability of 99.5% and has over 2,000 components, what is the average reliability of each component?
5. Research the CodeTest software coverage analysis tool, and describe how it provides test coverage criteria. Can it provide proof of MCDC coverage?

Chapter References

- [Campbell02] Iain Campbell, *Reliable Linux: Assuring High Availability*, Wiley, 2002.
- [Siewert05] S. Siewert, “Big Iron Lessons, Part 2: Reliability and availability: What’s the difference?”, IBM developerWorks, March 2005.

PART

3

PUTTING IT ALL TOGETHER

- Chapter 15 System Life Cycle
- Chapter 16 Continuous Media Applications
- Chapter 17 Robotic Applications
- Chapter 18 Computer Vision Applications

SYSTEM LIFE CYCLE

In this chapter

- Introduction
- Life Cycle Overview
- Requirements
- Risk Analysis
- High-Level Design
- Component Detailed Design
- Component Unit Testing
- System Integration and Test
- Configuration Management and Version Control
- Regression Testing

15.1 Introduction

Real-time embedded systems include at the very least mechanical, electrical, and software engineering to build a working and deliverable system. Some real-time embedded systems may involve chemical, biological, cryogenic, optical, or many other specialized subsystems and components as well that must be integrated and tested in addition to the more common mechanical, electrical, and software subsystems. For example, blood analysis machines provide real-time analysis of human blood samples using a robotic platform that can automate the biological testing of a large number of

samples in much less time than manual laboratory testing. The NASA great observatory series of space telescopes all include spacecraft and instrumentation real-time embedded systems that include visible, X-ray, infrared, and ultra-violet detectors. The infrared instrumentation requires cryogenic cooling. Likewise, real-time embedded systems are often found in process control applications at chemical, nuclear, or biological production plants. Given the wide range and complexity of disciplines, types of components, and types of subsystems that must be integrated to field a successful real-time embedded system, it's imperative that a well-defined and disciplined process be followed during all stages of development. This chapter provides a framework for real-time embedded systems life cycle planning, tips on tools that can help, and examples of methods used.

In this chapter, we'll walk through selected elements of a design for a stereo computer vision real-time tracking system. Completion of a project like this can be accomplished in 15 weeks if you want to engage in home-study practice to better understand the process of engineering a real-time embedded system.

15.2 Life Cycle Overview

Having a good understanding of critical steps in the system-engineering process for real-time embedded systems long before a project is started can make projects go much more smoothly. Many in-depth texts describe the life cycle phases and steps in much more detail than this book, but often don't provide the full context of engineering hardware, firmware, and software components and subsystems. Furthermore, we must learn how to ensure that all components and subsystems are integrated, debugged, and optimized to realize a quality system rather than just one aspect, such as software or hardware alone. Often trade-offs between hardware and software can be made with significant cost, risk, schedule, or quality impact if the process promotes periodic critical examination of the system as it evolves.

The spiral model for systems engineering (often presented as a software engineering process) is well suited for following a rigorous process, yet also ensuring that the system evolution is adaptive to ensure that optimizations can be made as lessons are learned during the process. Figure 15.1 shows a two phase spiral model for implementing a system starting from concept and resulting in delivery of the first functional system.

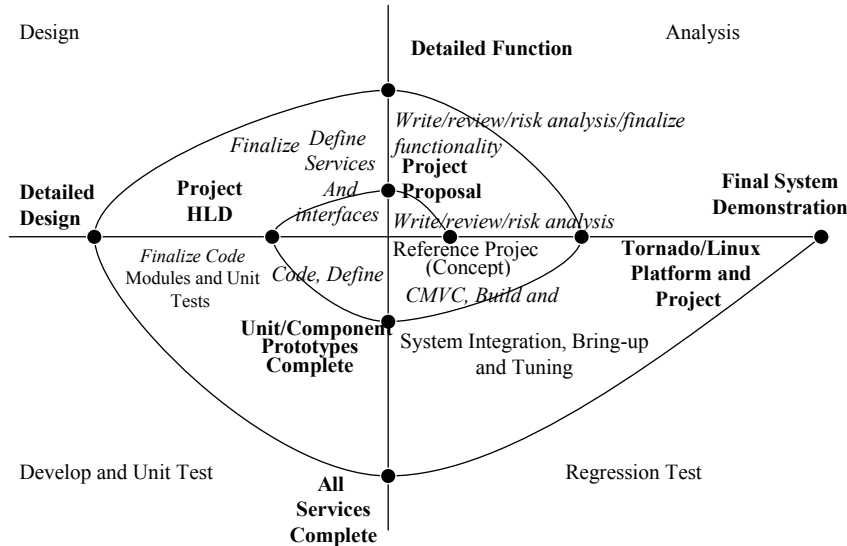


FIGURE 15.1 Spiral Process for Systems Engineering

Figure 15.1 shows the process used by students at the University of Colorado to build computer vision, robotics, and real-time media applications in a 15-week time period. This may seem very short, but often in industry, a new product concept must be quickly demonstrated and an early version put in a customer's hands to generate demand and to define the market, sometimes in less than 15 weeks as well. The beauty of the spiral model is that it can include many phases (turns through analysis, design, development and unit testing, and regression testing) for more complex systems and fit a wide range of systems development schedules. The main concept behind the spiral is to mitigate risk by analyzing the system requirements and design every turn through the spiral. The theory behind this is that a better risk assessment can be made with some implementation and testing experience completed and that implementation and test can be developed through an evolutionary process. Furthermore, the spiral model should include increased time and monetary investment with each successive turn as risk is better understood and managed with better insight from previous phases of the spiral.

In theory, spirals can be as short or long as the participants choose, from one day to one year, but ideally they are short phases initially so that pitfalls in requirements, design, implementation, or testing can be discovered sooner rather than later. Experience at the University of Colorado

has shown that this process works well for students developing complicated systems with imperfect information, limited budgets, and very limited time. You can use the process in Figure 15.1 to guide development of a home or an academic system and can expand the process for use with an industrial project.

The starting point is a reference design. This reference design should be a system that is the closest known system to the one proposed. The reference system can be a resource for design, code, subsystem, component, and test reuse. In the first analysis phase, the new system can be defined in terms of functional, performance, and feature-level requirements (perhaps leveraging the reference system), along with an analysis of risk, often associated with new features, functions, or new levels of performance required. This first analysis phase involves writing requirements, reviewing them, and estimating risks and cost. The second phase, design, involves using design tools and methods to define the high-level system design, clearly identifying the subsystems and interfaces between them, but not yet defining all components. The third phase, development and unit test, involves implementing software stubs (basic interface and simplified behavior) and hardware prototypes (emulators) to discover how well high-level design concepts will work and to better estimate risk and cost. The fourth phase, regression, involves a preliminary integration (or practice integration) of subsystem emulators and stubs, and the establishment of software nightly builds and testing. This practice integration also allows for better definition of not only unit tests but also how integrated subsystems and the system will be tested as the system is assembled.

The second turn of the spiral model repeats the same four phases, with the second analysis phase involving refinement of requirements and more accurate re-estimation of risk and cost. For short development projects, such as a demonstration or academic development project, the second turn may be the final turn through the spiral process to deliver a working system. If this is the case, then detailed design must be completed down to the component level, with implementation and unit test at the same level, and a regression phase that concludes with full system integration and prerelease product testing. For a longer-term project, the second turn may not complete design and implementation. The evolutionary spiral can be designed so that a subset of features are implemented in the second turn and a third, fourth, or more turns are used to complete the features and to achieve desired performance and quality.

For the purpose of a class or self-study project, Figure 15.2 shows the same spiral development process fit to 15 weeks.

The time spent in each phase of the spiral is doubled or tripled in the second turn. This schedule also works well in an academic setting or for a rapid prototype development project where status must be provided by engineers (students) to management (teachers) for quick feedback during the first turn.

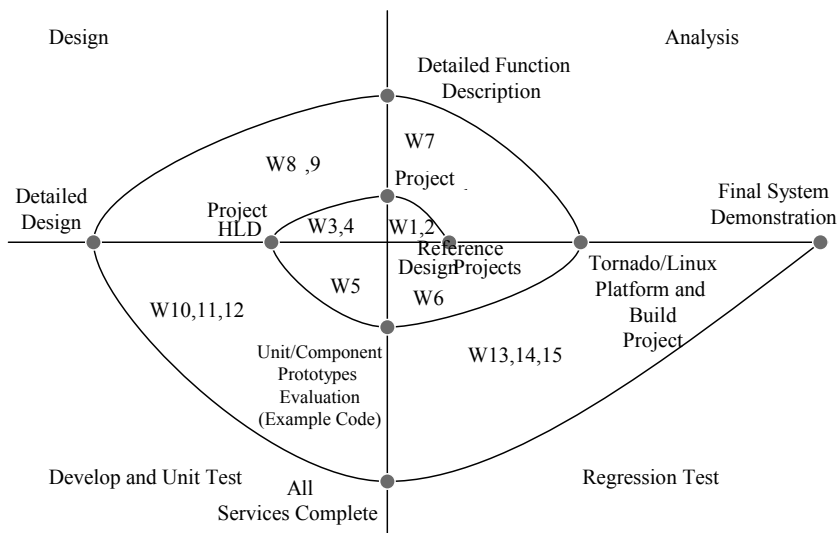


FIGURE 15.2 Fifteen-Week Spiral Process for Systems Engineering

15.3 Requirements

Defining requirements for a system during analysis is most often done by listing or enumerating functions, features, and performance metrics for the ultimate system. In the first turn of the spiral, requirements are defined in terms of subsystems and features with a few high-level performance metrics that will be evident to any user of the system. These are then refined to component level during subsequent turns through the spiral. One approach to define requirements and design with agility and quickness is to include the high-level requirements and design in one design document, making the tracing of requirements to design simple. This can also be done for detailed requirements definition and design, once again making trace simple. For large-scale projects, often an engineer will work only on

requirements, and a design engineer will iterate with the requirements engineer to achieve a design that maps to all the requirements. For the purpose of the 15-week development project, some subset of system requirements might be developed by a team, and then subsystem requirements and design developed by each engineer.

15.4 Risk Analysis

As requirements are defined, the risk of each requirement should be reviewed. Risk can be analyzed by the following:

- Probability of occurrence of a failure or process problem
- The impact of the same failure or process problem
- The cost of mitigating the risk item

This makes risk analysis simple. Risk can be reduced most significantly for the overall system design and development process by focusing on the most probable and highest-impact risk items first. These risks can then be further analyzed and mitigations proposed along with the cost of those mitigations to determine what goals are for the first or next spiral turn. For example, if development of a new driver for a new video encoder is properly realized as high risk for a 15-week project, this risk can be mitigated by investigating existing drivers and several off-the-shelf options for video encoder chips. This could be done quickly in the first spiral, and a recovery plan can be defined for the second turn. This is an example of a process risk. Likewise, a system risk can be handled in a similar way. For example, servos have limits in actuation quickness and accuracy. Characterization of a number of servos during the first turn of the spiral to determine limits for candidate servo components can help significantly reduce risk of not meeting performance requirements for a tilt/pan tracker in the end. Risk analysis should always accompany requirements analysis and should always follow each spiral turn.

15.5 High-Level Design

High-level design involves the enumeration and definition of all hardware and software interfaces and their characteristics as well as decomposition of the system into subsystems. If components can be identified during

this phase, this is ideal. High-level design methods should involve unified hardware/software design as well as individual design. A system design showing all system major elements should be agreed upon by the development team and be defined at a high enough level that it can fit on one page as shown in Figure 15.3.

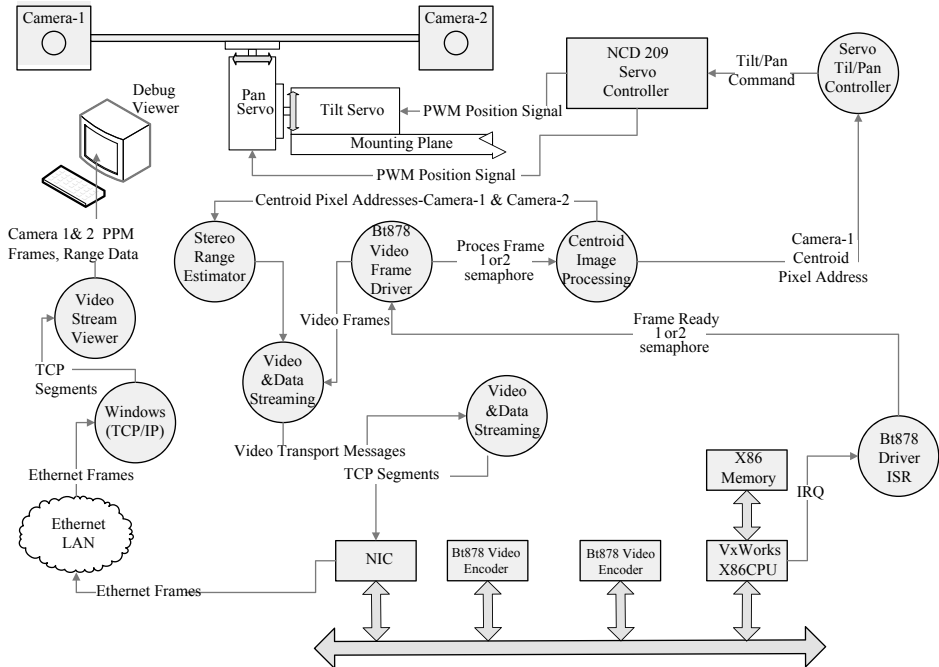


FIGURE 15.3 Stereo-Vision Example System Design

Although the single-page suggestion may be difficult and may lead to a busy block diagram, it provides a chance for the team or individual to look at all the major elements at once. The design can always be subdivided and cleaned up after this exercise, but the single-page view is often valuable throughout the project. A system view should show both hardware and software elements and the data flow between them with a high level of abstraction. The key for this design artifact is completeness and agreement by all involved that it is complete. To immediately clean up this busy view, the same diagram can be decomposed into a hardware-only view or software-only view. Figure 15.4 shows the hardware-only view for the Figure 15.3 system view.

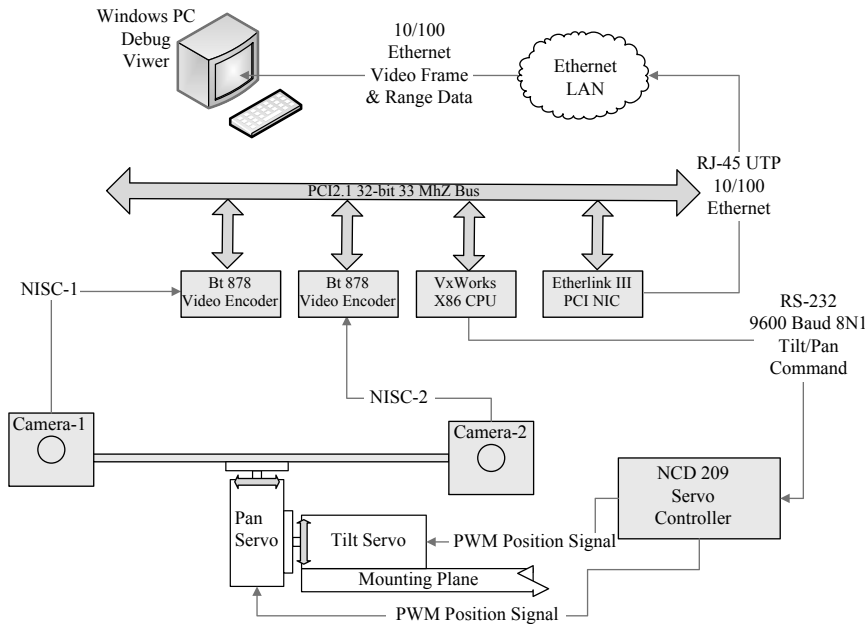


FIGURE 15.4 Stereo-Vision Example Hardware System View

Note that Figure 15.4 shows not only the electrical but also the mechanical hardware view. Once again, this can be broken down to show only mechanical views. In this way, subsystems are identified in a consistent manner and with a keystone (the system views) that allow a team to continually reevaluate the efficacy of the overall design.

In high-level design the fundamental concept is completeness rather than accuracy. For example, mechanical drawings might not include dimensions, software module and service sequence diagrams might not define all calling parameters or message attributed, and electrical schematics might not have all resistor values. The point is to fully define subsystems and their interfaces to set the stage for early testing and for detailed design in the next spiral. Methods used for design are specific to disciplines below the subsystem level (detailed design level), so the high-level design is the opportunity to ensure that the system can be integrated to meet system requirements and a good system-level design mitigates risk considerably. Often it is advisable to design to a level lower than subsystems to ensure that the subsystem high-level design can be advanced with more confidence. This also allows for early identification of components and can support early component characterization and testing for systems which will be built from off-the-shelf components and subsystems.

Numerous methodologies support system-level design, including the following:

- UML – Universal Modeling Language
- SDL – Specification and Description Language
- Structured Analysis
- Block diagrams
- Data flows

The DVD included with the text contains the start for a high-level system design for the stereo vision project using UML with the Visio UML template.



This start on a system design can be completed to practice use of the UML methodology. Both UML and SDL are well suited formalisms for system design. Structured analysis methods and block diagramming are less formal, but these methods can be used successfully if applied with rigor to ensure completeness as described in this chapter. Mostly, successful projects choose a methodology and stick to it, using the methodology framework to communicate design and to allow for team reviews and assessment of completeness.

15.6 Component Detailed Design

Detailed design must go down to the component level and divide the system into design elements specific to hardware and software. Specific design methods are more appropriate for mechanical, electrical, or software components and subsystems. Figure 15.5 shows a dimensioned cut-away mechanical design for a two-servo tilt/pan camera tracking subsystem. The figure shows only the mechanical design for this subsystem, and during detailed design review, the electrical and software detailed design for this subsystem should likewise show similar detail.

Figure 15.5 should show sufficient detail so that the mechanical subsystem can be assembled from the design drawings and information. Clearly, even more detail would be needed, including all dimensions, material types, fasteners, adhesives, servo part numbers, and perhaps even assembly instructions. The less detail included, the more chance that some unspecified aspect of the subsystem design will lead to a design flaw. For example,

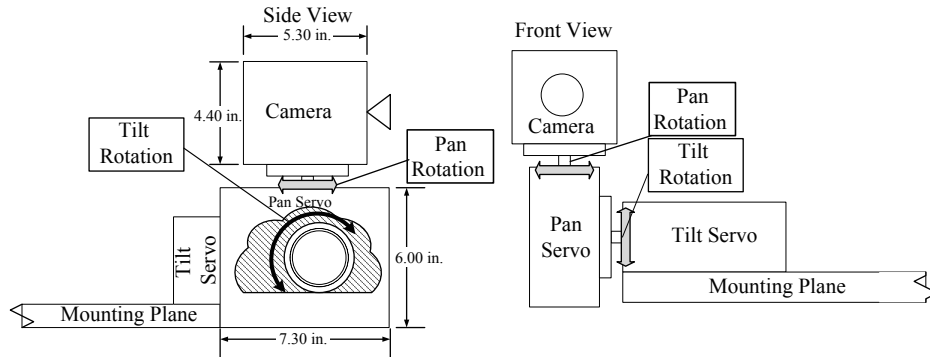


FIGURE 15.5 Detailed Mechanical Tilt/Pan Servo Subsystem Design

not specifying the servo part number could cause a problem in the end with positioning accuracy requirements. Details such as multiple views, cut-away views, dimensions, and material specification are expected in a mechanical detailed design.

Figure 15.6 shows a similar level of detail for an electrical subsystem for the stereo-vision system.

Part numbers are shown along with all resistor values, voltage levels, and a schematic that is not yet laid out for a PCB (Printed Circuit Board), but which could be entered into a schematic capture tool. All signal interfaces should be clearly labeled and described such as the NTSC (National Television Standards Council) signal from the camera to the video encoder chip (Bt878). Parts should be specified well enough that they can be entered into a BOM (Bill of Material) data base for order and assembly during implementation and test phases. Any simulation models of circuits completed, for example using SPICE (Simulation Program with Integrated Circuits Emphasis), should be included with this design as well.

Electrical design also should include not only circuit design but also logic design, which may often be captured in a mixed-signal circuit design, but logic elements often require additional explanation. Logic design should include all combinational and clocked state machine components and subsystems with schematic models or HDL (Hardware Design Language) models. Figure 15.7 shows logic diagram for a register component (JK flip-flop) and for an ALU (Arithmetic Logic Unit) for integer division.

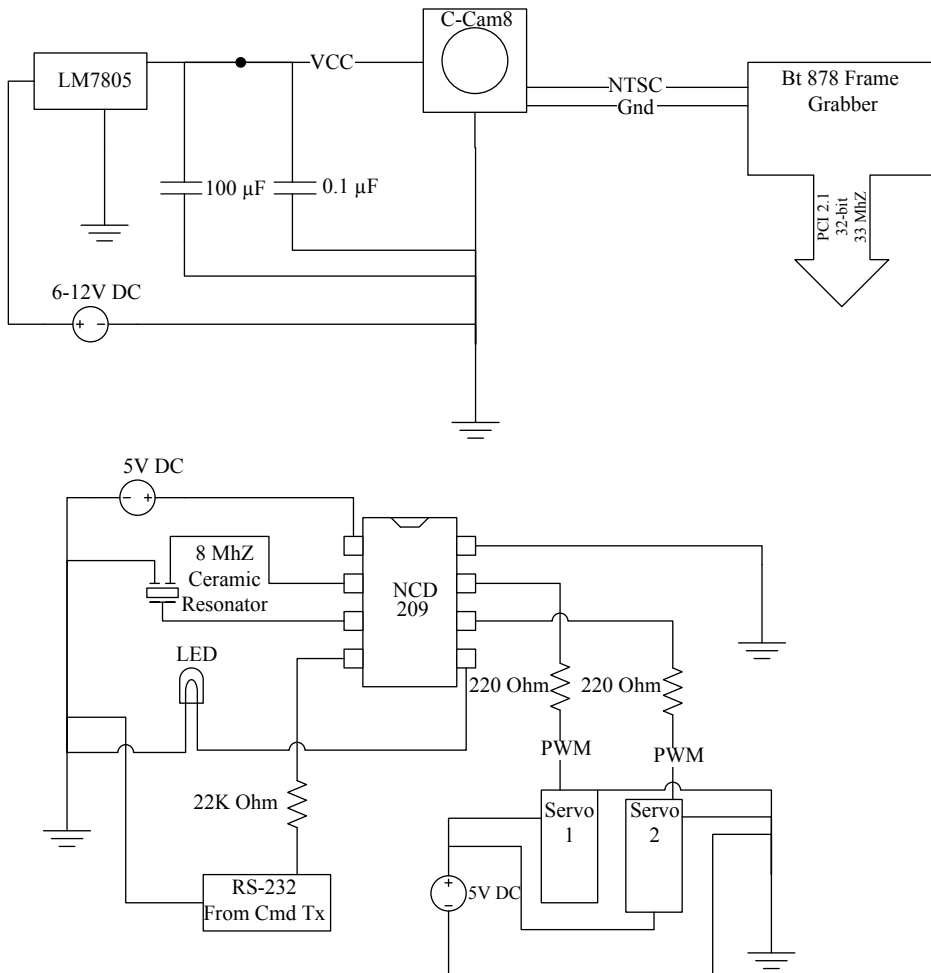


FIGURE 15.6 Detailed Electrical Design for Tilt/Pan Camera Subsystem

Most often, an HDL such as Verilog, VHDL, or SystemC, is used to implement logic designs and schematics to verify and simplify logic and to ensure that dynamic behavior is as required. This HDL design can be simulated for verification and built from discrete logic components, downloaded as a synthesized bit stream into an FPGA (Field Programmable Gate Array), or fabricated as an ASIC (Application Specific IC) for implementation. Detailed electrical logic design is aided by an EDA (Electronic Design Automation) tool chain that provides schematic capture, logic design using an HDL, simulation-based verification, synthesis, layout, and physical

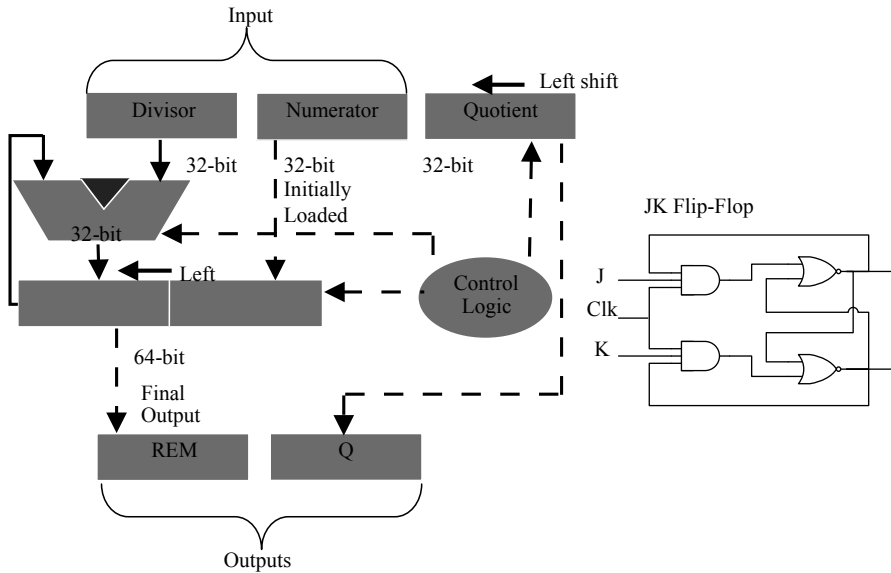


FIGURE 15.7 Detailed Logic Design Block Examples

realization. For example, the following is a SystemC HDL specification for the integer divide logic design depicted in Figure 15.7:

```
// Division algorithm based upon "Computer Organization and Design:
// The Hardware/Software Interface, but D. Patterson and J. Hennessy
//
// SystemC implementation

#define MASK32 0x00000000FFFFFFFF

SC_MODULE(divide)
{
    sc_in_clk          CLOCK;
    sc_in<bool>         RESET;
    sc_in<unsigned>     DIV, NUM;
    sc_out<unsigned>    Q, REM;
    sc_out<bool>        ERROR;
    sc_out<bool>        READY;

    void compute();
    SC_CTOR(divide)
    {
```

```

        SC_THREAD(compute, CLOCK.pos());
        watching(RESET.delayed() == true);
    }
};

void divide::compute()
{
    // reset section
    signed lhr, rhr;
    unsigned i, q, r, d, n;
    long long signed ir, save_ir;
    bool err;

    while(true)
    {

        // IO cycle for completion with or without ERROR
        Q.write(q);
        REM.write(r);
        ERROR.write(err);
        READY.write(true);
        wait();

        // IO cycle for divide request
        d = DIV.read();
        n = NUM.read();
        READY.write(false); // set busy state
        wait();

        // The divide computation
        if(d == 0)
        { q=0; r=0; err=true; }
        else if(n == 0)
        { q=0; r=0; err=false; }
        else
        {
            ir = n; q = 0; i = 0;

            while (i < 32)
            {
                ir = ir << 1;
                save_ir = ir;
            }
        }
    }
}

```



```

        lhr = ((ir >> 32) & MASK32);
        rhr = (ir & MASK32);
        lhr = lhr - d;
        ir = (lhr << 32) | ((long long unsigned)rhr
                           & MASK32);

        if(ir < 0)
        {
            ir = save_ir;
            q = q << 1;
        }
        else
        {
            q = (q << 1) + 1;
        }
        i++;
    }
}
}
}

```

This SystemC code can be executed on the SystemC simulator for verification.

One very important opportunity for early system-level design verification involves integrating software components with simulated or emulated hardware component and subsystem design. For example, boot code can be tested and run using an EDA co-simulation tool that provides an instruction set simulator that can run code on simulated hardware. The main issue with this is the immense processing power required to execute a cycle-accurate co-simulation model. Another option is to test software and hardware on an FPGA emulation platform that provides some level of the hardware/software interface defined in the system design, but may not implement the final physical form factor and may not implement all the required hardware interfaces. Likewise, some early testing between mechanical and electrical components may be possible. For example, the stereo-vision tracker servo subsystem could be tested with an FPGA-based servo controller interface to determine whether the hardware state machine for commanding the servo to set points using PWM is properly designed. Initial integrated testing of the servo subsystem might include a simplified single camera mono-vision test and simple command-line interface test software. The

stereo servo mechanical subsystem would, of course, require a two-camera tilt/pan mounting system as shown in Figure 15.8.

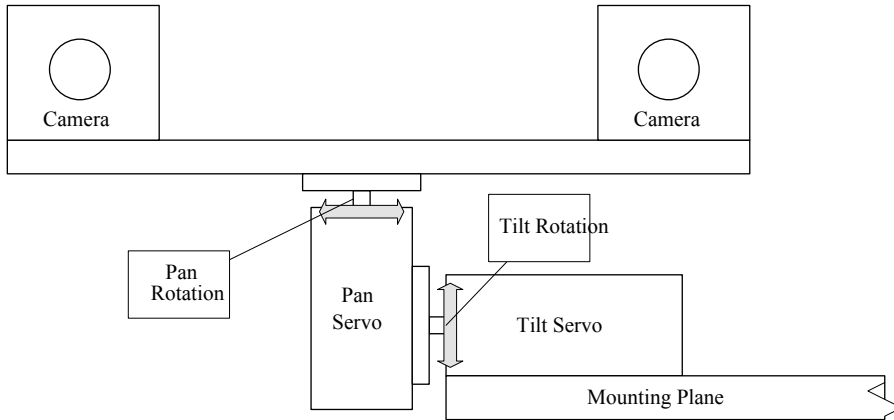


FIGURE 15.8 Two-Camera Stereo-Vision Mechanical Design

Part of the final stages of detailed design should involve planning and status checks to ensure that the integration plans are on track as they were defined during the analysis stage of the second turn in the spiral. If not, the plans should be adjusted so that the second integration and regression phase goes smoothly. Numerous detailed design reviews should be held, often with much more limited audiences than high-level design so that hardware experts are reviewing hardware designs and likewise for mechanical and software detailed designs. Design trade-offs, interface checks, and overall system efficacy should be determined during high-level design. High-level design reviews should include much more cross-discipline design review than detailed design. For real-time embedded systems, deciding which real-time services are implemented with hardware state machines versus software is a key decision. Analyzing exactly what software services are required and how they will interact is a significant aspect of software high-level and detailed design.

Real-time embedded system designs can benefit from design methods that specifically show software timing, service releases, and processing pipelines. Figure 15.9 shows a method for diagramming the services in a data processing pipeline where each pipeline stage must execute within a deadline.

Design methods that can be directly compared to measurement or debug tools can vastly simplify verification. Figure 15.9 can fairly easily be compared to a WindView (System Viewer) trace for a sequence of synchronously

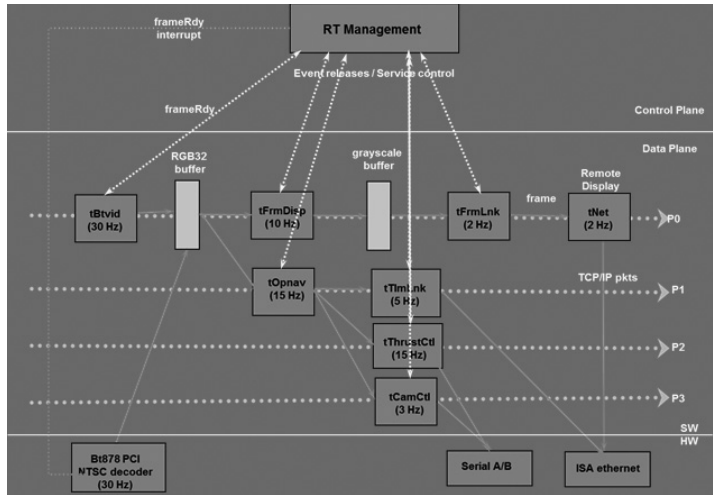


FIGURE 15.9 Stereo-Vision Processing Pipeline Sequence

released services in a pipeline. Software and the interfaces and deployment of software on hardware can be well described within the UML framework. Figure 15.10 shows the overall deployment of software modules on hardware for the stereo-vision system.

