REAL-TIME SYSTEMS DESIGN AND ANALYSIS

IEEE Press
445 Hoes Lane
Piscataway, NJ 08854

Technical Reviewers
Larry Bernstein, Stevens Institute of Technology
Bernard Sick, University of Kassel
Olli Vainio, Tampere University of Technology

REAL-TIME SYSTEMS DESIGN AND ANALYSIS

Tools for the Practitioner

Fourth Edition

PHILLIP A. LAPLANTE

SEPPO J. OVASKA

IEEE PRESS

A JOHN WILEY & SONS, INC., PUBLICATION

Cover photo courtesy of NASA.

Library of Congress Cataloging-in-Publication Data:
Laplante, Phillip A.
Real-time systems design and analysis : tools for the practitioner / Phillip A. Laplante, Seppo J. Ovaska.—4th ed.
p. cm.
ISBN 978-0-470-76864-8 (hardback)
1. Real-time data processing. 2. System design. I. Ovaska, Seppo J., 1956- II. Title.
QA76.54.L37 2012
004'.33–dc23

2011021433

Printed in the United States of America

oBook ISBN: 9781118136607
ePDF ISBN: 9781118136577
ePub ISBN: 9781118136591
eMobi ISBN: 9781118136584

10 9 8 7 6 5 4 3 2 1

Phil:
To Nancy, Chris and Charlotte, with all my love

Seppo:
To Helena, Sami and Samu—my everything

CONTENTS

PREFACE

This book is an introductory text about real-time systems—systems where *timeliness* is a crucial part of the correctness of the system. Real-time software designers must be familiar with computer architecture and organization, operating systems and related services, programming languages, systems and software engineering, as well as performance analysis and optimization techniques. The text provides a pragmatic discussion of these subjects from the perspective of the real-time systems designer. Because this is a staggering task, depth is occasionally sacrificed for breadth. Nevertheless, thoughtful suggestions for additional literature are provided where depth has been sacrificed due to the available page budget or other reasons.

This book is intended for junior–senior level and graduate computer science, computer engineering and electrical engineering students, as well as practicing software, systems and computer engineers. It can be used as a graduate level text if it is supplemented with an advanced reader or a focused selection of scholarly articles on a specific topic (which could be gathered from the up-to-date bibliographies of this edition). Our book is especially useful in an industrial setting for new real-time systems designers who need to get "up to speed" very quickly. Earlier editions of this book have been used in this way to teach short courses for several industrial clients. Finally, we intend for the book to be a desk reference of long-lasting value, even for experienced real-time systems designers and project managers.

The reader is assumed to have basic knowledge in programming in one of the more popular languages, but other than this, the prerequisites for this text are minimal. Some familiarity with discrete mathematics is helpful in understanding some of the formalizations, but it is not essential.

Since there are several preferred languages for real-time systems design, such as Ada, C, C++, C#, and increasingly, Java, it would be unjust to focus this book on one language, say C, when the theory and framework should be language independent. However, for uniformity of discussion, certain points are illustrated, as appropriate, in generic assembly language and C.

While the provided program codes are not intended to be ready-to-use, they can be easily adapted with a little tweaking for use in a real system.

This book is organized into nine chapters that are largely self-contained. Thus, the material can be rearranged or omitted depending on the background and interests of the instructor or reader. It is advised, however, that Chapter 1 would be explored first, because it contains an introduction to real-time systems as well as the necessary terminology.

Each of the chapters contains both easy and more challenging exercises that stimulate the reader to confront actual problems. The exercises, however, cannot serve as a substitute for carefully planned laboratory work or practical experience.

The first chapter provides an overview of the nature of real-time systems. Much of the basic vocabulary relating to real-time systems is developed along with a discussion of the main challenges facing the real-time system designer. Besides, a brief historical review is given. The purpose of this chapter is to foreshadow the rest of the book as well as quickly acquaint the reader with pertinent terminology.

The second chapter presents a detailed review of central computer architecture concepts from the perspective of the real-time systems designer. Specifically, the impact of advanced architectural features on real-time performance is discussed. The remainder of the chapter outlines different memory technologies, input/output techniques, and peripheral support for embedded systems. The intent here is to increase the reader's awareness of the impact of the computer architecture on various design considerations.

Chapter 3 provides the core of the text for those who are building practical real-time systems. This comprehensive chapter describes the three principal real-time kernel services: scheduling/dispatching, intertask communication/synchronization, and memory management. It also covers special problems inherent in these designs, such as deadlock and priority inversion.

Chapter 4 begins with a discussion of specific language features desirable in good software engineering practice in general and real-time systems design in particular. An evaluative review of several widely used programming languages in real-time systems design, with respect to these features, follows. Our intent is to provide explicit criteria for rating a language's ability to support real-time systems and to alert the user to the possible drawbacks of using each language in real-time applications.

In Chapter 5, the nature of requirements engineering is first discussed. Then a collection of rigorous techniques in real-time system specification is presented with illustrative examples. Such rigorous methods are particularly useful when automatic design and code-generation approaches are to be used

later in the development life cycle. Next, structured and object-oriented methodologies are discussed as alternative paradigms for requirements writing. At the end of this chapter, an extensive case study is provided.

Chapter 6 surveys several commonly applied design specification techniques used in both structured and object-oriented design. An emphasis on their applicability to real-time systems is made throughout. No single technique is a silver bullet, and the reader is encouraged to adopt his or her own formulation of specification techniques for the given application. A comprehensive design case study is also provided.

Chapter 7 discusses performance analysis techniques based on diverse estimation approaches. The proposed toolset is fully usable even before it is possible to perform any direct measurements. Moreover, a pragmatic discussion on the use of classical queuing theory for analyzing real-time systems is provided. Input/output performance issues are considered with an emphasis on buffer-size calculation. Finally, a focused analysis of memory utilization in real-time systems is presented.

Chapter 8 discusses additional software engineering considerations, including the use of software metrics and techniques for improving the fault-tolerance and overall reliability of real-time systems. Later in the chapter, different techniques for improving reliability through rigorous testing are discussed. Systems integration and performance optimization issues are also considered.

In Chapter 9, we look to the future of real-time systems hardware, software, and applications. Much of this chapter is speculative, and we had great fun imagining things yet to come and the way things ought to be with respect to real-time systems technology. This chapter forms a fruitful basis for class discussions, debates, and student projects.

When our book is used in a university course, typically students are asked to build a real-time multitasking system of their choice. Usually, it is a game on a PC, but some students can be expected to build embedded hardware controllers of moderate complexity. The authors' assignment to the reader would be to build such a game or simulation, using at least the coroutine model. The application should be useful or at least pleasing, so some sort of a game is a good choice. The mini-project should take no more than 20 hours and cover all phases of the software life cycle model discussed in the text. Hence, those readers who have never built a real-time system will have the benefit of the instructive experience.

Real-time systems engineering is based on more than 50 years of experience and global contributions by numerous individuals and organizations. Rather than clutter the text with endless citations for the origin of each idea, the authors chose to cite only the key ideas where the reader would want to seek out the source for further reading. Some of the text is adapted from two other books written by the first author on software engineering and computer architecture, Laplante (2003) and Gilreath and Laplante (2003), respectively. Where this has been done, it is so noted.

Many solid theoretical treatments of real-time systems exist, and where applicable, they are noted. Nonetheless, these books or journal articles are sometimes too theoretical for practicing software engineers and students who are often impatient to wade through the derivations for the resultant payoff. They want results that they can use now in the trenches, and they want to see *how* they can be used, not just know that they exist. In this text, an attempt is made to distill the most valuable of the theoretical results, combined with practical experience and insight to provide a toolkit for the practitioner.

This book contains extensive bibliographies at the end of each chapter. Where verbatim phrases were used, and where a figure came from another source, the authors tried to cite it appropriately. However, if any were inadvertently overlooked, the authors wish to correct the unfortunate error. Please notify the authors if you find any errors of omission, commission, citation, and so forth by e-mail, at plaplante@psu.edu or seppo.ovaska@aalto.fi, and they will be corrected at the next possible opportunity.

Since 1992, thousands of copies of the first three editions of this book have been sold to the college text and professional markets throughout the world. The only thing more gratifying than its adoption at such prestigious universities as Carnegie Mellon University, the University of Illinois at Urbana-Champaign, Princeton University, the United States Air Force Academy, Polytechnic University, and many others around the world, has been the enthusiastic feedback received from numerous individuals thankful for the influence that the book has had on them. The continuing international success of the first three editions along with recent technological advancements demanded that a fourth edition be produced.

The most fundamental change in the fourth edition is a new co-author, Dr. Seppo Ovaska, whose vast experience greatly complements that of the first author and adds a strong and timely international perspective.

The fourth edition addresses the important changes that have occurred in the theory and practice in the construction of real-time systems since the publishing of the third edition in 2004. Chapters 1–8 have been carefully revised to incorporate new material, correction of errors, and elimination of outdated material. Moreover, Chapter 9 is a brand-new chapter devoted to future visions on real-time systems. Totally new or substantially revised discussions include:

- Multidisciplinary design challenges
- Birth and evolution of real-time systems
- Memory technologies
- Architectural advancements
- Peripheral interfacing
- Distributed real-time architectures
- System services for application programs
- Supplementary criteria for multi-core and energy-aware support

- Automatic code generation
- Life cycle models
- Arguments related to parallelization
- Uncertainty in real-time systems
- Testing patterns and exploratory testing
- Real-time device drivers
- Future visions on real-time systems

While approximately 30% of previous material has been discarded, another 40% has been added, resulting in a unique and modern text. In addition, several new examples have been included to illustrate various important points. Hence, it is with pride and a sense of accomplishment that we are presenting this timely and carefully composed book to students and practicing engineers.

PHILLIP A. LAPLANTE

West Chester, Pennsylvania

SEPPO J. OVASKA

Hyvinkää, Finland

August 2011

REFERENCES

W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Approach*. Norwell, MA: Kluwer Academic Publishers, 2003.

P. A. Laplante, *Software Engineering for Image Processing*. Boca Raton, FL: CRC Press, 2003.

ACKNOWLEDGMENTS

Phil Laplante wishes to thank his dear friend Dr. Seppo Ovaska for being the perfect collaborator. Easy to work with, Seppo's industriousness, experience, insight, patience, and attention to detail perfectly complemented Phil's strengths and weaknesses. The vast majority of differences between the third and fourth editions are due to Seppo's hard work. As a result of Seppo's contributions, the fourth edition is far superior to any previous edition of this book. And this book is now as much his vision and legacy, as the first three editions were mine.

Phil also wishes to thank his wife Nancy and his children Christopher and Charlotte for putting up with the seemingly endless work on this manuscript and too many other projects to mention over these many years.

Seppo: I am grateful to my wife Helena and my sons Sami and Samu for everything we have experienced together. Although it is a tiny gesture compared with all that you have given to me, I humbly dedicate this book to you. And finally, Phil, it was a true pleasure to work with you in this exciting and rewarding book project.

P.A.L.
S.J.O.

1

FUNDAMENTALS OF REAL-TIME SYSTEMS

The term "real time" is used widely in many contexts, both technical and conventional. Most people would probably understand "in real time" to mean "at once" or "instantaneously." *The Random House Dictionary of the English Language* (2nd unabridged edition, 1987), however, defines "realtime" as *pertaining to applications in which the computer must respond as rapidly as required by the user or necessitated by the process being controlled*. These definitions, and others that are available, are quite different, and their differences are often the cause of misunderstanding between computer, software and systems engineers, and the users of real-time systems. On a more pedantic level, there is the issue of the appropriate writing of the term "real-time." Across technical and pedestrian literature, various forms of the term, such as *real time*, *real-time*, and *realtime* may appear. But to computer, software, and systems engineers the preferred form is *real-time*, and this is the convention that we will follow throughout this text.

Consider a computer system in which data need to be processed at a regular rate. For example, an aircraft uses a sequence of accelerometer pulses to determine its position. Systems other than avionic ones may also require a rapid response to events that occur at nonregular rates, such as handling an overtemperature failure in a nuclear power plant. Even without defining the term "real-time," it is probably understood that those events demand timely or "real-time" processing.

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

Now consider a situation in which a passenger approaches an airline check-in counter to pick up his boarding pass for a certain flight from New York to Boston, which is leaving in five minutes. The reservation clerk enters appropriate information into the computer, and a few seconds later a boarding pass is printed. Is this a real-time system?

Indeed, all three systems—aircraft, nuclear power plant, and airline reservations—are real-time, because they must process information within a specified interval or risk system failure. Although these examples may provide an intuitive definition of a real-time system, it is necessary to clearly comprehend when a system is real-time and when it is not.

To form a solid basis for the coming chapters, we first define a number of central terms and correct common misunderstandings in Section 1.1. These definitions are targeted for practitioners, and thus they have a strong practical point-of-view. Section 1.2 presents the multidisciplinary design challenges related to real-time systems. It is shown that although real-time systems design and analysis are subdisciplines of computer systems engineering, they have essential connections to various other fields, such as computer science and electrical engineering—even to applied statistics. It is rather straightforward to present different approaches, methods, techniques, or tools for readers, but much more difficult to convey the authors' insight on real-time systems to the audience. Nevertheless, our intention is to provide some insight in parallel with specific tools for the practitioner. Such insight is built on practical experiences and adequate understanding of the key milestones in the field. The birth of real-time systems, in general, as well as a selective evolution path related to relevant technological innovations, is discussed in Section 1.3. Section 1.4 summarizes the preceding sections on fundamentals of real-time systems. Finally, Section 1.5 provides exercises that help the reader to gain basic understanding on real-time systems and associated concepts.

1.1 CONCEPTS AND MISCONCEPTIONS

The fundamental definitions of real-time systems engineering can vary depending on the resource consulted. Our pragmatic definitions have been collected and refined to the smallest common subset of agreement to form the vocabulary of this particular text. These definitions are presented in a form that is intended to be most useful to the practicing engineer, as opposed to the academic theorist.

1.1.1 Definitions for Real-Time Systems

The hardware of a computer solves problems by repeated execution of machine-language instructions, collectively known as software. Software, on the other hand, is traditionally divided into system programs and application programs.

System programs consist of software that interfaces with the underlying computer hardware, such as device drivers, interrupt handlers, task schedulers, and various programs that act as tools for the development or analysis of application programs. These software tools include compilers, which translate high-level language programs into assembly code; assemblers, which convert the assembly code into a special binary format called object or machine code; and linkers/locators, which prepare the object code for execution in a specific hardware environment. An operating system is a specialized collection of system programs that manage the physical resources of the computer. As such, a real-time operating system is a truly important system program (Anh and Tan, 2009).

Application programs are programs written to solve specific problems, such as optimal hall-call allocation of an elevator bank in a high-rise building, inertial navigation of an aircraft, and payroll preparation for some industrial company. Certain design considerations play a role in the design of system programs and application software intended to run in real-time environments.

The notion of a "system" is central to software engineering, and indeed to all engineering, and warrants formalization.

Definition: System

A system is a mapping of a set of inputs into a set of outputs.

When the internal details of the system are not of particular interest, the mapping function between input and output spaces can be considered as a black box with one or more inputs entering and one or more outputs exiting the system (see Fig. 1.1). Moreover, Vernon lists five general properties that belong to any "system" (Vernon, 1989):

1. A system is an assembly of components connected together in an organized way.
2. A system is fundamentally altered if a component joins or leaves it.
3. It has a purpose.
4. It has a degree of permanence.
5. It has been defined as being of particular interest.

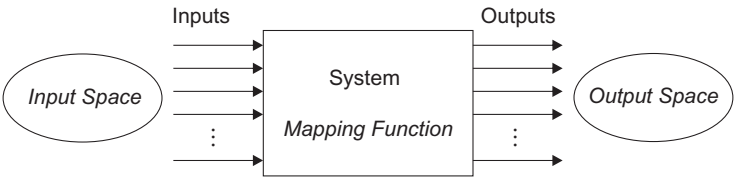


Figure 1.1. A general system with inputs and outputs.

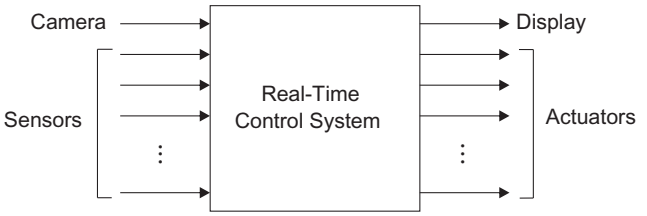


Figure 1.2. A real-time control system including inputs from a camera and multiple sensors, as well as outputs to a display and multiple actuators.

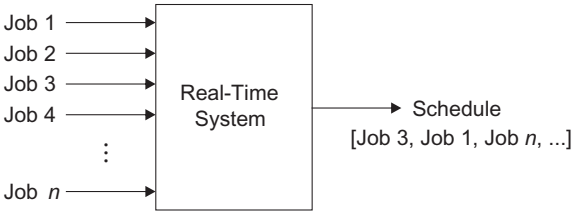


Figure 1.3. A classic representation of a real-time system as a sequence of schedulable jobs.

Every real-world entity, whether organic or synthetic, can be modeled as a system. In computing systems, the inputs represent digital data from hardware devices or other software systems. The inputs are often associated with sensors, cameras, and other devices that provide analog inputs, which are converted to digital data, or provide direct digital inputs. The digital outputs of computer systems, on the other hand, can be converted to analog outputs to control external hardware devices, such as actuators and displays, or used directly without any conversion (Fig. 1.2).

Modeling a real-time (control) system, as in Figure 1.2, is somewhat different from the more traditional model of the real-time system as a sequence of jobs to be scheduled and performance to be predicted, which is comparable with that shown in Figure 1.3. The latter view is simplistic in that it ignores the usual fact that the input sources and hardware under control may be highly complex. In addition, there are other, "sweeping" software engineering considerations that are hidden by the model shown in Figure 1.3.

Look again at the model of a real-time system shown in Figure 1.2. In its realization, there is some inherent delay between presentation of the inputs (excitation) and appearance of the outputs (response). This fact can be formalized as follows:

Definition: Response Time

The time between the presentation of a set of inputs to a system and the realization of the required behavior, including the availability of all associated outputs, is called the response time of the system.

How fast and punctual the response time needs to be depends on the characteristics and purpose of the specific system.

The previous definitions set the stage for a practical definition of a real-time system.

Definition: Real-Time System (I)

A real-time system is a computer system that must satisfy bounded response-time constraints or risk severe consequences, including failure.

But what is a "failed" system? In the case of the space shuttle or a nuclear power plant, for example, it is painfully obvious when a failure has occurred. For other systems, such as an automatic bank teller machine, the notion of failure is less obvious. For now, failure will be defined as the "inability of the system to perform according to system specification," or, more precisely:

Definition: Failed System

A failed system is a system that cannot satisfy one or more of the requirements stipulated in the system requirements specification.

Because of this definition of failure, rigorous specification of the system operating criteria, including timing constraints, is necessary. This matter is discussed later in Chapter 5.

Various other definitions exist for "real-time," depending on which source is consulted. Nonetheless, the common theme among all definitions is that the system must satisfy deadline constraints in order to be correct. For instance, an alternative definition might be:

Definition: Real-Time System (II)

A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness.

In any case, by making unnecessary the notion of timeliness, every system becomes a real-time system.

Real-time systems are often reactive or embedded systems. Reactive systems are those in which task scheduling is driven by ongoing interaction with their environment; for example, a fire-control system reacts to certain buttons pressed by a pilot. Embedded systems can be defined informally as follows:

Definition: Embedded System

An embedded system is a system containing one or more computers (or processors) having a central role in the functionality of the system, but the system is not explicitly called a computer.

For example, a modern automobile contains many embedded processors that control airbag deployment, antilock braking, air conditioning, fuel injection, and so forth. Today, numerous household items, such as microwave ovens, rice cookers, stereos, televisions, washing machines, even toys, contain embedded computers. It is obvious that sophisticated systems, such as aircraft, elevator banks, and paper machines, do contain several embedded computer systems.

The three systems mentioned at the beginning of this chapter satisfy the criteria for a real-time system. An aircraft must process accelerometer data within a certain period that depends on the specifications of the aircraft; for example, every 10 ms. Failure to do so could result in a false position or velocity indication and cause the aircraft to go off-course at best or crash at worst. For a nuclear reactor thermal problem, failure to respond swiftly could result in a meltdown. Finally, an airline reservation system must be able to handle a surge of passenger requests within the passenger's perception of a reasonable time (or before the flights leave the gate). In short, a system does not have to process data at once or instantaneously to be considered real-time; it must simply have response times that are constrained appropriately.

When is a system real-time? It can be argued that all practical systems are ultimately real-time systems. Even a batch-oriented system—for example, grade processing at the end of a semester or a bimonthly payroll run—is real-time. Although the system may have response times of days or even weeks (e.g., the time that elapses between submitting the grade or payroll information and issuance of the report card or paycheck), it must respond within a certain time or there could be an academic or financial disaster. Even a word-processing program should respond to commands within a reasonable amount of time or it will become torturous to use. Most of the literature refers to such systems as soft real-time systems.

Definition: Soft Real-Time System

A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints.

Conversely, systems where failure to meet response-time constraints leads to complete or catastrophic system failure are called hard real-time systems.

Definition: Hard Real-Time System

A hard real-time system is one in which failure to meet even a single deadline may lead to complete or catastrophic system failure.

Firm real-time systems are those systems with hard deadlines where some arbitrarily small number of missed deadlines can be tolerated.

Definition: Firm Real-Time System

> A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete or catastrophic system failure.

As noted, all practical systems minimally represent soft real-time systems. Table 1.1 gives an illustrative sampling of hard, firm, and soft real-time systems.

There is a great deal of latitude for interpretation of hard, firm, and soft real-time systems. For example, in the automated teller machine, missing too many deadlines will lead to significant customer dissatisfaction and potentially even enough loss of business to threaten the existence of the bank. This extreme scenario represents the fact that every system can often be characterized any way—soft, firm, or hard—real-time by the construction of a supporting scenario. The careful definition of systems requirements (and, hence, expectations) is the key to setting and meeting realistic deadline expectations. In any case, it is a principal goal of real-time systems engineering to find ways to transform hard deadlines into firm ones, and firm ones into soft ones.

Since this text is mostly concerned with hard real-time systems, it will use the term real-time system to mean embedded, hard real-time system, unless otherwise noted.

It is typical, in studying real-time systems, to consider the nature of time, because deadlines are instants in time. Nevertheless, the question arises, "Where do the deadlines come from?" Generally speaking, deadlines are based on the underlying physical phenomena of the system under control. For

TABLE 1.1. A Sampling of Hard, Firm, and Soft Real-Time Systems

System	Real-Time Classification	Explanation
Avionics weapons delivery system in which pressing a button launches an air-to-air missile	Hard	Missing the deadline to launch the missile within a specified time after pressing the button may cause the target to be missed, which will result in catastrophe
Navigation controller for an autonomous weed-killer robot	Firm	Missing a few navigation deadlines causes the robot to veer out from a planned path and damage some crops
Console hockey game	Soft	Missing even several deadlines will only degrade performance

example, in animated displays, images must be updated at least 30 frames per second to provide continuous motion, because the human eye can resolve updating at a slower rate. In navigation systems, accelerations must be read at a rate that is a function of the maximum velocity of the vehicle, and so on. In some cases, however, real-world systems have deadlines that are imposed on them, and are based on nothing less than guessing or on some forgotten and possibly eliminated requirement. The problem in these cases is that undue constraints may be placed on the systems. This is a primary maxim of real-time systems design—to understand the basis and nature of the timing constraints so that they can be relaxed if necessary. In cost-effective and robust real-time systems, a pragmatic rule of thumb could be: *process everything as slowly as possible and repeat tasks as seldom as possible*.

Many real-time systems utilize global clocks and time-stamping for synchronization, task initiation, and data marking. It must be noted, however, that all clocks keep somewhat inaccurate time—even the official U.S. atomic clock must be adjusted regularly. Moreover, there is an associated quantization error with clocks, which may need to be considered when using them for time-stamping.

In addition to the degree of "real-time" (i.e., hard, firm, or soft), also, the punctuality of response times is important in many applications. Hence, we define the concept of real-time punctuality:

Definition: Real-Time Punctuality

Real-time punctuality means that every response time has an average value, t_R, with upper and lower bounds of $t_R + \varepsilon_U$ and $t_R - \varepsilon_L$, respectively, and $\varepsilon_U, \varepsilon_L \rightarrow 0^+$.

In all practical systems, the values of ε_U and ε_L are nonzero, though they may be very small or even negligible. The nonzero values are due to cumulative latency and propagation-delay components in real-time hardware and software. Such response times contain jitter within the interval $t \in [-\varepsilon_L, +\varepsilon_U]$. Real-time punctuality is particularly important in periodically sampled systems with high sampling rates, for example, in video signal processing and software radio.

Example: Where a Response Time Comes From

An elevator door (Pasanen et al., 1991) is automatically operated, and it may have a capacitive safety edge for sensing possible passengers between the closing door blades. Thus, the door blades can be quickly reopened before they touch the passenger and cause discomfort or even threaten the passenger's safety.

What is the required system response time from when it recognizes that a passenger is between the closing door blades to the instant when it starts to reopen the door?

This response time consists of five independent components (their presumably measured numerical values are for illustration purpose only):

Sensor Response Time: $t_{S\_min} = 5$ ms, $t_{S\_max} = 15$ ms, $t_{S\_mean} = 9$ ms.

Hardware Response Time: $t_{HW\_min} = 1$ µs, $t_{HW\_max} = 2$ µs, $t_{HW\_mean} = 1.2$ µs.

System Software Response Time: $t_{SS\_min} = 16$ µs, $t_{SS\_max} = 48$ µs, $t_{SS\_mean} = 37$ µs.

Application Software Response Time: $t_{AS\_min} = 0.5$ µs, $t_{AS\_max} = 0.5$ µs, $t_{AS\_mean} = 0.5$ µs.

Door Drive Response Time: $t_{DD\_min} = 300$ ms, $t_{DD\_max} = 500$ ms, $t_{DD\_mean} = 400$ ms.

Now, we can calculate the minimum, maximum, and mean values of the composite response time: $t_{min} \approx 305$ ms, $t_{max} \approx 515$ ms, and $t_{mean} \approx 409$ ms.

Thus, the overall response time is dominated by the door-drive response time containing the required deceleration time of the moving door blades.

In software systems, a change in state results in a change in the flow-of-control of the computer program. Consider the flowchart in Figure 1.4. The decision block represented by the diamond suggests that the stream of program instructions can take one of two alternative paths, depending on the response in question. `case`, `if-then`, and `while` statements in any programming language represent a possible change in flow-of-control. Invocation of procedures in Ada and C represent changes in flow-of-control. In object-oriented

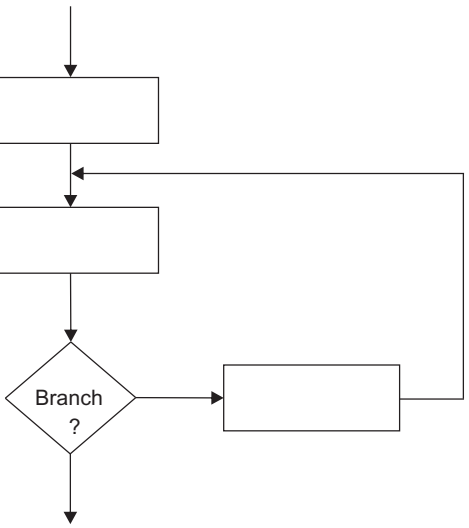


Figure 1.4. A partial program flowchart showing a conditional branch as a change in flow of control.

languages, instantiation of an object or the invocation of a method causes the change in sequential flow-of-control. In general, consider the following definition.

Definition: Event

Any occurrence that causes the program counter to change nonsequentially is considered a change of flow-of-control, and thus an event.

In scheduling theory, the release time of a job is similar to an event.

Definition: Release Time

The release time is the time at which an instance of a scheduled task is ready to run, and is generally associated with an interrupt.

Events are slightly different from jobs in that events can be caused by interrupts, as well as branches.

An event can be either synchronous or asynchronous. Synchronous events are those that occur at predictable times in the flow-of-control, such as that represented by the decision box in the flowchart of Figure 1.4. The change in flow-of-control, represented by a conditional branch instruction, or by the occurrence of an internal trap interrupt, can be anticipated.

Asynchronous events occur at unpredictable points in the flow-of-control and are usually caused by external sources. A real-time clock that pulses regularly at 5 ms is not a synchronous event. While it represents a periodic event, even if the clock were able to tick at a perfect 5 ms without drift, the point where the tick occurs with the flow-of-control is subject to many factors. These factors include the time at which the clock starts relative to the program and propagation delays in the computer system itself. An engineer can never count on a clock ticking exactly at the rate specified, and so any clock-driven event must be treated as asynchronous.

Events that do not occur at regular periods are called aperiodic. Furthermore, aperiodic events that tend to occur very infrequently are called sporadic. Table 1.2 characterizes a sampling of events.

For example, an interrupt generated by a periodic external clock represents a periodic but asynchronous event. A periodic but synchronous event is one

TABLE 1.2. Taxonomy of Events and Some Typical Examples

	Periodic	Aperiodic	Sporadic
Synchronous	Cyclic code	Conditional branch	Divide-by-zero (trap) interrupt
Asynchronous	Clock interrupt	Regular, but not fixed-period interrupt	Power-loss alarm

These items will be discussed further in Chapters 2 and 3.

represented by a sequence of invocation of software tasks in a repeated, circular fashion. A typical branch instruction that is not part of a code block and that runs repeatedly at a regular rate represents a synchronous but aperiodic event. A branch instruction that happens infrequently, say, on the detection of some exceptional condition, is both sporadic and synchronous. Finally, interrupts that are generated irregularly by an external device are classified as either asynchronous aperiodic or sporadic, depending on whether the interrupt is generated frequently or not with respect to the system clock.

In every system, and particularly in an embedded real-time system, maintaining overall control is extremely important. For any physical system, certain states exist under which the system is considered to be out of control; the software controlling such a system must therefore avoid these states. For example, in certain aircraft guidance systems, rapid rotation through a 180° pitch angle can cause loss of gyroscopic control. Hence, the software must be able to anticipate and avert all such scenarios.

Another characteristic of a software-controlled system is that the processor continues to fetch, decode, and execute instructions correctly from the program area of memory, rather than from data or other unwanted memory regions. The latter scenario can occur in poorly tested systems and is a catastrophe from which there is almost no hope of recovery.

Software control of any real-time system and associated hardware is maintained when the next state of the system, given the current state and a set of inputs, is predictable. In other words, the goal is to anticipate how a system will behave in all possible circumstances.

Definition: Deterministic System

A system is deterministic, if for each possible state and each set of inputs, a unique set of outputs and next state of the system can be determined.

Event determinism means the next states and outputs of a system are known for each set of inputs that trigger events. Thus, a system that is deterministic is also event deterministic. Although it would be difficult for a system to be deterministic only for those inputs that trigger events, this is plausible, and so event determinism may not imply determinism.

It is interesting to note that while it is a significant challenge to design systems that are completely event deterministic, and as mentioned, it is possible to inadvertently end up with a system that is nondeterministic, it is definitely hard to design systems that are deliberately nondeterministic. This situation arises from the utmost difficulties in designing perfect random number generators. Such deliberately nondeterministic systems would be desirable, for example, as casino gaming machines.

Finally, if in a deterministic system the response time for each set of outputs is known, then the system also exhibits temporal determinism.

A side benefit of designing deterministic systems is that guarantees can be given that the system will be able to respond at any time, and in the case of temporally deterministic systems, when they will respond. This fact reinforces the association of "control" with real-time systems.

The final and truly important term to be defined is a critical measure of real-time system performance. Because the central processing unit (CPU) continues to fetch, decode, and execute instructions as long as power is applied, the CPU will more or less frequently execute either no-ops or instructions that are not related to the fulfillment of a specific deadline (e.g., noncritical "housekeeping"). The measure of the relative time spent doing nonidle processing indicates how much real-time processing is occurring.

Definition: CPU Utilization Factor

The CPU utilization or time-loading factor, U, is a relative measure of the nonidle processing taking place.

A system is said to be time-overloaded if $U > 100\%$. Systems that are too highly utilized are problematic, because additions, changes, or corrections cannot be made to the system without risk of time-overloading. On the other hand, systems that are not sufficiently utilized are not necessarily cost-effective, because this implies that the system was overengineered and that costs could likely be reduced with less expensive hardware. While a utilization of 50% is common for new products, 80% might be acceptable for systems that do not expect growth. However, 70% as a target for U is one of the most celebrated and potentially useful results in the theory of real-time systems where tasks are periodic and independent—a result that will be examined in Chapter 3. Table 1.3 gives a summary of certain CPU utilizations and typical situations in which they are associated.

U is calculated by summing the contribution of utilization factors for each (periodic or aperiodic) task. Suppose a system has $n \geq 1$ periodic tasks, each with an execution period of p_i, and hence, execution frequency, $f_i = 1/p_i$. If task i is known to have (or has been estimated to have) a *worst-case* execution time of e_i, then the utilization factor, u_i, for task i is

TABLE 1.3. CPU Utilization (%) Zones

Utilization (%)	Zone Type	Typical Application
<26	Unnecessarily safe	Various
26–50	Very safe	Various
51–68	Safe	Various
69	Theoretical limit	Embedded systems
70–82	Questionable	Embedded systems
83–99	Dangerous	Embedded systems
100	Critical	Marginally stressed systems
>100	Overloaded	Stressed systems

$$u_i = e_i / p_i. \tag{1.1}$$

Furthermore, the overall system utilization factor is

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} e_i / p_i. \tag{1.2}$$

Note that the deadline for a periodic task i, d_i, is a critical design factor that is constrained by e_i. The determination of e_i, either prior to, or after the code has been written, can be extremely difficult, and often impossible, in which case estimation or measuring must be used. For aperiodic and sporadic tasks, u_i is calculated by assuming a worst-case execution period, usually the minimum possible time between corresponding event occurrences. Such approximations can inflate the utilization factor unnecessarily or lead to overconfidence because of the tendency to "not worry" about its excessive contribution. The danger is to discover later that a higher frequency of occurrence than budgeted has led to a time-overload and system failure.

The utilization factor differs from CPU throughput, which is a measure of the number of machine-language instructions per second that can be processed based on some predetermined instruction mix. This type of measurement is typically used to compare CPU throughput for a particular application.

Example: Calculation of the CPU Utilization Factor

An individual elevator controller in a bank of high-rise elevators has the following software tasks with execution periods of p_i and worst-case execution times of e_i, $i \in \{1, 2, 3, 4\}$:

Task 1: Communicate with the group dispatcher (19.2 K bit/s data rate and a proprietary communications protocol); $p_1 = 500$ ms, $e_1 = 17$ ms.

Task 2: Update the car position information and manage floor-to-floor runs, as well as door control; $p_2 = 25$ ms, $e_2 = 4$ ms.

Task 3: Register and cancel car calls; $p_3 = 75$ ms, $e_3 = 1$ ms.

Task 4: Miscellaneous system supervisions; $p_4 = 200$ ms, $e_4 = 20$ ms.

What is the overall CPU utilization factor?

$$U = \sum_{i=1}^{4} e_i / p_i = \frac{17}{500} + \frac{4}{25} + \frac{1}{75} + \frac{20}{200} \approx 0.31$$

Hence, the utilization percentage is 31%, which belongs to the "very safe" zone of Table 1.3.

The choice of task deadlines, estimation and reduction of execution times, and other factors that influence CPU utilization will be discussed in Chapter 7.

1.1.2 Usual Misconceptions

As a part of truly understanding the nature of real-time systems, it is important to address a number of frequently cited misconceptions. These are summarized as follows:

1. Real-time systems are synonymous with "fast" systems.
2. Rate-monotonic analysis has solved "the real-time problem."
3. There are universal, widely accepted methodologies for real-time systems specification and design.
4. There is no more a need to build a real-time operating system, because many commercial products exist.
5. The study of real-time systems is mostly about scheduling theory.

The first misconception, that real-time systems must be fast, arises from the fact that many hard real-time systems indeed deal with deadlines in the tens of milliseconds, such as the aircraft navigation system. In a typical food-industry application, however, pasta-sauce jars can move along the conveyor belt past a filling point at a rate of one every five seconds. Furthermore, the airline reservation system could have a deadline of 15 seconds. These latter deadlines are not particularly fast, but satisfying them determines the success or failure of the system.

The second misconception is that rate-monotonic systems provide a simple recipe for building real-time systems. Rate-monotonic systems—a periodic system in which interrupt (or software task) priorities are assigned such that the faster the rate of execution, the higher the priority—have received a lot of attention since the 1970s. While they provide valuable guidance in the design of real-time systems, and while there is abundant theory surrounding them, they are not a panacea. Rate-monotonic systems will be discussed in great detail in Chapter 3.

What about the third misconception? Unfortunately, there are no universally accepted and infallible methods for the specification and design of real-time systems. This is not a failure of researchers or the software industry, but is because of the difficulty of discovering universal solutions for this demanding field. After nearly 40 years of research and development, there is still no methodology available that answers all of the challenges of real-time specification and design all the time and for all applications.

The fourth misconception is that there is no more a need to build a real-time operating system from scratch. While there are a number of cost-effective, popular, and viable commercial real-time operating systems, these, too, are not

a panacea. Commercial solutions have certainly their place, but choosing when to use an off-the-shelf solution and choosing the right one are challenges that will be considered in Chapter 3.

Finally, while it is scholarly to study scheduling theory, from an engineering standpoint, most published results require impractical simplifications and clairvoyance in order to make the theory work. Because this is a textbook for practicing engineers, it avoids any theoretical results that resort to these measures.

1.2 MULTIDISCIPLINARY DESIGN CHALLENGES

The study of real-time systems is a truly multidimensional subdiscipline of computer systems engineering that is strongly influenced by control theory, operations research, and, naturally, software engineering. Figure 1.5 depicts some of the disciplines of computer science, electrical engineering, systems engineering, and applied statistics that affect the design and analysis of real-time systems. Nevertheless, those representative disciplines are not the only ones having a relationship with real-time systems. Because real-time systems engineering is so multidisciplinary, it stands out as a fascinating study area with a rich set of design challenges. Although the fundamentals of real-time systems are well established and have considerable permanence, real-time systems is a lively developing area due to evolving CPU architectures, distributed system structures, versatile wireless networks, and novel applications, for instance.

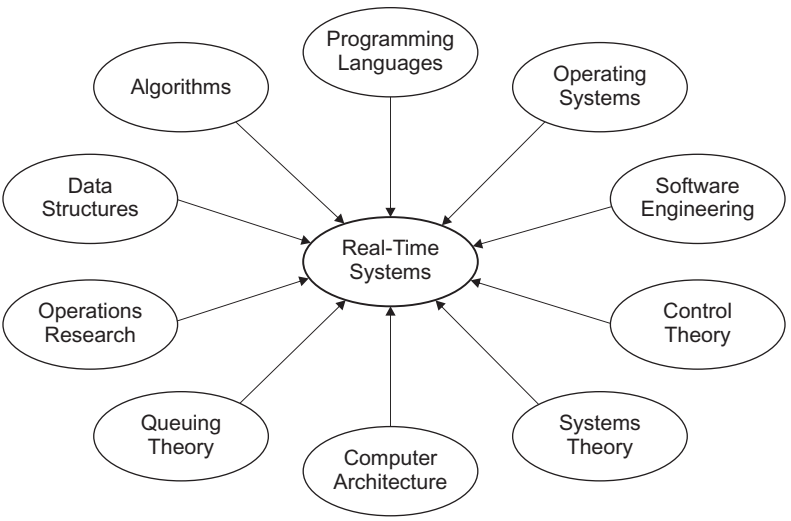


Figure 1.5. A variety of disciplines that affect real-time systems engineering.

1.2.1 Influencing Disciplines

The design and implementation of real-time systems requires attention to numerous practical issues. These include:

- The selection of hardware and system software, and evaluation of the trade-off needed for a competitive solution, including dealing with distributed computing systems and the issues of concurrency and synchronization.
- Specification and design of real-time systems, as well as correct and inclusive representation of temporal behavior.
- Understanding the nuances of the high-level programming language(s) and the real-time implications resulting from their optimized compilation into machine-language code.
- Optimizing (with application-specific objectives) of system fault tolerance and reliability through careful design and analysis.
- The design and administration of adequate tests at different levels of hierarchy, and the selection of appropriate development tools and test equipment.
- Taking advantage of open systems technology and interoperability. An open system is an extensible collection of independently written applications that cooperate to function as an integrated system. For example, several versions of the open operating system, Linux, have emerged for use in various real-time applications (Abbott, 2006). Interoperability can be measured in terms of compliance with open system standards, such as the real-time CORBA (common object request broker architecture) standard (Fay-Wolfe et al., 2000).
- Finally, estimating and measuring response times and (if needed) reducing them. Performing a schedulability analysis, that is, determining and guaranteeing deadline satisfaction, *a priori*.

Obviously, the engineering techniques used for hard real-time systems can be used in the engineering of all other types of systems as well, with an accompanying improvement of performance and robustness. This alone is a significant reason to study the engineering of real-time systems.

1.3 BIRTH AND EVOLUTION OF REAL-TIME SYSTEMS

The history of real-time systems, as characterized by important developments in the United States, is tied inherently to the evolution of the computer. Modern real-time systems, such as those that control nuclear power plants, military weapons systems, or medical monitoring equipment, are sophisticated, yet many still exhibit characteristics of those pioneering systems developed in the 1940s through the 1960s.

1.3.1 Diversifying Applications

Embedded real-time systems are so pervasive and ubiquitous that they are even found in household appliances, sportswear, and toys. A small sampling of real-time domains and corresponding applications is given in Table 1.4. An excellent example of an advanced real-time system is the Mars Exploration Rover of NASA shown in Figure 1.6. It is an autonomous system with extreme reliability requirements; it receives commands and sends measurement data over radio-communications links; and performs its scientific missions with the aid of multiple sensors, processors, and actuators.

In the introductory paragraphs of this chapter, some real-time systems were mentioned. The following descriptions provide more details for each system, while others provide additional examples. Clearly, these descriptions are not rigorous specifications. The process of specifying real-time systems unambiguously but concisely is discussed in Chapter 5.

Consider the inertial measurement system for an aircraft. The software specification states that the software will receive x, y, and z accelerometer pulses at a 10 ms rate from special hardware. The software will determine the acceleration components in each direction, and the corresponding roll, pitch, and yaw of the aircraft.

The software will also collect other information, such as temperature at a 1-second rate. The task of the application software is to compute the actual velocity vector based on the current orientation, accelerometer readings, and various compensation factors (such as for temperature effects) at a 40 ms rate. The system is to output true acceleration, velocity, and position vectors to a pilot's display every 40 ms, but using a different clock.

TABLE 1.4. Typical Real-Time Domains and Diverse Applications

Domain	Applications
Aerospace	Flight control
	Navigation
	Pilot interface
Civilian	Automotive systems
	Elevator control
	Traffic light control
Industrial	Automated inspection
	Robotic assembly line
	Welding control
Medical	Intensive care monitors
	Magnetic resonance imaging
	Remote surgery
Multimedia	Console games
	Home theaters
	Simulators

Figure 1.6. Mars Exploration Rover; a solar-powered, autonomous real-time system with radio-communications links and a variety of sensors and actuators. Photo courtesy of NASA.

These tasks execute at four different rates in the inertial measurement system, and need to communicate and synchronize. The accelerometer readings must be time-relative or correlated; that is, it is not allowed to mix an x accelerometer pulse of discrete time instant k with y and z pulses of instant $k + 1$. These are critical design issues for this system.

Next, consider a monitoring system for a nuclear power plant that will be handling three events signaled by interrupts. The first event is triggered by any of several signals at various security points, which will indicate a security breach. The system must respond to this signal within one second. The second and most important event indicates that the reactor core has reached an overtemperature. This signal must be dealt with within 1 millisecond (1 ms). Finally, an operator's display is to be updated at approximately 30 times per second. The nuclear-power-plant system requires a reliable mechanism to ensure that the "meltdown imminent" indicator can interrupt any other processing with minimal latency.

As another example, recall the airline reservation system mentioned earlier. Management has decided that to prevent long lines and customer dissatisfaction, turnaround time for any transaction must be less than 15 seconds, and no overbooking will be permitted. At any time, several travel agents may try to access the reservations database and perhaps book the same flight simultaneously. Here, effective record-locking and secure communications mechanisms

are needed to protect against the alteration of the database containing the reservation information by more than one clerk at a time.

Now, consider a real-time system that controls all phases of the bottling of jars of pasta sauce as they travel along a conveyor belt. The empty jars are first microwaved to disinfect them. A mechanism fills each jar with a precise serving of specific sauce as it passes beneath. Another station caps the filled bottles. In addition, there is an operator's display that provides an animated rendering of the production line activities. There are numerous events triggered by exceptional conditions, such as the conveyor belt jamming and a bottle overflowing or breaking. If the conveyor belt travels too fast, the bottle will move past its designated station prematurely. Therefore, there is a wide range of events, both synchronous and asynchronous, to be dealt with.

As a final example, consider a system used to control a set of traffic lights at a four-way traffic intersection (north-, south-, east-, and west-bound traffic). This system controls the lights for vehicle and pedestrian traffic at a four-way intersection in a busy city like Philadelphia. Input may be taken from cameras, emergency-vehicle transponders, push buttons, sensors under the ground, and so on. The traffic lights need to operate in a synchronized fashion, and yet react to asynchronous events—such as a pedestrian pressing a button at a crosswalk. Failure to operate in a proper fashion can result in automobile accidents and even fatalities.

The challenge presented by each of these systems is to determine the appropriate design approach with respect to the multidisciplinary issues discussed in Section 1.2.

1.3.2 Advancements behind Modern Real-Time Systems

Much of the theory of real-time systems is derived from the surrounding disciplines shown in Figure 1.5. In particular, certain aspects of operations research (i.e., scheduling), which emerged in the late 1940s, and queuing theory in the early 1950s, have influenced most of the more theoretical results.

Martin published one of the first and certainly the most influential early book on real-time systems (Martin, 1967). Martin's book was soon followed by several others (e.g., Stimler, 1969), and the influence of operations research and queuing theory can be seen in these works. It is also educational to study these texts in the context of the great limitations of the hardware of the time.

In 1973, Liu and Layland published their seminal work on rate-monotonic theory (Liu and Layland, 1973). Over the last nearly 40 years, significant refinement of this theory has made it a practical theory for use in designing real-time systems.

The 1980s and 1990s saw a proliferation of theoretical work on improving predictability and reliability of real-time systems, and on solving problems related to multitasking systems. Today, a rather small group of experts continues to study pure issues of scheduling and performance analysis, while a larger group of generalist systems engineers tackles broader issues relating to the

implementation of practical systems. An important paper by Stankovic et al. (Stankovic et al., 1995) described some of the difficulties in conducting research on real-time systems—even with significant restriction of the system, most problems relating to scheduling are too difficult to solve by analytic techniques.

Instead of any single "groundbreaking" technology, the new millennium saw a number of important advancements in hardware, viable open-source software for real-time systems, powerful commercial design and implementation tools, and expanded programming language support. These advancements have in some ways simplified the construction and analysis of real-time systems but on the other hand introduced new problems because of the complexities of systems interactions and the masking of many of the underlying subtleties of time constraints.

The origin of the term *real-time computing* is unclear. It was probably first used either with project Whirlwind, a flight simulator developed by IBM for the U.S. Navy in 1947, or with SAGE, the Semiautomatic Ground Environment air defense system developed for the U.S. Air Force in the late 1950s. Both of these projects qualify as real-time systems even by today's definitions. In addition to its real-time contributions, the Whirlwind project included the first use of ferrite core memory ("fast") and a form of high-level language compiler that predated Fortran.

Other early real-time systems were used for airline reservations, such as SABRE (developed for American Airlines in 1959), as well as for process control, but the advent of the national space program provided even greater opportunities for the development of more advanced real-time systems for spacecraft control and telemetry. It was not until the 1960s that rapid development of such systems took place, and then only as significant nonmilitary interest in real-time systems become coupled with the availability of equipment adapted to real-time processing.

Low-performance processors and particularly slow and small memories handicapped many of the earliest systems. In the early 1950s, the asynchronous interrupt was introduced and later incorporated as a standard feature in the Univac Scientific 1103A. The middle 1950s saw a distinct increase in the speed and complexity of large-scale computers designed for scientific computation, without an increase in physical size. These developments made it possible to apply real-time computation in the field of control systems. Such hardware improvements were particularly noticeable in IBM's development of SAGE.

In the 1960s and 1970s, advances in integration levels and processing speeds enhanced the spectrum of real-time problems that could be solved. In 1965 alone, it was estimated that more than 350 real-time process control systems existed (Martin, 1967).

The 1980s and 1990s have seen, for instance, distributed systems and non-von Neumann architectures utilized in real-time applications.

Finally, the late 1990s and early 2000s have set new trends in real-time embedded systems in consumer products and Web-enabled devices. The avail-

ability of compact processors with limited memory and functionality has rejuvenated some of the challenges faced by early real-time systems designers. Fortunately, around 60 years of experience is now available to draw upon.

Early real-time systems were written directly in microcode or assembly language, and later in higher-level languages. As previously noted, Whirlwind used an early form of high-level language called an algebraic compiler to simplify coding. Later systems employed Fortran, CMS-2, and JOVIAL, the preferred languages in the U.S. Army, Navy, and Air Force, respectively.

In the 1970s, the Department of Defense (DoD) mandated the development of a single language that all military services could use, and that provided high-level language constructs for real-time programming. After a careful selection and refinement process, the Ada language appeared as a standard in 1983. Shortfalls in the language were identified, and a new, improved version of the language, Ada 95, appeared in 1995.

Today, however, only a small number of systems are developed in Ada. Most embedded systems are written in C or C++. In the last 10 years, there has been a remarkable increase in the use of object-oriented methodologies, and languages like C++ and Java in embedded real-time systems. The real-time aspects of programming languages are discussed later in Chapter 4.

The first commercial operating systems were designed for the early mainframe computers. IBM developed the first real-time executive, the Basic Executive, in 1962, which provided diverse real-time scheduling. By 1963, the Basic Executive II had disk-resident system and user programs.

By the mid-1970s, more affordable minicomputer systems could be found in many engineering environments. In response, a number of important real-time operating systems were developed by the minicomputer manufacturers. Notable among these were the Digital Equipment Corporation (DEC) family of real-time multitasking executives (RSX) for the PDP-11, and Hewlett-Packard's Real-Time Executive (RTE) series of operating systems for its HP 2000 product line.

By the late 1970s and early 1980s, the first real-time operating systems for microprocessor-based applications appeared. These included RMX-80, MROS 68K, VRTX, and several others. Over the past 30 years, many commercial real-time operating systems have appeared, and many have disappeared.

A selective summary of landmark events in the field of real-time systems in the United States is given in Table 1.5.

1.4 SUMMARY

The deep-going roots of real-time systems were formed during the historical years of computers and computing—before the microprocessor era. However, the first "boom" of real-time systems took place around the beginning of 1980s, when appropriate microprocessors and real-time operating systems became

TABLE 1.5. Landmarks in Real-Time Systems History in the United States

Year	Landmark	Developer	Development	Innovations
1947	Whirlwind	IBM	Flight simulator	Ferrite core memory ("fast"), high-level language
1957	SAGE	IBM	Air defense	Designed for real-time
1958	Scientific 1103A	Univac	General purpose	Asynchronous interrupt
1959	SABRE	IBM	Airline reservation	"Hub-go-ahead" policy
1962	Basic Executive	IBM	General purpose	Diverse real-time scheduling
1963	Basic Executive II	IBM	General purpose	Disk-resident system/user programs
1970s	RSX, RTE	DEC, HP	Real-time operating systems	Hosted by minicomputers
1973	Rate-monotonic system	Liu and Layland	Fundamental theory	Upper bound on utilization for schedulable systems
1970s and 1980s	RMX-80, MROS 68K, VRTX, etc.	Various	Real-time operating systems	Hosted by microprocessors
1983	Ada 83	U.S. DoD	Programming language	For mission-critical embedded systems
1995	Ada 95	Community	Programming language	Improved version of Ada 83
2000s	–	–	Various advances in hardware, open-source, and commercial system software and tools	A continuously growing range of innovative applications that can be "real-time"

available (to be used in embedded systems) for an enormous number of electrical, systems, as well as mechanical and aerospace engineers. These practicing engineers did not have much software or even computer education, and, thus, the initial learning path was laborious in most fields of industry. In those early times, the majority of real-time operating systems and communications proto-

cols were proprietary designs—applications people were developing both system and application software themselves. But the situation started to improve with the introduction of more effective high-level language compilers, software debugging tools, communications standards, and, gradually, also methodologies and associated tools for professional software engineering.

What is left from those pioneering years approximately 30 years ago? Well, the foundation of real-time systems is still remarkably the same. The core issues, such as the different degrees of real-time and deterministic requirements, as well as real-time punctuality, are continuing to set major design challenges. Besides, the basic techniques of multitasking and scheduling, and the accompanying inter-task communication and synchronization mechanisms, are used even in modern real-time applications. Hence, real-time systems knowledge has a long lifetime. Nonetheless, much fruitful development is taking place in real-time systems engineering worldwide: new specification and design methods are introduced; innovative processor and system architectures become available and practical; flexible and low-cost wireless networks gain popularity; and numerous novel applications appear continuously, for example, in the field of ubiquitous computing.

We can fairly conclude that real-time systems engineering is a sound and timely topic for junior-senior level, graduate, and continuing education; and it offers growing employment potential in various industries. In the coming chapters, we will cover a broad range of vital themes for practicing engineers (see Fig. 1.7). While the emphasis is on software issues, the fundamentals of real-time hardware are carefully outlined as well. Our aim is to provide a comprehensive text to be used also in industrial settings for new real-time system designers, who need to get "up to speed" quickly. That aim is highlighted in this fourth edition of *Real-Time Systems Design and Analysis*, with the descriptive subtitle *Tools for the Practitioner*.

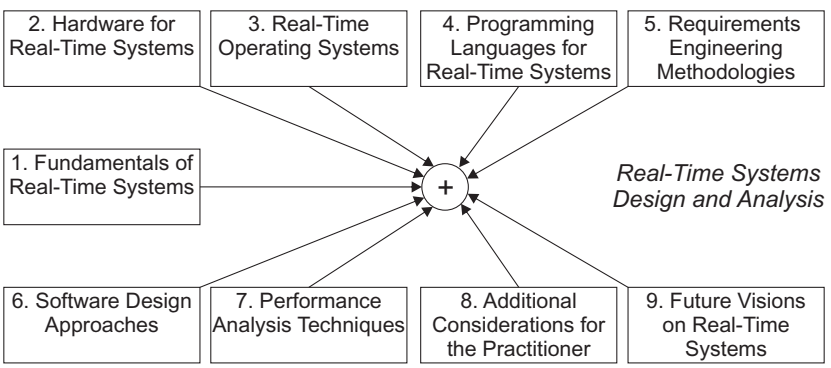


Figure 1.7. Composition of this unique text from nine complementary chapters.

1.5 EXERCISES

1.1. Consider a payroll processing system for an elevator company. Describe three different scenarios in which the system can be justified as hard, firm, or soft real-time.

1.2. Discuss whether the following are hard, firm, or soft real-time systems:

(a) The Library of Congress print-manuscript database system.
(b) A police database that provides information on stolen automobiles.
(c) An automatic teller machine in a shopping mall.
(d) A coin-operated video game in some amusement park.
(e) A university grade-processing system.
(f) A computer-controlled routing switch used at a telephone company branch exchange.

1.3. Consider a real-time weapons control system aboard a fighter aircraft. Discuss which of the following events would be considered synchronous and which would be considered asynchronous to the real-time computing system.

(a) A 5-ms, externally generated clock interrupt.
(b) An illegal-instruction-code (trap) interrupt.
(c) A built-in-test memory failure.
(d) A discrete signal generated by the pilot pushing a button to fire a missile.
(e) A discrete signal indicating "low on fuel."

1.4. Describe a system that is completely nonreal-time, that is, there are no bounds whatsoever for any response time. Do such systems exist in reality?

1.5. For the following systems concepts, fill in the cells of Table 1.2 with descriptors for possible events. Estimate event periods for the periodic events.

(a) Elevator group dispatcher: this subsystem makes optimal hall-call allocation for a bank of high-speed elevators that service a 40-story building in a lively city like Louisville.
(b) Automotive control: this on-board crash avoidance system uses data from a variety of sensors and makes decisions and affects behavior to avoid collision, or protect the occupants in the event of an imminent collision. The system might need to take control of the automobile from the driver temporarily.

1.6. For the real-time systems in Exercise 1.2, what are reasonable response times for all those events?

1.7. For the example systems introduced (inertial measurement, nuclear-power-plant monitoring, airline reservation, pasta bottling, and traffic-light control) enumerate some possible events and note whether they are periodic, aperiodic, or sporadic. Discuss reasonable response times for the events.

1.8. In the response-time example of Section 1.1, the time from observing a passenger between the closing door blades and starting to reopen the elevator door varies between 305 and 515 ms. How could you further justify if these particular times are appropriate for this situation?

1.9. A control system is measuring its feedback quantity at the rate of 100 μs. Based on the measurement, a control command is computed by a heuristic algorithm that uses complex decision making. The new command becomes available 27–54 μs (rather evenly distributed) after each sampling moment. This considerable jitter introduces harmful distortion to the controller output. How could you avoid (reduce) such a jitter? What (if any) are the drawbacks of your solution?

1.10. Reconsider the CPU utilization factor example of Section 1.1. How short could the execution period of Task 1, e_1, be made to maintain the CPU utilization zone no worse than "questionable" (Table 1.3)?

REFERENCES

D. Abbott, *Linux for Embedded and Real-Time Applications*, 2nd Edition. Burlington, MA: Newnes, 2006.

T. N. B. Anh and S.-L. Tan, "Real-time operating systems for small microcontrollers," *IEEE Micro*, 29(5), pp. 30–45, 2009.

V. Fay-Wolfe et al., "Real-time CORBA," *IEEE Transactions on Parallel and Distributed Systems*, 11(10), pp. 1073–1089, 2000.

C. L. Liu and J. W. Layland, "Scheduling algorithms for multi-programming in a hard real-time environment," *Journal of the ACM*, 20(1), pp. 46–61, 1973.

J. Martin, *Design of Real-Time Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1967.

J. Pasanen, P. Jahkonen, S. J. Ovaska, H. Tenhunen, and O. Vainio, "An integrated digital motion control unit," *IEEE Transactions on Instrumentation and Measurement*, 40(3), pp. 654–657, 1991.

J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo, "Implications of classical scheduling results for real-time systems," *IEEE Computer*, 28(6), pp. 16–25, 1995.

S. Stimler, *Real-Time Data-Processing Systems*. New York: McGraw-Hill, 1969.

P. Vernon, "Systems in engineering," *IEE Review*, 35(10), pp. 383–385, 1989.

2

HARDWARE FOR REAL-TIME SYSTEMS

There is an obvious need for basic hardware understanding among software and system engineers, particularly when embedded real-time systems are designed or analyzed. This chapter provides a focused introduction to fundamental hardware-related issues from the real-time point of view. Hence, it also forms a useful overview for hardware-oriented practitioners. In real-time systems, multi-step and time-variant delay paths from inputs (excitations) to outputs (responses) create considerable timing and latency challenges that should be understood and properly managed, for example, when designing real-time software or integrating software with hardware. Such challenges are naturally of different complexity and importance depending on the specific application we are dealing with. There is a wide array of large-scale and more compact real-time applications from global airline reservation and booking systems to emerging ubiquitous computing. In the same way, the hardware platforms may vary considerably from networked multi-core workstations to single 8-bit or even 4-bit microcontrollers. While the hardware-specific issues are rather abstract for application programmers developing software for workstation environments, they are truly concrete for system programmers and individuals working with embedded microcontrollers or digital signal processors.

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

Computing hardware together with networking solutions are dynamic fields of research and development; advanced processor architectures—even reconfigurable ones—and high-speed wireless networks are providing exciting opportunities for product innovators and designers. However, such hardware advancements make it typically harder to achieve *real-time punctuality*; in many cases, the mean performance is greatly improved, but the statistical response-time distributions become significantly wider. This is particularly true with the latest processor architectures, memory hierarchies, distributed system configurations, and ultra-low power constraints. In addition, real-time operating systems are major sources of similar uncertainty. The increasing uncertainty in response times may degrade the robustness and performance, for instance, in time-critical control systems with high sampling rates. Of course, it is always necessary to ask if real-time punctuality is really needed, and the answer depends solely on the nature of the application—whether it must behave as a hard, firm, or soft real-time system. The different degrees or strengths of "real-time" are defined in Chapter 1.

In Section 2.1, we first give an introduction to a basic processor architecture, the *rudimentary* von Neumann architecture. Implementations of this reference architecture are setting a baseline for achievable real-time punctuality. Next, memory hierarchies and their essential contributions to response-time uncertainties are discussed in Section 2.2. Section 2.3 presents the widespread advancements in processor architectures. While in most of the cases, remarkable performance improvements are attained compared with the reference architecture, the worst-case real-time punctuality degrades drastically with multi-stage and multiple pipelines. Peripheral interfacing techniques and interrupt processing alternatives are discussed in Section 2.4. The emphasis of that discussion is on latency and priority issues. Section 2.5 compares two computing platforms, microprocessor and microcontroller, from the applications point of view. An introductory discussion on fieldbus systems and time-triggered architectures is provided in Section 2.6. Those distributed and heterogeneous systems may have strict timing specifications, which could benefit from fault-tolerant clock synchronization within all nodes. Section 2.7 summarizes the preceding sections on real-time hardware. Finally, Section 2.8 provides a rich collection of exercises on real-time hardware.

While this chapter emphasizes the specific real-time characteristics of processor architectures and peripheral interfacing techniques, more general presentations on computer architectures and interfacing are available in the "classic" books, Hennessy and Patterson (2007) and Garrett (2000), respectively.

2.1 BASIC PROCESSOR ARCHITECTURE

In the following subsections, we first present a basic processor architecture and define some principal terminology on computer architectures, instruction processing, and input/output (I/O) organizations. That introduction forms a

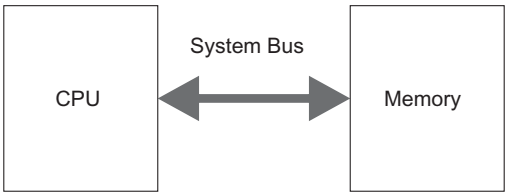


Figure 2.1. Von Neumann computer architecture without an explicit input/output element.

sound basis for latter sections of this chapter devoted to architectural and other hardware advancements.

2.1.1 Von Neumann Architecture

The traditional von Neumann computer architecture, also known as the Princeton architecture, is used in numerous commercial processors and can be depicted with only three elements: a central processing unit (CPU), a system bus, and memory. Figure 2.1 illustrates such an architecture, where the CPU is connected through the system bus to the memory. In a more detailed view, the system bus is actually a set of three individual buses: address, data, and control. Of those parallel buses, the address bus is unidirectional and controlled by the CPU; the data bus is bidirectional and transfers instructions as well as data; and the control bus is a heterogeneous collection of independent control, status, clock, and power lines. A processor in a real-time application has a group of 4, 8, 16, 24, 32, 64, or even more data lines that collectively form the data bus. On the other hand, the width of the address bus is usually between 16 and 32 bits. In the basic von Neumann architecture, the I/O registers are said to be memory mapped, because they are accessed in the same way as regular memory locations. As an example of the many implementation options for practical von Neumann computers, the data bus protocol can be either synchronous or asynchronous; the former providing simpler implementation structure, and the latter one being more flexible with respect to different access times of memory and I/O devices.

The CPU is the core unit where instruction processing takes place; it consists of a control unit, an internal bus, and a datapath, as illustrated in Figure 2.2. Moreover, the datapath contains a multi-function arithmetic-logic unit (ALU), and a bank of work registers, as well as a status register. The control unit interfaces to the system bus through a program counter register (PCR) that addresses the external memory location from which the next instruction is going to be fetched to an instruction register (IR). Each fetched instruction is first decoded in the control unit, where the particular instruction code is identified. After identifying the instruction code, the control unit commands the datapath appropriately like a Mealy-type finite state machine; an operand is loaded from memory, a specific ALU function is activated with a set of

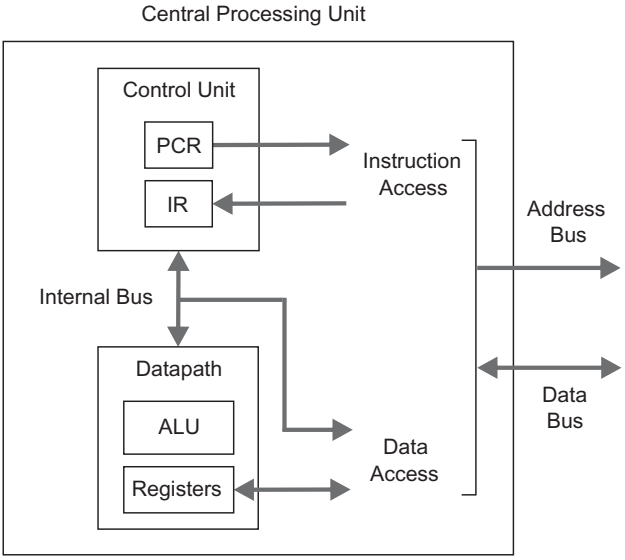


Figure 2.2. Internal structure of a simplified CPU. The *Instruction Access* and *Data Access* are merged pairwise to form the common address and data buses.

operands, and its result is finally stored to memory. While integer data can usually be stored in 1, 2, or 4 bytes, floating-point quantities typically occupy 4 or more bytes of memory. The bank of work registers forms a fast interface buffer between the ALU and memory. Status register's individual bits or flags are updated according to the result of previous ALU operation and current CPU state. Specific status flags, such as "zero" and "carry/borrow," are used for implementing conditional branch instructions and extended-precision additions/subtractions. There is an internal clock and other signals used for timing and data transfer, and numerous hidden registers that are found inside the CPU, but are not shown in Figure 2.2.

This architectural framework offers several design parameters to be tailored for application-specific requirements and implementation constraints: instruction set, control unit, ALU functions, size of the register bank, bus widths, and clock rate. Although the von Neumann architecture is used commonly in various processors, it is sometimes considered a serious limitation that instructions and data are accessible only sequentially using the single system bus. On the other hand, such a straightforward bus structure is compact to implement.

2.1.2 Instruction Processing

Instruction processing consists of multiple consecutive phases taking a varying number of clock cycles to complete. These independent phases form jointly an

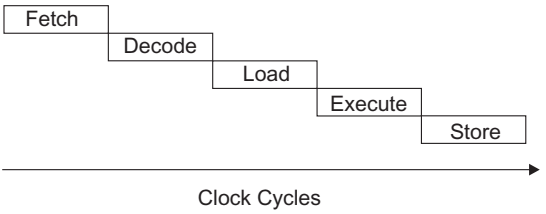


Figure 2.3. Sequential instruction cycle with five phases.

instruction cycle. In this text, we assume a five-phase instruction cycle: *Fetch* instruction, *Decode* instruction, *Load* operand, *Execute* ALU function, and *Store* result. Figure 2.3 shows a timing diagram of the sequential instruction cycle. The duration of an instruction cycle depends on the instruction itself; multiplication is typically more time-consuming than a simple register-to-register move. In addition, not all instructions need active Load, Execute, and/or Store phases, but those missing phases are either skipped or filled with idle clock cycles.

Every instruction is represented by its unique binary code that is stored in the memory, and a stream of such codes forms a machine-language program. In the following paragraphs, however, we are going to use mnemonic codes for instructions instead of binary codes. These mnemonic instruction codes are better known as assembly-language instructions, and they exist just for making our life easier—the CPU uses binary codes only. To understand the specifics of instruction processing, it is beneficial to give a brief introduction to assembly-language instructions. We can say that an instruction set describes a processor's functionality. It is also intimately connected to the processor's architecture.

When dealing with assembly-language programming, we have to know the existing instruction set, as well as the available addressing modes and work registers. A generic instruction has the following format:

```
op-code operand_1, operand_2, operand_3
```

Here, the `op-code` represents an assembly-language instruction code, which is followed by three operands. It should be noted that the physical length of an entire instruction (in bytes or words) depends on the number of operands. And instructions with integer operands are in general preferable over instructions with floating-point operands. Practical instructions may have one, two, three, or no operands, depending on their function. Illustrative examples of such instructions are given below (the mnemonic codes here are typical and vary from one processor to another):

```
INC  R1          ; Increment the content of work register R1.
ADD  R1, R2      ; Add the contents of R1 and R2, and store the sum
                   to R1.
```

```
SUB R1, R2, R3   ; Subtract the content of R3 from R2, and store the
                   result to R1.
NOP              ; No operation, just increment the program counter
                   register.
```

Depending on the maximum number of operands, a particular processor is said to have a one-address form, two-address form, or three-address form. In the above case, all the operands refer to the contents of certain work registers. Hence, this addressing mode is called *register direct*. To make it convenient to implement common data structures, such as vectors and matrices, some kind of indirect or indexed addressing mode is usually available. *Register indirect* addressing mode uses one of the work registers for storing a pointer (address) to a data structure located in memory:

```
ADD R1, [R2]     ; Add the content of R1 to the content of a memory
                   location that is pointed by the address in R2, and store
                   the sum to R1.
```

In addition to the basic register-direct and register-indirect addressing modes, every processor offers at least *direct* and *immediate* addressing modes as well:

```
INC &memory      ; Increment the content of memory location memory.
ADD R1, 5        ; Add the content of R1 to number 5, and store the sum
                   to R1.
```

Real-world processors have often a moderate collection of addressing modes and a comprehensive set of instructions. This evidently leads to considerable burden in the instruction-decoding phase, because every instruction with a different addressing mode is considered as an individual instruction when the instruction code is identified. For instance, the single `ADD` instruction is seen as four individual instructions if the four addressing modes that were discussed above are available. After identifying the instruction code, the control unit creates an appropriate command sequence for executing that instruction.

There are two principal techniques for implementing the control unit: microprogramming and hard-wired logic. In microprogramming, every instruction is defined by a microprogram consisting of a sequence of primitive hardware commands, microinstructions, for activating appropriate datapath functions and complementary suboperations. It is, in principle, straightforward to construct machine-language instructions by microprogramming, but such microinstruction sequences tend to use several clock cycles. This may become an obstacle with complicated instructions, which would require a relatively large number of clock cycles. Users of commercial processors do not have access to the microprogram memory, but it is configured permanently by those

who implement the instruction set. In processors with either a small instruction set or a demand for very fast instruction processing, the control unit is regularly implemented using hard-wired logic that consists of combinatorial and sequential digital circuits. This low-level implementation alternative takes more space per instruction compared with microprogramming, but it can offer noticeably faster instruction execution. Nonetheless, it is more difficult to create or modify machine-language instructions when a hard-wired control unit is used. As we will see later when advanced processor architectures are discussed, both microprogramming and hard-wired logic are used widely in modern commercial processors. This situation is mainly an implementation issue related to the size and complexity of the instruction set.

In previous paragraphs, we presented the basic instruction-processing principle that consists of five consecutive phases and no parallelism. Actually, it appears to rely on implicit thinking that consecutive instructions in a program have both internal and mutual dependency, which prevents any kind of instruction-level parallelism. This unrealistic constraint makes the utilization rate of ALU resources poor, and, therefore, it is relieved significantly in advanced computer architectures. Even with this reference architecture, it is still possible to increase computing performance by using wide internal and system buses, high clock rate, and a large bank of work registers to reduce the need for (slower) external memory access. These straightforward enhancements have a direct connection to hardware constraints: desired dimensions of the integrated circuit and the used fabrication technology. Besides, from the real-time systems viewpoint (response times and their punctuality), such enhancements are all well behaving.

To conserve energy and make software "green," many modern processors have a slowdown mode. Specific instructions can lower the circuit voltage and clock frequency, thereby slowing the computer and using less power and generating less heat. The use of this feature is especially challenging to real-time designers, who must worry about meeting deadlines and the variation in task execution time.

2.1.3 Input/Output and Interrupt Considerations

Every computer system needs input and output ports to bring in excitations and to feed out corresponding responses. There always exists some interaction between a computer and its operating environment or users. This relationship is of paramount importance in embedded systems. In real-time systems, such I/O actions have a critical requirement; they are often strictly time-constrained.

The von Neumann architecture of Figure 2.1 does not contain any I/O block, but the input and output registers are assumed to exist in the regular memory space—inside the memory block. Therefore, from the CPU's viewpoint, those I/O-specific registers form tiny memory segments with only a few pseudo-memory locations corresponding, for instance, to mode, status, and

data registers of configurable I/O ports. With memory-mapped I/O, I/O ports can be operated through all instructions that have a memory operand. This could be advantageous when implementing efficient device drivers.

Programmed I/O is a commonly used alternative to memory-mapped I/O. In this scheme, a slightly enhanced bus architecture offers a separate address space for I/O registers; the standard system bus is still used as an interface between the CPU and I/O ports, but now there is an additional control signal "Memory/IO" for distinguishing between memory and I/O accesses. Moreover, separate `IN` and `OUT` instructions are needed for accessing the I/O registers. There are different practices for realizing such instructions, but the following example shows a typical case where a certain work register is used for keeping the I/O data:

```
IN R1, &port   ; Read the content of port and store it to R1.
OUT &port, R1  ; Write the content of R1 to port.
```

In low-end microcontrollers with a small address space, the main advantage of programmed I/O is the saving of limited address space for memory components only. On the other hand, in high-end microcontrollers and powerful microprocessors, it is beneficial to place the slower I/O ports in a different address space than the faster memory components. In that way, no compromises are needed when specifying the speed of system bus. Figure 2.4 depicts the enhanced von Neumann architecture with separate memory and I/O elements.

An interrupt is an external hardware signal that initiates an event. Interrupts are used to signal that an I/O transfer was completed or needs to be initiated. While I/O ports are crucial for any computer system, it can be stated that interrupts are crucial at least in any hard real-time system. Hardware inter-

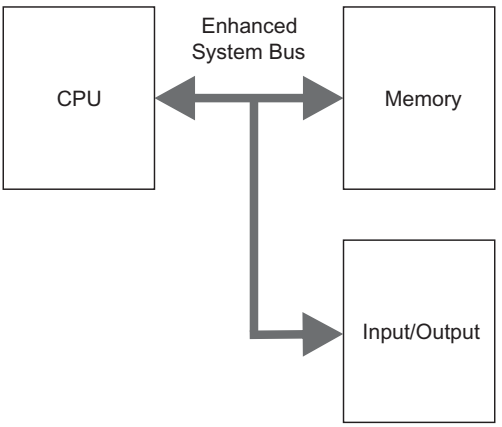


Figure 2.4. Von Neumann architecture with slightly enhanced system bus for programmed input/output.

rupts make it possible to give prompt service to important events occurring in the operating environment. The interrupt principle works fine as long as the number of (nearly) simultaneous interrupt requests is very low and the corresponding interrupt processing times are very short. Therefore, it should be carefully planned which devices or sensors are given a right to interrupt. Only the most time-critical events deserve such a privilege, as other events could possibly be recognized by periodic polling instead. In general, there exist two types of hardware interrupts: maskable interrupts and nonmaskable interrupts. Maskable interrupts are commonly used for such events that occur during regular operating conditions; and nonmaskable interrupts are reserved for extremely critical events that require immediate action, such as an alarm of a rapidly approaching power loss.

Although interrupts are often associated with truly prompt service, there are a few latency elements in the interrupt recognition and service process. These obviously reduce the real-time punctuality and make the response times somewhat nondeterministic. A typical interrupt service process is as follows:

- The interrupt-request line is activated.
- The interrupt request is latched by the CPU hardware (~).
- The processing of the ongoing instruction is completed (~).
- The content of program counter register (PCR) is pushed to stack.
- The content of status register (SR) is pushed to stack.
- The PCR is loaded with the interrupt handler's address.
- The interrupt handler is executed (~).
- The original content of SR is popped from stack.
- The original content of PCR is popped from stack.

The three specific steps of this interrupt-service process, denoted with a tilde, are sources of *variable-length* latency. While interrupt-request latching takes often no more than a single clock cycle, it may require any time between zero and the maximum length of the instruction cycle to complete the ongoing instruction. And the execution of the interrupt handler code needs naturally multiple instruction cycles. Solely in rare instances, certain *block-oriented* instructions, such as memory-to-memory block moves that take a great deal of time to complete, may need to be interruptible to reduce interrupt latency. However, interrupting such instructions could potentially lead to serious data integrity problems. Latency uncertainties are becoming even more severe with the advancement of computer and memory architectures. Thus, it can be concluded that the rudimentary von Neumann architecture with purely sequential instruction processing sets a baseline for real-time punctuality from the hardware point of view.

Processors provide two instructions for enabling and disabling maskable interrupts, which we will call enable priority interrupt (`EPI`) and disable priority interrupt (`DPI`), respectively. These are atomic instructions that should,

however, be used cautiously in real-time applications, because real-time punctuality may be severely compromised when interrupts are disabled. It is recommended to allow the use of `EPI` and `DPI` by system programmers only, and deny their usage totally from application programmers.

Finally, it should be noted that not all interrupts are initiated externally, but the CPU may have a special instruction for initiating software interrupts itself. Software interrupts are convenient when creating operating system services and device drivers, for instance. In addition, internal interrupts, or traps, are generated by execution exceptions, such as arithmetic overflow, divide-by-zero, or illegal instruction code.

2.2 MEMORY TECHNOLOGIES

An understanding of central characteristics of current memory technologies is necessary when designing and analyzing real-time systems. This is particularly important, for example, with such embedded applications where the CPU utilization factor is planned to remain within the "dangerous" zone of 83–99% (see Chapter 1). In those almost time-overloaded systems, the worst-case access latency of hierarchical memory architecture may cause aperiodically missed deadlines with considerable delay. The following subsections contain behavioral and qualitative discussions that are slanted toward software and system engineers rather than hardware designers. A thorough treatment of memories and memory systems for embedded applications is available in Peckol (2008).

2.2.1 Different Classes of Memory

Volatile RAM versus *nonvolatile* ROM is the traditional distinction between the two principal groups of semiconductor memory, where RAM states for random-access memory and ROM for read-only memory. For years, this distinction was crisp, as long as the ROM devices were purely of such type that their contents were "programmed" either during the manufacturing process of the memory chip or at the application factory. Today, the borderline between RAM and ROM groups is no longer that clear because the commonly used ROM classes, EEPROM and Flash, can be rewritten without a special programming unit; they are thus in-system programmable. Although there are many different classes of memory within the two main groups, only the most important ones are introduced below. Figure 2.5 depicts the ordinary interface lines of a generic memory component.

Electrically erasable programmable ROM (EEPROM) and its close relative Flash are based on the dynamic floating-gate principle, and they both can be rewritten in a similar way as RAM devices. However, the erasing and writing process of those ROM-type devices is much slower than in the case of RAM; the rewrite cycle of an EEPROM can be up to 100 times slower than

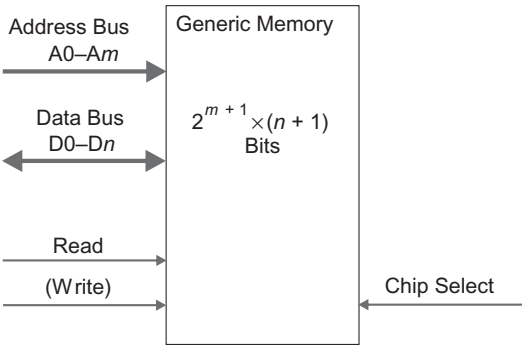


Figure 2.5. Interface lines of a generic memory component (*Write* is not used with ROM devices). The memory capacity is $2^{m+1} \times (n + 1)$ bits.

its corresponding read cycle. Moreover, each memory cell can typically be rewritten for 100,000–1,000,000 times only, because the stressful rewriting process wears the memory cells of EEPROM and Flash components. The basic 1-bit memory cells are configured in an array to form a practical memory device. While individual memory locations can be rewritten sparsely with EEPROMs, Flash memories can only be erased in large blocks. Thus, these rewritable ROMs, which may hold their data for approximately 10 years, are by no means rivals of RAM devices, but they are intended for different purposes. EEPROMs are normally used as a nonvolatile program and parameter storage, and Flash memory is used for storing both application programs and large data records. The nonrewritable mask-programmed ROM is still used as a low-cost program memory in certain standardized applications with a very large production volume. Finally, it should be mentioned that ROM-type memories are slower to read than typical RAM devices. Therefore, in many real-time applications, it may be viable or even necessary to run programs from faster RAM instead of ROM. Another common practice is to load the application program from a movable Flash memory card (or a USB memory stick) to RAM memory for execution. In that way, the Flash device behaves as a rugged and low-cost mass memory for embedded systems.

There are two classes of RAM devices: static RAM (SRAM) and dynamic RAM (DRAM). Either or both of these classes are used also in real-time systems. A single SRAM-type memory cell needs typically six transistors to implement a bistate flip-flop structure, while a DRAM cell can be implemented with a single transistor and capacitor only. Hence, if we compare memories of same size and similar fabrication technology, SRAMs are, by their structure, more space intensive and more expensive, but faster to access, and DRAMs are very compact and cheaper, but slower to access. Due to an inherent charge leakage in their storage capacitors, DRAMs must be refreshed regularly to avoid any loss of data; the refresh period should be no slower than 3–4 ms. The refreshing circuitry increases logically the dimensions of DRAM

chips, but that is not usually a critical issue, because individual DRAM devices contain much more memory than SRAM devices, and, thus, the relative proportion of refreshing circuitry is tolerable. An analogous control circuitry exists also in EEPROMs and Flash memories for managing the higher-voltage erase-and-write process.

When designing a RAM subsystem for some real-time application, there is a basic rule of thumb that if you need a large memory, then use DRAM; but if your memory needs are no more than moderate, SRAM is the recommended alternative—particularly with small embedded systems. Nevertheless, the practice is not always that straightforward, because there may exist the so-called CPU–memory gap—"the increasingly large disparity between the memory latency and bandwidth required to execute instructions efficiently versus the latency and bandwidth the external memory system can actually deliver" (Hadimioglu et al., 2004). In other words, the CPU's fastest bus cycle may be (much) shorter than the minimum access time of available memory components. If that is the case, the CPU cannot run at full speed when accessing the slower memory. This creates a CPU–memory bottleneck in high-performance applications; and it can be relieved by hierarchical memory organizations. In lower-performance applications, the possible conflict can be overcome simply by extending the length of bus cycle to match it to the access time specifications of memory components.

2.2.2 Memory Access and Layout Issues

Memory access principles are intimately connected to the specific computing hardware. Nonetheless, they cannot be ignored even by real-time software or system engineers. It is not sufficient to be aware of the CPU's architecture and peak performance only, because of the CPU–memory bottleneck introduced above. Quite often, the system bus is not operated at its full speed due to limitations set by the memory access time. This is affecting negatively to the response times of a real-time system. Memory-read access time is the essential time delay between enabling an addressed memory component and having the requested data available on the data bus. This is illustrated in the timing diagram of Figure 2.6, which uses the signals of a generic memory component (Fig. 2.5). Memory-write access time is defined correspondingly. The typical read and write cycles contain handshaking between the CPU and the memory device. And the time to complete the handshaking is dependent on the electrical characteristics of the CPU, system bus, and memory device.

When determining the length of a suitable bus cycle, we need to know the worst-case access times of memory and I/O ports, as well as the latencies of address decoding circuitry and possible buffers on the system bus. With a synchronous bus protocol, it is possible to add wait states (or additional clock cycles) to the default bus cycle, and adapt it dynamically to the possibly different access times of memory and I/O components. Asynchronous system buses do not need such wait states, because the data transfer between CPU

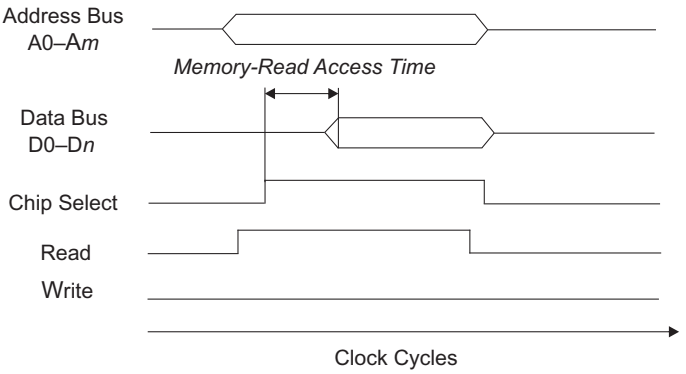


Figure 2.6. Timing diagram of a memory-read bus cycle. The angles "< >" shown in the data and address buses indicate that multiple lines with different logic states are involved during this period.

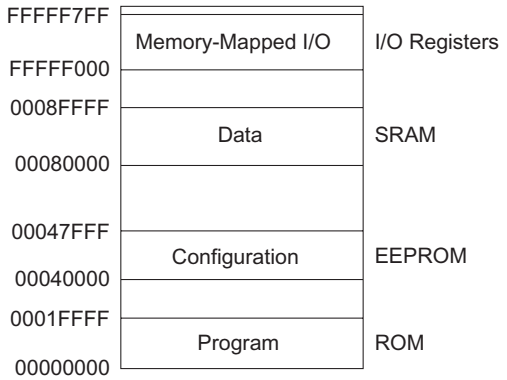


Figure 2.7. Typical memory map showing allocated regions (not to scale). Note, a large proportion of the memory space is not allocated for any specific purpose.

and memories or I/O ports is based on a handshaking-type protocol. Both bus protocols are used in commercial processors.

Another critical constraint may sometimes be the overall power consumption of the real-time hardware, which is growing with increasing CPU clock rate. From that point of view, the clock rate should be as low as possible and memories as slow as possible. Thus, the appropriate bus-cycle length is an application-specific parameter—maybe a critical one in battery-powered embedded systems.

To the real-time software engineer, the memory and I/O layout or map is of great importance. Consider, for instance, a 16-bit embedded microprocessor that supports a 32-bit address space organized, as shown in Figure 2.7. These starting and ending addresses are arbitrary, but could be representative of a

particular embedded system. For example, such a map might be consistent with the memory organization of an elevator controller.

In our imaginary elevator controller, the executable program code resides in memory addresses 00000000 through 0001FFFF *hexadecimal*. This standard control system has such a high production volume that it is practical to use mask-programmed ROM devices (128 K words). Miscellaneous configuration data, possibly related to various factory settings and installation-specific parameters, are stored at locations 00040000 through 00047FFF in EEPROM (32 K words) that can be rewritten during service or maintenance visits. Locations 00080000 through 0008FFFF are SRAM memory (64 K words), and they are used for the real-time operating system's data structures and general-purpose data storage. Finally, the upper locations FFFFF000 through FFFFF7FF (2 K memory locations) contain addresses associated with interface modules that are accessed through memory-mapped I/O, such as parallel inputs and outputs for various status and command signals; fieldbus connections for serial communication with the group dispatcher and car computer; RS-232C interface for a service terminal; and real-time clock and timer/counter functions. This memory map is fixing the freely relocatable addresses of the system and application software to the physical hardware environment.

Before going to the important discussion on hierarchical memory architectures, it is beneficial to bring up some categories of DRAM according to the mode of data access. While the basic DRAM device is meant to be randomly accessible, the different DRAM modules offer significant performance improvement in special data-access modes that are designed for rapid access of *consecutive* memory locations. These types of devices utilize such techniques as row access, access pipelining, synchronized interface, and access interleaving, for narrowing the CPU–memory gap. These advanced modes are highly valuable when the memory organization is hierarchical, and fast loading of data blocks to cache memory from the DRAM-based main memory is needed. On the other hand, the remarkable speed improvement is not actually realized if advanced DRAM modules are accessed randomly. Thus, the DRAM modules are used mostly in workstation environments, where large main memories are needed. Below is a sampled evolution path of DRAM modules with advanced access modes in their order of appearance (from the late eighties to 2007):

- Fast page mode (FPM) DRAM
- Extended data output (EDO) DRAM
- Synchronous DRAM (SDRAM)
- Direct Rambus DRAM (DRDRAM)
- Double data rate 3 synchronous DRAM (DDR3 SDRAM)

In such real-time systems, which are implemented on regular office or more reliable industrial PCs, the advanced DRAM modules are naturally used at the level of main memory. Typical applications include a centralized monitor-

ing system for a bank of elevators and a distributed airline reservation and booking system. Under specific circumstances, the most advanced DRAM modules may offer minimum access times comparable with those of fast SRAMs.

2.2.3 Hierarchical Memory Organization

The CPU–memory gap started to build up gradually in the early 1980s, and already in the nineties, the CPU clock rates were increasing 60% per year, while the access times of DRAM modules were improving less than 10% per year. Hence, the troublesome performance gap was continuously widening. A somewhat similar situation existed also with high-performance microcontrollers and digital signal processors, although their smaller memory subsystems are typically assembled of SRAMs and ROM-type devices. A vast majority of low-end microcontrollers, however, do not suffer the CPU– memory bottleneck at all, because their clock rates are no higher than a few tens of MHz. But in the early 2000s, the increase of CPU clock rates was practically saturated due to the overly high power consumption and severe heat problems associated with multi-GHz processors. While the fastest possible memory is desired in real-time systems, often, cost dictates the technology that can be used.

An efficient way to relieve the CPU–memory gap is to implement a cache memory between the main memory and the CPU. Cache memories rely on the *locality of reference* principle. Locality of reference refers to the address distance in memory between consecutive code or data accesses. If the code or data fetched tends to reside close in memory, then the locality of reference is high. Conversely, when programs execute instructions that are scattered locality of reference is low. Well-written programs in procedural languages tend to execute sequentially within code modules and within the body of instruction loops, and hence have usually a high locality of reference. While this is not necessarily true for object-oriented code, problematic portions of such code can often be linearized. For example, arrays tend to be stored in blocks in sequence, with elements commonly accessed sequentially. When software is executed in a linear sequential fashion, instructions are in sequence and therefore are stored in nearby memory locations, thus yielding a high locality of reference.

Locality of reference forms a powerful basis for hierarchical memory organizations, which can effectively utilize the advanced DRAM modules with fast block-access capabilities for loading of sequential instruction codes or data from DRAM (main memory) to SRAM (cache). A cache is a relatively small storage of fast memory where frequently used instructions and data are kept. The cache also contains a swiftly accessible list of memory blocks (address tags) that are currently in the cache. Each memory block can hold a small number of instruction codes or data, typically no more than a few hundred words.

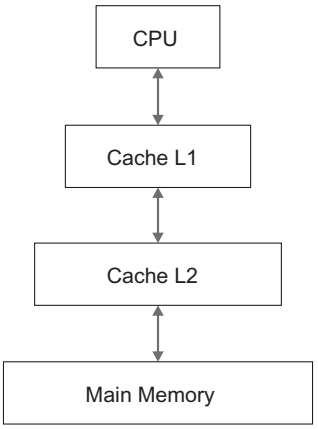


Figure 2.8. Hierarchical memory organization with two cache levels, L1 and L2, between the CPU and main memory.

The basic operation of the cache is as follows. Suppose the CPU requests the content of a DRAM location. First, the cache controller checks the address tags to see if the particular location is in the cache. If present, the data is immediately retrieved from the cache, which is significantly faster than a fetch from main memory. However, if the needed data is not in the cache, the cache contents must be written back and the required new block loaded from main memory to the cache. The needed data is then delivered from the cache to the CPU, and the address tags are updated correspondingly by the cache controller.

Cache design considerations include: access time, cache size, block size, mapping function (e.g., direct-mapped; set associative; fully associative), block replacement algorithm (e.g., first-in-first-out, FIFO; least-recently used, LRU), write policy (e.g., should altered data be written immediately through or wait for block replacement), number of caches (e.g., there can be separate data and instruction caches, or an instruction cache only), and number of cache levels (typically 1–3). A thorough discussion on these design considerations is available in Patterson and Hennessy (2009). Figure 2.8 illustrates a three-level memory hierarchy with cache levels L1 and L2.

Example: Performance Estimation of Cache Structures

What performance benefit could a practical cache provide? Consider a two-level memory hierarchy with a single 8 K cache built inside the CPU. Assume a noncached memory reference costs 100 ns, whereas an access from the cache takes only 20 ns. Now assume that the cache hit ratio is 73% (and miss ratio 27%). Then the average access time would be

$$\tau_{AVG\_1} = 0.73 \cdot 20\ \text{ns} + 0.27 \cdot 100\ \text{ns} \approx 42\ \text{ns}.$$

Next, we consider a three-level memory hierarchy with an 8 K upper level cache built inside the CPU and an external 128 K lower level cache. Assume the access times of 20 and 60 ns, respectively, and the cost of non-cached memory reference is 100 ns. The upper level hit rate is again 73%, and the lower level hit rate is 89%. Now the average access time would be

$$\tau_{AVG\_2} = 0.73 \cdot 20 \text{ ns} + 0.27 \cdot 0.89 \cdot 60 \text{ ns} + 0.27 \cdot 0.11 \cdot 100 \text{ ns} \approx 32 \text{ ns}.$$

Because access time for cache is faster than for main memory, performance benefits are a function of the cache hit ratio, that is, the percentage of time that the needed instruction code or data is found in the cache. A low hit ratio can result in worse performance than if the cache did not actually exist. That is, if data required is not found in the cache, then some cache block needs to be written back (if any data were altered) and replaced by a memory block containing the required data. This overhead can become significant when the hit ratio is poor. Therefore, a low hit ratio can degrade performance. Hence, if the locality of reference is low, a low number of cache hits would be expected, degrading real-time performance.

Another drawback of using a cache is that the effective access time is non-deterministic; it is impossible to know *a priori* what the cache contents and hence the overall access time will be. In the above two examples, the effective access time varies between 20 and 100 ns with averages 42 and 32 ns, respectively. Thus, response times in a real-time system with hierarchical memory organization contain a cache-originated element of nondeterminism. In multitasking real-time systems, frequent switching between different software tasks as well as aperiodically serviced interrupts, do temporarily violate the locality of reference leading to high probability of cache misses.

In some embedded processors, it is possible to load a time-critical code sequence permanently to the instruction cache, and thus reduce the possible nondeterminism in its execution time. This is a potential option in many digital signal processing, control, and image processing applications requiring strict real-time punctuality.

2.3 ARCHITECTURAL ADVANCEMENTS

CPU architectures have evolved remarkably since the introduction of the first microprocessors. The limitations of the sequential instruction cycle of the basic von Neumann architecture have caused various architectural enhancements to evolve. Most of these enhancements are built on the assumption of high locality of reference that is valid most of the time with a high probability. While the steady development of design automation and integrated-circuit technologies has made it possible to design and integrate more and more functionality to a single chip, architectural innovators have exploited this capability to introduce new forms of parallelism into instruction processing.

Thus, an understanding of advanced computer architectures is essential to the real-time systems engineer. While it is not our intent to provide a comprehensive review of computer architectures, a discussion of the most important issues is necessary.

In Section 2.1, we presented a sequential instruction cycle: Fetch instruction (F), Decode instruction (D), Load operand (L), Execute ALU function (E), and Store result (S). That instruction cycle contains two kinds of memory references, *instruction* fetching and *data* loading/storing. In the classical von Neumann architecture of Figure 2.1 or 2.4, the F and L/S phases are not independent of each other, because they are sharing the single system bus. Therefore, in pipelined architectures to be discussed shortly, it would be beneficial to have separate buses for instructions and data to be able to perform simultaneous F and L/S phases. On the other hand, two parallel address/data buses occupy a sizeable chip area—but that is the price to be paid for the improved performance. Such architecture is called the Harvard architecture, and it became first popular in digital signal processors. Many modern CPUs comprise both Harvard and von Neumann characteristics: the separate on-chip instruction and data caches have a Harvard-type interface, while the common off-chip cache memory is interfaced through a single system bus. Thus, this kind of hybrid architecture is internally Harvard but externally von Neumann (i.e., Princeton).

In the Harvard architecture, it is possible to have different bus widths for instruction and data transfer. For example, the instruction bus could be 32 bits wide and the data bus only 16 bits wide. Moreover, the instruction-address bus could have 20 bits and the data-address bus 24 bits. That would mean 2 M words of instruction memory and 16 M words of data memory. Hence, the architectural designer has flexibility when specifying the bus structures. Figure 2.9 depicts the Harvard architecture with parallel instruction and data access capabilities. From the real-time systems viewpoint, the basic Harvard architecture represents a well-behaved enhancement; it does not introduce any addi-

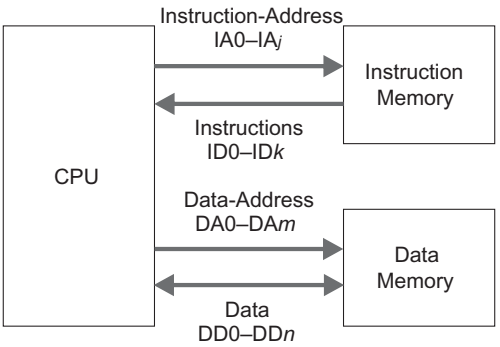


Figure 2.9. Harvard architecture with different bus widths.

tional latency or nondeterminism to the instruction cycle. It could even be seen as a potential relief to the CPU–memory bottleneck; but that is not the way the Harvard architecture is presently utilized.

Today, both the Harvard and von Neumann architectures include a number of enhancements increasing the level of parallelism in instruction processing. The most important architectural enhancements are discussed below. In spite of their great average-case benefits, they typically degrade the timing predictability and worst-case performance as discussed in Thiele and Wilhelm (2004).

2.3.1 Pipelined Instruction Processing

Pipelining imparts implicit execution parallelism in the different phases of processing an instruction, and hence aims to increase the instruction throughput. Suppose execution of an instruction consists of the five phases discussed above (F–D–L–E–S). In the sequential (nonpipelined) execution suggested in Section 2.1, one instruction can be processed through a single phase at a time. With pipelining, multiple instructions can be processed in different phases simultaneously, improving processor performance correspondingly.

For example, consider the five-stage pipeline of Figure 2.10. The upper picture shows the sequential execution of the fetch, decode, load, execute, and store phases of two instructions, which requires 10 clock cycles. Beneath that sequence is another set of the same two instructions, plus four more instructions, with overlapping processing of the individual F–D–L–E–S phases. This pipeline works perfectly if the instruction phases are all of equal length, and every instruction needs the same amount of time to complete. If we assume that one pipeline stage takes one clock cycle, the first two instructions are completed in only six clock cycles, and the remaining instructions are completed within the 10 clock cycles. Under ideal conditions with a continuously full pipeline, a new instruction is completed at the rate of one clock cycle. In general, the best possible instruction completion time of an N-stage pipeline is $1/N$ times the completion time of the nonpipelined case. Therefore, the ALU

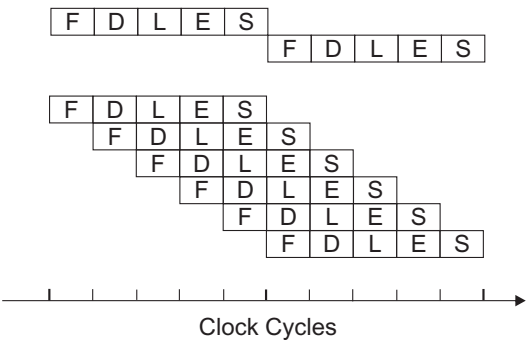


Figure 2.10. Pipelined instruction processing in a five-stage pipeline.

and other CPU resources are utilized more effectively. It should be mentioned, however, that pipeline architecture requires buffer registers between the different stages of instruction processing. That causes an additional delay to a pipelined instruction cycle compared with the nonpipelined cycle, where transitions from one phase to another may be direct without intermediate buffer writing and reading.

Another disadvantage of pipelining is that it can actually degrade performance in certain situations. Pipelining is a form of speculative execution in that the instructions that are prefetched are assumed to be the next sequential instructions. Speculative execution works well if the locality of reference remains high. If an instruction in the pipeline is a conditional branch instruction, the succeeding instructions in the pipeline may not be valid, and the pipeline must be flushed (the pipeline registers and flags are all reset) and refilled one stage at a time. To avoid probabilistically the negative effect of pipeline flushing/refilling, many processors have advanced branch prediction and speculation capability. A similar, but unpredictable, situation arises with external interrupts. In addition, data and input dependencies between consecutive machine-language instructions can slow pipeline flowthrough by requiring temporary stalls or wasted clock cycles.

Higher-level pipelines, or superpipelines, can be constructed if the instruction cycle is decomposed further. For example, a six-stage pipeline can be constructed, consisting of a fetch stage, two decode stages (needed to support indirect addressing modes), an execute stage, a write-back stage (which finds completed operations in the buffer and frees corresponding functional units), and a commit stage (in which the validated results are written back to memory). In practice, there exist superpipelines with much more than 10 stages in high-performance CPUs with GHz-level clock rates. Superpipelines with short stage-lengths offer, in principle, short interrupt latencies. However, that potential benefit is typically buried behind unavoidable cache misses and necessary pipeline flushing/refilling when the locality of instruction reference is severely violated. Extensive pipelining is thus a source of significant nondeterminism in real-time systems.

2.3.2 Superscalar and Very Long Instruction Word Architectures

Superscalar architectures further increase the level of speculation in instruction processing. They have at least two parallel pipelines for improving the instruction throughput. One of those pipelines may be reserved for floating-point instructions only, while all other instructions are processed in a separate pipeline or even in multiple pipelines. Figure 2.11 illustrates the operation of two superscalar pipelines with five stages. Those pipelines are supported with highly redundant ALU and other hardware resources. Theoretically, the instruction completion time in a K-pipeline architecture with N-stage pipelines may be as short as $1/(K \cdot N)$ times the completion time of the nonpipelined case—hence more than one instruction may be completed in a single clock

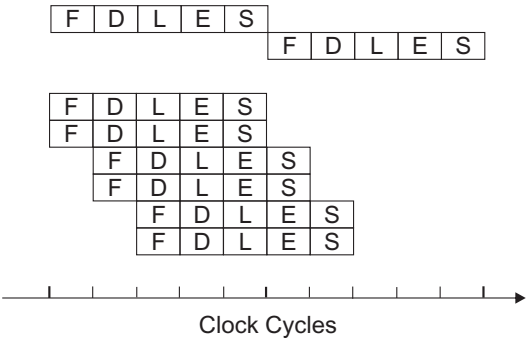


Figure 2.11. Superscalar architecture with two parallel instruction pipelines.

cycle. Such a parallel scheme would work fine if the executed instructions were fully independent of each other and the branch prediction ability was perfect. Nonetheless, that is not usually the case with real-world programs, and, therefore, the average utilization rate of parallel resources is far from 100%. If compared with an architecture with a single pipeline or superpipeline, a multipipeline CPU has even a greater variance between the best- and worst-case performances.

Superscalar CPUs are complex integrated-circuit implementations, not just because they have extensive functional redundancy, but also due to the sophisticated interdependency checking and dispatching logic. The hardware complexity may still increase if out-of-order instruction execution is used for maximizing the utilization rate of expensive ALU resources. Thus, superscalar processors are mainly used in workstations and nonreal-time applications. It is difficult to build a deterministic embedded system on a superscalar platform, although it could offer a very high peak performance.

Very long instruction word (VLIW) architecture is similar to the superscalar architecture in the sense that they both have extensive hardware redundancy for supporting parallel processing of instructions. However, there is a fundamental difference in the process of checking the interdependency between consecutive instructions and dispatching them optimally to appropriate functional units. While the superscalar architecture relies completely on hardware-based (on-line) dependency checking and dispatching, the VLIW architecture does not need any hardware resources for those purposes. The high-level language compilers of VLIW processors handle both the dependency checking and dispatching tasks offline, and very long instruction codes (typically at least 64 bits) are composed of multiple regular instruction codes. Since only mutually independent instructions can be combined, any two accessing the data bus cannot. In VLIW architectures, there is no online speculation in instruction dispatching, but the instruction-processing behavior is well predictable.

It should be noted, however, that the efficiency of VLIW architecture depends solely on the capabilities of the advanced compiler and the properties of the native instruction set. Compiler support for VLIW processors is studied in Yan and Zhang (2008). The compiler's code-generation process becomes very challenging with a number of parallelization goals and inter-dependency constraints. Therefore, the application programmer should assist the compiler in the difficult dispatching problem by tailoring the critical algorithms for the specific VLIW platform. In general, programs written for one VLIW processor are rather poorly transferable to other VLIW environments. While superscalar CPUs are used in general-purpose computing applications, VLIW CPUs are usually customized for some specific class of applications, such as multimedia processing, and they are used even in real-time systems.

2.3.3 Multi-Core Processors

As an architectural innovation, a processor with multiple interconnected cores or CPUs is nothing new. For a long time, such parallel architectures were considered special ones until the introduction of general-purpose multi-core processors in the early years of 2000. Those special architectures were used for different number-crunching applications, such as finite-element modeling or multimodal optimization using population-based algorithms. Today, multi-core processors are used in high-end real-time systems with high computational burden or strict requirements for task concurrency.

What are the driving forces behind the substantial multi-core developments? By the early 2000s, it became evident that the development of CPU architectures could no more be dependent on the continuously growing clock rates. Superpipelined architectures with 20 or even 30 pipeline stages assumed clock rates at the multi-GHz level. On the other hand, such very high clock frequencies greatly increase the power consumption of CPU chips, and that leads unavoidably to serious problems with the heat generated. It is a major challenge to keep the high-speed chips cool enough with costly and space-hungry cooling accessories. Without adequate cooling, those chips would destroy themselves in a short time. In addition, sub-ns clock periods create difficult data synchronization problems within large integrated circuits. Hence, there appears to be a consensus among the leading CPU manufactures to maintain the highest clock rates below 2–3 GHz and put emphasis on the development of multi-core architectures. With the continuing evolution of integrated-circuit fabrication technology, it is still possible to increase the number of gate equivalents on state-of-the-art processor chips. Currently, the term "multi-core" refers to processors with two (dual-core), four (quad-core), or eight identical cores, but the number of parallel cores is going to increase along with the advancement of integrated-circuit technology.

In multi-core processors, each individual core has usually a private cache memory, or separate instruction and data caches. These small on-chip caches are interfaced to a larger on-chip cache memory that is common to all cores.

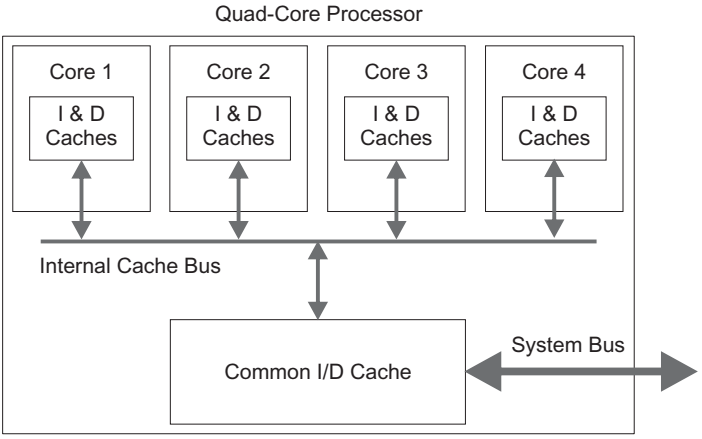


Figure 2.12. Quad-core processor architecture with individual on-chip caches and a common on-chip cache ("I" = instruction and "D" = data).

A representative multi-core architecture is depicted in Figure 2.12. An integrated multi-core processor needs a smaller footprint on the printed circuit board than a comparable implementation with multiple separate cores would require, which is quite an advantage in many applications.

The introduction of standard multi-core processors presents opportunities for parallel processing to a huge group of research and development (R&D) engineers in nearly all application domains. Nonetheless, serious R&D work leading to novel contributions in any parallel environment requires a complete collection of software tools for supporting the entire development process, and for creating multitasking real-time systems that could use the whole potential of true task concurrency. Moreover, software engineers should learn to design their algorithms for parallelism; otherwise, the potential of multi-core architectures remains largely unused. Manual load balancing between different cores is an important task that needs both human expertise and appropriate tools for performance analysis.

It is time consuming to port existing single-CPU software efficiently to a multi-core environment. This challenge will certainly reduce the application companies' interests in switching to multi-core processors in *matured* real-time applications, like elevator bank control or cell phone exchanges, which could undoubtedly benefit from such switching.

The nondeterministic instruction processing in multi-core architectures is caused primarily by the underlying memory hierarchy, pipelining, and possible superscalar features. On the other hand, the opportunity for task concurrency is certainly of great interest to engineers developing real-time systems. Lastly, it should be remembered that punctual and fast inter-core communication is a key issue when developing high-performance parallel systems. The

communications channel is a well-known bottleneck in multi-processor systems. We will return to the parallelization challenges in Chapter 7, where Amdahl's law is presented. It establishes a theoretical foundation for estimating the speedup when the number of parallel cores increases. The limit of parallelism in terms of speedup appears to be a software property, not a hardware one.

2.3.4 Complex Instruction Set versus Reduced Instruction Set

Complex instruction set computers (CISC) supply relatively sophisticated functions as part of the native instruction set. This gives the high-level language compiler a rich variety of machine-language instructions with which to produce efficient system or application code. In this way, CISC-type processors seek to increase execution speed and minimize memory usage. Moreover, in those early years, when assembly language still had an important role in real-time programming, CISC architectures with sophisticated instructions reduced and simplified the programmer's coding effort.

The traditional CISC philosophy is based on the following nine principles:

1. Complex instructions take multiple clock cycles.
2. Practically any instruction can reference memory.
3. No instruction pipelining.
4. Microprogrammed control unit.
5. Large number of instructions.
6. Instructions are of variable format and length.
7. Great variety of addressing modes.
8. Single set of work registers.
9. Complexity handled by the microprogram and hardware.

Besides, obvious memory savings are realized because implementing sophisticated functions in high-level language would require many words of program memory. Finally, functions written in microcode always execute faster than those coded in some high-level language.

In a reduced instruction set computer (RISC), each instruction takes only one clock cycle. Usually, RISCs employ little or no microcode. This means that the instruction-decode procedure can be implemented as a fast digital circuitry, rather than a slower microprogram. In addition, reduced chip complexity allows for more work registers within the same chip area. Effective use of register-direct instructions can decrease the number of slower memory fetches.

The more recent RISC criteria are a complementary set of the nine principles to CISC. These are:

1. Simple instructions taking one clock cycle.
2. Memory access by load/store instructions only.
3. Highly pipelined instruction processing.
4. Hard-wired control unit.
5. Small number of instructions.
6. Instructions are of fixed format and length.
7. Few addressing modes.
8. Multiple sets of work registers.
9. Complexity handled by compilers and software.

A more quantitative definition of RISC is available in Tabak (1991). Any RISC-type architecture could be viewed as a processor with a minimal number of vertical microinstructions, in which programs are directly executed in the hardware. Without any microcode interpreter, all instruction operations can be completed in a single (hard-wired) "microinstruction."

RISC has fewer instructions; hence, operations that are more complicated must be implemented by composing a sequence of simple instructions. When this is some frequently used operation, the compiler's code generator can use a preoptimized template of instruction sequence to create code as if it were that complex instruction. RISC needs naturally more memory for the sequences of instructions that form a complex instruction. CISC, on the other hand, uses more clock cycles to execute the microinstructions used to implement the complex instruction within the native instruction set.

RISCs have a major advantage in real-time systems in that the average instruction execution time is shorter than for CISCs. The reduced instruction execution time leads to shorter interrupt latency and thus shorter response times. Moreover, RISC instruction sets tend to help compilers to generate faster code. Because the instruction set is significantly limited, the number of special cases that the compiler must consider is considerably reduced, thus permitting a greater variety of code optimization approaches.

On the downside, RISC processors are usually associated with caches and elaborate multistage pipelines. Generally, these architectural enhancements improve the average-case performance of the processor by shortening the effective memory access times for frequently accessed instruction codes and data. However, in the worst case, response times are increased because low cache hit ratios and frequent pipeline flushing can degrade performance. Nonetheless, the greatly improving average-case performance at the expense of degraded worst-case performance is often tolerable at least in firm and soft real-time applications. Lastly, it should be mentioned that modern CISC-type processors share some principles of RISC architectures; for instance, virtually all CISC processors contain some form of instruction pipelining. Thus, the borderline between CISC and RISC is not crisp at all. A specific CPU architecture belongs to the CISC category if it fulfills *most* of the nine CISC principles; the same applies with the RISC definition.