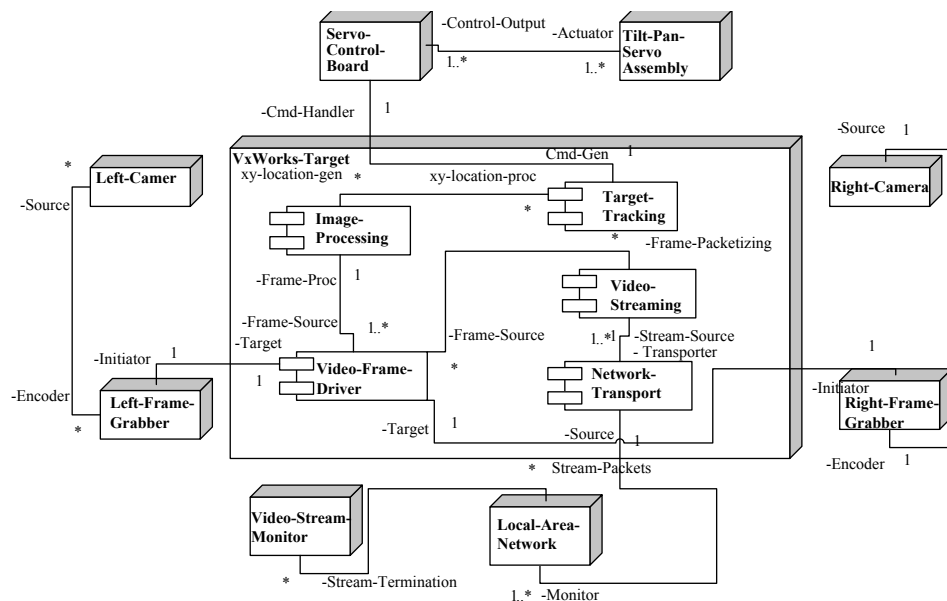


FIGURE 15.10 Stereo-Vision Deployment Diagram

The deployment diagram in Figure 15.10 clearly shows the major software modules that must be developed. A given software component, for example the video driver, can now be designed with a class model that specifies functions, data, and a decomposition of a module into a class hierarchy. Figure 15.11 shows a possible class hierarchy for the video driver.

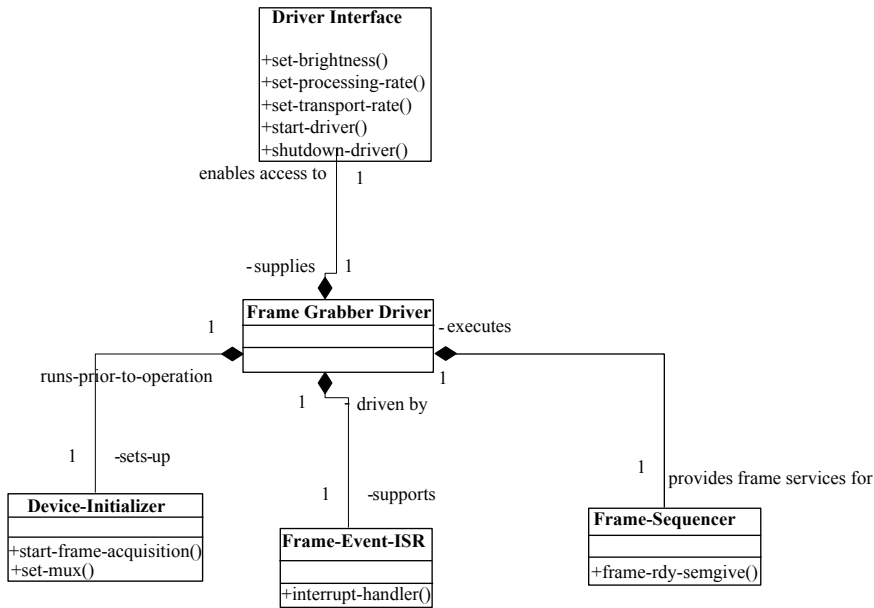


FIGURE 15.11 Stereo-Vision Video Driver Class

The classes (and objects that instantiate them) can then be used in sequence or collaboration diagrams to show which modules call which class functions in each component and in what order. Figure 15.12 shows a sequence diagram for components in the stereo-vision system.

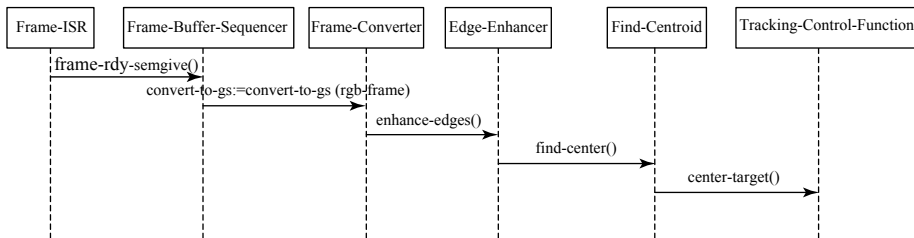


FIGURE 15.12 Stereo-Vision Sequence Diagram

Finally, to specify the design for the class functions (sometimes called methods), state machine design notation can be used to specify the function behavior now that it has been fully defined statically (class model) and in terms of the dynamic behavior at an interface level. Figure 15.13 shows a state machine design for the stereo-vision frame sequencing in the processing pipeline by the tBtvid driver.

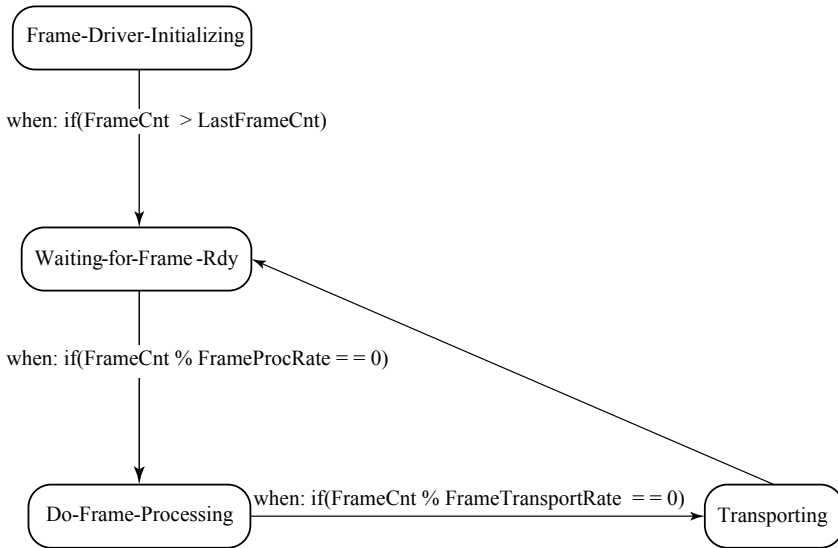


FIGURE 15.13 Stereo-Vision Sequencer State Machine

15.7 Component Unit Testing

Component testing can be divided into a number of test types, including the following:

- Functional and interface tests
- Stress testing
- Soak testing
- Performance testing

Testing exit criteria should be defined for each one of these types of tests ideally. Both at the unit level and for system-level regression testing.

Functional tests verify that subsystems and components meet their interface and behavior requirements for those interfaces as specified by analysis phase requirements definition. Often it is impossible to fully test a functional interface due to the massive number of input/output combinations that might be required even for a simple software or hardware component. For example, a 64-switch interface has enough switch permutations that there is no way it can be fully tested. In this regard, functional tests should focus on input patterns and sequences that are expected and desired as well as those specifically not-desired and for which the system is expected to reject and handle and prevent.

Because not all input combinations, timing-patterns, and communication sequences can be tested between components and subsystems, *stress testing* provides important verification for unexpected inputs and timing. Stress testing is most often done with random input generation and with test harnesses that drive components harder than they are ever expected to be driven. For stress testing, the electrical subsystems are tested at higher voltage, higher temperatures, and higher speeds than the requirements. For example, a PCB layout designed for 100-MHz signal buses might be designed for gigahertz signaling and tested with signal rates at twice the frequency of normal operation. Likewise, a software module might be tested with random inputs at higher-than-expected calling frequencies, and driven by service request workloads more demanding than those expected in actual operation.

Soak testing provides confidence in subsystem and component stability and life-time estimates. For example, a software subsystem might be run for more than 48 hours with continuous workload to ensure that the software does not lose track of resources over time (a memory leak for a dynamic memory management service). Furthermore, soak time tests will uncover issues that are rare, such as unlikely combinations of events that might lead to timing problems, for example, a cache miss that causes very occasional software service deadline overruns. Similarly, for hardware, soak tests involve continuous runtime to verify component durability and life-time expectations.

Finally, *performance tests* with units provide critical information for bottleneck analysis. If the performance of each unit can be tested in isolation, then the upper limits on the throughput, fidelity, or reliability of that unit can be determined apart from the system as a whole. Performance unit testing is fundamental to overall system risk analysis. The slowest and least

reliable components can be identified from these unit tests and factored into a system-level performance and reliability analysis.

Software includes several types of unit tests that are specific to the nature of software modules:

- Black box test
- Glass box test

A black box test is a functional test or stress, stability, performance test that is run without any knowledge of exactly what code is being exercised inside a software module. The danger of black box testing is that while rigorous functional, stress, stability, and performance tests might be defined, they may still exercise only a small percentage of the software in a module. A measure of the coverage of the software module for a given suite of tests indicates whether sufficient testing is being completed—this is a glass box test. A glass box test involves definition of test criteria that includes the following:

- Number of execution paths driven out of total number in a module
- Number of statements or instructions covered out of total
- Number of decisions fully evaluated in a module

The foregoing conditions provide metrics for how complete the functional, stress, stability, or performance tests are in terms of fully exercising the software unit under test. More importantly, quantifying coverage allows the tester to know when they are done. Path coverage is simply all of the instruction sequences that may be executed from every single branch point in code. Statement coverage is simply how many statements have been executed out of the sum of all unique instruction addresses. Less obvious is the decision coverage.

Criteria called MCDC (Multiple Condition, Decision Coverage) evolved due to compiler and instruction set optimizations that allow for expressions or individual instructions to be executed conditionally. For example the following C code includes two expressions with a logical or test and two paths, one for a true outcome and another for the false outcome.

```
void control_update(void)
{
    if(inactive || (within_centroid_tolerance() &&
                  within_servo_deadbands()))
```

```

{
    monitor_and_wait();
}
else
{
    update_servo_control();
}
}

```

In this code fragment, there are clearly only two paths. Both paths can be driven by evaluating *inactive* and the function *within_centroid_tolerance()*. For example, the *monitor_and_wait()* path is driven by an *inactive*=TRUE. The *update_servo_control()* path can be driven by either *inactive*=FALSE and either one of the *within_centroid_tolerance()* or *within_servo_deadbands()* functions returning FALSE. Likewise the *monitor_and_wait()* path could also be driven by *inactive*=FALSE and both *within_centroid_tolerance()* or *within_servo_deadbands()* returning TRUE. So, there is a case where both paths are driven, but *within_servo_deadbands()* is never called and evaluated. So, while path coverage for the function *control_update()* would indicate full coverage, code in *within_servo_deadbands()* could execute and cause a failure in the field. If full path coverage criteria is measured on all functions, this might be caught, but if *within_servo_deadbands()* is also called from another context, the function may appear covered, but was in fact never tested in the context of *control_update()*. The MCDC coverage criteria require that all logical sub-expressions be evaluated for full coverage.

The GNU (GNU's Not Unix!) open source compilers known as gcc, the GNU C compiler, provides profiling and path coverage analysis that can help with Glass box analysis. With gcc path coverage, the paths tested (covered) by unit test cases can be monitored in terms of execution statistics (number of times each path was executed). This is easily done by simply compiling a software module with path coverage instrumentation directives as the following example shows. First the code is built with the “make” command and specific path coverage instrumentation directives:

```

%make
cc -Wall -O0 -fprofile-arcs -ftest-coverage -g    sclogic.c
-o sclogic

```


Second, the code is run as normal and might include some Black box tracing output using system logging or simple print statements as shown:

```
%./sclogic
function_A
function_B
do function_C
do function_D
function_A
do function_D
...
do function_C
function_A
do function_D
function_A
function_B
do function_D
```

Third, the source code is analyzed using the gcov tool so that the original source can be annotated with execution statistics as follows:

```
%gcov sclogic.c
File 'sclogic.c'
Lines executed:100.00% of 29
sclogic.c:creating 'sclogic.c.gcov'
%gcov sclogic
File 'sclogic.c'
Lines executed:100.00% of 29
sclogic.c:creating 'sclogic.c.gcov'
```

The result is a new gcov file with execution statistics annotated for each line of original source code:

```
%cat sclogic.c.gcov
-:      0:Source:sclogic.c
-:      0:Graph:sclogic.gcno
-:      0:Data:sclogic.gcda
-:      0:Runs:1
-:      0:Programs:1
-:      1:#include <stdio.h>
...
1:      41:int main(void)
-:      42:{
...

```

```

-: 52:    // Test Case #2, Test use in logic
11: 53:    for(testIdx=0; testIdx < 10; testIdx++)
-: 54:    {
10: 55:        if((rc=(function_A() && function_B()))
2: 56:            function_C());
-: 57:        else
8: 58:            function_D();
-: 59:
-: 60:    }
-: 61:
1: 62:    return(1);
-: 63:}

```

The annotated file shows counts for each line of code that can be executed. For example, in the gcov output above for the `sclogic.c` C source code, we see that line 53 (for loop) was executed eleven times, but the loop body composed of an if statement was executed only ten times. This is well known to C programmers because for loops must execute one more time than the for loop body to test the exit condition. The `sclogic.c` code includes a C code short-circuit logic if expression, which makes the coverage analysis interesting. To make the coverage easier to read and understand, Linux includes a tool known as “lcof” that can produce an HTML (Hyper Text Markup Language) version of the instrumented code as follows:

LCOV - code coverage report

Current view:	top level - MCDC2 - sclogic.c (source / functions)	Hit	Total	Coverage	
Test:	sclogic.info	Lines:	29	29	100.0 %
Date: 2014-10-13		Functions:	5	5	100.0 %
23:27:32					

	Line data	Source code
1	:	#include <stdio.h>
2	:	
3	:	
4	11 :	int function_A(void)
5	:	{
6	:	static int toggle_A=0;
7	:	
8	11 :	printf("function_A\n");
9	11 :	if(toggle_A == 0)

```

10         6 :         toggle_A=1;
11         :         else
12         5 :         toggle_A=0;
13         :
14         11 :         return toggle_A;
15         :     }
16         :
17         6 : int function_B(void)
18         : {
19         :     static int toggle_B=0;
20         :
21         6 :     printf("function_B\n");
22         6 :     if(toggle_B == 0)
23         3 :         toggle_B=1;
24         :     else
25         3 :         toggle_B=0;
26         :
27         6 :     return toggle_B;
28         : }
29         :
30         3 : void function_C(void)
31         : {
32         3 :     printf("do function_C\n");
33         3 : }
34         :
35         9 : void function_D(void)
36         : {
37         9 :     printf("do function_D\n");
38         9 : }
39         :
40         :
41         1 : int main(void)
42         : {
43         :     int rc;
44         1 :     int testIdx=0;
45         :
46         :     // Test Case #1 - Call all functions
47         1 :     rc=function_A();
48         1 :     rc=function_B();
49         1 :     function_C();
50         1 :     function_D();
51         :

```

```

52          :      // Test Case #2, Test use in logic
53      11 :      for(testIdx=0; testIdx < 10; testIdx++)
54          :      {
55      10 :          if((rc=(function_A() && function_B()))))
56          2 :              function_C();
57          :          else
58          8 :              function_D();
59          :
60          :      }
61          :
62      1 :      return(1);
63      :  }

```

The goal of Glass box testing and path coverage analysis is to verify that all paths have been well tested by unit test cases. Unit test cases are often designed as Black box tests, unit test drivers written to simply produce expected results, sometimes for negative testing (to drive unexpected inputs into a function), but the path coverage provides source-level analysis to verify that the tests are complete. The coverage analysis ensures that the tests meet quantifiable criteria such as executing all lines of code at least once (or more). The statistics in the coverage analysis also help make obvious the behavior of C short-circuit logic.

Another Glass box verification that is useful is profiling code. This means producing statistics to measure how much of the overall execution time is spent in each code module, each function and even each line of C code. This allows for optimization of key functions and lines of code to improve efficiency. As shown in Chapter 13, “Performance Tuning”, specialized tools can be used that make use of hardware support such as the MONster example showed. However, simple software instrumentation is also possible using the GNU gprof—GNU profiler.

The software RAID (Redundant Array of Inexpensive Disk) example included on the DVD which uses XOR (exclusive OR logic) to encode data so that loss of a data segment can be recovered, can be run for thousands of iterations and profiled as follows:



```

%make
cc -O3 -Wall -pg -msse3 -malign-double -g -c raidtest.c
cc -O3 -Wall -pg -msse3 -malign-double -g -c raidlib.c
cc -O3 -Wall -pg -msse3 -malign-double -g -o raidtest

```

```

raidtest.o raidlib.o
%./raidtest
Will default to 1000 iterations
Architecture validation:
sizeof(unsigned long long)=8
RAID Operations Performance Test
Test Done in 453 microsecs for 1000 iterations
2207505.518764 RAID ops computed per second

```

The statistics for the profile test shows that overall more than two million RAID encoding operations were completed per second, but this is just a Black box measure of performance. We still have no idea how much of that time was spent on XOR computation compared to the overhead of managing the data, looping, and just running the test. In the “make” we note that the “-pg” option was used to add GNU gprof profiling, which means that a gmon.out file will be generated by the instrumentation. We can then create a profile report as follows:

```

%ls
Makefile      gmon.out      raidlib.h      raidlib64.c      raidtest
raidtest.o
Makefile64    raidlib.c      raidlib.o      raidlib64.h      raidtest.c      raid-
test64
%gprof raidtest gmon.out > raidtest_analysis.txt

```

Looking at the generated report summary, we see that in fact only 15.47% of the time is spent computing XOR for each LBA (Logical Block Address, a collection of 512 bytes to be written to a disk drive).

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ns/call	ns/call	name
82.13	1.54	1.54				main
15.47	1.83	0.29	2000001	145.38	145.38	xorLBA
2.67	1.88	0.05	2000001	25.07	25.07	rebuildLBA

The results provided by gprof are similar to the MONster results we saw in Chapter 13, but this is done using only software instrumentation without any hardware support and with a simple compile directive. While SWIC (SoftWare In Circuit) instrumentation for path coverage and profiling can be more intrusive than HWIC (Hardware In Circuit), it is convenient and often accurate enough to provide good Glass box testing and verification of software unit correctness and efficiency.

15.8 System Integration and Test

As unit tests are completed, units can be assembled, including both hardware and software for integrated testing. Most often, it's best to take a building approach where components are integrated to form a single subsystem, which is tested. If the subsystem tests well, then two such subsystems are integrated into a partial system, and again tested. This is repeated until the whole system is integrated and meets expectations. Invariably, some unanticipated subsystem interaction results in two subsystems testing good, but when combined, the partial system does not test as expected. At this point, rerunning each subsystem test, a regression test, provides validation that the two subsystems still are functional on their own. More complex still is a situation where two subsystems are integrated and meet test expectations, but when a third subsystem is added, the partial system does not meet expectations. The ability to quickly isolate the problem is accelerated by quick unit regression test capability. Often a subsystem may fail when integrated due to integration error, a systemic interaction causing failure, or just random bad luck. Either way, the ability to test partial systems and regression test components and subsystem units is fundamental to the integration-by-steps approach.

15.9 Configuration Management and Version Control

Ideally hardware and software will be developed on projects concurrently with integration tests using ISS (Instruction Set Simulation), TLM (Transaction-Level Modes) for hardware, and with early testing of software on hardware emulation platforms. During this concurrent development here are some tips on how to keep features and modules on track:

- Identify hardware and firmware module owners to take responsibility through entire life cycle.
- Require tests to be developed in parallel with module development.
- Require early adoption of nightly testing using TLM simulation and/or RTL simulation.
- Adopt configuration management version control (CMVC) tools that allow for feature addition branches and version tagging.

Although these recommendations are followed in most projects, they often aren't implemented until the end of the process shown in Figure 15.14.

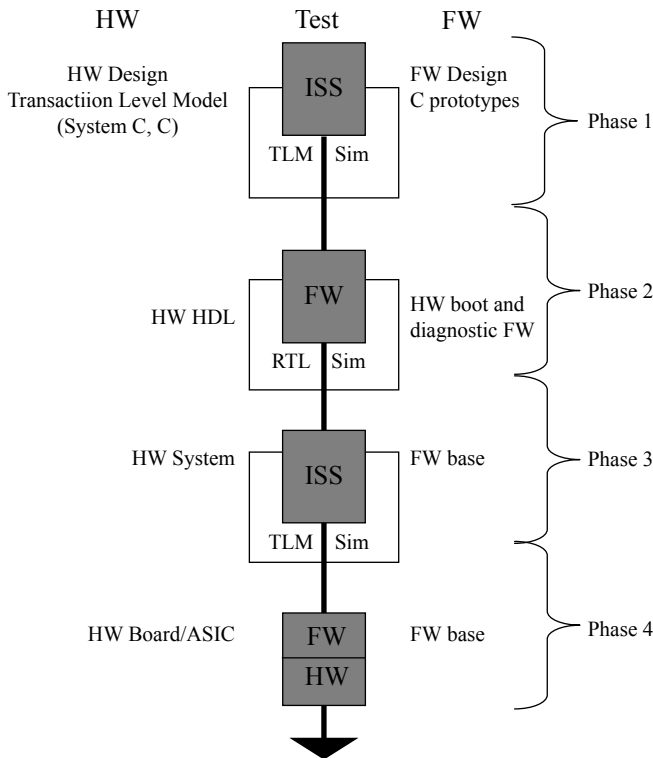


FIGURE 15.14 Concurrent HW and SW Development Timeline

Starting early and automating tests for nightly regression is now possible with EDA and co-simulation tools available for hardware and software development. In the days before early verification tools were available, hardware and firmware development proceeded much more independently than they can now. A typical process included independent development of firmware on an emulator while hardware was designed and developed, with most of the concurrent testing done during the final post-silicon verification. Despite advances in verification tools, many developers still work along lines established in those days, and thus don't adopt testing and regression processes, or configuration and version control, to the extent that they should.

Because EDA and HDLs for hardware design make the hardware development process similar in nature to firmware development, both hardware and firmware can and should use configuration management tools—the same ones, if at all possible! This almost seems blasphemous to organizations that have grown accustomed to a silo model for hardware and

software development, where a quick hand-off is made post-silicon, and interaction is otherwise minimal.

One difficulty when testing changing firmware on changing hardware is that stability often suffers: this can greatly impede the progress of both hardware and firmware development teams. This problem can be solved by having the hardware team make releases of simulators to the firmware team. Likewise, the firmware team should make releases of boot code and diagnostic code to the hardware team. Both teams need well-disciplined processes for maintaining versions and releases. One way to do this is to maintain a main line of C code or HDL that is guaranteed to be stable. As hardware or firmware developers add code, they can do this on branches from the stable main line, and merge new features and bug fixes made on code branches back to the line. Figure 15.15 depicts this basic disciplined practice.

Figure 15.15 shows how a developer can take a stable baseline of C code or HDL, branch it for modification, add new features and test them on the branch, and then merge them with other potential changes on the main line. After the merge is completed, the new result must once again be tested, and then it can be put back on the main line, advancing the overall system and maintaining stability. The only place unstable code should be found with this process is on a branch. After a CVS repository has been set up and code checked into it, developers can create a working *sandbox* for

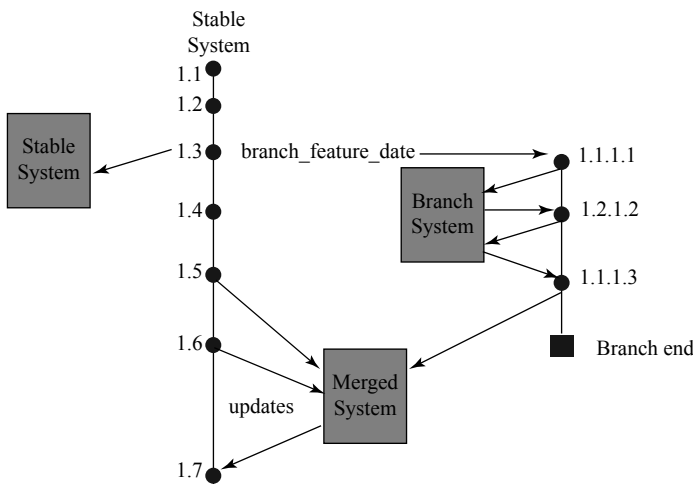


FIGURE 15.15 Example of CVS Module Branching

their code; the sandbox is their own private copy from the repository. For simple modifications to the code base, a file can be modified and, after testing, put back into the base with a simple command:

```
cvs commit filename
```

Branching is a more advanced method, which is useful when sets of files need to be modified, tested, and shared with other developers. A CVS branch has a tag, and therefore other developers can check out copies of a branch to their own branch sandbox, as Figure 15.15 shows. In Listing 15.1, the first set of commands does a checkout of the current code and then tags this revision of the code with a branch tag. *Tags* are simply sets of revisions of all files from the repository. The branch tag is a special tag that not only defines a set of file revisions but also allows for modification to those files with a new revision that remains separate from the main repository revisions. This is shown in Figure 15.15 as the branch line that includes a main line revision and branch revision. The developer or developers working on the branch can share updates and test the branch revisions without affecting the main line of code.

The middle set of commands in Listing 15.1 provides an example of updates to the branch revision. When the developers are happy with the branch, the branched code set can be merged back to the main line of code with the final set of commands in Listing 15.1.

LISTING 15.1 CVS Commands for Branching

```
cd stable_directory
cvs checkout system
make system; test system
cvs tag -b -c branch_feature_date

cd branch_directory
cvs checkout -r branch_feature_date
modify files
make system; test system
cvs commit
modify files
make system; test system
cvs commit

cd merge_directory
cvs checkout system
```

```
cvs update -j branch_feature_date -kk system
make system; test system
cvs commit
```

Branches can be useful for almost any modification to a design maintained as a file set, but most often they are used for the following:

- Complicated multi-file bug fixes
- Addition of new features
- Performance optimization and tuning
- Special releases to external customers or internal users

Optimization is a great example of an area where branches combined with regression testing can allow for significant and aggressive performance improvements while minimizing risk to system stability. You may very well have optimized a system to improve performance, only to find after subsequent development of more sophisticated regression tests that the optimization has destabilized the system. Or, in some cases, it may take some soak time before the destabilization is noticed. For example, if the optimization introduces a new race condition, then that condition might not be hit for many days or weeks, long after the optimization has been initially tested and integrated back into the system. At this point, the optimization might be harder to back out.

Optimizations performed on branches can be tested and maintained on the branch and merged with a very clear change set. You can more readily back out of merging a destabilizing optimization back into the main line if you use tags on the branch.

The CVS (Code Versioning System) is used less than it was when the first edition of this text was published and has been mostly replaced by use of CMVC (Configuration Management and Version Control) tools that not only provide source code version control but also manage the configuration of files and directory structure for larger software systems. Many of the principles of the main line of code, branches, and merges remain the same, but commands and concepts are somewhat different. Two of the most popular CMVC tools are Subversion and the GNU Git tools. Many of these tools can now be used on Cloud web sites where code can be hosted and managed either publically or with private repositories (e.g., <https://github.com/>, <https://bitbucket.org>). The DVD includes example instructions on how to use these newer CMVC tools.



15.10 Regression Testing

Regression testing is critical not only for integration steps but also for ensuring that a system configuration that worked yesterday still works today after incremental changes. If the system is broken due to incremental changes, this should be detected before such changes are made permanent to the working system. For example, a code change is suggested to a module to improve performance, but this change might break the functional specification. This should be caught by a nightly run system regression test.

Summary

A good understanding of system life cycle and planning for spiral turns and phases can help minimize surprises, delays, and costly errors during development of a project. The spiral life-cycle approach can also assist developers with risk maintenance and mitigation. More frequent reporting of progress to management also allows for better overall planning within an organization.

Exercises

1. Set up CVS or the Subversion configuration management and version control tool for your work on a real-time project to be completed using this book.
2. Complete the example stereo vision UML Visio template design provided on the DVD.
3. Write a SystemC model for a PWM generator state machine to control a hobby servo. Verify this design using the SystemC simulator and finally implement it on an FPGA ASIC such as the Virtex-II or Spartan.
4. Describe whether CVS is better than subversion or vice versa and why.
5. Describe whether Subversion is better than GNU Git or vice versa and why.
6. Place some of the example code under CMVC management using GitHub or Bitbucket.



Chapter References

[Bitbucket] <https://bitbucket.org>

[Gcov] <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>

[GitHub] <https://github.com/>

[Gprof] <https://sourceware.org/binutils/docs/gprof/>

CONTINUOUS MEDIA APPLICATIONS

In this chapter

- Introduction
- Video
- Uncompressed Video Frame Formats
- Video Codecs
- Video Streaming
- Video Stream Analysis and Debug
- Audio Codecs and Streaming
- Audio Stream Analysis and Debug
- Voice-Over Internet Protocol (VoIP)

16.1 Introduction

Digital video processing is an excellent way to explore and learn more about real-time embedded systems. The camera or video stream frame rate provides a fundamental service frequency, typically at 30 Hz for NTSC (National Television Systems Committee), from which other services may run at the same frequency or a sub-period of this basic rate. Video processing services for a wide range of system projects include the following:

- Video stream compression and decompression (codec)

- Video stream transport over a network
- Image processing to detect edges of a target object and to determine its center in XY plane
- Tracking a target object with tilt and pan servo control loop to keep object center in FOV center
- Two-camera stereo ranging to determine distance to a target object being tracked
- Digital video recording and playback
- Motion detection with video stream storage, motion stream playback
- Multiple video stream server (Video on Demand)
- Line-following mobile robot with forward-looking obstacle detection
- Embedded camera in robotic arm for object and target recognition pick and place
- Fixed overhead camera-based navigation of a robotic arm for pick and place

The basic requirement is a video-capture card and driver for it. The Video for Linux project maintains a number of drivers for Linux that can be used in an embedded Linux project, and, likewise, Linux drivers can be ported to VxWorks.



The DVD includes a VxWorks driver for the Bt878 NTSC capture chip, which provides rudimentary 320x240 RGB frame capture at 30 fps. This example driver was loosely based upon the original bttv Linux driver, but does not implement all of the capture modes that the bttv driver does. This example driver was originally built from the chipset manuals, but when problems with documentation were encountered, the Linux bttv driver source helped immensely. One of the best ways to really learn about the video hardware/software interface is to port a driver to VxWorks from Linux or start with the chipset manuals and build one from the ground up.

From the basic capture driver codec and/or image processing services can be run at the same 30Hz rate or a sub-rate, such as 15, 10, 6, 5, 3, 2, or 1 Hz, easily derived from the basic 30Hz interrupt rate from the frame encoder.

16.2 Video

The predominant video analog signal formats are NTSC (National Television Systems Committee) used in North America, Japan, and Korea; PAL (Phase Alteration by Line), used in South America, Africa, and Asia; and SECAM (Sequential Color with Memory), used in France and northern Asia. The NTSC standard used in North America was first defined in 1941 for black-and-white transmission and further refined in 1953 to include a color standard. The NTSC standard is used for television broadcast (HDTV, or High-Definition Television, is the emerging replacement with much higher resolution), for CCTV (Closed-Circuit Television), and for numerous digital video-capture and video-editing devices. An NTSC camera input to a video encoder, such as the Bt878 hosted on an x86 PC, provides real-time digital video capture and can be used as a basic platform for real-time video projects.

The basic NTSC signal will raster a television CRT with odd and even lines (interlacing) so that there is a retrace between the odd and even raster traces from the lower-right corner back to the upper-left corner of the screen. The interlacing was designed into NTSC to control flicker between frames for early television systems. The odd and even lines are updated at 59.94 Hz, with 486 lines out of a total of 525 used to display the image, and the remaining lines used for signal sync, vertical retrace, and the vertical blank lines that used to carry closed-caption data. This basic signal was modified to carry color with two chrominance signals at 3.57955 MHz that are 90 degrees out of phase. A luminance signal took the place of the black-and-white signal in color NTSC. Video-capture chips, such as the Bt878, encode this color NTSC signal into luminance and chrominance digital data by sampling the signal and using ADCs (Analog-to-Digital Converters) to generate a digital measurement of the luminance and chrominance signals. The video-capture chip must maintain PLL (Phase Locked Loop) with the NTSC signal to properly sample the NTSC signal at the right points in time. The NTSC signal lock is obtained during sync and retrace periods where the signal has a flat output known as the *front and back porch* [Luther99].

The digitized NTSC data is initially in the form of luminance and chrominance samples that can be converted to RGB data. For digital processing, RGB data is the most easily processed and the most common standard for display using a computer graphics adapter. Figure 16.1 shows the conceptual color cube representation of RGB, where the basic Red, Green,

and Blue outputs can be additively combined to derive all other common colors as a 24-bit pixel composite of the 8-bit R, G, and B values.

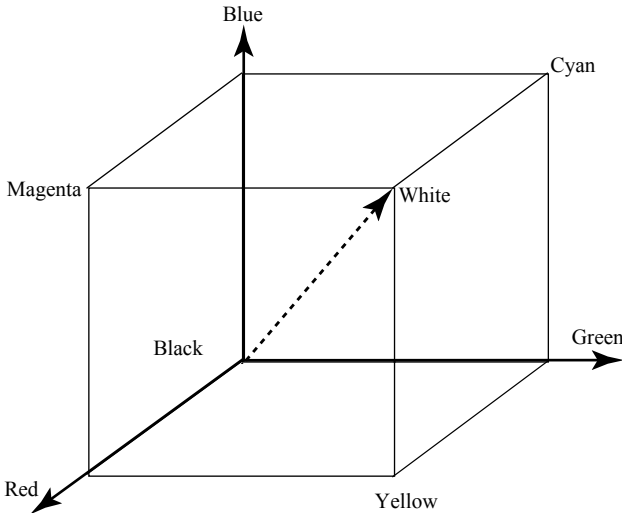


FIGURE 16.1 RGB Color Cube

A standard digital conversion among luminance, chrominance, and RGB defines a method for deriving RGB from NTSC sampling:

- **YUV (Where Y is luminance, and U, V are the chrominance) to RGB Conversion**

- $B = 1.164(Y - 16) + 2.018(U - 128)$
- $G = 1.164(Y - 16) - 0.813(V - 128) - 0.391(U - 128)$
- $R = 1.164(Y - 16) + 1.596(V - 128)$

To convert back from RGB to YUV, the following is used:

- **RGB to YUV Conversion (For Computers with RGB [0-255])**

- $Y = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16$
- $Cr = V = (0.439 * R) - (0.368 * G) - (0.071 * B) + 128$
- $Cb = U = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128$

In both cases all computed outputs should be clamped to a range of 0 to 255. The Bt878 video encoder performs this digital conversion with a hardware state machine so that RGB data can be captured from NTSC and pushed by DMA (Direct Memory Access) into the host PC memory.

Often grayscale or monochrome digital video is sufficient for basic computer vision applications or video monitoring applications, like motion detection security camera systems. Acquisition of NTSC into RGB digital data provides a very flexible approach since monochrome can be derived from it and since the RGB data can be displayed locally in full 24-bit color, but stored in a grayscale or in a compressed 16-bit luminance chrominance format. Grayscale monochrome frames can be derived from RGB by two methods:

- Selection of one color band from the three
- Conversion of the three to grayscale with a summing linear relationship

Figure 16.2 compares both methods showing grayscale derived by selection of R, G, or B bands alone and comparing this to the standard mixing conversion (based upon characteristics of human vision), which is defined as:

$$Y = 0.3R + 0.59G + 0.11B$$

In Figure 16.2, the balance does appear most pleasing to the eye; however, for computer vision, it's not clear that a balance provides better image processing for functions such as segmenting a scene with edge detection.

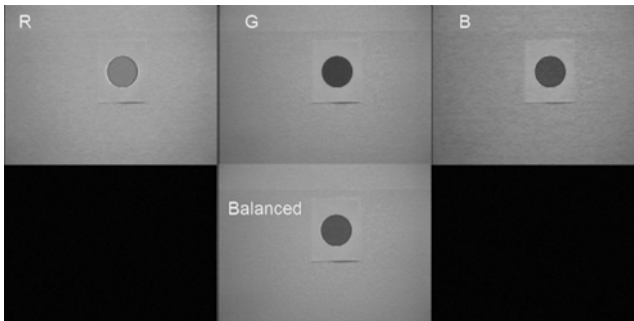


FIGURE 16.2 Grayscale Monochrome Derived from RGB by Three Methods

Color is one dimension of streaming video that also can be thought of as having a time and space dimension. Codec engines can take advantage of color, time, and space dimensions to compress a stream of video frames for storage or transport over a network to reduce required capacity and bandwidth. Conversion from color RGB to grayscale is a 3 to 1 compression, converting each pixel (picture element) into an 8-bit data value from 24 bits. Similarly, a 24-bit pixel RGB frame can be converted into a smaller luminance/chrominance (YCrCb) 16-bit pixel frame, providing a 3 to 2

compression with some color loss. The most commonly used YCrCb format is YCrCb 4:2:2, where four luminance samples are stored for every two Cr and Cb chrominance samples, as depicted in Figure 16.3.

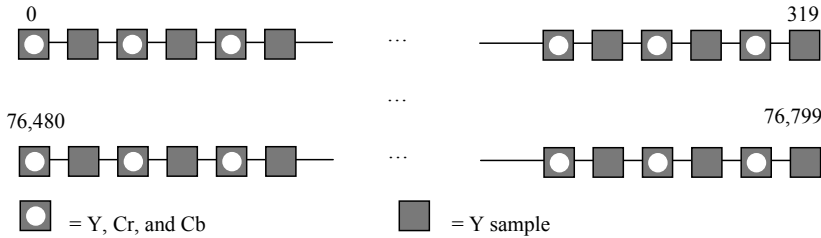


FIGURE 16.3 YCrCb 4:2:2 Format

Figure 16.3 shows YCrCb samples for a 320×240 frame size. The Bt878 encoder actually encodes RGB as α RGB, where an 8-bit alpha luminance is combined with the 24-bit RGB values for a 32-bit pixel. The alpha can be used as the Y sample, and the Cr and Cb can be derived from RGB using the YUV to RGB conversion. If alpha is not available, Y can be computed from RGB as well [Jack07]. In YCrCb 4:2:2, each Y, Cr, and Cb sample is 8 bits. The 2 RGB pixels require 48 bits, whereas 2 YCrCb pixels require 32 bits total (or 16 bits per pixel compared to 24 bits per pixel for RGB), yielding a one-third smaller frame size overall. The sample to pixel map format for YCrCb is:

- Pixel-0 = Y7:Y0₀, Cb7:Cb0₀; Pixel-1 = Y7:Y0₁, Cr7:Cr0₀
- Pixel-2 = Y7:Y0₂, Cb7:Cb0₁; Pixel-3 = Y7:Y0₃, Cr7:Cr0₁

So, for YCrCb, for every four Y samples, there are two Cr and two Cb samples packed into 4 total pixels. The frame size of 76,800 total pixels in a 320×240 frame is divisible by four, so YCrCb works for any $N \times M$ frame where N and M are divisible by two.

16.3 Uncompressed Video Frame Formats

Before diving into how to compress, transport, and decompress a video stream, it is useful to understand basic uncompressed video formats that can be used for debug and early testing. The two simplest single-frame formats are PPM (Portable PixMap) and PGM (Portable Gray Map). These two single-frame formats can be displayed by tools included with the DVD and commonly available viewers like the Irfan viewer. Both simply require



a header describing the data pixel and frame format along with the raw binary data in the format specified. For example, a PPM file might have the following header:

```
P6
#test
320 240
255
```

The header is followed by 76,800 24-bit RGB pixels arranged into 320 columns and 240 rows, with each color band ranging from 0 to 255 in value. The PPM and PGM headers are always in plain ASCII text, and the data is in binary form. The # character is used to add comment lines in the header. The P6 indicates this is a PPM file and not some other Netpbm format, such as PGM. The PGM header is almost identical—for example:

```
P5
# grayscale
320 240
255
```

This header indicates PGM with the “P5” and again specifies that 76,800 8-bit monochrome pixels will follow the header, with a value range from 0 to 255.

Several example PPM and PGM files are included on the DVD.



16.4 Video Codecs

The compression from RGB to YCrCb is a lossy compression in the color dimension. Codecs (compression/decompression) can provide compression of video streams in three dimensions:

- Color space
- X, Y frame space
- Time

Compression in X, Y frame space can be lossless or lossy with varying degrees of compression performance. Methods typically applied to strings of information such as RLE (Run Length Encoding) or Huffman encoding

can be used on frame data within a single frame. Pixels, like any string of symbols, can be compressed by encoding repeating patterns. The RLE encoding simply replaces all repeating symbol (pixel) sequences with a count and value. A simple approach like this can provide lossless compression in images. Huffman encoding can likewise provide string compression for repeating pixel sequences. Often the compression provided by string-oriented lossless compression is not significant because video data, unlike text, often has random variations, even in mostly flat backgrounds. For significant compression within a frame, lossy methods must be applied that use transforms or pixel averaging to combine neighboring pixels. The neighboring pixels can be regenerated during decompression with an inverse transform or interpolation. Information is lost, but compression can be 4 to 1 or higher compared to RLE or Huffman, which might provide a 10% smaller frame size reduction.

Compression over multiple frames can be significant. The most basic method is to transmit change-only data. Change-only data is computed by computing a difference frame for every stream frame pair. Pixel differences above a threshold (zero threshold for lossless) are transmitted as pixel address and pixel change, and all other pixels are considered unchanged. This leads to high compression rates for streams where the scene is not changing quickly. The use of a threshold helps eliminate changes due to small background noise in lighting, the detector itself, the atmosphere, and other perturbations to an otherwise static scene. In the extreme cases a totally static scene yields infinite compression and a totally changing scene actually inflates the frame size. For each change-only pixel the address plus the change must be encoded. For an 8-bit grayscale pixel this would require 8 bits for the change pixel and 17 bits for the pixel address for a 320x240 frame size. Most often, change-only compression includes evaluation of the number of changes, and if there are so many that the change-only frame provides no compression (or worse yet inflation), then the raw data frame is sent instead. This is an adaptive form of compression.

Optimal codecs apply a series of lossless and lossy compression schemes for maximum performance. For example, a stream could first be compressed in the color space by transforming RGB to YCrCb (a 33% decrease in frame size), followed by difference imaging with a threshold, and finally compressed further by a lossless method, such as RLE. Lossy methods followed by lossless can vastly improve the performance of the lossless compression; in cases where change-only compression won't help, the

color space and lossless compression will still decrease frame size. Adaptive change-only compression requires a compression header describing the compression applied so that the decompression can correctly treat each frame as a difference frame or not and so the decompression can be applied in the reverse order of compression.

Codec standards, such as M-JPEG (Motion-Joint Photographic Experts Group), MPEG (Moving Picture Experts Group), DivX, and Theora can be used, but building your own codec is the best way to learn about and appreciate how codecs work. The MPEG standard uses lossy image transforms, including frequency space and entropy encoding [Solari97]. The most current extensions, such as MPEG-4, also include prediction and difference-image encoding over time. In contrast, M-JPEG uses wavelet transform within frames rather than the DCT (Discrete Cosine Transform) used in MPEG and also does not employ compression over time (multiple frames), which has the advantage of being independent of motion in the video stream and allowing video editing to include cuts on any frame boundary. The DivX codec is a proprietary codec that uses MPEG-4, and Theora is an open source codec designed to be competitive with MPEG-4. Aside from building your own codec, integrating an open source codec, such as Theora, is the next best way to learn about video codec technology. The codec can be combined with a transport protocol for streaming, such as RTP (Real-Time Transport Protocol) built on top of UDP (User Datagram

Protocol) and most often used in conjunction with RTSP (Real-Time Streaming Protocol) and RTCP (Real-Time Control Protocol). Since publication of the first edition of this text, the examples for computer vision and digital media have been expanded on the included DVD to include OpenCV examples of the DCT encoding used in MPEG.



16.5 Video Streaming

RTSP such as RTP avoid the overhead of reliable connection-oriented transport methods, such as TCP (Transmission Control Protocol), and the complication of retransmission because video and audio streams can and do allow occasional data dropouts. For a real-time isochronal stream of video or audio, retransmission not only adds overhead but also can be detrimental to QoS. A frame or audio sound bite dropout is preferable to guaranteed delivery of data long after the deadline for continuous decompression and playback of a media bit-stream. In general, the smaller the buffering of

playback data, the better for streaming protocols because buffering and holding data add latency. For example, in audio, data buffer-induced latency makes a two-way conversation feel like you're talking on a satellite link. Likewise, buffering video streams makes stream control difficult and inaccurate. The ideal transport would provide constant bit rate with minimal buffering and latency. Streaming protocols such as RTP provide this type of performance on top of simple datagram transport with the added features of packet reordering, time-stamping, and delivery monitoring, but without any form of retransmission [Topic02].

16.6 Video Stream Analysis and Debug



The VxWorks Bt878 video encoder driver included on the DVD can be used to capture video streams for compression, transport, decompression, image processing, and computer vision. The Linux Video for Linux project version of this driver and the UVC (Universal Video Class) driver can likewise be used to build similar applications on Linux or Real-Time Linux.

With either frame capture driver, basic debug methods include the following:

- Dumping single frames for analysis
- Streaming frames to a viewer for observation
- Frame metadata derived from stream or image processing

Dumping a single frame with the VxWorks driver can be done by calling a function, *write_save_buffer()*, to dump the current frame over the Tornado TSFS (Target-Server File System). Care should be taken to ensure that the byte ordering is correct between the target and host if they are different architectures (e.g., an x86 target and a Sun Solaris host) and if the pixel size is larger than a single byte. Figure 16.4 shows an image dumped by TSFS and viewed with the freely available Irfan viewer for Windows.

Frame dumps provide basic analysis of lighting, focus, field of view, truth data for edge detection and object centroid calculation, and rudimentary debug to ensure that NTSC encoding is properly configured so that frames are being captured. The Irfan viewer provides point-and-click pixel address information. So a dumped image can be analyzed to determine the centroid of the target, the red circle in Figure 16.4, by reading the pixel address and comparing to the address calculated by target-based image pro-

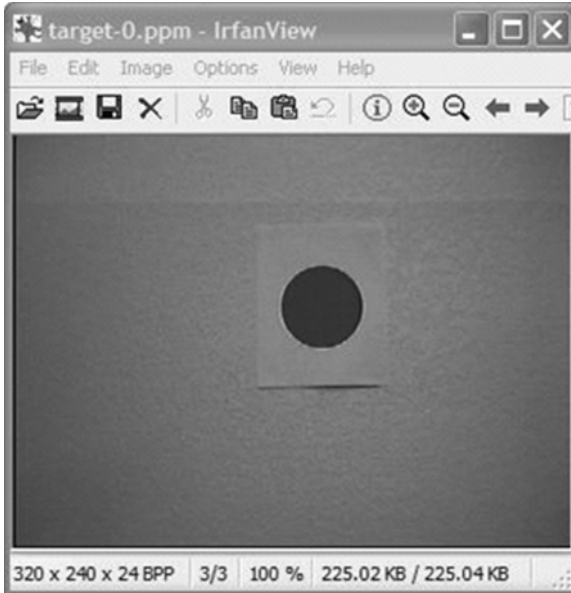


FIGURE 16.4 Bt878 Driver Frame Dump Example (see DVD for color image)

cessing. Lighting can be adjusted in the room or by setting ADC sensitivity using the function `set_brightness()`. The impact of calling `set_brightness()` can be observed with frame dumps. You can focus a camera using frame dumps, but it's difficult and slow, given the latency in feedback. Debugging can be greatly enhanced by providing an uncompressed debug stream and by using analog equipment, such as a television, to ensure that camera hardware is functional and focused well.

Debug streaming should be done with a simple uncompressed frame stream unless a reliable and lossless codec is available. The point of frame dumps and debug streaming is to view the raw data or to introduce image processing or compression after the raw data stream has been verified.

The DVD includes a basic PPM (Portable PixMap) stream viewer, which can be used to view uncompressed video streams at low frame rates. This viewer was built using the Python high-level programming language so that it is easy to modify and portable to almost any platform (any platform that runs Python). Figure 16.5 shows the *vpipes_display.py* Python application displaying the host-based stream that can be generated with *frametx_test.py*.



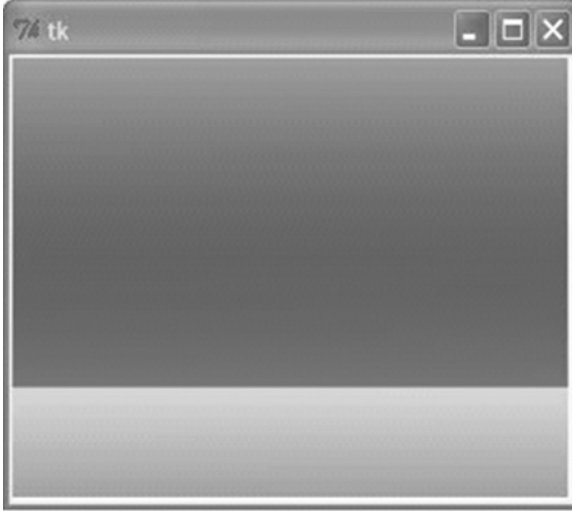


FIGURE 16.5 Vpipe Display Test Stream

The point of this test stream generator is to ensure that the vpipe display tool is working and the *frametx_test.py* provides an example of a TCP/IP frame source client that can connect to the vpipe display server for streaming debug. The streaming rainbow pattern provides a quick and easy way to make sure there are no bugs in the debug tool itself.

The following VxWorks target code can be spawned as a task (debug streaming service) and will connect to the vpipe display server and send PPM format frames once a second, based upon a semaphore given in the driver main loop:

```
void stream_client(void)
{
    /* opens client socket and connects to server */
    init_frametx();

    enable_streaming=1;

    while(!streamShutdown)
    {
        semTake(streamRdy, WAIT_FOREVER);
        frame_to_net(rgb_buffer);
    }
}
```

The `init_frametx()` function requires that the vpipe display server is already up and running and in the listen state for a TCP/IP socket. This utility is included with the video driver source code for VxWorks. The VxWorks video driver main loop is released by the encoder end of frame interrupt, and this loop in turn gives the semaphore `streamRdy` so that a frame is written out over TCP to the vpipe display server once every 30th frame, or once a second. This results in streaming to the vpipe display tool, as shown in Figure 16.6.

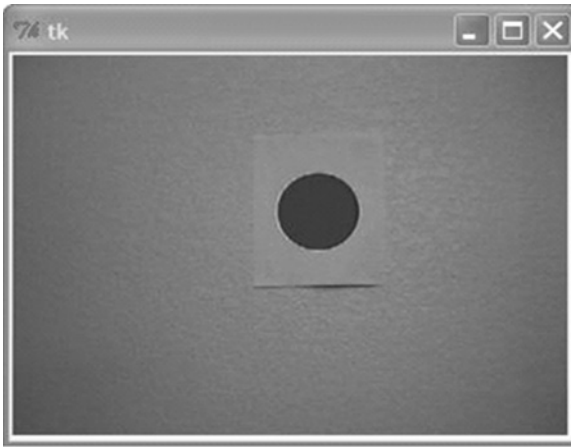


FIGURE 16.6 Vpipe Display Debug Video Stream

The debug streaming rate can be increased by simply increasing the frequency of the semaphore given in the driver main loop up to a maximum of 30 frames per second. However, care should be taken that the target has sufficient processing capability and that the Ethernet link has sufficient bandwidth to keep up with this rate.

Image processing on the video-capture target can compute frame metadata that can be very useful for debug—for example, parameters such as frame difference sums, target centroid pixel address, or current frame compression ratio achieved. This data derived during image processing, compression, and decompression can assist immensely with performance analysis as well as debugging. The data can be dumped on command with a VxWorks function call made in the windshell or can be sent for remote monitoring over a TCP/IP connection.

Since the publication of the first edition, many more examples for Linux that make use of the UVC driver have been added to the DVD, along with



OpenCV computer vision application examples. The Linux example code can be used with the Jetson board quite readily as it has been designed to run OpenCV and in fact provides GP-GPU hardware acceleration for many of the OpenCV functionalities.

16.7 Audio Codecs and Streaming

Most any sound card can be used to encode audio data into a digital format suitable for compression, transport, and decompression. The Linux ALSA (Advanced Linux Sound Architecture) provides drivers for numerous audio cards, including the Cirrus Crystal 4281. This driver was adapted for VxWorks and can be found on the DVD.



The Crystal 4281 includes the Cirrus 4297 codec, which can encode an analog audio source into 8-bit mono or 16-bit stereo digital data and can play back data in the same format. The 4281 also includes a controller that is used to coordinate data transfer from the codec FIFOs (First In First Out hardware buffer) to processor memory and vice versa. Recorded data is pushed into the host processor's memory by the 4281 DMA. Likewise, the 4281 DMA pulls data for playback from the host processor's memory. The 4281 encodes the incoming audio signal with ADC sampling at a selectable rate and DMA buffering that can be specified. The example driver code programs 11,025 samples per second and sets up DMA for recorded data to a 512-byte buffer so that a DMA completion interrupt is raised approximately 21 times every second ($11,025/512$ times a second) for 8-bit mono encoding. Likewise, the playback is interrupt-driven by the 4281 DMA so that at the same rate as record, an interrupt indicates that new data should be moved into the playback buffer. The driver uses double buffers so that one half of the record buffer is being written into by the DMA, while the other half is being copied out for audio transport, and one half of the playback buffer is being read by DMA into the DAC playback channel, while the other half is being written by software with data to be played back.

The Cirrus 4297/4281 does no compression, but does provide digital encoding suitable for software compression or encryption. The DMA interrupt-based synchronization with software also allows for easy coordination of audio stream transport from record to playback services. Some of the more tricky aspects of this driver are the 4297 gain and attenuation settings, properly tuning the timing to avoid playback and record data dropouts, and selecting the proper 4297 PCM channels for playback.

16.8 Audio Stream Analysis and Debug

One of the simplest ways to debug an audio driver is to examine the record and playback buffer data with memory dumps. The data in each should be a PCM waveform. Zero data in the record or playback buffer is indicative of a codec or DMA misconfiguration and will cause noisy pops and clicks in the playback audio. Most often the Crystal 4281 is used to implement two-way voice transport over Internet (VoIP). To make initial debugging simpler, it is useful to use headphones and a constant audio stream source like a CD or MP3 play to pipe in music. Initially the record function can be tested and record DMA buffer update verified with record buffer memory dumps. This can be done in VxWorks using the windshell “d” command, which will dump data from any address. The record data can be looped back through transport for local playback and verification. Again, the playback buffer should be dumped to verify that the data is being updated. The audio cards often include analog loop-back features, so it is important to verify that data in the playback buffer is truly being updated to avoid being fooled by analog loop-back and mistaking this as a working digital transport loop-back.

16.9 Voice-Over Internet Protocol (VoIP)

After a basic record and playback audio driver has been debugged and is working, this can be combined with session and transport services along with a codec or encryption engine for VoIP. Figure 16.7 shows one end of a VoIP digital terminal.

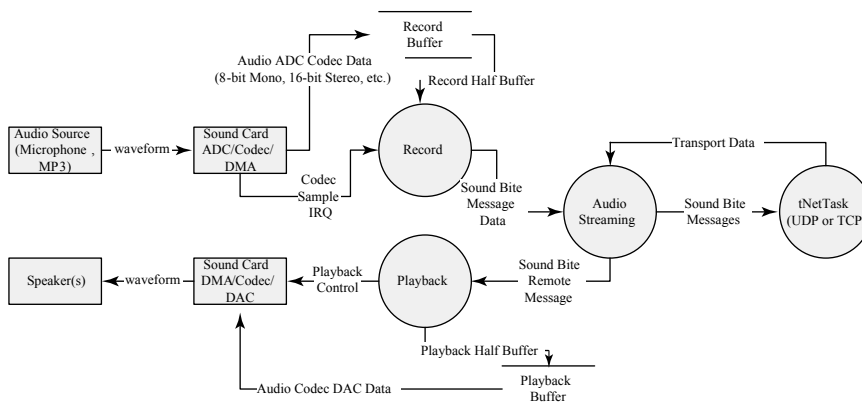


FIGURE 16.7 Voice-Over Internet Protocol Data Flow for One End

Each terminal must have record and playback services interfacing to streaming and transport services. The basic sound-bite record, transport, and playback rates are driven by the sampling rate, the audio format, and the buffer size for a sound bite. Ideally this process should emulate a constant bit rate encoding, transport, and decoding between the record and playback channels. Figure 16.8 shows how two of these digital terminals can be combined to provide full-duplex VoIP.

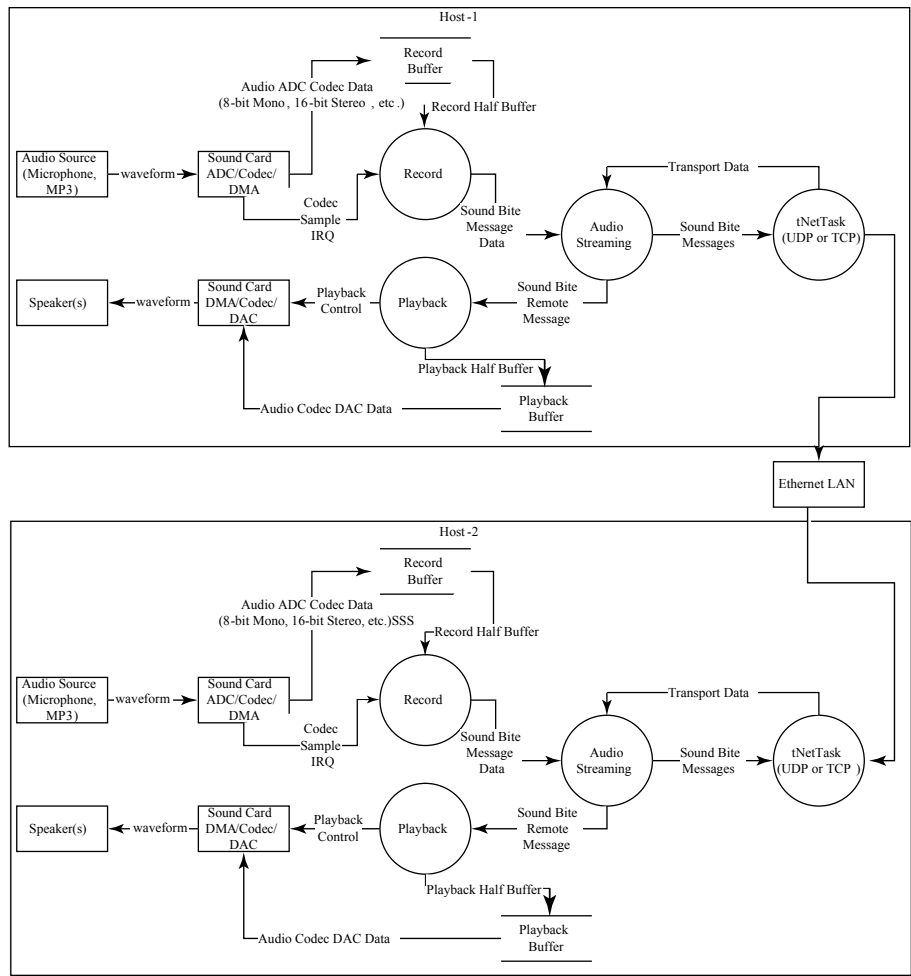


FIGURE 16.8 Full-Duplex Voice-Over Internet Protocol Data Flow

The basic full-duplex VoIP implementation still does not define how calls are initiated or any common voice services, such as voice mail, call waiting, conference calling, and hold. These features, however, can be added fairly easily once the basic full-duplex capability is implemented and debugged.

Summary

Continuous media applications include any media source and usage where periodic frames must be processed and transported in real time. This could be a broad range of media and multimedia applications, including video, audio, digital teleconferencing, virtual reality, video monitoring, video editing, video-on-demand streaming servers, and many more emergent soft real-time media applications. In the future, the complexity of emergent TV standards, such as HDTV (High-Definition TV) and DLP (Digital Light Projection), will require significantly more real-time embedded control [Poynton03]. Since publication of the first edition of this text, this application domain for real-time embedded systems has grown tremendously and 1080p (and even 4K) smart televisions that connect to the Internet have become commodity devices in homes as well as much expanded use of virtual reality and emergence of augmented reality, whereby graphics are overlaid on views provided to users by camera systems. Because a whole new book could be written on this application area alone, the authors have instead decided to simply provide more resources for the reader on the DVD in the form of code examples and reference materials.



Exercises

1. Use the Bt878 btvid.c driver included on the DVD or the Linux equivalent bttv driver to capture NTSC frames using a Hauppauge WinTV frame grabber. Prove that you got the driver working by streaming output to the Python vpipe_display.py viewing tool.
2. Use the image processing sharpen.c program on the DVD to provide edge enhancement to a PPM 320x240 image of your choice.
3. Write a program to detect the edges of the example target-0.ppm image and to produce an output that shows only red lines outlining the edges of the target object with an otherwise white background.



4. Download the ALSA (Advanced Linux Sound Architecture) driver for the Cirrus 4281 or a similar audio card, and write a Linux application to record analog audio input into files for later playback through the same driver.

Chapter References

- [Jack07] Keith Jack, *Video Demystified—A Handbook for the Digital Engineer*, 5th Edition, Newnes, an imprint of Elsevier, Burlington Mass., 2007.
- [Luther99] Arch Luther and Andrew Inglis, *Video Engineering*, 3rd ed., McGraw- Hill, New York, 1999.
- [Poynton03] Charles Poynton, *Digital Video and HDTV: Algorithms and Interfaces*, Morgan Kaufmann, Elsevier Science (USA), 2003.
- [Solari97] Stephen J. Solari, *Digital Video and Audio Compression*, McGraw-Hill, New York, 1997.
- [Topic02] Michael Topic, *Streaming Media Demystified*, McGraw-Hill, New York, 2002.

ROBOTIC APPLICATIONS

In this chapter

- Introduction
- Robotic Arm
- Actuation
- End Effector Path
- Sensing
- Tasking
- Automation and Autonomy

17.1 Introduction

Robotic applications are great examples of real-time embedded systems because they clearly use sensing and actuators to affect objects in the real world within the real-time physical constraints of environments that humans often operate in as well. Figure 17.1 shows the Toyota Robot, which is intended to have real-time coordination similar to a human trumpet player. Real-time applications also might have deadlines that are beyond human ability. A real-time system must simply operate within an environment to monitor and/or control a physical process at a rate required by the physics of the process. In the case of robotics, this is a distinct advantage that robotics have over human labor, the capability to keep up with a process that requires faster response and more accuracy than are humanly possible.

Furthermore, robots can perform repetitive tasks for long hours without tiring. Figure 17.2 shows an industrial robotic assembly line.

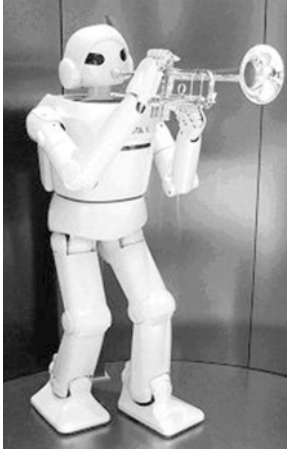


FIGURE 17.1 Toyota Robot



FIGURE 17.2 Industrial Robots on an Assembly Line

Robots are often deployed in controlled environments, such as assembly lines, rather than in uncontrolled environments, where humans often operate better, at least presently.

This chapter reviews basic concepts that are important to the design and implementation of basic real-time robotic systems.

17.2 Robotic Arm

The *robotic arm* approximates the dexterity of the human arm with a minimum of five degrees of rotational freedom, including base rotation, shoulder, elbow, wrist, and a gripper. The gripper can be a simple claw or approximate the dexterity of a human hand with individual fingers. In general, the gripper is often called an end *effector* to describe the broad range of devices that might be used to manipulate objects or tools. These basic arms are available as low-cost hobby kits and can be fit with custom controllers and sensors for fairly advanced robotics projects. The main limitations of low-cost hobby arms are that they are unable to grip and move any significant mass, offer less accurate and repeatable positioning, and are less dexterous than industrial or research robotic arms. Most industrial or research robotic arms have six or more degrees of freedom (additional wrist motion and complex end effectors) and can manipulate masses from one to

hundreds of kilograms. Robotic arms are often combined with computer vision with cameras either fixed in the arm or with views of the arm from fixed locations. A basic five-degree-of-freedom arm with end effector vision can be used to implement interesting tasks, including search, target recognition, grappling, and target relocation. Figure 17.3 shows the OWI-7 robotic trainer arm with a reference coordinate system.