2.4 PERIPHERAL INTERFACING

Peripheral, sensor, and actuator interfacing (Patrick and Fardo, 2000) is a central area of real-time hardware that is developing much slower than, for instance, memory subsystems and processor architectures. While the latter ones seem to be under incessant evolution, the peripheral interfacing principles have remained largely the same for decades. The fundamental practices for input and output handling are still the same as in the late seventies:

- Polled I/O
- Interrupt-driven I/O
- Direct memory access

In a polled I/O system, the status of the I/O device is checked periodically, or, at least, regularly. Therefore, such I/O activity is software controlled; only accessible status and data registers are needed in the hardware side. An obvious advantage of such an approach is its simplicity, but, on the other hand, it loads the CPU due to possibly unnecessary status requests. Typically, only a minority of status requests leads to either input or output transactions with the data register. This unnecessary loading could be reduced by less frequent polling of the I/O status. However, that would increase the worst-case I/O latency. Hence, an appropriate polling interval is an application-specific compromise between the desired CPU utilization factor and allowed I/O latency.

Figure 2.13 depicts a generic peripheral interface unit (PIU) with three internal registers. In addition to the status and data registers, there is a configuration register for selecting the desired operation mode. Actually, the

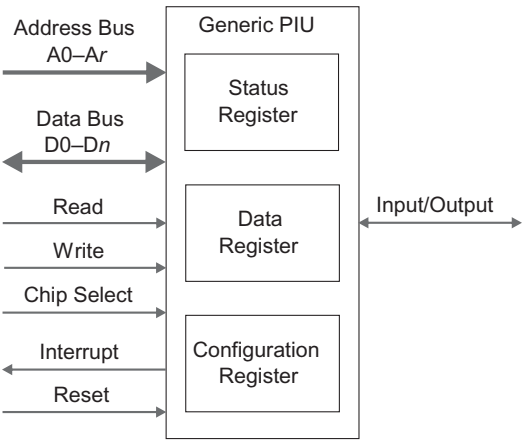


Figure 2.13. Interface lines of a generic peripheral input/output unit with three internal registers.

programmable PIU is, in some cases, a special-purpose processor that can manage independently such complicated functions as a communications network protocol or multichannel pulse-width modulation. Hence, advanced PIUs may relieve the CPU loading significantly in embedded real-time applications. Application programmers should not have direct access to PIUs, but they are used through device drivers that belong to system software. These device drivers hide the hardware-specific details from application programmers, and, in this way, make it easier to port the application code to another hardware environment with somewhat different peripheral interface devices. This situation can be found in embedded systems that have a long life cycle. For example, the lifetime of a high-rise elevator control system may be around 25 years—this sets a notable challenge for the availability of spare parts, and, sometimes, new hardware has to be developed for the existing application software.

The next two subsections present the operating principles of interrupt-driven I/O and direct memory access, which can greatly improve the I/O performance of real-time systems.

2.4.1 Interrupt-Driven Input/Output

Interrupt-driven I/O processing has remarkable advantages over the straightforward polled I/O: the service latency can, in general, be reduced and made less uncertain without increasing the loading of the CPU. In Section 2.1, we already presented a typical interrupt service process in a case when interrupts are enabled and only a single interrupt request is active at a time. However, in many practical situations, there might appear multiple interrupt requests simultaneously. This raises two obvious questions: How to identify the various interrupt sources, and in which order should the interrupts be serviced? There are standard procedures for identifying the interrupting peripherals, as well as for determining their service order. Some of those procedures are practical with small real-time systems, while others are particularly effective in larger-scale systems. Nonetheless, they are usually not visible to application programmers, but are managed in the system software.

In small real-time systems with no more than a moderate number of possible interrupt sources, it is often practical to identify the interrupting peripheral by polling the status registers of all PIUs. A status register contains typically some flag that is set when the particular PIU is requesting an interrupt. Moreover, by selecting the static polling order suitably, certain high-priority peripherals may always be serviced before some lower-priority ones. And, if needed, the polling order could be modified dynamically to provide rotating priorities, for instance.

When the number of interrupting peripherals is large, it is no longer feasible to identify and prioritize interrupts using the simple polling scheme. Vectored interrupt handling is a convenient technique for larger real-time systems, because it moves the interrupt identification burden from system software to

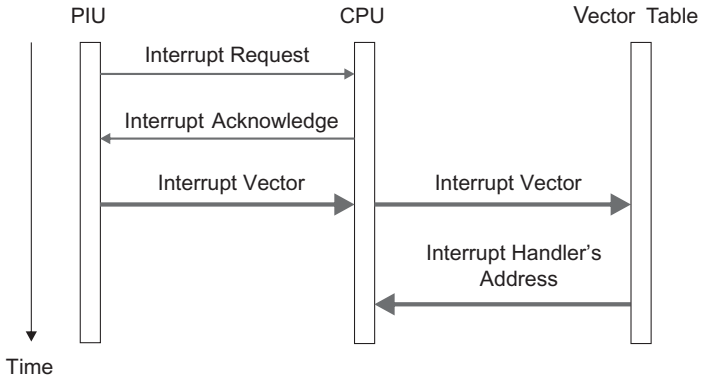


Figure 2.14. Interrupt-identification procedure between the CPU and PIU using vectored interrupt.

real-time hardware. Figure 2.14 illustrates the interrupt identification process with a vectored interrupt. The cost of using a vectored interrupt is in the more complex CPU and PIU hardware. Besides, some priority interrupt controller is needed to manage the priorities of individual interrupt sources.

A CPU that supports vectored interrupts has usually a substantial number of interrupt vectors available. If the number of vectors is 256, there could be 256 distinguishable interrupt sources. In most cases, however, not all the available interrupt vectors are needed in a real-time application. Still, it is recommended to write interrupt service routines for those unused interrupt codes as well. But why? In some operating environments, electromagnetic interference (EMI) radiation, charged particles, and various disturbances and noise may cause spurious problems by inverting some bits in main memory, in registers, or on the system bus. These kinds of problems are sometimes classified as "single-event upsets" (Laplante, 1993). The results of such problems can be catastrophic. For example, if even a single bit in the interrupt vector is inverted, the altered interrupt code may correspond to such an interrupt that is not in use (a phantom interrupt), and, thus, does not have an interrupt service routine. The effect can lead to a system crash. Fortunately, there is a simple solution to this crash problem: every interrupt vector should have a corresponding service routine, and in the case of phantom interrupts, it is just a return-from-interrupt or `RETI` instruction. It is advisable, though, that some phantom-interrupt counter in a nonvolatile memory is incremented as well. The value of such a counter could be monitored during the early phase of the product life cycle; if the hardware is properly designed and implemented, the counter should never be incremented. Unfortunately, while any real-time hardware should be designed to fulfill certain electromagnetic compatibility (EMC) and radiation hardening standards (Morgan, 1994), and appropriate software prevention techniques are available to deal with single event upsets (Laplante,

1993), unrealistic cost/schedule pressure and inadequate system testing often lead to the kinds of problems just described.

A priority interrupt controller (PIC) is used for prioritizing different interrupts when the vectored-interrupt scheme is used for identifying them. PICs have multiple interrupt inputs that are coming from PIUs (or directly from peripherals), and a single interrupt output that is going to the CPU. Some processors may even have a built-in PIC function. These programmable devices provide the ability to dynamically prioritize and mask interrupts of different priority levels. Each interrupt can be independently set to be either edge (rising or falling) or level triggered, depending on the needs of the attached peripheral. Edge-triggered interrupts are used with very long or very short interrupt pulses, and when overlapping interrupt requests on a single line are not possible. Level-triggered interrupts, on the other hand, are used more seldom—only when overlapping interrupt requests on a single line are expected—because edge-triggered interrupts are time-wise more precise. Figure 2.15 depicts the handling of multiple interrupts with an external PIC. The procedure contains 10 main steps (assuming that interrupts are enabled):

1. The PIC receives several simultaneous interrupt requests.
2. The PIC processes first the request with highest priority.
3. The CPU receives an interrupt request from the PIC.
4. The CPU completes the currently executing instruction.
5. The CPU stores the content of the program counter register (PCR) to memory.
6. The CPU acknowledges the interrupt to the PIC.
7. The PIC sends the interrupt vector to the CPU.
8. The CPU loads the corresponding interrupt-handler address to the PCR.

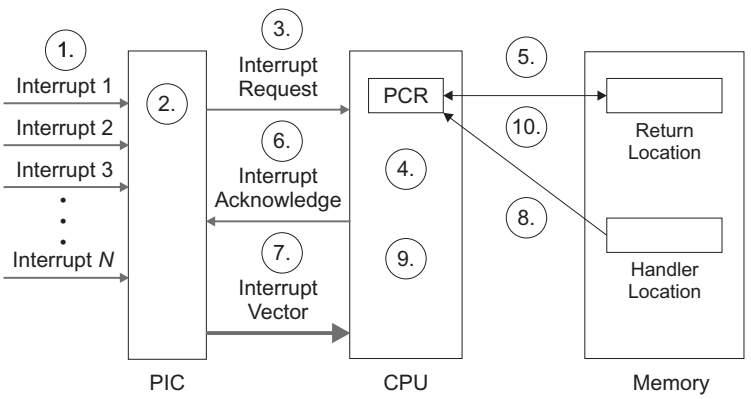


Figure 2.15. Handling multiple interrupts with an external priority interrupt controller; the circled numbers are referring to the 10-step procedure described in the text.

9. The CPU executes the interrupt handler.
10. The CPU reloads the original PCR content from memory.

Although interrupt-driven I/O is an effective technique for (hard/firm) real-time systems, it ought to be remembered that the privilege to interrupt should be given to time-critical I/O events only. Otherwise, a large number of concurrent interrupt requests may sporadically lead to excessive response times. A complementary discussion on interrupt-related issues is available in Ball (2002).

2.4.2 Direct Memory Access

While interrupt-driven I/O handling is effective when the number of transferred data bytes or words between memory and I/O ports is reasonably small, it becomes ineffective if large blocks of data are transferred. Each data element must first be read from memory or an input port into the CPU's work register and then written to an output port or a memory location. Such block-transfer processes take place regularly, for example, with communications networks, graphics controllers, and hard-disk interfaces—or even between two memory segments. To eliminate the time-consuming circulation of data through the CPU, another I/O handling practice, direct memory access (DMA), is available. In DMA, access to the computer's memory is given to other devices in the system without any CPU intervention. That is, data is transferred directly between main memory and some external device. Here, a separate DMA controller is required unless the DMA-handling circuitry is integrated into the CPU itself. Because no CPU participation is required, data transfer is faster than in polled or interrupt-driven I/O. Therefore, DMA is often the best I/O method for real-time systems; and it is becoming increasingly widespread due to extensive use of communications networks and distributed system architectures, for instance. Some real-time systems have even multiple DMA channels.

An I/O device requests DMA transfer by activating a DMA-request signal (D_REQ). This makes the DMA controller issue a bus-request signal (B_REQ) for the CPU. The CPU finishes its present bus cycle and activates a bus-acknowledgment signal (B_ACK). After recognizing the active B_ACK signal, the DMA controller activates a DMA-acknowledgment signal (D_ACK), instructing the I/O device to begin data transfer. When the transfer is completed, the DMA controller deactivates the B_REQ signal, giving buses back to the CPU (Fig. 2.16).

The DMA controller is responsible for assuring that only one device can place data on the bus at any one time through bus arbitration. This essential arbitration procedure resembles the interrupt prioritization discussed above. If two or more devices attempt to gain control of the bus simultaneously, bus contention occurs. When some device already has control of the bus and another device obtains access, a collision occurs. The DMA controller prevents

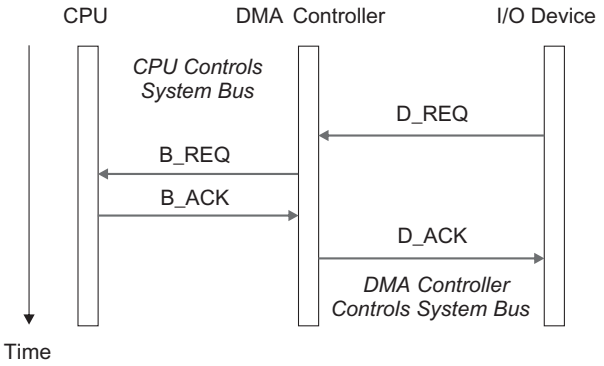


Figure 2.16. Establishing a data-transfer connection between an I/O device and main memory using DMA.

collisions by requiring each device to issue the D_REQ signal that must be acknowledged with the D_ACK signal. Until the D_ACK signal is given to the requesting device, its connection to the system bus remains in a high-impedance condition. Any device that is in the high-impedance state (i.e., disconnected) cannot affect the data bits on the memory data bus. Once the D_ACK signal is given to the requesting device, its memory-bus lines become active, and data transfer occurs similarly as with the CPU. For each data transfer occasion, the DMA controller needs a memory address specifying where the data block exists or where it will be placed, and the amount of transferable bytes or words. Such information is programmed to the control registers of the DMA controller by the CPU (a function of system software).

During a DMA transfer, the ordinary CPU data-transfer processes cannot proceed. At this point, the CPU could proceed solely with nonbus-related activities until the DMA controller releases the buses or until it gives up and issues a bus time-out signal (after some predetermined time). Yet a CPU with a cache memory may still execute instructions for some time during a DMA transfer. From the real-time viewpoint, a long DMA cycle is somewhat similar to the condition when interrupts are disabled, because the CPU cannot provide service for any interrupts until the DMA cycle is over. This may be critical in real-time systems with high sampling rates and strict response-time requirements. To tackle the problem, a single transfer cycle of a large data block can be split to several shorter transfer cycles by using a cycle-stealing mode instead of the full-block mode. In the cycle-stealing mode, no more than a few bus cycles are used at a time for DMA transfer. Hence, the interrupt-service latency does not become unreasonably long when transferring a large block of data using DMA.

In certain hard real-time applications, however, the use of DMA is avoided by placing a dual-port SRAM (or DPRAM) device between a block-oriented I/O device and the CPU. The DPRAM contains a single memory array with

private bus connections for both the primary CPU and some I/O processor. Hence, the primary CPU is never giving the control of its system bus to any other device; but block-oriented data transfer takes place in the dual-port memory without disturbing the CPU. Dual-port SRAMs are used widely with communications networks and graphics controllers.

2.4.3 Analog and Digital Input/Output

Real-time system designers should be aware of certain characteristics of I/O signals and functions, which are associated with timing and accuracy. There is a variety of I/O categories, particularly in embedded real-time systems. The core categories are outlined below:

- Analog
- Digital parallel
- Digital pulse
- Digital serial
- Digital waveform

In the following paragraphs, a discussion of this important topic is provided with a few hardware examples. We point out key issues related to analog and digital I/O signals and their trouble-free interfacing. A supplementary presentation on specific peripheral interface units is available in Ball (2002) and Vahid and Givargis (2002), for instance.

Analog-to-digital conversion, or A/D circuitry, converts continuous-time (analog) signals from various devices and sensors into discrete-time (digital) ones. Similar circuitry can be used to convert pressure, sound, torque, and other current or voltage inputs from sensors and transducers by using a variety of conversion schemes. The output of A/D circuitry is a *discrete-time* and *quantized* version of the analog signal being monitored. At each sampling moment, the A/D circuitry makes available an n-bit approximation that represents a quantized version of the signal. This data can be passed on to the real-time computer system using any of the three I/O handling methods. Samples of the original continuous-amplitude waveform are treated in application programs as scaled integer numbers.

The fundamental aspect in the use of A/D circuitry for time-varying signals is the sampling rate. In order to convert a continuous-time signal into a discrete-time form without loss of any information, samples of the analog signal must be taken at a rate of at least twice that of the highest frequency component of the signal (the Nyquist–Shannon sampling theorem). Hence, a signal with a highest frequency component at 500 Hz must be sampled at least 1000 times per second. This implies that software tasks serving A/D circuitry must run at the same rate, or risk losing information. Besides, high punctuality of consecutive sampling moments is essential in many control and signal pro-

cessing applications. These considerations form an intrinsic part of the design process for the scheduling of software tasks. In most control applications, however, the applied sampling rate is 5–10 times higher than the minimum rate. One reason for this is the common noise content in the measured signal, which should be low-pass filtered to avoid violation of the sampling theorem, leading to harmful aliasing. Nonetheless, traditional band-selective filters always introduce some delay (or phase shift) to the filtered primary signal (Vainio and Ovaska, 1997), and that could reduce the controllability of the plant or process. By using a higher sampling rate, the control performance can often be improved, and the aliasing effect reduced without using a highly selective low-pass filter. Fortunately, in many monitoring and audio signal-processing applications, a moderate phase delay is tolerable, and appropriate low-pass filters can thus be used in front of A/D converters.

It should be noted, however, that the Nyquist–Shannon sampling theorem does not consider the nonlinear quantization effect at all. While pure sampling with an adequate sampling rate is a truly reversible operation, quantization always introduces some irreversible error to the digital signal. One additional bit of quantization resolution corresponds approximately to a 6 dB increase in the signal-to-noise ratio (SNR) of the digitized signal (Garrett, 2000). The number of bits in A/D converters is typically 8–16 in control applications, but can be more than 20 in hi-fi audio systems, for example. Figure 2.17 illustrates the varying quantization error in a simplified case with a 3-bit A/D converter. In a real-time system, the A/D-converter's resolution is usually a compromise between the application's accuracy requirements and the

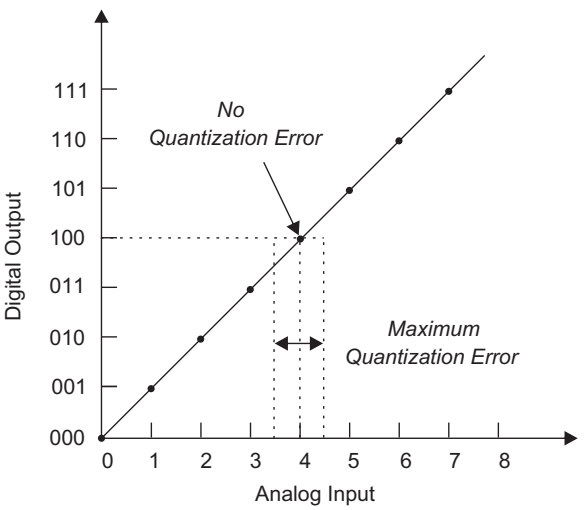


Figure 2.17. Quantization of an analog ramp signal using a 3-bit A/D converter. The quantization error varies between –1/2 LSB (least significant bit) and +1/2 LSB, and it is thus proportional to the number of bits in the conversion process.

product's cost pressure. Moreover, the accuracy of a practical A/D-conversion channel is never the same as its resolution, but, typically, one or two least-significant bits should be considered erroneous. That has to be remembered when implementing control and signal-processing algorithms.

Another design issue related to analog input channels is the occasional need for truly simultaneous sampling of two or more measurement quantities. There is usually an analog multiplexer in front of the A/D converter to provide selectable measurement channels for a single A/D-converter. That is a compact and low-cost solution, but cannot provide simultaneous sampling of multiple quantities. An additional A/D converter would be a straightforward solution to this problem, but it could be a relatively expensive option in many embedded systems. Therefore, it is often practical to use individual sample-and-hold (S&H) circuits in those measurement channels that require simultaneous sampling. The CPU gives a concurrent "sample" command to those S&H circuits that memorize their analog inputs for a short period of time. After this, all the S&H outputs are converted sequentially to the digital form by the one A/D converter. Although the digital samples become available one after another, they still correspond to the same sampling moment.

Digital-to-analog conversion, or D/A circuitry, performs the inverse function of A/D circuitry; it converts a digital quantity to an analog one. D/A devices are used to allow the computer to output analog currents or voltages based on the digital version stored internally. Nevertheless, D/A converters are not as common in real-time systems as A/D converters, because many actuators and devices are commanded directly with digital signals. D/A converters are sometimes included solely for providing real-time outputs of critical or selectable intermediate results of computational algorithms. This may be useful during the hardware-software integration and verification phases of sophisticated control and signal-processing algorithms. The communication with D/A circuitry also uses one of the three I/O handling methods discussed.

Digital I/O signals can be classified into four categories: parallel, pulse, serial, and waveform. Diverse parallel inputs are practical for reading the status of on/off-type devices like electro-mechanical limit switches or relay contacts in machine-automation applications, for instance. Parallel outputs are used similarly for providing on/off commands to a variety of actuators, such as fans or pumps, in building automation. While the PIU output ports need some driver circuit to be able to sink/source high load currents, the input ports must be protected against interferences that are corrupting the incoming signals. Severe EMI levels are usual in industrial applications of real-time systems (Patrick and Fardo, 2000). Typical input circuitry contains first some overvoltage suppressor for protecting the interface channel. It is followed by an optical isolator that converts the voltage levels (e.g., from +24/0 V I/O logic to +5/0 V CPU logic) and creates galvanic isolation between the I/O-ground potential and the CPU ground. This is necessary for preventing electric coupling of disturbances from the possibly harsh operating environment to the sensitive computer system. After the galvanic isolation, on/off-type signals are

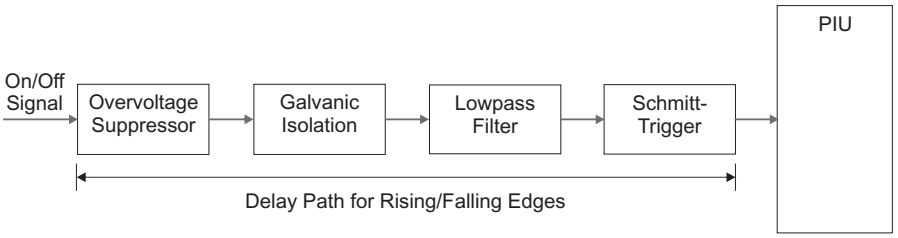


Figure 2.18. Block diagram of a digital input channel intended for an operating environment with high EMI levels.

usually low-pass filtered by an RC filter to attenuate high-frequency disturbances and noise. Finally, the smoothened signal edge (rising or falling) is restored by a Schmitt-trigger circuit containing some hysteresis. All this is necessary to make sure that the digital input signal is of adequate quality before feeding it to the PIU (Fig. 2.18). Furthermore, special attention should be paid to such digital signals that cause interrupts directly, because a noisy on/off edge may be interpreted as multiple edges, leading to a burst of false interrupts instead of a single desired interrupt. Thus, the interface-hardware requirements are very different in industrial environments from those adequate in home or office environments.

But why is the protection of parallel input ports of interest to real-time software engineers? Well, it is certainly straightforward to clean on/off-type signals by using appropriate signal processing techniques, but, at the same time, the transition edges (from "on" state to "off" state or vice versa) are necessarily delayed. This increases the latency of excitation signals, as well as the response time that is measured from the *true* transition moment to the corresponding output action. Hence, with time-critical events, all kinds of filtering should be kept minimal to avoid intolerable hardware latency. This initial latency component is accumulated with a possible chain of nondeterministic latency components originating, for example, from a "dangerous" CPU utilization factor, pipeline flushes, cache misses, sensor-network's variable load, and software-task scheduling. If adequate filtering cannot be afforded, then the principal solution is to use shielded signal cables—or even optical fibers—to prevent disturbances from corrupting edge-critical signals. The same is also valid with pulse-type inputs.

Pulse and waveform outputs, on the other hand, also have accuracy requirements, because the widths of generated pulses have specific tolerances. This is central when individual pulses are used for turning on/off devices or functions for a precise duration of time. Moreover, in high-performance pulse-width modulation, the tolerances of consecutive pulses may be rather strict. Both pulses and waveforms are typically generated by some timer circuit; and the timing accuracy depends on the reference frequency, as well as the length of the counter register. In addition, there is a nondeterministic latency

component due to interrupt handling and software-task scheduling. This latency should be taken into account when prioritizing different interrupts and associated tasks in a real-time system. Similar considerations are needed with pulse and waveform inputs.

Serial digital I/O is used for transferring data over a single line instead of multiple parallel lines (or a bus). Embedded systems often have two kinds of serial links: a low-speed one for a local user interface, and a high-speed connection to some longer-distance communications (or fieldbus) network. While the low-speed serial links do not set any challenges for the real-time software engineer, the high-speed networks may demand a lot of computing performance. Hence, the receiver/transmitter buffering and communications protocols are often handled by a special-purpose processor, which is interfaced to the main CPU by using DMA.

Today, an increasing number of network connections are implemented using some wireless medium—either an infrared or a radio connection. The emerging wireless sensor networks use tiny computer nodes for performing autonomous measurements in an environment where it is not possible to provide an external power supply for those nodes. Therefore, the distributed nodes are battery powered, and the battery lifetime should be maximized to avoid impractical service of the nodes. This economy is accomplished by effectively utilizing the CPU's sleep mode; the communications protocol can adjust how often the hardware is awakened for a communications session. The awake–sleeping duty cycle is application dependent, and network latency is clearly sacrificed over battery lifetime and vice versa. This ultra-low power consumption is a new type of requirement for certain real-time systems.

2.5 MICROPROCESSOR VERSUS MICROCONTROLLER

Up to now, we have used the general term "processor" for the entire assortment of processing units containing some kind of CPU—from high-performance microprocessors to application-specific cores in systems on a chip. However, under the processor class, there are two distinct subclasses, microprocessors and microcontrollers, which deserve an introductory discussion. From the real-time systems viewpoint, microprocessors are currently used mainly in nonembedded applications, while various microcontrollers are dominating the embedded-systems field. That has not always been the case, though. Therefore, it is good to discuss the evolution paths of real-time processors, beginning from the introduction of the first microprocessor in the early 1970s. The purpose of the following paragraphs is to provide some insight for understanding the few divergent development paths of processor technology (Fig. 2.19).

2.5.1 Microprocessors

A microprocessor is an integrated circuit that contains the functions of a complete CPU. At the time of its introduction—about 40 years ago—it opened

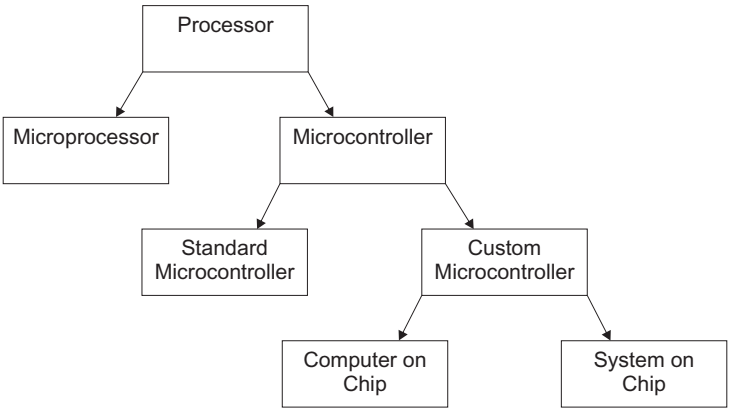


Figure 2.19. Principal evolution paths of processor technology.

totally new opportunities for the research and development community to innovate and design *intelligent* systems and products. In this context, we adopt the following definition for the term "intelligence":

> Intelligence can be defined in terms of the capability of a system to adapt its behavior to meet its goals in a range of environments (Fogel, 2006).

The first microprocessor decade was, in many ways, confusing, because the microprocessor components and software development tools were in their very infancy; and the users of those new microprocessors were more or less self-educated without the benefits of experience. Still, by the mid-1970s, the first microprocessor-based elevator control systems were successfully being developed. Those early implementations were not intelligent according to the definition, since they were just replacing certain relay-based logic by straightforward microprocessor code. Nonetheless, the introduction of microprocessors in embedded applications was certainly a turning point for that conservative branch of industry. The same applies with countless other fields that gradually started to benefit from microprocessors and the exciting opportunity to create novel functionality—or even machine intelligence—by software.

When the instruction-processing throughput of microprocessors steadily increased, and the memory and peripheral interface devices became more advanced, the era of embedded systems was truly begun—that was in the early 1980s. At the beginning, the hardware clearly had a central role in all development work, but by the middle of the 1980s, the logical need for proper software engineering procedures and associated support tools started to emerge; real-time software development was no more just code writing. Today, we observe that most of the real-time systems development effort in microprocessor environments is software engineering, not hardware engineering. The used hardware platforms are typically either standard office PCs or industrial PCs with special interface modules.

From the early days, microprocessors have evolved significantly, and the architectural advancements discussed in Section 2.3 are available specifically in the latest microprocessors. The foremost goal in the development of microprocessors is the further increasing instruction-processing throughput. In parallel with the innovative architectural developments, such as superpipelining and superscalar processing with out-of-order execution, the use of microprocessors in embedded systems has greatly diminished. The increased nondeterminism in interrupt latency causes insuperable problems for hard real-time systems. Nevertheless, many embedded applications could, in principle, benefit from the high instruction-processing throughput of microprocessors.

2.5.2 Standard Microcontrollers

Soon after the introduction of the first 8-bit microprocessors, another development path, microcontrollers, emerged. A microcontroller is an integrated circuit containing a CPU, as well as an interconnected set of memory devices, peripheral interface units, timers/counters, etc. Hence, the microcontroller can take direct input from devices and sensors and directly control external actuators. The need for "single-chip computers" became apparent as soon as the first embedded systems were designed that were based on microprocessors and a set of *external* memory and I/O devices. Later on, when the microprocessor path utilized consistently the remarkable developments of integrated-circuit technology for advancing the CPU architecture, the microcontroller path had its main emphasis on extending the available RAM and ROM spaces, as well as the variety of peripheral interface units. To make the package compact and inexpensive, some microcontrollers do not have an external system bus that, on the other hand, would make it also possible to use external memory and PIU devices. The CPU of a high-performance microcontroller may have a short instruction pipeline, a clear-cut RISC architecture with a duplicate set of work registers for interrupt handlers, and possibly Harvard architecture. As a converged result of evolution, a modern microcontroller could contain the following set of PIUs and memory devices:

- EEPROM or Flash
- SRAM
- Analog-to-digital converter with a multiplexer
- Direct-memory-access controller
- Parallel inputs and outputs
- Serial interface
- Timers and counters
- Pulse-width modulators
- Watchdog timer

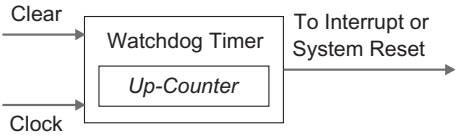


Figure 2.20. Block diagram of a watchdog timer with its inputs and output.

The list is not all encompassing, but it contains a representative collection of functions that are available in numerous commercial microcontrollers.

An interesting element, the watchdog timer, is worthy of an introduction, because it can be used as a supervision unit in real-time systems, particularly in those ones that operate autonomously. Many embedded systems are equipped with a watchdog up-counter that is incremented periodically by a clock signal. The counter must be cleared regularly by an appropriate pulse before it overflows and generates a watchdog interrupt (this clearing action is sometimes called "petting the dog"). In normal operating conditions, the application software issues regularly a pulse via memory-mapped or programmed I/O to clear the counter frequently enough.

Watchdog timers are used to ensure that certain devices are serviced at regular intervals, that certain software tasks execute according to their prescribed rate, and that the CPU continues to function normally. To make sure that a crashed real-time system can be recovered successfully, it is sometimes wise to connect the watchdog timer's interrupt output to the nonmaskable interrupt line, or even to the line that is used to reset the whole system (Fig. 2.20). In addition, whenever the watchdog interrupt is activated, a variable in nonvolatile memory should be incremented to record such an abnormal event that is always an indication of some software or hardware problem—or maybe a system-level problem related to electro-magnetic interferences.

The first-generation microcontrollers were intended for general embedded applications—their memory capacity and PIU selection were not tailored for any specific field of application. However, by the early 1980s, a variety of application-specific microcontrollers started to emerge. Around this time, the so-called digital signal processors also became available for data communications and other signal processing applications.

Digital signal processors have a CPU architecture that supports fast processing of a limited set of instructions, and provides short interrupt latency. This capability is effectively accomplished by a RISC-type Harvard architecture with truly parallel multiplication and addition units. The availability of multiplication-accumulation (MAC) instructions, which take only one clock cycle to execute, is the key characteristic that ties such architecture to digital signal processing (DSP) applications; because many DSP algorithms (e.g., convolution) contain a chain of multiplication-addition operations. Besides, the sampling rates of those algorithms are often relatively high. More recently, some digital signal processors with VLIW architectures have become available

for specific DSP applications. Nonetheless, a typical digital signal processor is nothing more than an application-specific microcontroller with a special-purpose CPU architecture and appropriate PIU and memory support.

In addition to digital signal processors, there are also other application-specific microcontrollers for common application areas, such as automotive, communications, graphics, motor control, robotics, and speech processing. Moreover, in the mid-1980s, an exceptional family of networkable microcontrollers, transputers, was introduced for creating parallel-processing implementations easily. A transputer contains a rather traditional von Neumann CPU (either with or without floating-point support), but *its novel instruction set includes directives to send or receive data via four serial links that are connected to other transputers* (nodes). The transputers, though capable of acting as a uniprocessor, are best utilized when connected together in a nearest-neighbor configuration. Nonetheless, transputers never attained the true acceptance of the *global* R&D community, and thus their production was terminated. Although the transputer itself disappeared, its pioneering architectural innovations were adopted to a few networkable microcontrollers available today. Those microcontrollers with automatically (by hardware) updated network variables are used particularly in building automation and elevator control applications (Loy et al., 2001). Perhaps the transputer concept was introduced too early, when the potential of convenient networking over various media was not yet recognized.

Most microcontrollers are standard components, and a few billions of them—mostly simple 8-bit microcontrollers—are produced annually. Hence, there are usually certain memory or PIU features of the off-the-shelf microcontroller that are not (fully) utilized in a specific real-time system. This ineffectiveness could be avoided by creating *product-specific* custom microcontrollers. That is, indeed, taking place when developing particular high-volume products, as we will see shortly.

2.5.3 Custom Microcontrollers

Custom microcontrollers (or core processors) began to appear in the late 1980s for applications like high-speed telephone modems, and for miscellaneous systems where low-power consumption is a major issue. For example, while the availability of SRAM in a standard microcontroller would be 2 K words, a core processor could contain 1234 words of memory to fulfill the *exact* needs of an imaginary software implementation. Thus, the memory array would be approximately 40% smaller, and the chip size would be reduced correspondingly. This potential benefit is realizable only if the production volume of the core processor is large enough to compensate for the high design expenses of the custom integrated circuit (Vahid and Givargis, 2002). Such designs can be seen as computers on chip (Fig. 2.19), and they require an extensive verification phase, because the final design does not offer flexibility to make modifications. However, if the core processor contains EEPROM

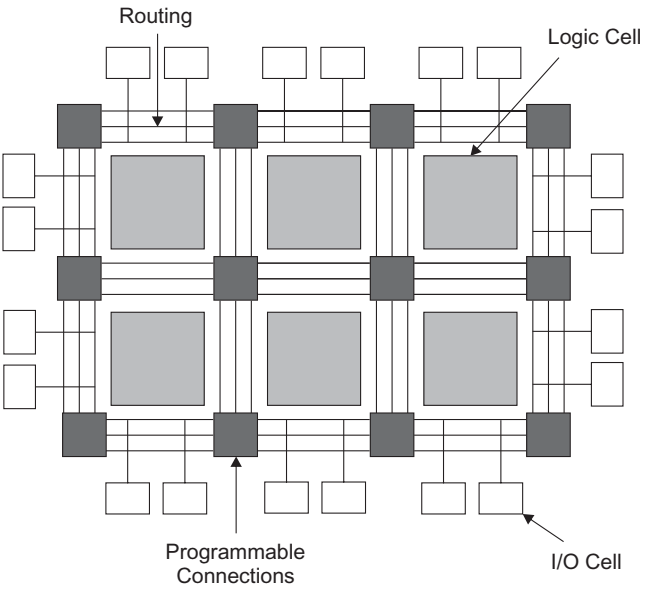


Figure 2.21. Conceptual architecture of an FPGA device with 6 logic cells (Mayer-Baese, 2007). In reality, the number of logic cells may be even more than 100,000.

or Flash blocks for program code, it is possible to modify the software within the limits of fixed memory space. The CPU of a core processor is either some version of a standard CPU or a special custom design.

Moreover, some (dynamically) reconfigurable processor architectures have recently been proposed in field-programmable gate array (FPGA) environments (Hauck and Dehon, 2007). FPGAs provide new opportunities for innovating real-time systems with flexible computing performance. The configurable FPGA technology provides for the construction of a custom microcontroller with an application-specific CPU, memory, and I/O—even for products with low or moderate production volume, because the FPGAs are standard components that are configured by application designers. Figure 2.21 illustrates the general architecture of FPGA devices. In addition to buffering I/O cells and basic logic cells, containing elementary combinatorial and sequential logic, an FPGA may include more advanced cells, such as:

- Multiplier
- Tri-state bus
- CPU core
- SRAM and ROM
- Interface support for external DRAM modules
- Application-specific immaterial-property (IP) blocks provided by the FPGA manufacturer

Systems on chip take the core-processor approach even further by integrating on the same chip functionalities other than pure digital blocks (Saleh et al., 2006). A system on chip, or SoC, may contain analog-signal, analog-power, mixed analog-digital, radio-frequency, or micro-electro-mechanical blocks, as well. Thus, the design and fabrication of an SoC device may become a major challenge and a significant expense. A digital camera is a typical real-time application where practically all electronics are integrated on a single SoC; it is produced in high quantities, it must be compact, and its power consumption should be low. In cases when it is not feasible or possible to design and manufacture an SoC due to the presence of overly diverse integrated-circuit technologies, a relevant alternative might be a system in package, or SiP, where a few heterogeneous chips are placed in a single package. Both SoC and SiP devices are particularly attractive for novel ubiquitous-computing applications, where sensors and real-time computation are intimately integrated into everyday activities and objects (Poslad, 2009).

2.6 DISTRIBUTED REAL-TIME ARCHITECTURES

As soon as embedded control systems began to emerge in the 1970s, the need for distributed real-time architectures became obvious in many applications. The main motivators behind spatial distribution over serial communications interfaces were typically: considerable savings in wiring expenses, flexibility in designing and upgrading large-scale systems, and making computing power available where it is needed. In the beginning, different subsystems were interconnected point-to-point via asynchronous serial links that were using some proprietary communications protocol. Such implementations offered low data rates, and it was difficult to modify them, because the primary CPU was also handling the low-level communications protocol; there was often moderate overlapping between the application software and the communications protocol. No multi-level layering of the communications protocol or special-purpose communications processors yet existed. Still, by the early 1980s, an elevator bank control system with eight elevators could contain 11 microprocessor subsystems that were communicating over two bus-type serial links at the data rate of 19.2 K bit/s. And, those microprocessors were of same type as the CPU of the first IBM PC.

2.6.1 Fieldbus Networks

Appropriate layering breaks the communications protocol into multiple pieces that are easier to design and implement than a flat protocol. Hence, in modern communications networks, the concept of layering is fundamental, and it usually follows the seven-layer open systems interconnection (OSI) model (Wetteroth, 2002):

1. *Physical* (bits): Conveys the bit stream across the network.
2. *Data Link* (frames): Builds data packets and synchronizes traffic.
3. *Network* (packets): Routes data to the proper destination.
4. *Transport* (segments): Error checking and delivery verification.
5. *Session* (data): Opens, coordinates, and closes sessions.
6. *Presentation* (data): Converts data from one format to another.
7. *Application* (data): Defines communications partners.

The use of OSI model makes it possible to change the data transfer medium (copper wire, optical fiber, wireless radio, or wireless infrared) and other properties of the protocol stack independently. In this section, we discuss the fieldbus networks that form a layered communications platform for distributed systems.

Fieldbus is a general name for communications protocols intended for real-time control systems (Mahalik, 2003), and there are several standard protocols used, for example, in automotive and factory automation applications. An example of fieldbus networks is the widespread controller area network (CAN), which was originally intended for automotive applications, but became widely used in industrial applications as well. The data rates with CAN are up to 1 M bit/s, and the protocol is supported by special-purpose microcontrollers that can handle independently the whole communications session. Such CAN controllers could communicate with the primary CPU through a dual-port RAM. Fieldbus networks are similar to regular computer networks existing in office environments (e.g., the Ethernet), but they are designed to support time-critical data transfer in operating environments with high EMI level. However, there are also industrial modifications of the prevalent Ethernet network, and thus the borderline between office and fieldbus networks is becoming blurred.

Fieldbus networks may be implemented using a variety of topologies (or physical structures), such as bus, ring, star, and tree (Fig. 2.22), depending on the nature and requirements of the particular application. These solutions offer flexibility for architectural designers. In large-scale systems, the number of nodes in a fieldbus network may be from hundreds to thousands. And in the case of the elevator bank control system previously mentioned, well over 100 nodes containing a microcontroller could nowadays be communicating over a few LON-type networks (bus topology and 78 K bit/s data rate) (Loy et al., 2001).

In addition to the clear advantages of distribution, networking creates a challenge for real-time system designers to work with: the inherent message transfer delay and its variation due to time-variant load on the transmission medium. These constraints may become a significant component in composite response times, and make the synchronization of distributed software tasks problematic.

The delay issue becomes critical in closed-loop control systems, which must provide satisfactory performance and guaranteed stability at all times. There

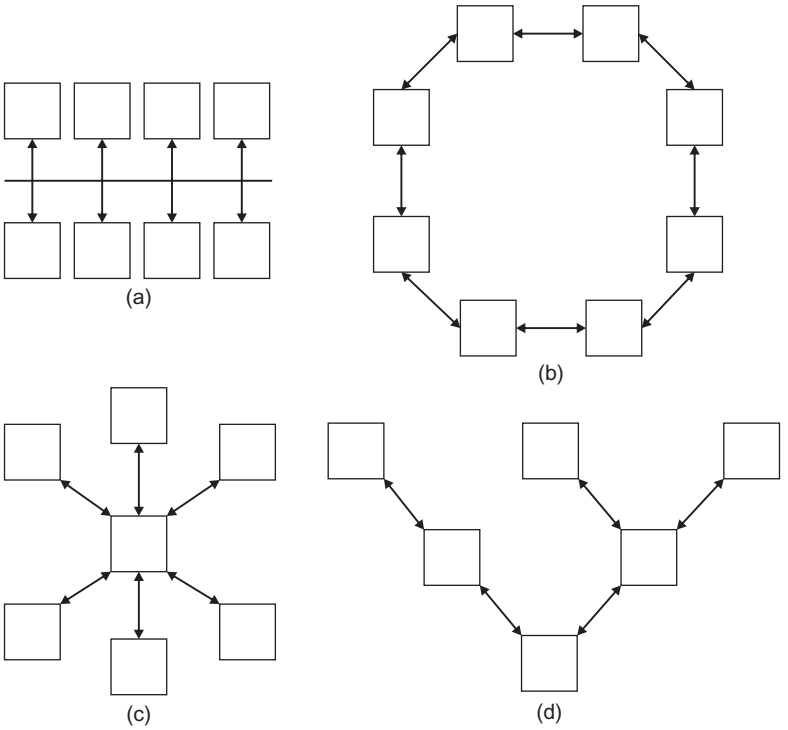


Figure 2.22. Bus (a), ring (b), star (c), and tree (d) topologies used commonly in fieldbus networks.

are two principal approaches for designing networked control systems (Chow and Tipsuwan, 2001). The first approach consists of multiple subsystems, in which each subsystem contains a set of sensors and actuators, as well as the control algorithm itself. Another approach is to connect sensors and actuators directly to the fieldbus network. In the latter case, the control algorithm is located in a separate node and it performs closed-loop control over the network. Such control systems are called network-based control systems, and they must tolerate the message transfer delays and their variation. Traditional control-system designs assume strictly periodic sampling of inputs and outputs; otherwise, the robustness and performance of the control system may degrade drastically. There are robust design techniques and stochastic control methods for tackling the delay problem, but the problem has been solved only in undemanding special cases—no generic solution exists.

If the communications platform could provide minimal and predictable delays in all practical conditions, the use of special algorithms would not be necessary. To achieve this goal, it is sometimes essential to provide two fieldbus connections between certain nodes: a *regular* channel and a *priority* channel.

The "regular" fieldbus network carries most of the data transmission load, while the "priority" connection is reserved for the most urgent and delay-sensitive messages only. This straightforward solution is naturally a complexity issue that decreases the system's reliability and increases material and assembly costs. Therefore, it would be valuable to have a communications architecture and corresponding protocol that would guarantee punctuality in message transfer.

2.6.2 Time-Triggered Architectures

Synchronous communications architecture with a common clock would provide a reliable platform for distributed real-time systems. However, it is not trivial to synchronize multiple nodes precisely when the physical distance between individual nodes may vary drastically. The time-triggered architecture (TTA), developed by Kopetz and others, can be used for implementing distributed hard real-time systems (Kopetz, 1997). The TTA models a distributed real-time system as a set of independent nodes interconnected by a real-time communications network (Fig. 2.23), and it is based on fault-tolerant clock synchronization. Each node consists of a communications controller and a host computer, which are provided with a global, synchronized clock. Furthermore, every node is truly autonomous, but communicates with other nodes over a replicated broadcast channel (in fact, two redundant channels). Therefore, each individual node in the TTA should be designed as a self-sufficient real-time system. Such synchronous architecture provides a very reliable and predictable mechanism for communications between nodes. In addition, a TTA-based system is fault tolerant, because should a node fail, the failure can be detected by another node that could assume, at least in principle, the failed node's responsibilities.

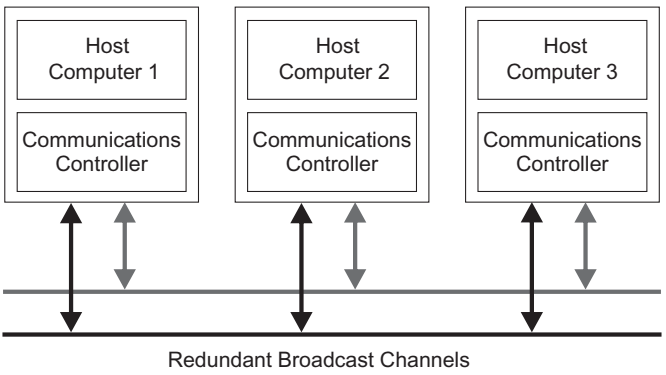


Figure 2.23. Time-triggered architecture with three nodes and two redundant broadcast channels.

Using time-division multiple access (TDMA), each node is allocated a fixed time slot in which it can send information on the broadcast channel to one or more receiving nodes, through a unique addressing scheme. It is thus possible to predict the latency of all messages on the channel, which guarantees hard real-time message delivery. Furthermore, since the messages are sent at a predetermined point in time, the latency jitter (or uncertainty) is minimal. Hence, time-triggered architectures can achieve real-time punctuality. By comparing the known point in time at which a particular message was sent and when it is received, host computers can synchronize their clocks with sufficient accuracy. Nonetheless, it is not possible to synchronize distributed clocks exactly, and thus there is always residual phase difference between the clocks of different nodes. This unavoidable condition is overcome by introducing a sparse timing lattice within the entire distributed system. The uniform spacing of the lattice is chosen such that the temporal order of any two observations taken anywhere in a time-triggered system can be reestablished from their time stamps (Kopetz, 1995).

Coordinated communications between the nodes of a TTA is implemented by the corresponding time-triggered protocol (TTP). The TTP is a dual-channel protocol with 25 M bit/s data rate on each redundant channel. Multiple manufacturers provide TTP communications controllers as integrated circuits or IP. There are two versions of the TTP available: the comprehensive TTP/C intended for safety-critical, hard real-time applications (Kopetz, 1997); and the simplified TTP/A for low-cost fieldbus applications (Kopetz, 2000).

The time-triggered architecture has been used successfully in numerous safety- and reliability-critical automotive and avionics applications, for instance. Such human-involved applications must contain *ultra-dependable* real-time systems to minimize the risk for a catastrophic failure. Therefore, ultra-dependable systems must be certified by professional certification agencies. As stated by Kopetz, such a certification process is greatly simplified if the certification agency can be convinced that the following three concerns are fulfilled (Kopetz, 1995):

1. "The subsystems that are critical for the safe operation of the system are protected by stable interfaces that eliminate the possibility of error propagation from the rest of the system into these safety-relevant subsystems."
2. "It can be shown that all scenarios that are covered by the given load and fault hypotheses can be handled according to the specification without reference to probabilistic arguments."
3. "The architecture supports constructive certification, that is, the certification of subsystems can proceed independently of each other, for example, the proof that a communications subsystem meets all deadlines can proceed independently of the proof of the performance of a node."

It is understandable that constructive collaboration between system, software, and hardware teams is required throughout the development project to fulfill

the concerns discussed above, because every critical subsystem is an integral hardware–software component.

An alternative to the synchronous TTA is naturally some event-triggered architecture (ETA), where computing and communications operations are activated *asynchronously* by specific events occurring within the real-time system or its environment. Although ETA approaches are used widely in various applications, they are more demanding to design, implement, and maintain when the goal is an ultra-dependable real-time system with minimal randomness in its timing behavior.

2.7 SUMMARY

A rigid foundation for real-time systems is laid out when decisions concerning the system architecture and specific hardware devices are made in the early stages of a development project. Such decisions establish the baseline for achievable response times and their uncertainty, too. Later on, the system and application software further contribute to these critical quantities. Hence, every response time consists of multiple components, and it is highly beneficial to know the average and possible variation of each latency component when making selections concerning the real-time operating system or key application algorithms. In general, cost-effective systems tend to have well-balanced latency components throughout the entire response-time chain; it is not needed to have a CPU with minimal interrupt latency if the main latency, caused by the fieldbus network, is orders of magnitude longer.

The selection of processor type (Fig. 2.19) is connected to the required strength of "real-time." For hard real-time systems, the recommended computing environment is obviously some microcontroller—without extensive pipelining and complicated memory hierarchy. In this way, the interrupt latency is kept minimal, which is crucial in many embedded applications. Soft real-time systems, on the other hand, are either embedded or nonembedded. In embedded cases, microcontrollers are also the primary choices, while microprocessor-based workstations are dominating the nonembedded and typically networked applications. The latency uncertainties, caused by extensive pipelining and sophisticated memory hierarchies, are tolerable in networked nonembedded applications, such as airline reservation and booking systems, because the unavoidable latencies in long-distance (even intercontinental) communications networks are highly nondeterministic and only weakly bound. Firm real-time systems are mostly embedded ones, and, thus, practical to implement on microcontroller platforms. Nonetheless, multi-core microprocessors have potential for applications where high instruction throughput is needed. In such cases, an *application-optimized hardware* (memory and I/O subsystems) should be designed around some multi-core CPU. This approach would reduce the latency uncertainties from those of general-purpose workstations, but still offer the opportunity of truly parallel multitasking in firm real-time systems.

I/O subsystems are possible sources of measurement inaccuracy, as well as considerable latency. Of those, analog inputs and outputs are common sources of subconscious inaccuracy, because a designer might consider the accuracy of A/D and D/A converters to be equal to their resolution. That is not, however, the case, and the accuracy issue must be addressed when designing signal processing and control algorithms, for instance. Other important I/O issues are the use of interrupts and their prioritization; only the most critical I/O events deserve the right to interrupt. In this way, the critical response times will have less variation. Fieldbus and other communications networks can be seen as potential threats in hard and firm real-time systems, because their latency characteristics may be varying substantially under different network loading conditions. Therefore, it is sometimes necessary to implement parallel networks for regular and priority messages, or even use some synchronous communications architecture, like the time-triggered architecture, to ensure that the tight response-time specifications are fulfilled in all practical conditions.

The material presented in this chapter has been loosely confined to the real-time effects of various hardware architectures, their practical implementations, as well as some specific devices. Hence, it forms a solid basis for the following chapter on real-time operating systems, which are immediate users of the hardware resources through device drivers, interrupt handlers, and scheduling procedures.

2.8 EXERCISES

2.1. Compose a table providing the available memory spaces for the following address-bus widths: 16, 20, 24, and 32 bits.

2.2. It is common practice for programmers to create continuous test-and-loop code in order to poll I/O devices or wait for interrupts to occur. Some processors provide an instruction (`WAIT` or `HALT`) that allows the processor to hibernate until an interrupt occurs. Why is the latter form more efficient and desirable?

2.3. In general terms, suggest a possible scheme that would allow a machine-language instruction to be interruptible. What would be the overall effect on instruction's execution time and CPU's throughput and response times?

2.4. Figure 2.5 illustrates the interface lines of a generic memory component. Assume $m = 15$ and $n = 7$. The address bus of your microprocessor is 24 bits wide. How, in principle, could you locate this particular memory block to begin from the address 040000 (hexadecimal)? What is the corresponding end address?

2.5. Compare and contrast the different memory technologies discussed in this chapter as they pertain to embedded real-time systems.

2.6. How would you test the validity and integrity of factory parameters stored in EEPROM? Sketch a suitable procedure for that purpose.

2.7. Assume a hierarchical memory system having a joint instruction/data cache with a memory-access cost of 10 ns on a hit and 90 ns on a miss. An alternative design without hierarchical memory architecture has a memory-access cost of 70 ns. What is the minimum cache-hit percentage that would make the hierarchical memory system useful?

2.8. The Harvard architecture (Fig. 2.9) offers separate address and data buses for instruction codes and data. Why is not it feasible to have separate buses for programmed I/O as well?

2.9. Show with an illustrative example how the five-stage pipeline discussed in this chapter (Fig. 2.10) could benefit from the Harvard architecture.

2.10. What special problems do superpipelined and superscalar architectures pose for real-time system designers? Are they any different for nonreal-time systems?

2.11. In CISC-type processors, most instructions are having memory operands, while RISC-type processors access memory by `LOAD` and `STORE` instructions only. What are the advantages and disadvantages of both schemes?

2.12. You are designing the architecture of a high-performance CPU for hard real-time applications. List and justify the principal architectural selections that you would make.

2.13. Discuss the relative advantages and disadvantages of memory-mapped I/O, programmed I/O, and DMA as they pertain to real-time systems.

2.14. Why is DMA controller access to main memory in most systems given higher priority than CPU access to main memory?

2.15. An embedded system has a 12-bit A/D converter for measuring voltages between –10 V and +10 V. What is the digital value corresponding to +5.6 V?

2.16. Find a microcontroller with unique, special instructions and, considering the application area for that processor, discuss the need for those special instructions.

2.17. What are the advantages of systems on chip over computers on chip (Fig. 2.19)? Find a few examples of commercial systems on chip from the Web.

2.18. A watchdog timer (Fig. 2.20) is used for supervising the operation of an embedded system in a high-EMI environment. Why is it practical to connect the watchdog-circuit's output to the CPU's nonmaskable (instead of maskable) interrupt input?

2.19. List the different data-transmission media mentioned in this chapter and give typical applications for each.

2.20. The time-triggered protocol (TTP/C or TTP/A) is used in safety- and time-critical applications. Make a Web search to find specific commercial applications where it is used.

REFERENCES

S. R. Ball, *Embedded Microprocessor Systems: Real World Design*, 3rd Edition. Burlington, MA: Elsevier Science, 2002.

M.-Y. Chow and Y. Tipsuwan, "Network-based control systems: A tutorial," *Proceedings of the 27th Annual Conference of the IEEE Industrial Electronics Society*, Denver, CO, 2001, pp. 1593–1602.

D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, 3rd Edition. Hoboken, NJ: Wiley-Interscience, 2006.

P. H. Garrett, *Advanced Instrumentation and Computer I/O Design: Real-Time Computer Interactive Engineering*. Hoboken, NJ: Wiley-Interscience, 2000.

H. Hadimioglu, D. Kaeli, J. Kuskin, A. Nanda, and J. Torrellas, *High Performance Memory Systems*. New York: Springer-Verlag, 2004.

S. Hauck and A. Dehon, *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation*. Burlington, MA: Morgan Kaufmann Publishers, 2007.

J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th Edition. Boston: Morgan Kaufmann Publishers, 2007.

H. Kopetz, "Why time-triggered architectures will succeed in large hard real-time systems," *Proceedings of the 5th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Cheju Island, Korea, 1995, pp. 2–9.

H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA: Kluwer Academic Publishers, 1997.

H. Kopetz, W. Elmenreich, and C. Mack, "A comparison of LIN and TTP/A," *Proceedings of the 3rd IEEE International Workshop on Factory Communication Systems*, Porto, Portugal, 2000, pp. 99–107.

P. A. Laplante, "Fault-tolerant control of real-time systems in the presence of single event upsets," *Control Engineering Practice*, 1(5), pp. 9–16, 1993.

D. Loy, D. Dietrich, and H.-J. Schweinzer, *Open Control Networks: LonWorks/EIA 709 Technology*. Norwell, MA: Kluwer Academic Publishers, 2001.

N. P. Mahalik, *Fieldbus Technology: Industrial Network Standards for Real-Time Distributed Control*. New York: Springer, 2003.

U. Mayer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd Edition. New York: Springer, 2007.

D. Morgan, *A Handbook for EMC Testing and Measurement*. London, UK: Peter Peregrinus, 1994.

D. R. Patrick and S. W. Fardo, *Industrial Electronics: Devices and Systems*, 2nd Edition. Lilburn, GA: The Fairmont Press, 2000.