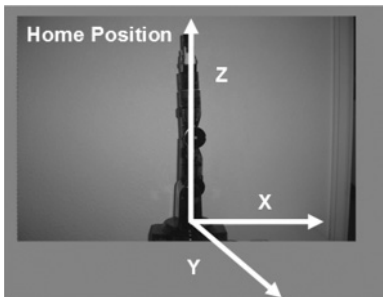


FIGURE 17.3 Robotic Arm Coordinates and Home Position

With a reference coordinate system with an origin at the fixed base for the arm, the reach capability of an arm can be defined based upon the arm mechanical design and kinematics. Figure 17.4 shows the OWI arm with elbow rotation so that the forearm is held parallel to the base surface. In this position, the base can be rotated to move the end effector over a circular trace around the arm base.

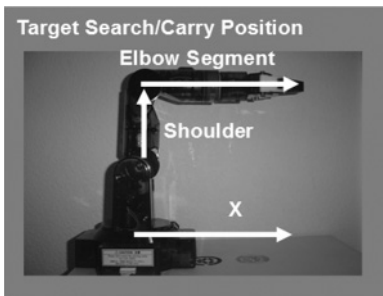


FIGURE 17.4 Robotic Elbow Rotation Only

The five-degree-of-freedom arm is capable of tracing out reachable circles around its base between an inner and outer ring. Figure 17.5 shows the innermost ring of reach capability for the OWI arm.

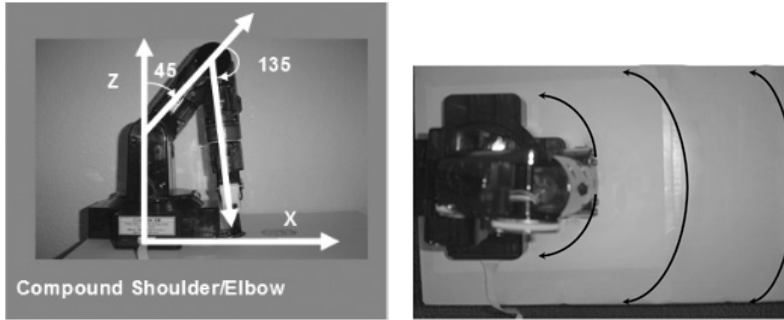


FIGURE 17.5 Innermost Surface Ring Reach Ability

Combined rotation of the shoulder and elbow allows the OWI arm end effector to reach locations on circular arcs around the base at various radii from the innermost ring. Figure 17.6 shows an intermediate ring of reach capability.

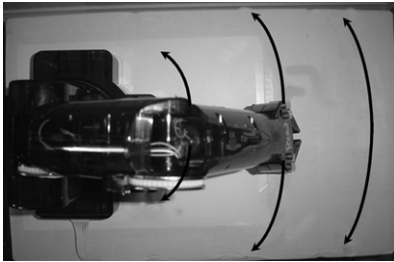


FIGURE 17.6 Intermediate Surface Ring Reach Ability

Finally, the outermost ring of reach capability for the OWI arm is defined by arm length with no elbow rotation and shoulder rotation so that the end effector reaches the surface. Figure 17.7 shows the limit of outermost reach capability.

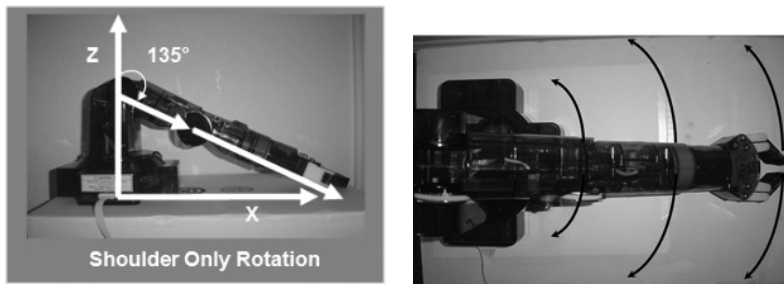


FIGURE 17.7 Outermost Surface Ring Reach Ability

This basic analysis considers only the surface reach capability of the OWI arm on its X-Y base plane. More sophisticated tasks might require three-dimensional reach capability analysis. After the kinematics and reach capability analysis has been completed so that the joint rotations are known for moving the end effector to and from desired target locations, an actuation and control interface must be designed.

17.3 Actuation

Actuation and end effector control is greatly simplified when the target object masses that the end effector must work with are negligible. Significant target mass requires more complex active joint motor torque control. Moving significant mass requires geared motor controllers with torque controlling DAC output. Another option for actuation is the use of stepper motors with active feedback control channels for each degree of freedom. For the OWI arm and negligible payload mass, the actuation can be designed using relays or simple H-bridge motor controllers. The motors must be reversible. The simplest circuit for reversing a motor can be implemented with switches to change the polarity across the motor leads, as shown in Figure 17.8.

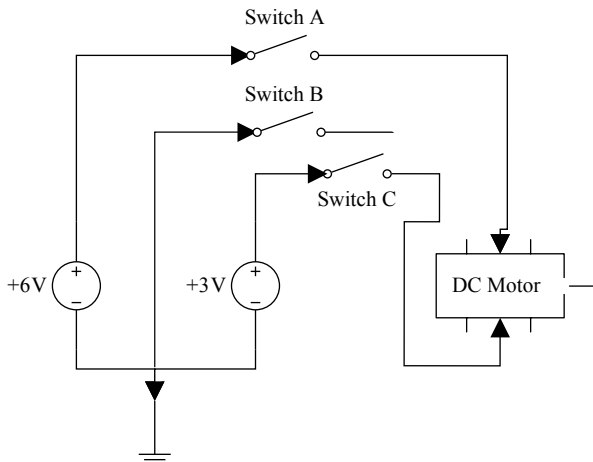


FIGURE 17.8 Three-Switch Reversible Motor

The possible switch states are enumerated in Table 17.1 along with the motor actuation provided.

TABLE 17.1 Three-Switch Reversible Motor Controls

SW-A	SW-B	SW-C	MOTOR
Off	X	Off	Off
Off	On	On	Forward
On	Off	On	Reverse

Setting three switches is not very practical because this requires three relays and therefore 15 total positions for a five-degree-of-freedom arm. Figure 17.9 shows how two relays can be used to implement the three-switch reversible motor circuit by using relays that include normally open and normally closed poles.

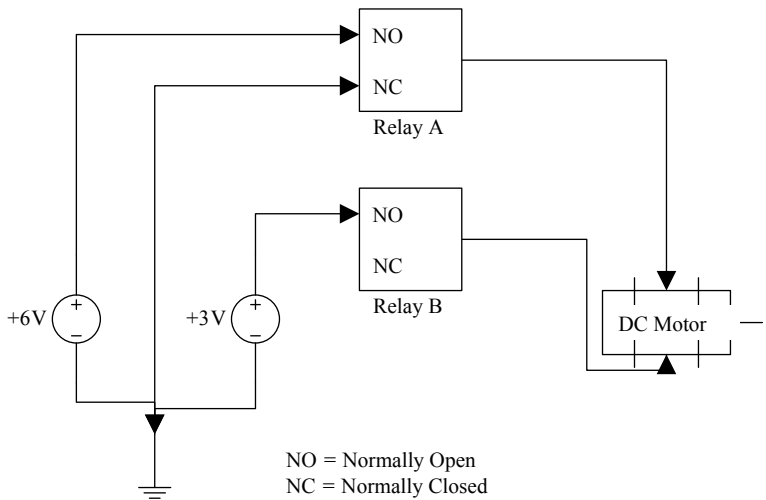


FIGURE 17.9 Two-Relay Reversible Motor

This simplifies the relay reversible motor actuation to 10 relays required for a five-degree-of-freedom arm. Table 17.2 summarizes the motor actuation as a function of the relay setting for this design.

TABLE 17.2 Two-Relay Reversible Motor Controls

RLY-A	RLY-B	MOTOR
Off	Off	Off
Off	On	Forward
On	Off	Off
On	On	Reverse

This is scaled to actuate a five-degree-of-freedom arm using 10 relays, as shown in Figure 17.10.

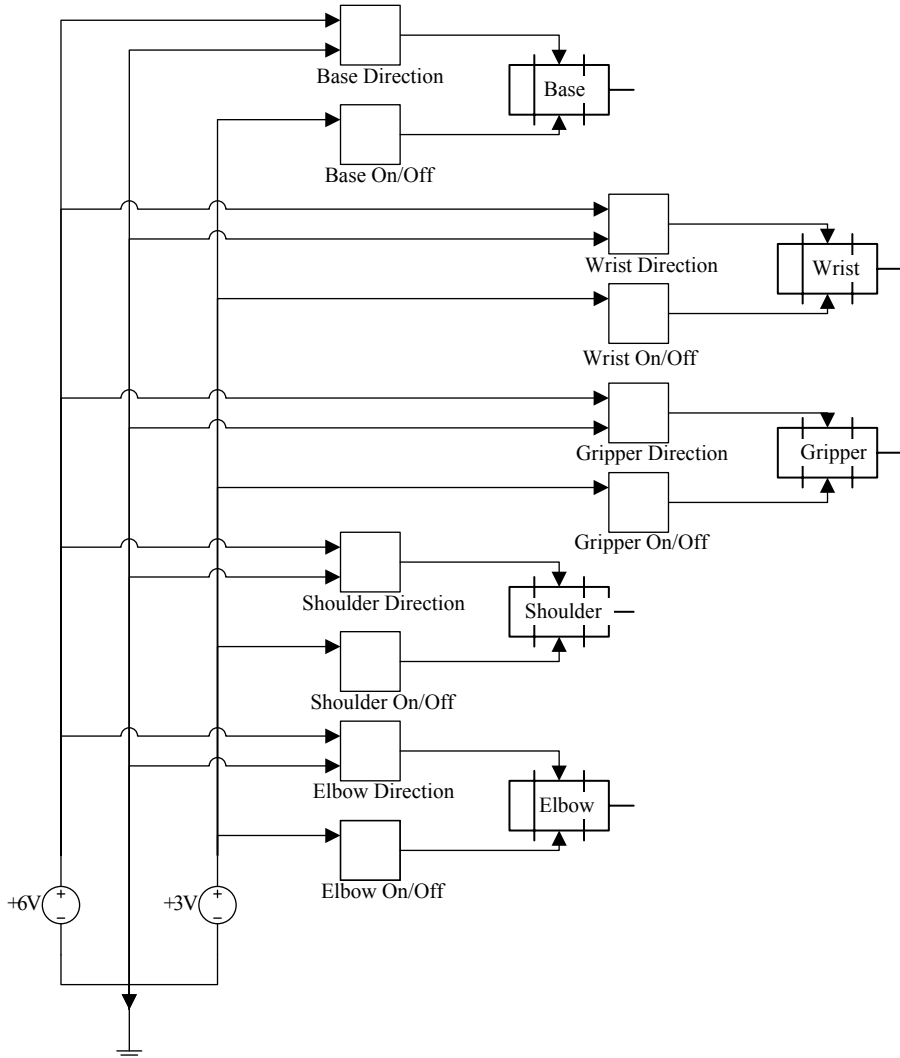


FIGURE 17.10 Five-Degree-of-Freedom Robotic Arm Relay Circuit

Actuation of a robotic arm can lead to mechanical arm failure if the motors are allowed to overdrive the joint rotations beyond mechanical limits of rotation for each joint. To avoid gear damage, a mechanical clutch or slip system can be employed, which is a feature of the OWI arm; however, reliance upon a clutch or slip system is still not ideal. Joints designed with mechanical slip or clutches can slip under the weight of the arm and cause positioning errors if they are too loose, and if they are too tight, overdriving

a joint will still cause gear damage. A better approach is to integrate hard- and soft-limit switches so that electrical and software protection mechanisms prevent overrotation of joints. Figure 17.11 shows a circuit design for hard-limit switches.

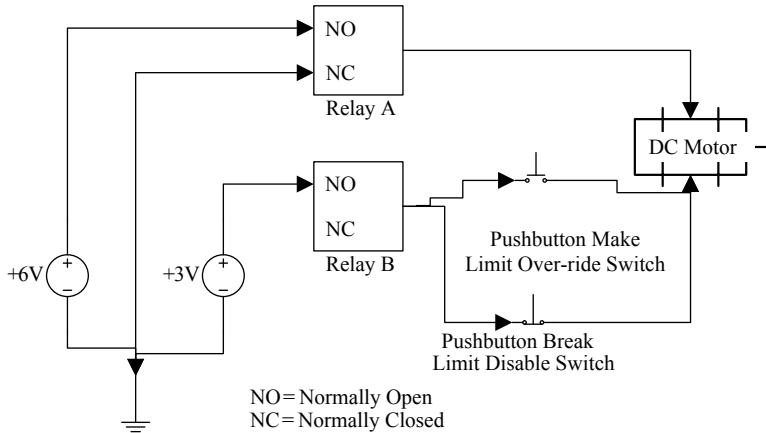


FIGURE 17.11 Use of Hard-Limit Switches for Arm Joint Motor Control

The limit switches shown in Figure 17.11 must be mounted on the arm so that the joints cause the switch to be activated at each limit of mechanical motion. The downside to this circuit is that the arm joint that hits a limit remains inoperable until it is manually reset.

A better approach is to use soft-limits monitoring with a software service that periodically, or on an interrupt basis, samples the output of a switched circuit through an ADC so that software can disable motors that hit a limit. This design is shown in Figure 17.12.

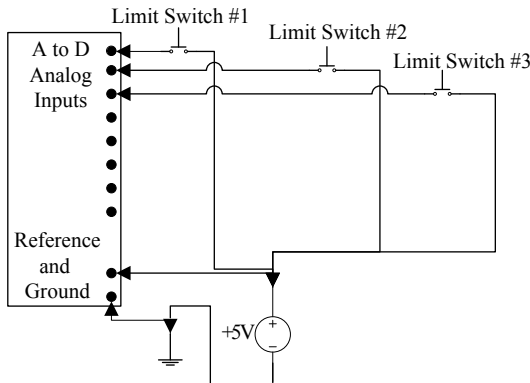


FIGURE 17.12 Use of Soft-Limit Switches for Motor Control Safing

Each of the soft-limit switches can be mechanically integrated so it will trigger before the hard-limit switches, allowing software to safe (disable) a potentially overrotated joint, decide whether a limit override for recovery is feasible, and then recover by commanding rotation back to the operable range. If software-limits monitoring fails or the software controller is not sane, the hardware limits will continue to protect the arm from damage. The relay actuation design with hard- and soft-limit switches provides basic arm actuation, but only with binary on/off motor control.

The concept of reversible motor poles can be generalized using relays in an H-bridge, providing more motor control states than the two-relay design. The H-bridge relay circuit is shown in Figure 17.13.

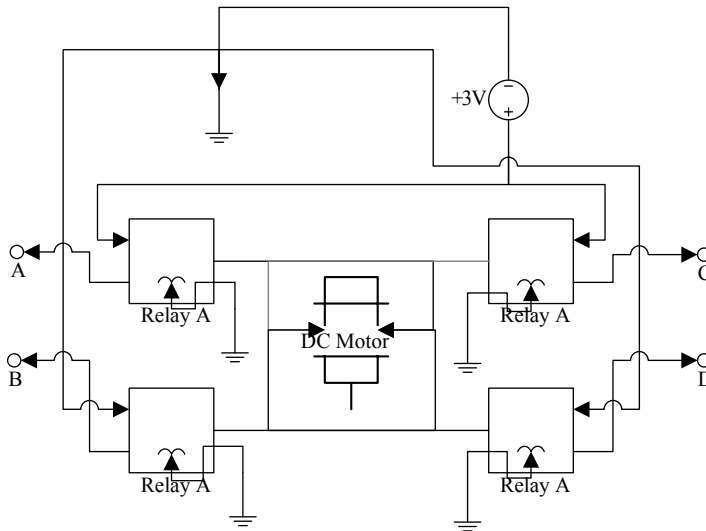


FIGURE 17.13 Relay H-Bridge Motor Control

Inspection of the relay H-bridge states shows that the H-bridge also provides additional control features, as listed in Table 17.3.

The braking features of an H-bridge can provide the basis for torque and overshoot control so that the motor controller can ramp up torque and ramp it down while positioning. The ramp-up can be provided by a DAC, and the ramp-down braking can be provided by the H-bridge braking states. The short-circuit states of the H-bridge, *fuse tests*, must specifically be avoided by H-bridge controller logic.

TABLE 17.3 Relay H-Bridge Motor Control States

A	B	C	D	MOTOR
0	0	0	0	Off
0	0	1	1	Brake
0	1	0	1	FuseTest
0	1	1	0	Reverse
1	0	0	1	Forward
1	0	1	0	FuseTest
1	1	0	0	Brake

Relay actuation provides only on and off motor control and requires the use of electromechanical relay coils, which create noise, dissipate significant power, and take up significant space, even for compact reed relays. Figure 17.14 shows the same H-bridge controller design as Figure 17.13, but using solid state MOSFETs (Metal Oxide Substrate Field Effect Transistors).

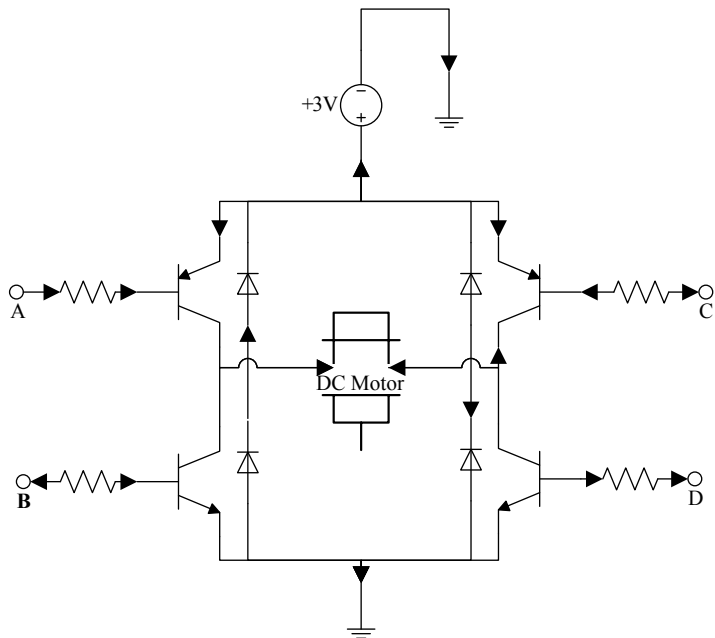


FIGURE 17.14 MOSFET H-Bridge Motor Control

The MOSFET design provides the same states and control as the four-relay H-bridge, but much more efficiently with less power required compared to electromagnetic coil actuation.

Since publication of the first edition, the real-time embedded systems course at the University of Colorado incorporated a number of off-the-shelf servo-controller five- and six-degree-of-freedom arms from CrustCrawler and Lynxmotion. The advantage of this is that students spend less time integrating cabling, switches, and relay systems with the DC motor version of the arm previously used and can focus on a multi-channel PWM controller as well as kinematics and kinetics for real-time control. The downside is that the experience with basic DC motor circuits did lead to some interesting exercises in hardware/software integration. The servo-controller arms have become much more affordable and have torque control (extra torque) output servos and a number of features that enable better control of end effectors and more emphasis on other aspects of computer vision-guided robotics.

17.4 End Effector Path

The ability to actuate an arm still does not provide the ability to navigate the arm's end effector to and from specific reachable locations. This must be done by path planning software and by end effector guidance. The simplest path planning and end effector guidance function implements arm motion dead reckoning and single-joint rotation sequences. *Dead reckoning* turns on a joint motor for a period of time based upon a rotation rate that is assumed constant, perhaps calibrated during an arm initialization sequence between joint limits. Using a dead reckoning estimation for joint rotation leads to significant positioning error, but can work for positioning tasks where significant error is tolerable. Moving one joint at a time is also tolerable for paths that do not need to optimize the time, energy, or distance for the motion between two targets. Many more optimal paths between two targets can be implemented using position feedback and multiple concurrent joint rotation tasking. Concurrent joint rotation is simple to do with a relay or H-bridge controller, although the kinematics describing the path taken is more complex. Motion feedback requires active sensing during joint rotation.

17.5 Sensing

Joint rotation sensing can be provided by joint position encoders and/or computer vision feedback. Position encoders include the following:

- Electrical (multi-turn potentiometer)
- Optical (LED and photodiode with light-path occlusion and counting)
- Mechanical switch (with a momentary switch counter)

Position encoders provide direct feedback during arm positioning. This feedback can be used to drive the feedback in a control loop when the arm is moved to a desired target position. This assumes the desired target is known, either pre-programmed or known through additional sensing, such as computer vision. Figure 17.15 shows the basic feedback control design for a position-encoded controlled process to move an arm from one target position to another.

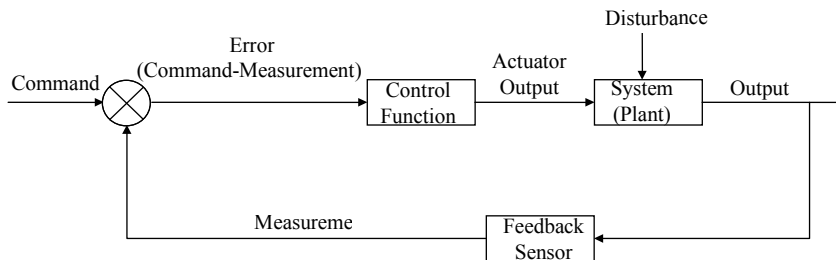


FIGURE 17.15 Basic Feedback Control Arm Positioning

Figure 17.15 can be further refined to specifically show an actuation with feedback design using relays and a potentiometer position encoder feedback channel. The main disturbance to constant rotation will come from stick/slip friction in the joint rotation and motor ramp-up and ramp-down characteristics in the motor/arm plant. Figure 17.16 shows this specific feedback control design.

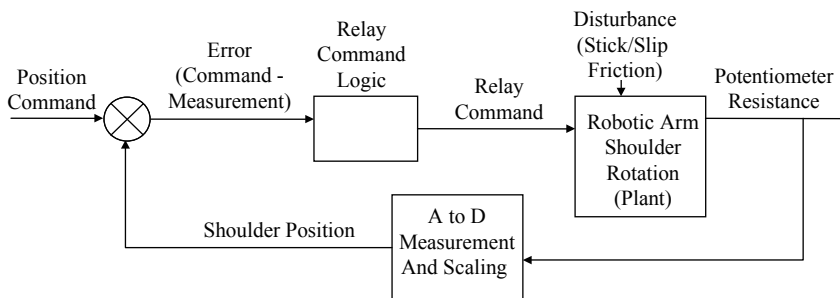


FIGURE 17.16 Relay Control with Position Encoder Feedback through an A/D Converter

Closer inspection of the design in Figure 17.16 reveals that the control loop has an analog and a digital domain, as shown in Figure 17.17.

This mixed-signal control loop design requires sampling of the feedback sensors and a digital control law. The control law can be implemented

for each joint on an individual basis as a basic PID (Proportional, Integral, and Differential) process control problem. For the PID approach, a proportional gain, an integral gain, and a differential gain are used in the control transfer function in Equation 17.1:

$$G_c(s) = K_p + \frac{K_i}{s} + K_d s = \frac{K_p s + K_i + K_d s^2}{s} \quad (17.1)$$

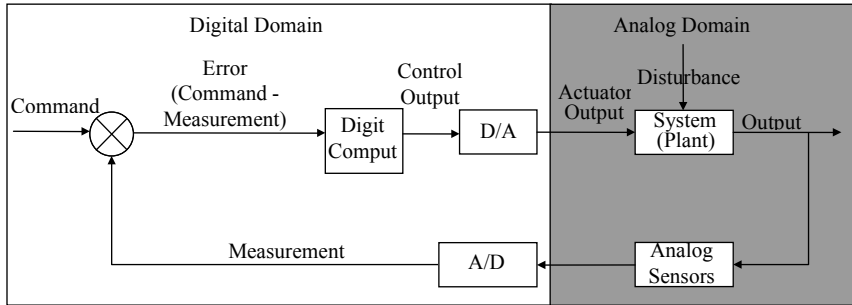


FIGURE 17.17 Arm Positioning with Feedback Digital and Analog Domains

The transfer function defined by a Laplace transform is depicted as a control loop block diagram in Figure 17.18.

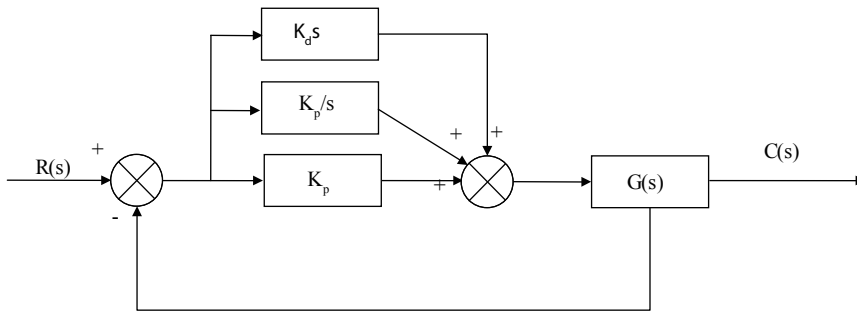


FIGURE 17.18 PID Control Loop

The Laplace transform for the PID control law makes traditional stability analysis simple; however, to implement a PID control law on a digital computer, a state space or time domain formulation for the PID control law must be understood. Furthermore, a relationship between the measured error and the control function output must be known in terms of discrete samples. This time domain relationship is shown in Equation 17.2.

$$u(t) = K_p y(t) + K_i \sum y(t) + K_d \frac{\Delta y}{\Delta t} \quad (17.2)$$

This equation can then be used to design the control loop as shown in Figure 17.19.

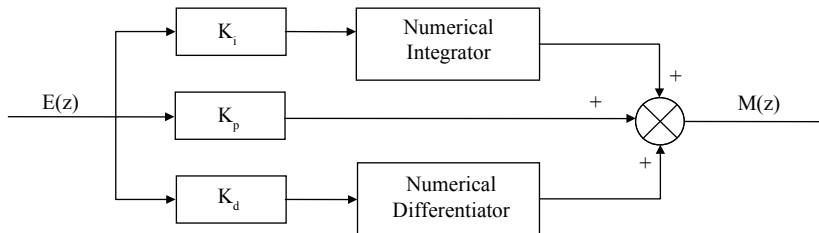


FIGURE 17.19 PID Digital Control Loop

The numerical integration can be done using an algorithm such as forward integration, trapezoidal, or Runge-Kutta over a series of time samples. Likewise, differentiation over time samples can be approximated as a simple difference. The proportional, integral, and differential gains must then be applied to the integrated and differentiated functions and summed with the proportional for the next control output. Applying these three components of the digital control law with appropriately tuned gains leads to quick rise time, minimum overshoot, and quick settling time, as shown in Figure 17.20.

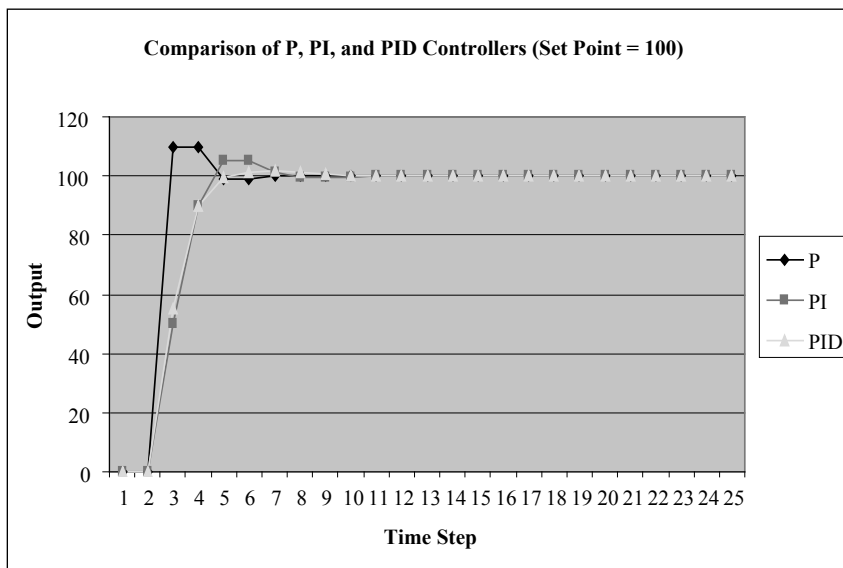


FIGURE 17.20 PID Time Response

Figure 17.20 shows proportional control alone, proportional with integral, and finally the full PID. Tuning the gains for a PID control law can be accomplished as summarized by Table 17.4.

TABLE 17.4 PID Gain Tuning Rules

Parameter	Rise time	Overshoot	Settling time
K_p gain increase	Decreases	Increases	Small change
K_i gain increase	Decreases	Increases	Increases
K_d gain increase	Small change	Decreases	Decreases

The PID controller provides a framework for basic single-input, single-output control law development. More advanced control can be designed using the modern control state space methods for multiple inputs and outputs. State space control provides a generalized method to analyze and design a control function for a set of differential equations based upon the kinematics and mechanics of a robotic system, as shown in Equation 17.3.

\mathbf{A} = system – matrix, \mathbf{B} = input – matrix

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \quad (17.3)$$

\mathbf{y} = output – vector, \mathbf{u} = input – vector

\mathbf{C} = output – matrix, \mathbf{D} = feed – forward – matrix

Figure 17.21 shows the feedback control block diagram for the generalized set of state space control system of differential equations.

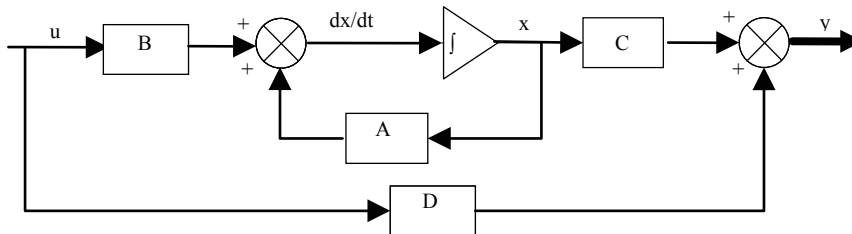


Figure 17.21 State Space Feedback Control

A detailed coverage of state space control analysis and design methods is beyond the scope of this book, but many excellent resources are available for further study [Nise03].

17.6 Tasking

The robotic arm must be commanded and controlled at a higher level than actuation and feedback control to provide these basic capabilities:

- Searching, identifying, and acquiring a visual target for pickup
- Path planning and execution to pick up a target
- Object grappling and grapple feedback
- Carry path planning and execution to relocate the object to a new target location

These three tasks compose the larger task of pick and place. The OWI arm command and control software and hardware actuation and feedback control system can be designed so that a single high-level command can initiate pick and place tasks within the arm reach ability space. The target search, identification, and acquisition task is a fundamental computer vision task. It requires an overhead, side, or front/back fixed camera system or a simpler embedded camera in the end effector. With an embedded camera, the arm can start a search sequence to sweep the camera field of view over the concentric reach ability rings in Figures 17.5, 17.6, and 17.7. While the camera is being swept over the rings, frame acquisition of NTSC frames at 30 fps or less can format the digital camera data into a digital frame stream. The digital frame stream images can be processed so that each frame is enhanced to better define edges and to segment objects for comparison matching with a target object description. The target object description can include target geometry and color (invariants) as well as target size (variant). Once the target is seen (matched with a segmented object in the field of view), then the arm can start to track and close in on the target.

Closing in on a visual target in the reach capability space of the OWI requires constant video to visual object centering based upon computation of the object's centroid in the FOV. The arm can use the centroid visual feedback to rotate the base to control XY plane errors to keep the object centered as the arm is lowered toward the XY plane. The kinematics requires that the arm shoulder and elbow be lowered to approach a target on the XY plane and to control the X translation of the end effector and embedded camera. Simultaneously the base rotation can be controlled to coordinate the target Y translation to keep the target centered as the arm is lowered. This basic task requires actuation and control of three degrees of

freedom at a minimum. The OWI wrist has only a single degree of freedom, which rotates about the forearm. More sophisticated robotic arms also include rotation about the other two axes of the wrist joint. So, a fixed camera embedded in the OWI arm may need independent tilt/pan control so that the camera angle can be maintained perpendicular to the XY plane as the arm is lowered. More sophisticated wrist degrees of freedom to allow for articulation with high accuracy (e.g., tenths of a degree) would also provide fine camera pointing.

Target pickup requires the arm to use position and limits feedback so that the arm knows when it has intersected the XY plane and acquired the XY plane-located target object. Furthermore, the grapppler should be in the fully open position at this time. Once this ready-to-grapple position has been achieved with feedback computer vision and positioning, the grapppler can be closed around the target object. Positive indication of successful target grappling can be provided by sensing switches built into the end effector fingers. Most often brass contacts separated by semiconducting foam (IC packing foam) or micro-switches with interface plates on each finger provide good feedback for grappling.

Once the target has been successfully grappled, the arm can now switch to a carry path planning task to guide it to the drop-off target location. This drop-off path planning might once again involve search or a pre-determined target position at a relative offset from the target acquisition location. Either way this is essentially identical to the acquire path planning and execution except that the arm is raised to a carry height at a desired reach ability ring, and then carried with base translation. The raise and carry sequence can be concurrent for a more optimal path in terms of time and distance traveled. The arm is again lowered to the drop-off target, and the target object is released, using the grapppler feedback to ensure that the fingers no longer sense a grip on the object.

This overall sequence is a high-level automation using tasking. It is still commanded by the operator, but the robotic arm performs the sub-tasks composing the overall task autonomously. Two major architectural concepts in the field of robotics have emerged that provide a framework for robot tasking and planning interfaced to lower-level controls and with or without human interaction. Two of the most important concepts are shared control and the degree of autonomy that a robotic system has along with Rodney Brooks's subsumption architecture. In the next section these architectural concepts are briefly introduced.

17.7 Automation and Autonomy

Figure 17.22 shows command and control loops for a robotic system that ranges from fully autonomous to telerobotic. Telerobotic operation is commonly known as “joystick” operation, where all robotic motion mimics operator inputs.

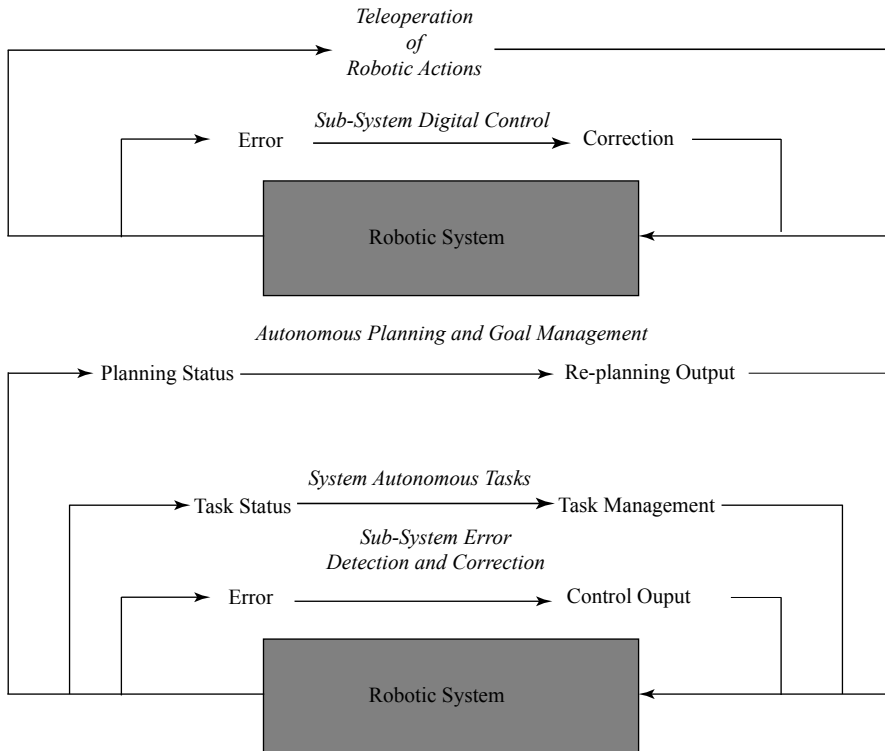


Figure 17.22 Teleoperated and Fully Autonomous Robotic Task Control Loops

An intermediate level of control between fully autonomous and telerobotic is called *shared control*. In shared control, some aspects of robotic tasking are autonomous, some are telerobotic, and others are automated but require operator concurrence to approve the action or initiate the action. The concept of shared control is shown in Figure 17.23.

Telerobotic systems may still have closed-loop digital control, but all tasking and decision making come from the operator; in the extreme case,

literally every movement of the robot is an extension of the user's movements, and no actuation is initiated autonomously. After the robotic action is commanded by the user, controlled motion may be maintained by closed-loop digital control. This is similar to concepts in virtual reality, where a user's input is directly replicated in the virtual model, and the concept of remotely piloting aircraft. For example, you could set up an OWI interface so that the OWI arm attempts to match the movement and position of the human arm based upon operator arm acceleration and flexure measurements.

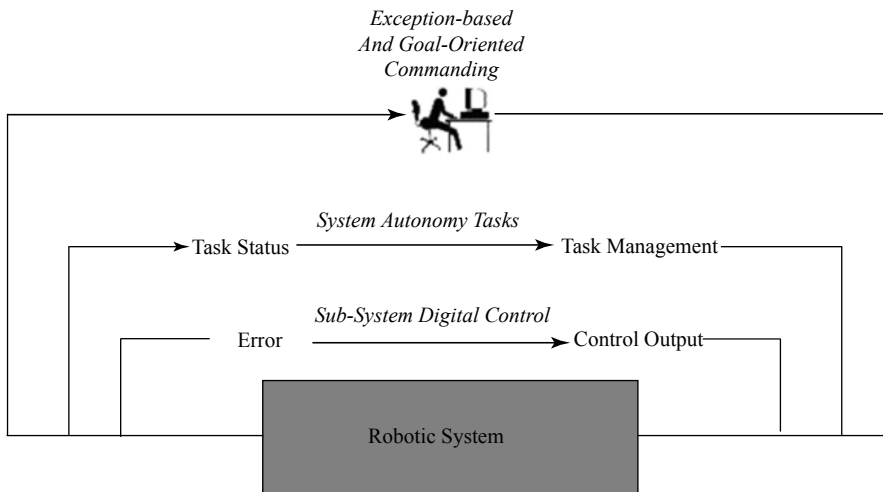


Figure 17.23 Shared Control of a Robotic System

Summary

Robotics requires a hierarchy of automation, the subsumption architecture, where low-level component actuation and sensing are interfaced to higher-level subsystem control. Subsystems in turn are tasked with goals and configured with behaviors. For example, a rolling robot may be tasked with exploring and mapping a room, but is also configured to have a collision avoidance behavior. This control, behavior, and tasking must be coordinated by an intelligent human operator or by artificially intelligent goal-based planning. This chapter introduced robotic system architecture and design from a bottom-up viewpoint, providing practical examples for how to control and task a five-degree-of-freedom robotic arm.

Exercises

1. Develop a kinematics model for the reach ability of the OWI five-degree-of-freedom robotic arm using a C program or MATLAB model.
2. Build a simple single-change relay or MOSFET H-bridge reversible motor controller, and design a serial or parallel port command interface for it.
3. Implement a PID control C function with tunable gains that can be used to control a robotic arm joint rotation or similar actuator.

Chapter References

- [Brooks86] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot,” IEEE Journal of Robotics and Automation, Vol. RA-2 (April 1986): pp. 14–23.
- [Nise03] Norman S. Nise, Control Systems Engineering with CD, 4th ed., Wiley, New York, 2003.

Chapter Web References

- The OWI-7 five-degree-of-freedom arm comes from OWI Robotics and offers a great way to learn about basic robotics: <http://owirobots.com/>.
- Lynxmotion offers the Lynx 5 and 6 servo controlled robotic arm kits with good repeatability and precision: <http://www.lynxmotion.com/>.
- The CrustCrawler robotic arms with torque controlled servos: <http://www.crustcrawler.com/products/AX-18F%20Smart%20Robotic%20Arm/>.
- Garage Technologies Inc. also offers a six-degree-of-freedom arm: <http://www.garage-technologies.com/index.html>.
- Robotics Research manufactures seven-degree-of-freedom highly dexterous arms with torque control: <http://www.robotics-research.com/>.
- Honda’s ASIMO robot: <http://world.honda.com/ASIMO/>.
- RobotWorx industrial robotics integration: <http://www.robots.com/>.
- Motoman industrial robotics: <http://motoman.com/>.
- ABB industrial robotics: <http://www.abb.com/robotics>.
- NASA Johnson Space Center Robonaut: http://vesuvius.jsc.nasa.gov/er_er/html/robonaut/robonaut.html.

COMPUTER VISION APPLICATIONS

In this chapter

- Introduction
- Object Tracking
- Image Processing for Object Recognition
- Characterizing Cameras
- Pixel and Servo Coordinates
- Stereo-Vision

18.1 Introduction

Computer vision requires video frame acquisition, digital video processing, and the use of information extracted from the image stream to control a process or provide information. For example, a stereo mapping vision system can move a laser pointer to positions and measure distance to the reflected spot over and over to create a three-dimensional model of a room. Or, computer vision might be used by a robotic platform to navigate a vehicle or end effector to a visual target. Processing video streams is inherently a real-time process because the processing must keep pace with a periodic video frame rate. Furthermore, when video processing is embedded in a digital control loop where the digital video is used as feedback sensing, the overall digital control loop must meet real-time requirements for control stability. Computer vision is an excellent application for studying real-time concepts due to clear and obvious real-time processing requirements.



Since publication of the first edition of this volume, OpenCV examples have been added to the DVD for scene segmentation, stereo correspondence, and numerous other examples that complement the original material. A nice approach for designing real-time computer vision applications is to prototype image processing using MATLAB, GIMP or Octave (interactive image processing environments), and then to move to desktop OpenCV, and finally to embedded OpenCV.

18.2 Object Tracking

Object tracking with computer vision is a basic vision feedback control problem used in many automation and instrumentation applications. For example, telescopes automatically track celestial targets and can scan the sky to find them initially and to center objects of interest in the telescope field of view (FOV). Assembly-line robotics that spot weld structures can use position feedback for coarse alignment, but often use computer vision for fine positioning and position verification before applying a weld. Space probes often use optical navigation in deep space where radio tracking and navigation may be difficult due to light time latencies at great distances. A basic bright object or shape/color target tracking camera system can be constructed from two servos and a camera. Figure 18.1 shows a front and side view of a tilt/pan camera tracking subsystem.

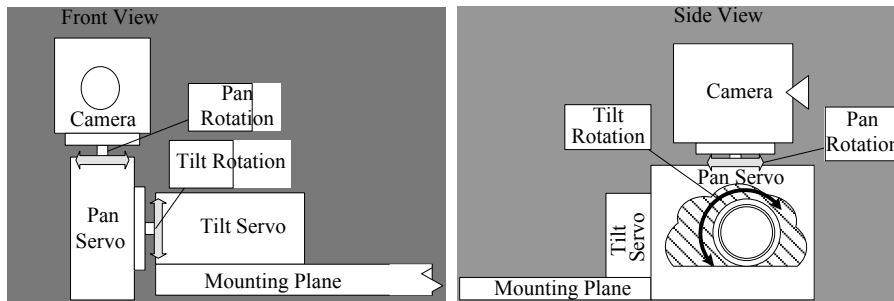


FIGURE 18.1 Tilt/Pan Computer Vision Tracking Subsystem

In this design, the camera FOV is moved to track an object in motion or is moved to scan a larger FOV to find an object. Alternatively, a fixed camera with a large FOV might observe a target illuminator, such as a laser pointer, which is tilted and panned instead of the camera. This can be useful for stereo ranging or simply tracking the motion of an object.

An observer can judge how well the fixed camera is able to locate a given object by seeing how well the laser spot is able to track the object, providing a tracker debug method.

Figure 18.2 shows a stereo-vision tracking subsystem. This subsystem can be used to track not only the XY location of an object in a FOV but also the distance to it from the camera baseline.

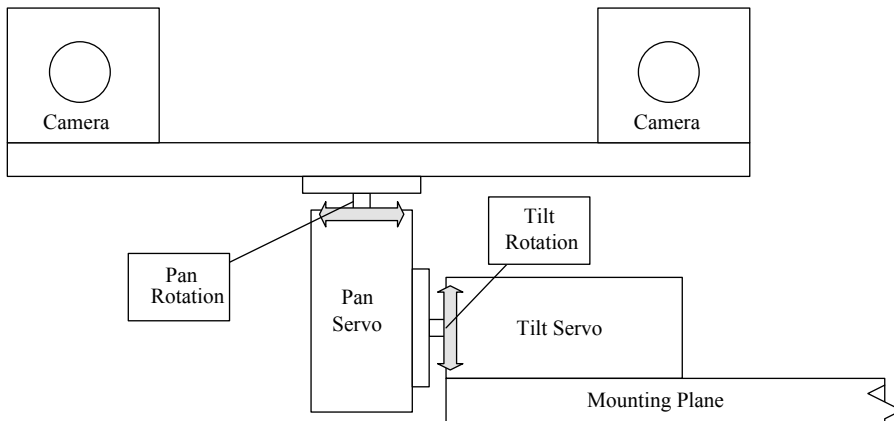


FIGURE 18.2 Stereo Tilt/Pan Computer Vision Tracking Subsystem

The tracking problem can be handled in almost the same fashion as the mono-vision tracker; however, either one camera can drive the tilt/pan control to keep the target centered, or a scheme to use information from both can be used. If just one camera is used, right or left side, then the parallax of the object as seen by one camera will not be the same as the other. For more accurate stereo-vision systems, each camera, separated by a common baseline, can be independently tilted and panned on the baseline to keep the object centered as human vision systems do. Most stereo tracking and ranging systems simply tilt and pan the entire baseline and either favor one camera or take an average from the two to split the difference on centering the object in each camera's FOV.

The tracking performance for a tilt/pan subsystem is based on the following:

- Vision frame rate
- Servo actuation limits
- Digital control processing latency

Ideally the frame rate is high enough and the processing latency low enough such that the system is servo-limited. The processing must be completed before the next observation frame is available (deadline is the same as service request period). As discussed in Chapter 14, a PID control loop can be designed to optimize tracking control for quick rise time, minimal overshoot, and quick settling. With a mono-vision tracker it is easy to observe tracking control issues since tracking latency, overshoot, and settling can be observed easily with system testing. Tracking requires not only that the image processing recognize the object to be tracked, but also that the edges can be detected and the geometrical center of the object computed. With this information from image processing, the tilt/pan error can be calculated in terms of pixel distances between the center of the image and the center of the object to be tracked. The center of an object is called the centroid. In the next section methods for finding a centroid are presented.

18.3 Image Processing for Object Recognition

Visual objects can be recognized based upon shape, size, and color. Color is the most invariant property because it changes only with changes in lighting. If lighting can be controlled or if the system can provide illumination, this is a huge advantage for a computer vision system. Shape is also mostly invariant, changing only when an object orientation is changed for non-symmetric objects. Symmetric objects, such as spheres, are always easier to detect and track. A highly focused laser spot is also fairly easy to track, but any defocusing or beam spread can create problems. Finally, size is the most variant because it changes with distance to the object in the Z plane. Size can be ignored, or if the actual size of an object is known, it can actually be used to judge distance even with mono-vision. Tracking color and shape is most often used. Colors can be detected based upon luminance and chrominance levels or RGB levels for each pixel. Shape is best determined by enhancing and detecting object edges in a scene.

A simple PSF (Point Spread Function) can be applied to a digital luminance (grayscale) or RGB image to enhance object edges. For example, Table 18.1 provides a commonly used PSF for edge enhancement:

TABLE 18.1 Edge Enhancement Kernel

$-k/8$	$-k/8$	$-k/8$
$-k/8$	$k+1$	$-k/8$
$-k/8$	$-k/8$	$-k/8$

This matrix is applied to an image array such that each pixel is replaced by the sum of $-k/8$ times each neighboring pixel and $k+1$ times itself.

The DVD contains C functions and examples for applying this kernel to 320x240 RGB and grayscale images. To apply the kernel to a 320x240 pixel image it is applied to pixel address {1, 1} through {1, 318} for the first row, likewise for each row, and through pixel address 238,318 for the last row. Figure 18.3 shows the standard convention for image pixel addressing, starting with {0, 0} in the upper-left corner down to {319, 239} in the lower-left corner.

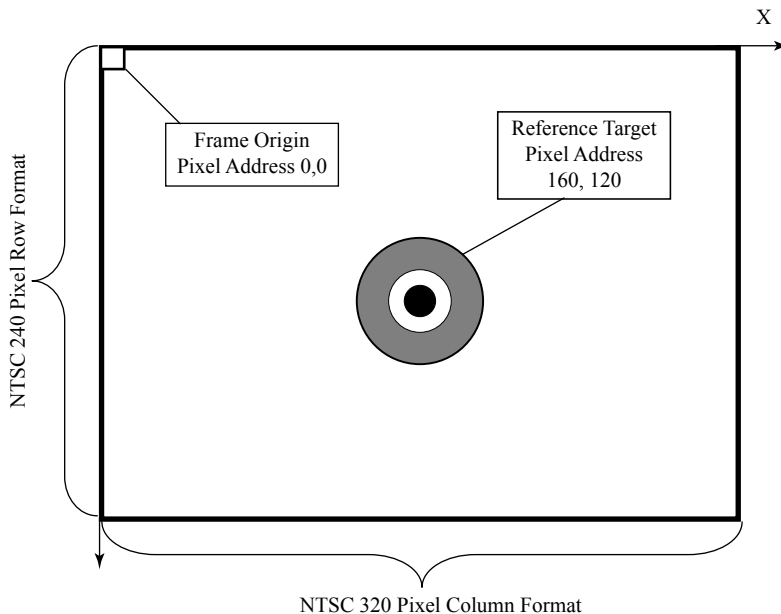


FIGURE 18.3 Image Coordinates for a 320x240 Pixel Frame

Applying this kernel as described previously along with a threshold filter yields the image enhancement for a red circular target as shown in Figure 18.4 in both color and grayscale (color available on the DVD).



Edge enhancement alone finds edges that are not necessarily of interest. The threshold filter eliminates background edges based on color thresholds so that only edges matching color criteria remain in the final enhanced and filtered image. Now that the target image has been segmented from other objects in the scene, the centroid of the target can be found by walking through each row of the image and recording the X and Y location of



object entry and exit based upon color or intensity threshold. The X, Y locations define the object boundary, and the centroid can be computed by finding the center of the greatest X extent and the center of the greatest Y extent for a symmetrical object, such as a circle. Example code for centroid location is included with the DVD. Figure 18.5 shows the centroid found by rastering with the location indicated by adding green lines to the image array.

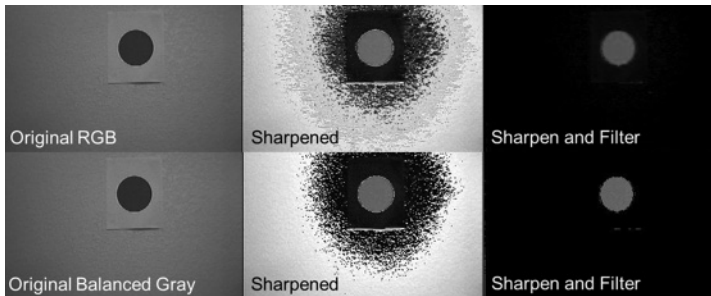


FIGURE 18.4 Edge Enhancement and Filtering for Common Color and Grayscale Image

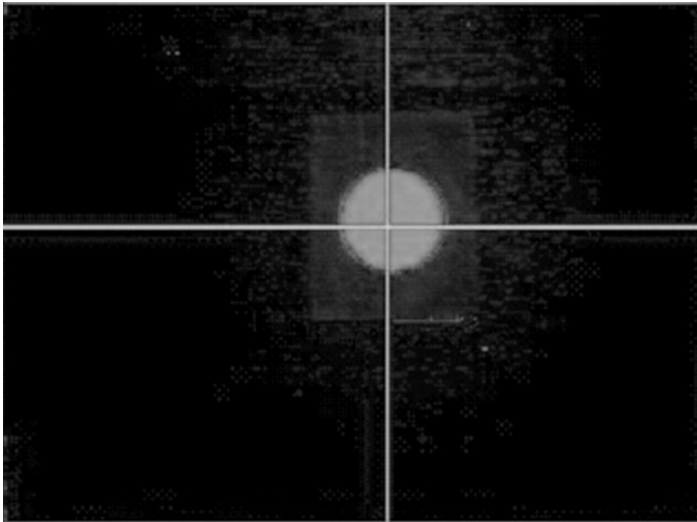


FIGURE 18.5 Centroid Found by Rastering Enhanced Target Image

This basic process must be tuned for the lighting conditions, the target shape and color, and the camera characteristics. It's often simpler to segment scenes, detect objects, and find extents and centroids in grayscale or one-color band rather than RGB color space. To better understand how to

tune target recognition and analysis, it's best to start with camera characterization and to control lighting.

18.4 Characterizing Cameras

Cameras have FOVs (extents that can be seen through the camera lens at a given distance) that are determined by the camera optics. Furthermore, optical-electrical cameras, such as NTSC CCTV cameras, have detectors with arrays of charge-sensitive devices, most often a CCD (Charge Coupled Device). The pixel detectors most often charge up when exposed to light. The charge voltage can be read out from the array of pixel detector devices through an ADC interface to digitize data. An NTSC camera actually produces an analog signal (described more fully in Chapter 16, “Continuous Media Applications”) from the CCD readout, which in turn is sampled and digitized by a frame grabber. Given the overall system, the images can be affected by camera optics, detector physics, and video signal sampling and digital conversion in the frame grabber. Lighting can saturate pixel detectors or ADC in the frame acquisition interface. Controlled lighting and well-matched ADC sensitivity should be used to ensure that an image is not washed out or dark. Washed-out images have average intensity that is way above the mid-level output of 128 for an 8-bit pixel. Dark images, of course, have average intensity that is way below a low-level output of, say, 10, for example. Robotic systems often employ target illumination. They carry their own lighting and calibrate the illumination to ensure that images are neither washed out or dark.

Basic characterization of the camera optics and NTSC output can be achieved by interfacing the camera up to a television and taking FOV measurements. A tape measure can be placed in the FOV, and the camera distance can be varied to determine the physical extents observable by the camera optics and detector as a function of distance. Figure 18.6 shows this relationship for a CCTV NTSC camera for X extents measured with optical observations of a rule.

For this camera, the relationship is very linear; however, optical affects often cause nonlinear variation in extents. For example, many lower-cost optical systems cause optical aberrations, such as *fish-eye*, where the extents of an object are exaggerated in the center of the FOV. So, a square object appears to bulge in the middle. The camera should be carefully characterized so that any nonlinear effects can be corrected by curve fitting if they are significant.

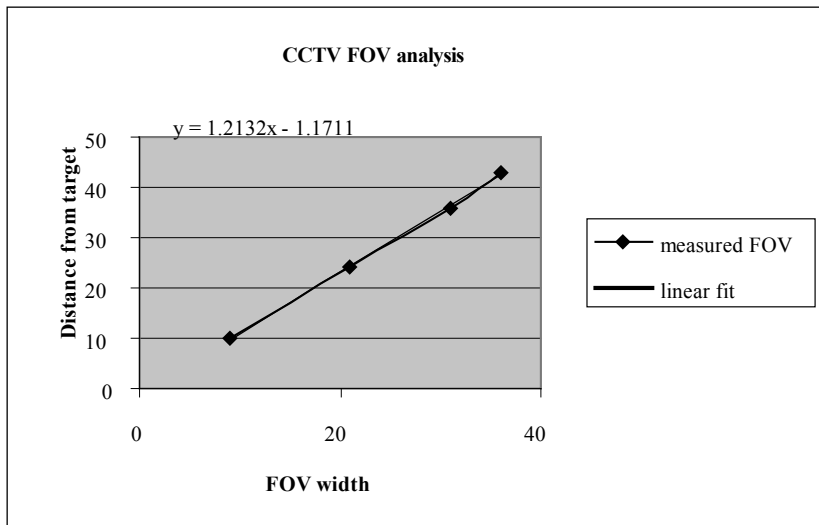


FIGURE 18.6 Physical Extents Visible in Camera FOV as a Function of Distance

For the same CCTV NTSC camera, the detector is larger than the NTSC sampled digital output. The number of detector pixels in the C-Cam8 NTSC thumbnail camera is 510×492 ; however, the Bt878 frame grabber can convert this into a 320×240 pixel image. The FOV can be characterized further in this sample space by calculating the sample pixels per physical inch as a function of camera distance from target. Figure 18.7 shows that this is a nonlinear relationship for the same CCTV NTSC camera optical/NTSC characteristics shown in Figure 18.6. The nonlinearity is therefore introduced in the NTSC sampling and digitization by the Bt878 encoder for the 320×240 RGB format.

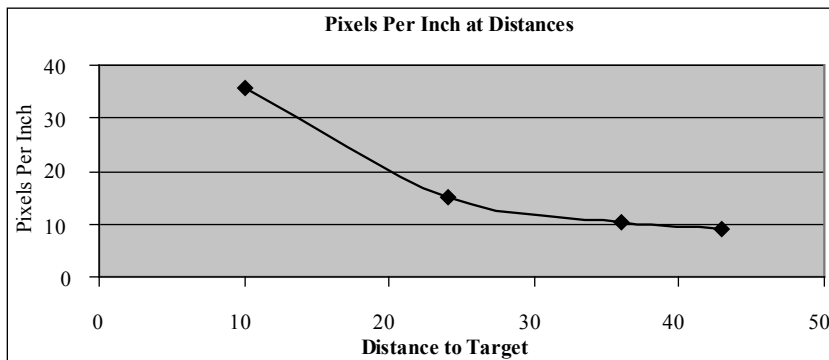


FIGURE 18.7 Physical Extent Relationship to Sampled Pixels for a CCTV Camera

It's important that the camera and overall image acquisition system be characterized to identify and model the mapping from physical space to sample pixel coordinates because all image processing will be done with the sample pixels.

18.5 Pixel and Servo Coordinates

As shown earlier in Figure 18.3, pixel coordinates are addresses ranging from $\{0, 0\}$ to $\{n, m\}$ for an n by m image. Servo coordinates are defined by the X extent traversed at a distance by the smallest unit of pan rotation the servo can provide. The Y extents are likewise defined in servo coordinates for a given distance. This is easy to see if a laser pointer is attached to the tilt/pan platform so that the smallest tilt/pan servo motions can be charted on a wall by noting locations of the laser spot and physically measuring the distance between spots. Knowing the relationship between one unit of servo tilt or pan in terms of overall camera system FOV pixel translation can be very helpful for calibration of a tilt/pan system. Figure 18.8 shows an automatable process for determining the servo to pixel coordinate function on the x- and y-axis for a given distance.

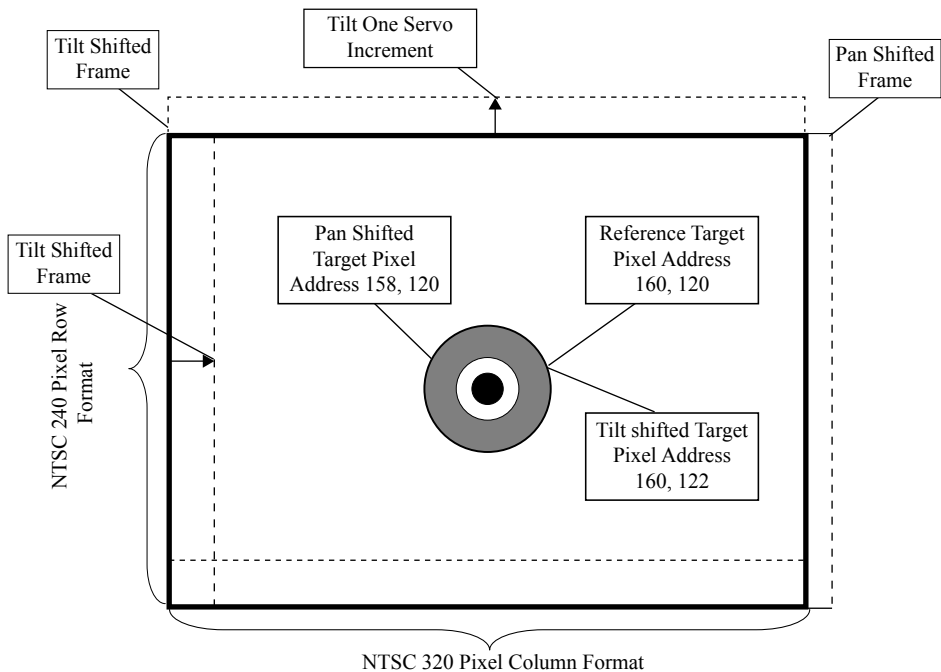


FIGURE 18.8 Pixel to Servo Coordinate Calibration

This information can be used to compute the servo rotation required to center an object in the FOV. So, for an optical tracker set up where a single servo increment causes a 2-pixel FOV change, then when a target is 10 pixels right of the center, the servo must be panned 5 servo increments right. This relationship is linear near the center of the camera tilt/pan and becomes increasingly nonlinear at maximum tilt/pan. So be sure to characterize the relationship over target range extents observable at maximum tilt/pan corners of the effective tilt/pan FOV.

18.6 Stereo-Vision

Stereo-vision is based upon the phenomena of parallax, where an observation at two locations along a baseline of a common object appears to cause an offset of the object. This can be observed by holding one finger out, closing the right eye and observing, and then closing the left and opening the right, causing the finger to apparently change position. In fact, the observation is due to the change in eye observation angle for the object. The apparent shift of the object therefore is based upon its distance from the observation baseline. Distances to stars are measured using parallax by observing a star one half-year apart as the earth orbits the sun and defines the unit in distance called a *parsec*. Likewise, two cameras on a common baseline can be used to observe a common object, and the apparent shift of the object in the right camera's FOV compared to the left camera can be used to compute the distance from the baseline to the target. The geometry for a stereo observation is illustrated in Figure 18.9. It is important to note that the diagram is valid only for a simple coplanar camera mount with identical left/right cameras and ignores the challenges of accounting for inaccuracies in the external camera mounts (extrinsic errors) as well as internal camera characteristics like fish-eye (intrinsic errors).

Note that the triangle formed by the target, left lens, and baseline center is similar to the triangle formed by the left lens, detector center, and the image offset dl . The triangle on the right side formed by the target, right lens, and baseline center is similar to the triangle formed by the right lens, detector center, and the image offset dr . By the property of similar triangles, $(dl + dr)/f = b/d$. The baseline b is known, the focal length f is known for a given camera (or can be found by characterization of the camera), and dl and dr can be computed from the offsets of the target centroid from each camera FOV center. This leaves d , the distance to the target, as the only

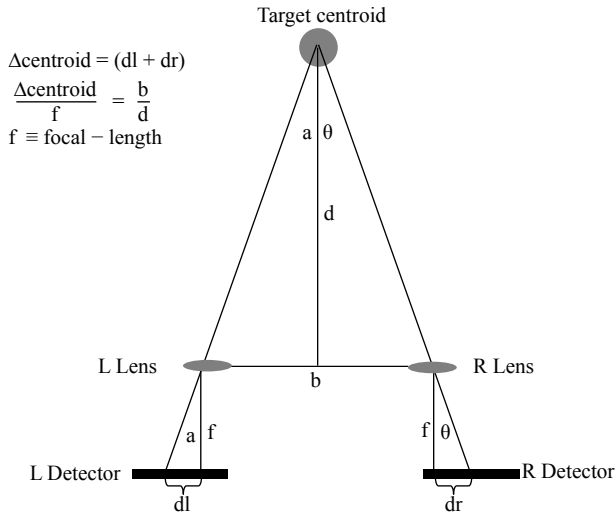


FIGURE 18.9 Stereo-Vision Observation Geometry

unknown. New examples developed in OpenCV have been included on the DVD in this new edition, but a full exploration of computer vision, OpenCV, and general issues of continuous image processing goes beyond the scope of this book. The reader should refer to the references provided.



Summary

Stereo-vision systems emulate human vision systems and can track and judge distances in real-time much like we can. Many tasks that we intuitively would call real-time are tasks that we complete using vision, audio, body kinematics, and kinetics, and that are governed by real-world physics. Many real-time embedded tasks are also similar to human real-time tasks. This is one reason that studying robotics, vision systems, video/audio recording, transport, playback, and digital control is an excellent application for understanding real-time theory. Note, however, that a real-time system is not fast, slow, or necessarily constrained by the same physics or timescales that humans are. Real-time systems must simply complete requests for service prior to a deadline relative to the service request time. Furthermore, embedding places real-time computing into specific service requirements and requires observation and response for process control. Often the most useful real-time embedded systems operate in dangerous

environments and must complete tasks much faster, more accurately, and with less fatigue than a human ever could. An industrial welding robot on an automobile assembly line provides exactly this type of invaluable real-time service. Similarly, an earthquake fault-line monitoring system might need to detect land mass movement only over very long periods of time with slow yet highly accurate measurements completed within a deadline. So, while the intuitive real-time tasks are very helpful for understanding real-time embedded systems and applications, the fundamental theory and broader applications should also be studied and appreciated.