D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th Edition. Boston: Morgan Kaufmann Publishers, 2009.

J. K. Peckol, *Embedded Systems: A Contemporary Design Tool*. Hoboken, NJ: John Wiley & Sons, 2008.

S. Poslad, *Ubiquitous Computing: Smart Devices, Environments and Interactions*. Hoboken, NJ: John Wiley & Sons, 2009.

R. Saleh et al., "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, 94(6), pp. 1050–1069, 2006.

D. Tabak, *Advanced Microprocessors*. New York: McGraw-Hill, 1991.

L. Thiele and R. Wilhelm, "Designing for timing predictability," *Real-Time Systems*, 28(2/3), pp. 157–177, 2004.

F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. Hoboken, NJ: John Wiley & Sons, 2002.

O. Vainio and S. J. Ovaska, "A class of predictive analog filters for sensor signal processing and control instrumentation," *IEEE Transactions on Industrial Electronics*, 44(4), pp. 565–570, 1997.

D. Wetteroth, *OSI Reference Model for Telecommunications*. New York: McGraw-Hill, 2002.

J. Yan and W. Zhang, "A time-predictable VLIW processor and its compiler support," *Real-Time Systems*, 38(1), pp. 67–84, 2008.

3

REAL-TIME OPERATING SYSTEMS

Every real-time system contains some operating system-like functionality to provide an interface to the input/output hardware and coordination of virtual concurrency in a uniprocessor environment—or even true concurrency with multi-core processors and distributed system architectures. Such a central piece of system software has the following principal aims: to offer a reliable, predictable, and low-overhead platform for multitasking and resource sharing; to make the application software design easier and less hardware bound; and to make it possible for engineers in various industries to concentrate on their core product knowledge and leave the computer-specific issues increasingly to specialized consultants and software vendors. While the variety of real-time applications is enormous, the variety of "real-time operating systems" is also considerable: from application-tailored pseudokernels to commercial operating systems. A pragmatic overview on architectures, principles, and paradigms of real-time operating systems is available in Stankovic and Rajkumar (2004).

Most application developers would be happy to have the functionality of a full-blown operating system available, but obvious design constraints like system cost and complexity, as well as response times and their punctuality, often direct the practitioner toward understandable compromises. This situation is particularly true with low-end embedded systems having high

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

production volumes. Although complete operating systems would provide highly valuable services for application programmers, the extra online effort of executing system software (or "running the bureaucracy") is taken from the same processor that is also executing the possibly time-critical application code. Hence, even from the real-time point of view, these services are not free.

It could be stated that this chapter is the most important one of our text, because it establishes the fundamental framework for real-time multitasking. And everything else is connected to this essential framework. We have carefully composed the following seven sections to present main aspects of real-time operating systems that are of immediate interest to practicing engineers. Section 3.1 introduces a variety of kernels and operating systems for different types of real-time applications. The section further defines the multi-level taxonomy from microkernels to operating systems, and discusses related implementation approaches with practical examples. Theoretical foundations of scheduling are outlined in Section 3.2, where selected fixed and dynamic priority scheduling principles are briefly analyzed and compared. Next, a thorough presentation of typical services for application programs is provided in Section 3.3. The emphasis is on intertask communication and synchronization, as well as in deadlock and starvation avoidance. Section 3.4 is devoted to memory management issues in real-time operating systems, and it considers the principles and implementation of the common task control block model, for instance. After establishing a solid understanding of real-time operating systems, we are ready to discuss the complicated process of selecting a suitable operating system, in Section 3.5. The critical consideration of "buying versus building" is studied from different viewpoints, and a practical selection metric is introduced. Section 3.6 summarizes the preceding sections on real-time operating systems. A substantial collection of exercises on various subtopics of this chapter is available in Section 3.7.

3.1 FROM PSEUDOKERNELS TO OPERATING SYSTEMS

A process (synonymously called "task" throughout this text) is an abstraction of a running program and is the logical unit of work schedulable by the real-time operating system. A process is usually represented by a private data structure that contains at least an identity, priority level, state of execution (e.g., running, ready, or suspended), and resources associated with the process. A thread is a lightweight process that must reside within some regular process and make use of the resources of that particular process only. Multiple threads that reside logically within the same process may share resources with each other. While processes are active participants of system-level multitasking, threads can be seen as members of process-level multitasking; this hierarchy is illustrated in Figure 3.1. It should be noted, however, that both processes and threads are available solely in full-featured operating systems executing typically in workstation environments. Such high-end environments are typi-

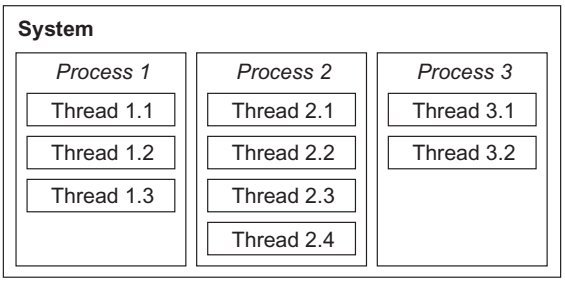


Figure 3.1. Hierarchical relationships between the system, multiple processes, and multiple threads.

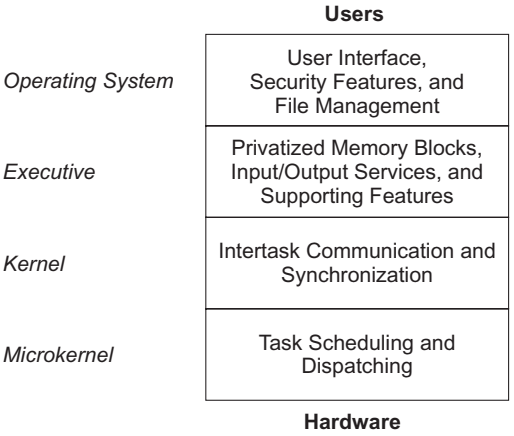


Figure 3.2. The role of the kernel in operating systems; moving up the taxonomy stack shows the additional functionality provided and indicates the relative closeness to hardware versus users.

cally running soft real-time applications. On the other hand, most embedded real-time systems have just a single category of tasks.

Real-time operating systems provide three essential functions with respect to software tasks: scheduling, dispatching, as well as intertask communication and synchronization. The kernel of the operating system is the smallest entity that provides for all these functions. A scheduler determines which task will run next in a multitasking system, while a dispatcher performs the necessary bookkeeping to start that particular task. Moreover, intertask communication and synchronization assures that parallel tasks may cooperate effectively. Four layers of operating system functionality and an associated taxonomy are shown in Figure 3.2.

The bottom layer of Figure 3.2, a *microkernel*, provides for plain task scheduling and dispatching. A *kernel* also provides for intertask communication and

synchronization via mailboxes, queues, pipes, and semaphores, for example. A real-time *executive* is an extended kernel that includes privatized memory blocks, input/output services, and other supporting features. Most commercial real-time kernels are actually executives by this definition. Finally, an *operating system* is an advanced executive that provides for a generalized user interface, security features, and a sophisticated file management system.

Regardless of the operating system architecture used, the ultimate objective is to satisfy real-time behavioral and temporal requirements, and make available a seamless multitasking environment that is flexible and robust.

3.1.1 Miscellaneous Pseudokernels

Real-time multitasking, in its rudimentary form, can be achieved without interrupts and even without an operating system per se. When feasible, these pseudokernel approaches are preferred because resultant systems are often highly predictable and easier to analyze. Nonetheless, they might be more laborious to extend and maintain than real-time systems using true kernels. Today, pseudokernels are generally only found in low-end embedded systems.

Straightforward polled loops are used for providing fast response to single devices. In a polled loop system, a repetitive instruction is testing a flag that indicates whether or not some event has occurred. If the event has not occurred, then the polling continues.

Example: Polled Loop

Suppose a piece of software is needed to handle packets of data that arrive at a rate of no more than one packet per second. A flag named `packet_here` is set by a fieldbus controller, which writes the data into the main memory via direct memory access. The data are available when `packet_here=1`.

Using a C code fragment, write a polled loop to handle such a system.

```
for(;;) {                    /* do forever */
        if (packet_here)     /* check flag */
          {
          process_data();    /* process data */
          packet_here=0;     /* reset flag */
          }
        }
```

Polled loop schemes work well when a single CPU is dedicated to handling the I/O for some fast device and when overlapping of events is not possible. Moreover, polled loops are commonly implemented as a background task in an interrupt-driven system, or as an individual task in a cyclic code structure.

In the latter case, the polled loop polls each cycle for a finite number of times to allow other tasks to run. Those other tasks handle the nonevent-driven processing. Interrupt-driven systems and cyclic code structures are discussed shortly.

A variation on the polled loop uses a fixed clock interrupt to pause between the time when the signaling event is triggered and then reset. Such a scheme is used, for example, to treat problematic events that exhibit contact bounce in various control applications. Contact bounce is a physical phenomenon that occurs because it is practically impossible to build an electromechanical switch that could change its state instantaneously without any contact oscillation. Events triggered by various buttons, contactors, relays, and switches all exhibit this undesired phenomenon. If, however, a sufficient delay occurs between the initial triggering of the event and the reset, the system will avoid interpreting the settling oscillations as separate events. These false events would likely overwhelm any polled loop service.

Example: Polled Loop with Delay

Suppose a polled loop system is used to handle an event that occurs aperiodically, but no more than once per second. The event is known to exhibit a strong contact bounce behavior that disappears after no more than 20 ms. A system timer `pause` with 1 ms resolution (or tick length) is available for creating a suitable delay. The event is signaled by an external device that sets a memory location `flag=1` via DMA.

Write a C code fragment for implementing a polled loop structure that is not sensitive to the contact bounce described.

```
for(;;) {                    /* do forever */
        if (flag)            /* check flag */
          {
          process_event(); /* process event */
          pause(21);         /* wait 21 ms */
          flag=0;            /* reset flag */
          }
        }
```

To make sure that all spurious events have disappeared before resetting the flag, the delay length was set 1 ms longer than the known burst of contact oscillation. Assuming the `pause` system call is available, polled loop systems are simple to program and debug, and the response time is easy to determine.

Polled loops are excellent for handling high-speed data channels, especially when the events occur at widely dispersed intervals and the CPU is dedicated to handling the data channel. Polled loop systems may sometimes fail, however, because event bursts are not taken into account. Furthermore, polled loops by

themselves are not sufficient to handle complex systems. In addition, polled loops inherently waste CPU time, especially if the event being polled occurs infrequently.

Cyclic code structures are noninterrupt-driven systems that can provide the illusion of simultaneity by taking advantage of relatively short tasks on a fast processor in a continuous loop.

Example: Cyclic Code Structure

Consider a set of *n* self-contained tasks, `task_1` through `task_n`, in a continuous loop. Provide C code fragments for implementing cyclic code structures with different cycle rates.

```
for(;;) {          /* do forever */
        task_1();
        task_2();
        ...
        task_n();
        }
```

In this case, the cycle rate is the same for each task, as they execute in round-robin fashion. Different cycle rates can be achieved by repeating a task appropriately in the list as shown below.

```
for(;;) {          /* do forever */
        task_1();
        task_2();
        ...
        task_n();
        task_2();
        }
```

Here, `task_2` runs twice in a single cycle, while other tasks are executed only once.

When using the cyclic code approach, the task list can be made dynamically adjustable by keeping a list of pointers to tasks that are managed by the "pseudo operating system" as tasks are created and completed. Intertask communication could be achieved through global variables, for instance. *Global variables*, however, *should always be used with utmost care to avoid data integrity problems*. If each task is relatively short and uniform in size, then adequate reactivity and simultaneity can often be achieved without interrupts. Moreover, if all tasks are carefully constructed including proper syn-

chronization through global variables, complete determinism and definite schedulability can be achieved. Cyclic code structures are, however, inadequate for all but the simplest of real-time systems, because of the difficulties in uniformly dividing the tasks and in the lengthy response times that are potentially created.

State-driven code uses nested `if-then` statements, `case` statements, or finite state machines (FSMs) to break up the processing of individual functions into multiple code segments. The separation of tasks allows each to be temporarily suspended before completion, without loss of critical data. This capability to be suspended and then resumed, in turn, facilitates multitasking via a scheme, such as coroutines, which we will discuss shortly. State-driven code works well in conjunction with cyclic code structures when the tasks are too long or largely nonuniform in size. Finally, because rigorous techniques for reducing the number of states exist, programs based on FSMs can be automatically optimized. A rich theory surrounds FSMs, and relevant theoretical results will be outlined in Chapter 5.

Not all tasks lend themselves naturally to division into multiple states; some tasks are therefore unsuitable for this technique. In addition, the schedule tables needed to implement the code can become quite large. Besides, the manual translation process from the finite state machine notation to tabular form is error-prone.

Coroutines or cooperative multitasking systems require highly disciplined programming and an appropriate application. These types of pseudokernels are employed in conjunction with code driven by FSMs. In this scheme, two or more tasks are coded in the state-driven fashion just discussed, and after each phase is completed, a call is made to a central dispatcher. The dispatcher holds the program counter for a list of tasks that are executed in round-robin fashion; that is, it selects the next task to execute. This task then executes until its next phase is completed, and the central dispatcher is called again. Note that if there is only one coroutine, then it will be repeated cyclically. Such a system is called a cyclic code structure. Communication between the tasks is achieved via global variables. Any data that need to be preserved between dispatches must be deposited to the global variable space.

Example: Coroutines

Consider a system in which two tasks are executing "in parallel" and in isolation. After executing `phase_a1`, `task_a` returns control to the central dispatcher by executing `break`. The dispatcher initiates `task_b`, which executes `phase_b1` to completion before returning control to the dispatcher. The dispatcher then starts `task_a`, which begins `phase_a2`, and so on. An illustrative C code is given below for `task_a` and `task_b`, in the case of three phases.

```
void task_a(void)
{
for(;;)
   {
   switch(state_a)
    {
    case 1: phase_a1();
      break;   /* to dispatcher */
    case 2: phase_a2();
      break;   /* to dispatcher */
    case 3: phase_a3();
      break;   /* to dispatcher */
    }
   }
}
void task_b(void)
{
for(;;)
   {
   switch(state_b)
    {
    case 1: phase_b1();
      break;   /* to dispatcher */
    case 2: phase_b2();
      break;   /* to dispatcher */
    case 3: phase_b3();
      break;   /* to dispatcher */
    }
   }
}
```

Note that the variables `state_a` and `state_b` in the above example are state counters that are global variables managed by the dispatcher. Indeed, for simplicity, intertask communication and synchronization are maintained entirely via global variables and coordinated by the dispatcher. The coroutine approach can be extended to any number of tasks, each divided into an arbitrary number of phases. If each programmer provides calls to the dispatcher at known intervals, the response time is easy to determine because this system is written without hardware interrupts.

A variation of this scheme occurs when a polled loop must wait for a particular event while other processing can continue. Such a scheme reduces the amount of time wasted polling the event flag, and allows for processing time for other tasks. In short, coroutines are the easiest type of "fairness scheduling"

that can be implemented. It should be noted, however, that in most embedded applications, the fairness of scheduling does not have much value, because different tasks are typically of different importance and urgency. In addition, the coroutine tasks can be written by independent parties, and the final number of tasks need not be known beforehand. Certain programming languages, such as Ada, have efficient built-in constructs for implementing coroutines.

In the past, even large and complex real-time applications were successfully implemented using coroutines; for example, IBM's transaction processing system, Customer Information Control System (CICS), was originally constructed entirely via coroutines. Unfortunately, any use of coroutines assumes that each task can relinquish the CPU at regular intervals. It also requires a communication scheme involving global variables, which is usually an undesired approach. Finally, tasks cannot always be decomposed uniformly, which can adversely affect response times, since the minimum response time is asymptotically constrained by the longest phase.

3.1.2 Interrupt-Only Systems

In interrupt-only systems, the "main program" is just a single `jump-to-self` instruction. The various tasks in the system are scheduled via either hardware or software interrupts, whereas task dispatching is performed by interrupt-handling routines.

When pure hardware interrupt scheduling is used, a real-time clock or other external device issues interrupt signals that are directed to an interrupt controller. The interrupt controller issues interrupt signals for the CPU, depending on the order of arrival and priority of the interrupts involved. If the processor architecture supports multiple interrupts, then the hardware handles explicit dispatching as well. If only a single interrupt level is available, then the interrupt-handling routine will have to read the interrupt status register on the interrupt controller, determine which interrupt(s) occurred, and dispatch the appropriate task. Some processors implement this in microcode, and so the operating system designer is relieved of this duty.

In embedded applications, the real-time software usually needs to service interrupts from one or more special purpose devices. In some cases, the software engineer will need to write a device driver from scratch or adapt a generic driver code in which interrupts are needed for synchronization. Whatever the case, it is important for the software engineer to understand interrupt mechanisms and their proper handling.

There are two kinds of interrupts: hardware interrupt and software interrupt (see Sections 2.1.3 and 2.4.1). The fundamental difference between hardware and software interrupts is in the trigger mechanism. While the trigger of a hardware interrupt is an electrical signal from some external device, the trigger of a software interrupt is the execution of a specific machine-language instruction. An additional feature found in most processors is that of an exception, which is an internal interrupt that is triggered by a program's attempt to

perform a special, illegal, or unexpected operation. These three situations cause the CPU to transfer execution to a predetermined location and then execute the interrupt handler (IH) associated with that specific situation.

Hardware interrupts are asynchronous or sporadic in nature, that is, an interrupt may take place at any time. When interrupted, the program is suspended while the CPU invokes the IH. Frequently, an application developer is required to write an IH for a specific type of hardware interrupt. In such case, it is important to understand what constitutes the CPU state, and whether IHs must preserve anything in addition to work registers.

Access to resources shared with an IH is usually controlled by disabling interrupts in the application around any code that reads or writes to the shared resource. Standard synchronization mechanisms cannot be used in an IH, because it is not practical for an IH to wait for a resource to be available. When interrupts are disabled, the system's ability to receive stimuli from the outside world is minimal (only via the nonmaskable interrupt). Therefore, it is important to keep the critical sections of code in which the interrupts are disabled as short as possible. If the interrupt handler takes a significant time to process an interrupt, the external device may be kept waiting too long before its next interrupt is serviced.

Reentrant code can execute simultaneously in two or more contexts. An IH is said to be reentrant if, while the IH is servicing an interrupt, the same interrupt can occur again and the IH can safely process the second occurrence of the interrupt before it has finished processing the first. To create strictly reentrant code, the following general rules must be fulfilled (Simon, 1999):

- Reentrant code is not allowed to use any data in a nonatomic way except when they are saved on the stack.
- Reentrant code is not allowed to call any other code that is not itself reentrant.
- Reentrant code is not allowed to use any hardware resources in a nonatomic way.
- Reentrant code is not allowed to change its own code.

An *atomic* operation refers to a group of suboperations that can be combined to appear as a single (noninterruptible) operation. Regardless of the type of IH to be written, a snapshot of the current system state—called the context—must be preserved upon switching tasks so that it can be restored upon resuming the interrupted task. Context switching is thus the process of saving and restoring sufficient information for a software task so that it can be resumed after being interrupted. The context is ordinarily saved to a stack data structure managed by the CPU. Context-switching time is a major contributor to the composite response time, and, therefore, must be minimized. The pragmatic rule for saving context is simple: *save the minimum amount of information necessary to safely restore any task after it has been interrupted*. This information typically includes: contents of work registers, content of the

program counter register, content of the memory page register, and images of possible memory-mapped I/O locations.

Normally, within the interrupt handlers, interrupts are disabled during the critical context-switching period. Sometimes, however, after sufficient context has been saved, interrupts can be enabled after just a partial context switch in order to handle a burst of interrupts, to detect spurious interrupts, or to manage a time-overload condition.

The flexible stack model for context switching (see Section 3.4) is used mostly in embedded systems where the number of interrupt-driven tasks is fixed. In the stack model, each interrupt handler is associated with a hardware interrupt and is invoked by the CPU, which vectors to the stream of instructions stored at the appropriate interrupt-handler location. The context is then saved to a stack data structure.

Example: Interrupt-Only System

Consider the following pseudocode for a partial real-time system, written in C, and consisting of a trivial `jump-to-self` main program and three interrupt handlers. Each of the interrupt handlers saves the context using the stack model. The interrupt handlers' starting addresses should be loaded into appropriate interrupt vector locations upon system initialization.

```
void main(void)
{
  init();            /* system initialization */
  while(TRUE);       /* jump-to-self */
}
void int_1(void)     /* interrupt handler 1 */
{
  save(context);     /* save context to stack */
  task_1();          /* execute task 1 */
  restore(context); /* restore context */
}
void int_2(void)     /* interrupt handler 2 */
{
  save(context);     /* save context to stack */
  task_2();          /* execute task 2 */
  restore(context); /* restore context */
}
void int_3(void)     /* interrupt handler 3 */
{
  save(context);     /* save context to stack */
  task_3();          /* execute task 3 */
  restore(context); /* restore context */
}
```

In this simplified example, the procedure `save` pushes critical registers and possibly other context information to a stack data structure, whereas `restore` pops that information back from the stack. Both `save` and `restore` may have one argument: a pointer to data structure representing the context information. The stack pointer is automatically adjusted by the CPU, as will be discussed in Section 3.1.4.

3.1.3 Preemptive Priority Systems

A higher-priority task is said to preempt a lower-priority one if it interrupts the executing lower-priority task. Systems that use preemption schemes instead of round-robin or first-come-first-served scheduling are called preemptive priority systems. The priorities assigned to each interrupt are based on the importance and urgency of the task associated with the interrupt. For example, the nuclear power plant supervision system is best designed as a preemptive priority system. While appropriate handling of intruder events, for example, is critical, nothing is more important than processing the core overtemperature alert.

Prioritized interrupts can be either of fixed priority or dynamic priority type. Fixed priority systems are less flexible, since the task priorities cannot be changed after system initialization. Dynamic priority systems, on the other hand, allow the priority of tasks to be adjusted at runtime to meet changing real-time demands. Nonetheless, most embedded systems are implemented with fixed priorities, because there are limited situations in which the system needs to adjust priorities at runtime. A common exception to this practice is, however, the problematic priority inversion condition that is discussed in Section 3.3.6.

Preemptive priority schemes can suffer from resource hogging by higher-priority tasks. This may lead to a lack of available resources for lower-priority tasks. In such case, the lower-priority tasks are said to be facing a serious problem called starvation. The potential hogging/starvation problem must be carefully addressed when designing preemptive priority systems.

A special class of fixed-rate, preemptive priority, interrupt-driven systems, called rate-monotonic systems, comprises those real-time systems where the priorities are assigned so that the higher the execution rate, the higher the priority. This scheme is common in embedded applications like avionics systems, and has been studied extensively. For example, in the aircraft navigation system, the task that gathers accelerometer data every 10 ms has the highest priority. The task that collects gyro data, and compensates these data and the accelerometer data every 40 ms, has the second highest priority. Finally, the task that updates the pilot's display every second has the lowest priority. The valuable theoretical aspects of rate-monotonic systems will be studied in Section 3.2.4.

3.1.4 Hybrid Scheduling Systems

Some hybrid scheduling systems include interrupts that occur at both fixed rates and sporadically. The sporadic interrupts can be used, for example, to

handle a critical error condition that requires immediate attention, and thus have the highest priority. This type of hybrid-interrupt system is common in embedded applications.

Another type of hybrid scheduling system found in commercial operating systems is a combination of round-robin and preemptive priority systems. In these systems, tasks of higher priority can always preempt those of lower priority. However, if two or more tasks of the same priority are ready to run simultaneously, then they run in round-robin fashion, which will be described shortly.

To summarize, interrupt-only systems are straightforward to code and typically have fast response times because task scheduling can be done via hardware. Interrupt-only systems are a special case of foreground/background systems, which are widely used in embedded systems. Typical weaknesses of interrupt-only systems are the time wasted in the `jump-to-self` loop and the difficulty in providing advanced services. Such services include device drivers and interfaces to layered communications networks. Besides, interrupt-only systems are vulnerable to several types of malfunctions due to timing variations, unanticipated race conditions, electromagnetic interferences, and other problems.

Foreground/background systems are a small but meaningful improvement over the interrupt-only systems in that the `jump-to-self` loop is replaced by low-priority code that performs useful processing. Foreground/background systems are, in fact, the most common architecture for embedded applications. They involve a set of interrupt-driven tasks called the foreground and a single noninterrupt-driven task called the background (Fig. 3.3). The foreground tasks run in round-robin, preemptive priority, or hybrid fashion. The background task is fully preemptable by any foreground task, and, therefore, represents the lowest priority task in the real-time system.

All the real-time solutions discussed above can be seen as special cases of the foreground/background system. For example, the simple polled loop is a foreground/background system with no foreground, and a polled loop as a background. Adding a delay (based on the real-time clock interrupt) for avoiding problems related to contact bouncing yields a full foreground/background

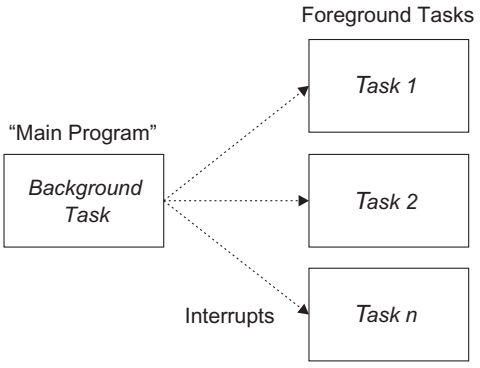


Figure 3.3. An interrupt-driven foreground/background system.

system. State-driven code is a foreground/background system with no foreground and phase-driven code for a background. A coroutine system is just a complicated background task. Finally, interrupt-only systems are foreground/background systems without any useful background processing.

As a noninterrupt-driven task, the background processing should solely include anything that is not time critical. While the background task is the task with the lowest priority, it should always execute to completion provided the system utilization is below 100% and no deadlock occurs. It is common, for instance, to increment a counter in the background task in order to provide a measure of time loading or to detect if any foreground task has hung up. It might also be desirable to provide individual counters for each of the foreground tasks, which are reset in the corresponding tasks. If the background task detects that one of the counters is not being reset often enough, it can be assumed that the corresponding task is not being executed properly and, that some kind of failure is indicated. This is a form of *software watchdog* timer. Certain low-priority self-testing can also be performed in the background. Moreover, low-priority display updates, parameter entry through keypads, logging to printers, or other interfaces to slow devices can be conveniently performed in the background.

Initialization of the foreground/background system usually consists of the following six steps:

1. Disable interrupts.
2. Set up interrupt vectors and stacks.
3. Initialize peripheral interface units and other configurable hardware.
4. Perform self-tests.
5. Perform necessary software initializations.
6. Enable interrupts.

Initialization is always the first part of the background task. It is important to disable interrupts because some systems start up with interrupts enabled, while certain time is definitely needed to set up the entire real-time system. This setup consists of initializing the interrupt vector addresses, setting up the stack or stacks (if it is a multiple-level interrupt system), configuring hardware appropriately, and initializing any buffers, counters, data, and so on. In addition, it is often useful to perform some self-diagnostic tests before enabling interrupts. Finally, real-time processing can begin.

Example: Initialization and Context Saving/Restoring

Suppose it is desired to implement an interrupt handler for a CPU with a single interrupt. That is, we have just one interrupt-driven task in addition to the background task. The `EPI` and `DPI` instructions can be used to enable and disable the maskable interrupt, and it is assumed that upon receiving

an interrupt request, the CPU will hold off other interrupts until explicitly reenabled with an `EPI` instruction.

For simplicity, assume it is adequate to save the four work registers, `RO–R3`, on the stack for context-switching purposes. Here, the context switching involves saving the status of the CPU when it is used by the background task. The foreground task will run to completion so its context is never saved. Further, assume that the interrupt-handler exists in memory location `int`, and the stack should begin from memory location `stack`.

The following assembly code could be used to minimally initialize this foreground/background system:

```
DPI                     ; Disable interrupts.
MOVE handler, &int      ; Set interrupt handler's address.
LDSP &stack             ; Set stack pointer.
EPI                     ; Enable interrupts.
```

Now, the interrupt handler could look as follows:

```
DPI       ; Disable interrupts.
PUSH R0   ; Save R0.
PUSH R1   ; Save R1.
PUSH R2   ; Save R2.
PUSH R3   ; Save R3.
...       ; Code of TASK_1.
POP R3    ; Restore R3.
POP R2    ; Restore R2.
POP R1    ; Restore R1.
POP R0    ; Restore R0.
EPI       ; Enable interrupts.
RETI      ; Return from interrupt.
```

It should be noted that the order of *pushing* registers to the stack must be reverse from the order of *popping* them from the stack. Figure 3.4 shows the behavior of the stack during context saving and restoring. In many processors, specific machine language instructions are available for saving and restoring all relevant registers with just a single instruction.

Example: Background Task

The background task would include a one-time initialization procedure and any continuous processing that is not time critical. If the program were to be written in C, it might appear as:

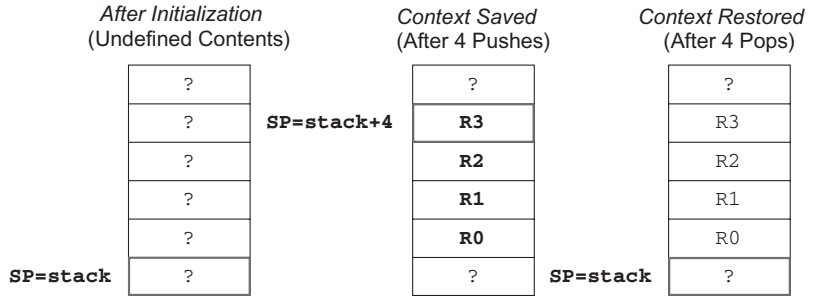


Figure 3.4. The behavior of stack during context saving and restoring; here, the context is solely four registers.

```
void main(void)
{
init();          /* initialize system */
while(TRUE)      /* repetitious loop */
  background(); /* not time critical */
}
```

Foreground/background systems (as well as interrupt-only systems) typically have good response times, since they rely completely on hardware to perform task scheduling. Hence, they are the preferred solution for many embedded real-time systems. Nevertheless, "homegrown" foreground/background systems have at least one noticeable drawback: interfaces to complicated devices and possible communications networks usually need to be written from scratch (occasionally, open-source software can be used or adapted, but then there are software licensing issues to be considered). The process of writing device drivers and interfaces can be tedious and error-prone. In addition, foreground/background systems are best implemented when the number of foreground tasks is fixed and known *a priori*. Although programming languages that support dynamic allocation of memory could handle a variable number of tasks, this can be tricky. In general, the straightforward foreground/background system shares nearly all the weaknesses of interrupt-only systems discussed above.

The foreground/background scheme can be extended into a full-blown operating system by adding typical complementary functions, such as communications network interfaces, device drivers, and real-time debugging tools. Such complete systems are widely available as commercial products. These commercial products rely on relatively complex software structures using round-robin, preemptive priority, or a hybrid of those schemes to provide scheduling, and the operating system itself represents the highest priority task.

3.1.5 The Task Control Block Model

The task control block (TCB) model is the most popular approach for implementing commercial, full-featured, real-time operating systems, because the number of software tasks can vary. This architecture is used particularly in interactive online systems of soft real-time type where tasks, typically associated with users, come and go. The TCB model can be used in round-robin, preemptive priority, or hybrid scheduling systems, although it is often associated with round-robin systems with a fixed time-slicing clock. In preemptive systems, however, it can be used to facilitate dynamic task prioritization. The main drawback of the flexible TCB model is that when a large number of tasks are created, the bookkeeping overhead of the dispatcher may become significant.

In the task control block model, each task is associated with a private data structure, called a task control block depicted in Figure 3.5. The operating system stores these TCBs in one or more data structures, typically in a linked list.

The operating system manages the TCBs by keeping track of the state or status of each task. Typically, a task can be in any one of the four following states:

1. Executing
2. Ready
3. Suspended
4. Dormant

The executing task is the one that is currently executing, and in a uniprocessor environment, there can be only one such task at any time. A task can enter the executing state when it is created (if no other tasks are ready), or from the ready state (if it is eligible to run based on its priority or position in the round-robin queue). When a task is completed, it returns to the suspended state.

Figure 3.5. A typical task-control block.

Tasks in the ready state are those that are ready to execute but are not executing. A task enters the ready state if it was executing and its time slice ran out, or it was preempted by a higher priority task. If it was in the suspended state, then it can enter the ready state if an event that initiates it occurs. Tasks that are waiting for a particular resource, and thus are not ready to execute, are said to be suspended or blocked.

The dormant state is used only in systems where the number of TCBs is fixed. This state allows for determining specific memory requirements beforehand, but limits available system memory. A task in this state is best described as a task that exists but is currently unavailable for scheduling. Once a task has been created, it can become dormant by deleting it.

The operating system is in essence the highest priority task. Every hardware interrupt and every system call (such as a request on a resource) invokes the real-time operating system. The operating system is responsible for maintaining a linked list containing the TCBs of all the ready tasks, and another linked list of those in the suspended state. It also keeps a table of resources and a table of resource requests. Each TCB contains the essential information normally tracked by the interrupt handler. Hence, the difference between the TCB model and the interrupt-handler model is that the resources are managed by the operating system in the TCB model, while in the IH model, tasks track their own resources. The TCB model is advantageous when the number of tasks is indeterminate at design time or can change when the real-time system is in operation. That is, the TCB model is very flexible.

When it is invoked, the operating system checks the ready list to see if another task is eligible for execution. If that is the case, then the TCB of the currently executing task is moved to the end of the ready list (round-robin approach), and the newly eligible task is removed from the beginning of the ready list and its execution begins.

Task state management can be achieved by manipulating the status word appropriately. For example, if all of the TCBs are set up in the list with their status words initialized to "dormant," then each task can be added to active scheduling by simply changing the status to "ready." During runtime, the status words of tasks are updated accordingly, either to "executing" in the case of the next eligible task or back to "ready" in the case of the preempted task. Blocked tasks have their status word changed to "suspended." Completed tasks can be removed from the active task list by resetting the status word to "dormant." This approach reduces runtime overhead, because it eliminates the need for dynamic memory management of the TCBs. It also provides more deterministic performance because the TCB list is of constant size.

In addition to scheduling, the operating system tracks the status of resources waited in the suspended list. If a task is suspended due to some wait for a resource, then that task can enter the ready state only upon availability of the resource. The list structure is used to arbitrate multiple tasks that are suspended on the same resource. If a resource becomes available to a suspended task, then the resource and resource request tables are updated

correspondingly, and the eligible task is moved from the suspended list to the ready list.

3.2 THEORETICAL FOUNDATIONS OF SCHEDULING

In order to take advantage of certain theoretical results in real-time operating systems, a somewhat rigorous formulation is necessary. Most real-time systems are inherently concurrent, that is, their natural interaction with external events typically requires multiple *virtually* (in uniprocessor environments) or *truly* (in multiprocessor environments) simultaneous tasks to cope with various features of control systems, for example. A task is the active object of a real-time system and is the basic unit of processing managed by the scheduler. As a task is running, it dynamically changes its state, and at any time, it may be in one, but only one, of the four states defined above in Section 3.1.5 (i.e., *executing*, *ready*, *suspended*, or *dormant*). In addition, a fifth possible state, *terminated*, is sometimes included. In the "terminated" state, the task has finished its running, has aborted or self-terminated, or is no longer needed.

A partial state diagram corresponding to task (process or thread) states is depicted in Figure 3.6. It should be noted that different operating systems have different naming conventions, but the fundamental states represented in this nomenclature exist in one form or another in all real-time operating systems.

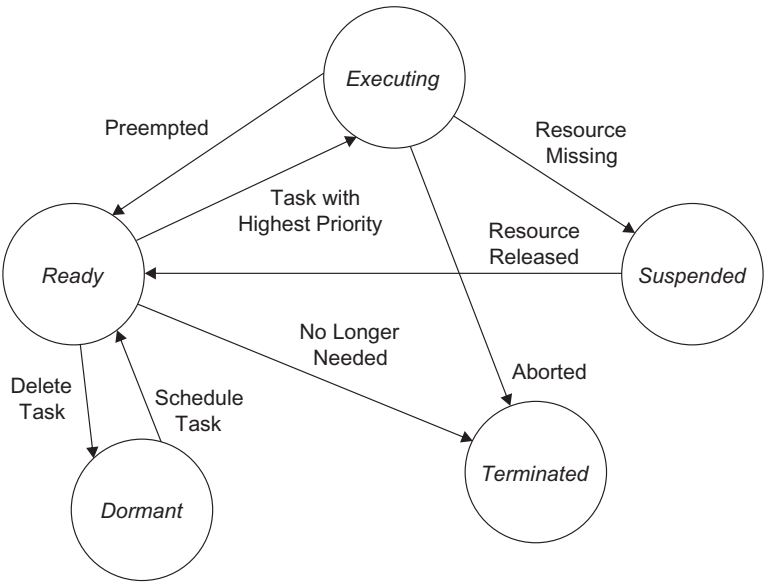


Figure 3.6. A representative task state diagram as a partially defined finite state machine.

Many full-featured operating systems allow processes created within the same program to have unrestricted access to the shared memory through a thread facility. The hierarchical relationship between regular processes and threads is discussed in Section 3.1.

3.2.1 Scheduling Framework

Scheduling is a primary operating system function. In order to meet a program's temporal requirements in some real-time environment, a solid strategy is required for ordering the use of system resources, and a practical scheme is needed for predicting the worst-case performance (or response time) when a particular scheduling policy is applied. There are two principal classes of scheduling policies: pre-runtime and runtime scheduling. The obvious goal of both types of scheduling is to strictly satisfy response time specifications.

In pre-runtime scheduling, the objective is to manually (or semiautomatically) create a feasible schedule offline, which guarantees the execution order of tasks and prevents conflicting access to shared resources. Pre-runtime scheduling also takes into account and reduces the cost of context switching overhead, hence increasing the chance that a feasible schedule can be found.

In runtime scheduling, on the other hand, fixed or dynamic priorities are assigned and resources are allocated on a priority basis. Runtime scheduling relies on relatively complex runtime mechanisms for task synchronization and intertask communication. This adaptive approach allows events to interrupt tasks and demand resources periodically, aperiodically, or even sporadically. In terms of performance analysis, engineers must usually rely on stochastic system simulations to verify these types of designs.

The workload on processors consists of individual tasks each of which is a *unit of processing* to be allocated CPU time and other resources when needed. Every single CPU is assigned to at most one task at any time. Moreover, every task is assigned to at most one CPU at a time. No task (or job) is scheduled before its release time. Each task, τ_i, is typically characterized by the following temporal parameters:

- *Precedence constraints:* Specify if any task needs to precede other tasks.
- *Release time $r_{i,j}$:* The release time of the jth instance of task τ_i.
- *Phase ϕ_i:* The release time of the first instance of task τ_i.
- *Response time:* The time span between the task activation and its completion.
- *Absolute deadline d_i:* The instant by which task τ_i must complete.
- *Relative deadline D_i:* The maximum allowable response time of task τ_i.
- *Laxity type:* The notion of urgency or leeway in a task's execution.
- *Period p_i:* The minimum length of interval between two consecutive release times of task τ_i.

- *Execution time e_i:* The maximum amount of time required to complete the execution of task τ_i when it executes alone and has all the resources it needs.

Mathematically, some of the parameters just listed are related as follows:

$$\phi_i = r_{i,1} \quad \text{and} \quad r_{i,j} = \phi_i + (j-1)\,p_i. \tag{3.1}$$

$d_{i,j}$ is the absolute deadline of the jth instance of task τ_i, and it can be expressed as follows:

$$d_{i,j} = \phi_i + (j-1)\,p_i + D_i. \tag{3.2}$$

If the relative deadline of a periodic task τ_i is equal to its period p_i, then

$$d_{i,k} = r_{i,k} + p_i = \phi_i + kp_i, \tag{3.3}$$

where k is a positive integer greater than or equal to one, corresponding to the kth instance of that task.

Next, a basic task model is presented in order to describe some standard scheduling policies used in real-time systems. The task model has the following simplifying assumptions:

- All tasks in the task set considered are strictly periodic.
- The relative deadline of a task is equal to its period.
- All tasks are independent; there are no precedence constraints.
- No task has any nonpreemptible section, and the cost of preemption is negligible.
- Only processing requirements are significant; memory and I/O requirements are negligible.

For real-time systems, it is of utmost importance that the scheduling algorithm applied produces a predictable schedule, that is, at all times, it is known which task is going to execute next. Many real-time operating systems use a round-robin scheduling policy, because it is simple and predictable. Therefore, it is of interest to describe that popular scheduling algorithm more rigorously.

3.2.2 Round-Robin Scheduling

In a round-robin system, several tasks are executed sequentially to completion, often in conjunction with a cyclic code structure. In round-robin systems with time slicing, each executable task is assigned a fixed time quantum called a time slice in which to execute. A fixed-rate clock is used to initiate an interrupt

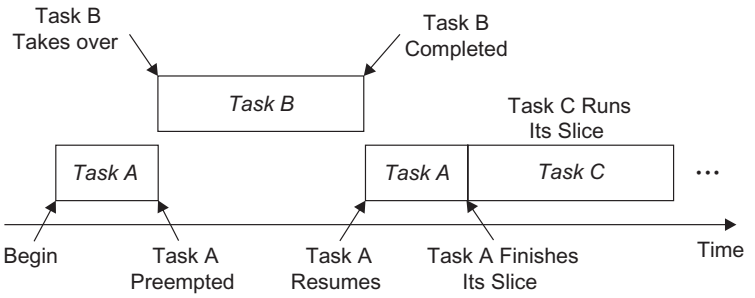


Figure 3.7. Hybrid (round-robin/preemptive) scheduling of three tasks with two priority levels.

at a rate corresponding to the time slice. The dispatched task executes until it completes or its time slice expires, as indicated by the clock interrupt. If the task does not execute to completion, its context must be saved and the task is placed at the end of the round-robin queue. The context of the next executable task in the queue is restored, and it resumes execution. Essentially, round-robin scheduling achieves fair allocation of the CPU resources to tasks of the same priority by straightforward time multiplexing.

Furthermore, round-robin systems can be combined with preemptive priority systems, yielding a kind of hybrid system. Figure 3.7 illustrates such a hybrid system with three tasks and two priority levels. Here, Tasks A and C are of the same priority, whereas Task B is of higher priority. First, Task A is executing for some time before it is preempted by Task B, which executes until completion. When Task A resumes, it continues until its time slice expires, at which time Task C begins executing for its time slice.

3.2.3 Cyclic Code Scheduling

The cyclic code (CC) approach is widely used, as it is simple and generates a complete and highly predictable schedule. The CC refers to a scheduler that deterministically interleaves and makes sequential the execution of periodic tasks on a CPU according to a pre-runtime schedule. In general terms, the CC scheduler is a fixed table of procedure calls, where each task is a procedure, within a single `do` loop.

In the CC approach, scheduling decisions are made periodically, rather than at arbitrary times. Time intervals during scheduling decision points are referred to as frames or minor cycles, and every frame has a length, f, called the frame size. The major cycle is the minimum time required to execute tasks allocated to the CPU, ensuring that the deadlines and periods of all tasks are met. The major cycle or the hyperperiod is equal to the least common multiple (lcm) of the periods, that is, $p_{\text{hyper}} = \text{lcm}(p_1, \ldots p_n)$.

As scheduling decisions are made only at the beginning of every frame, there is no preemption within each frame. The phase, ϕ_i, of each periodic task

is a non-negative integer multiple of the frame size. Furthermore, it is assumed that the scheduler carries out certain monitoring and enforcement actions at the beginning of each frame.

Frames must be sufficiently long so that every task can start and complete within a single frame. This implies that the frame size, f, is to be longer than the execution time, e_i, of every task, τ_i, that is,

$$C_1 : f \geq \max_{i \in [1,n]} (e_i). \tag{3.4}$$

In order to keep the length of the cyclic schedule as short as possible, the frame size, f, should be chosen so that the hyperperiod has an integer number of frames:

$$C_2 : \lfloor p_{\text{hyper}} / f \rfloor - p_{\text{hyper}} / f = 0. \tag{3.5}$$

Moreover, to ensure that every task completes by its deadline, frames must be short so that between the release time and deadline of every task, there is at least one frame. The following relation is derived for a worst-case scenario, which occurs when the period of a task starts just after the beginning of a frame, and, consequently, the task cannot be released until the next frame:

$$C_3 : 2f - \gcd(p_i, f) \leq D_i. \tag{3.6}$$

where "gcd" is the greatest common divisor, and D_i is the relative deadline of task τ_i. This condition should be evaluated for all schedulable tasks.

Example: Calculation of Frame Size

To demonstrate the calculation of frame size, consider a set of three tasks specified in Table 3.1. The hyperperiod, p_{hyper}, is equal to 660, since the least common multiple of 15, 20, and 22 is 660. The three necessary conditions, C_1, C_2, and C_3, are evaluated as follows:

TABLE 3.1. Example Task Set for Frame-Size Calculation

τ_i	p_i	e_i	D_i
τ_1	15	1	15
τ_2	20	2	20
τ_3	22	3	22

$$C_1 : f \geq \max_{i \in [1,3]}(e_i) \Rightarrow f \geq 3$$

$$C_2 : \lfloor p_{\text{hyper}} / f \rfloor - p_{\text{hyper}} / f = 0 \Rightarrow f = 2, 3, 4, 5, 6, 10, \ldots$$

$$C_3 : 2f - \gcd(p_i, f) \leq D_i \Rightarrow f = 2, 3, 4, 5, 6, 7$$

From these conditions that must be valid *simultaneously*, it can be inferred that a possible value for f could be any one of the values of 3, 4, 5, or 6.

3.2.4 Fixed-Priority Scheduling: Rate-Monotonic Approach

In the fixed-priority scheduling policy, the priority of each periodic task is fixed relative to other tasks. A seminal fixed-priority algorithm is the rate-monotonic (RM) algorithm of Liu and Leyland (Liu and Layland, 1973). It is an optimal fixed-priority algorithm for the basic task model previously described, in which a task with a shorter period is given a higher priority than a task with a longer period. The theorem, known as the rate-monotonic theorem is, from the practical point of view, the most important and useful result of real-time systems theory. It can be stated as follows (Liu and Layland, 1973).

Theorem: Rate-Monotonic

Given a set of periodic tasks and preemptive priority scheduling, then assigning priorities such that the tasks with shorter periods have higher priorities (rate-monotonic), yields an optimal scheduling algorithm.

In other words, the optimality of the RM algorithm implies that *if a schedule that meets all the deadlines exists with fixed priorities, then the RM algorithm will produce a feasible schedule*. The formal proof of the theorem is rather involved. However, a compact sketch of the proof by Shaw uses an effective inductive argument (Shaw, 2001).

Proof: Rate-Monotonic Theorem

Initial step: consider two fixed- but non-RM priority tasks $\tau_1 \equiv \{p_1, e_1, D_1\}$ and $\tau_2 \equiv \{p_2, e_2, D_2\}$, where τ_2 has the highest priority and $p_1 < p_2$. Suppose both tasks are released at the same time. It is obvious that this leads to the worst-case response time for τ_1. However, at this point, in order for both tasks to be schedulable, it is necessary that $e_1 + e_2 \leq p_1$; otherwise, τ_1 could not meet its period (or deadline). Because of this explicit relation between the execution times and the period of τ_2, we can obtain a feasible schedule by simply reversing priorities, thereby scheduling τ_1 first—that is, with RM assignment. Thus, the RM theorem holds true at least with two tasks.

Induction step: suppose next that n tasks, $\tau_1, \ldots \tau_n$, are schedulable according to RM, with priorities in ascending order, but the assignment is

not RM. Let τ_i and τ_{i+1}, $1 \leq i < n$, be the first two tasks with non-RM priorities. That is, $p_i < p_{i+1}$. This sketch of the proof proceeds by interchanging the priorities of these two tasks and showing the set is still schedulable using the initial step result with $n = 2$. The inductive proof continues by interchanging non-RM task pairs in this way until the assignment becomes RM. Therefore, if some fixed-priority assignment can produce a feasible schedule, so can RM assignment.

A critical instant of a task is defined to be an instant at which a request for that task will have the largest response time. Liu and Layland further proved that a critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher-priority tasks. It is then shown that to check for rate-monotonic schedulability, it suffices to check the case where all tasks phases are zero (Liu and Layland, 1973). This useful result was used also in the above proof sketch.

Example: Rate-Monotonic Scheduling

To illustrate rate-monotonic scheduling, consider the set of three tasks defined in Table 3.2. All tasks are released at time 0. Since task τ_1 has the shortest period, it is the highest priority task and is scheduled first. The successful RM-schedule for the task set is depicted in Figure 3.8. Note that at time 4, the second instance of task τ_1 is released, and it preempts the currently running task τ_3, which has the lowest priority. The utilization factor, u_i, is equal to the fraction of time a task with execution time e_i and period p_i keeps the CPU busy. Recall that the overall CPU utilization factor for n tasks is given by Equation 1.2, and is here $U = \sum_{i=1}^{3} e_i / p_i = 0.9$ (corresponds to the "dangerous" zone of Table 1.3).

From a practical point of view, it is important to know under what conditions a feasible schedule exists in the fixed-priority case. The following theorem (Liu and Layland, 1973) yields a schedulable utilization of the rate-monotonic algorithm (RMA). Note that the relative deadline of every task is assumed to be equal to its period.

TABLE 3.2. Example Task Set for RM Scheduling

τ_i	p_i	e_i	u_i
τ_1	4	1	0.25
τ_2	5	2	0.4
τ_3	20	5	0.25

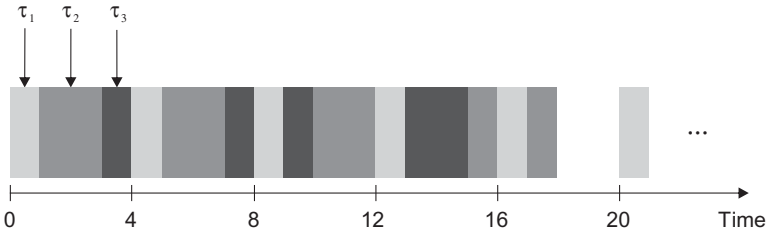


Figure 3.8. Rate-monotonic task schedule for the task set of Table 3.2.

TABLE 3.3. Upper Bounds of the CPU Utilization Factor, *U* (%), for *n* Periodic Tasks Scheduled Using the RMA

n	1	2	3	4	5	6	. . .	→∞
U (%)	100	83	78	76	74	73	. . .	69

Theorem: The RMA Bound

Any set of *n* periodic tasks is RM schedulable if the CPU utilization factor, *U*, is no greater than $n(2^{1/n} - 1)$.

This means that whenever *U* is at or below the given utilization bound, a successful schedule can be constructed with RM. In the limit, when the number of tasks $n \to \infty$, the maximum utilization limit is

$$\lim_{n\to\infty} n\left(2^{1/n} - 1\right) = \ln 2 \approx 0.69 \tag{3.7}$$

Table 3.3 shows the RMA bounds (%) for various values of *n*. Note that these RMA bounds are *sufficient*, but not necessary. That is, it is not uncommon in practice to compose a periodic task set with CPU utilization greater than the corresponding RMA bound but still being RM schedulable. For example, the task set shown in Table 3.2 has a total utilization of 0.9, which is greater than the RM utilization bound of 0.78 for three tasks, but it is still schedulable using the RM policy as illustrated in Figure 3.8.

3.2.5 Dynamic Priority Scheduling: Earliest Deadline First Approach

In contrast to fixed-priority algorithms, in dynamic-priority schemes, the priority of a task with respect to that of other tasks changes as tasks are released and completed. One of the most well-known dynamic algorithms, the earliest deadline first algorithm (EDFA), deals with deadlines rather than execution times. At any point of time, the ready task with the earliest deadline has the highest priority. The following theorem gives the necessary and sufficient con-

dition under which a feasible schedule exists under the earliest deadline first (EDF) priority scheme (Liu and Layland, 1973).

Theorem: The EDFA Bound

A set of n periodic tasks, each of which relative deadline equals its period, can be feasibly scheduled by the EDFA if and only if $U = \sum_{i=1}^{n} e_i / p_i \leq 1$.

EDF is optimal for a uniprocessor with task preemption being allowed. In other words, if a feasible schedule exists, then the EDF policy will also produce a feasible schedule. And there is never processor idling prior to a missed deadline.

Example: Earliest Deadline First Scheduling

To illustrate earliest deadline first scheduling, consider the pair of tasks defined in Table 3.4, with $U = 0.97$ ("dangerous"). The EDF schedule for this task pair is depicted in Figure 3.9. Although τ_1 and τ_2 release simultaneously, τ_1 executes first because its deadline is earliest. At $t = 2$, τ_2 can begin to execute. Even though τ_1 releases again at $t = 5$, its deadline is not earlier than that of τ_2. This regular sequence continues until time $t = 15$, when τ_2 is preempted, because its deadline is later ($t = 21$) than the deadline of τ_1 ($t = 20$); τ_2 resumes when τ_1 completes.

TABLE 3.4. Task Pair for the Example of EDF Scheduling

τ_i	p_i	e_i	u_i
τ_1	5	2	0.4
τ_2	7	4	0.57

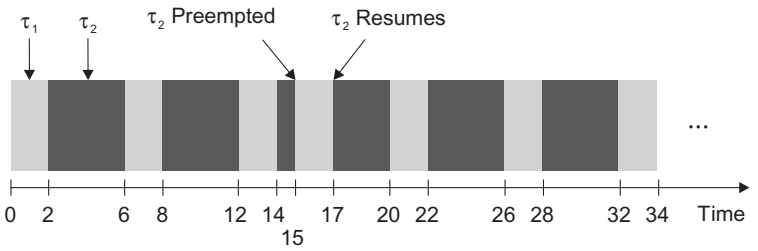


Figure 3.9. Earliest deadline-first task schedule for the task pair of Table 3.4.

What are the principal differences between RM and EDF scheduling? Schedulable CPU utilization is an objective measure of performance of algorithms used to schedule periodic tasks. It is desired that a scheduling algorithm yields a maximally schedulable utilization. By this criterion, dynamic-priority algorithms are evidently better than fixed-priority scheduling algorithms. Thus, EDF is more flexible and achieves better utilization. However, the temporal behavior of a real-time system scheduled by a fixed-priority algorithm is more predictable than that of a system scheduled according to a dynamic-priority algorithm. In case of overloads, RM is stable in the presence of missed deadlines; the same lower-priority tasks miss their deadlines every time. There is no effect on higher-priority tasks. In contrast, when tasks are scheduled using EDF, it is difficult to predict which tasks will miss their deadlines during possible overload conditions. Also, note that a late task that has already missed its deadline has a higher priority than a task whose deadline is still in the future. If the execution of a late task is allowed to continue, this may cause numerous other tasks to be late. An effective overrun management scheme is hence needed for dynamic-priority algorithms employed in such systems where occasional overload conditions cannot be avoided. Finally, as a general comment, RM tends to need more preemption; EDF only preempts when an earlier deadline task arrives.

3.3 SYSTEM SERVICES FOR APPLICATION PROGRAMS

The basic task model being considered in Section 3.2 assumes that all tasks are independent, and they can be preempted at any point of their execution. However, from a practical viewpoint, this assumption is unrealistic, and coordinated task interaction is needed in most real-time applications. In this section, the use of synchronization mechanisms to maintain the consistency and integrity of shared data or resources is discussed together with various approaches for intertask communication. The main concern is how to minimize time-consuming blocking that may arise in a real-time system when concurrent tasks use shared resources. Related to this concern is the issue of sharing critical resources that can only be used by one task at a time. Moreover, the potential problems of *deadlock* and *starvation* should always be kept in mind when designing and implementing resource-sharing schemes.

In Section 3.2, fundamental techniques for multitasking were discussed in a way that each task operated in isolation from the others. In practice, strictly controlled mechanisms are needed that allow tasks to communicate, share resources, and synchronize their activities. Most of the mechanisms and phenomena discussed in this section are easy to understand casually, but a deep understanding may be harder to attain. Misuse of these techniques, semaphores in particular, can lead to disastrous effects—such as a deadlock.

3.3.1 Linear Buffers

A variety of mechanisms can be employed to transfer data between individual tasks in a multitasking system. The simplest and fastest among these is the use of global variables. Global variables, though considered contrary to good software engineering practices, are still used successfully in high-speed operations with semaphore protection. One of the potential problems related to using global variables alone is that tasks of higher priority can preempt lower-priority tasks at inopportune times, hence corrupting the global data.