Exercises

1. Implement a C function with tunable gains for a PID control law that can be used for a tilt/pan tracking subsystem. Build a tilt/pan subsystem, integrate the `btvid.c` or Linux `bttv` driver, and track a moving object using image processing to follow the object centroid.
2. Implement a stereo range finder for a known target, and show that it can correctly read the distance.
3. Implement a stereo range finder that can also tilt/pan track a moving object.
4. Implement a stereo imaging system that uses a laser pointer to paint a three-dimensional scene, and create a three-dimensional map of the area.
5. Install OpenCV on the Jetson board running embedded Linux, and download, build, and test some of the OpenCV examples found on the DVD.



Chapter References

- [Bradski12] Gary Bradski and Adrian Kaehler, *Learning OpenCV*, 2nd ed., O'Reilly, Sebastopol California, 2012.
- [CameraLink] "Specifications of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers," <http://www.imagelabs.com/wp-content/uploads/2010/10/CameraLink5.pdf>
- [Cutting95] James E. Cutting and Peter M. Vishton, "Perceiving Layout and

- Knowing Distances: The Integration, Relative Potency, and Contextual Use of Different Information about Depth,” in *Perception of Space and Motion*, W. Epstein and S. Rogers, Eds., Academic Press, Waltham Mass., 1995.
- [Davies12] E. R. Davies, *Computer and Machine Vision: Theory, Algorithms, Practicalities*, Elsevier, Amsterdam Netherlands, 2012.
- [Duda72] Richard Duda and Peter Hart, “Use of the Hough Transformation to Detect Lines and Curves in Pictures,” *Communications of the ACM*, Vol. 15, No. 1 pp. 11–15, (January 1972).
- [Hill97] Rhys Hill, Christopher Madden, Anoton van den Hengel, Henry Detmold, and Anthony Dick, “Measuring Latency for Video Surveillance Systems,” Australian Centre for Visual Technologies, School of Computer Science, University of Adelaide, 1997.
- [Lowe04] David G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision* Volume 60, Number 2, pp. 91–110, (January 2004).
- [OpenCV] OpenCV, <http://opencv.org/>
- [OpenNI] OpenNI application programmer’s interface, <http://www.openni.org/>
- [PCL] Point Cloud Library, <http://pointclouds.org/>
- [Prince12] Simon J. D. Prince, *Computer Vision: Models, Learning, and Inference*, Cambridge University Press, Cambridge England, 2012.
- [Ren13] Xiaofeng Ren, Dieter Fox, and Kurt Konolige, “Change Their Perception—RGB-D Cameras for 3-D Modeling and Recognition.” *IEEE Robotics and Automation Magazine*, December 2013.
- [Siewert06] Sam Siewert, *Real-Time Embedded Components and Systems*, Charles River, Charlestown Mass., 2006 (ISBN 1584504684).
- [Siewert14a] Sam Siewert, “Big Data Interactive: Machine Data Analytics—Drop in Place Security and Safety Monitors,” IBM developerWorks, January 2014.
- [Siewert14b] Sam Siewert, M. Ahmad, and K. Yao, “Verification of Video Frame Latency Telemetry for UAV Systems Using a Secondary Optical Method,” AIAA SciTech, National Harbor, MD, January 2014.
- [Siewert14c] Sam Siewert, J. Shihadeh, R. Myers, J. Khandhar, and V. Ivanov, “Low Cost, High Performance and Efficiency Computational Photometer

- Design,” SPIE Sensing Technology and Applications, Baltimore, MD, May 2014.
- [Siewert15] S. Siewert and J. Pratt, *Real-Time Embedded Components and Systems Using Linux and RTOS*, 2nd ed., Mercury Learning and Information, Dulles, VA, August 2015 (ISBN 978-1-942270-04-1).
- [Szeliski11] Richard Szeliski, *Computer Vision: Algorithms and Applications*, Springer, New York, 2011.
- [Viola04] Paul Viola and Michael Jones, “Robust Real-Time Face Detection,” *International Journal of Computer Vision* Volume 57, Number 2, pp. 137–154, (2004).
- [Wagstaff13] K. L. Wagstaff, J. Panetta, A. Ansar, R. Greeley, M. Pendleton Hoffer, M. Bunte, and N. Schörghofer, “Dynamic Landmarking for Surface Feature Identification and Change Detection,” *ACM Transactions on Intelligent Systems and Technology*, Vol. 3, No. 3, Article 49 (2012).
- [Xilinx] Low-voltage differential serial CameraPort, as documented by Xilinx, http://www.xilinx.com/support/documentation/application_notes/xapp582-ccp2-sublvds-hr-io.pdf



TERMINOLOGY GLOSSARY

- **Actuator:** Electromechanical device that converts analog or digital electrical inputs into mechanical energy interacting with the physical world.
- **ADC:** Analog to digital converter; encodes analog signals into digital values.
- **Amdahl's Law:** If F is the sequential portion of a calculation, and $((1 - F)$ is the portion that can be executed concurrently (in parallel), then the maximum speedup that can be achieved by using N processors is $1/(F + ((1 - F)/N))$. So, for example, if $F = 0$, then with $N = 4$, speedup is linear and equal to 4, but if $F = 50\%$, then speedup is only 1.6 times faster for 4 times as many processors.
- **API:** Application programmer's interface; provides function call interface to lower-level software and/or hardware functionality.
- **Application executive:** Also known as a *cyclic executive*, a main loop program that calls functions on a periodic sub-rate of the main loop period.
- **Asynchronous:** An event or stimulus that occurs at any point in time rather than at known predictable points in time—for example, an external interrupt may occur at any time and will immediately change the thread of execution on a *CPU*.
- **Asynchronous logic:** Digital logic that is not globally clocked, but rather changes state based on edge triggering in a combinational logic circuit or edge triggered by multiple independent clocks.

- **Atomic operation:** A non-interruptable CPU instruction—that is, any instruction that can be fetched and completed before the CPU can be interrupted.
- **Bandwidth:** Data transfer per unit time—for example, bytes/second.
- **BDM:** Background debug mode; a variant of *JTAG* that allows data and instructions to be clocked into and out of a 10-pin interface to a processor.
- **Best effort:** Scheduling policy that does not guarantee any particular response time for a service request, but attempts to make progress on all such requests and maximize total throughput.
- **Binary semaphore:** A *semaphore* that has only two states: full and empty. A take on an empty binary *semaphore* will *block* the calling *thread*, and a take on a full binary *semaphore* will change the state to empty. A give on an empty binary *semaphore* will change the state to full, and a give on a full *semaphore* has no effect.
- **Black-box test:** A set of test vectors and driver that operate only on the functional interface of a subsystem or system with no knowledge of the internal workings or execution paths in the case of software.
- **Block-oriented driver:** A software IO device interface that enables memory blocks to be transferred to and from the IO device, rather than one memory word at a time.
- **Block transfer:** Transfer of data (typically contiguous, but may be a *scatter/gather list*) that includes multiple memory words/bytes on a bus with automatic addressing of each element in the block, rather than addressing and performing a full bus cycle to transfer each word.
- **Blocking:** When a *thread* of execution has been *dispatched* on the CPU for execution, but it needs some other resource, such as memory access, an IO interface, or some other external condition, to be true, such that it must give up the CPU and wait, the *thread* is said to be blocked.
- **Boot code:** Software that is the very first to execute after a processor is reset and hardware sets the *PC* (*program counter*) to an initial address for execution; boot normally completes after initializing fundamental resources, such as memory, cache, and memory-mapped devices, installing inter-

rupt vector handlers, initializing basic critical IO devices, and disabling others, finally loading a higher-level program or *RTOS kernel image* and then jumping to its entry point.

- **Bottom half (device interface):** Software interfacing to IO device hardware that services interrupts related to the device, provides basic configuration and control, monitors status, and buffers IO data; the *top half* makes a bottom half usable for application software.
- **BSP:** Board support package; the *boot* code and basic IO interface initialization code needed by an *RTOS* to *boot* and cycle on an embedded system board.
- **BSS:** Uninitialized global C program data; because the data is not initialized, this data need not take up space in nonvolatile memory, but must be allocated a data segment in working main memory.
- **Bt878:** Brooktree video/audio encoder that can digitize an *NTSC* input.
- **Burst transfer:** A bus transaction that involves an initial address cycle followed by many data read/write cycles terminated by the *bus master* (similar to *block transfer*, but of unlimited length).
- **Bus:** A parallel interface for reading/writing data words from/to addresses and includes digital data lines, address lines, and control lines; note that address and data lines may be multiplexed rather than separate lines.
- **Bus analyzer:** A passive device that snoops on a bus to capture a record of all bus cycles; typically acts like a specialized *logic analyzer* and can be set up to trigger and start collecting a bus cycle trace when a particular address, data, or control bit pattern is active on the bus.
- **Bus master:** A device that can initiate bus cycles to address a target device and then read/write data to the target device, which supplies data or receives data.
- **Byte-oriented driver:** A device interface that provides the ability to read/write single words/bytes to and from the IO device one at a time.
- **C (in RMA):** The execution time required by a *service* to provide a response not including any time spent blocking (only time where the CPU was, in fact, being used to compute a response output).

- **Cache:** High-speed access memory that typically can be read or written in a single CPU cycle, but, due to high cost per storable word, is used as an efficient copy of a much larger main memory device; hardware functionality is typically included to aid with cache memory management, including maintenance of cache/memory coherency, mapping of main memory addresses to cache lines (*direct-mapped*, *set-associative*, *fully associative*), and loading/write-back of data between cache and main memory.
- **Cache coherency:** A cached copy of data at a given address will be different than the data at the same address in main memory after a cached write to this address; when this happens the cache control hardware/software must restore agreement between the data in cache and main memory sometime before data would otherwise be corrupted. Two main policies are used to maintain coherency: *write-back* and *write-through*; however, when memory addresses are cached and also used for DMA or other types of IO, special care must also be taken by application code to ensure that data is not corrupted by intelligently performing write-backs and reloads of cache lines as needed.
- **Cache hit:** When a read or write is performed by an application on data cached at the address accessed/updated, then this is said to be a cache hit.
- **Cache line eviction:** A system event where data is written back to memory, freeing up a cache line.
- **Cache line invalidation:** A system event where a cache line is marked, typically with a status bit called “dirty,” which indicates that the cache line must be reloaded from memory before data is read from it.
- **Cache line locking:** Many caches have control features allowing a program to lock a particular address into a line of cache, preventing this line from being replaced when other addresses are loaded (makes most sense for *set-associative* caches rather than *direct-mapped*); cache line size varies, but is often 16–64 bytes.
- **Cache line pre-fetch:** Many caches have a feature allowing a program to request the cache to load a cache line despite the fact that the associated address has not been accessed yet; the idea is that this address will eventually be accessed in the future, and rather than stalling the *CPU pipeline* at the time it is accessed, intelligent applications can plan ahead.

- **Cache miss:** When a read or write is performed by an application on data that is not presently in cache and therefore the CPU must first load the data at the address being accessed/modified, this is said to be a cache miss.
- **Cache miss penalty:** The number of CPU core cycles that the CPU pipeline must be stalled when a cache line must be loaded after a cache miss in order for a thread of execution to continue.
- **Call-back:** A programming technique where a pointer to a function is passed to a different function (registered) so that the function that obtains this pointer can call the function passed to it by reference later on, a technique commonly used in user interfaces so that an event such as a mouse click can be handled generically by code that, however, will call any number of user application call-backs when the mouse click event is detected.
- **Canonical service:** A coding style template used for a real-time service provided by an *RTOS task* or *thread*. This style may vary, but at a minimum includes a main loop that executes as long as the service is “in service” and has a code section that either polls for input or *synchronously* or *asynchronously* waits for a service request.
- **CCTV:** Closed circuit television; a common term used to describe a camera that outputs an NTSC signal.
- **Ceiling:** The ceiling is a mathematical operation that can be performed on a real number (floating point); the ceiling(*n*) is the closest integer whole number greater than or equal to *n*—for example, ceiling(1.1) = 2 (note that floor(1.0) = ceiling(1.0) assuming that the significance is 1, which is the typical definition of floor and ceiling unless otherwise noted).
- **CFD:** Control flow diagram; a diagram used in structured analysis/design that indicates where control signals in the system originate, where they terminate and how they change the flow of data and/or the processing of data in a *DFD* (note that a CFD is typically a subset of a DFD that shows both data flow and control flow).
- **Chaining interrupt service routine:** A chaining *ISR* is an *ISR* that calls more than one handler for the very same interrupt source and priority, a technique often used in software when a hardware interrupt line

is shared by multiple devices (note that most *chaining ISRs* also perform *ISR polling*).

- **Check-stop:** When an error condition on a CPU that cannot be handled and further execution by the CPU is considered either dangerous or impossible, then the CPU hardware may enter a state known as check-stop, where it ceases to fetch and execute instructions and can leave this state only via a reset—for example, a detectable memory error that cannot be corrected normally causes the CPU to enter check-stop.
- **Circuit switched IO:** An IO channel that is dedicated to one and only one data source and sink; often the channel may be point-to-point, but may be switched before the circuit is established.
- **Cirrus crystal 4281:** An audio encoder/decoder used in ECEN 4623/5623.
- **CLI:** Command line interface, a simple ASCII terminal type interface that can operate over serial or any other byte-stream IO interface to provide the ability to command a device and obtain basic status information.
- **Codec:** Compression and decompression protocol, such as MPEG-4, which is used in streaming to compress video frames prior to transport and to decompress them after transport to be displayed. Video and audio codecs are often implemented in hardware or software and inherently operate on continuous media streams rather than files.
- **Completion test:** This *necessary and sufficient scheduling feasibility test* is based upon the Lehoczky, Sha, and Ding theorem, documented in this book.
- **Computational complexity:** The mathematical magnitude of operations required to successfully execute a given algorithm—for example, searching a data set can take N operations for N items linearly searched or $\log(N)$ operations for a balanced tree search of N items, or even constant C operations for N items with a perfect hashing function.
- **Context switch:** When a CPU is multiplexed (shared) by more than one *thread* of execution and the scheduler provides *preemption*, when the scheduler does preempt a thread in order to *dispatch* another, it must save state information associated with the currently executing thread (e.g., register values, including *PC*) so that this *thread* can later

be *dispatched* again to restore its thread of execution without a state error.

- **Context switch overhead:** The number of machine code instructions (and clock cycles) that an RTOS scheduler must execute to perform a *context switch*.
- **Continuous media:** IO stream that requires isochronal deliver of data between a source and sink—for example, video stream, audio stream, and possibly a telemetry stream.
- **Control flow:** A control flow is a *CFD* unidirectional association between two processes and/or external entities that indicates an asynchronous mechanism used to control a process or data source/sink.
- **Coverage criteria:** When unit tests and/or system tests are completed on software, coverage criteria define the completeness of the testing by specifying the percentage of execution paths, statements, conditions, and decisions that must be covered.
- **CPI:** Cycles per instruction, a measure of CPU efficiency with the ideal that a CPU pipeline should have a CPI of 1.0 or less if the pipeline can retire an instruction every clock; if the pipeline is also *superscalar* such that multiple instruction pipelines may execute, then this type of *micro-parallelism* can theoretically yield a CPI less than 1.0.
- **CPU:** Central processing unit, a processor core providing arithmetic and logic operations, possibly floating point arithmetic, and basic register and memory operations.
- **CPU bound:** When an application program is unable to execute any faster due to the clock rate of the CPU and the CPI.
- **CPU pipeline:** The use of micro-parallelism in the CPU core to provide a stage of instruction processing every clock such that once the parallel pipeline is started, an instruction is completed every clock; stages typically include: fetch, decode, execution, and write-back as a minimum. The key to pipelining is that it is possible for the pipeline to fetch, decode, execute, and write back all at the same time for four instructions at various stages; each instruction will actually take multiple cycles to complete, but in the aggregate one instruction is completed every clock (note that pipelines may also be superscalar such that whole pipelines may be run in parallel as well).

- **Critical instant:** This assumption made by Liu and Layland when they formalized fixed-priority RM describes a worst-case scenario where all services in a system would be released simultaneously.
- **Critical section:** When two independent threads of execution share a resource, such as a shared memory location, the section of code that accesses and possibly updates this shared resource in each thread is called a critical section; to ensure correctness, both threads will employ a synchronization mechanism, such as a *mutex semaphore*, to protect the critical section.
- **Cross compiler:** A compiler that can generate code for a target processor that may be different than the host system that it runs on.
- **Cross debugger:** A debugger that can single-step through code executing on a target processor different than the host system the debugger interface is running on; most often this works with a host debugger that communicates with and controls a target agent debugger.
- **CSMA/CD:** Carrier sense multi-access / collision detection; a protocol used in Ethernet to detect when a node is already transmitting on the shared link and to back off and attempt to use the network later.
- **Cycle-based profiling:** Profiling code executing on a processor by periodically saving off the current PC in a trace buffer, most often implemented by an interrupt-generating counter that counts cycles and can be programmed to raise an interrupt every N cycles; the ISR associated can then service the interrupt and save off the PC each time into a trace buffer.
- **Cyclic executive:** An embedded software architecture that is composed of one or more main loop application(s) and interrupt service routines; the main loop(s) execute on a periodic basis. In some cases the cyclic executive may be an extension of Main+ISR such that several loops run concurrently or are multiplexed on a single CPU and provide different rates of execution—for example, a high-, medium-, and low-frequency executive.
- **D (in RMA):** The deadline for a service that is relative to a request for the service.
- **DAC:** Digital to analog conversion; most often used to provide analog output to an actuator from a digital IO interface—for example, a motor or speaker.

- **Data flow diagram:** A diagram used in structured analysis/design that indicates where data in the system originates, where terminates, and how it is processed in between (note that a DFD typically includes a *CFD* and therefore shows both data flow and control flow).
- **Data segment:** A memory region reserved for global variables and constants in a C program thread; most often each thread has its own data segment (note that most programs include a Stack, Data, and Text segment as a minimum).
- **Datagram transport:** Transmission of packets on a link such that errors in transmission can be detected, but are not automatically corrected nor is there automatic retransmission of lost data; furthermore, there is no concept of a connection (real or virtual) such that multiple messages are unrelated and if fragmented will not be reordered or reassembled automatically.
- **DDR:** Double data rate; a bus data encoding technique where read or write data is transferred on both edges of a reference clock rather than just one (rising edge and falling edge); this doubles the data rate.
- **Dead reckoning:** A technique used in robotics and vehicle navigation whereby a direction or motion or rotation is selected and executed at a constant rate for a calculated period of time in order to produce a desired amount of translation or rotation to reach a target—for example, a vehicle might be pointed north and drive at 5 feet per second for 1 hour in order to get to a target city due north of a starting point south of this target. The major disadvantage of dead reckoning is that there is no mid-course correction possible and overshoot and undershoot are also likely.
- **Deadline:** A time relative to a request for service when the service must be completed to realize full utility of the service.
- **Deadline-monotonic:** A real-time theory directly related to RM, but with a policy such that shortest deadline receives highest priority (rather than shortest period) and a feasibility test based on deadlines rather than periods.
- **Deadlock:** A multithread condition where two or more threads of execution are waiting on resources held by another and the graph of wait-for associations is circular—for example, if A is waiting on resource

R1 to produce resource R2, and B is waiting on resource R2 to produce resource R1, this is a deadlock.

- **Decoder:** A digital device that takes a bit-encoded input and produces an analog actuator output—for example, audio playback decoder that drives a speaker.
- **Delayed task:** The state of a VxWorks task that has been programmed arbitrarily to yield the CPU for a period of time before replacing itself back on the ready queue—for example, `taskDelay` is called.
- **DFD:** Data flow diagram; a diagram used in structured analysis/design that indicates where data in the system originates, how it is processed, and where it terminates (from data source to data sink).
- **Digital control:** Feedback control where the control law is driven by discrete periodic sensor samples and based upon a Z-transform (rather than a Laplace transform in analog control).
- **Direct-mapped cache:** A *cache* memory that has cache lines directly mapped to main memory locations such that a given main memory address can be loaded into one and only one cache line, yet a set/range of main memory locations may be loaded into that particular line.
- **Dispatch:** When an RTOS scheduler selects a thread ready to run, restores state associated with the thread, and transfers execution control back to the thread's last PC if it was preempted earlier (or to its entry point if the thread is ready to run for the first time).
- **DMA:** Direct memory access; a hardware state machine independent of the CPU core that is able to transfer data in or out of memory without directly executing core instructions, thus allowing the core to continue execution while regions of memory are copied, updated by an IO device, or read out to an IO device.
- **DOF:** Degree of freedom; a rotational or translational dimension that a mechanical device can move in—for example, a typical robotic arm has five rotating joints: base, shoulder, elbow, wrist, and gripper; the robot is therefore said to have five degrees of freedom.
- **Double-buffering:** A technique often used in continuous media applications to allow for data acquisition into one buffer while another is being read out and processed; when the acquisition buffer is full, the

buffer pointers are swapped such that the newly acquired data is processed and the already processed buffer can now be used for acquisition.

- **Driver:** A driver is software composed of code that interfaces to a hardware device and provides buffering, control, and status and that also interfaces to RTOS threads/applications and provides controlled access to the hardware device for IO.
- **DSP:** Digital signal processing; a specialized embedded processor core that includes parallel mixed analog and digital processing for typical signal processing functions—for example, for a Fourier transform.
- **Dynamic linking:** A technique where *PIC* software compiled into an object file format, such as *ELF*, can be loaded and linked into existing software on an RTOS platform on the fly after the RTOS has already been booted and is up and running.
- **Dynamic-priority:** When thread or interrupt processing priorities are changed during runtime by code, they are said to be dynamic.
- **Earliest deadline first:** A dynamic-priority scheme for scheduling where services are assigned priority dynamically every time the ready queue is updated, with highest priority given to the service with the earliest impending deadline; the scheme requires not only dynamic-priority but also *preemption* to work.
- **ECC:** Error correction code; a digital logic circuit that automatically corrects an SBE using an error detection and correction encoding, such as the Hamming code; normally the data readout of memory is corrected before the final value is placed in the read buffer, but not necessarily also corrected in main memory. A *write-back* may be required to correct the actual memory location.
- **EDAC:** Error detection and correction; an information encoding scheme that allows for not only detection of errors but also correction of those errors—for example, the Hamming code.
- **EDTV:** Enhanced definition television.
- **EEPROM:** Electrically erasable programmable read-only memory; a nonvolatile memory device that can be erased and rewritten in circuit if so desired.

- **EFSM:** Extended finite state machine; a formal method based upon state machines that extends the basic state transition on IO to include side effects on transitions, such as global data update and data processing.
- **ELF:** Executable and linking format; an object file format that includes significant annotation and is PIC such that these files can be dynamically loaded and linked and such that they can serve for supporting debug and trace analysis to map addresses back to source code.
- **Embedded system:** A digital and analog computer system that provides a specific set of services, driven by sensor inputs, and producing sensor outputs to provide services—for example, digital control in an antilock braking system or call switching and billing management for a telecommunications main trunk (note that the scale of the services provided and of the hardware itself does not matter).
- **Encoder:** A circuit that takes analog signal inputs and, using an ADC, converts them to digital and bit-encodes them—for example, an audio recorder that takes analog microphone input and encodes the input signal into 255 different tones.
- **Entry point:** An address in a *text segment* that is the first instruction in a function and serves as the starting point for a thread such that a scheduler can simply set the *PC* to this address in order to start execution of this thread.
- **EPROM:** Erasable programmable read-only memory; a nonvolatile memory device that typically can be erased by a UV light source and electrically reprogrammed, but not in circuit, rather by pulling the device from a socket, exposing it to UV, and then placing it in an external programmer.
- **Event-based profiling:** A profiling technique where the *PC* is saved into a trace buffer whenever events of a specific type exceed a threshold—for example, when data cache misses exceed *N* misses, the *PC* is saved into a trace buffer.
- **Exception (NMI):** An exception is normally a non-maskable interrupt because it signifies a serious error condition that must be handled before any program should continue execution—for example, a bus error.

- **Execution jitter:** When a service is dispatched and the number of cycles and/or instructions required to complete the service varies on each release, this service is said to have execution jitter.
- **Extended finite state machine:** An FSM (finite state machine) with more features than just states and IO transitions so that the Von Neumann architecture and general programs may be modeled formally—for example, on a state transition a procedural function may be called and or global memory updated.
- **External fragmentation:** When blocks of a resource, such as memory, are allowed to be arbitrarily sized, small sections of the resource between used sections may evolve from successive allocations and frees such that significant resource exists, but is unusable unless allocations are moved to provide larger contiguous free spaces from small many non-contiguous spaces (fragmentation outside of blocks).
- **FCFS:** First come, first served; the policy often used by an RTOS when services/threads are at the same priority level—that is, the first service ready is the first one dispatched.
- **Feasibility test:** An algorithmic or formulaic operation that takes a set of services and their *RM* characteristics and will provide a binary output indicating whether this service set can be scheduled given resources available and resources required by the service set.
- **FEC:** Forward error correction; an EDAC method provided in-line such that bit errors are handled at the link layer—for example, Reed Solomon encoding (in contrast to EDAC memory).
- **Feedback:** A signal used in control systems that provides sensor inputs to compute the difference between desired and actual plant state such that a control law can drive the plant to a desired target control point.
- **FIFO:** First in, first out; a policy for queues (e.g., a dispatch queue) where the first element queued is always the first element de-queued.
- **Firmware:** The first code to execute on a processor and therefore must initially execute out of an *NV-RAM* device, although it may load itself into memory and continue execution to complete a boot process before an RTOS is initialized and run. Less specifically, firmware is usually thought of as any software that directly interfaces to hardware to make the hardware usable by higher levels of software.

- **Fixed-priority scheduling:** A scheduling policy whereby threads on the ready queue are dispatched in priority order and the priority of any given thread is not modified over time (except by the application itself).
- **Flash memory:** A nonvolatile memory technology that can be erased and reprogrammed in circuit like EEPROM, but has much higher density for a given cost.
- **Floor:** The floor is a mathematical operation that can be performed on a real number (floating point); the $\text{floor}(n)$ is the closest integer whole number less than or equal to n —for example, $\text{floor}(1.1) = 1$ (note that $\text{floor}(1.0) = \text{ceiling}(1.0)$ assuming that the significance is 1, which is the typical definition of floor and ceiling unless otherwise noted).
- **Flow control:** Signals between a data transmitter and receiver used to indicate buffer capabilities on each side so that a transmitter does not overdrive a receiver, resulting in data loss when the receiver is unable to buffer incoming data.
- **Form factor:** The physical dimensions of an electronic device that may be independent of the electrical characteristics—for example, the PCI bus electrical specification and protocol is implemented as compact PCI, stackable PC/104+, and PMC (PCI Mezzanine).
- **FPGA:** Field programmable gate array; an array of generic transistors that can be programmed once or on power-up to provide combinational logic and state machines for digital processing.
- **Fully associative cache:** A *cache* that allows main memory addresses to be loaded to any cache line; this is the ideal cache since replacement is not constrained at all, but associative memory is complex and expensive. By comparison a *direct-mapped cache* is completely constrained and a *set-associative* is a compromise.
- **Gather read list:** A list of not necessarily contiguous addresses in memory that are to be read into a contiguous buffer—for example, a host memory may have multiple blocks in memory scattered through memory space that are to be read by an IO device that will gather all these blocks into a single contiguous buffer before an IO operation.
- **GPIO:** General-purpose IO; digital inputs and outputs at TTL logic levels that can be used as a generic interface to digital devices, such as LEDs.

- **Hamming code:** A bit encoding used to detect and correct *SBEs* (single bit errors) and to detect *MBEs* (multi-bit errors) for memory devices that may be subject to *SEUs*.
- **Hard real-time:** A service or set of services that are required to meet their deadlines relative to request frequency; if such deadlines are missed, there is not only no utility in continuing the service, but in fact the consequences to the system are considered fatal or critical.
- **Harmonic:** When the relative periods of services are all common multiples of each other; this characteristic can yield cases where the CPU resource can be deterministically scheduled to full utility.
- **Harvard architecture:** A core CPU architecture that splits the memory hierarchy into separate instruction and data streams, typically including an L1 instruction cache that is independent of an L1 data cache.
- **HDTV:** High-definition television.
- **Heap:** A memory space used for dynamic buffer management and/or dynamic allocation of memory as requested by an application; heap space is memory outside the data, text, and stack segments and is most often reserved by the boot or RTOS during initialization.
- **High availability:** A system that guarantees that it will be ready to provide services with a quantifiable reliability—for example, a system is said to provide five nines availability if it is ready to provide service upon request 99.999% of a given year (i.e., is unavailable only for a total of about 5 minutes per year). Note that HA systems can crash, but they can't be out of service very long if they do.
- **High reliability:** A system that has been designed to have a very low probability of failure to provide services; typically measures such as redundancy, cross strapping, and fail-safe modes are designed in to ensure that critical services have an extremely low likelihood of failure.
- **Highest locker protocol:** See priority ceiling emulation; this is the same basic mechanism to avoid unbounded or lengthy priority inversions possible with mutual exclusion.
- **Host:** Desktop development computing system used in *IDE* for cross compilation, cross debugging, connection to the target agent, trace

tools, and any number of other tools that connect to a target server on the host to communicate with *target agent* software resident on the embedded system.

- **H-reset:** Hardware reset; either from a power-on reset state transition or from assertion of an external signal to drive a hardware reset.
- **HSTL:** High-speed transceiver logic; a 0.0-1.5v logic level standard used for high-speed single-ended digital IO, most often for memory IO (speeds of 180 MhZ and greater).
- **HWIC:** Hardware in circuit; a concept whereby debug and trace tools have hardware probes in circuit with a CPU by interfacing to signals coming from the CPU/SoC ASIC to the rest of the system board for the purpose of snoop tracing and/or control—for example, JTAG debug emulator, Vision ICE Event Trace, RISCWatch trace port probe, and CodeTEST Universal trace probe.
- **ICE:** In-circuit emulator; a debug and trace device that monitors all IO pins on a CPU/SoC ASIC, provides memory trace, external interrupt trace, JTAG, and IO pin trace, and emulates the state of the system, including all registers, cache, and addressing, to aid in firmware development and board verification.
- **IDE:** Integrated development environment; a software development system that for an embedded system includes a cross and native compiler, cross and native debugger, and many target tools interfaced through a host-based target server and a target-based target agent.
- **Importance:** In real-time systems theory services with low priority based upon *RM* policy may still be critical to system operation; they are important despite being low-priority.
- **Interference:** When a higher-priority thread preempts a lower one in a fixed-priority preemptive system the time that the CPU is unavailable to lower-priority threads is referred to as interference time.
- **Internal fragmentation:** When a resource such as memory is made available in minimum-sized blocks, this can help reduce *external fragmentation*, but when a user of the resource requires less than a full block, this causes internal fragmentation.

- **Interrupt handler:** During the normal CPU pipeline processing (fetch, decode, execute, write-back) an external device may assert an signal input or an internal sub-block may also assert a signal input to the CPU core that causes it to asynchronously branch to an interrupt vector (a memory location) where basic code called the handler acknowledges and services the hardware and then calls application *ISRs*.
- **Interrupt latency:** The delay between assertion of an interrupt signal by a device and the time at which the PC is vectored to an interrupt handler is known as the interrupt latency.
- **Interrupt vector:** An address in memory where the CPU sets the PC after an interrupt signal is asserted, causing the CPU to asynchronously branch to this location and to execute the instruction there; normally a CPU will have a number of interrupt inputs (e.g., x86 IRQ0-15), and each signal asserted causes the CPU to vector to a different address such that different handlers can be associated with each interrupt signal.
- **Interval timer:** A double-buffered state machine in a CPU core that allows software to set a value in a register that is loaded into a separate countdown register that asserts an interrupt at zero (or perhaps all Fs if it counts up) and automatically is reloaded with the interval register value to repeat the process over and over; this hardware can therefore be used for basic timer services in an RTOS.
- **IO bound:** A condition where an application does not have sufficient IO bandwidth to meet throughput goals or real-time deadlines.
- **ISA legacy interrupt:** Industry standard architecture legacy interrupt; specifically refers to x86 architecture IRQ0-15, which has been part of the x86 architecture from the beginning (8086) and support a number of well-known PC devices and services, such as booting from a hard drive.
- **Isochronal:** Literally the same in time, which in real-time systems means that a service is required to produce a response at a precise time relative to a service request—not too early and not too late. This is important to continuous media applications and digital control that are sensitive to *jitter*. Most often isochronal services hold a response computed ahead of deadline that is delivered to an interface within a narrow band around the optimal time.

- **ISR:** Interrupt service routine; the application level of an interrupt handler that is often a call-back function registered with an RTOS that installs the *interrupt handler* at an *interrupt vector*.
- **Jiffy:** A Linux term for the tick of an interval timer, the smallest unit of time that the OS can track—for example, on x86 architecture the standard interval timer ticks about every 0.45 microseconds, but the Linux OS typically loads an interval timer count such that it can control processes on a 10-millisecond software tick.
- **Jitter:** When latency and/or timing of an operation or process changes with each iteration, this is jitter—that is, when latency/timing is not constant. Jitter as a term can be used to describe many different types of operations or processes—for example, execution jitter, period jitter, and response jitter.
- **JTAG:** Joint Test Applications Group; an IEEE committee that standardized the concept of boundary scan and the *TAP* (test access port), which is used to verify integration of ASICS in a system (boundary scan), but is now also typically used in firmware development to control and single-step a CPU by loading data and commands through the *TAP* with JTAG. JTAG includes the following signals: TDI (Test Data In), TDO (Test Data Out), TRST (Test Reset), Clock, and Test Mode Set.
- **Keep-alive:** An indication from a thread/process/task on a system that it is functioning normally or perhaps similar indication from a subsystem in a larger system; the keep-alive is most often a simple ID and count indicating that the subsystem/thread/process/task is advancing through its service loop, often referred to as a heartbeat as well.
- **Kernel:** The software in an RTOS that directly controls all critical resources, such as CPU, memory, and device IO; the kernel is typically interfaced to by applications through an *API* or *device driver*.
- **Kernel image:** The binary machine code text segment, data segment, stack, and BSS used for the RTOS kernel software.
- **Kernel instrumentation:** Tools like WindView and CodeTest, which provide active tracing of C code and/or RTOS events require that code, often specifically kernel code, be instrumented with trace instructions that provide efficient update of a trace buffer with a trace token to track progress of the code and to mark events for later timeline analysis.

- **L1 cache:** Level-1 cache; a high-speed memory integrated on-chip with a CPU core, on the same ASIC for data access that can most often be completed in a single clock.
- **L2 cache:** Level-2 cache; a high-speed memory off-chip that can be accessed in several clocks.
- **Latch-up:** A non-recoverable bit error due to permanent transistor logic damage to a memory device or register.
- **Latency:** Delay in an operation or process due to physical limitations, such as electronic propagation delay, the speed of light, the number of clock cycles required to execute instructions, or time to modify a physical memory device.
- **Laxity:** $\text{Laxity} = (\text{Time_to_Deadline} - \text{Time_to_Completion})$, but the time to the completion of a service can be difficult to determine, so most often an estimate of the $\text{Time_to_Completion}$ is used, which is derived from $(\text{WCET} - \text{Computation_Time_So_far})$.
- **Layered driver:** A layered driver includes a *top half* and *bottom half*; the *top half* provides an interface to application code wishing to use a hardware resource, and the *bottom half* provides an interface to a hardware device.
- **LCM:** Least common multiple; the LCM is the smallest number that is also multiple of two different numbers—for example, given $x = 3$, $y = 5$, the $\text{LCM}(x,y) = 15$. This concept is key to periodic service analysis in real-time theory because it is necessary to diagram service times over the LCM of all periods in order to fully analyze timing demands upon a resource.
- **Least laxity first (LLF):** A dynamic-priority policy where services on the ready queue are assigned higher priority if their laxity is the least (where laxity is the time difference between their deadline and remaining computation time); this requires the scheduler to know all outstanding service request times, their deadlines, the current time, and remaining computation time for all services, and to re-assign priorities to all services on every preemption. Estimating remaining computation time for each service can be difficult and typically requires a worst-case approximation.

- **LED:** Light emitting diode; a device typically used to provide visual IO and status for an embedded system.
- **Lehoczky, Sha, and Ding theorem:** If a set of services can be scheduled over the period of the longest period service after a *critical instant*, then the system is feasible (i.e., is guaranteed not to miss a deadline in the future).
- **Limit sensor:** A sensor that detects when hardware has reached a physical limit—for example, when a robotic arm has driven a joint through full rotation after which continued motor drive will break the joint.
- **Linking (dynamic or static):** Linking is the process by which an executable image is assigned addresses for all function entry points, all global variables, and all constants that may be referenced by other software modules; these addresses can be statically assigned once and for all at a pre-determined offset in physical memory (static linking) or may be position-independent such that only relative addresses are assigned until the module is loaded, at which time physical addresses are derived from the relative (dynamic linking).
- **Livelock:** Related to deadlock, this situation arises when a circular wait for resources evolves and an attempt to break the deadlock is made by having each requester drop their requests and then re-request them; if the requests are well synchronized, then the system may cycle between deadlock and dropping requests over and over.
- **Logic analyzer:** A hardware, firmware, and software analysis tool that provides generic acquisition of digitally clocked signals (or arbitrary digital signals that are clocked by the analyzer internally).
- **LSP:** Linux support package; an embedded Linux term, much like a BSP, that refers to code required to boot Linux on a given architecture and platform—for example, the PowerPC 750 LSP.
- **LVDS:** Low-voltage differential serial; an electrical standard for transmission of high-rate serial signals on wire pairs that carry differential signals to encode data.
- **Main+ISR:** This is essentially the same software architecture as a *cyclic executive*; however, Main+ISR may be much simpler in that it normally has just one main loop and a small number of ISRs compared

to a *cyclic executive*, which may have multiple loops operating at different frequencies.

- **MBE:** Multi-bit error; a condition when more than one bit in a word is in error. Typically this cannot be corrected.
- **Memory hierarchy:** The whole memory system design from the fastest and typically smallest devices to the slowest and typically largest devices—for example, L1/L2 cache, main memory, and flash.
- **Memory-mapped IO:** IO devices that can be read or written can be mapped into the address space of a processor, allowing software to simply update an address in order to write to the device or read an address to read from the device; the device must respond to the addressing by the CPU—that is, decode it and then read/write data on a bus that both the device and CPU interface with.
- **Memory protection:** An *MMU* feature that allows address ranges on page boundaries (a minimum-size memory block) to be specified as read-only; if an update to such a range is attempted, the *MMU* will assert an *NMI* exception.
- **Message queue:** An RTOS software mechanism that abstracts shared memory data into atomic enqueue and de-queue operations on a buffer controlled by the RTOS and known only to applications by an ID, accessible to them only through RTOS message queue operations. Operations are atomic with respect to threads only (not interrupts), and so most often only a message queue send is allowed in interrupt context, never a message queue receive.
- **Message sequence chart:** A diagramming method used in the Specification and Design Language (SDL) as well as UML (Universal Modeling Language) that shows threads of execution and all messages (or function call interfaces) that associate the threads in a protocol.
- **Microcode:** Machine code that executes on a state machine internal to a processor or on a simple state machine device that is independent of the main execution pipeline—for example, the Bt878 RISC processor executes code fetched from the x86 processor's memory over the PCI bus; this code is microcode from the viewpoint of the x86 system.
- **Micro-parallelism:** Parallel processing inside the CPU core.

- **MMU:** Memory management unit; a block in most CPU cores that provides virtual to physical address mapping and address range checking, and can protect read-only address ranges from unintentional update.
- **Module loading:** When an *ELF* module is transferred to an embedded target and dynamically linked into the kernel and other application code on the fly.
- **MTD:** Mapping to device; a term used to describe bottom half code used in a flash file system driver.
- **Multi-access network:** A network such as Ethernet where more than one device can use the physical and link layer of the network, thus requiring a *CSMA/CD* protocol for shared use.
- **Multitasking:** When a CPU is shared and multiplexed by a scheduler in order that multiple threads with state information may execute on a single CPU or may be mapped onto a set of CPUs dynamically. Tasks include state information that goes beyond the minimal requirements of register state, stack, and PC for a thread—for example, task variables, a task error indicator, name, and many other elements of a VxWorks *TCB*.
- **Multithreaded:** When a CPU is shared and multiplexed by a scheduler with the minimal management of execution state for each thread of execution (register state, stack, and PC).
- **Mutex semaphore:** A specialized semaphore (compared to a binary semaphore) that is specifically used to protect critical sections of code for multithread safety; this semaphore is used to guarantee mutually exclusive access to a shared resource such that only one thread may access a common resource at a time. With shared memory this prevents data corruption that could be caused by multiple readers/writers—for example, if a writer has partially updated a shared data structure, is preempted/interrupted, and then a reader accessed the partially updated data, the data may be completely inconsistent.
- **Nand flash:** A flash memory device that is normally erased to all F's and writes are bitwise masked in with an and operation.
- **NCD SCAM chip:** A pre-burned microchip PIC that includes code to generate PWM signals for hobby servos (2 channels) based upon an RS-232 command.

- **Necessary and sufficient:** A feasibility test in real-time theory that will pass all service sets that can be scheduled and will never fail a set that can be scheduled (more precise than a sufficient test that may falsely reject some service sets, but will never falsely okay a service set than cannot be safely scheduled).
- **Nesting:** When a construct is used inside the same sort of construct, one inside the other—for example, if a critical section encloses another critical section, the critical sections are said to be nested.
- **Non-blocking:** When a request for a resource cannot be met immediately, the RTOS can either block the calling thread until it is available or return it an error code, indicating why the request cannot be met and letting the thread go on; the latter is non-blocking.
- **Nor flash:** A flash memory device that is normally erased to all zeroes and writes are bitwise masked in with an or operation.
- **NTSC:** National Television Systems Council; the standard for analog color television transmission with 640x480 pixels. The standard defines up to 525 scan lines, but only 480 are used for an NTSC display frame, with the remaining used for sync, vertical retrace, and closed caption data (sometimes referred to as vertical blanking lines). Most NTSC cameras produce 492 scan lines. The NTSC signal interlaces scan lines, drawing odd-numbered scan lines in odd-numbered fields and even-numbered scan lines in even-numbered fields to produce a non-flickering image at approximately 60 Hz refresh frequency (the frequency was adjusted for color to 59.94 Hz). This yields approximately 30 interlaced video frames per second. For color, NTSC carries luminance and chrominance signals where luminance carries the normal monochromatic NTSC signal. The luminance and chrominance can be combined on a composite cable or carried separately as they are with S-video cables. The chrominance signal is carried at 3.579545 MHz and can either be ignored or recovered, using a color burst reference signal that is output on the back porch of each horizontal scan line. Separating luminance and chrominance from a composite output (most often found on CCTV cameras) into S-video requires a filtering/splitting cable. Passive splitters can work, but sometimes the chrominance signal can be lost, depending upon the quality of the camera's composite output. Due to some of the quality issues with the color subcarrier scheme, NTSC is sometimes referred to as "Never Twice the Same Color." Newer

higher-definition standards for television have since emerged including EDTV and HDTV, but for embedded computer vision projects, NTSC is still an affordable and readily available camera and frame capture standard that is in wide usage.

- **NV-RAM:** Nonvolatile random access memory; memory that persistently holds data regardless of whether a system is powered—for example, a battery backed-up *DRAM*, a *Flash memory* device, *EEPROM*, or *EPROM*.
- **Object code:** Machine code annotated with symbol information (variable and function names and addresses) and information to support debugging (source file names and locations).
- **OCD:** On-chip debugging; a type of JTAG front end that allows a typical line debugger to single-step code through the JTAG protocol.
- **Offloading:** The concept of taking a software service and re-allocating it to a hardware implementation on a parallel processing unit in order to free the main CPU of loading—for example, a network interface card may perform functions basic to TCP/IP, such as checksums, in order to offload those operations from the host CPU.
- **OnCE:** On-chip emulation; a type of JTAG front end that allows for not only debug through JTAG but also additional control, such as register viewing and setting.
- **Online admission:** When a system can run a feasibility test while currently providing other services in order to determine whether new services can safely be added to the current safe set.
- **On-off control:** The use of relays to turn on and off motors to control a mechanical device.
- **Optical navigation:** Using computer vision images of a scene to determine ranges to targets and to plan paths to navigate to a target using only video data.
- **Optimal policy:** A fixed-priority assignment policy that will successfully schedule any set of services that can be scheduled by any other fixed-priority policy.
- **Overrun policy:** How a system handles a service that attempts to continue execution beyond its advertised deadline—for example, the scheduler could terminate the service.

- **Packet switched:** A network protocol that allows links to be shared by multiple datagram or virtual circuit protocols and routes packets between end points based upon their header information.
- **PC:** The program counter is normally a register used by a CPU to track the current or next address of main memory that contains a machine instruction to execute (note that a trace of the PC over time provides the definition of the *thread of execution* until a *context switch* occurs, if it does at all).
- **PCI:** Peripheral component interconnect; see Shanley, p. 771.
- **PCI bus probing:** A process that allows a BIOS or OS software to find all PCI devices and functions on a given PCI bus using configuration space registers.
- **PCI configuration space:** A well-known port address on x86 architecture where a PCI bus master can read/write registers in order to find other PCI devices and their functions and configure them as far as memory mapping and interrupts as a minimum.
- **PCI Express:** Previously known as 3GIO, this standard is a scalable differential serial bus architecture for 2.5 Gbps mainboard interconnection and peripheral connection.
- **PCI interrupt routing:** PCI interrupts A-D can be routed onto x86 legacy interrupts IRQ0-15 in order to allow PCI devices to interrupt an x86 core.
- **Peak-up:** A computer vision algorithm that finds a bright spot or the center of an object by segmenting an image and finding the centroid of a target within the image.
- **Pending task:** A VxWorks task state that indicates the task is blocking on a resource not presently available.
- **Period jitter:** When the period of a service request is not constant.
- **Period transform:** A real-time theory adjustment to a services characteristics to simplify analysis or to elevate importance of a service whereby the services period is assumed to be shorter than it really is.
- **Pessimistic assumption:** RM is full of assumptions that are worst-case and therefore make it a very safe form of analysis, but also may

lead to excessive resource margin in order to guarantee deadlines—for example, *WCET*.

- **PID controller:** Proportional integral differential controller, a controller that sets outputs proportional to error, integrates sensor inputs to find, for example, velocity from acceleration, and also uses derivatives, such as velocity from position measurements, in order to control a system and obtain a target operational state—for example, a cruise control provides acceleration proportional to the difference between current and target speed and integrates to determine when the target will be achieved and when to decelerate.
- **Pipeline hazard:** A condition in a CPU pipeline that forces it to stall—for example, a cache miss.
- **Pipeline stall:** When a CPU pipeline must stop until a resource is made available.
- **Pixel:** A picture element; an array of picture elements forms an NxM image where each pixel encodes the XY position, brightness, and RGB color mix for the picture element in the image.
- **Point-to-point:** A network topology that connects nodes one-to-one.
- **Polling:** When status is checked periodically (synchronously) by a looping construct.
- **Polling interrupt service routine:** An *ISR* that must determine the source of an interrupt by reading status registers when a hardware interrupt is shared by multiple devices (note that most polling *ISRs* also provide *ISR chaining*).
- **Position-independent code:** Code that is base address-independent such that it can be mapped in at any base address and all other entry points, jumps, and memory locations are set relative to the dynamically determined base address.
- **POSIX:** Portable operating systems interface; a standard for operating system mechanisms and APIs. POSIX includes a number of standards, such as 1003.1b, which covers basic real-time mechanisms.
- **Power-on reset:** A CPU state after initial power-on, which most often causes the CPU to branch to a known address and perform basic operations, like resetting the memory controller, bus, and other basic interfaces.

- **Preemption:** When the current thread executing on a CPU is placed back on the ready queue by the scheduler and state information saved so that a different thread can be allocated the CPU.
- **Priority:** An encoding that controls the order of dispatch for threads by a scheduler when more than one is ready to use the CPU resource.
- **Priority ceiling and priority ceiling emulation:** A priority is defined that is the highest priority a thread can have that may lock a resource; this priority level is stored as the resource's priority ceiling. A thread that has locked the resource is given priority as high as the highest-priority thread blocking on the resource up to the ceiling value for the ideal priority ceiling, but for priority ceiling emulation it will be set to a ceiling value specified by the programmer at time of critical section construction—that is, the thread holding the resource always has a priority higher than or equal to all threads waiting to obtain the resource, but amplification is limited to the ceiling value.
- **Priority inheritance:** See [Briand99], p. 66; if a thread is holding a resource and another thread of higher priority is blocking on the same resource, the thread holding the resource inherits the blocked threads priority for the duration of the *critical section*. There is no limit on the priority level that may be inherited.
- **Priority inversion (unbounded):** Whenever a thread is unable to obtain the CPU and a thread of lower priority is holding it, this is called priority inversion. The condition is most often caused by a secondary resource needed by a thread, such as a shared memory critical section; in a simple two-thread case, if a lower-priority thread is in a critical section, then a higher-priority thread experiences priority inversion for the duration of the critical section; however, if the low-priority thread suffers interference from a medium-priority thread, the high-priority thread could potentially be blocked for an indeterminate amount of time, an unbounded priority inversion.
- **Priority-preemptive run-to-completion:** A scheduling mechanism that dispatches any thread ready to run based on priority as soon as the set of ready threads is updated (preemptive) and allows a dispatched thread to run indefinitely unless another higher-priority thread is added to the ready set (via an interrupt or a call to the RTOS by the currently running thread). One danger of this type of system is that a high-priority non-terminating thread will take over the CPU resource completely.

- **Priority queue:** A mechanism for implementing a first-in-first-out policy, but with N levels of priority such that all items at the highest priority level are de-queued first-in-first-out before all items at the next lower priority level.
- **Process:** A thread of execution with stack, register state, and PC state along with significant additional software state, such as copies of all IO descriptors (much more than a task TCB for example), including a protected memory data segment (protected from writes by other processes).
- **Programmed IO:** A technique where software reads and writes each word to and from a device interface involving the CPU in each and every transfer.
- **Protocol stack:** A layered driver that includes data processing between the *bottom half* and *top half* layers; each layer can be separated and has a distinct interface—for example, TCP/IP.
- **PWM:** Pulse width modulation; a technique to control a motor or other normally analog device by creating a pulse train of digital TTL output to simulate an analog output.
- **Quality of service:** Definition of service levels based upon guarantees of resource availability for each service—for example, processor capacity can be reserved for each service in advance (say, 10%) and the system guarantees that this capacity will be available within in a worst-case period of time, however may not guarantee all services will meet their deadlines.
- **Rate-monotonic:** A hard real-time theory for fixed priority-preemptive run-to-completion systems where priority is assigned according to service request period (higher priority for shorter period) and where feasibility of a set of services can be determined by the RM least upper bound or an iterative test, such as the completion test.
- **Reachability space:** The points in space where a robotic device can place and end effector—for example, places in space where a robotic arm can grapple an object.
- **Ready:** The VxWorks task state where a task is ready to running and waiting only on the CPU to be granted by the scheduler.

- **Real-time (system):** A system driven by external events (typically sensors that provide input through *ADCs*) for which services provide computation to produce a response (typically actuators that are interfaced to *DACs*) before a deadline relative to the event-driven request for service; a hard real-time system must never miss a deadline, but a soft or best-effort system may.
- **Real-time clock:** A hardware clock circuit that maintains an absolute date and time (e.g., Gregorian or Julian date), often employing a battery-backed clock circuit and/or a method to synchronize with an external time source, such as Universal Coordinated Time.
- **Real-time correctness:** A real-time service must produce functionally correct outputs and also provide the outputs prior to a relative deadline to be real-time correct.
- **Reboot:** When a system is commanded or as a part of a recovery mode reenters the boot code entry point causing re-initialization of memory, IO interfaces, and restart of all services.
- **Recovery:** A key feature of a high-availability system, this is the mechanism by which a system that is experiencing system failures restarts those services. A system may need to start a recovery process for a number of reasons—for example, deadlock, priority inversion, livelock, and resource exhaustion. Often recovery is achieved by the hardware *watchdog* that reboots the system.
- **Regression test:** Rerunning a test to verify that features previously verified still work after bug fixes or feature addition, intended to prevent unintentional interactions between software modifications that might introduce new problems.
- **Relay:** A mechanical or solid state device that provides a simple switch—for example, double pole double throw or single pole single throw.
- **Reliable transport:** A data transport protocol on a network that includes error detection/correction and retransmission and supports diverse routing such that overall data is delivered if at all possible.
- **Resource arbiter:** A subsystem that implements a resource grant policy—for example, a bus arbiter coordinates bus grants for bus requests from multiple masters and targets.

- **Response time:** The latency between a request for service (typically by an ISR) and the generation of a response output.
- **Ring buffer:** A data structure that provides multiple serially reusable buffers, most often used to buffer incoming data from a device interface before it can be processed, likewise for output data before it can be transmitted.
- **RISC:** Reduced instruction set computer.
- **RM:** Rate-monotonic; the basic theory formulated by Liu and Layland for fixed-priority multiplexing of a single CPU that is intended to provide multiple services over time.
- **RM least upper bound:**

$$U = \sum_{i=1}^m (C_i / T_i) \leq m(2^{1/m} - 1)$$

- **RM policy:** Services with shorter period are assigned higher priority.
- **RMA:** Rate-monotonic analysis; the process of analyzing the C , T , and D characteristics of a set of *services* to be executed on a CPU and determination of priorities according to *RM policy* and *feasibility* according to a *sufficient* or, better yet, *necessary and sufficient* test.
- **ROM based:** A boot or kernel image that is PIC and initially runs out of a nonvolatile device, but tests and initializes memory and then copies itself to working memory and continues execution there.
- **ROM resident:** A boot or kernel image that executes out of nonvolatile memory and sets up a data and stack segment in working memory, but the text segment remains always in the nonvolatile memory.
- **Round robin:** A best-effort scheme with preemptive time-slicing where the scheduler assigns threads a slice in a fair fashion where all ready threads are given a slice of CPU and put back on the end of the queue if needed.
- **Sanity monitor (software):** A service that periodically resets the hardware watchdog timer and also monitors keep-alive messages from other critical services in the system; if a critical service fails to post a keep-alive, then the sanity monitor provides error handling and attempts to recover that service. If the sanity monitor itself fails to

function, then the hardware watchdog timer will time out and the whole system will reboot and start a system-level recovery process.

- **SBE:** Single bit error; when an SEU causes a bit flip or other form of unintended bit flip occurs in a memory word.
- **Scatter write list:** A list of not necessarily contiguous addresses in memory that are to be written from a contiguous buffer—for example, a host memory may have multiple blocks in memory scattered through memory space that are updated by an IO device that contains all of the data to be updated in a single contiguous buffer.
- **Scheduling point:** A necessary and sufficient test based upon the Sha, Lehoczky, and Ding theorem that determines whether all services can be scheduled within the longest period.
- **SDRAM:** Synchronous dynamic random access memory.
- **Semaphore:** An RTOS mechanism that can be used for synchronization of otherwise asynchronous tasks in order to coordinate resource usage, such as shared memory, or to simply indicate a condition, such as data is available on an interface.
- **Semaphore give:** A semaphore operation that allows a thread to indicate that a resource is available; if another thread is blocking on this resource, then this will unblock that thread.
- **Semaphore take:** A semaphore operation that allows a thread to check if a resource is available; if not, the RTOS can either block the calling thread until it is, or simply return an error code.
- **Sensor:** A transducer device that indicates physical status of a system or the environment in which it operates with an electrical signal to encode the system/environment characteristic it is designed to measure—for example, a thermistor, a position encoder, limit switch, stress/strain gauge, and pressure transducer.
- **Service:** A specific computation provided based on inputs that produces a required output in order to meet a system requirement.
- **Service release:** When an external event sensed by an embedded system indicates a request for service, the thread that provides the service is released—for example, an *ISR* can do a semaphore give to indicate sensor data is available for processing.

- **Set-associative cache:** A *cache* that allows main memory addresses to be loaded in N different cache lines; a set-associative cache is said to be N ways, where each way is a different cache line that the same address data may be loaded—for example, 32-way set-associative cache.
- **SEU:** Single event upset; a phenomena where a memory bit is flipped due to an environmental influence, such as electromagnetic radiation. The bit's original value may be restored if the SEU can be detected and corrected by a system monitoring technique.
- **Shared interrupt:** When an interrupt can be asserted by multiple devices, it is shared and requires the interrupt handler to poll status—that is, the handler must read the status of every device that may have asserted the interrupt to figure out which device in fact did.
- **Shared memory:** When more than one thread on a single CPU or on multiple CPUs can access the same memory locations, this memory is shared, and shared memory must be protected by a synchronization mechanism if reads and writes are allowed.
- **Signal (software):** A software signal is often also called a software interrupt and in fact functions much like a hardware interrupt does but at the scheduler/thread level; when a signal is thrown by one thread to another, the throw call causes the RTOS to potentially dispatch the catching thread's handler instead of the code it is currently executing after the catch kernel code is executed. So, a signal can be used to asynchronously interrupt a running thread.
- **Signal block diagram:** A systems design method used in SDL (Specification and Description Language) where hardware and software elements can be modeled as blocks with signal list inputs and outputs; inside the blocks at a lower level, all signals are ultimately consumed or generated by *EFSMs*.
- **Signal catch:** When a signal is received by a thread by the RTOS scheduler on behalf of the thread; the catch modifies the catching thread's state such that the PC, registers, and stack are saved, and when the thread is dispatched next, the scheduler dispatches the threads registered signal handler rather than where it was last preempted.
- **Signal throw:** When a thread wants to asynchronously interrupt the normal flow of execution of another thread, it can call an RTOS

mechanism to throw a signal to the other thread, instructing the RTOS to dispatch the other thread's signal handler rather than its last context.

- **Slack time:** On a real-time system, when no real-time services are requesting CPU time (i.e., waiting on the ready queue or actively running), this unused CPU time is called slack time and often can be used for non-real-time best-effort processing. Slack time is often created by service releases where the actual execution time taken is much shorter than WCET due to *execution jitter*.
- **SoC:** System-on-chip; an ASIC that includes one or more CPU cores, a bus, and IO interfaces such that it essentially places devices previously on a board in earlier products on a single ASIC.
- **Soft real-time:** When a service can occasionally miss a deadline and overrun it or terminate and drop a service release without system failure, these services are considered soft—for example, a video encoder compression and transport service might occasionally drop a frame when compression takes too long; as long as the video stream is not critical and an occasional dropout is acceptable regarding system requirements, this service can be considered a soft real-time service.
- **Software profiling:** Periodically tracing the cycle count and the current PC or actively tracing it by instrumenting all function entry and exit points to save cycle count to a trace buffer allows for determination of where most of the execution time is spent and how many cycles basic blocks of code such as functions require; this type of tracing provides a profile of the software. Profiles can be at a function level, basic code block level (bounded by branches), or a C statement level; overhead is higher for lower-level profiling.
- **Software sanity:** Software is said to be sane when embedded services check in with a sanity monitor by posting a keep-alive; the sanity monitor itself is known to be sane (functioning correctly) if it resets the hardware watchdog timer.
- **SRAM:** Static random access memory.
- **S-reset:** Soft reset; a reset state for a CPU that can be commanded by an application program.
- **Stack segment:** A segment of memory allocated for a thread that provides buffer space for function arguments and local variables; each

application thread, the kernel thread, ISRs, and signal handlers typically all have their own stack space for the purpose of parameter passing and local variable instantiation.

- **State machine:** A formal design notation that includes a start state, state transitions driven by inputs made while in a specific state, and outputs on transitions.
- **Static priority stereo vision:** The use of two cameras separated by a known constant distance to judge distances to objects of unknown physical dimensions through the use of triangulation.
- **Stress testing:** Test vectors that are designed to stress the system by going beyond the requirements-based specification for limits—for example, commanding at high rate, exposure to high voltage ESD, shock testing, and thermal cycling.
- **Superscalar:** A *CPU pipeline* feature that employs parallel hardware within a single CPU core to allow for two or more instructions to be fetched, executed, and retired concurrently. Note that this feature of a *CPU pipeline* can yield a *CPI* less than 1.0 for the CPU core.
- **Suspended task:** A VxWorks task state entered when an exception (NMI) is generated by a task; the RTOS handles the exception and suspends the task to protect the system.
- **SWIC:** Software in circuit; a technique where software instrumentation is used to trace execution—for example, logging messages to a file from an application.
- **Switch-hook:** A VxWorks call-back mechanism where the kernel calls a user function on each and every context switch.
- **Symbol table:** An array of function and global variable names and addresses where they are stored in their text and data segments respectively.
- **Synchronous:** An event or stimulus that occurs at a specific point in time relative to other events in the system rather than at any time—for example, a thread of execution can perform a *semaphore take* to synchronize with an *ISR*; the *ISR* will execute asynchronously, but the processing provided by the thread performing the semaphore take will be *synchronous* since it is known that this processing will be provided

only after the *semaphore take* AND the *semaphore give* performed by the *ISR*.

- **Synchronous bus:** A bus that has clocked address, data, and control cycles.
- **Syndrome:** The encoded bits in an error detection and correction scheme that indicate SBE or MBE and contain the code for correction of SBEs.
- **System life cycle:** The process of turning an embedded system concept into a working maintained system. The steps potentially include: concept, requirements, high-level design, detailed design, implementation of units and subsystems, unit/subsystem test, integration, system test, acceptance testing, fielding, maintenance, and unit/system regression testing.
- **System test:** End-to-end and feature tests performed after the units and subsystems in a larger system have been unit tested and are integrated for the first time.
- **T (in RMA):** The period of a service request type. In many cases this will be based upon the worst-case frequency of the event(s) that cause a service to be released.
- **Target:** The embedded computing system, including the processor complex and all IO devices.
- **Target agent:** An embedded service that provides development, debug, and performance analysis features, such as cross debugging, code loading and linking, and RTOS event traces.
- **Target server:** A service on the host development system that provides an interface to host tools and translates user inputs into target agent commands and target agent responses into application data.
- **Task:** A thread with normal thread state, including stack, registers, PC, but also including signal handlers, task variables, task ID and name, priority, entry point, and a number of state and inter-task communication data contained in a TCB.
- **Task spinning:** When a task loops where it is expected to block and wait for a resource before proceeding.

- **Task wedging:** When a task blocks on a resource indefinitely or is suspended or fails to loop and post a keep-alive periodically.
- **TCB:** Task control block; the data structure associated with a VxWorks task that contains all task data in addition to task stack and context.
- **Text segment (code segment):** A segment of memory used for storing the machine code associated with an application, kernel image, or boot image.
- **TFTP:** Trivial file transfer protocol; a simplified FTP (File Transfer Protocol) that allows a client to download files from one known directory in a file system.
- **Thread (of execution):** A thread is simply the trace of a CPU's *PC* over time not including *context switch* code execution by an *RTOS*. State information may or may not be associated with a thread of execution, but the value of the *PC* before a context switch is the minimum state that must be maintained on a system that includes *preemption*.
- **Throughput:** An aggregate measure of speed and efficiency for a device—for example, for a processor the measure is MIPS (millions of instructions per second) and for an IO device the measure would be Mbps or Gbps (megabits/sec or gigabits/sec).
- **Tick:** A counter that counts interval timer interrupts and is used by an *RTOS* for basic timer services—for example, to provide the minimum resolution for timeouts on blocking calls (the *RTOS* will unblock a call made with a timeout specified within tick accuracy).
- **Time slice:** A unit of CPU called a quantum that can be allocated to a thread in a preemptible best-effort system; in these systems timer services often makes a call into the scheduler on each system tick in order to provide quantum preemption. So, the tick, a quantum, and timeout resolution are typically all the same—for example, Linux/Unix scheduling.
- **Timeout:** When making a blocking call, in order to avoid “wedging,” where a thread is blocked indefinitely, it is most often advisable to specify a timeout for any blocking call, at which time the thread will be asynchronously awoken and will continue execution; this can be done by setting a timer that is set up to throw a signal to a timeout handler prior

to making a blocking call if the API does not directly support a timeout option.