In another typical case, one task may produce data at a constant rate of 1000 units per second, whereas another task may consume these data at a rate less than 1000 units per second. Assuming that the data production burst is relatively short, the slower consumption rate can be accommodated if the producer task fills an intermediate storage buffer with the data. This linear buffer holds the excess data until the consumer task can process it; such a buffer can be a queue or some other data structure. Naturally, if the consumer task cannot keep up with the speed of producer task, overflow problems occur. Selection of an appropriate buffer size is critical in avoiding such problems.

A common use of global variables is in double buffering. This flexible technique is used when time-correlated data need to be transferred between tasks of different rates, or when a full set of data is needed by one task, but can only be supplied gradually by another task. This situation is clearly a variant of the classic bounded buffer problem in which a block of memory is used as a repository for data produced by "writers" and consumed by "readers." A further generalization is the readers-and-writers problem in which there are multiple readers and multiple writers of a shared resource, as shown in Figure 3.10. The bounded buffer can only be written to or read from by one writer or reader at a time.

Many telemetry systems, which transmit blocks of data from one unit to another, use double-buffering schemes with a software or hardware switch to alternate the buffers. This effective strategy is also used routinely in graphics

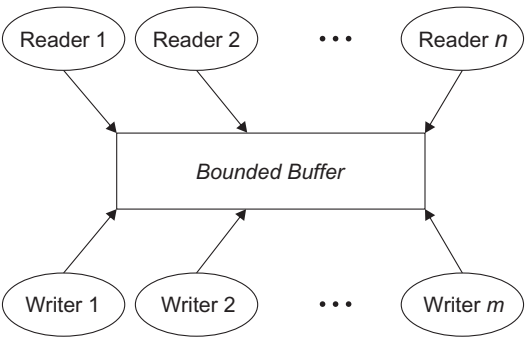


Figure 3.10. Readers and writers problem, with n readers and m writers; the shared resource is a bounded buffer.

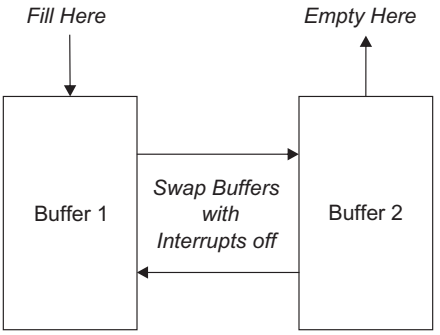


Figure 3.11. Double-buffering; two identical buffers are filled and emptied by alternating tasks; switching is accomplished by either software or hardware.

interfaces, navigation equipment, elevator control systems, and many other places. For example, in the operator display for the pasta sauce factory, suppose lines and other graphics objects are drawn on the screen one by one until the entire image is completed. In this animated system, it is undesirable to see the object-by-object drawing process. If, however, the software draws the full image first on a hidden screen while displaying the other, and then flips the hidden/shown screens, the individual drawing actions will not disturb the process supervisor (Fig. 3.11).

Example: Time-Correlated Buffering

Consider again the inertial measurement unit implemented as a preemptive priority system. It reads x, y, and z accelerometer pulses in a 10-ms task. These raw data are to be processed in a 40-ms task, which has lower priority than the 10-ms task (RM scheduling). Therefore, the accelerometer data processed in the 40-ms task must be time-correlated; that is, it is not allowed to process x and y accelerometer pulses from time instant k along with z accelerometer pulses from instant $k + 1$. This undesired scenario could occur if the 40-ms task has completed processing the x and y data, but gets interrupted by the 10-ms task before processing the z data. To avoid this problem, use buffered variables xb, yb, and zb in the 40-ms task, and buffer them with *interrupts disabled*. The 40-ms task might contain the following C code to handle the buffering:

```
introff();          /* disable interrupts */
xb=x;               /* buffer x data */
yb=y;               /* buffer y data */
zb=z;               /* buffer z data */
intron();           /* enable interrupts */
process(xb,yb,zb); /* use buffered data */
...
```

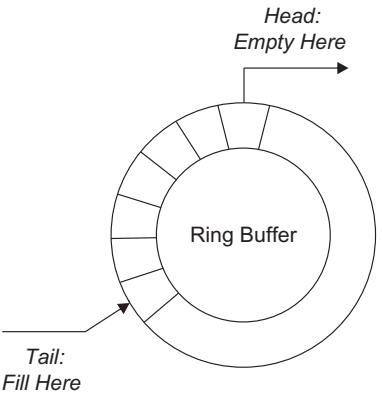


Figure 3.12. A ring buffer; tasks write data to the buffer at the tail index and read from the head index.

3.3.2 Ring Buffers

A special data structure, called a ring buffer (or circular queue), is used in the same way as a regular queue and can be used to solve the problem of synchronizing multiple reader and writer tasks. Ring buffers are easier to manage than double buffers or regular queues when there are more than two readers or writers. In the ring buffer, simultaneous input and output operations are possible by maintaining separate head and tail indices. Data are loaded at the tail and read from the head. This is illustrated in Figure 3.12.

Example: Ring Buffering

Suppose the ring buffer is a data structure of type `ring_buffer` that includes an integer array of size `n` called `contents`, as well as the head and tail indices called `head` and `tail`, respectively. Both of these indices are initialized to `0`, that is, the start of the buffer, as shown below:

```
typedef struct ring_buffer
{
  int contents[n]; /* buffer area */
  int head=0;      /* head index */
  int tail=0;      /* tail index */
}
```

An implementation of the `read(data,&s)` and `write(data,&s)` operations, which reads from and writes to the ring buffer `s`, respectively, are given below in C code:

```
void read(int data,ring_buffer *s)
{
 if (s->head==s->tail)
```

```
  data=NULL; /* buffer underflow */
 else
 {
  data=s->contents+head;    /* read data */
  s->head=(s->head+1) % n; /* update head */
 }
}
void write(int data,ring_buffer *s)
{
 if ((s->tail+1) % n==head)
  error();   /* buffer overflow */
 else
 {
  s->contents+tail=data;    /* write data */
  tail=(tail+1) % n;        /* update tail */
 }
}
```

An additional piece of code, `error()`, is needed to handle the possible overflow condition in the ring buffer. In addition, the task using the ring buffer needs to test the read data for the underflow (`NULL`) value. An overflow occurs when an attempt is made to write data to a full buffer. Underflow, on the other hand, is the condition when a task attempts to retrieve data from an empty buffer.

3.3.3 Mailboxes

Mailboxes provide an intertask communication mechanism, and are available in many commercial operating systems. A mailbox is actually a special memory location that one or more tasks can use to transfer data, or more generally for synchronization. The tasks rely on the kernel to allow them to write to the mailbox via a `post` operation or to read from it via a `pend` operation—*direct access to any mailbox is not allowed*. Two system calls, `pend(d,&s)` and `post(d,&s)`, are used to *receive* and *send* mail, respectively. Here, the first parameter, `d`, is the mailed data and the second parameter, `&s`, is the mailbox location. Recall that C language passes parameters by value unless forced to pass by reference with a pointer; therefore, when calling functions like `pend` and `post`, the dereferencing operator "`&`" must be used.

The important difference between the `pend` operation and simply polling the mailbox location is that the pending task is *suspended* while waiting for data to appear. Thus, no CPU time is wasted for polling the mailbox.

The mail that is passed via a mailbox can be a flag used to protect a critical resource (called a key), a single piece of data, or a pointer to a data structure. For example, when the key is taken from the mailbox, the mailbox is emptied. Thus, although several tasks can `pend` on the same mailbox, only one task can

TABLE 3.5. Task Resource Request Table

Task No.	Resource	Status
10	A/D converter	Has it
11	Mailbox 1	Has it
12	Mailbox 1	Pending

TABLE 3.6. Resource Table Used Together with Task Resource-Request Table

Resource	Status	Owner
A/D converter	Busy	10
Mailbox 1	Busy	11
Mailbox 2	Empty	None

receive the key. Since the key represents access to a critical resource, simultaneous access is precluded.

Mailboxes are typically implemented in operating systems based on the TCB model with a *supervisor task* of highest priority. A status table containing a list of tasks and needed resources (e.g., mailboxes, A/D converters, printers, etc.) is kept along with a second table containing a list of resources and their current states. For example, in Tables 3.5 and 3.6, three resources exist: an A/D converter and two mailboxes. Here, the A/D converter is being used by task 10, while Mailbox 1 is being used (read from or written to) by task 11. Task 12 is pending on Mailbox 1, and is suspended because the resource needed is not available. Mailbox 2 is currently not being used or pended on by any task.

When the supervisor task is invoked by some system call or hardware interrupt, it checks those tables to see if some task is pending on a mailbox. If the corresponding mail is available (mailbox status is "full"), then the state of that task is changed to ready. Similarly, if a task posts to a mailbox, then the supervisor task must ensure that the mail is placed in the mailbox and its status updated to "full."

Sometimes, there are additional operations available on the mailbox. For example, in certain implementations, an `accept` operation is provided; `accept` allows tasks to read the mail if it is available, or immediately return an error code if the mail is not available. In other implementations, the `pend` operation is equipped with a timeout to prevent deadlocks. This feature is particularly useful in autonomous systems operating in harsh environments with high levels of electromagnetic interferences (to recover from sporadically vanishing interrupts).

Some operating systems support a special type of mailbox that can queue multiple pend requests. These systems provide `qpost`, `qpend`, and `qaccept` operations to post, pend, and accept data to/from the queue. In this case, the

queue can be regarded as any array of mailboxes, and its implementation is facilitated through the same resource tables already discussed.

Mailbox queues should not be used ineffectively to pass arrays of data; pointers should be preferred in such purposes. A variety of device servers, where a pool of devices is involved, can be conveniently implemented using mailbox queues. Here, a ring buffer holds requests for a device, and mailbox queues are used at both the head and tail to control access to the ring buffer. Such a secure scheme is useful in the construction of device-controlling software.

3.3.4 Semaphores

Multitasking systems are usually concerned with resource sharing. In most cases, these resources can only be used by a single task at a time, and use of the resource cannot be interrupted. Such resources are said to be serially reusable, and they include certain peripherals, shared memory, and also the CPU. While the CPU protects itself against simultaneous use, the code that interacts with the other serially reusable resources cannot do the same. Such a segment of code is called a critical region. If two tasks enter the same critical region simultaneously, even a catastrophic error may occur.

To illustrate, consider two tasks, `Task_A` (high priority) and `Task_B` (low priority), which are running in a preemptive priority system and sharing a single printer. `Task_B` prints the message "`Louisville is in Kentucky`" and `Task_A` prints the message "`Finland, Europe `". In the midst of printing, `Task_B` is interrupted by `Task_A`, which begins and completes its printing. The result is the incorrect printout: "`Louisville is in` **`Finland, Europe`** `Kentucky`". The emphasis is placed on the text of `Task_A` to highlight that it interrupted the text of `Task_B`.

Truly serious complications could arise in practice if both tasks were performing measurements by a single A/D converter for selectable (an analog multiplexer in front of the A/D converter; see Section 2.4.3) quantities in an embedded control system. Overlapping use of a serially reusable resource results in a collision. Hence, the imperative concern is to provide a reliable mechanism for preventing collisions.

The most common mechanism for protecting critical resources involves a binary variable called a semaphore, which is functionally similar to the traditional railway semaphore device. A semaphore, `s`, is a specific memory location that acts as a lock to protect critical regions. Two system calls, `wait(&s)` and `signal(&s)`, are used either to *take* or to *release* the semaphore. Analogous to the mailboxes discussed above, tasks rely on the kernel to allow them to take the semaphore via a `wait` operation or to release it via a `signal` operation—*direct access to any semaphore is not allowed*. The `wait(&s)` operation suspends the calling task until the semaphore `s` is available, whereas the `signal(&s)` operation makes the semaphore `s` available. Thus, each `wait`/`signal` call also activates the scheduler.

Any code that enters a critical region is surrounded by appropriate calls to `wait` and `signal`. This prevents more than one task from entering the critical region simultaneously.

Example: Serially Reusable Resource

Consider a preemptive priority embedded system with separate measurement channels for acceleration and temperature, and a single A/D converter to be used by `Task_1` and `Task_2` for periodically measuring those two quantities. Before starting an A/D conversion, the desired measurement channel must be selected. How would you share the serially reusable resource with `Task_1` (high priority) and `Task_2` (low priority)?

A binary semaphore, `s`, can be used to protect the critical resource, and it should be initialized to `1` ("one resource available") before either task is started. Proper use of the semaphore `s` is shown by the following pseudocode fragments:

```
/* Task_1 */
...
wait(&s);  /* wait until A/D available */
select_channel(acceleration);
a_data=ad_conversion(); /* measure */
signal(&s) /* release A/D */
...

/* Task_2 */
...
wait(&s);  /* wait until A/D available */
select_channel(temperature);
t_data=ad_conversion(); /* measure */
signal(&s) /* release A/D */
...
```

If the semaphore primitives are not provided by the operating system, mailboxes can be used to implement binary semaphores. Using a dummy mail, `key`, the `wait` operation can be implemented as shown below:

```
void wait(int s)
{
 int key=0;
 pend(key,&s);
}
```

The accompanying `signal` operation utilizes the mailbox `post` operation in the following way:

```
void signal(int s)
{
 int key=0;
 post(key,&s);
}
```

Until now, the semaphores have been called binary semaphores, because they can only take one of two values: 0 or 1. Alternatively, a counting semaphore (or general semaphore) can be used to protect pools of resources. This particular semaphore must be initialized to the total number of free resources before real-time processing can commence. For instance, when using ring buffers, the data access is often synchronized with a counting semaphore initialized to the size of ring buffer. Corresponding wait and signal semaphore primitives, `multi_wait` and `multi_signal`, are needed with counting semaphores. Some real-time kernels provide only binary semaphores, while others have just counting semaphores. The binary semaphore is a special case of the counting semaphore, where the count never exceeds 1. In some operating systems, the `wait`/`multi_wait` operation is equipped with a timeout to recover from possible deadlocks.

Semaphores provide an effective solution to a variety of resource-sharing problems. However, their trouble-free usage requires strict rules for applying them, a high level of programming discipline, and careful coordination between different programmers within the software project. Typical problems associated with the use of semaphores are listed below (Simon, 1999):

- *The use of a specific semaphore is forgotten*: Leads to conflicts between simultaneous users of a single resource or shared data.
- *A wrong semaphore is used in error*: Equally serious as forgetting to use a specific semaphore.
- *The semaphore is held for an overly long time*: Other tasks—even higher-priority ones—may miss their deadlines.
- *The semaphore used is not released at all*: Eventually leads to a deadlock.

All these problems are clearly programmer originated, and should be managed, therefore, as an integral part of the product development and quality control processes applied in the organization—through the entire software lifecycle.

3.3.5 Deadlock and Starvation Problems

When several tasks are competing for the same set of serially reusable resources, then a deadlock situation (or deadly embrace) may ensue. The notion of deadlock is best illustrated by an example.

Example: Deadlock Problem

Suppose TASK_A requires resources 1 and 2, as does Task_B. Task_A is in possession of resource 1, but is waiting on resource 2. Task_B is in possession of resource 2, but is waiting on resource 1. Neither Task_A nor Task_B will relinquish the resource until its other request is satisfied. The cumbersome situation is illustrated as follows, where two semaphores, s1 and s2, are used to protect resource 1 and resource 2, respectively:

```
/* Task_A */
...
wait(&s1); /* wait for resource 1 */
...    /* use resource 1 */
wait(&s2); /* wait for resource 2 */
deadlock here
...    /* use resource 2 */
signal(&s2); /* release resource 2 */
signal(&s1); /* release resource 1 */
...

/* Task_B */
...
wait(&s2); /* wait for resource 2 */
...    /* use resource 2 */
wait(&s1); /* wait for resource 1 */
deadlock here
...    /* use resource 1 */
signal(&s1); /* release resource 1 */
signal(&s2); /* release resource 2 */
...
```

Pictorially, if semaphore s1 guards resource 1 and semaphore s2 guards resource 2, then the realization of resource sharing might appear as the resource diagram in Figure 3.13.

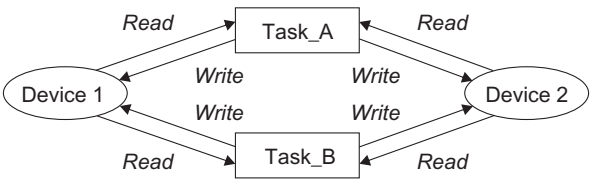


Figure 3.13. Deadlock realization as a loop in a resource diagram.

Deadlock is a burdensome problem, because it cannot always be detected even through relatively comprehensive testing. Besides, it may occur very infrequently, making the pursuit of a known deadlock problem difficult. The general solution to the deadlock problem is by no means straightforward and may have unintended consequences, such as increasing response times.

Although it is unlikely that such an obvious deadlock scenario as the one just described is going to be created in practice, bad designs and their careless implementations might be masked by complex structures. If the system resource diagram contains subgraphs that resemble Figure 3.13, that is, *it contains loops*, then a deadlock can occur. Petri-net simulation and analysis can be helpful in identifying such situations (see Chapter 5).

In a deadlock condition, two or more tasks cannot advance due to simultaneously waiting for some resource from each other, and this condition lasts infinitely. A related problem, starvation, differs from deadlock in that at least one task is satisfying its requirements, while one or more others are not able to fulfill their needs within a reasonable period (Tai, 1994). The following four conditions are necessary for a deadlock (Havender, 1968):

1. Mutual exclusion
2. Circular wait
3. Hold and wait
4. No preemption

Mutual exclusion applies to those resources that cannot be shared, for instance, communications channels, disk drives, and printers. It can be relieved or even eliminated using special buffering services, such as daemons and spoolers, that allow these resources to be virtually shareable by multiple tasks.

The *circular wait* condition occurs when a sequential chain of tasks exists that holds resources needed by other tasks further down the chain (such as in typical cyclic code structures). One way to eliminate circular wait is to impose an explicit ordering on the resources and to force all tasks to request *all* resources above the number of the lowest one needed. For example, suppose that a collection of devices is ranked as shown in Table 3.7. Now, if some task needs to use just the printer, it will be assigned the printer, scanner, and monitor. Then, if another task requests the monitor only, it will have to

TABLE 3.7. Device Ordering Scheme to Eliminate the Circular Wait Condition

Device	Number
Disk drive	1
Printer	2
Scanner	3
Monitor	4

wait until the first task releases the reserved three resources—although the first task does not actually use the monitor. It is easy to see that such a straightforward approach eliminates the circular wait at the potential cost of starvation.

The *hold-and-wait* condition occurs when tasks request a resource and then lock that resource until other subsequent resource requests are also filled. One solution to this problem is to allocate to a task all potentially required resources at the same time, as in the previous case. However, this approach can lead to starvation in other tasks. Another solution is to never allow a task to lock more than one resource at a time. For example, when copying one semaphore-protected file record to another file, lock the source file and read the record, unlock that file, lock the destination file and write the record, and finally unlock that file. This, of course, can lead to inefficient resource utilization, as well as apparent windows of opportunity for other tasks to interrupt and interfere with disk-drive utilization.

Finally, eliminating the *no-preemption* condition will preclude a deadlock. This can be accomplished, for example, by using a timeout with the problem-causing `wait` (or `pend`) system call. However, such a violent action leads to starvation in the low-priority task that was using the preempted resource, as well as to other potential problems. For instance, what if the low-priority task had locked the printer for output, and now the high-priority task starts printing? Nevertheless, this is the ultimate solution to any deadlock condition.

In complex real-time systems, the detection and identification of deadlock may not always be easy, although watchdog timers or real-time debuggers can be used for this purpose. Therefore, the best way to deal with deadlock is to avoid it altogether! Several techniques for avoiding deadlock are available. For example, if the semaphores (or "key" mailboxes) protecting critical resources are implemented with timeouts, then true deadlocking cannot occur, but starvation of one or more tasks is highly probable.

Suppose a lock refers to any semaphore used to protect a critical region. Then the following six-step resource-management approach is recommended to help avoid deadlock:

1. Minimize the number of critical regions and their length.
2. All tasks must release any lock as soon as possible.
3. Do not suspend any task while it controls a critical region.
4. All critical regions must be 100% error free.
5. Do not lock any devices in interrupt handlers.
6. Always perform validity checks on pointers used within critical regions.

Nevertheless, rules 1–6 may be difficult to fulfill, and, hence, additional means are often necessary to avoid deadlocks.

Assuming that a deadlock situation can be detected by using a semaphore timeout, what could be done about it? If the deadlock appears to occur very

infrequently, for instance, once per month, and the real-time system is not a critical one, simply ignoring the problem may be acceptable. For example, if in a console game this problem is known to occur rarely, the effort needed to identify and correct the problem may not be justified, given the cost and purpose of the system. Nonetheless, for any hard or firm real-time system discussed in Chapter 1, ignoring this problem is naturally unacceptable. How about handling the deadlock by resetting the system (possibly by a watchdog timer; see Section 2.5.2)? Again, this may be unacceptable for critical systems. Finally, if a deadlock is detected, some form of rollback to a predeadlock state could in certain cases be performed, although this may lead to a recurrent deadlock; and particular operations, such as writing to some files or peripherals, cannot always be rolled back without special hardware/software arrangements.

3.3.6 Priority Inversion Problem

When a lower-priority task blocks a higher-priority one, a priority inversion is said to occur. Consider the following example, where priority inversion takes place.

Example: Priority Inversion Problem

Let three tasks, τ_1, τ_2, and τ_3, have decreasing priorities (i.e., $\tau_1 \succ \tau_2 \succ \tau_3$, where "$\succ$" is the precedence symbol), and τ_1 and τ_3 share some data or resource that requires exclusive access, while τ_2 does not interact with either of the other two tasks. Access to the critical section is carried out through the `wait` and `signal` operations on semaphore `s`.

Now, consider the following execution scenario, illustrated in Figure 3.14. Task τ_3 starts at time t_0, and locks semaphore `s` at time t_1. At time t_2, τ_1 arrives and preempts τ_3 inside its critical section. After a while, τ_1 requests to use the shared resource by attempting to lock `s`, but it gets blocked, as τ_3 is currently using it. Hence, at time t_3, τ_3 continues to execute inside its critical section. Next, when τ_2 arrives at time t_4, it preempts τ_3, as it has a higher

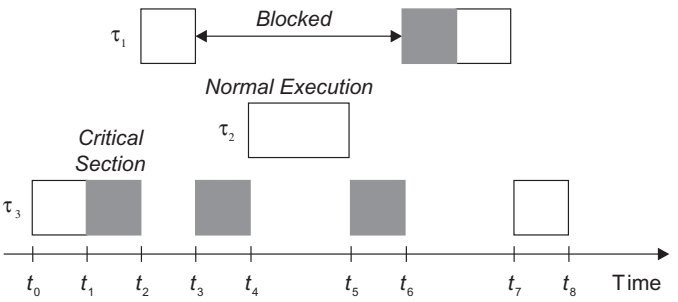


Figure 3.14. A typical priority-inversion scenario.

priority and does not interact with either τ_1 or τ_3. The execution time of τ_2 increases the blocking time of τ_1, as it is no longer dependent solely on the length of the critical section executed by τ_3. Similar unfair conditions could also arise between other intermediate priority tasks—if available—and thereby could lead to an excessive blocking delay. Task τ_1 resumes its execution at time t_6, when τ_3 finally completes its critical section. A priority inversion is said to occur within the time interval $[t_4, t_5]$, during which the highest priority task, τ_1, has been unduly prevented from execution by a medium-priority task τ_2. On the other hand, the acceptable blocking of τ_1 during the periods $[t_3, t_4]$ and $[t_5, t_6]$ by τ_3, which holds the lock, is necessary to maintain the integrity of the shared resources.

The problem of priority inversion in real-time systems has been studied intensively for both fixed-priority and dynamic-priority scheduling. One useful result, the *priority inheritance protocol* (Sha et al., 1990), offers a simple solution to the problem of unbounded priority inversion.

In the priority inheritance protocol, the priorities of tasks are dynamically adjusted so that the priority of any task in a critical region gets the priority of the highest-priority task pending on that same critical region. In particular, when a task, τ_i, blocks one or more higher-priority tasks, it temporarily inherits the highest priority of the blocked tasks. The fundamental principles of the protocol are:

- The highest-priority task relinquishes the CPU whenever it seeks to lock the semaphore guarding a critical section that is already locked by some other task.
- If task τ_1 is blocked by τ_2, and $\tau_1 \succ \tau_2$, task τ_2 inherits the priority of τ_1 as long as it blocks τ_1. When τ_2 exits the critical section that caused the block, it reverts to the priority it had when it entered that section.
- Moreover, priority inheritance is transitive: if τ_3 blocks τ_2 that blocks τ_1 (with $\tau_1 \succ \tau_2 \succ \tau_3$), then τ_3 inherits the priority of τ_1 via τ_2.

Thus, in the three-task example just discussed, the priority of τ_3 would be temporarily raised to that of τ_1 at time t_3, thereby preventing τ_2 from preempting it at time t_4. The resulting schedule incorporating the priority inheritance protocol is shown in Figure 3.15. Here, the priority of τ_3 reverts to its original at time t_5, and τ_2 gets to execute only after τ_1 completes its execution, as desired.

It is important to point out that the priority inheritance protocol does not prevent a deadlock occurring. In fact, priority inheritance can sometimes lead to deadlock or multiple blocking. Nor can it prevent any other problems induced by semaphores. For example, consider the following lock–unlock sequences (with $\tau_1 \succ \tau_2$):

τ_1: Lock S_1; Lock S_2; Unlock S_2; Unlock S_1
τ_2: Lock S_2; Lock S_1; Unlock S_1; Unlock S_2

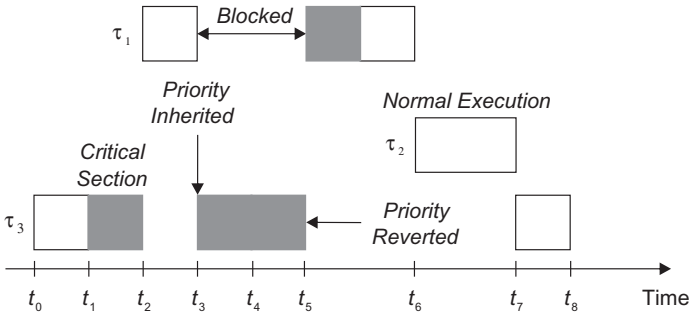


Figure 3.15. Illustration of the priority-inheritance protocol.

Here two tasks, τ_1 and τ_2, use two semaphores for locking critical sections, S_1 and S_2, in a nested fashion, but in reverse order. This problem is similar to the one depicted in Figure 3.13. Although this deadlock does not depend in any sense on the priority inheritance protocol (it is caused by careless use of semaphores), the priority inheritance protocol cannot either prevent this kind of problem. To get around such a problem, it is necessary to use the *priority ceiling protocol* (Chen and Lin, 1990), which imposes a total ordering on the semaphore access. This protocol will be introduced shortly.

A notorious incident of the priority inversion problem occurred in 1997 in NASA's Mars Pathfinder space mission's Sojourner rover vehicle, which was used to explore the surface of Mars. In that case, the MIL-STD-1553B information bus manager was synchronized with mutexes. A mutex is an enhanced binary semaphore that contains priority inheritance and other optional features. Accordingly, a meteorological data-gathering task that was of low priority and low execution rate blocked a communications task that was of higher priority and higher rate. This infrequent scenario caused the entire system to reset. The problem would have been avoided if the optional priority inheritance mechanism provided by the (commercial) real-time operating system had been enabled. But, unfortunately, it had been disabled. Nevertheless, the problem was successfully diagnosed in ground-based testing and remotely corrected by simply enabling the priority inheritance mechanism (Cottet et al., 2002).

The priority ceiling protocol extends to the priority inheritance protocol through chained blocking in such a way that no task can enter a critical section in a way that leads to blocking it. To achieve this, each resource is assigned a priority (the priority ceiling) equal to the priority of the highest priority task that can use it.

The priority ceiling protocol is largely the same as the priority inheritance protocol, except that a task, τ_i, can also be blocked from entering a critical section if there exists any semaphore currently held by some other task whose priority ceiling is greater than or equal to the priority of τ_i. For example, consider the scenario illustrated in Table 3.8. Suppose that τ_2 currently holds a lock on section S_2, and τ_1 is initiated. Task τ_1 will be blocked from entering

TABLE 3.8. Data for the Priority Ceiling Protocol Illustration

Critical Section	Accessed by	Priority Ceiling
S_1	τ_1,τ_2	$P(\tau_1)$
S_2	τ_1,τ_2,τ_3	$P(\tau_1)$

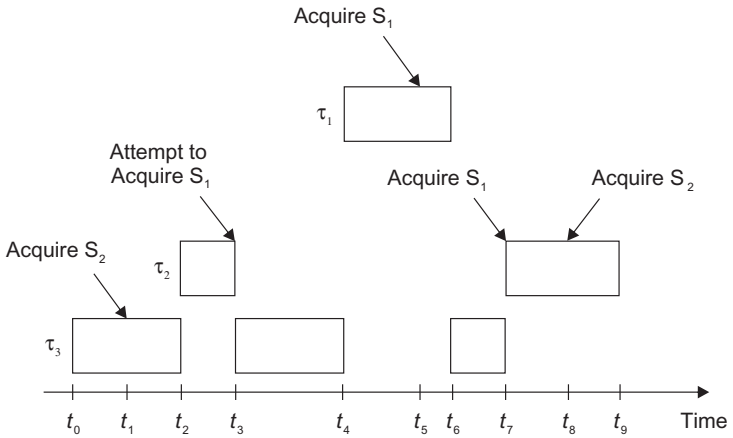


Figure 3.16. Illustration of the priority-ceiling protocol.

section S_1, because its priority is not greater than the priority ceiling of section S_2. A more demanding example will be discussed next to clearly show the advantages of the priority ceiling protocol.

Example: Priority Ceiling Protocol

Consider the three tasks with the following sequence of lock–unlock operations, and having decreasing priorities ($\tau_1 > \tau_2 > \tau_3$):

τ_1: Lock S_1; Unlock S_1

τ_2: Lock S_1; Lock S_2; Unlock S_2; Unlock S_1

τ_3: Lock S_2; Unlock S_2

Following the basic rules of assigning a priority ceiling to semaphores, the priority ceilings of S_1 and S_2 are $P(\tau_1)$ and $P(\tau_2)$, respectively. The following description and Figure 3.16 illustrate the operation of the priority ceiling protocol. Suppose that τ_3 starts executing first, locks the section S_2 at time t_1, and enters the critical section. At time t_2, τ_2 preempts τ_3, starts executing, and attempts to lock section S_1 at time t_3. At this time, τ_2 is suspended, because its priority is not higher than the priority ceiling of section S_2,

currently locked by τ_3. Now task τ_3 temporarily inherits the priority of τ_2 and resumes execution. At time t_4, τ_1 arrives, preempts τ_3, and executes until time t_5, when it needs to lock the section S_1. Note that τ_1 is allowed to lock the section S_1 at time t_5, as its priority is greater than the priority ceiling of all the sections currently being locked (in this case, it is compared with S_2). Task τ_1 completes its execution at t_6, and makes τ_3 execute to completion at t_7. Task τ_2 is then allowed to lock S_1, and subsequently S_2 at t_8, and it finally completes at t_9.

When applying the priority ceiling protocol, a task can be blocked by a lower-priority task only once, and at most the duration of one critical section only.

3.3.7 Timer and Clock Services

In developing real-time software, it is desirable to have easy-to-use timing services available. For example, suppose a diagnostic task checks the "health" of an elevator system periodically. Essentially, the task would execute one round of diagnostics and then wait for a notification to run again, with the task repeating forever. This is usually accomplished by having a programmable timer that is set to create the required time interval.

A system call, `delay`, is commonly available to suspend the executing task until the desired time has elapsed, after which the suspended task is moved to the ready list. The `delay` function has one integer parameter, `ticks`, to specify the length of the delay. In order to generate an appropriate time reference, a timer circuit is configured to interrupt the CPU at a fixed rate, and the internal system time is incremented at each timer interrupt. The interval of time with which the timer is programmed to interrupt defines the unit of time in the system—also called a "tick" or time resolution.

Example: Delay Uncertainty

Suppose we have a delay service available and the tick is initialized to 25 ms. Now, if we want to suspend the diagnostics task for 250 ms (corresponding to 10 ticks), we could simply call `delay(10)`.

But how accurate is this delay? As the *clock signal's phase* and the *calling instant* of the delay function are asynchronous to each other, `delay(10)` will actually generate a varying delay from 225 to 250 ms. Hence, this kind of delay function always has an uncertainty of one tick at maximum. The random variation could naturally be reduced by reducing the tick length. However, a very short tick length may cause significant interrupt overhead to the CPU. An appropriate tick value is a compromise between the delay uncertainty allowed and the interrupt overhead tolerated.

Timers provide a convenient mechanism to control the rate of task execution. In some operating system environments, it is possible to select what kind of timer functionality is used—a one-shot timer or a repeating (periodic) timer. The one-shot timer is armed with an initial expiration time, expires only once, and then is disarmed. A timer becomes a repeating timer with the addition of a repetition period. The timer expires, and then loads the repetition period again, rearming the timer to expire after the repetition period has elapsed, and so on. The delay function discussed above represents a basic one-shot timer.

If very precise timing is needed for some task and it is not practical to shorten the tick to an adequate length, it is best to use a dedicated hardware timer for that particular purpose. Nonetheless, the obvious advantage of the `delay` function over the use of dedicated timers is that a single hardware timer can support multiple timing needs simultaneously.

In addition to these timer functions, it is also desirable to have facilities to set and get the real time and possibly the date. For those purposes, specific `set_time` and `get_time` functions are available in many real-time operating systems.

3.3.8 Application Study: A Real-Time Structure

After presenting a variety of operating system services for application programs, it is instructive to take a look at a real-world example of their usage. In this section, we study an elevator control system from its real-time structure's viewpoint. Hence, our interests are in such issues as tasks and their priorities, the use of hardware interrupts and semaphores, buffering and safe usage of global variables, as well as the use of real-time clock.

The elevator control system considered represents a controller of a single elevator, which operates as a part of a multi-car elevator bank. Therefore, the elevator controller communicates with the so-called group dispatcher that periodically performs optimal hall call allocation for the entire bank of elevators. The number of elevators in a typical bank is up to eight, and the number of floors served is usually no more than 30—true high-rise buildings with more floors are handled with separate low-rise, mid-rise, and high-rise banks of elevators. Such multi-bank elevator installations are used, for instance, in major office buildings and large hotels.

Figure 3.17 illustrates the serial communications connection between the group dispatcher and five individual elevator controllers with 15 floors to service. This bus-type connection is of *master–servant* type: the group dispatcher is the "master" and fully coordinates the communications sessions, while the elevator controllers are "servants" that are allowed to send data solely when requested to do so. The group dispatcher has a serial interface for registering and canceling hall calls (a "hall call" is the event when a person presses the "up" or "down" button to summon an elevator), and it allocates registered calls dynamically to the most suitable elevators depending on their current status (e.g., occupancy, car position, running direction, and registered

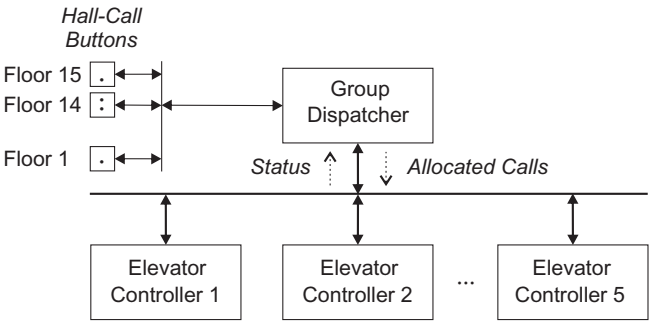


Figure 3.17. Architecture of an elevator bank control system.

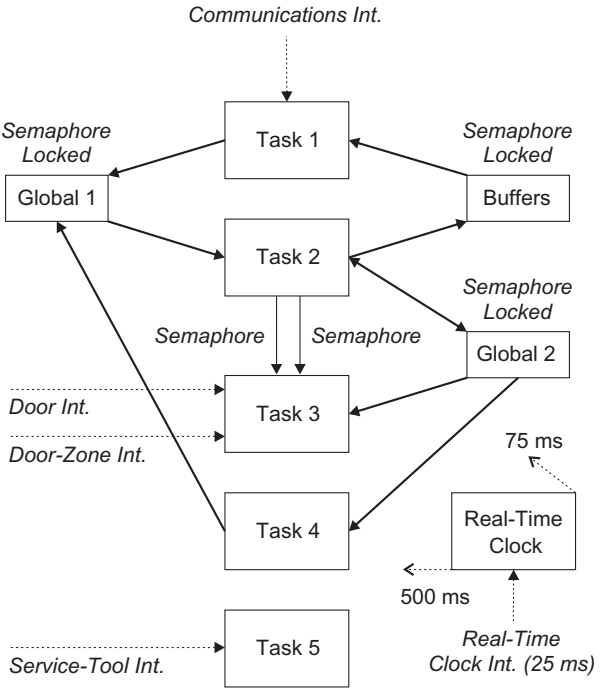


Figure 3.18. Real-time structure of an elevator controller.

car calls). Thus, the group dispatcher needs to periodically collect status information from each individual elevator. Hall calls are allocated using computationally intelligent optimization methods with multiple objectives to minimize the average waiting times of passengers and to avoid excessive waiting periods.

All elevator controllers within a bank are identical, and hence we concentrate on a single (somewhat simplified) controller and its specific real-time structure. A high-level diagram of the real-time framework is depicted in Figure 3.18; it contains five software tasks, Task 1–Task 5, which are next intro-

duced in priority order. The real-time operating system is a foreground/background kernel with preemptive priority scheduling, counting semaphores for synchronization, and a delay system call for creating desired execution periods.

1. This highest-priority task communicates with the group dispatcher through a 19.2 K bit/s serial link, and takes care of the proprietary communications protocol. Its execution period is approximately 500 ms; each communications session lasts no more than 15 ms and is always initiated by the group dispatcher. In addition, Task 1 unpacks the received data and writes them to a global variable area, Global 1 (to be read by Task 2).
2. This task has an execution period of 75 ms, and it performs multiple functions that are related to each other: updates the car position information; registers and cancels car calls; determines the destination floor of next/current run; and packs status data—to be sent to the group dispatcher by Task 1—to a double buffer (Buffers). In addition, Task 2 writes some status variables to another global variable area, Global 2 (to be read by Tasks 3 and 4).
3. The actual floor-to-floor runs are performed by this task (a "floor-to-floor run" is the sequence of operations between a start and a stop). Moreover, this task controls the door opening and closing operations, as well as the car position indicator and direction arrows. There is no regular execution period for Task 3, but it runs when specifically requested to do so—in fact, it is a finite state machine.
4. The lowest-priority foreground task performs various supervision and self-diagnostics operations at the rate of 500 ms. This task also runs a shuttle traffic-type backup system (the "shuttle traffic-type backup system" circulates the elevator according to a predetermined floor schedule) when there is no communications connection to the group dispatcher, or the critical hall call interface in the group dispatcher is broken. This uncomplicated backup solution is needed for providing at least some service to waiting passengers in a failure situation. When the backup system is in use, Task 4 writes commands to the same global variable area, Global 1, where Task 1 unpacks the received data during normal operation.
5. Finally, the background task is executed when the CPU does not have anything more urgent to process. Task 5 runs a versatile real-time debugger that is commanded from a service tool through a 2.4 K bit/s serial link.

The priority order of Tasks 1–5 is based on the following rationale. While the elevator controller is a servant for the group dispatcher, it must be ready to communicate whenever the master wants to begin a communications session.

Therefore, Task 1 has the top priority. Task 2 performs fundamental and time-critical operations related to updating the destination floor, and hence its priority is the highest one of the elevator-specific tasks. The complete run to the destination floor and associated door operations are handled by Task 3 per need basis. Thus, its priority is just below that of Task 2. The next task, Task 4, is performing supervision-type operations that are not directly related to the normal operation of the elevator. Its priority is hence below the priorities of the primary tasks. Finally, the remaining CPU capacity is allocated to the background task, Task 5.

A few hardware interrupts are used with the most time-critical inputs/outputs. Only minimal processing is performed in interrupt handlers (with interrupts disabled), which signal the corresponding tasks by specific semaphores. The more time-consuming interrupt-triggered service is thus performed at tasks (with interrupts enabled). Below is a list of hardware interrupts in priority order:

- Communications interrupts: receiver ready, transmitter ready, and transmitter empty (asynchronous).
- Real-time clock interrupt: tick length 25 ms.
- Door zone interrupt for initiating door opening (asynchronous).
- Door interrupts: closed, some need to reopen, and closing timeout (asynchronous).
- Service tool interrupts: receiver ready and transmitter ready (asynchronous).

In addition to semaphores that are explicitly connected to the hardware interrupts, other semaphores are used for locking global variable areas and buffers, as well as for signaling from one task to another. Those noninterrupt-related semaphores are listed below:

- Semaphore for protecting the swapping of double buffers (Buffers), which Task 2 fills periodically for Task 1.
- Semaphore that Task 2 sets for Task 3 when there is a need to start a run.
- Semaphore that Task 2 sets for Task 3 when there is a need to start deceleration to the next possible floor.
- Two semaphores for protecting the global variable areas Global 1 and Global 2.

Double buffering (see Section 3.3.1) is used between Tasks 1 and 2, because they have very different execution periods (500 ms/75 ms), and Task 1 should always obtain the most recent status from Task 2. Furthermore, this status data is strictly time-correlated. It should be emphasized that although global variables are generally considered as a source of potential problems in real-time programming, they can be used safely *if appropriate locking mechanisms are used with them*. Nonetheless, it is a good practice to minimize the number of

global variables, and allow every variable to be written by a single task only (while the others are just reading).

The delay function discussed in Section 3.3.7 is used in Tasks 2 and 4 for generating the 75 and 500 ms execution periods, respectively. However, it should be remembered that this kind of timing is never precise, but always has a maximum uncertainty of one tick (here 25 ms). Hence, the two execution periods are, in practice, 50–75 ms and 475–500 ms. Such wide tolerances are acceptable for this application.

The real-time structure discussed represents a kind of minimum solution; everything is kept simple and hence the predictability of this firm real-time system is high. Because semaphores are used for protecting critical regions, their consistent use is of utmost significance and requires high discipline among the programming team.

3.4 MEMORY MANAGEMENT ISSUES

An often-neglected topic in real-time operating systems, dynamic memory allocation, is important in terms of both the use of on-demand memory by applications tasks and the memory requirements of the operating system itself. Application tasks use memory explicitly, for example, through requests for heap memory, and implicitly through the maintenance of the runtime memory needed to support sophisticated high-level languages. The operating system has to perform effective memory management in order to keep the tasks isolated, for instance.

Risky allocation of memory is any allocation that can preclude system determinism. Such an allocation can destroy event determinism by overflowing the stack, or it can destroy temporal determinism by causing a deadlock situation. Therefore, it is truly important to avoid risky allocation of memory, while at the same time reducing the overhead incurred by memory management. This overhead is a significant component of the context-switch time and must be minimized.

3.4.1 Stack and Task Control Block Management

In a multitasking system, the context of each task needs to be saved and restored in order to switch tasks successfully. This can be accomplished by using one or more runtime stacks or the task control block model. Runtime stacks are adequate for interrupt-only and foreground/background systems, whereas the TCB model is more appropriate for full-featured operating systems.

If a stack is to be used to handle the runtime saving and restoring of context, two simple routines—`save` and `restore`—are needed. The `save` routine is called by an interrupt handler to save the current context of the system into a stack area; this call should be made immediately after interrupts have been

disabled. Moreover, the `restore` routine should be called just before interrupts are reenabled, and before returning from the interrupt handler (see Section 3.1.4 for an example of context saving and restoring).

On the other hand, if the alternative task control block model (see Section 3.1.5) is used, then a list of TCBs needs to be maintained. This list can be either fixed or dynamic. In the fixed case, *n* task control blocks are allocated during system initialization, with all tasks in the dormant state. As a task is created, its status in the TCB is changed to "ready." Prioritization or time slicing will then move the ready task to the execute state. If some task is to be deleted, its status in the task control block is simply changed to "dormant." In the case of a fixed number of TCBs, no real-time memory management is needed.

In the more flexible dynamic case, task control blocks are inserted to a linked list or some other dynamic data structure as tasks are created. The tasks are in the suspended state upon creation and enter the ready state via an operating system call or some event. The tasks enter the execute state owing to priority or time slicing. When a task is deleted, its TCB is removed from the linked list, and its heap memory allocation is returned to the available or unoccupied status. In this scheme, real-time memory management consists of managing the heap needed to supply the task control blocks.

A runtime stack cannot be used in a round-robin system because of the first-in, first-out (FIFO) nature of the scheduling principle. In this case, a ring buffer can be used conveniently to save context. The context is saved to the tail of the ring buffer and restored from the head. To accomplish these operations, the basic `save` and `restore` functions should be modified accordingly.

The maximum amount of memory space needed for the runtime stack needs to be known *a priori*. In general, the stack size can be determined rather easily if recursion is not used and heap data structures are avoided. If no (conservative) stack memory estimate is available, then a risky memory allocation may occur, and the real-time system may fail to satisfy its behavioral and temporal specifications. In practice, a provision for at least one additional task than anticipated should be allocated to the stack to allow margin for spurious interrupts and time overloading, for example.

3.4.2 Multiple-Stack Arrangement

Often, a single runtime stack is inadequate or cumbersome to manage with several tasks in, say, a foreground/background system. A more flexible multiple-stack scheme uses a single runtime stack for the context and one additional task stack for every task. A typical multiple-stack arrangement is illustrated in Figure 3.19. Using multiple stacks in real-time systems offers clear advantages:

- It permits tasks to interrupt themselves, thus allowing for handling transient overload conditions or for detecting spurious interrupt bursts.
- The real-time software can be written in a programming language that supports reentrancy and recursion. Individual task stacks, which contain

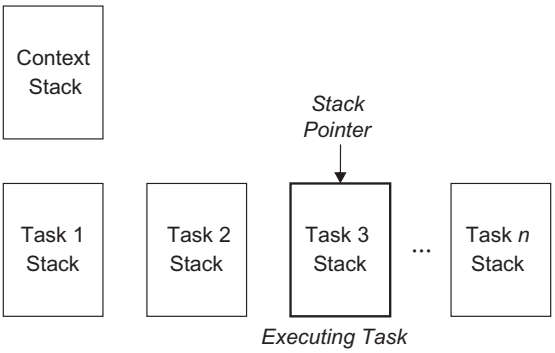


Figure 3.19. Multiple-stack arrangement.

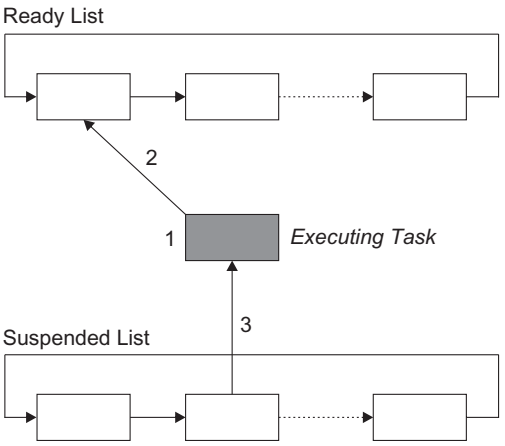


Figure 3.20. Linked-lists management in the task-control-block model.

the appropriate activation records with dynamic links needed to support recursion, can be maintained for each task. A pointer to these stacks needs to be saved in the context or task control block associated with the particular task.

- Only elementary nonreentrant languages, such as assembly language, are recommended with a single-stack model.

3.4.3 Memory Management in the Task Control Block Model

When implementing the TCB model for real-time multitasking, the principal memory management issue is the maintenance of two linked lists for the ready and suspended tasks. This bookkeeping activity is illustrated with an example in Figure 3.20. In step 1, the currently executing task releases some resource

needed by a suspended high-priority task. Therefore, the executing task is inserted to the ready list in step 2, and the suspended high-priority task begins executing in step 3. Hence, by properly managing the linked lists, updating the status word in the TCBs (see Fig. 3.5), and adhering to the appropriate scheduling policy by checking the priority word in the TCBs, round-robin, preemptive priority, or some hybrid scheduling scheme can be induced. Other memory management responsibilities may include the maintenance of certain blocks of memory that are allocated to individual tasks as requested.

An alternative to multiple linked lists involves just a single linked list, in which only the status variable in the TCB is modified rather than moving the entire block to another list. Thus, for instance, when a task is switched from the suspended to ready state or from the ready to executing state, only the single status word needs to be changed. This straightforward approach has the obvious advantage of lighter list management. Nonetheless, it leads to slower traversal times, since the entire list must be traversed during each context switch to search for the next highest priority task that is ready to run.

3.4.4 Swapping, Overlaying, and Paging

Probably the simplest scheme that allows an operating system to allocate memory to two tasks "simultaneously" is swapping. In this case, the operating system itself is always memory resident, and only one task can co-reside in the available memory space not used by the operating system, called the user space. When a second task needs to run, the first task is suspended and then swapped, along with its context, to a secondary storage device, usually a hard disk. The second task, along with its context, is then loaded into the user space and initiated by the task dispatcher. This type of memory management scheme can be used along with round robin or preemptive priority systems, and it is highly desirable to have the execution time of each task be long relative to the lengthy memory-disk-memory swap delay. The varying access time to the secondary storage—typically milliseconds with a hard disk—is the principal contributor to the context-switch overhead and real-time response delays. Hence, it ruins the real-time punctuality of such a real-time system.

Overlaying is a general technique that allows a single program to be larger than the allowable memory. In this case, the program is broken up into dependent code and data sections called overlays, which can fit into the available memory space. Special program code must be included that permits new overlays to be swapped into memory as needed (over the existing overlays), and care must be exercised in the design of such systems. Also, this technique has negative real-time implications, because the overlays must be swapped from slow and nondeterministic secondary storage devices. Nevertheless, flexible overlaying can be used to extend the available address space. Some commercial real-time operating systems support overlaying in conjunction with commonly used programming languages and popular CPUs.

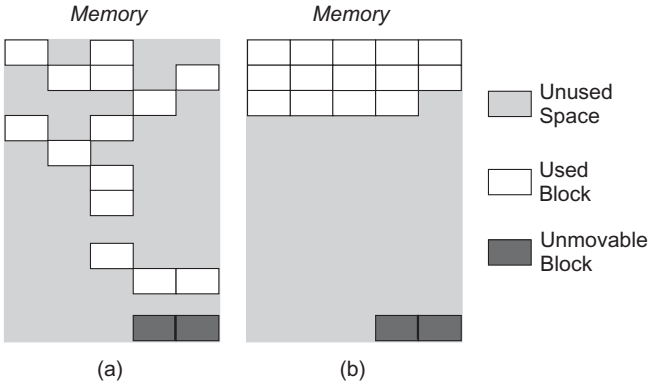


Figure 3.21. Fragmented memory (a) before and (b) after compaction; the unmovable blocks represent the root program of the real-time operating system.

Note that in both swapping and overlaying, one portion of memory is never swapped or overlaid. This critical memory segment contains the swap or overlay manager; in the case of overlaying, any code that is common to all overlays is called the *root*.

A more efficient scheme than simple swapping allows more than one task to be memory resident at any one time by dividing the user space into a number of fixed-size partitions. This scheme is particularly useful in systems where the fixed number of tasks to be executed is known *a priori*. Partition swapping to disk can occur when a task is preempted. Tasks, however, must reside in continuous partitions, and the dynamic allocation and deallocation of memory may be challenging.

In some cases, the main memory can become fragmented with unused but existing partitions, as illustrated in Figure 3.21. In this case, the "checkered" memory space is said to be externally fragmented. This type of fragmentation causes problems when memory requests cannot be satisfied because a *contiguous* block of the size requested does not exist, even though a lot of memory is still available.

Another related problem, internal fragmentation, occurs in fixed-partition schemes when, for example, a task in a real-time Unix environment requires 1 M bytes of memory when only 2 M-byte partitions are available. The amount of wasted memory (or internal fragmentation) can be reduced by creating fixed partitions of several sizes and then allocating the smallest partition greater than the required memory space. Both internal and external fragmentation hamper efficient memory usage and ultimately degrade real-time performance because of the considerable overhead associated with their regular repairing.

This type of dynamic memory allocation uses memory inefficiently, because of the overhead associated with fitting a task to available memory and

performing disk swapping. However, in some implementations, particularly in commercial real-time operating systems, memory can be divided into regions in which each region contains a collection of different-sized, fixed-sized partitions. For example, one region of memory might consist of 10 blocks of size 16 M bytes, while another region might contain 5 blocks of 32 M bytes, and so on. The operating system then tries to satisfy a memory request so that the smallest available partitions are used. This approach tends to reduce internal fragmentation effectively.

In an alternative scheme, memory is allocated in chunks that are not fixed, but rather are determined by the requirements of the task to be loaded into memory. This technique is more appropriate when the number of real-time tasks is unknown or varies. In addition, memory utilization is better for this technique than for fixed-block schemes, because little or no internal fragmentation occurs, as the memory is allocated in the exact amount needed for each task. External fragmentation can still occur because of the dynamic nature of memory allocation and deallocation, and because memory must still be allocated to every single task contiguously.

Compressing fragmented memory or compaction has to be used to mitigate internal fragmentation (see Fig. 3.21). Compaction is a CPU-intensive process and, therefore, is not practical in hard or firm real-time systems during normal operation. If compaction must be performed, it should be done in the background, and it is imperative that interrupts be disabled while memory is being shuffled.

In demand page systems, program segments are permitted to be loaded in noncontiguous memory as they are requested in fixed-size chunks called pages. This scheme helps to eliminate external fragmentation. Program code that is not held in main memory is swapped to some secondary storage, usually a disk. When a memory reference is made to a location within a page not loaded in main memory, a page-fault exception is raised. The interrupt handler for this exception checks for a free page slot in memory. If none is found, a page block must be selected and swapped to disk (if it has been altered), a process called page stealing. Paging, which is provided by most commercial operating systems, is advantageous because it allows nonconsecutive references to pages via a page table. In addition, paging can be used in conjunction with bank-switching hardware to extend the virtual address space. In either case, pointers are used to access the desired page. These pointers may represent memory-mapped locations to map into the desired hardwired memory bank, may be implemented through associative memory, or may be simple offsets into memory, in which case the actual address in main memory needs to be calculated for each memory reference.

Nevertheless, paging can lead to problems, including very high paging activity called thrashing, internal fragmentation, and even a deadlock. It is unlikely that an operating system would use so complex a scheme as paging in an embedded real-time application, where the overhead would be overly high and the associated hardware support is not usually available. On the other

hand, in nonembedded real-time applications, such as the airline booking and reservation system, paging is used routinely.

Several standard methods are used to determine which page should be swapped out of memory to disk, and the same techniques are applicable to cache block replacement as well (Torng, 1998). The most straightforward algorithm is FIFO. Its management overhead is only the recording of the exact loading sequence of the pages. However, the best nonclairvoyant scheme is the least recently used (LRU) algorithm, which states that the least recently used page will be swapped out if a page fault occurs. The management overhead for the LRU scheme rests in recording the access sequence to all pages, which can be quite substantial. Therefore, the benefits of using LRU need to be weighed against the effort in implementing it *vis-à-vis* FIFO.

In addition to thrashing, the main disadvantage of page swapping in real-time systems is the lack of predictable execution times. Therefore, it is often desirable to lock certain parts of a task into main memory in order to reduce the overhead involved in paging and to make the execution times more predictable. Some commercial real-time operating systems provide this feature, called memory locking. These operating systems typically allow code or data segments (or both) for a particular task, as well as the task-stack segment, to be locked into main memory. Any task with one or more locked pages is then prevented from being swapped out to disk. Memory locking decreases execution times for the locked modules and, more importantly, can be used to improve real-time punctuality. At the same time, it makes fewer pages available for the application, encouraging contention.

Garbage is memory that has been allocated but is no longer being used by a task, that is, the task has abandoned it. Garbage can accumulate when processes terminate abnormally without releasing memory resources. In C, for example, if memory is allocated using the `malloc` procedure and the pointer for that memory block is lost, then that block can neither be used nor properly freed. Garbage can also develop in object-oriented systems and as a normal byproduct of nonprocedural languages such as C++. Real-time garbage collection is an important function that must be performed either by the programming language's runtime support (e.g., in Java) or by the operating system if garbage collection is not part of the language. Garbage collection techniques are discussed further in Chapter 4.

3.5 SELECTING REAL-TIME OPERATING SYSTEMS

Selecting a specific real-time operating system (RTOS) for a particular application is a problem for which there is no obvious solution strategy. A related question that is typically asked at the time of systems requirements specification is "Should a commercial RTOS be used or should one be built from scratch?"

3.5.1 Buying versus Building

While the answer to that vital question depends naturally on the entire situation, commercial kernels are frequently chosen because they generally provide robust services, are easy to use, and may even be portable. Commercially available real-time operating systems are wide-ranging in features and performance, and can support many standard devices and communications network protocols. Often, these systems come equipped with helpful development and debugging tools, and they can run on a variety of hardware platforms. In short, commercial RTOSs are best used when they can satisfy the response-time requirements at a competitive cost, and if the real-time system must run on a variety of platforms.

While full-blown RTOSs provide flexibility in scheduling discipline and the number of tasks supported, there are clear drawbacks in their use. For example, they are usually slower than using the plain interrupt-driven framework, because significant overhead is incurred in implementing the task control block model, discussed in Section 3.4.3, which is the typical architecture for commercial real-time operating systems. Furthermore, commercial solutions can include many unneeded features, which are incorporated in order for the RTOS product to have the widest appeal on the market. The execution time and memory costs of these features may be excessive. Finally, manufacturers may be tempted to make somewhat misleading claims, or give best-case performance figures only. The worst-case response times, on the other hand, which would be truly valuable, are generally not available from the RTOS vendors. If they are known, they are often not published because they could place the product in an unfavorable light among its rivals.

For embedded systems, when the per-unit license charge for commercial RTOS products is too high, or when some desired features are unavailable, or when the system overhead is too high, the only alternative is to develop/subcontract the real-time kernel oneself. But this is not a trivial task, and requires substantial development and maintenance effort during the whole lifecycle. Therefore, commercial real-time operating systems should be seriously considered wherever possible.

There are many commercial RTOSs available for real-time systems, but deciding which one is most suitable for a given application is difficult (Anh and Tan, 2009). Many features of embedded real-time operating systems must be considered, including cost, reliability, and speed. However, there are many other characteristics that may be as important or even more important, depending on the application. For example, the RTOS usually resides in some form of ROM and often controls hardware that will not tolerate any faults; hence, the RTOS should also be fault tolerant. Besides, the hardware typically needs to be able to react to different events in the system very rapidly; therefore, the real-time operating system should be able to handle multiple tasks in an efficient manner. Finally, because the hardware platform on which the operating system will reside may have a strictly limited memory space, the

RTOS must use a reasonable amount of memory for its code and data structures.

In fact, there are so many functional and nonfunctional attributes of any commercial RTOS that evaluation and comparison become unavoidably a subjective endeavor. Nonetheless, some rational criteria and metrics should be utilized to support the heuristic decision-making process. Using a standard set of carefully formulated criteria provides a guiding "road sign" toward a successful decision (Laplante, 2005).

3.5.2 Selection Criteria and a Metric for Commercial Real-Time Operating Systems

From business and technical perspectives, the selection of a commercial real-time operating system represents a potential make-or-break decision. It is therefore imperative that a broad and rigorous set of selection criteria be used. The following are desirable characteristics for real-time systems (this discussion is adapted from Laplante [2005]):

- Fault tolerance
- Maintainability
- Predictability
- Survival under peak load
- Timeliness

Hence, the selection criteria should explicitly reflect these desiderata (Buttazzo, 2000). Unfortunately, unless a comprehensive experience base exists using several commercial RTOSs in multiple, identical application domains, there are basically two ways to determine the fitness of an RTOS product for a given application. The first is to rely on third-party reports of success or failure. These abound and are published widely on the Web, and, particularly, in real-time systems conferences. The second is to compare alternatives based on the manufacturer's published information from brochures, technical reports, and websites.

The following discussion presents a *semi-objective* "apples-to-apples" technique for comparing commercial real-time operating systems based on marketing information. This straightforward technique should be used in conjunction with supplemental information from actual experience and third-party reports.

Consider 13 selection criteria, $m_1, \ldots m_{13}$, each having a range $m_i \in [0, 1]$, where unity represents the highest possible satisfaction of the criterion and zero represents complete nonsatisfaction.

1. The *minimum interrupt latency*, m_1, measures the time between the occurrences of hardware interrupt and when the corresponding interrupt service routine begins executing. A low value represents relatively

high interrupt latency, while a high value represents a lower latency. This criterion is important because, if the minimum latency is greater than that required by the particular embedded system, a different operating system must be selected.

2. This criterion, m_2, defines the *maximum number of tasks* the RTOS can simultaneously support. Even though the operating system could support a large number of tasks, this metric is usually constrained by the available memory. This criterion is important for high-end systems that need numerous simultaneous tasks. A relatively high number of tasks supported would result in $m_2 = 1$, while fewer tasks supported would suggest a lower value for m_2.
3. Criterion m_3 specifies the *total memory required* to support the RTOS. It does not include the amount of additional memory required to run the system's application software. Rating $m_3 = 1$ suggests a minimal memory requirement, while $m_3 = 0$ would represent a large memory requirement.
4. The *scheduling mechanism* criterion, m_4, enumerates whether preemptive, round-robin, or some other task-scheduling mechanism is used by the operating system. If several alternative or hybrid mechanisms were supported, then a high value would be assigned to m_4.
5. Criterion m_5 refers to the available methods the operating system has to allow tasks to *communicate/synchronize* with each other. Among possible choices are binary, counting and mutual-exclusion (mutex) semaphores, mailboxes, message queues, ring buffers, shared memory, and so on. Let $m_5 = 1$ if the RTOS provides all desired communication and synchronization mechanisms. A lower value for m_5 implies that fewer mechanisms are available.
6. Criterion m_6 refers to the *after-sale support* an RTOS company puts behind its product. Most vendors offer some sort of free technical support for a short period of time after the sale, with the option of purchasing additional support if required. Some companies even offer on-site consultation. A high value might be assigned to a strong and relevant support program, while $m_6 = 0$ if no support is provided.
7. *Application availability*, m_7, refers to the amount of application software available (either that ships with the RTOS or is available elsewhere) to develop applications to run on the real-time operating system. For example, RTLinux is supported by the GNU' s suite of software, which includes the gcc C compiler and many freely available software debuggers, as well as other supporting software. This may be an important consideration, especially when starting to use an unfamiliar RTOS. Let $m_7 = 1$ if a large amount of software were available, while $m_7 = 0$ would mean that little or nothing was available.
8. Criterion m_8 refers to the different *CPUs supported*, and is important in terms of portability and compatibility with off-the-shelf hardware

and software. This criterion also encompasses the range of peripherals that the operating system can support. A high value for the criterion represents a highly portable and compatible RTOS.

9. Criterion m_9 refers to whether the *source code* of the real-time operating system will be available to the developer, for tweaking or changes. The source code also gives insight to the RTOS architecture, which may be useful for debugging purposes and systems integration. Setting $m_9 = 1$ would suggest open source code or free source code, while a lower value might be assigned in proportion to the purchase price of the source code. Let $m_9 = 0$ if the source code were unavailable.
10. Criterion m_{10} refers to the time it takes for the RTOS kernel to *save the context* when it needs to switch from one task to another. A relatively fast context switch time would result in a higher value for m_{10}.
11. The criterion m_{11} is directly related to the *cost* of the RTOS alone (one-time license fee and possible per-unit royalties). This is critical because for some low-end systems, the RTOS cost may be disproportionately high. In any case, a relatively high cost would be assigned a very low value, while a low cost would merit a higher value for m_{11}.
12. This criterion, m_{12}, rates which *development platforms* are available. In other words, it is a record of the other real-time operating systems that are compatible with the given RTOS. A high value for m_{12} would represent wide compatibility, while a low m_{12} would indicate a single platform.
13. Finally, the criterion m_{13} is based on a record of what *communications networks and network protocols* are supported by the given RTOS. This would be useful to know because it rates what communications methods the software running on this RTOS would be able to use to communicate with other computers. A high value for the criterion represents a relatively large variety of networks supported.

Recognizing that the importance of individual criteria will differ greatly depending on the application, a weighting factor, $w_i \in [0, 1]$, will be used for each criterion m_i, $i \in \{1, 2, \ldots 13\}$, where unity is assigned if the criterion has highest importance, and zero if the criterion is unimportant in the particular application. Then an average fitness metric, $\bar{M} \in [0, 1]$, for supporting the decision-making process, is formed as:

$$\bar{M} = \frac{1}{13} \sum_{i=1}^{13} w_i m_i \tag{3.8}$$

Obviously, a high value of $\bar{M}$ means that the RTOS is well suited to the application, and a low value means that the RTOS is poorly suited for the application. While selection of the values for w_i and m_i will be somewhat

subjective for any given RTOS and any given application, the availability of this clear-cut metric provides a handle for *semi-objective* comparison.

3.5.3 Case Study: Selecting a Commercial Real-Time Operating System

A representative commercial RTOS is first examined based on the 13 criteria just introduced. Although the data are mostly real, the manufacturer name is omitted, as our intention is not to imply a recommendation of any product—this case study is for illustration purposes only. For all the compared RTOSs, equal and fair operating conditions are assumed.

In the cases where a quantitative criterion value can be assigned, this is done right away. Where the criteria are "CPU dependent" or indeterminate, absent a real application, assignment of a numerical rating is postponed, and a value of "*" is given. This "unknown" value is approximated later at the time of application analysis. Note too that the values between columns of the comparison table need to be consistent. For example, if a 6-μs interrupt latency yields $m_1 = 1$ for RTOS X, the same 6-μs latency should yield $m_1 = 1$ for RTOS Y as well.

Consider a commercial RTOS A. Table 3.9 summarizes the criteria and ratings, which were based on the following rationale. The product literature indicated that the minimum interrupt latency is CPU dependent, therefore $m_1 = *$ is assigned here. Context switch time and compatibility with other RTOSs are not given, and so $m_{10} = m_{12} = *$ are indicated. In all these cases, the "*" will be later resolved as 0.5 for the purposes of evaluating the metric of Equation 3.8. RTOS A supports 32 task-priority levels, but it is not known if there is a limit on the total number of tasks, so a value of $m_2 = 0.5$ is assigned. RTOS A itself requires 60 K bytes of memory, which is somewhat more than some of the alternatives, so a value of $m_3 = 0.7$ is assigned. The operating

TABLE 3.9. Summary Data for RTOS A

Criterion	Description	Rating	Comment
m_1	Minimum interrupt latency	*	CPU dependent
m_2	Maximum number of tasks	0.5	32 task-priority levels
m_3	Total memory required	0.7	ROM: 60 K bytes
m_4	Scheduling mechanism	0.25	Preemptive only
m_5	Communicate/synchronize	0.5	Direct message passing
m_6	After-sale support	0.5	Paid phone support
m_7	Application availability	1	Various
m_8	CPUs supported	0.8	Various
m_9	Source code	1	Available
m_{10}	Save the context	*	Unknown
m_{11}	Cost	0.5	$2500 + royalty fee
m_{12}	Development platforms	*	Unknown
m_{13}	Networks and protocols	1	Various

system provides only one form of scheduling, preemptive priority, so a low value, $m_4 = 0.25$, is assigned here. Intertask communication and synchronization is available only through direct message passing, so a relatively low $m_5 = 0.5$ is assigned. RTOS A is available for various hardware platforms, but fewer than its competitors, hence $m_8 = 0.8$.

The company behind RTOS A provides paid phone support, which is not as generous as other companies, so a value of $m_6 = 0.5$ is assigned. The initial license fee is moderate, and there is some royalty per each produced unit, so $m_{11} = 0.5$ was assigned. Finally, there is a wide range of software support for the product, including the available source code and communications network protocols, so values of unity are given for these three criteria ($m_7 = m_9 = m_{13} = 1$).

Consider the following application and a set of five real-time operating systems, including RTOS A just described and RTOS B-E, whose criteria were determined in a similar manner; see Laplante (2005) for more details.

The hard real-time software controlling an inertial measurement system of a fighter aircraft requires substantial input/output processing, which inherently causes a high rate of hardware interrupts. This is an extremely reactive and mission-critical system that requires fast context switching ($w_{10} = 1$), minimal interrupt latency ($w_1 = 1$), compact hardware implementation ($w_3 = 1$), versatile synchronization ($w_5 = 1$), and a well-supported system ($w_6 = w_7 = 1$). Hardware compatibility is not critical, because there is little need to port the system, and the number of tasks supported is relatively low, therefore, $w_2 = w_8 = 0.1$. Cost of the RTOS is not so important in this application, hence $w_{11} = 0.4$. The other criteria are set to 0.5, because they are only moderately important, $w_4 = w_9 = w_{12} = w_{13} = 0.5$.

The weights and corresponding ratings assigned are summarized in Table 3.10, and the metric $\bar{M}$ suggests that RTOS D is the best match for our inertial measurement system with $\bar{M} = 0.527$, while the maximum possible rating is here $\bar{M}_{\max} = 0.662$. Nevertheless, the metric of RTOS E ($\bar{M} = 0.489$) is only 7.2% lower than that of RTOS D, and hence it represents the second-best match. The other candidates have their metrics 20.4–23.0% below the best one. Furthermore, it should be noted that all the real-time operating systems considered have relatively high standard deviation (0.269–0.384) of the weighted selection metrics. This is partly explained by the comparable standard deviation of the weights w_i (0.352).

In practice, both RTOS D and RTOS E would next be taken a closer look at before approaching the final decision. Figure 3.22 illustrates the weighted ratings for RTOS D and RTOS E. We can observe that RTOS D has a higher weighted rating than RTOS E in four of the six imperative criteria weighted with unity. Moreover, the rating of first criterion (minimum interrupt latency) is 0.5 for RTOS D, because it is defined vaguely as "CPU dependent." On the other hand, the minimum interrupt latency of RTOS E is provided explicitly as 6 μs ($\Rightarrow m_1 = 1$). These and other specific details are available from Laplante (2005). Before making the actual decision, an exact value corresponding to this critical criterion should definitely be found out. The same

TABLE 3.10. Decision Table for the Inertial Measurement System

Criterion	Description	Weight w_i	A	B	C	D	E
m_1	Minimum interrupt latency	1	0.5	0.8	1	0.5	1
m_2	Maximum number of tasks	0.1	0.5	0.5	0.5	1	1
m_3	Total memory required	1	0.7	0.2	0.5	1	0.9
m_4	Scheduling mechanism	0.5	0.25	0.5	0.25	1	0.25
m_5	Communicate/synchronize	1	0.5	1	0.5	1	1
m_6	After-sale support	1	0.5	0.5	1	0.8	1
m_7	Application availability	1	1	0.75	1	1	0.5
m_8	CPUs supported	0.1	0.8	0.5	0.2	1	0.2
m_9	Source code	0.5	1	1	0	0.4	1
m_{10}	Save the context	1	0.5	0.5	0.5	1	0.5
m_{11}	Cost	0.4	0.5	0.5	0.1	0.1	0.7
m_{12}	Development platforms	0.5	0.5	0.5	0.5	0.5	0.5
m_{13}	Networks and protocols	0.5	1	1	1	1	0.6
$\bar{M}$			0.405	0.417	0.419	0.527	0.489
STD_{weighted}			0.269	0.295	0.384	0.382	0.364

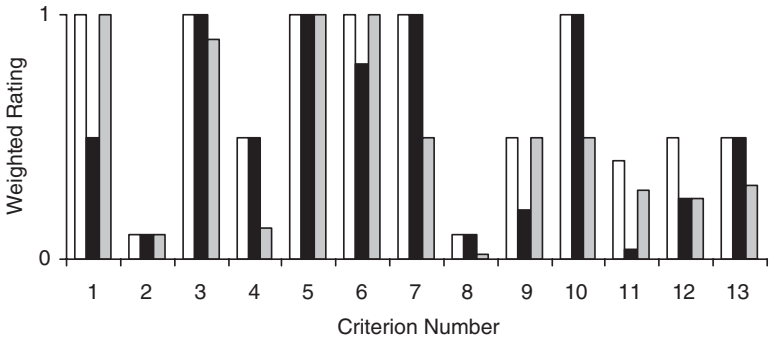


Figure 3.22. Individual criterion weights (white) and weighted criterion ratings for RTOS D (black) and RTOS E (gray).

applies to the "save the context" criterion for RTOS E. After obtaining those additional data, and considering supplemental information from concrete experience and possible third-party reports, we are ready to begin close negotiations with the company behind RTOS D or RTOS E. Finally, all the factors that guided us to the semi-objective decision should be thoroughly documented for future needs.

3.5.4 Supplementary Criteria for Multi-Core and Energy-Aware Support

For many years, the fundamental selection criteria, m_i, $i \in \{1, 2, \ldots 13\}$, just discussed have remained fairly static; no major advancement has occurred in

evaluating the suitability of an RTOS for a specific application by utilizing the standard metric of Equation 3.8. Only quantitative expansion has taken place, particularly, within the application availability (criterion m_7), as well as the availability of communications networks and network protocols (criterion m_{13}). Recently, however, two new criteria have become pertinent when selecting a real-time operating system for multi-core environments (Sindhwani and Srikanthan, 2005) or energy-aware embedded systems (Saran et al., 2005).

Multi-core processors (see Section 2.3.3) are used increasingly in both nonembedded and embedded real-time applications, because they can offer high instruction throughput and true concurrency for real-time multitasking. To take full advantage of the available parallel-processing capability, a special RTOS is needed that is designed and configured for multi-core architectures. Just a few commercial RTOSs exist with explicit multi-core support, such as a hybrid multitasking scheme that provides both *intra-core* and *inter-core* multitasking with a double-level operating system hierarchy. This intra-core multitasking resembles the behavior of a conventional multitasking system in a uniprocessor environment, while the higher-level inter-core multitasking provides true concurrency that is only possible with two or more cores. In some cases, even a single task could be split across more than one core by an online scheduler (Lakshmanan et al., 2009). Therefore, to support the latest needs, we introduce another criterion, *multi-core support* (m_{14}), which is used when the desired processing environment is not a traditional uniprocessor. Here, $m_{14} = 0$ corresponds to an RTOS with no multi-core support, and $m_{14} = 1$ if extensive multi-core features and associated load-balancing utility software are available.

Energy-aware operating systems are used increasingly in wireless sensor network applications, including environmental monitoring, high-tech bridges, military surveillance, smart buildings, and traffic monitoring (Eswaran et al., 2005). Those tiny, spatially distributed, and highly cooperative real-time systems are usually battery operated, and there is a primary requirement to maximize the battery lifetime. In its simplest form, "energy-aware" means just a capability of the RTOS to put the idling CPU (no task is scheduled to execute within a specified time window) to a sleep mode, where the power consumption can be reduced to $<<1\%$ of the active mode consumption. A hardware interrupt from a real-time clock or communications controller awakes the CPU back to the active mode in a few microseconds. More sophisticated energy-aware operating systems may provide adaptable quality-of-service (QoS) for communications performance; higher data loss and transmission error probabilities are traded off against lower energy consumption, and vice versa (Raghunathan et al., 2001). The QoS is regulated by adjusting the supply voltage and clock frequency of the CPU appropriately. In that way, significant energy savings can be obtained with acceptable QoS levels in ad hoc networks. For supporting these emerging needs, we introduce another criterion, *energy-aware support* (m_{15}). It is used when the application has specific energy awareness requirements. If the RTOS offers no energy-aware support, m_{15} is set to

zero. A high value for m_{15} implies that several energy-saving options are provided. Some of the light-overhead energy-aware operating systems are based on an event-driven programming model that differs from the traditional scheduling approaches (Rossetto and Rodriguez, 2006).

Finally, after introducing two supplementary selection criteria, m_{14} and m_{15}, we have to modify the metric of Equation 3.8 accordingly. In the future, when the processor and applications technologies evolve, it is likely that additional criteria will be introduced every now and then. Moreover, certain real-time applications have already unique application-specific criteria to consider in addition to the standard ones discussed above.

3.6 SUMMARY

In this chapter, we gave a thorough presentation on central issues related to real-time operating systems. The practical discussion has both extensive breadth and considerable depth, and hence it forms a solid basis for understanding, designing, and analyzing multitasking systems having shared resources. Our presentation—contrary to most other textbooks on real-time systems—also covers a heterogeneous collection of pseudokernels, because the primary goal of a software designer is *to create a competitive real-time system*, not merely to use an operating system (that should be seen as a tool).

But what are the general conclusions and suggestions that follow from that discussion? Real-time software engineering requires understanding the purpose to be achieved, the available resources, and the manner in which they may be allocated collectively to achieve the ultimate objective—a predictable and maintainable real-time system, which fulfills all response-time requirements with adequate punctuality. To contribute to the burdensome process of achieving the ultimate objective, we next compose a set of pragmatic rules that are derived from the contents of Sections 3.1–3.5:

- *From Pseudokernels to Operating Systems.* There is a variety of "operating-system" architectures available but consider first the simple ones, since they are usually more predictable and their computational overhead is lower; if you decide to use other than pseudokernels in embedded systems, minimize the number of tasks, because task switching and synchronization are time-consuming operations.
- *Theoretical Foundations of Scheduling.* Internalize the general principles of fixed- and dynamic-priority scheduling, as they are helpful when you prioritize tasks or interrupts in a practical real-time system.
- *System Services for Application Programs.* Never use global buffers without a safe locking mechanism; always pay special attention to deadlock avoidance when you share critical resources between multiple tasks; beware of priority inversion; although system services make the program-

ming work easier, remember that most system calls are time-consuming because they end up to scheduling.

- *Memory Management Issues.* If your operating system uses multiple stacks, reserve an adequate (worst-case) space for them—sporadic stack overflows are disastrous and difficult to debug; computing platforms with virtual memory are for soft and firm real-time systems only, since page swapping between the main memory and secondary storage is overly time-consuming; moreover, the main memory and secondary storage need regular garbage collection and compaction, which may disturb time-critical tasks.
- *Selecting Real-Time Operating Systems.* If you choose above "kernel" in the operating system taxonomy (see Fig. 3.2), a commercial solution is usually the best choice; devote substantial expertise and effort to the selection process, because you usually have to live with the selected RTOS for several years; collect first the relevant technical information to obtain some objectiveness to your decision-making strategy; admit that the final decision can only be semi-objective (at its best), because you have a multi-objective optimization problem with obvious uncertainties in the composite cost function; do not be afraid of subjective criteria or "feelings," as many complicated problems, such as selecting one's career or employer, are often successfully solved with largely subjective arguments; anyhow, the RTOS selected should only be "good enough."

During recent years, the requirements set for real-time operating systems have somewhat evolved due to developments in processor and applications technologies. Multi-core processors and energy-aware sensor networks have set totally new challenges for RTOS developers. Besides, there is still a lot of space for innovations and associated research in those and other emerging segments. Thus, the field of real-time operating systems is vital from both engineering and research viewpoints.

3.7 EXERCISES

3.1. Explain what is meant by task concurrency in a uniprocessor environment.

3.2. What are the desirable features that an operating system should have to provide for predictability in time-critical applications?

3.3. For some sample real-time systems described in Chapter 1, discuss which operating system architecture is most appropriate.

(a) Inertial measurement system.
(b) Nuclear monitoring system.
(c) Airline reservations system.

(d) Pasta sauce bottling system.

(e) Traffic light controller.

Make whatever assumptions you like, but document them and justify your answers.

3.4. What determines the priority of a task in an embedded application with fixed priorities?

3.5. Construct a cyclic code structure with four procedures, A, B, C, and D. Procedure A runs twice as frequently as B and C, and procedure A runs four times as frequently as D.

3.6. What is the principal difference between a background task and a foreground task?

3.7. Exceptions can be used conveniently as a framework for error recovery. Define the following terms:

(a) A synchronous exception.

(b) An asynchronous exception.

(c) An application-detected error.

(d) An environment-detected error.

3.8. Should an interrupt service routine be allowed to be interruptible? If it is, what are the consequences?

3.9. Write some simple assembly language routine that is not reentrant. How could you make it reentrant?

3.10. Write `save` and `restore` routines (see Section 3.1.4) in assembly code, assuming push all (`PUSHALL`) and pop all (`POPALL`) instructions are available for saving and restoring all work registers.

3.11. Discuss the difference between fixed and dynamic, online and offline, optimal and heuristic scheduling algorithms.

3.12. Show mathematically that the upper limit for CPU utilization with the rate-monotonic approach, $\lim_{n\to\infty} n\left(2^{1/n}-1\right)$, is exactly $\ln 2$ as stated in Equation 3.7.

3.13. Discuss the advantages of earliest deadline first scheduling over rate-monotonic scheduling and vice versa.

3.14. Explain what is meant by context-switching overhead, and how to account for it in the rate-monotonic and earliest deadline first schedulability analysis.

3.15. Show with an example that the earliest deadline first algorithm is no longer an optimal scheduling algorithm if preemption is not allowed.

3.16. Give two different explanations why the following three periodic tasks are schedulable by the rate-monotonic algorithm: $\tau_1 \equiv \{0.8, 2\}$, $\tau_2 \equiv \{1.4, 4\}$, and $\tau_3 \equiv \{2, 8\}$. Here, the notation $\tau_i \equiv \{e_i, p_i\}$ gives the execution time, e_i, and period, p_i, of task τ_i.

3.17. Verify the schedulability under rate-monotonic algorithm and construct the schedule of the following task set: $\tau_1 \equiv \{3, 7\}$, $\tau_2 \equiv \{5, 16\}$, and $\tau_3 \equiv \{3, 15\}$. Here, the notation $\tau_i \equiv \{e_i, p_i\}$ gives the execution time, e_i, and period, p_i, of task τ_i.

3.18. Verify the schedulability under earliest deadline first algorithm and construct the schedule of the following task set: $\tau_1 \equiv \{1, 5, 4\}$, $\tau_2 \equiv \{2, 8, 6\}$, and $\tau_3 \equiv \{1, 4, 3\}$. Here, the notation $\tau_i \equiv \{e_i, p_i, D_i\}$ gives the execution time, e_i, period, p_i, and relative deadline, D_i, of task τ_i.

3.19. What effect does the length of a ring buffer have on its performance? How would you determine the suitable length for a specific case?

3.20. Write `save` and `restore` routines (see Section 3.1.4) in assembly code so that they save and restore the context to/from the head and tail of a ring buffer, respectively—instead of using a stack.

3.21. Assume a preemptive priority system with two tasks, τ_1 and τ_2 (with $\tau_1 \succ \tau_2$), which share a single critical resource. Show with an appropriate execution scenario that a simple software flag (a global variable) at the level of application tasks is not adequate for providing safe sharing of the critical resource.

3.22. An operating system provides 256 fixed priorities to tasks in the system, but only 32 priority levels for their messages exchanged through message queues. Suppose that each posting task chooses the priority of its messages by mapping the 256 priorities to 32 message-priority levels. Discuss some potential problems associated with this uniform mapping scheme. What kind of approach would you take?

3.23. Write two pseudocode routines to access (*read* from and *write* to) a 20-item ring buffer. The routines should use binary semaphores to allow more than one user to access the buffer safely.

3.24. Give an example from the real world in which a deadlock sometimes occurs in practice. How is that situation usually solved?

3.25. Show how priority inheritance can cause a deadlock. Consider three tasks (with $\tau_1 \succ \tau_2 \succ \tau_3$) and appropriate lock–unlock sequences.

3.26. What knowledge is needed to determine the size of runtime stack in a multiple-interrupt system? What safety precautions are necessary?

3.27. Write a pseudocode procedure compacting 64 M bytes of memory that is divided into 1 M-byte pages. Use a pointer scheme.

3.28. Write a pseudocode procedure that allocates pages of memory on request. Assume that 100 pages of size 1 M byte, 2 M bytes, and 4 M bytes are available. The procedure should take the size of the page requested as an argument, and return a pointer to the allocated page. The smallest available page should be used, but if the smallest size is unavailable, the next smallest should be used.

3.29. By performing a Web search, obtain as much relevant data as you can for at least two commercial real-time operating systems. Summarize your findings and compare those operating systems to each other. What are their main differences?

3.30. Identify some of the limitations of existing commercial real-time kernels for the development of mission- and safety-critical applications. Perform a Web search to collect the necessary information.

REFERENCES

T. N. B. Anh and S.-L. Tan, "Real-time operating systems for small microcontrollers," *IEEE Micro*, 29(5), pp. 30–45, 2009.

G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Norwell, MA: Kluwer Academic Publishers, 2000.

M.-I. Chen and K.-J. Lin, "Dynamic priority ceilings: A concurrency control protocol for real-time systems," *Real-Time Systems*, 2(4), pp. 325–346, 1990.

F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in Real-Time Systems*. Chichester, UK: John Wiley & Sons, 2002.

A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An energy-aware resource-centric RTOS for sensor networks," *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, Miami, FL, 2005, 10 pp.

J. Havender, "Avoiding deadlock in multitasking systems," *IBM Systems Journal*, 7(2), pp. 74–84, 1968.

K. Lakshmanan, R. Rajkumar, and J. Lohoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, Dublin, Ireland, 2009, pp. 239–248.

P. A. Laplante, "Criteria and a metric for selecting commercial real-time operating systems," *Journal of Computers and Applications*, 27(2), pp. 82–96, 2005.

C. L. Liu and J. W. Layland, "Scheduling algorithms for multi-programming in a hard real-time environment," *Journal of the ACM*, 20(1), pp. 46–61, 1973.

V. Raghunathan, P. Spanos, and M. B. Srivastava, "Adaptive power-fidelity in energy-aware wireless embedded systems," *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, London, UK, 2001, pp. 106–115.

S. Rossetto and N. Rodriguez, "A cooperative multitasking model for networked sensors," *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems Workshops*, Lisbon, Portugal, 2006, 6 pp.

L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, 39(9), pp. 1175–1185, 1990.

A. C. Shaw, *Real-Time Systems and Software*. New York: John Wiley & Sons, 2001.

D. E. Simon, *An Embedded Software Primer*. Boston: Addison-Wesley, 1999.

M. Sindhwani and T. Srikanthan, "Framework for automated application-specific optimization of embedded real-time operating systems," *Proceedings of the 5th International Conference on Information, Communications and Signal Processing*, Bangkok, Thailand, 2005, pp. 1416–1420.

J. A. Stankovic and R. Rajkumar, "Real-time operating systems," *Real-Time Systems*, 28(2/3), pp. 237–253, 2004.

K.-C. Tai, "Definitions and detection of deadlock, livelock, and starvation in concurrent programs," *Proceedings of the International Conference on Parallel Processing*, Raleigh, NC, 1994, pp. 69–72.

E. Torng, "A unified analysis of paging and caching," *Algorithmica*, 20(1), pp. 175–200, 1998.

4

PROGRAMMING LANGUAGES FOR REAL-TIME SYSTEMS

Programming languages inspired vigorous debates among programmers in the *early* years of embedded real-time systems: should one continue to use the assembly language, go ahead with one of the PL/I derivatives (Intel's PL/M, Motorola's MPL, or Zilog's PL/Z), or even consider the emerging C language for a software project? *Today*, there is very little such debate between practitioners developing embedded software. If we consider the global community of professional real-time programmers, we could even condense the programming-language selection dilemma to two principal alternatives: C++ or C—increasingly in this order of consideration. Of course, there are exceptions to our somewhat naive simplification: Ada has a place in new and legacy projects for the U.S. Department of Defense (DoD), and Java is used widely in applications to be run on multiple platforms. Nevertheless, we choose to simplify the decision around C++ and C: if you have a large software project where the productivity of programmers and the long-term maintainability of code produced are of primary importance, then C++ is a good choice, while in smaller projects with tight response-time specifications and/or considerable material-cost pressure to make the hardware platform overly reduced, C is the appropriate language. And specifically, we are talking about new product generations only, that is, software to be developed largely from scratch. The

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

situation is obviously different if we want to reuse some software from earlier projects or merely extend an existing product.

Writing software is considered, increasingly, to be commoditized work that could be subcontracted to a software consulting company if rigorous requirements-engineering processes have been followed. This situation is especially true with large-scale projects, like building automation systems or cell-phone exchanges. On the other hand, in highly time-critical applications and in the core sections of innovative products, the software development effort may take place very close to the corresponding algorithms development to ensure that the required sampling rates are achieved in embedded systems within the "dangerous" CPU utilization zone (see Table 1.3), or critical intellectual property needs to be protected within the organization.

In this chapter, we provide an evaluative discussion on programming languages for real-time systems. Recognizing that each organization and application is unique, and the need to consider multiple language choices is often necessary, the discussion goes beyond the simplified C++/C viewpoint that was expressed in the beginning of the chapter. Section 4.1 introduces the general topic of writing real-time software with a brief overview on coding standards. The limited but continuing use of assembly language is contemplated in Section 4.2, while Sections 4.3 and 4.4 provide pragmatic discussions on the advantages and disadvantages of procedural and object-oriented languages, respectively. Section 4.5 contains a focused overview of mainstream programming languages: Ada, C, C++, C#, and Java. Automatic code generation has been a dream of software engineers for a long time, but there are no general-purpose techniques for creating real-time software "automatically." An introduction to automatic code generation and its challenges is given in Section 4.6. Section 4.7 presents some standard code-optimization strategies used in compilers. These optimization strategies are particularly valuable to note when writing time-critical code with a procedural language, or when debugging an embedded system at the level of assembly-language instructions. An insightful summary of the preceding sections is provided next in Section 4.8. Finally, a carefully composed set of exercises is available in Section 4.9.

4.1 CODING OF REAL-TIME SOFTWARE

Misuse of the underlying programming language can be the single greatest source of performance deterioration and missed deadlines in real-time systems. Moreover, when using object-oriented languages in real-time systems, such performance problems can be more difficult to analyze and control. Nonetheless, object-oriented languages are steadily displacing procedural languages as the language of choice in real-time embedded systems development. Figure 4.1 depicts the *mainstream* use of programming languages in embedded real-time applications from the 1970s to the present decade.

Some parts of this section have been adapted from Laplante (2003).

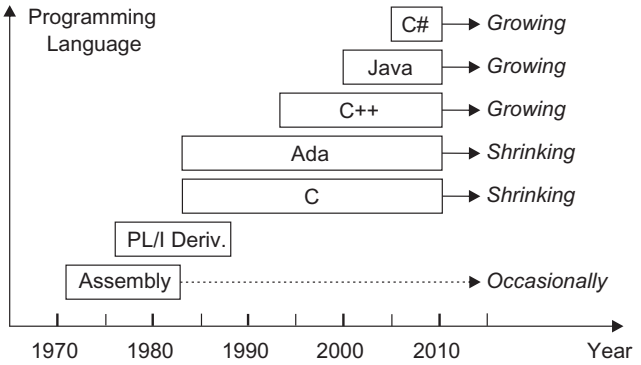


Figure 4.1. Mainstream usage of real-time programming languages over the years (the year limits are approximate).

4.1.1 Fitness of a Programming Language for Real-Time Applications

A programming language represents the nexus of design and structure. Hence, because the actual "build" of software depends on tools to compile, generate binary code, link, and create binary objects, "coding" should take proportionally less time than the requirements engineering and design efforts. Nevertheless, "coding" (synonymous to "programming" and "writing software") traditionally has been more craft-like than production based, and as with any craft, the best practitioners are known for the quality of their tools and the associated skills to use them effectively.

The main tool in the code generation process is the language compiler. Real-time systems are currently being built with a variety of programming languages (Burns and Wellings, 2009), including various dialects of C, C++, C#, Java, Ada, assembly language, and even Fortran or Visual Basic. From this heterogeneous list, C++, C#, and Java are all object-oriented, while the others are procedural. It should be pointed out, however, that C++ can be abused in such a way that all object-oriented advantages are lost (e.g., by embedding an old C program into one "God" class). Furthermore, Ada 95 has elements of *both* object-oriented and procedural languages, and can hence be used either way, depending on the skills and preferences of the programmer as well as local project policies.

A relevant question that is often asked is: "What is the fitness of a programming language for real-time applications and what metrics could be used to measure or at least estimate such fitness?" To address this multidimensional question consider, for instance, the five criteria of Cardelli (Cardelli, 1996):

C1. Economy of Execution. How fast does a program run?

C2. Economy of Compilation. How long does it take to go from multiple source files to an executable file?

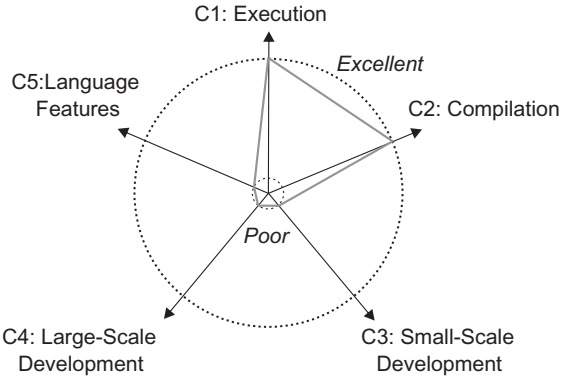


Figure 4.2. Pentacle diagram for illustrating the Cardelli criteria of various economies (the gray pentacle corresponds to assembly language).

C3. Economy of Small-Scale Development. How hard must an individual programmer work?

C4. Economy of Large-Scale Development. How hard must a team of programmers work?

C5. Economy of Language Features. How hard is it to learn or use a programming language?

Every programming language offers undoubtedly its own strengths and weaknesses with respect to real-time systems, and these qualitative criteria, C1–C5, can be used to calibrate the features of a particular language for oranges-to-oranges comparison within a given application. The Cardelli criteria can be illustrated with a pentacle diagram (Sick and Ovaska, 2007) shown in Figure 4.2. Such diagrams provide simple means for visual comparison of candidate programming languages.

In this chapter, we do not intend an exhaustive programming-language survey; instead, our focus is on those language features that could be used to minimize the final code execution time and that lend themselves to performance prediction. The compile-time prediction of execution performance directly supports a schedulability analysis. In the design of special real-time programming languages, the emphasis is on eliminating those constructs that render the language nonanalyzable, for example, unbounded recursion and unbounded `while` loops. Most so-called "real-time languages" strive to eliminate all of these. On the other hand, when mainstream languages are used for real-time programming, certain problematic code structures may simply be prohibited through coding standards.

4.1.2 Coding Standards for Real-Time Software

Coding standards (Li and Prasad, 2005) are different from language standards. A language standard, for example, C++ ANSI/ISO/IEC 14882:2003, embodies

the syntactic rules of the C++ programming language. A source program violating any of those rules will be rejected by the compiler. A coding standard, on the other hand, is a set of stylistic conventions or "best practices." Violating these conventions will not lead to compiler rejection. In another sense, compliance with language standards is mandatory, while compliance with coding standards is, at least in principle, voluntary.

Adhering to language standards fosters portability across different compilers, and, hence, hardware environments. Complying with coding standards, on the other hand, will not foster portability, but rather in many cases, readability, maintainability, and reusability. Some practitioners even contend that the use of strict coding standards can increase software reliability. Coding standards may also be used to foster improved performance by encouraging or mandating the use of certain language constructs that are known to generate code that is more efficient. Many agile methodologies, for instance, eXtreme Programming (Hedin et al., 2003), embrace special coding standards.

Coding standards typically involve standardizing some or all of the following elements of programming language use:

- Header format
- Frequency, length, and style of comments
- Naming of classes, data, files, methods, procedures, variables, and so forth
- Formatting of program source code, including use of white space and indentation
- Size limitations on code units, including maximum and minimum number of code lines, and number of methods used
- Rules about the choice of language construct to be used; for example, when to use `case` statements instead of nested `if-then-else` statements

While it is unclear if conforming to these rules fosters significant improvement in reliability, clearly close adherence can make programs easier to read and understand, and hence likely more reusable and maintainable (Hatton, 1995).

There exist many different standards for coding that are either language independent or language specific. Coding standards can be companywide, teamwide, user-group specific (e.g., the GNU software group has standards for C and C++), or customers can require conformance to a certain standard of their own. Furthermore, other standards have come into the public domain. One example is the Hungarian notation standard (Petzold, 1999), named in honor of Charles Simonyi, who is credited with first promulgating its use. Hungarian notation is a public-domain standard intended to be used with object-oriented languages, particularly C++. The standard uses a purposeful naming scheme to embed type information about the objects, methods, attributes, and variables in the name. Because the standard essentially provides a set of rules about naming various identifiers, it can be and has been used with

other languages, such as Ada, Java, and even C, as well. Another example is in Java, which, by convention, uses all uppercase for constants such as `PI` and `E`. Moreover, some classes use a trailing underscore to distinguish an attribute like `x_` from a method like `x( )`.

A general problem with style standards, like the Hungarian notation, is that they can lead to mangled variable names, and that they direct the programmer's focus on how to name in "Hungarian" rather than choosing a meaningful name of the variable for its use in code. In other words, the desire to conform to the standard may not always result in a particularly meaningful variable name. Another problem is that the very strength of a coding standard can also be its undoing. For example, in Hungarian notation, what if the type information embedded in the object name is, in fact, wrong? There is no way for any compiler to recognize this mistake. There are commercial rules wizards, reminiscent of the C language checking tool, lint, which can be tuned to enforce coding standards, but they must be programmed to work in conjunction with the compiler. Moreover, they can miss certain inconsistencies, leading the developers to a sense of false confidence.

Finally, adoption of coding standards is not recommended mid-project. It is easier and more motivating to start conforming from the beginning than to be required to change an existing style to comply. The decision to use a specific coding standard is an organizational one that requires significant forethought and open discussion.

4.2 ASSEMBLY LANGUAGE

In the mid- to late 1970s, when the first high-level languages became available for microprocessors, some college instructors told their students that "in five years, nobody will be writing real applications in assembly language." However, after more than 30 years from those days, assembly language still has a continuing, but limited role in real-time programming.

What are the reasons behind this? Well, although lacking user-friendliness and productivity features of high-level languages, assembly language does have a particular advantage for use in real-time programming; it provides the most direct control of the computer hardware over high-level languages. This advantage has extended the use of assembly language in real-time systems, despite the fact that assembly language is unstructured and has very limited abstraction properties. Moreover, assembly-language syntax varies greatly from processor to processor. Coding in assembly language is, in general, time-consuming to learn, tedious, and error prone. Finally, the resulting code is not easily ported across different processors, and hence the use of assembly language in embedded real-time systems—or in any professional system—is to be discouraged.

A generation ago, the best programmers could generate assembly code that was often more efficient than the code generated by a procedural-language

compiler. But with significant improvements in optimizing compilers over the past decades, this is rarely the case today—if you can write your program in a procedural language like C, the compiler should be able to generate very efficient machine-language code in terms of execution speed and memory usage. Thus, the need to write assembly code exists only in special circumstances when the compiler does not support certain machine-language instructions, or when the timing constraints are so tight that manual tuning is the only way to produce code that fulfills the extreme response-time requirements. Furthermore, you will find assembly-language code in many legacy real-time applications, and even today you can still occasionally encounter situations where small portions of a real-time system need to be written using assembly language. We will discuss some of these situations shortly.

In terms of Cardelli's criteria of various economies, assembly languages have excellent economy of execution, and vacuously, of compilation too because they are not compiled. Assembly languages, however, have poor economies of small- and large-scale development and of language features (see Fig. 4.2). Hence, assembly language programming should be limited to use in very tight timing situations or in controlling hardware features that are not supported by the compiler. The continuing role of assembly language in this decade is summarized below:

- For certain kinds of code, such as interrupt handlers and for device drivers for unique hardware where the "intellectual distance" between the hardware and software needs to be minimized.
- For situations where predictable performance for the code is extremely difficult or impossible to obtain because of undesirable programming-language–compiler interactions.
- For effectively using all architectural features of a CPU, for instance, parallel adders and multipliers.
- For writing code with minimum execution time achievable for time-critical applications, such as sophisticated signal-processing algorithms with high sampling rates.
- For writing the entire software for custom-designed CPUs with a small instruction set (see Section 2.5.3)—if no high-level language support is available.
- For debugging hard problems below the level of high-level language code and tracing the stream of fetched instructions by a logic analyzer.
- For teaching and learning computer architectures and internal operation of processors.

To deal with these special situations, the software developer will usually write a shell of the program in a high-level language and compile the code to an intermediate assembly representation, which is then fine-tuned manually to

obtain the desired effect. Some languages, such as Ada, provide a way for assembly code to be placed inline with the high-level-language code. In any case, the use of assembly language within a real-time system must be done reluctantly and with extreme caution.

4.3 PROCEDURAL LANGUAGES

Procedural languages such as Ada, C, Fortran, and Visual Basic, are those in which the action of the program is defined by a set of operations executed in sequence. These languages are characterized by facilities that allow for instructions to be grouped together into procedures or modules. Appropriate structuring of the procedures allows for achievement of desirable properties of the software, for example, modularity, reliability, and reusability.

There are several programming-language features standing out in procedural languages that are of interest in real-time systems, particularly:

- Modularity
- Strong typing
- Abstract data typing
- Versatile parameter passing mechanisms
- Dynamic memory allocation facilities
- Exception handling

These language features, to be discussed shortly, help promote the desirable properties of software design and best real-time implementation practices.

4.3.1 Modularity and Typing Issues

Procedural languages that are amenable to the principle of information hiding tend to promote the construction of high-integrity real-time systems. While C and Fortran both have mechanisms that can support information hiding (procedures and subroutines), other languages, such as Ada, tend to foster more *modular design* because of the requirement to have clearly defined inputs and outputs in the module parameter lists.

In Ada, the notion of a package embodies the concept of Parnas information hiding (Parnas, 1972) exquisitely. The Ada package consists of a specification and declarations that include its public or visible interface and its private or invisible elements. In addition, the package body, which has more externally invisible components, contains the working code of the package. Individual packages are separately compilable entities, which further enhances their application as black boxes. Furthermore, the C language provides for separately compiled modules and other features that promote a rigorous top-down design approach, which should lead to a solid modular design.

While modular software is desirable for many reasons, there is a price to pay in the overhead associated with procedure calls and essential parameter passing. This adverse effect should be considered carefully when sizing modules.

Typed languages require that each variable and constant be of a specific type (e.g., Boolean, integer, or real), and that each be declared as such before use. *Strongly typed* languages prohibit the mixing of different types in operations and assignments, and thus force the programmer to be exact about the way data are to be handled. Precise typing can prevent corruption of data through unwanted or unnecessary type conversion. Moreover, compiler type-checking is an important step to find errors at compile time, rather than at runtime, when they are more costly to repair. Hence, strongly typed languages are truly desirable for real-time systems.

Generally, high-level languages provide integer and real types, along with Boolean, character, and string types. In some cases, *abstract data types* are supported, too. These allow programmers to define their own types along with the associated operations. Use of abstract data types, however, may incur an execution-time penalty, as complicated internal representations are often needed to support the abstraction.

Some languages are typed, but do not prohibit mixing of types in arithmetic operations. Since these languages generally perform mixed calculations using the type that has the *highest storage complexity*, they must promote all variables to that highest type. For example, in C, the following code fragment illustrates automatic promotion and demotion of variable types:

```
int x,y;
float a,b;
y=x*a+b;
```

Here the variable `x` will be promoted to a `float` (real) type and then multiplication and addition will take place in floating point. Afterward, the result will be truncated and stored in `y` as an integer. The negative performance impact is that hidden promotion and more time-consuming arithmetic instructions are generated, with no additional accuracy achieved. Accuracy can be lost due to the truncation, or worse, an integer overflow can occur if the real value is larger than the allowable integer value. Programs written in languages that are weakly typed need to be scrutinized for such effects. Fortunately, most C compilers can be tuned to catch type mismatches in function parameters, preventing unwanted type conversions.

4.3.2 Parameter Passing and Dynamic Memory Allocation

There are several methods of *parameter passing*, including the use of parameter lists and global variables. While each of these techniques has preferred uses, each has a different performance impact as well. Note that these

parameter-passing mechanisms are also found in object-oriented programming languages.

The two most widely available parameter-passing methods are *call-by-value* and *call-by-reference*. In call-by-value parameter passing, the value of the actual parameter in the procedure call is copied into the procedure's formal parameter. Since the procedure manipulates the formal parameter only, the actual parameter is not altered. This technique is useful either when a test is being performed or the output is a function of the input parameters. For instance, in passing accelerometer readings from the 10-ms cycle to the 40-ms cycle, the raw data need not be returned to the calling routine in changed form. When parameters are passed using call-by-value, they are copied onto a runtime stack, at additional execution-time cost.

In call-by-reference (or call-by-address), the address of the parameter is passed by the calling routine to the called procedure so that the corresponding memory content can be altered there. Execution of a procedure using call-by-reference can take longer than one using call-by-value, since in call-by-reference, indirect addressing mode instructions are needed for any operations involving the variables passed. However, in the case of passing large data structures, such as buffers between procedures, it is more desirable to use call-by-reference, since passing a pointer is more efficient than passing the data by byte.

Parameter lists are likely to promote modular design because the interfaces between the modules are clearly defined. Clearly defined interfaces can reduce the potential of untraceable corruption of data by procedures using global access. However, both call-by-value and call-by-reference parameter-passing techniques can impact real-time performance when the lists are long, since interrupts are often disabled during parameter passing to preserve the integrity of the data passed. Moreover, call-by-reference may introduce subtle function side effects, depending on the compiler.

Before deciding on a specific set of rules concerning parameter passing for optimum performance, it is advisable to construct a set of test cases that exercise different alternatives. These test cases need to be rerun every time the compiler, hardware, or application changes in order to update the rules.

Global variables are variables that are within the scope of all code. "Within scope" usually means that references to these variables can be made with minimal memory fetches to resolve the target address, and thus are faster than references to variables passed via parameter lists, which require additional memory references. For example, in many image-processing applications, global arrays are defined to represent entire images, hence allowing costly parameter passing to be avoided.

However, global variables are dangerous because references to them can be made by unauthorized code, potentially introducing faults that can be hard to isolate. Use of global variables also violates the principle of information hiding, making the code difficult to understand and maintain. Therefore, unnecessary and wanton use of global variables is to be avoided. Global

parameter passing is only recommended when timing constraints so require, or if the use of parameter lists leads to obfuscated code. In any case, the use of global variables must be strictly coordinated and clearly documented.

The decision to use one method of parameter passing or the other may represent a trade-off between good software engineering practice and performance needs. For instance, often timing constraints force the use of global parameter passing in instances when parameter lists would have been preferred for clarity and maintainability.

Most programming languages provide recursion in that a procedure can either call itself or use itself in its construction. While recursion may be elegant and is sometimes necessary, its adverse impact on real-time performance must be considered. Procedure calls require the allocation of storage on the stack for the passing of parameters and for storage of local variables. The execution time needed for the allocation and deallocation, as well as for storing and retrieving those parameters and local variables, can be costly. In addition, recursion necessitates the use of a large number of expensive memory- and register-indirect instructions. Moreover, precautions need to be taken to ensure that the recursive routine will terminate, otherwise the runtime stack will eventually overflow. The use of recursion often makes it impossible to determine the exact size of runtime memory requirements. Thus, iterative techniques, such as `while` and `for` loops, must be used where performance and determinism are crucial or naturally in those languages that do not support recursion.

The ability to dynamically allocate memory is important in the construction and maintenance of many data structures needed in real-time systems. While *dynamic memory allocation* can be time-consuming, it is necessary, especially in the construction of interrupt handlers, memory managers, and the like. Linked lists, trees, heaps, and other dynamic data structures can benefit from the clarity and economy introduced by dynamic memory allocation. Furthermore, in cases where just a pointer is used to pass a data structure, the overhead for dynamic allocation can be reasonable. When coding real-time systems, however, care should be taken to ensure that the compiler will always pass pointers to large data structures and not the data structures themselves.

Languages that do not allow dynamic allocation of memory, for example, some primitive high-level languages or assembly language require data structures of fixed size. While this may be faster, flexibility is sacrificed and memory requirements must be predetermined. Modern procedural languages, such as Ada, C, and Fortran 2003, have dynamic allocation facilities.

4.3.3 Exception Handling

Some programming languages provide facilities for dealing with errors or other anomalous conditions that may arise during program execution. These conditions include the obvious, such as floating-point overflow, square root of a negative argument, divide-by-zero, as well as possible user-defined ones. The ability

to define and handle exceptional conditions in the high-level language aids in the construction of interrupt handlers and other critical code used for real-time event processing. Moreover, poor handling of exceptions can degrade performance. For instance, floating-point overflow errors can propagate bad data through an algorithm and instigate time-consuming error-recovery routines.

In ANSI-C, the `raise` and `signal` facilities are provided for creating exception handlers. A `signal` is a type of software interrupt handler that is used to react to an exception indicated by the `raise` operation. Both are provided as function calls, which are typically implemented as macros.

The following prototype can be used as the front end for an exception handler to react to signal `S`.

```
void (*signal (int S, void (*func) (int)))(int);
```

When signal `S` is set, function `func` is invoked. This function represents the actual interrupt handler. In addition, we need a complementary prototype:

```
int raise (int S);
```

Here `raise` is used to invoke the task that reacts to signal `S`.

ANSI-C includes a number of predefined signals needed to handle anomalous conditions, such as overflow, memory access violations, and illegal instruction, but these signals can be replaced with user-defined ones. The following C code portrays a generic exception handler that reacts to a certain error condition:

```
#include <signal.h>
main ()
{
 void handler (int sig);
 ...
 signal (SIGINT, handler);    /* SIGINT handler */
 ... /* do some processing */
 if (error) raise (SIGINT);   /* anomaly detected */
 ... /* continue processing */
}
void handler (int sig)
{
 ... /* handle error here */
}
```

In the C language, the `signal` library-function call is used to construct interrupt handlers to react to a signal from external hardware and to handle certain traps, such as floating-point overflow, by replacing the standard C library handlers.

Of the procedural languages discussed in this chapter, Ada has the most explicit exception handling facility. Consider an Ada exception handler to determine whether a square matrix is singular (i.e., its determinant is zero). Assume that a matrix type has been defined, and it can be determined that the matrix is singular. An associated code fragment might be:

```
begin
-- calculate determinant
-- ...
--
 exception
 when SINGULAR : NUMERIC/ERROR => PUT ("SINGULAR");
 when others => PUT ("FATAL Error");
 raise ERROR;
end;
```

Here, the `exception` keyword is used to indicate that this is an exception handler and the `raise` keyword plays a role similar to that of `raise` in the C exception handler just presented. The definition of `SINGULAR`, which represents a matrix whose determinant is zero, is defined elsewhere, such as in a header file.

4.3.4 Cardelli's Metrics and Procedural Languages

Taking the common set of procedural languages as a whole, Cardelli considered them for use in real-time systems with respect to his criteria. His comments are paraphrased in the foregoing discussion. First, he notes that variable typing was introduced to improve code generation. Hence, economy of execution is high for procedural languages provided the compiler is efficient. Further, because modules can be compiled independently, compilation of large systems is efficient, at least when interfaces are stable. The more challenging aspects of system integration are thus eliminated.

Small-scale development is economical since type checking can catch many coding errors, reducing testing and debugging efforts. The errors that do occur are easier to debug, simply because large classes of other errors have been ruled out. Finally, experienced programmers usually adopt a coding style that causes some logical errors to show up as type checking errors; hence, they can use the type checker as a development tool. For instance, changing the name of a type when its invariants change even though the type structure remains the same yields error reports on all its previous uses.

Moreover, data abstraction and modularization have methodological advantages for large-scale code development. Large teams of programmers can negotiate the interfaces to be implemented, and then proceed separately to implement the corresponding pieces of code. Dependencies between such pieces of code are minimized, and code can be locally rearranged without any fear of global effects.

Finally, procedural languages are economical because certain well-designed constructions can be naturally composed in orthogonal ways. For instance, in C, an array of arrays models two-dimensional arrays. Orthogonality of language features reduces the complexity of a programming language. The learning curve for programmers is thus reduced, and the relearning effort that is constantly necessary in using complex languages is minimized (Cardelli, 1996).

4.4 OBJECT-ORIENTED LANGUAGES

The benefits of object-oriented languages, such as improved programmer productivity, increased software reliability, and higher potential for code reuse, are well known and appreciated. Object-oriented languages include Ada, C++, C#, and Java. Formally, object-oriented programming languages are those that support data *abstraction*, *inheritance*, *polymorphism*, and *messaging*.

Objects are an effective way to manage the increasing complexity of real-time systems, as they provide a natural environment for information hiding, or protected variation and encapsulation. In encapsulation, a class of objects and methods associated with them are enclosed or encapsulated in class definitions. An object can utilize another object's encapsulated data only by sending a message to that object with the name of the method to apply. For example, consider the problem of sorting objects. A method may exist for sorting an object class of integers in ascending order. A class of people might be sorted by their height. A class of image objects that has an attribute of color might be sorted by that attribute. All these objects have a comparison message method with different implementations. Therefore, if a client sends a message to compare one of these objects to another, the runtime code must resolve which method to apply dynamically—with obvious execution-time penalty. This matter will be discussed shortly.