- **Timer services:** An interrupt handler set up by the RTOS that counts ticks on each interval timer interrupt and signals any threads that have reached a timeout threshold.
- **Top half (application interface):** The interface presented to calling threads/tasks/processes by a driver. The top half includes thread control features, such as blocking (using a `semTake` most often), and policy, such as how many threads it will allow to read/write a device at once.
- **Trace:** A linear buffer with records that include time (cycle count) and state information for a processor core and/or application code.
- **TTL:** Transistor to transistor logic; traditional 5v digital logic levels. Also, time-to-live: counter in a datagram that is decremented on each node-to-node hop such that the packet is discarded when TTL=0; this prevents a packet from hopping around the network indefinitely and creating a problem.
- **Unbounded:** When a condition can persist for a nondeterministic amount of time—for example, unbounded priority inversion where the set of middle-level priority interference tasks may cause the inversion to persist for an arbitrary time.
- **Unit test:** A test designed to validate and verify a software and/or hardware unit that is a building block for a larger system in isolation.
- **Utility curve:** An XY graph that shows time on the x-axis between a service release and relative deadline and shows utility or damage caused to the system caused by service response generation.
- **Virtual timer:** A timer that is not directly supported by a hardware interval timer, but rather is a software tick counter that can generate a signal after the passing of N software ticks.
- **VoIP:** Voice-over IP; a protocol for transporting voice duplex audio over the Internet protocol.
- **Watchdog timer:** A hardware-based interval timer that counts down (or up) and when it reaches zero (or all F's) it generates an *H-reset* signal causing the system to reboot; critical software services are expected to post keep-alives to a system sanity monitor that normally in turn

resets the watchdog timer before it expires. If the software loses sanity—that is, the sanity monitor fails to reset the watchdog timer (e.g., if a deadlock were to occur), then the idea is that the system will be able to recover by rebooting.

- **WCET:** Worst-case execution time; the longest number of CPU cycles required by a service release ever observed and/or theoretically possible, given the hardware architecture and algorithm for data processing used in the service.
- **Wear leveling:** A flash file system method to ensure that maximum capacity and operational longevity are maintained in a flash device that hosts a file system; since flash is divided into sectors, with each sector having a maximum expected number of erase/write cycles, this method attempts to keep erase counts for all sectors approximately the same so no one sector wears out early.
- **White-box test:** A set of test vectors that drive specific execution paths in a software unit by design so that the software unit test meets specific path, statement, condition, and/or decision coverage criteria; such tests require intimate knowledge of the software unit, such as API return codes, error conditions, and IO ranges.
- **Write-back:** When a processor updates memory from registers or cache.
- **Write-through:** When a processor maintains cache/memory coherency by always writing cache and the corresponding memory location on all writes to locations that are cached.

ABOUT THE DVD

This DVD has all of the RTECS original content and adds new FreeRTOS examples, expanded Linux examples, and up-to-date resources for real-time embedded systems development with RTOS and Linux, including the following:

- **Index:** The index has a browsing page for contents of the DVD to allow readers to self-instruct and explore contents.
- **VxWorks-Examples:** VxWorks example C code demonstrating RTOS features and mechanisms including the POSIX 1003.1b real-time API extensions
- **VxWorks-Drivers:** Bt878 video frame grabber driver and Cirrus 4281 example driver C code for VxWorks RTOS
- **Material-and-Examples (original content):** From CD content of RTECS book by author Sam Siewert. Folders include:
 - **Image-Processing:** Image processing C code examples that can be built as applications for Linux or VxWorks
 - **PMAPI:** Performance monitoring API code for PowerPC Darwin OS and for x86 PCs
 - **Contributed:** Contributed code implementing projects in computer vision, Voice/IP, and robotics
- **FreeRTOS Example Code:** Examples from the FreeRTOS community as well as co-author John Pratt.

- **Linux Examples:** Linux example application code for embedded and desktop systems. These examples have been tested on the Beagle xM and the NVIDIA Jetson embedded Linux systems. The 2006-Examples includes basic Linux examples to test memory, profile, assert on error and basic threading. The 2015-Examples extend these basic examples in categories that provide mixed C and Assembly code, computer-vision (using OpenCV), digital media (MPEG), and operating systems including both application and kernel code examples.
- **Linux Getting Started Documents:** For anyone brand new to Linux, this is what the author provides for students new to Linux to get them a rapid start using Virtual Box and Ubuntu Linux.
- **Utilities:** Includes a copy of Win32DiskImager, open source utility for Windows 7 (and earlier versions of Windows) to create a boot and root Micro-SD card.
- **Documents to Assist with Recommended Linux and FreeRTOS Embedded Systems:** Including the Jetson TK1, Altera DE1-SoC, and the Beagle xM. Readers should refer to the websites noted for the latest document, but this is a good starting point.
- **HTML:** Images used in the index.html file.
- **IBM In Print Links to DeveloperWorks:** Articles related to RTECS with Linux and RTOS published by Sam Siewert with IBM are available on DeveloperWorks.
- **IBM Out of Print:** Access to IBM articles related to the book that are no longer in print by IBM on DeveloperWorks.

The DVD also has these resources:

- **Figures:** All figures contained in the book.
- **Visio-Design examples:** A partially completed Visio UML design template for a stereo-vision system designed for an x86 VxWorks platform.
- **Photos and Videos:** JPEG, MPEG, and AVI photos and streaming video of example project demonstrations completed at the University of Colorado.

All VxWorks C code has been tested using Workbench 3.2 and VxWorks 6.8. Much newer versions of Workbench and VxWorks are available, and

the code is expected to compile and work on these new revisions, but has not been tested for them. VxWorks C code should be imported into a Workbench workspace and project that is a “loadable project” and built within the Workbench build framework.

All Linux C code has been tested using Linux 3.13.x with the Ubuntu 14.04 LTS (Long Term Support) distribution [Linux sam-ubuntu-Virtual-Box 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux]. Make files are included with the Linux code.

All Linux C code using the OpenCV library has also been tested on the Jetson TK1 embedded system, a recommended embedded Linux board for the book.

All Linux C code, excluding the OpenCV library-based examples, have been tested on both the Beagle xM using the included reference bootable image and instructions to create a boot/root image and the Altera DE1-SoC embedded system running the recommended Linux distribution by Altera.

B.1 Minimum System Requirements

VxWorks 5.4 or newer running on a Pentium, Pentium II, III, or IV x86 microprocessor, a host development system running Windows 2000 or XP, and Ethernet network to interconnect the host and target system.

For Linux, an x86 Pentium or better system running a 2.4.x or newer kernel and Red Hat 9 or newer Linux distribution.



WIND RIVER SYSTEMS UNIVERSITY PROGRAM FOR WORKBENCH/ VxWORKS

Wind River provides an RT Linux distribution as well as Workbench/Vx-Works, which can be licensed free of charge for academic teaching and research programs. This was done for the University of Colorado's course ECEN 5623, Real-Time Embedded Systems, taught since fall 2000 at the Boulder campus.

Information on the program can be found at:

<http://www.windriver.com/universities/>

The FAQ page is very useful as well:

<http://www.windriver.com/universities/faq>

The program has worked very well at the University of Colorado at Boulder. The tools have been used in an x86, PowerPC, and Xscale mixed platform lab to implement all of the example projects included with Real-Time Embedded Components and Systems.

REAL-TIME AND EMBEDDED LINUX DISTRIBUTIONS AND RESOURCES

Featured in *Real-Time Embedded Components and Systems with Linux and RTOS*, 2nd edition:

- Ubuntu Linux distributions, <http://www.ubuntu.com/>
- Debian Linux distributions, <https://www.debian.org/>
- Yocto Linux distributions, <https://www.yoctoproject.org/>

Widely used Linux distributions for soft real-time and embedded systems not featured:

- The Linux Foundation, <http://www.linuxfoundation.org>, <http://www.linux.com/>, <http://events.linuxfoundation.org/events/embedded-linux-conference>
- Concurrent Computer, <https://www.ccur.com/linux/>
- Wind River Linux, <http://www.windriver.com/linux>
- Monta Vista Linux, <http://www.mvista.com/>
- TimeSys Linux, <http://www.timesys.com/>
- Robot Operating System, <http://www.ros.org/>
- Embedded Linux Wiki, <http://elinux.org>
- Real-Time Linux Wiki, <https://rt.wiki.kernel.org>
- Open Source Automation Development Lab, <https://www.osadl.org/>
- Wikipedia Linux distributions, https://en.wikipedia.org/wiki/Linux_distribution

BIBLIOGRAPHY

- [Abbott03] Doug Abbott, *Linux for Embedded and Real-Time Applications*, Elsevier Science (USA), 2003.
- [Anderson99] Don Anderson, *Firewire System Architecture*, 2nd Edition, Addison-Wesley, 1999.
- [Anderson01] Don Anderson and Dave Dzatko, *Universal Serial Bus System Architecture*, 2nd Edition, Addison-Wesley, 2001.
- [Barr99] Michael Barr, *Programming Embedded Systems in C and C++*, O'Reilly, 1999.
- [Bovet03] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, 2nd Edition, O'Reilly, 2003.
- [Briand99] Loïc P. Briand and Daniel M. Roy, *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*, IEEE Computer Society, 1999.
- [Brooks86] R. A. Brooks (1986) "A Robust Layered Control System For A Mobile Robot," IEEE Journal of Robotics and Automation, RA-2, April, pp. 14–23.
- [Budruk04] Ravi Budruk, Don Anderson, and Tom Shanley, *PCI Express System Architecture*, Addison-Wesley, 2004.
- [Buttazzo02] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic, 2002.
- [Campbell02] Iain Campbell, *Reliable Linux: Assuring High Availability*, Wiley, New York, 2002.
- [Ciletti03] Michael D. Ciletti, *Advanced Digital Design with the Verilog HDL*, Prentice Hall, Pearson Education, 2003.

- [Cook02] David Cook, *Robot Building for Beginners*, Springer-Verlag, New York, 2002.
- [Cook04] David Cook, *Intermediate Robot Building*, Springer-Verlag, New York, 2004.
- [Corbet03] Karim Corbet, *Building Embedded Linux Systems*, O'Reilly, 2003.
- [Corbet05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartmen, *Linux Device Drivers*, 3rd Edition, O'Reilly, 2005.
- [Di Steffano67] Di Steffano III, Stubberud, and Williams, *Feedback and Control Systems*, Schaum's Outline Series, McGraw-Hill, New York, 1967.
- [Douglass03] Bruce Powel Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*, Addison-Wesley, 2003.
- [Douglass04] Bruce Powel Douglass, *Real-Time UML Third Edition: Advances in the UML for Real-Time Systems*, Addison-Wesley, 2004.
- [Gallmeister95] Bill O. Gallmeister, *POSIX.4: Programming for the Real World*, O'Reilly, 1995.
- [Ghezzi91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [Gomaa00] Hassan Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, 2000.
- [Goody00] Roy W. Goody, *OrCAD PSpice for Windows Volume 1: DC and AC Circuits*, 3rd Edition, Pearson Education, 2000.
- [Grötke02] Thorsten Grötke, Stan Liao, Grant Martin, and Stuart Swan, *System Design with SystemC*, Kluwer Academic, 2002.
- [Hennessy90] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.
- [Hennessy03] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd Edition, Morgan Kaufmann, 2003.
- [Herbert04] Thomas F. Herbert, *The Linux TCP/IP Stack: Networking for Embedded Systems*, Charles River, Hingham, MA, 2004.
- [IBM00] IBM Corporation, *PowerPC Microprocessor Family: The*

- Programming Environments for 32-Bit Microprocessors*, G522-0290-01, IBM, 2000.
- [Intel02a] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel, 2002.
- [Intel02b] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Intel, 2002.
- [Intel02c] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, Intel, 2002.
- [Intel02d] Intel Corporation, *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, Intel, 2002.
- [Jack96] Keith Jack, *Video Demystified*, 2nd Edition, HighText Interactive, San Diego, CA, 1996.
- [Kernighan88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall, 1988.
- [Labrosse02] Jean J. Labrosse, *MicroC/OS-II, The Real-Time Kernel*, 2nd Edition, CMP Books, 2002.
- [Lewis02] Daniel W. Lewis, *Fundamentals of Embedded Software: Where C and Assembly Meet*, Prentice Hall, 2002.
- [Li03] Qing Li and Caroline Yao, *Real-Time Concepts for Embedded Systems*, CMP Books, 2003.
- [Luther99] Arch Luther and Andrew Inglis, *Video Engineering*, 3rd Edition, McGraw-Hill, 1999.
- [Mano91] M. Morris Mano, *Digital Design*, 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [Motorola97] Motorola Inc., *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, MPCFPE32B/AD REV. 1, Motorola, 1997.
- [Nichols96] Bradford Nichols, Dick Buttler, and Jacqueline Proulx Farrell, *Pthreads Programming*, O'Reilly, 1996.
- [Nise03] Norman S. Nise, *Control Systems Engineering with CD*, 4th Edition, Wiley, 2003.
- [Noergaard05] Tammy Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*, Elsevier, 2005.

- [Patterson94] David A. Patterson and John L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.
- [Phillips94] Charles L. Phillips and Troy Nagle, *Digital Control System Analysis and Design*, 3rd Edition, Pearson Education, 1994.
- [Poynton03] Charles Poynton, *Digital Video and HDTV: Algorithms and Interfaces*, Morgan Kaufmann, Elsevier Science (USA), 2003.
- [Pressman92] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 3rd Edition, McGraw-Hill, 1992.
- [Schroeder98] Chris Schroeder, *Inside OrCAD Capture for Windows*, Elsevier Science and Technology Books, 1998.
- [Shanley95] Tom Shanley and Don Anderson, *ISA System Architecture*, 3rd Edition, Addison-Wesley, 1995.
- [Shanley99] Tom Shanley and Don Anderson, *PCI System Architecture*, 4th Edition, Addison-Wesley, 1999.
- [Shanley01] Tom Shanley, *PCI-X System Architecture*, Addison-Wesley, 2001.
- [Shaw01] Alan C. Shaw, *Real-Time Systems and Software*, Wiley, 2001.
- [Simon99] David E. Simon, *An Embedded Software Primer*, Addison-Wesley, 1999.
- [Smith97] Steven Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical, 1997.
- [Solari97] Stephen J. Solari, *Digital Video and Audio Compression*, McGraw-Hill, 1997.
- [Stevens98] W. Richard Stevens, *Unix Network Programming, Networking APIs: Sockets and XTI*, 2nd Edition, Volume 1, Prentice Hall, 1998.
- [Stevens99] W. Richard Stevens, *Unix Network Programming, Interprocess Communications*, 2nd Edition, Volume 2, Prentice Hall, 1999.
- [Topic02] Michael Topic, *Streaming Media Demystified*, McGraw-Hill, 2002.
- [Wind River 99] Wind River Systems, Inc., *VxWorks Programmer's Guide*, 5.4, 1st Edition, Wind River Systems, 1999.

World Wide Web References

- [ABB] ABB industrial robotics, <http://www.abb.com/robotics>.
- [AcroRob] Acroname Robotics, <http://acroname.com/>.
- [ALSA] Advanced Linux Sound Architecture, <http://www.alsa-project.org/>.
- [CIMG] CIMG Library with C++ image processing templates, <http://cimg.sourceforge.net/>.
- [Concurrent] The Concurrent Versions System (CVS) can be very helpful for managing source code bases for real-time embedded development projects, <http://www.nongnu.org/cvs/>.
- [DocumentPC] Documentation for the PowerPC from IBM can be found on the main technical documentation site, <http://www-306.ibm.com/chips/techlib>.
- [eCos RTOS] The eCos RTOS is available through the Free Software Foundation with GPL licensing, <http://ecos.sourceware.org/>.
- [FSMLabs] FSMLabs provides a university program for use of its RT Linux distributions, <http://www.fsmlabs.com/universities.html>.
- [GarageTech] Garage Technologies Inc. also offers a six-degree-of-freedom arm, <http://www.garage-technologies.com/index.html>.
- [Hitec] Hitec offers a wide range of servos suitable for simple robotics projects, http://www.hitecrcd.com/homepage/product_fs.htm.
- [Honda ASIMO] Honda's ASIMO robot, <http://world.honda.com/ASIMO/>.
- [IBM] IBM provides a download for PowerPC 4xx processor core models for use with Open SystemC design tools, <http://www-128.ibm.com/developerworks/power/systemc/>.
- [IETF RTP] IETF RTP, <http://www.ietf.org/rfc/rfc3550.txt>.
- [IETF RTSP] IETF RTSP, <http://www.ietf.org/rfc/rfc2326.txt>.
- [Intel Dual-Core] Intel Dual-Core Pentium documentation, <http://www.intel.com/design/pentiumd/documentation.htm>.
- [Intel Mobile] Intel Mobile Pentium documentation, <http://www.intel.com/design/mobile/pentiumm/documentation.htm>.
- [Intel Pentium] Intel Pentium (IA32) documentation, <http://www.intel.com/design/pentium4/documentation.htm>.

- [Irfanview] Irfanview, provides viewing of PPM and PGM formats and many more, <http://www.irfanview.com/>.
- [Latest Python] The latest version of the Python programming language used for video stream display in examples, <http://python.org/>.
- [Lynxmotion] Lynxmotion offers the Lynx 5 and 6 servo-controlled robotic arm kits, <http://www.lynxmotion.com/>.
- [Microc] The Microc RTOS can be purchased for a very reasonable cost for academic or personal use along with a book describing it, <http://www.rabbitsemiconductor.com/etc/microc.shtml>.
- [Microsoft Vis] Microsoft Visio can be used for UML design, basic circuit, mechanical, and system-level design. The tool can be downloaded for free trial from Microsoft, <http://www.microsoft.com/office/visio/prodinfo/trial.mspx>.
- [Motoman] Motoman industrial robotics, <http://motoman.com/>.
- [NASA] NASA Johnson Space Center Robonaut, http://vesuvius.jsc.nasa.gov/er_er/html/robonaut/robonaut.html.
- [NCD] National Control Devices, source for NCD209 Microchip PIC Hobby Servo Controller, <http://www.controlanything.com/>.
- [NeuroRob] NeuroRobotics Ltd. offers a research-grade robotic arm, <http://www.neurorobotics.co.uk/>.
- [OpenCD] Open Source Computer Vision Library project, <http://sourceforge.net/projects/opencvlibrary/>.
- [OpenSystemC] The Open SystemC Initiative provides tools and a simulation environment for SystemC design automation, <http://www.systemc.org/>.
- [OrCAD] OrCAD has a number of free downloads for lightweight versions of their products for schematic capture and simulation, <http://orcad.com/downloads.aspx>.
- [OWI-7] The OWI-7 five-degree-of-freedom arm comes from OWI Robotics, <http://owirobots.com/>.
- [PeeWee] PeeWee Linux is an enhanced 2.2 Linux kernel with a small footprint for embedding, <http://peeweelinux.com/>.
- [Real-Time Java] The Real-Time Java Specification, <https://rtsj.dev.java.net/>.

- [Real-TimeLin] The Real-Time Linux Foundation has many resources for configuring your own RT Linux platform, <http://www.realtimelinux-foundation.org/>.
- [RivaTV] The RivaTV project is developing drivers to support NTSC frame capture, like Video for Linux, but specifically for nVidia, <http://rivatv.sourceforge.net/>.
- [RobotWorx] RobotWorx industrial robotics integration, <http://www.robots.com/>.
- [RobResearch] Robotics Research manufactures seven-degree-of-freedom highly dexterous arms with torque control, <http://www.robotics-research.com/>.
- [RTEMS] RTEMS is an open source or free licensed real-time operating system, <http://www.rtems.com/>.
- [Sony Ent.] Sony entertainment robotics, <http://www.sony.net/Products/aibo/>.
- [Subversion] The Subversion source code control system offers an alternative to CVS, <http://subversion.tigris.org/>.
- [TAO] TAO Real-Time CORBA, <http://www.cs.wustl.edu/~schmidt/TAO.html>.
- [Toyota Partner] Toyota partner robot, <http://www.toyota.co.jp/en/special/robot/>.
- [UML Visio] The UML Visio stencils used are available for download, <http://www.phrubby.com/stencildownload.html>.
- [VidLinux] Video for Linux, <http://www.exploits.org/v4l/>.
- [Weeder] Weeder Technologies has a wide range of serial multi-drop data acquisition and control boards, <http://www.weedtech.com/>.
- [Wind River] The Wind River University program can be joined to obtain RT Linux and VxWorks for use in classroom instruction, <http://www.windriver.com/universities/>.
- [Xilinx] The Xilinx University Program offers access to many reconfigurable computing resources for embedded systems design, <http://www.xilinx.com/univ/>.

INDEX

A

- ADC (Analog-to-Digital Converter), 10, 156
 - Flush, 157
 - Successive approximation, 157
- ALSA (Advanced Linux Sound Architecture), 378
- Anytime algorithm, 34
- Apollo 11 lunar module, 5

B

- Bottom-up approach, 153
- Burn and learn approach, 278

C

- Circular wait, 130
- Computer vision applications, 403–413
 - Characterizing Cameras, 409–411
 - Image Processing for Object Recognition, 406–409
 - PSF (Point Spread Function), 406
 - Introduction, 403–404
 - Object Tracking, 404–406
 - Centroid, 406
 - Field of view (FOV), 404
 - Tracking performance for a tilt/pan subsystem, 405
 - Pixel and Servo Coordinates, 411–412
 - Stereo-Vision, 412–413
 - Parsec, 412
- Context switch latency, 14
- Continuous media real-time applications, 15, 365–381
 - Audio Codecs and Streaming, 378
 - Audio Stream Analysis and Debug, 379
 - Introduction, 365–366
 - Uncompressed Video Frame Formats, 370–371

- Video, 367–370

- A method for deriving RGB from NTSC sampling, 368

- Video Codecs, 371–373

- 3 dimensions of, 371

- Video Stream Analysis and Debug, 374–378

- Video Streaming, 373–374

- Voice-Over Internet Protocol (VoIP), 379

- CPU resource management, 5, 22, 34, 67

D

- Data-dependency stall, 109

- Deadline, 67

- Debugging, 253–292

- Application-Level, 291–292

- wvEvent(), 292

- Assert, 262–263

- Checking Return Codes, 263–264

- Reasons of API failure, 263–264

- Exceptions, 254–262

- Setout, logMsg, gdb, 256–262

- External Test Equipment, 287–291

- LA (Logic Analyzer), 291

- Introduction, 253–254

- Kernel Scheduler Traces, 273–278

- Linux Trace Toolkit, 276

- Tornado/VxWorks development framework, 273

- WindView, 273, 275

- POST (Power-On Self-Test) and

- Diagnostics, 282–287

- Memory tests, 282–263

- Otool, 285

- Single-Step Debugging, 264–273

- Processor-level debugging, 271

- System- or kernel-level debugging, 270

- Task- or process-level debugging, 264

- Test Access Ports, 278–280
 - EEPROM, 278
 - JTAG, 278
- Trace Ports, 280–282
 - Heisenbug, 280
- Digital video processing, 365
- Distributed Continuous Media Real-Time Service, 16
- DMA (Direct Memory Access) engine, 10

E

- Earliest Deadline First (EDF), 46, 91
- EEPROM (Electrically Erasable Programmable Read-Only Memory), 126
- Embedding, 4, 7
- Embedded system(s), 3, 4
 - Brief systems, 7
- Embedded System Components, 153–188
 - Firmware Components, 176–178
 - Boot Code or board support package (BSP), 176
 - Device interface drivers, 177
 - Operating System Services, 178
 - Hardware Components, 154–176
 - Actuators, 158–159
 - Characteristics of, 158
 - Bus Interconnection, 165–170
 - Common Memory Map for, 175
 - Comparison of PCI and VME Buses, 166
 - High-Speed Serial Interconnection, 170–172
 - Interconnection Systems, 173–174
 - IO Interfaces, 159–163
 - Low-Speed Serial Interconnection, 172–173
 - Memory Subsystems, 174–176
 - PCI Express Byte Lane Network Architecture, 172
 - Processor and IO Interconnection, 164–165
 - Processor Complex or SoC, 163–164
 - Sensors, 155–158
 - Introduction, 153–154

- Stereo-vision tracking system, 154
- RTOS System Software, 178–182
 - Binary Semaphores, 180–181
 - Coding practices, 179
 - Message queues, 179–180
 - Mutex Semaphores, 181
 - Software Virtual Timers, 181–182
 - Software Signals, 182
- Software Application Components, 182–188
 - Application Services, 182–184
 - Communicating and Synchronized Services, 186–188
 - Reentrant Application Libraries, 185
 - Introduction, 3–4
- Equal Utility Curve, 81
- ECC (Error Correcting Circuitry) memory, 121
- Execution Efficiency, 107–110
 - Hazards affecting CPI, 108

F

- FCFS (First Come First Served), 44
- Fixed-priority preemptive scheduling, 47, 68
- FPGA (Field Programmable Gate Array), 199

G

- Graphical user interfaces (GUI), 5

H

- High availability and high reliability (HA/HR), 319–327
 - Design Trade-Offs, 324–326
 - FMEA (Failure Modes and Effects Analysis), 325
 - Hierarchical Approaches for Fail-Safe Design, 327
 - Introduction, 319
 - Reliability, 321–323
 - Dual-String, Cross-Strapped Subsystem Interconnection, 322
 - Simple parameters for, 323
 - Reliable Software, 323–324

Similarities and Differences, 320–321
 Mathematical relationship, 320

I

Input latency, 14, 84
 Intermediate IO, 104–106
 Execution and IO overlap definitions,
 104–106
 IO Architecture, 110–111
 ISRs (Interrupt Service Routines), 41
 Interference, 48

L

Latency hiding, 30
 Least Laxity First (LLF), 46
 Level-1 cache, 108

M

Main+ISR or Cyclic Executive approach,
 10, 41
 Mars Pathfinder spacecraft, 6
 MATLAB model, 404
 Memory, 115–127
 Capacity and allocation, 120
 ECC Memory, 121–125
 Flash-file systems, 126–127
 Introduction, 115–116
 Physical Hierarchy, 116–119
 Shared memory, 120–121
 Memory-Mapped IO (MMIO), 116
 M-JPEG (Motion-Joint Photographic
 Experts Group), 373
 MMR (Memory-Mapped Register), 99
 MMU (Memory Management Unit), 115
 MPEG (Motion Picture Experts Group),
 21, 373
 Mutex (mutual exclusion semaphore),
 133, 248
 Multi-resource, 129–138
 Blocking, 130
 Critical Sections to Protect Shared
 Resources, 132
 Deadlock and Livelock, 130–132
 Introduction, 129–130

Power Management and Processor Clock
 Modulation, 138
 Priority Inversion, 132–137
 Unbounded Priority Inversion
 Solutions, 134–137

N

N&S feasibility test, 71, 91
 NVRAM (NonVolatile RAM), 126

O

OASDL (Open Source Automation
 Development Lab), 237
 Online or on-demand scheduling, 71

P

PCI (Peripheral Component
 Interconnect), 165
 Performance Tuning, 25, 295–317
 Building Performance Monitoring into
 Software, 304–305
 Basic methods for, 304
 Drill-Down Tuning, 296–300
 Characteristics of code segments that
 affect performance, 297
 Characteristics of services, 298
 Workload, 300
 Fundamental Optimizations, 317
 Fibonacci sequence, 317
 Hardware-Supported Profiling and
 Tracing, 300–303
 Basic methods for, 300
 Event tracing, 303
 Hot spot, 303
 RISCTrace, 302
 Introduction, 295–296
 Path Length, Efficiency, and Calling
 Frequency, 305–316
 Example codes, 305–316
 POSIX (Portable Operating Systems
 Interface), 16–17
 Real-time operating system
 mechanisms, 17
 Processing, 67–95

- Deadline-Monotonic Policy, 89–91
 - Dynamic-Priority Policies, 91–95
 - Feasibility, 72–73
 - Sufficient and necessary and sufficient, 72
 - Introduction, 67–68
 - Necessary and Sufficient Feasibility, 84–89
 - Completion Time Test, 87–89
 - Scheduling Point Test, 84–87
 - Preemptive Fixed-Priority Policy, 68–73
 - Necessary, sufficient, and necessary and sufficient conditions, 70–71
 - Rate-Monotonic Least Upper Bound, 73–84
 - PWM (Pulse-Width Modulation), 158
- Q**
- QoS (Quality of Service) systems, 5
- R**
- Real-Time Embedded system, 22, 118, 295
 - Real-Time Linux, 237–249
 - Embedding Mainline Linux, 238–244
 - Introduction, 237–238
 - Linux as a Non-Real-Time
 - Management and User Interface Layer, 244–245
 - Linux for Soft Real-Time Systems, 249
 - Methods to Patch and Improve Linux for Predictable Response, 245–248
 - Big Kernel Lock (BKL), 246
 - Tools for Linux for Soft Real-Time Systems, 249
 - Real-time service(s), 7–16
 - A Simple Polling State Machine for Real-Time Services, 9
 - Listing 1.1: Pseudo Code for Basic Real-Time Service, 8–11
 - Timeline, 12–14
 - Real-time standards, 16–17
 - Real time systems, 3, 4
 - Brief history of, 4–7
 - Introduction, 3–4
 - RM LUB, 68, 72
 - RM analysis or Policy, 129
 - Robotic applications, 383–401
 - Actuation, 387–393
 - MOSFET, 392
 - Relay Circuit, 389
 - Automation and Autonomy, 400–401
 - Shared control, 400
 - End Effector Path, 393
 - Introduction, 383–384
 - Robotic Arm, 384–387
 - Five-degree-of-freedom arm, 365
 - Sensing, 393–397
 - Laplace transforms, 395
 - PID Digital Control Loop, 396
 - PID Gain Tuning Rules, 397
 - Position encoders, 393
 - Tasking, 397–399
 - Basic capabilities, 398
 - RTOS (Real-Time Operating System, 191–234, 254, 292
 - Additional features of FreeRTOS, 229–230
 - AMP (Asymmetric Multi-core Processing), 194–199
 - AMP Architecture for Computer Vision, 195
 - Flynn's Taxonomy of Concurrent/Parallel System Architectures, 198
 - Evolution of Real-Time Scheduling and Resource Management, 192–194
 - FreeRTOS Real-Time Service Programming Fundamentals, 228–234
 - FreeRTOS Alternative to Proprietary RTOS, 221–225
 - FreeRTOS Platform and Tools, 225–228
 - Future Directions for, 216–217
 - Introduction, 191–192
 - Hypervisor concept, 218
 - Processor Core Affinity, 200–216
 - SMP (Symmetric Multi-core Processing), 199–200
 - SMP Support Models, 217–218

S

- SECDDED (Single Error Correction, Double Error Detection) Method, 122
- Service, 52
- Service dropout, 142
- Servomechanism, 158
- SJN (Shortest Job Next), 44
- Soft real time services, 141–148
 - Alternatives to Rate-Monotonic Policy, 144–147
 - Introduction, 141
 - Missed Deadlines, 142
 - Mixed Hard and Soft Real-Time Services, 147–148
 - Quality of Service, 143–144
 - MTTF, MTTR, MTBF, 143
- System life cycle, 331–362
 - Component Detailed Design, 339–348
 - Component Unit Testing, 348–356
 - Black box and glass box test, 350
 - Functional and interface tests, 349
 - Performance testing, 349
 - Soak testing, 349
 - Stress testing, 349
 - Configuration Management and Version Control, 357–361
 - CVS (Code Versioning System), 361
 - High-Level Design, 336–339
 - Methodologies support system-level design, 339
 - Introduction, 331–332
 - Overview, 332–335
 - 2-phase spiral model, 332–333
 - Regression Testing, 362
 - Requirements, 335–336
 - Risk Analysis, 336
 - Points for, 336
 - System Integration and Test, 357
- System resources, 21–61
 - Introduction, 21–23
 - RTOS (Real-Time Operating Systems), 50–58
 - Key features of an RTOS, 50–51
 - Scheduling Policies, 53
 - Real-Time Service Utility, 31–38

- Resource Analysis, 23–31
 - Processing, memory, IO, 23
- Scheduler Concepts, 41–50
 - Preemptive Fixed-Priority Scheduling Policy, 47–50
 - Preemptive vs. Non-preemptive Schedulers, 44–47
 - Thread Dispatch Policy, 43
- Scheduling Classes, 38–39
 - CPU Resource-Scheduling Taxonomy, 38
 - Multiprocessor Systems, 39
- The Cyclic Executive, 39–41
 - Cyclic schedule, 40
- Thread-Safe Reentrant Functions, 59–61

T

- Task context, 52
- TCM (Tightly Coupled Memory), 109, 119
- The Lehoczky, Sha, Ding theorem, 29, 84–87
- Thermistor, 155
- Thrashing, 117
- Thread context, 24, 52
- Top-down approach, 153

V

- VME (Versa Module Extension), 165
- VxWorks, 273, 292
 - Cross Compiling Code for, 63
 - semTake() and semGive, 132

W

- waitFor function, 11–12
- WCET (Worst-Case Execution Time), 24, 100–103, 110
 - Equations, 103
 - System characteristics, 100
- Weakly consistent, 109
- Worst-case latency, 30