Object-oriented languages provide a fruitful environment for information hiding; for instance, in image-processing systems, it might be useful to define a class of type pixel, with attributes describing its position, color, and brightness, and operations that can be applied to a pixel, such as add, activate, and deactivate. It might also be desirable to define objects of type image as a collection of pixels with other attributes of width, height, and so on. In certain cases, expression of system functionality is easier to do in an object-oriented manner.

4.4.1 Synchronizing Objects and Garbage Collection

Rather than extending classes through inheritance, in practice, it is often preferable to use composition. However, in doing so, there is the need to support different synchronization policies for objects, due to different usage contexts. Specifically, consider the following common synchronization policies for objects:

- *Synchronized Objects.* A synchronization object, such as a mutex, is associated with an object that can be concurrently accessed by multiple threads. If internal locking is used, then on method entry, each public method acquires a lock on the associated synchronization object and releases the lock on method exit. If external locking is used, then clients are responsible for acquiring a lock on the associated synchronization object before accessing the object and subsequently releasing the lock when finished.
- *Encapsulated Objects.* When an object is encapsulated within another object (i.e., the encapsulated object is not accessible outside of the enclosing object), it is redundant to acquire a lock on the encapsulated object, since the lock of the enclosing object also protects the encapsulated object. Operations on encapsulated objects therefore require no synchronization.
- *Thread-Local Objects.* Objects that are only accessed by a single thread require no synchronization.
- *Objects Migrating between Threads.* In this policy, ownership of a migrating object is transferred between threads. When a thread transfers ownership of a migrating object, it can no longer access it. When a thread receives ownership of a migrating object, it is guaranteed to have exclusive access to it (i.e., the migrating object is local to the thread). Hence, migrating objects require no synchronization. However, the transfer of ownership does require synchronization.
- *Immutable Objects.* An immutable object's state can never be modified after it is instantiated. Thus, immutable objects require no synchronization when accessed by multiple threads since all accesses are read-only.
- *Unsynchronized Objects.* Objects within a single-threaded program require no synchronization.

To illustrate the necessity of supporting parameterization of synchronization policies, consider a class library. A developer of a class library wants to ensure the widest possible audience for this library, so he makes all classes synchronized so that they can be used safely in both single-threaded and multi-threaded applications. However, clients of the library whose applications are single-threaded are unduly penalized with the unnecessary execution overhead of synchronization that they do not need. Even multi-threaded applications can be unduly penalized if the objects do not require synchronization (e.g., the objects are thread-local). Therefore, to promote reusability of a class library without sacrificing performance, classes in a library ideally would allow clients to select on a per-object basis which synchronization policy to use.

Garbage refers to allocated memory that is no longer being used but is not otherwise available either. Excessive garbage accumulation can be detrimental, and therefore garbage must be regularly reclaimed. Garbage collection algorithms generally have unpredictable performance, although average

performance may be known. The loss of determinism results from the unknown amount of garbage, the tagging time of the nondeterministic data structures, and the fact that many incremental garbage collectors require that every memory allocation or deallocation from the heap be willing to service a page-fault trap handler.

Furthermore, garbage can be created in both procedural and object-oriented languages. For example, in C, garbage is created by allocating memory, but not deallocating it properly. Nonetheless, garbage is generally associated with object-oriented languages like C++ and Java. Java is noteworthy in that the standard environment incorporates garbage collection, whereas C++ does not.

4.4.2 Cardelli's Metrics and Object-Oriented Languages

Consider object-oriented languages in the context of Cardelli's metrics as paraphrased from his analysis. In terms of economy of execution, object-oriented style is intrinsically less efficient than procedural style. In pure object-oriented style, every routine is supposed to be a method. This introduces additional indirections through method tables and prevents straightforward code optimizations, such as inlining. The traditional solution to this problem (analyzing and compiling whole programs) violates modularity and is not applicable to libraries.

With respect to economy of compilation, often there is no distinction between the code and the interface of a class. Some object-oriented languages are not sufficiently modular and require recompilation of superclasses when compiling subclasses. Hence, the time spent in compilation may grow disproportionately with the size of the system.

On the other hand, object-oriented languages are superior with respect to economy of small-scale development. For example, individual programmers can take advantage of class libraries and frameworks, drastically reducing their workload. When the project scope grows, however, programmers must be able to understand the details of those class libraries, and this task turns out to be more difficult than understanding typical module libraries. The type systems of most object-oriented languages are not expressive enough; programmers must often resort to dynamic checking or to unsafe features, damaging the robustness of their programs.

In terms of economy of large-scale development, many developers are frequently involved in developing new class libraries and tailoring existing ones. Although reuse is a benefit of object-oriented languages, it is also the case that these languages have extremely poor modularity properties with respect to class extension and modification via inheritance. For instance, it is easy to override a method that should not be overridden, or to reimplement a class in a way that causes problems in subclasses. Other large-scale development problems include the confusion between classes and object types, which

limits the construction of abstractions, and the fact that subtype polymorphism is not good enough for expressing container classes.

Object-oriented languages have low economy of language features. For instance, C++ is based on a fairly simple model, but is overwhelming in the complexity of its many features. Unfortunately, what started as economical and uniform language ("everything is an object") ended up as a vast collection of class varieties. Java, on the other hand, represents a step toward reducing complexity, but is actually more complex than most people realize (Cardelli, 1996).

4.4.3 Object-Oriented versus Procedural Languages

There is no general agreement on which is better for real-time systems—object-oriented or procedural languages. This is partially due to the fact that there is a huge variety of real-time applications—from nonembedded airline booking and reservation systems to embedded wireless sensors in running shoes, for example.

The benefits of an object-oriented approach to problem solving and the use of object-oriented languages are clear, and have already been described. Moreover, it is possible to imagine certain aspects of a real-time operating system that would benefit from objectification, such as process, thread, file, or device. Furthermore, certain application domains can clearly benefit from an object-oriented approach. The main arguments against object-oriented programming languages for real-time systems, however, are that they can lead to unpredictable and inefficient systems, and that they are hard to optimize. Nonetheless, we can confidently recommend object-oriented languages for soft and firm real-time systems.

The unpredictability argument is hard to defend, however, at least with respect to object-oriented languages, such as C++, that *do not use garbage collection*. It is likely the case that a predictable system—also a hard real-time one—can be just as easily built in C++ as C. Similarly, it is probably just as easy to build an unpredictable system in C as in C++. The case for more unpredictable systems using object-oriented languages is easier to sustain when arguing about garbage-collecting languages like Java.

In any case, the inefficiency argument against object-oriented languages is a powerful one. Generally, there is an execution-time penalty in object-oriented languages in comparison with procedural languages. This penalty is due in part to late binding (resolution of memory locations at runtime rather than at compile time) necessitated by function polymorphism, inheritance, and composition. These effects present considerable and often uncertain delay factors. Another problem results from the overhead of the garbage collection routines. One way to reduce these penalties is not to define too many classes and only define classes that contain coarse detail and high-level functionality.

Vignette: Object-Oriented Languages Lack Certain Flexibility

> The following anecdote (reported by one of Laplante's clients who prefers to remain anonymous) illustrates that the use of object-oriented language for real-time systems may present subtle difficulties as well. A design team for a particular real-time system insisted that C++ be used to implement a fairly simple and straightforward requirements specification. After coding was complete, testing began. Although the developed system never failed, several users wished to add a few requirements; however, adding these features caused the real-time system to miss important deadlines. The client then engaged an outside vendor to implement the revised design using a procedural language. The vendor met the new requirements by writing the code in C and then hand-optimizing certain assembly-language sections from the compiler output. They could use this optimization approach because of the close correspondence between the procedural C code and compiler-generated assembly-language instructions. This straightforward option was not available to developers using C++.

The vignette is not an endorsement of this solution strategy, however. It is simply an illustration of a very special case. Sometimes such cases are used to dispute the viability of object-oriented languages for real-time applications, which is not fair—many punctual and robust real-time systems are built in object-oriented languages. Moreover, while the client's problem was solved in the straightforward manner described in the vignette, it is easy to see that the understandability, maintainability, and portability of the system will be problematic. Hence, the solution on the client's case should have involved a complete reengineering of the system to include reevaluation of the deadlines and the overall system architecture.

A more general problem is the inheritance anomaly in object-oriented languages. The inheritance anomaly arises when an attempt is made to use inheritance as a code reuse mechanism, which does not preserve substitutability (i.e., the subclass is not a subtype). If the substitutability were preserved, then the anomaly would not occur. Since the use of inheritance for reuse has fallen out of favor in object-oriented approaches (in favor of composition), it seems that most inheritance anomaly rejections of object-oriented languages for real-time systems reflect an antiquated view of object orientation.

Consider the following example from an excellent text on real-time operating systems (Shaw, 2001):

```
BoundedBuffer
{
 DEPOSIT
 pre: not full
 REMOVE
 pre: not empty
```

```
}
MyBoundedBuffer extends BoundedBuffer
{
 DEPOSIT
 pre: not full
 REMOVE
 pre: not empty AND lastInvocationIsDeposit
}
```

Assuming that preconditions are checked and have "wait semantics" (i.e., wait for the precondition to become true), then clearly `MyBoundedBuffer` has strengthened the precondition of `BoundedBuffer`, and hence violated substitutability—and as such is a questionable use of inheritance.

Most opponents of object-oriented languages for real-time programming assert that concurrency and synchronization are poorly supported. However, when built-in language support for concurrency does not exist, it is a standard practice to create "wrapper-facade" classes to encapsulate system-concurrency application program interface (API) for use in object orientation (e.g., wrapper classes in C++ for POSIX threads). Furthermore, there are several concurrency patterns available for object-oriented real-time systems (Douglass, 2003; Schmidt et al., 2000). While concurrency may be poorly supported at the language level, it is not an issue since developers use libraries instead.

In summary, critics of current object-oriented languages for real-time systems seem fixated on Java, ignoring C++. C++ is more suitable for real-time programming since, among other things, it does not have built-in garbage collection and class methods, and by default does not use "dynamic binding." In any case, there are no strict guidelines when object-oriented approaches and languages should be preferred. Each specific situation needs to be considered individually.

4.5 OVERVIEW OF PROGRAMMING LANGUAGES

For purposes of illustrating the aforementioned language properties, it is useful to review some of the languages that are currently used in programming real-time systems. Selected procedural and object-oriented languages are discussed in alphabetical order, and not in any rank of endorsement or salient properties.

4.5.1 Ada

Ada was originally planned to be the mandatory language for all U.S. DoD projects that included a high proportion of embedded systems. The first version, which became standardized by 1983, had rather serious problems. Ada was intended to be used specifically for programming real-time systems, but, at the

time, systems builders found the resulting executable code to be bulky and inefficient. Moreover, major problems were discovered when trying to implement multitasking using the limited tools supplied by the language, such as the roundly criticized rendezvous mechanism. The programming language community had been aware of these problems, and virtually since the first delivery of an Ada 83 compiler, had sought to resolve them. These reform efforts eventually resulted in a new version of the language. The thoroughly revised language, called "Ada 95," is considered the first internationally standardized *object-oriented* programming language, and, in fact, some individuals refer to Ada 95 as the "first real-time language."

Three particularly useful constructs were introduced in Ada 95 to resolve shortcomings of Ada 83 in scheduling, resource contention, and synchronization:

1. A pragma that controls how tasks are dispatched.
2. A pragma that controls the interaction between task scheduling.
3. A pragma that controls the queuing policy of task- and resource-entry queues.

Moreover, other additions to the language strived to make Ada 95 fully object-oriented. These included:

- Tagged types
- Packages
- Protected units

Proper use of these constructs allows for the construction of objects that exhibit the four characteristics of object-oriented languages: abstract data typing, inheritance, polymorphism, and messaging.

In October 2001, a Technical Corrigendum to the Ada 95 Standard was announced by ISO/IEC, and a major Amendment to the international standard was published in March 2007. This latest version of Ada is called "Ada 2005." The differences between Ada 95 and Ada 2005 are not extensive—in any case, not nearly as significant as the changes between Ada 83 and Ada 95. Therefore, when we refer to "Ada" for the remainder of this book, we mean Ada 95, since Ada 2005 is not a new standard, but just an amendment.

The amendment, ISO/IEC 8652:1995/Amd 1:2007, includes a few changes that are of particular interest to the real-time systems community, such as:

- The real-time systems Annex contains additional dispatching policies, support for timing events, and support for control of CPU-time utilization.
- The object-oriented model was improved to provide multiple inheritance.
- The overall reliability of the language was enhanced by numerous improvements.

Ada has never lived up to its promise of universality. Nevertheless, the revised language is staging somewhat of a comeback, particularly because selected new DoD systems and many legacy systems use Ada, and because of the availability of open-source versions of Ada for the popular Linux environment.

4.5.2 C

The C programming language, invented around 1972 at Bell Laboratories, is a good language for "low-level" programming. The reason for this is that it is descended from the clear-cut language, BCPL (whose successor, C's parent, was "B"), which supported only one type, the machine word. Consequently, C supports machine-related items like addresses, bits, bytes, and characters, which are handled directly in this high-level language. These basic entities can be used effectively to control the CPU's work registers, peripheral interface units, and other memory-mapped hardware needed in real-time systems.

The C language provides special variable types, such as `register`, `volatile`, `static`, and `constant`, which allow for effective control of code generation at the procedural language level. For example, declaring a variable as a `register` type indicates that it will be used frequently. This guides the compiler to place such a declared variable in a work register, which often results in faster and smaller programs. Furthermore, C supports call-by-value only, but call-by-reference can be implemented easily by passing a pointer to anything as a value. Variables declared as type `volatile` are not optimized by the compiler at all. This feature is necessary in handling memory-mapped I/O and other special instances where the code should not be optimized.

Automatic coercion refers to the implicit casting of data types that sometimes occurs in C. For example, a `float` value can be assigned to an `int` variable, which can result in a loss of information due to truncation. Moreover, C provides functions, such as `printf`, that take a variable number of arguments. Although this is a convenient feature, it is impossible for the compiler to thoroughly type check the arguments, which means problems may mysteriously arise at runtime.

The C language provides for exception handling through the use of signals, and two other mechanisms, `setjmp` and `longjmp`, are provided to allow a procedure to return quickly from a deep level of nesting—a particularly useful feature in procedures requiring an abort. The `setjmp` procedure call, which is actually a macro (but often implemented as a function), saves environment information that can be used by a subsequent `longjmp` library function call. The `longjmp` call restores the program to the state at the time of the last `setjmp` call. For example, suppose a procedure is called to do some processing and error checking. If an error is detected, a `longjmp` can be used to transfer to the first statement after the `setjmp`.

Overall, the C language is particularly good for embedded programming, because it provides for structure and flexibility without complex language

restrictions. The latest version of the international standard of C language is from 1999 (ANSI/ISO/IEC 9899:1999).

4.5.3 C++

C++ is a hybrid object-oriented programming language that was originally implemented as a macro extension of C in the 1980s. Today, C++ stands as an individual compiled language, although C++ compilers should accept standard C code as well. C++ exhibits all characteristics of an object-oriented language and promotes better software-engineering practice through encapsulation and more advanced abstraction mechanisms than C.

C++ compilers implement a preprocessing stage that basically performs an intelligent search-and-replace on identifiers that have been declared using the `#define` or `#typedef` directives. Although most advocates of C++ discourage the use of the preprocessor, which was inherited from C, it is rather widely used. Most of the preprocessor definitions in C++ are stored in header files, which complement the actual source-code files. The problem with the preprocessor approach is that it provides a way for programmers to inadvertently add unnecessary complexity to a program. An additional problem with the preprocessor approach is that it has weak type checking and validation.

Most software developers agree that the misuse of pointers causes the majority of bugs in C/C++ programming. Previously C++ programmers used complex pointer arithmetic to create and maintain dynamic data structures, particularly during string manipulation. Consequently, they spent a lot of time hunting down complicated bugs for simple string management. Today, however, standard libraries of dynamic data structures are available. For example, the standard template language (STL), is a standard library of C++, and it has both a `string` and `wstring` data type for regular and wide character strings, respectively. These data types neutralize any arguments against early C++ releases, which were based on string manipulation issues.

There are three complex data types in C++: classes, structures, and unions. However, C++ has no built-in support for text strings. The standard technique is to use `null`-terminated arrays of characters to represent strings.

The regular C code is organized into functions, which are global subroutines accessible to a program. C++ adds classes and class methods, which are actually functions that are connected to classes. However, because C++ still supports C, there is nothing, in principle, to prevent C++ programmers from using the regular functions. This would result in a mixture of function and method use that would create confusing programs, however.

Multiple inheritance is a helpful feature of C++ that allows a class to be derived from multiple parent classes. Although multiple inheritance is indeed powerful, it may be difficult to use correctly and causes many problems otherwise. It is also complicated to implement from the compiler perspective.

Today, more and more embedded systems are being constructed in C++, and many practitioners ask, “Should I implement my system in C or C++?”

The immediate answer is always "it depends." Choosing C in lieu of C++ in embedded applications is a difficult trade-off: a C program would be faster and more predictable but harder to maintain, and C++ program would be slower and less predictable but potentially easier to maintain. So, language choice is tantamount to asking should I eat a "green apple" or a "red apple?"

C++ still allows for low-level control; for instance, it can use inline methods rather than a runtime call. This kind of implementation is not particularly abstract, nor completely low-level, but is acceptable in typical embedded environments.

To its detriment, there may be some tendency to take existing C code and objectify it by simply wrapping the procedural code into objects with little regard for the best practices of object-orientation. This kind of approach is to be strictly avoided since it has the potential to incorporate all of the disadvantages of C++, and none of its benefits. Furthermore, C++ does not provide automatic garbage collection, which means dynamic memory must be managed manually or garbage collection must be homegrown. Therefore, when converting a C program to C++, a complete redesign is required to fully capture all of the advantages of an object-oriented design while minimizing the runtime disadvantages.

4.5.4 C#

C# (pronounced "C sharp") is a C++-like language that, along with its operating environment, has similarities to Java and the Java virtual machine, respectively. Thus, C# is first compiled into an intermediate language, which is then used to generate a native image at runtime. C# is associated with Microsoft's .NET framework for scaled-down operating systems like Windows CE. Windows CE is highly configurable, capable of scaling from small, embedded system footprints (<1 M bytes) and upwards (e.g., for real-time systems requiring user-interface support). The minimum kernel configuration provides basic networking support, thread management, dynamic link library support, and virtual memory management. While a detailed discussion is beyond the scope of this text, it is clear that Windows CE was originally intended as a real-time operating system for the .NET platform.

Much of this discussion is adapted from Lutz and Laplante (2003).

C# supports "unsafe code," allowing pointers to refer to specific memory locations. Objects referenced by pointers must be explicitly "pinned," disallowing the garbage collector from altering their location in memory. The garbage collector collects pinned objects; it just does not move them. This capability could increase schedulability, and it also allows for direct memory access (DMA) to write to specific memory locations; a necessary capability in embedded real-time systems. .NET offers a generational approach to garbage collection intended to minimize thread blockage during mark and sweep. For instance, a means to create a thread at a particular instant, and guarantee the thread completes by a particular point in time, is not supported. Moreover, C#

provides many thread synchronization mechanisms, but none with this level of precision. C# supports an array of thread-synchronization constructs: lock, monitor, mutex, and interlock. A `Lock` is semantically identical to a critical section—a code segment guaranteeing entry into itself by only one thread at a time. `Lock` is a shorthand notation for the monitor class type. A mutex is semantically equivalent to a lock, with the additional capability of working across process spaces. The downside to mutexes is their performance penalty. Finally, interlock, a set of overloaded static methods, is used to increment and decrement numerics in a thread-safe manner in order to implement the priority-inheritance protocol.

Timers that are similar in functionality to the widely used Win32 timer exist in C#. When constructed, timers are configured how long to wait in milliseconds before their first invocation, and are also supplied an interval, again in milliseconds, specifying the period between subsequent invocations. The accuracy of these timers is machine dependent, and thus not guaranteed, reducing their usefulness in real-time systems to be used in multiple hardware platforms.

C# and the .NET platform are not appropriate for the majority of hard real-time systems for several reasons, including the unbounded execution of its garbage-collection environment and its lack of threading constructs to adequately support schedulability and determinism. Nonetheless, C#'s ability to interact effectively with operating-system APIs, shield developers from complex memory management logic, together with C#'s good floating-point performance, make it a programming language that is highly potential for soft and even firm real-time applications. However, disciplined programming style is required (Lutz and Laplante, 2003).

4.5.5 Java

Java, in the same way as C#, is an interpreted language, that is, the code compiles into machine-independent intermediate code that runs in a managed execution environment. This environment is a virtual machine (see Fig. 4.3), which executes "object" code instructions as a series of program directives.

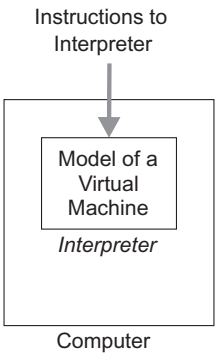


Figure 4.3. The Java interpreter as a model of a virtual machine.

The obvious advantage of this arrangement is that Java code can run on any device that implements the virtual machine. This "write once, run anywhere" philosophy has important applications in mobile and portable computing, such as in cell phones and smart cards, as well as in Web-based computing.

However, there are also native-code Java compilers, which allow Java to run directly "on the bare metal," that is, the compilers convert Java to assembly code or object code. For example, beginning with Java 2, Java virtual machines support special compilers that compile into machine code for several standard architectures. Furthermore, there are even special Java microprocessors, which directly execute Java byte code in hardware (El-Kharashi and Elguibaly, 1997).

Java is an object-oriented language and the code appears very similar to C++. Like C, Java supports call-by-value, but call-by-reference can be simulated, which will be discussed shortly. But Java is a pure object-oriented language, that is, all functionality in Java has to be implemented by creating object classes, instantiating objects of those classes (or base classes), and manipulating objects' attributes through methods. Thus, it is virtually impossible to take legacy code that was written in a procedural language, say C, and "convert" that code to Java without truly embodying an object-oriented design approach. Of course, a good object-oriented design is not guaranteed, but the design obtained in the conversion will be a true object-oriented one based on the rules of the language. This situation is quite different from the kind of false object-oriented conversion that can be obtained from C to C++ in the blunt manner previously highlighted.

Java does provide a preprocessor. Constant data members are used in place of the `#define` directive, and class definitions are used in lieu of the `#typedef` directive. The result is that Java source code is typically more consistent and easier to read than C++ source code. The Java compiler builds class definitions directly from the source code files, which contain both class definitions and method implementations. However, there are natural performance penalties for the resultant portability.

The Java language does not support pointers, but it provides similar functionality via references. Java passes all arrays and objects by reference, which prevents common errors due to pointer mismanagement. The lack of pointers might seem to preclude implementation of data structures, such as dynamic arrays. However, any pointer functionality can be conveniently accomplished with references, with the safety provided by the Java runtime system, such as boundary checking on array indexing operations—all this with performance penalty.

Java only implements one complex data type: classes. Java programmers use classes when the functionality of structures and unions is desired. This consistency comes at the cost of increased execution time over simple data structures.

The Java language does not support standalone functions. Instead, Java requires programmers to bundle all routines into class methods again with significant cost.

Moreover, Java has no direct support for multiple inheritance. Interfaces, however, allow for implementation of multiple inheritance. Java interfaces provide object-method descriptions, but contain no implementations.

In Java, strings are implemented as first-class objects (`String` and `StringBuffer`), meaning that they are at the core of the Java language. Java's implementation of strings as objects provides several advantages. First, string creation and access is consistent across all systems. Next, because the Java string classes are defined as part of the Java language strings function predictably every time. Finally, the Java string classes perform extensive runtime checking, which helps eliminate errors. But all of these operations increase execution time.

Operator overloading is not supported in Java. However, in Java's string class, "+" represents concatenation of strings, as well as numeric addition.

The Java language does not support automatic coercions. In Java, if a coercion will result in a loss of data, then it is necessary to explicitly cast the data element to the new type. Java does have implicit "upcasting." However, any instance can be upcast to `Object`, which is the parent class for all objects. Downcasting is explicit, and requires a cast. This explicitness is important to prevent hidden loss of precision.

The command line arguments passed from the system into a Java program differ from the usual command-line arguments passed into a C++ program. In C and C++, the system passes two arguments to a program: `argc` and `argv`. `argc` specifies the number of arguments stored in `argv`, and `argv` is a pointer to an array of characters containing the actual arguments. In Java, on the other hand, the system passes a single value to a program: `args`. `args` is an array of strings that contains the command-line arguments.

4.5.6 Real-Time Java

This section is devoted to the real-time adaptation of Java. Although real-time Java is just a modification of the standard Java language, it deserves a separate discussion, because it is used increasingly in implementing soft, firm, and even hard real-time systems, while the standard Java is mainly used for soft real-time systems only. While we include the discussion on real-time Java for completeness and because it illustrates several interesting points, we reiterate our preference for C++ over versions of Java in most cases.

In addition to the unpredictable performance of garbage collection, the Java specification provides only broad guidance for scheduling. For example, when there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. This preference is not, however, a guarantee that the highest-priority one of ready threads will always be running, and thread priorities cannot be used to reliably implement mutual exclusion. It was soon recognized that this and other shortcomings rendered standard Java inadequate for most real-time systems.

In response to this problem, a National Institute of Standards and Technology (NIST) task force was charged with developing a version of Java that was particularly suitable for embedded real-time applications. The final workshop report, published already in September 1999, defines nine core requirements for the real-time specification of Java (RTSJ 1.0):

R1. The specification must include a framework for the lookup and discovery of available profiles.
R2. Any garbage collection that is provided shall have a bounded preemption latency.
R3. The specification must define the relationships among real-time Java threads at the same level of detail as is currently available in existing standards documents.
R4. The specification must include APIs to allow communication and synchronization between Java and non-Java tasks.
R5. The specification must include handling of both internal and external asynchronous events.
R6. The specification must include some form of asynchronous thread termination.
R7. The core must provide mechanisms for enforcing mutual exclusion without blocking.
R8. The specification must provide a mechanism to allow code to query whether it is running under a real-time Java thread or a nonreal-time Java thread.
R9. The specification must define the relationships that exist between real-time Java and nonreal-time Java threads.

The RTSJ 1.0 satisfies all but the first requirement, which was considered irrelevant because access to physical memory is not part of the NIST requirements, but industry input led the group to include it (Bollella and Gosling, 2000). In 2006, an enhanced version of the real-time specification, RTSJ 1.1, was announced (Dibble and Wellings, 2009).

Most of the following discussion has been adapted from Bollella and Gosling (2000).

The RTSJ defines the real-time thread class to create threads, which the resident scheduler executes. Real-time threads can access objects on the heap, and therefore can incur delays because of garbage collection.

For garbage collection, the RTSJ extends the memory model to support memory management in a way that does not interfere with the real-time code's ability to provide deterministic behavior. These extensions allow both short- and long-lived objects to be allocated outside the garbage-collection heap. There is also sufficient flexibility to use familiar solutions, such as preallocated object pools.

RTSJ uses "priority" somewhat more loosely than is traditionally accepted. "Highest priority thread" merely indicates the most eligible thread—the thread that the scheduler would choose from among all threads ready to run. It does not necessarily presume a strict priority-based dispatch mechanism.

The system must queue all threads waiting to acquire a resource in priority order. These resources include the processor as well as synchronized blocks. If the active scheduling policy permits threads with the same priority, the threads are queued using the FIFO principle. Specifically, the system (1) orders waiting threads to enter synchronized blocks in a priority queue; (2) adds a blocked thread that becomes ready to run to the end of the ready queue for that priority; (3) adds a thread whose priority is explicitly set by itself or another thread to the end of the ready queue for the new priority; and (4) places a thread that performs a yield to the end of its priority queue. The priority-inheritance protocol is implemented by default. The real-time specification also provides a mechanism by which a systemwide default policy can be implemented.

The asynchronous event facility comprises two classes: `AsyncEvent` and `AsyncEventHandler`. An `AsyncEvent` object represents something that can happen—like a hardware interrupt—or it represents a computed event—like an aircraft entering a monitored region. When one of these events occurs, indicated by the `fire()` method being called, the system schedules associated `AsyncEventHandlers`. An `AsyncEvent` manages two things: the dispatching of handlers when the event is fired, and the set of handlers associated with the event. The application can query this set and add or remove handlers. An `AsyncEventHandler` is a schedulable object roughly similar to a thread. When the event fires, the system invokes `run()` methods of the associated handlers.

Unlike other runable objects, however, an `AsyncEventHandler` has associated scheduling, release, and memory parameters that control the actual execution of read or write.

Asynchronous control transfer allows for identification of particular methods by declaring them to throw an `AsynchronouslyInterrupted Exception` (AIE). When such a method is running at the top of a thread's execution stack and the system calls `java.lang.Thread.interrupt()` on the thread, the method will immediately act as if the system had thrown an AIE. If the system calls an interrupt on a thread that is not executing such a method, the system will set the AIE to a pending state for the thread and will throw it the next time control passes to such a method, either by calling it or returning to it. The system also sets the AIE's state to "pending" while control is in, returns to, or enters synchronized blocks.

The RTSJ defines two classes for programmers who want to access physical memory directly from Java code. The first class, `RawMemoryAccess`, defines methods that let you build an object representing a range of physical addresses and then access the physical memory with `byte`, `word`, `long`, and multiple `byte` granularity. The RTSJ implies no semantics other than the set and get methods. The second class, `PhysicalMemory`, allows the construction of a

`PhysicalMemoryArea` object that represents a range of physical memory addresses where the system can locate Java objects. For example, a new Java object in a particular `PhysicalMemory` object can be built using either the `newInstance()` or `newArray()` methods. An instance of `RawMemoryAccess` models a raw storage area as a fixed-size sequence of bytes. Factory methods allow for the creation of `RawMemoryAccess` objects from memory at a particular address range or using a particular memory type. The implementation must provide and set a factory method that interprets these requests accordingly. A full complement of get and set methods lets the system access the physical memory area's contents through offsets from the base—interpreted as `byte`, `short`, `int`, or `long` data values—and copy them to or from `byte`, `short`, `int`, or `long` arrays.

4.5.7 Special Real-Time Languages

A large variety of specialized languages for real-time programming have appeared and received more or less success over the past decades. These include, for instance:

- *PEARL.* The process and experiment automation real-time language was developed in the early 1970s by a group of German researchers. PEARL uses the augmentation strategy and has fairly wide application in Germany, especially in industrial controls settings. The current version is PEARL-90.
- *Real-Time Euclid.* An experimental language also from the 1970s that enjoys the distinction of being one of the only languages to be completely suited for schedulability analysis. This is achieved through language restrictions. It descended from the Pascal programming language, but never found its way into mainstream applications.
- *Occam 2.* A language based on the communicating-sequential-processes formalism that was designed to support concurrency on transputers (see Section 2.5.2). It appeared in the late eighties and found practical implementations mainly in the United Kingdom, but disappeared together with the transputer.
- *Real-Time C.* Actually a generic name for any of a variety of C macroextension packages. These macroextensions typically provide timing and control constructs that are not found in standard C.
- *Neuron® C.* An enhancement to the standard C with extensions for event handling, network communications, and hardware I/O. It is intended for supporting LonWorks (a fieldbus standard for control networking) applications for Neuron® processors and corresponding smart transceivers, and is used widely within the building automation community.
- *Real-Time C++.* A generic name for one of several object-class libraries specifically developed for C++. These libraries augment standard C++ to provide an increased level of timing and control.

Furthermore, there are numerous other real-time languages and corresponding operating environments, such as Anima, DROL, Erlang, Esterel, Hume, JOVIAL, LUSTRE, Maruti, RLUCID, RSPL, and Timber. Some of these are used for highly specialized applications or in research only.

4.6 AUTOMATIC CODE GENERATION

At the beginning of the embedded-systems era, it was quickly recognized that the productivity of average assembly language programmers was poor, and the availability of skilled programmers continued to be limited. As we know, this situation was largely relieved by adopting high-level languages (see Fig. 4.1); the productivity of programmers improved remarkably, and it was much easier and less time-consuming to become a skilled high-level-language programmer than a skilled assembly-language programmer. Of course, there were also other reasons behind that major transition as discussed earlier in this chapter.

For many years, there was a relevant concern within the developers of hard real-time systems that high-level-language compilers produced less efficient code than would have been possible to create manually with assembly language. Today, there are generally far fewer efficiency concerns, as modern compilers are able to perform truly effective code optimization automatically; this will be outlined shortly in Section 4.7.

The efficiency of compilers allows us to increase the level of abstraction and move from programmer-generated code to automatically generated code—or moving from the solution space toward the higher-level problem space (also referred to as "minimizing the intellectual distance"). Since the productivity of programmers is not improving at the same rate as the size of most applications code is growing, there is considerable pressure toward automatic code generation. At the same time, the required time-to-market of typical products is decreasing, but a significant proportion of embedded-software projects are completed behind the schedule. Furthermore, the general availability of experienced real-time programmers is not adequate, because of the great expansion of the embedded-systems field. Clearly, something needs to be changed to keep up with the growing technological opportunities.

4.6.1 Toward Production-Quality Code

Automatic code generation has been a dream of software engineers and project managers for decades. In fact, the original Fortran language was described as an "automatic program generator" (Backus et al., 1957). In this context, an automatic code generator is assumed to produce high-level language code (seldom assembly language) directly from a system specification in some form without programmer's intervention. Presently, however, it is common practice for programmers to improve automatically generated high-level language code manually before it is compiled.

By the early 1980s, some pioneering organizations were developing "automatic code generators" to speed-up and increase the reliability of the time-consuming and error-prone process of assembly coding. These efforts proved successful, for instance, in the field of automotive control, where a special-purpose, proprietary application language was used for specifying the control software at a high level and then generating assembly code (Srodawa et al., 1985).

During the past few decades, numerous organizations have adopted automatic code generators for limited use in parallel with programmer-generated code. Certain rigorously specifiable parts of real-time software, such as *finite state machines* and various *numerical algorithms*, are automatically composed from system specifications to some high-level programming language, such as Ada (Alonso et al., 2007) or Java (Hagge and Wagner, 2004). However, most of the *production-quality* code is still created manually. Figure 4.4 illustrates the ongoing evolution path from programmer-generated code to automatically generated code. While the ultimate destination of Figure 4.4c is somewhat hazy, the *hybrid* code generation approach is the present and growing reality in many real-time software projects and corresponding organizations. Nonetheless, automatic code generation is typically used just for small portions of code and relatively simple applications in well-understood domains. The

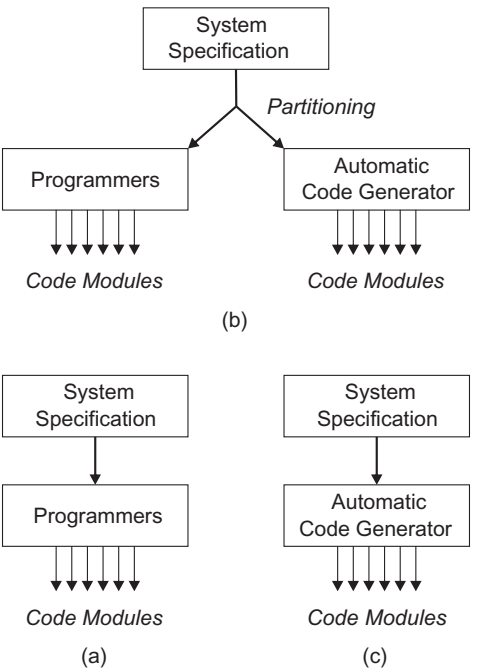


Figure 4.4. A three-step evolution path, (a) → (b) → (c), from programmer-generated code to automatically generated code.

hybrid approach of Figure 4.4b is analogous to the classical two-culture situation, where coexistence of complementary cultures would eventually benefit the entire community.

4.6.2 Remaining Challenges

There are two principal challenges that need to be tackled before automatic code generation could become the dominant approach for generating production-quality code for real-time systems:

1. How to create rigorous specifications of *complex* and *heterogeneous* systems efficiently?
2. How to improve the execution speed and memory usage of automatically generated code?

The first challenge is related to the requirements-engineering methodologies to be discussed in Chapter 5, and the second one resembles the past inefficiency concerns of high-level language compilers. Both of these areas need a lot of research and development effort, because designing an efficient code generator even for a traditional compiler is one of the most difficult parts of compiler design—both practically and theoretically. And the abstract level of system specifications used as an input to the automatic code generator makes the problem much harder. To sum up, can the programmer's vast knowledge and experience ever be captured by (artificial intelligence-based) automatic code generation?

Maclay sees the issue of automatic code generation connected to the aims of software reuse, as they both tend to minimize the amount of work performed by software engineers in a real-time system project (Maclay, 2000). He points out that automatic code generation is particularly useful in *creating prototypes* rapidly, and thus accelerating the algorithm-centered innovation process of novel embedded systems. However, to be acceptable for demanding applications, such as automotive control, the automatically generated code should have less than ~10% efficiency penalty compared with manually created code. Such a reasonable penalty could be compensated by a slightly more efficient processor.

Glass, on the other hand, argues that *large-scale* automatic code generation is "extremely unlikely to happen," because the generator would have to know enough about:

- The application domain to translate the problem specification into a high-level design.
- The application and implementation domains to translate this high-level design into a detailed design.
- Further about the implementation domain to translate the detailed design into the actual code (Glass, 1996).

Although these pragmatic arguments were already presented in the mid-nineties, they are still valid since no breakthroughs on automatic code generation have taken place. Nevertheless, automatic code generation and software reuse are key techniques in responding to the increasing complexity of real-time systems.

4.7 COMPILER OPTIMIZATIONS OF CODE

For every piece of source code, there exist infinitely many object codes that implement the same computations, in the sense that the codes produce the same outputs when presented with the same inputs. Some of these object codes may be faster while others may require less memory; this motivates well the topic of the present section. Aho and Ullman state in their classic book (Aho and Ullman, 1977) that it is theoretically impossible for a compiler to produce the best possible object code for every source code under any reasonable cost function. Hence, a more appropriate term for code optimization would simply be "code improvement." Long tradition has provided us with the somewhat overstated term "code optimization," though.

When beginning to use a new compiler, it is important to experiment with it to learn how it handles certain high-level language constructs, such as `case` statements versus nested `if-then-else` statements, integer versus character variables, and so on. Therefore, a set of relevant test cases should be prepared for the high-level language in question to expose the intricacies of the compiler. No matter which programming language you use in an embedded real-time application, make sure that you know both the language and your compiler thoroughly.

Moreover, many of the techniques used in code optimization underscore the fact that in any arithmetic expression, there is no substitute for a sound mathematical technique. Hence, it is beneficial to reformulate any algorithm or expression to eliminate time-consuming function calls, such as those that compute exponentials, square roots, or transcendental functions, where possible, to improve real-time performance.

Most of the code optimization techniques used by compilers can be exploited to reduce response times. Often these strategies are employed invisibly by the compiler, or can be turned on or off with compiler directives or switches. Furthermore, if a particular strategy is not being used by the compiler itself, it could be implemented manually at the code level instead. Nonetheless, it should be remembered that optimization efforts should be carried out, in general, solely if there is a concrete *demand* for such optimizations. Optimization for the sake of optimization is just wasting resources and creating unnecessary expenses in a software project.

Consider some commonly used code-optimization techniques and their explicit impact on real-time performance. These techniques include:

- Use of arithmetic identities
- Reduction in strength
- Common subexpression elimination
- Use of intrinsic functions
- Constant folding
- Loop invariant removal
- Loop induction elimination
- Use of registers and caches
- Dead-code removal
- Flow-of-control optimization
- Constant propagation
- Dead store elimination
- Dead variable elimination
- Short-circuit Boolean code
- Loop unrolling
- Loop jamming
- Cross-branch elimination

Many of these techniques are facilitated through the use of so-called "peephole" optimization. In peephole optimization, a small window or peephole of machine code is compared against known patterns that yield specific optimization opportunities. These types of code optimizers are fairly straightforward to implement and allow for multiple optimization passes to be performed.

4.7.1 Standard Optimization Techniques

Good compilers *use arithmetic identities* to eliminate useless code. For example, multiplication by the constant "1" or addition by the constant "0" should naturally be eliminated from executable code, although the common use of symbolic constants can obscure these situations.

Reduction in strength refers to the use of the fastest machine-language instructions possible to accomplish a given operation. For instance, when optimizing for speed, some compilers will replace multiplication of an integer by another integer that is a power of two by a series of shift operations. Shift instructions are faster than integer multiplication in certain CPU environments.

In some compilers, character variables are rarely loaded in registers, whereas integer variables are. It is assumed that arithmetic operations involving integers will take place, whereas those involving characters are more unlikely. Care should therefore be taken in deciding whether a particular variable should be defined as a character or an integer.

Furthermore, it is well known that division instructions typically take longer to execute than multiplication instructions. Hence, it may be better to multiply

by the reciprocal of a number than to divide by that number. For example, `x*0.5` would likely be faster than `x/2.0`. Many compilers will not do this replacement automatically.

Repeated calculations of the same subexpression in two different expressions should be avoided. For instance, the following C program fragment:

```
x=6+a*b;
y=a*b+z;
```

could be replaced with:

```
t=a*b;
x=6+t;
y=t+z;
```

thus eliminating the other multiplication. This can result in significant savings if `a` and `b` are floating-point numbers and the code exists in a tight loop.

When possible, *use intrinsic functions* rather than ordinary functions. Intrinsic functions are simply macros where the actual function call is replaced by inline code during compilation. This improves real-time performance because the need to pass parameters, create space for local variables, and eventually release that space is eliminated.

Most compilers perform *constant folding*, but this should not be assumed when beginning to use a new compiler. As an example, the expression:

```
x=2.0*x*4.0;
```

would be optimized by folding `2.0*4.0` into `8.0`. However, performing this operation manually leads to code that is easier to debug. And although the original expression may be more descriptive, a comment can be provided to explain the optimized one.

For example, if the program uses $\pi/2$, it could be precomputed during the initialization phase and stored as a constant named, for example, `pi_div_2`. This will typically save one floating point load and one floating point divide instruction—potentially several microseconds. In a 5-ms real-time cycle, this alone could lead to time-loading savings of ~0.1%. Incidentally, using this strategy illustrates the common inverse relationship between execution time and memory utilization: code execution time has been reduced, but extra memory is needed to store the pre-computed constant.

Most compilers will move such computations outside loops that do not need to be performed within the loop, a process called *loop invariant removal*. For instance, consider the following code fragment in C:

```
x=100;
while (x>0)
 x=x-y+z;
```

It can be replaced by:

```
x=100;
t=y+z;
while (x>0)
 x=x-t;
```

This moves an addition outside the loop, but again requires more memory.

An integer variable `i` is called an *induction variable of a loop* if it is incremented or decremented by some constant on each cycle of the loop. A common situation is one in which the induction variable is `i` and another variable, `j`, which is a linear function of `i`, is used to offset into some array. Often `i` is used solely for a test of loop termination. In such case, variable `i` can be eliminated by replacing its test for one on `j` instead. For example, consider the following C program fragment:

```
for (i=1;i<=10;i++)
 a[i+1]=1;
```

An optimized version is:

```
for (j=2;j<=11;j++)
 a[j]=1;
```

eliminating the extra addition within the loop.

When programming in assembly language or when using languages that support register-type variables, such as C, it is usually advantageous to perform calculations using *work registers*. Typically, register-to-register operations are faster than register-to-memory ones. Thus, if certain variables are used frequently within a module, and if enough registers are available, the compiler should be forced to generate register-direct instructions, if possible.

If the CPU architecture supports *memory caching*, then it may be possible to force frequently used variables into the cache at the language level. Although most optimizing compilers will cache variables when possible, the nature of the source-level code affects the compiler's abilities.

One of the easiest methods for decreasing memory utilization is to *remove dead or unreachable code*—that is, code that can never be reached in the normal flow-of-control. Such code might be debug instructions that are executed only if a debug flag is set, or some redundant initialization instructions. For instance, consider the following C program fragment:

```
if (debug)
 {
 ...
 }
```

In a microcontroller environment, the test of the variable `debug` may take several microseconds, time that is consumed regardless of whether or not the code is in debug mode. Therefore, debug code should preferably be implemented using the conditional compile facilities available with most compilers. Thus, replace the previous fragment with:

```
#ifdef DEBUG
 {
 ...
 }
#endif
```

Here, `#ifdef` is a compiler directive that will include the code between it and the first `#endif` *only* if the symbolic constant `DEBUG` is so defined. Dead code removal may increases program reliability as well.

In *flow-of-control optimization*, unnecessary branch-to-branch instructions are replaced by a single-branch instruction. The following pseudocode illustrates such a situation:

```
          goto label_1;
label_0: y=1;
label_1: goto label_2;
```

It can be replaced by:

```
          goto label_2;
label_0: y=1;
label_1: goto label_2;
```

While such code is not normally generated by skilled programmers, it might result from an automatic code generation or language-to-language translation process and escape unnoticed.

Certain variable-assignment expressions can be changed to *constant assignments*, thereby permitting registerization opportunities or the use of faster immediate addressing mode. In C language, the following code could appear as the result of an automated translation process:

```
x=100;
y=x;
```

The corresponding assembly language code generated by a *nonoptimizing* compiler might look like:

```
LOAD  R1,100  ; Load constant 100 to work register R1.
STORE &x,R1   ; Store the content of R1 to memory location x.
```

```
LOAD R1,&x  ; Load the content of memory location x to R1.
STORE &y,R1 ; Store the content of R1 to memory location y.
```

This can be replaced by:

```
x=100;
y=100;
```

leading to associated assembly-language output:

```
LOAD R1,100 ; Load constant 100 to work register R1.
STORE &x,R1 ; Store the content of R1 to memory location x.
STORE &y,R1 ; Store the content of R1 to memory location y.
```

Variables that contain the *same value within a short segment of code* can be combined into a single temporary variable. For example,

```
t=y+z;
x=func(t);
```

Although many compilers might generate an implicit temporary location for `y+z`, this cannot always be relied on. Replacing the code in question with the following:

```
x=func(y+z);
```

forces the generation of a temporary location and eliminates the need for the local variable, `t`.

A variable is said to be alive at a point in a program if its value can be used subsequently; otherwise it is *dead and subject to removal*. The following code illustrates that `z` is a dead variable:

```
x=y+z;
x=y;
```

After removal of `z`, what is left is:

```
x=y;
```

While this example is trivial, again, it could arise as a result of careless coding or an automated code generation or translation process.

The test of *compound Boolean expressions* can be optimized by testing each subexpression separately. Consider the following:

```
if ((x>0) && (y>0))
 z=1;
```

It could be replaced by:

```
if (x>0)
 if (y>0)
  z=1;
```

In many compilers, the code generated by the second fragment will be superior to the first. ANSI-C, however, executes `if (expression)` constructs sequentially inside the `( )` and drops out at the first `FALSE` condition. That is, it will automatically short-circuit Boolean code.

Loop unrolling duplicates instructions executed in a loop in order to reduce the number of operations, and hence the loop overhead incurred. This technique is used frequently by programmers when coding time-critical signal-processing algorithms. In the exaggerated case, the entire loop is replaced by inline code. For instance,

```
for (i=1;i<=6;i++)
 a[i]=a[i]*8;
```

is replaced by:

```
a[1]=a[1]*8;
a[2]=a[2]*8;
a[3]=a[3]*8;
a[4]=a[4]*8;
a[5]=a[5]*8;
a[6]=a[6]*8;
```

Loop jamming or loop fusion is a technique for combining two similar loops into one, thus reducing loop overhead by a factor of two. For example, the following C code:

```
for (i=1;i<=100;i++)
 x[i]=y[i]*8;
for (i=1;i<=100;i++)
 z[i]=x[i]*y[i];
```

can be effectively replaced by:

```
for (i=1;i<=100;i++)
 {
 x[i]=y[i]*8;
 z[i]=x[i]*y[i];
 }
```

If the same code appears in more than one case in a `case` or `switch` statement, then it is better to combine such cases into one. This *eliminates an additional branch or cross branch*. For example, the following code:

```
switch (x)
 {
 case 0: x=x+1;
         break;
 case 1: x=x*2;
         break;
 case 2: x=x+1;
         break;
 case 3: x=2;
         break;
 }
```

can be replaced by:

```
switch (x)
 {
 case 0:
 case 2: x=x+1;
         break;
 case 1: x=x*2;
         break;
 case 3: x=2;
         break;
 }
```

4.7.2 Additional Optimization Considerations

A sampling of supplementary optimization considerations follows next (Jain, 1991). Note that in most cases, these techniques will optimize the average case, not necessarily the worst case.

- *Arrange entries in a table* so that the most frequently sought values are the first to be compared.
- *Replace threshold tests on monotone functions* (continuously decreasing or increasing) *by tests on their parameters*, thereby avoiding evaluation of the function itself. For instance, if `exp(x)` is a function computing e^x, then instead of using:

  ```
  if (exp(x) < exp(y)) then ...
  ```

  use:

  ```
  if (x < y) then ...
  ```

which will save two evaluations of the costly function `exp()`.

- *Link the most frequently used procedures together* to maximize the locality of reference (applies only in cached or paging systems).
- Store procedures in memory in sequence so that calling and called procedures will be loaded together to increase the locality of reference. Again, this only applies in cached or paging systems.
- *Store redundant data elements close to each other* to increase the locality of reference (applies only in cached or paging systems).

Even though many of the optimization techniques discussed above can be and have been automated, some compilers only perform one optimization pass, overlooking opportunities that are not revealed until after at least a single pass. Hence, manual optimization may provide additional execution-time savings. To see the *cumulative effects of multiple-pass optimization*, consider the following example.

Example: Multiple-Pass Optimization

Begin with the nonoptimized C-code fragment:

```
for (j=1;j<=3;j++)
 {
 a[j]=0;
 a[j]=a[j]+2*x;
 }
for (k=1;k<=3;k++)
 b[k]=b[k]+a[k]+2*k*k;
```

Pass 1: First the code will be optimized by loop jamming, loop invariant removal, and removal of extraneous code (in this case the initialization of `a[j]`). The resultant code is:

```
t=2*x;
for (j=1;j<=3;j++)
 {
 a[j]=t;
 b[j]=b[j]+a[j]+2*j*j;
 }
```

Pass 2: Loop unrolling yields:

```
t=2*x;
a[1]=t;
```

```
b[1]=b[1]+a[1]+2*1*1;
a[2]=t;
b[2]=b[2]+a[2]+2*2*2;
a[3]=t;
b[3]=b[3]+a[3]+2*3*3;
```

Pass 3: After constant folding, the code is:

```
t=2*x;
a[1]=t;
b[1]=b[1]+a[1]+2;
a[2]=t;
b[2]=b[2]+a[2]+8;
a[3]=t;
b[3]=b[3]+a[3]+18;
```

Pass 4: Reduction in strength (assuming that multiplication is slower than addition) and noticing that `a[1]=a[2]=a[3]=t` (thus the content of `t` should be kept in a work register) lead to the final code:

```
t=x+x;
a[1]=t;
a[2]=t;
a[3]=t;
b[1]=b[1]+t+2;
b[2]=b[2]+t+8;
b[3]=b[3]+t+18;
```

The original code involved nine additions and nine multiplications, numerous data movement instructions, and loop overhead. The optimized code requires only seven additions (22% reduction), no multiplications (100% reduction), less data movement, and no loop overhead. Hence, the improvement is significant. It is very unlikely that any compiler would have been able to carry out such an effective optimization automatically.

As we saw above, it is highly beneficial to know the optimization techniques used by compilers when coding real-time software for time-critical applications. Understanding the explicit mapping between high-level language source and assembly language translation for a particular compiler is essential in generating code that is optimal in either execution time or memory utilization viewpoints. The easiest and most reliable way to learn about any compiler is to run a series of tests on specific language constructs. For example, in many compilers, the `case` statement is efficient only if more than three cases are to be compared, otherwise nested `if` statements should be used. Sometimes, the code generated for a `case` statement can be quite convoluted, for instance,

containing a branch through a register, offset by a table value. This process can be time-consuming.

As mentioned earlier, procedure calls are costly in terms of passing of parameters via the stack. Hence, the software engineer should determine whether the compiler passes the parameters by byte or by word.

While modern compilers do provide effective optimization of the assembly language code output so as to, in many cases, make the decisions just discussed, it is important to discover what that optimization is specifically doing to produce the resultant code. For instance, compiler output can be affected by optimization for speed, memory and register usage, branches, and so on, which can sometimes lead to inefficient code, timing problems, or even critical regions. Thus, real-time systems engineers should preferably be masters of their compilers. That is, at all times, the engineer should know what assembly language code will be output for a given high-level language instruction. A thorough understanding of a compiler can only be accomplished by developing a set of test cases to exercise it. The conclusions suggested by these tests can be included in the set of coding standards to foster improved use of the language, and, ultimately, improved real-time performance (Hatton, 1995).

Finally, although modern compilers usually perform effective code optimization, that might not yet be the case when a new CPU architecture is introduced together with a compiler having a rather low and still unstable version number.

Vignette: Early Compiler Version

This anecdote (reported by one of Ovaska's students who wants to stay anonymous) shows that it should not be assumed that a *new* compiler can effectively use all the advanced features of a sophisticated CPU architecture. Some years ago, a digital signal processor with floating-point support was launched with an early version of the ANSI-C compiler. In a research project, a fairly complex signal-processing algorithm was to be implemented in the new processor environment. The single software designer had two competing assignments: (1) use assembly language and do everything you can to minimize the execution time of the real-time algorithm; (2) use C language only and pay special attention to the clarity and understandability of the code. After developing and evaluating those two codes, the following conclusions were made: the assembly program was 53% faster than the C program; the assembly code used 45% less program memory and 63% less data memory than the C code. Hence, the differences were significant. But what were the main reasons behind the remarkable speed improvement when assembly language was used? The C compiler did not use the efficient circular addressing mode for implementing delay lines, but a `for` loop instead. In addition, the compiler did not utilize the parallel instructions of the processor effectively. The extra data memory locations were used by the C program at the initialization stage. It took about 18 hours to write

and test the assembly code, while the C code was completed in less than 6 hours. Obviously, recent versions of that C compiler perform much better than this early one—but you cannot know such things without running appropriate tests yourself.

Most of this code optimization discussion applies particularly to time-critical embedded systems, while soft real-time systems are typically programmed without much concern on such low-level issues—the main emphasis being on productivity of programmers, as well as maintainability and reusability of code, instead.

4.8 SUMMARY

Programming languages continue to have a vital role in the development of real-time systems, because they form an explicit interface between software engineers and real-time hardware. There is a clear trend related to the volume of code in embedded applications: *new products are going to have considerably more code than their predecessors*. This increase is mainly due to enhancements in functionality and the use of more sophisticated computational algorithms.

The hardware community has responded to these demands by spatially and temporally distributed system architectures, advanced communications networks, CPUs with higher instruction throughput, and larger memories. These are needed for running complex signal processing and supervisory control algorithms, intelligent fault prognosis and self-diagnostics functions, model-based virtual sensors replacing physical ones, more comprehensive user- and system-level interfaces, and so forth.

How is that global trend affecting the use of programming languages and the writing of real-time software? Well, the ongoing determined transition from procedural languages to object-oriented languages is one response to the growing amount of software found in typical embedded applications. Object-oriented languages, such as Ada, C++, C#, and Java (or Real-Time Java), provide basic means for improving the productivity of programmers, as well as maintainability and reusability of developed code.

Software reuse, which is seen as an opportunity to reduce the amount of redundant coding work in software projects, has also a flip side: excessive reuse of existing code may even limit the innovativeness of new products. In addition, there is a risk of propagating bad code. Hence, code reuse should be focused on naturally time-invariant and well-proven modules that exist in a specific application from generation to generation. Typical examples include local control algorithms, reference-signal generators, handling of analog and digital I/O, as well as standard fieldbus interfaces.

From the practicing engineer's viewpoint, automatic code generation can still be considered as only an emerging technology—even after several decades of evolution. Nonetheless, the need for automatic code generators is now

greater than ever, because of the growing code size in real-time systems. It is, however, unlikely that automatic code generators will become a mainstream tool for practitioners in the foreseeable future. But they will certainly have steadily increasing use in coding such routine structures as finite state machines and certain numerical algorithms that do not require the problem-solving ability of programmers.

While productivity, maintainability, and reusability are truly important factors, they do not matter if the real-time requirements of a system are not met. It is challenging to run sophisticated algorithms at high sampling rates in a cost-effective hardware platform. Therefore, in certain application domains, special real-time programming languages need to be used instead of the general purpose ones. Such tailored languages lead to highly predictable real-time behavior and minimal language-originated overhead. Furthermore, it is still justifiable to use procedural languages, like the C language, for coding smaller and time-critical applications, or even use assembly language in particular rare occasions.

The ultimate desire of real-time programmers is a standardized programming language and an associated compiler with a *high level of abstraction*, *strict real-time predictability*, and *effectively optimized object-code generation*. In the following chapter, we will discuss requirements-engineering methodologies that have an obvious link to the increasing level of abstraction, which appears to be a central issue when improving productivity of software engineers.

Finally, it may be time to reconsider the pioneering but largely forgotten work of Weinberg, where he proposed a new field of study: "computer programming as a human activity, or, in short, the psychology of computer programming" (Weinberg, 1998). Although that field never became a major one, it could offer complementary ideas for the continuing struggle to improve the productivity of real-time programmers.

4.9 EXERCISES

4.1. What are the reasons why the once-popular PL/I-derivative languages, such as Intel's PL/M, Motorola's MPL and Zilog's PL/Z, practically disappeared by the late 1980s (see Fig. 4.1)?

4.2. It can be argued that in some cases there exists an apparent conflict between good software engineering practices and real-time performance. Consider the relative merits of recursive program design versus interactive techniques, and the use of global variables versus parameter lists. Using these topics and an appropriate programming language for examples, compare and contrast real-time performance versus good software engineering practices as you understand them.

4.3. What programming restrictions should be used in a programming language to permit straightforward analysis of real-time applications?

4.4. Write a set of coding standards for use with any of the real-time applications introduced in Chapter 1 for the programming language of your choice. Document the rationale for each provision of the coding standard.

4.5. Why is it very important to cite the reference for any computational algorithm that is used in a real-time program in the program's annotation?

4.6. In a procedural language of your choice, develop an abstract data type called "image" with associated functions. Be sure to follow the principle of information hiding. Make any assumptions that you need to about the properties of the images.

4.7. In the object-oriented language of your choice, design and code an "image" class that could be useful across a wide range of projects. Be sure to follow the best principles of object-oriented design.

4.8. How can misuse or misunderstanding of a software technology impede a software project? For instance, writing structured C code instead of classes in C++, or reinventing a tool for each project instead of using a standard one.

4.9. Java has been compared with Ada 95 in terms of "hype" and "unification"—defend or refute the arguments for this comparison.

4.10. By using the five metrics of Cardelli, compare the fitness of C and C++ languages for real-time programming; use pentacle diagrams (see Fig. 4.2) for visualizing your justified comparison.

4.11. Are there any language features that are exclusive to C/C++? Do these features provide specific advantage or disadvantage in embedded environments?

4.12. You are hired to define a set of principal requirements for a new real-time programming language for embedded control applications. What are the most important requirements that your definition would contain? Justify your answer.

4.13. What compiler options are available in your favorite C compiler and what do they specifically do?

4.14. Develop a set of tests to exercise a compiler to determine the best use of the language in a real-time processing environment. For example, your tests should determine such things as when to use `case` statements versus nested `if-then-else` statements; when to use integers versus Boolean variables for conditional branching; whether to use `while` or `for` loops, and when; and so on.

4.15. Use standard compiler-optimization methods and multiple optimization phases to optimize the following C code by hand:

```
#define UNIT 1
#define FULL 1
void main(void)
{
 int a,b;
 a=FULL;
 b=a;
 if ((a==FULL) && (b==FULL))
  {
  if (debug)
   printf("a=%d b=%d",a,b);
  a=(b*UNIT)/2;
  a=2.0*a*4;
  b=b*sqrt(a);
  }
}
```

REFERENCES

A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1977.

D. Alonso, C. Vicente-Cicote, P. Sánchez, B. Álvarez, and F. Losilla, "Automatic Ada code generation using a model-driven engineering approach," *Lecture Notes in Computer Science*, 4498, pp. 168–179, 2007.

J. W. Backus et al., "The FORTRAN automatic coding system," *Proceedings of the Western Joint Computer Conference*, Los Angeles, CA, 1957, pp. 188–198.

G. Bollella and J. Gosling, "The real-time specification for Java," *IEEE Computer*, 33(6), pp. 47–54, 2000.

A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java, and C/Real-Time POSIX*, 4th Edition. Harlow, UK: Pearson Education Limited, 2009.

L. Cardelli, "Bad engineering properties of object-oriented languages," *ACM Computing Surveys*, 28(4), pp. 150–158, 1996.

P. Dibble and A. Wellings, "JSR-282 status report," *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, Madrid, Spain, 2009, pp. 179–182.

B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley, 2003.

M. W. El-Kharashi and F. Elguibaly, "Java microprocessors: Computer architecture implications," *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, Canada, 1997, pp. 277–280.

R. L. Glass, "Some thoughts on automatic code generation," *ACM SIGMIS Database*, 27(2), pp. 16–18, 1996.

N. Hagge and B. Wagner, "Mapping reusable control components to Java language constructs," *Proceedings of the 2nd IEEE International Conference on Industrial Informatics*, Berlin, Germany, 2004, pp. 108–113.

L. Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. Maidenhead, UK: McGraw-Hill, 1995.

G. Hedin, L. Bendix, and B. Magnusson, "Introducing software engineering by means of Extreme Programming," *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, 2003, pp. 586–593.

R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York: John Wiley & Sons, 1991.

P. A. Laplante, *Software Engineering for Image Processing*. Boca Raton, FL: CRC Press, 2003.

X. Li and C. Prasad, "Effectively teaching coding standards in programming," *Proceedings of the 6th Conference on Information Technology Education*, Newark, NJ, 2005, pp. 239–244.

M. Lutz and P. A. Laplante, "An analysis of the real-time performance of C#," *IEEE Software*, 20(1), pp. 74–80, 2003.

D. Maclay, "Click and code," *IEE Review*, 46(3), pp. 25–28, 2000.

D. L. Parnas, "On the criteria to be used in decomposing system into modules," *Communications of the ACM*, 15(12), pp. 1053–1058, 1972.

C. Petzold, *Programming Windows*, 5th Edition. Redmond, WA: Microsoft Press, 1999.

D. C. Schmidt, M. Stal, H. Robert, and F. Bushmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons, 2000.

A. C. Shaw, *Real-Time Systems and Software*. New York: John Wiley & Sons, 2001.

B. Sick and S. J. Ovaska, "Fusion of soft and hard computing: Multi-dimensional categorization of computationally intelligent hybrid systems," *Neural Computing & Applications*, 16(2), pp. 125–137, 2007.

R. J. Srodawa, R. E. Gach, and A. Glicker, "Preliminary experience with the automatic generation of production-quality code for the Ford/Intel 8061 microprocessor," *IEEE Transactions on Industrial Electronics*, IE-32(4), pp. 318–326, 1985.

G. M. Weinberg, *The Psychology of Computer Programming: Silver Anniversary Edition*. New York: Dorset House Publishing, 1998.

5

REQUIREMENTS ENGINEERING METHODOLOGIES

Since the embedded-systems era, the emphasis of real-time software development has evolved remarkably from programming toward requirements engineering. In a typical software project today, requirements engineering activities may take an equal amount of effort (in person months) as code development and debugging. Requirements engineering is a core discipline of software and systems engineering that is concerned with determining the objectives, functionality, and constraints of software systems in the problem space, as well as the representation of these aspects in forms amenable to modeling and analysis. The ultimate goal of requirements engineering is to compose a requirements document that is complete, balanced, unambiguous, correct, and easily understandable to both nontechnical customers and software developers. This last goal creates somewhat of a dilemma, as it indicates the duality of purpose of requirements documents: to provide (1) adequate insight for the *customers* to ensure the product under development meets their needs and expectations, and (2) a complete representation of the features and constraints of the software system as a basis for *developers*. In the real-time systems domain, the situation is further complicated by the obvious need to represent exact timing and performance constraints, as well as the more readily elicited requirements.

While programming (or exploration of the solution space) is considered increasingly as a commoditized activity that can be outsourced, requirements

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

engineering is crucial activity of any systems development project, and it should, therefore, be conducted by the development organization—together with an appropriate group of customer representative. Requirements engineering has a principal role in providing real-time software *on-time* and *on-budget* (Laplante, 2009), and it relies heavily on well-defined documentation practices, appropriate methodologies and supporting tools, as well as skills and discipline in using them.

In Section 5.1, an introductory discussion on the requirements engineering process and the different classes of requirements is given to form a foundation for the succeeding sections. The discussion shows that requirements elicitation involves gathering requirements through a diverse collection of techniques. Moreover, there are standardized requirement classes that are applicable to practically all software projects. Formal methods in real-time system specification are discussed with illustrative examples in Section 5.2. These rigorous methods are particularly useful when automatic design and code generation approaches are to be used later in the development project. Section 5.3 is a dual section to the previous one as it provides a pragmatic presentation on leading semiformal methods for system specification. The outcome of the requirements engineering phase, the *requirements document*, is introduced from the structural and contents points of view in Section 5.4. Section 5.5 gives a ruminative summary of this chapter. A diverse collection of revealing exercises on requirements engineering is available in Section 5.6. Lastly, a comprehensive case study on specifying requirements for real-time software is given in Section 5.7. That study of a sophisticated traffic light control system will be continued from the design viewpoint in an appendix of Chapter 6.

Some parts of this chapter have been adapted from Laplante (2003; 2009), which should be considered general references throughout.

5.1 REQUIREMENTS ENGINEERING FOR REAL-TIME SYSTEMS

5.1.1 Requirements Engineering as a Process

A multistep workflow for the requirements engineering phase is shown in Figure 5.1, where specific engineering activities are represented as thin-line rectangles and the documents resulting from those activities are thick-line rectangles. Every requirements engineering process should begin with a preliminary study. This study is an investigation into the motivation for the possible development project and the nature of primary problems to be solved. Such an investigation may consist of stakeholder perspectives and constraints, determination of project scope and feature priorities, as well as some early analysis of the temporal constraints imposed upon the entire real-time system. One of the main deliverables of the requirements engineering process is a feasibility report that may even advise discontinuing development of the planned software product. Usually, however, this will not be the case, and the preliminary study will be followed smoothly by requirements elicitation.

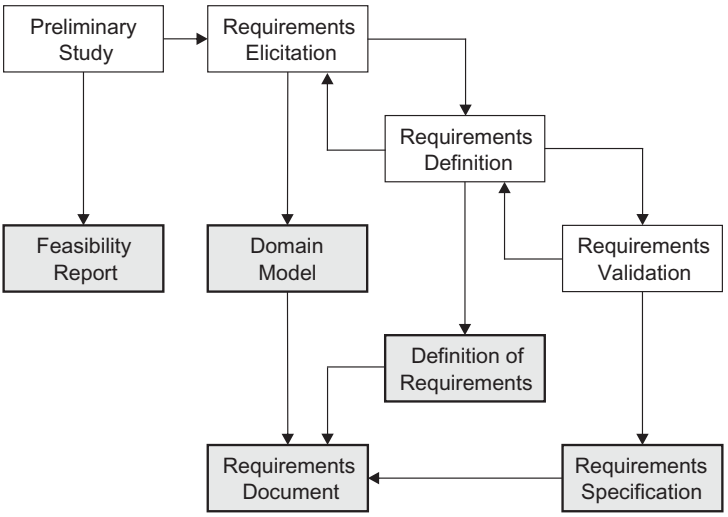


Figure 5.1. The requirements engineering process; adapted from Sommerville (2000).

Requirements elicitation involves gathering various requirements through a variety of techniques that may include stakeholder interviews and questionnaires, focus groups, company- or customer-wide workshops, and prototyping. While requirements can be expressed in several forms ranging from natural language text through mathematical formalisms, it is common for high-level requirements to be outlined in the form of a domain model, that is, a model of the application domain that may include such artifacts as context diagrams, use cases, or entity relationship diagrams—depending on the methodology preferred.

The next stage is requirements definition. It is important to define, precisely enough, each of the captured requirements so that they can be analyzed for completeness, consistency, and correctness in the validation stage. The overall outcome of this process is a requirements document containing a software (or systems) requirements specification (SRS), which is a description of the features, behaviors, and constraints of the final system. Precise software specifications provide an essential basis for analyzing the requirements, validating that they are the stakeholder's true intentions, defining what the designers have to build, and finally verifying that they have done so correctly (Robertson and Robertson, 2005).

5.1.2 Standard Requirement Classes

While there are a number of alternative taxonomies of requirements available, the most established one is the simple functional versus nonfunctional classification. A general software-specification scheme, applicable also for

specifying *real-time* software, is defined by the Institute of Electrical and Electronics Engineers (IEEE) Std 830–1998, Recommended Practice for Software Requirements Specifications (IEEE, 1998). It describes the content and qualities of a solid software requirements document. This widely used standard defines the following six classes of requirements:

C1. Functional: Fundamental actions or features
C2. External Interfaces: Inputs and outputs
C3. Performance: Static and dynamic numerical requirements
C4. Logical Database: Logical requirements for any database information
C5. Design Constraints: Standards and hardware restrictions
C6. Software-System Attributes: Various quantifiable attributes

Here, classes C2 through C6 are considered to be nonfunctional.

Functional requirements include a description of all system inputs and the sequence of operations associated with each particular input set. Either through case-by-case description or some other general form of description (e.g., using universal quantification), the exact sequence of operations and outputs to normal and abnormal situations must be provided for every input possibility. Moreover, abnormal situations might include error handling and recovery, including failure to meet deadlines. In essence, functional requirements describe the complete deterministic behavior of the real-time system. Generally, the functional requirements are partitioned to software and hardware before requirements analysis begins, although a careful trade-off analysis may cause these to shift later in the project lifecycle.

External interface requirements are a description of all inputs and outputs to/from the system including:

- Name of item
- Description of purpose
- Source of input or destination of output
- Valid range, accuracy, and/or tolerance
- Units of measure
- Timing
- Relationships to other inputs/outputs
- Screen formats/organization
- Window formats/organization
- Data formats
- Command formats
- End messages

Performance requirements include both static and dynamic numerical requirements placed on the software or on human interaction with the software as a

whole. For a nonembedded real-time system, static requirements could include the number of simultaneous users to be supported. The dynamic requirements, on the other hand, might include the number of transactions and tasks and the amount of data to be processed within certain time limits under both normal and peak workload conditions. With embedded systems, however, the individual performance requirements may vary greatly from those of software that is nonembedded.

Logical-database requirements include the definitions of tabulated information used by various functions, such as accessing capabilities, data entities and their relationships, data-retention requirements, frequency of use, and integrity constraints.

Design-constraint requirements are related to such imperative issues as standards compliance and hardware limitations.

Lastly, software system attribute requirements include availability, maintainability, portability, reliability, security, and energy usage of the real-time software. It is important to specify the attributes explicitly to make it possible to verify their proper existence objectively. These attributes are often architecture driven.

At this point, it is worth noting that the traditional nomenclature for functional versus nonfunctional requirements is somewhat inaccurate, because the terms "functional" and "nonfunctional" are not necessarily separable in the context of real-time systems. Thus, a more logical classification could be between a behavior that is *observable* via execution (e.g., some response time) and behavior that is *nonobservable* via execution (e.g., maintainability). Such a classification principle has an analogy to the concept of observability in control theory.

5.1.3 Specification of Real-Time Software

There is no single approach for specification of real-time software, but real-time systems engineers typically use a case-specific combination of the following approaches:

- Top-down decomposition or structured analysis
- Object-oriented approaches
- Software description languages or in-house pseudocode
- High-level functional specifications that are not further decomposed
- *Ad hoc* techniques, including natural language, mathematical descriptions, and various models

There are three general classifications of specification techniques: *formal*, *informal*, and *semiformal*. Formal methods have a rigorous, mathematical or logical basis. A representative sampling of these approaches is discussed in the following sections. Any requirements specification technique is informal if it

cannot be completely transliterated into a rigorous mathematical notation and associated rules. Rudimentary informal specifications, such as flowcharting, have little or no underlying mathematical/logical structure, and hence they cannot be *completely* analyzed. In the case of flowcharts, the topological structure is mathematically rigorous—either sequence or branch—but the semantics in the process and decision blocks, typically expressed using natural language, are not. All that can be done with informal specifications is to find counterexamples of where the system fails to meet the requirements or where there are conflicts. This is simply not adequate for most real-time systems, where formal substantiation of performance characteristics of requirements is necessary. Approaches to requirements specification that defy classification as either formal or informal are called semiformal. Semiformal approaches, while not appearing to be fully rigorous, might be. For example, some contend that the unified modeling language (UML) is semiformal, because the statechart is formal while other metamodeling techniques it employs have a pseudomathematical basis. Others contend, however, that UML is not even semiformal, because it has serious holes and inconsistencies—this heavy criticism applies to UML 1.x only. The thoroughly revised UML 2.x contains additional formal components, and there are serious intentions to formalize it even further (Miles and Hamilton, 2006). In any case, UML largely enjoys the benefits of both informal and formal techniques and is extensively used in real-time systems specification and design, because it is supporting the ongoing transition from procedural programming languages to object-oriented ones.

5.2 FORMAL METHODS IN SYSTEM SPECIFICATION

Formal methods contribute significantly to requirements formulation and validation by the use and extension of effective mathematical techniques (Liu, 2010). And this practice is becoming more and more feasible with the increasing availability of supporting tools. These techniques and associated tools employ some combination of abstract algebra, discrete mathematics, λ-calculus, number theory, predicate calculus, programming language semantics, recursive function theory, and so forth. One of the primary benefits of formal methods is that they provide an exact scientific perspective to system specification and software design. Formal requirements offer the opportunity of discovering errors at the earliest phase of development, when the errors can be corrected more easily and at a lower cost. Informal specifications, on the other hand, might not support this goal, because while they can be used to refute a specific requirement by counterexamples, such counterexamples may be difficult to create.

By their nature, specifications for real-time systems usually contain some formalism in the mathematical expression of the interactions with the operating environment or within systems in which they are embedded. While this does not justify the claim that every real-time system specification is fully

formalizable, it does lead to certain optimism that most real-time systems are suitable for, at least, partial formalization.

Nevertheless, formal methods are generally perceived to be difficult to use by even expertly trained engineers, and may be error-prone if used without proper computer-based tools. For these reasons, and because they are often believed to increase early lifecycle costs and even delay projects, formal methods are, unfortunately, too often avoided.

It should be understood, however, that formal methods are not intended to take on an all-encompassing role in real-time software specification and design. Instead, carefully selected techniques may be used in one or two stages of the development process. There are three typical uses for formal methods among software engineers:

1. *Consistency Checking.* This is where the system's behavioral requirements are described using a mathematics-originated notation.
2. *Model Checking.* Finite state machines or their extensions are used to verify whether a given property is satisfied under all conditions.
3. *Theorem Proving.* Here, axioms of system behavior are used to derive a proof that a system will behave in a given way.

In addition, formal methods offer unique opportunities for reusing requirements. Embedded systems are often developed as families of similar products, or as incremental redesigns of existing products. For the first situation, formal methods can help to identify a consistent set of core requirements and abstractions to reduce duplicate engineering effort. For redesigns, having formal specifications for the existing system provides an unambiguous reference for baseline behavior and a convenient way to analyze proposed changes (Bowen and Hinchey, 1995).

Example: Consistency Proof of Requirements

Consider the following excerpt from a software requirements specification:

R1. If interrupt A arrives, then task B stops executing.

R2. Task A begins executing upon arrival of interrupt A.

R3. Either Task A is executing and Task B is not, or Task B is executing and Task A is not, or both are not executing.

These textual requirements can be formalized by rewriting each in terms of their component propositions, namely:

p: Interrupt A arrives.

q: Task B is executing.

r: Task A is executing.

Then rewriting the requirements using these propositions and standard logical connectives yields:

R1. $p \Rightarrow \neg q$

R2. $p \Rightarrow r$

R3. $(r \wedge \neg q) \vee (q \wedge \neg r) \vee (\neg q \wedge \neg r)$

Notice the obvious difficulties in dealing with the articulation of temporal behavior. For instance, in requirement R2, Task A begins executing upon arrival of interrupt A; but for how long does it continue executing? This relationship needs to be clarified using some other methodology.

In any case, the consistency of these requirements can be proved by demonstrating that there is *at least one set of truth values that makes all requirements true simultaneously*. This can be verified explicitly by creating the corresponding truth table (see Table 5.1). Looking at the table, rows 3, 6, 7, and 8 in columns 6, 7, and 8, corresponding to requirements R1, R2, and R3, are all true, and hence this set of requirements is consistent.

Consistency checking, leading to a formal proof, is particularly useful when there is a large number of complicated requirements. If automated tools with a convenient user interface are available to perform the checking process, even large specifications could be consistency checked this way. However, aside from the difficulties in formalizing the notation, finding a set of individual truth values that yield a composite truth value for the set of propositions is, in fact, a Boolean satisfiability problem, which is an NP-complete problem (to be discussed in Chapter 7).

TABLE 5.1. Truth Table Used to Verify the Consistency of the Example Set of Requirements (T = True and F = False)

	1	2	3	4	5	**6**	**7**	**8**
	p	q	r	$\neg q$	$\neg r$	$P \Rightarrow \neg q$	$p \Rightarrow r$	$(r \wedge \neg q) \vee (q \wedge \neg r) \vee (\neg q \wedge \neg r)$
1	T	T	T	F	F	F	T	F
2	T	T	F	F	T	F	F	T
3	T	F	T	T	F	**T**	**T**	**T**
4	T	F	F	T	T	T	F	T
5	F	T	T	F	F	T	T	F
6	F	T	F	F	T	**T**	**T**	**T**
7	F	F	T	T	F	**T**	**T**	**T**
8	F	F	F	T	T	**T**	**T**	**T**

5.2.1 Limitations of Formal Methods

Formal methods have two major limitations that are of special concern to real-time system developers. First, although formalism is often used in pursuit of absolute correctness and safety, it can eventually guarantee neither. And, second, formal techniques do not offer efficient or intuitive ways to reason about alternative architectures or designs.

Correctness and safety are two of the original motivating factors driving adoption of formal methods. Aerospace, automotive, defense, elevator, mass transportation, and nuclear regulations in several countries mandate (or strongly suggest) the use of formal methods for specifying safety-critical subsystems. Furthermore, some academic researchers emphasize the "correctness" properties of particular mathematical approaches, without clarifying that mathematical correctness in one part of the development process might not translate into implemented correctness in the entire system.

Nevertheless, it is the specification that must be produced and proven at this stage—not the software product itself. Formal software specifications need to be converted to a design, and then encoded using some programming language(s). The translation process is subject to the potential pitfalls of any programming effort. For this reason, testing is just as important when using formal requirements engineering methods as when using informal or semiformal ones, though the testing effort can be reduced with the use of formal methods. Formal verification is also subject to many of the same limitations as traditional testing, namely, that testing cannot prove the absence of errors, only their presence.

Notation evolution is a slow but ongoing process in the formal methods community. It can take many years from when a new notation is introduced until it is adopted universally. A major challenge in applying formal methods to real-time embedded systems is choosing an appropriate technique to match the problem at hand. Still, to make formal models truly usable for a wide spectrum of people, requirements documents should also use complementary nonmathematical notations, such as natural language, structured text, or some form of graphical diagrams.

5.2.2 Finite State Machines

The finite state machine (FSM), finite state automaton (FSA), or state-transition diagram (STD) is a formal mathematical model used in the specification and design of real-time software (Wagner et al., 2006). Of those three equivalent terms, we use "finite state machine" throughout this text. Intuitively, finite state machines rely on the fact that many systems can be represented by a fixed number of unique states and certain transitions between them. The system may change its state depending on time (real-time clock) or the occurrence of specific events. Formally, finite state machines can be represented by the five tuple:

$$M = \{S, i, T, \Sigma, \delta\}, \tag{5.1}$$

where S is a finite, nonempty set of states; i is the initial state ($i \in S$); T is the finite set of terminal states ($T \subseteq S$); Σ is a finite alphabet of symbols or events used to mark transitions between states; and δ is a transition function that describes the next state of the FSM given the current state and a symbol from the alphabet. That is, $\delta: S \times \Sigma \rightarrow S$. A finite state machine can be expressed in diagrammatic, matrix, and set-theoretic representations, but we prefer the two first ones in this text. While graphical diagrams are easy to create and understand by engineers, matrices are appropriate inputs for automatic code generators.

Example: Representations of a Practical Finite State Machine

To illustrate the diagrammatic and matrix representations, suppose it is desired to model the door-control subsystem of an elevator controller. This safety-critical subsystem has the following seven states:

Closed: The door is fully closed.

Opening: The door is opening due to an initial open command or a later reopen command.

Open: The door is fully open.

Closing: The door is closing normally.

Nudging: The door is closing at a creeping speed and with a reduced force after several reopenings.

Fault C: The door could not be fully closed due to some failure.

Fault O: The door could not be fully opened due to some failure.

The first five states (Closed, Opening, Open, Closing, and Nudging) are visited regularly in normal operation of the elevator, but the two last ones (Fault C and Fault O) represent serious fault conditions when the elevator must be shut down since the door cannot be either closed or opened due to some (typically) mechanical failure. In those abnormal terminal states, certain fault-recovery procedures are initiated, often leading to a service visit by an elevator technician. It is generally known that broken doors are causing most of the elevator shut-downs.

The door-control subsystem reacts to various events generated by the elevator controller itself, door contacts and safety sensors, push buttons inside the car, as well as different timeout timers. These events are listed below:

CC: Command from the elevator controller to close the door.

OC: Command from the elevator controller to open the door.

DC: Door-closed contact (the door is fully closed).

DO: Door-open contact (the door is fully open).

CB: Door-close button.

OB: Door-open button.

SE: Safety edge to sense a passenger (or some obstacle) between closing door blades.

PC: Photocell(s) to sense a passenger (or some obstacle) between closing door blades.

T1: Timeout to indicate the door could not be closed in a fairly long time due to several reopenings.

T2: Timeout to indicate the door could not be closed in an overly long time due to a likely failure.

T3: Timeout to indicate the door could not be opened in a nominal (plus some margin) time due to a possible failure.

The possible transitions from state to state triggered by specific events are illustrated in Figure 5.2. It is assumed that "Closed" is the initial state.

This FSM can be expressed using the five tuple of Equation 5.1 as follows:

$$S = \{\text{Closed, Opening, Open, Closing, Nudging, Fault C, Fault O}\}$$

$$i = \text{Closed}$$

$$T = \{\text{Fault C, Fault O}\}$$

$$\Sigma = \{\text{CC, OC, DC, DO, CB, OB, SE, PC, T1, T2, T3}\}$$

The transition function, δ, is embodied in the diagram itself, and is convenient to represent with a transition matrix, as shown in Table 5.2.

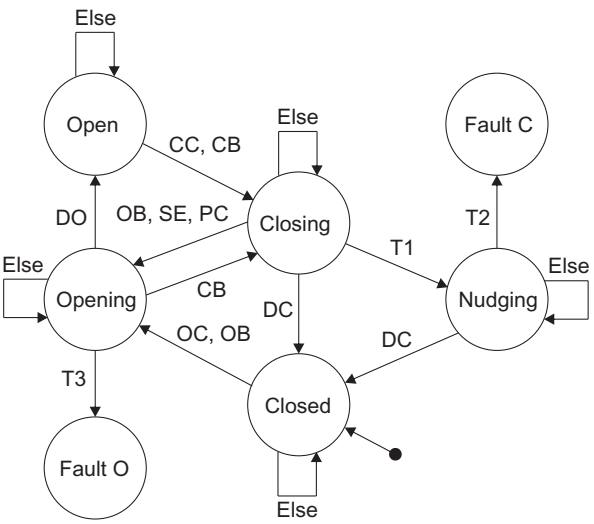


Figure 5.2. A diagrammatic representation of a finite state machine for the elevator door-control subsystem.

TABLE 5.2. Transition Matrix Representation for the Finite State Machine in Figure 5.2

	CC	OC	DC	DO	CB	OB	SE	PC	T1	T2	T3
Closed	Closed	**Opening**	Closed	Closed	Closed	**Opening**	Closed	Closed	Closed	Closed	Closed
Opening	Opening	Opening	Opening	**Open**	**Closing**	Opening	Opening	Opening	Opening	Opening	**Fault O**
Open	**Closing**	Open	Open	Open	**Closing**	Open	Open	Open	Open	Open	Open
Closing	Closing	Closing	**Closed**	Closing	Closing	**Opening**	**Opening**	**Opening**	**Nudging**	Closing	Closing
Nudging	Nudging	Nudging	**Closed**	Nudging	Nudging	Nudging	Nudging	Nudging	Nudging	**Fault C**	Nudging
Fault C	Fault C	Fault C	Fault C	Fault C	Fault C	Fault C	Fault C	Fault C	Fault C	Fault C	Fault C
Fault O	Fault O	Fault O	Fault O	Fault O	Fault O	Fault O	Fault O	Fault O	Fault O	Fault O	Fault O

A finite state machine that does not depict any outputs during state transitions is called a *Moore* machine, where all outputs are dependent on states only. Thus far, we have considered solely Moore machines in this discussion. However, outputs during transitions can be depicted by an extension of the Moore machine called a *Mealy* machine. The Mealy machine can be described accordingly by a six-tuple:

$$M = \{S, i, T, \Sigma, \delta, \Gamma\} \tag{5.2}$$

where the first five elements are the same as for the Moore machine of Equation 5.1, while the sixth element, Γ, represents the set of possible outputs. However, the transition/output function, δ, is here somewhat different from the pure transition function of the Moore FSM, as it describes the next state and the associated output given the current state and an input symbol from the alphabet. Hence, it can be expressed as $\delta: S \times \Sigma \rightarrow S \times \Gamma$. A generic Mealy machine for a system with three states, three inputs, and three outputs is illustrated in Figure 5.3. The corresponding transition matrix is given in Table 5.3. It is commonly known that the number of states required in a Mealy FSM is less than or equal to the number of states in a corresponding Moore machine.

Finite state machines are straightforward to construct, and program code can be easily (or even automatically) generated using matrices to specify the transitions between states. FSMs are also unambiguous, since they can be represented with a formal mathematical description. In addition, concurrency

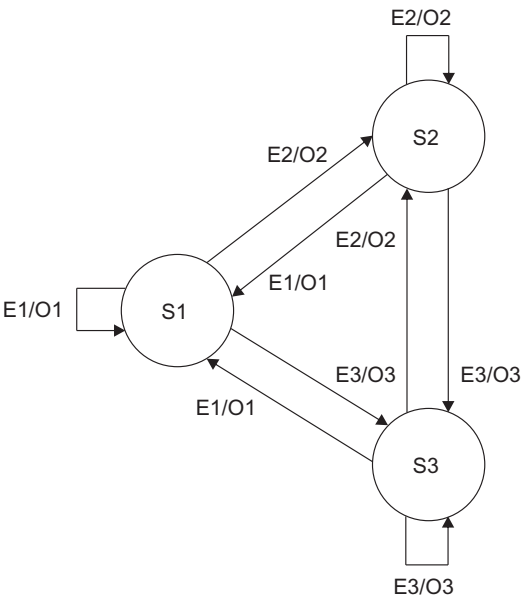


Figure 5.3. A fully connected Mealy FSM with states S1, S2, and S3, inputs E1, E2, and E3, and outputs O1, O2, and O3.

TABLE 5.3. Transition Matrix Representation for the Finite State Machine in Figure 5.3

	E1	E2	E3
S1	S1/O1	S2/O2	S3/O3
S2	S1/O1	S2/O2	S3/O3
S3	S1/O1	S2/O2	S3/O3

in a real-time system can be depicted by using multiple finite state machines. When constructing a finite state machine, special attention should be given to the following concerns (Yourdon, 1989):

- Have you defined all necessary states?
- Can all the states be reached?
- Can all the states, except the terminal ones, be exited?

Because rigorous mathematical techniques for reducing the number of states exist, program code based on FSMs can be formally optimized. Such optimization could even be automated. A rich theory surrounds finite state machines, and this can be exploited in the development of system specifications. On the other hand, a major disadvantage of FSMs is that the internal aspects of modules cannot be depicted. That is, there is no way to indicate how functions (states) can be broken down into subfunctions (substates). In addition, intertask communication between multiple FSMs is difficult to depict. Finally, depending on the particular system and alphabet used, the number of states may sometimes grow very large. Both of these problems, however, can be overcome through the use of statecharts to be introduced shortly. Furthermore, the use of finite state machines in the design of real-time software is discussed in Chapter 6.

5.2.3 Statecharts

Statecharts—or originally Harel's statecharts (Harel, 2009)—have their roots in the avionics industry, and they provide "diagrammatic/visual formalism" for system and software engineers. They combine finite state machines' user-friendliness with data flow diagrams and a feature called broadcast communication, in a way that can depict both synchronous and asynchronous operations. Statecharts can be defined informally as:

$$\text{Statechart} = \text{FSM} + \text{Depth} + \text{Orthogonality} + \text{Broadcast Communication}.$$

Here, FSM is a finite state machine, depth represents hierarchical levels of detail, orthogonality represents the existence of parallel states, and broadcast communication is a method for allowing multiple orthogonal states to react

to the same event. Actually, hierarchy and orthogonality are the two principal ideas behind statecharts, and they can be flexibly compounded. The value of hierarchy and orthogonality is mainly in their *convenience* and *naturalness*. In principle, hierarchy could always be flattened and orthogonality could be removed. However, "convenience" is exactly what engineers appreciate when using any methodology.

The statechart is an extension of a finite state machine, where each state can contain its own FSM that further describes its behavior. The fundamental components of the statechart are introduced below (see also Figures 5.4–5.7):

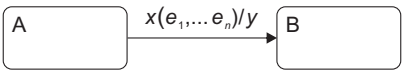


Figure 5.4. Statechart format where A and B are states, x is an event that causes the transition marked by the arrow, y is an optional event triggered by x, and $e_1, \ldots e_n$ are optional conditions qualifying the primary event; adapted from Laplante (2003).

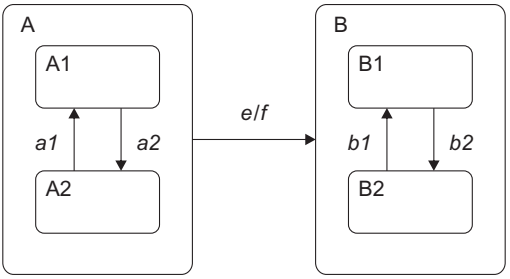


Figure 5.5. A statechart depicting insideness; adapted from Laplante (2003).

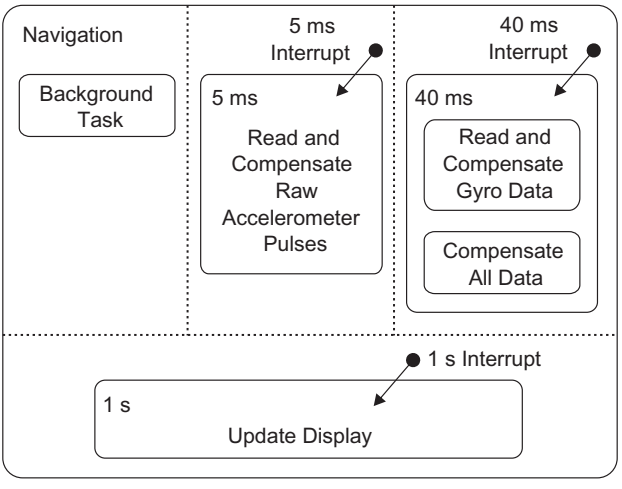


Figure 5.6. Navigation subsystem containing four orthogonal tasks.

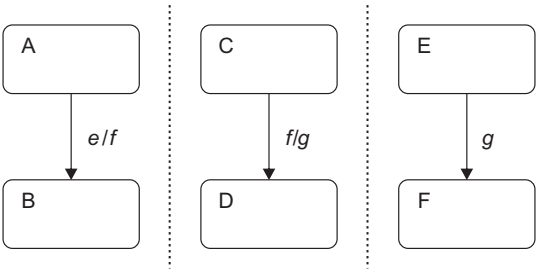


Figure 5.7. A statechart depicting a chain reaction; adapted from Laplante (2003).

- The FSM is represented in the usual way, with capital letters or descriptive phrases used to label the states.
- Depth or hierarchy is represented by the insideness of states.
- Orthogonality is represented by dashed lines separating parallel states (or tasks in a multitasking system).
- Broadcast communications are represented by labeled arrows, similarly as transitions in FSMs.
- Symbols $a, b, \ldots z$ represent events that trigger transitions, in the same way transitions are represented in FSMs.
- Lowercase letters within parentheses stand for conditions that must be true for the associated transition to occur.

A noteworthy characteristic of statecharts is the explicit encouragement of top-down design of modules or suchlike. As an example, for any module (represented like a state in an FSM), increasing detail is depicted as substates internal to it. In Figure 5.5, the system is composed of two primary states, A and B, drawn as rounded rectangles. Each of these, in turn, has been decomposed into substates Al and A2, and B1 and B2, respectively, which might represent individual program modules. Those internal states can also be decomposed, and so forth. To the programmer using some procedural language, each nested substate within a state represents a procedure within another procedure.

Orthogonality depicts concurrency in the system for states that run in isolation, called AND states. Orthogonality is represented by separating the orthogonal components by dashed lines. For instance, if a state S consists of AND components P and Q, S is called the *orthogonal product* of P and Q. If S is entered from the outside without any conditional information, then the states P and Q are entered simultaneously. Communication between the AND states can be achieved conveniently through careful use of global memory, whereas synchronization can be achieved through a feature of statecharts called broadcast communication. Figure 5.6 illustrates a statechart for the aircraft-navigation subsystem discussed earlier in the text containing four orthogonal

tasks and six internal states or substates. This highly condensed visual description of the entire subsystem gives a clear picture of the multitasking functionality. However, it is necessary to understand the underlying algorithms and constraints before it is possible to compose such a statechart.

Broadcast communication is depicted by the transition of orthogonal states based on the same event, and it is a simple way to coordinate the orthogonal statechart components. For example, if the inertial measurement system switches from "standby" to "ready" mode, an event indicated by an interrupt can cause a simultaneous state change in multiple tasks (or even subsystems). Another valuable aspect of broadcast communication is the concept of *chain reaction*; that is, events triggering other events in a sequence. Its implementation follows from the observation that statecharts can be viewed as an extension of Mealy-type finite state machines, and output events can be attached to the triggering event. In contrast with the standard Mealy machine, however, the output is not seen by the outside world; instead, it affects the behavior of an orthogonal component only. For instance, in Figure 5.7 suppose there exists a transition labeled *e*/*f*, and if event *e* occurs, then event *f* is immediately activated. Furthermore, event *f* could trigger another transition, such as *f*/*g*. The length of a chain reaction is the number of transitions triggered by the first event. Such chain reactions are assumed to occur instantaneously, although in practical uniprocessor implementations, this is not possible to be accomplished precisely. In the system of Figure 5.7, a chain reaction of length two will occur when the *e*/*f* transition first occurs.

Statecharts are excellent for representing real-time systems since they can depict concurrency while preserving modularity. In addition, the concept of broadcast communication allows for easy intertask-communication representation. As a real-world example, Figure 5.A5 found in the case study of Section 5.7 illustrates a statechart corresponding to the sophisticated traffic-light control system.

In summary, the statechart combines the best of data flow diagrams and finite state machines. Commercial products allow a practicing engineer to define graphically a real-time system using statecharts, perform comprehensive simulation analysis, and even generate program code automatically. Moreover, statecharts can be used in conjunction with both structured and object-oriented methods. Statecharts are widely used today, because an object-oriented variant has become a standard part of UML (Samek, 2009).

5.2.4 Petri Nets

Petri nets are another class of formal methods used to specify and analyze concurrent operations in real-time systems (Mazzeo et al., 1997). Commercially available Petri net tools can produce executable specifications and are particularly suitable for modeling synchronizations among asynchronous tasks. While Petri nets have a rigorous mathematical basis, they can still be described graphically as interconnections of only two basic entities. A set of circles called

"places" is used to represent data stores or conditions. Rectangular boxes, on the other hand, represent transitions or events. Each *place* (P) and *transition* (T) is labeled with a data count and transition function, respectively, and they are connected by unidirectional arrows. Petri nets are sometimes called Place/Transition nets in reference to the central role of places and transitions. In addition, finite state machines can be interpreted as a subclass of Petri nets, but their expressiveness is obviously weaker.

The initial Petri net graph is labeled with a *marking* given by m_0, which represents the initial data count in all the places. Subsequent markings, m_i, $i \in \{1, 2, 3, \ldots\}$, are results of the *firing* of transitions, where each firing is an atomic operation by its nature. A transition fires if it has as many input data as required for producing an associated output. In Petri nets, the graph topology does not change over time; only the marking or data count of the places do. The modeled real-time systems may also advance nondeterministically (i.e., there is more than one possible next state) as transitions fire. To illustrate the notion of firing, consider the simple Petri net given in Figure 5.8 and the corresponding firing table provided in Table 5.4.

As another example, consider the Petri net of Figure 5.9. Moving from top to bottom and left to right indicates the consecutive stages of firings in the net. Table 5.5 depicts the corresponding firing table. When the number of outgoing arrows is lower than that of incoming arrows, the particular transition is called a *consumer*; and when a transition has more outgoing than incoming arrows, it is a *producer* (Bucci et al., 1995).

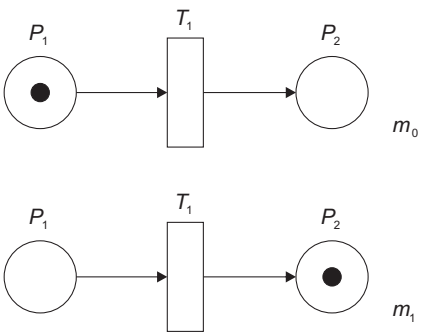


Figure 5.8. A simple Petri net before (m_0) and after (m_1) firing; adapted from Laplante (2003).

TABLE 5.4. Firing Table for the Petri Net Shown in Figure 5.8; Adapted from (Laplante, 2003)

	P_1	P_2
m_0	1	0
m_1	0	1

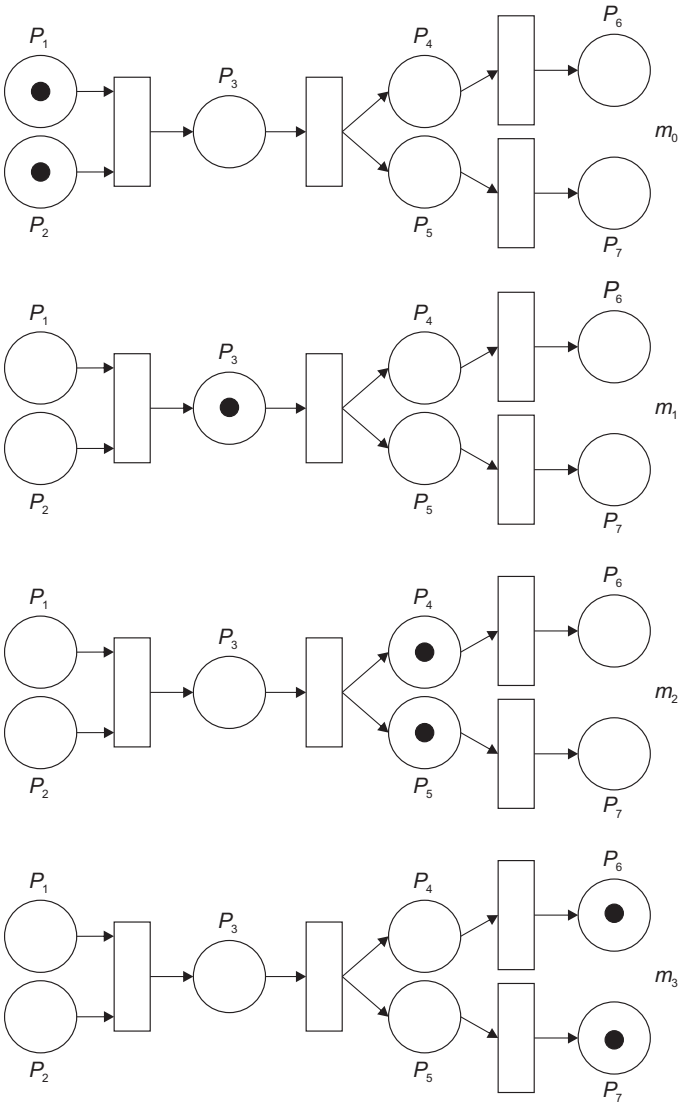


Figure 5.9. Sequential behavior of a Petri net with both "consumer" and "producer" transitions.

TABLE 5.5. Firing Table for the Petri Net Shown in Figure 5.9

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
m_0	1	1	0	0	0	0	0
m_1	0	0	1	0	0	0	0
m_2	0	0	0	1	1	0	0
m_3	0	0	0	0	0	1	1

Petri nets can be used to model (parts of) real-time systems and to search for possible timing conflicts, as well as race conditions. They are excellent for representing distributed and event-driven systems, such as communications protocols and discrete manufacturing systems. As Petri nets are purely mathematical in nature, rigorous techniques for system optimization and program proving can be employed. However, Petri nets can be overkill if the system to be modeled is very simple. Similarly, if the system is highly complex, the overall timing behavior can easily become obscured.

The Petri net is a powerful tool that is frequently used for the analysis of *race-conditions* and for *deadlock* identification. For example, suppose a requirements specification contains a subnet that resembles Figure 5.10. Clearly, it is impossible to tell which of the two transitions (labeled with a question mark) will fire, though in any case, only one of them will fire. Moreover, Petri nets can be used effectively to identify such cycles that indicate a potential deadlock. For instance, suppose a set of requirements can be modeled as in Figure 5.11, which is, in fact, a formal replica of Figure 3.13 involving two parallel tasks and two shared resources. Obviously, this scenario represents an inevitable deadlock. And while it is unlikely that such a conflict situation would be created intentionally, Petri nets can also be used to identify unreachable states, which would be represented by a marking that can never be reached. Petri net analysis by appropriate simulation tools can be used to identify nonobvious

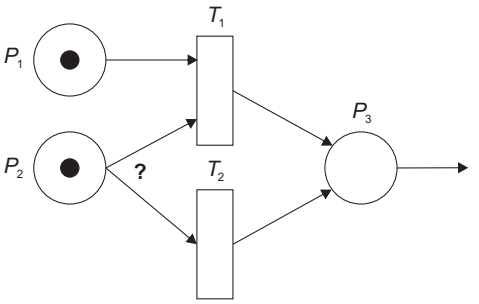


Figure 5.10. Race-condition identification with a Petri net.

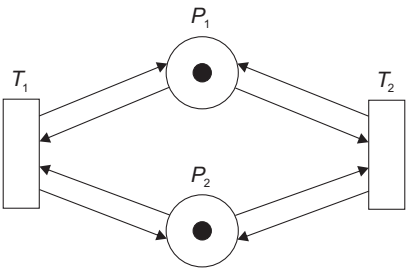


Figure 5.11. Deadlock in Figure 3.13 illustrated using a Petri net.

cycles that appear as subnets in complex diagrams. This kind of situation is briefly discussed in the following vignette by Ovaska.

Vignette: Petri Net Simulation Helps in Identifying Deadlocks

> A communications protocol was developed for the elevator control system of Figure 3.17. The protocol was implemented and tested, and everything seemed to work just fine. However, the critical communications between the group dispatcher and up to eight elevator controllers could enter a deadlocked state sporadically. This situation happened infrequently, once every few days. Hence, it was difficult to identify the reason for such a dramatic failure. For some time, it was assumed the deadlock was hardware originated and caused by electromagnetic interferences. But no evidence to such a hypothetical problem was found. The responsible software engineer was practice oriented and did not have any trust in formal methods for system specification. Nonetheless, two computer science students were invited to create a detailed Petri net model of the communications protocol as their special assignment. They performed systematical simulations with the formal model and soon identified a rare error condition leading to the hunted deadlock. The bug was corrected and the communications protocol worked perfectly since that. After this shaking experience, the engineer removed Petri nets from his subjective "to be avoided list."

The basic Petri net model described in this section is just one of a variety of available models. For example, there are timed Petri nets, which enable synchronization of firings, colored Petri nets, which allow for labeled data to propagate through the net, and timed-colored Petri nets, which embody both features.

5.3 SEMIFORMAL METHODS IN SYSTEM SPECIFICATION

Semiformal methods are used extensively in specifying real-time systems because of their typical versatility—some of those methods are even considered to advance the laborious *system-specification process* itself. In the following discussion, we contemplate two widespread approaches for specifying real-time software: structured analysis and structured design (SA/SD) methods and the unified modeling language (UML). While the use of UML is expanding rapidly with the ongoing transition from procedural programming languages to object-oriented ones in real-time applications (see Figure 4.1), SA/SD still has a solid position with the users of procedural languages in embedded systems. Moreover, the easy-to-learn SA/SD methods are particularly convenient when introducing the topic of system specification to undergraduate students or other newcomers in the field of software engineering.

5.3.1 Structured Analysis and Structured Design

During the past, for over three decades, methods for SA/SD have evolved gradually from the early definitions of De Marco (1978) to the latest extensions of Yourdon (2006), and are used widely in diverse real-time applications throughout the world. One reason behind the exceptional popularity of SA/SD is that those techniques are closely associated with the procedural programming languages with which they co-evolved (such as the C language) and in which countless real-time systems are written. Another reason is obviously the global availability of established SA/SD engineering tools. Although structured methods appear in many forms, the de facto standard is undisputedly Yourdon's *Modern Structured Analysis* (Yourdon, 1989).

Several extensions to the original structured analysis (SA) emerged already in the 1980s to account, for instance, system dynamics and the usage of SA for the specification of embedded systems. Particularly, Ward and Mellor extended data flow diagrams by adding a way to model control flows (such as interrupts), as well as finite state machines (or state-transition diagrams), for defining control processes (Ward and Mellor, 1985). Other real-time extensions include Gomaa's DARTS (design approach for real-time systems) (Gomaa, 1988), for example.

Structured analysis for *real-time systems* is still based on the fundamental notion of the flow of data between successive data transformations, and it provides very little support for identifying concurrency. Hence, depending upon the detail of the analysis phase, there is usually something arbitrary in identifying the appropriate set of processes. This may result in the implementation of unnecessary processes (causing extra scheduling overhead) and the possibility that some process needs concurrency internally (causing additional implementation complexity) (Bucci et al., 1995). To prevent these inefficiencies, it is truly important to use the SA/SD methods *iteratively*—such a multipass approach is supported by the SA/SD engineering tools.

Yourdon's *Modern Structured Analysis* has three complementary models (or viewpoints) to describe a real-time system:

M1. Environmental model
M2. Behavioral model
M3. Implementation model

The elements of each model are shown in Figure 5.12. The environmental model embodies the *analysis* aspect of SA/SD and consists of a context diagram and an associated event list. The purpose of the environmental model is to model the system at a high level of abstraction. On the other hand, the behavioral model embodies the *design* aspect of SA/SD as a series of data flow and control flow diagrams, entity-relationship diagrams, process and control specifications, state transition diagrams, and a data dictionary. Using suitable combinations of these elements, the designer models the real-time system in detail.

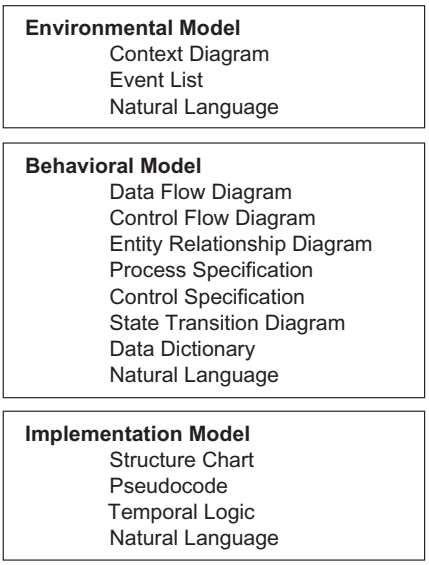


Figure 5.12. Models and elements of structured analysis and structured design.

It is suggested, however, that the data flow and control-flow analysis should be performed separately—not concurrently. Finally, in the implementation model, the developer uses a selection of structure charts, pseudocode, and temporal logic to describe the system to a level that can be readily translated to some procedural programming language. In addition, all the models M1–M3 may also contain some natural language descriptions. The presence of natural language descriptions is usually an indication that diagrams are not clear enough to the customer or developer, and textual clarifications are needed to complement visual modeling.

Structured analysis is a highly potential way to overcome the problems of classic analysis using graphical tools and a top-down, functional decomposition method to define system requirements. SA deals only with those aspects of analysis that can be structured: the *functional specifications* and the *environment/user interface*. Moreover, structured analysis is used to model a system's *context* (where inputs come from and where outputs go to), *processes* (what functions the system performs, how the functions interact, and how inputs are transformed to outputs), and *content* (the data the system needs to perform its functions).Structured analysis seeks to overcome the heterogeneous challenges inherent in system analysis through:

- Easy maintainability of the target document.
- Use of illustrative graphics.
- Effective reduction of ambiguity and redundancy.

- Providing a supportive method for functional partitioning.
- Building a logical model of the system before implementation.

The target document for SA is called the *structured specification*. It consists of a system context diagram, a hierarchical set of data flow diagrams showing the decomposition and interconnectivity of various components, and an event list to represent the set of external events that drive the system.

To illustrate the use of the structured analysis technique, consider the following example of the elevator control system introduced in Section 3.3.8. Some liberties have been taken with the notation, but this is common, as each organization tends to have its own "house style," that is, conventions that are dependent either on computer-aided software engineering (CASE) tools being used or individual preferences.

Example: Context Diagram of an Embedded System

The context diagram of Figure 5.13 defines the operating environment of the elevator control system (ECS). To make it easier to understand the diagram from the application viewpoint, it is necessary to give brief introductions to the purpose and operation of the six terminators (rectangular boxes) that are connected to the data transformation (a bubble):

1. *Motion Control Unit (*MCU*).* Drives the elevator car safely, smoothly, and precisely from one floor to another; provides car position and operational status information.
2. *Door Operator (*DO*).* Opens and closes the door blades swiftly but safely (see the finite state machine of Figure 5.2); provides operational as well as safety sensor and open/close button status information.
3. *Car Operating Panel (*COP*).* Shows the car-position and other run-specific information to the passengers; provides operational status information and an interface to car-call buttons.
4. *Hall Operation Panel (*HOP*).* Shows the car-position information to waiting passengers; controls a "lantern" and "gong," which indicate the run direction of the arriving/departing elevator; provides operational status information.
5. *Group Dispatcher (*GD*).* Assigns registered hall calls to appropriate elevators in the elevator bank using some optimal call-allocation strategy.
6. *Service Tool (ST).* Provides a password-protected user-interface for service personnel to give special commands, monitor the elevator in detail, and access operational statistics.

Here, the single data transformation of the context diagram models the whole elevator control system. The communication between the ECS and

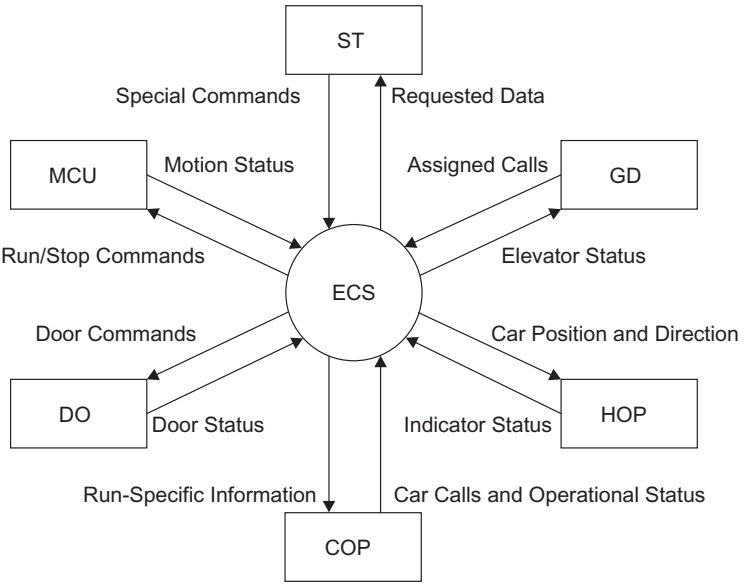


Figure 5.13. Context diagram for the elevator control system.

all terminators (external devices or subsystems) is bidirectional as indicated by data flows, and the transferred data is identified by descriptive labels. It should be noted that the data flows appearing in the context diagram ought to have the same abstraction level as the context diagram itself has. To conclude, the context diagram of Figure 5.13 forms a sound starting point for the remaining SA/SD process.

While the intent in the above example is not to present a complete system design—which means there are a few simplifications—a point to be made is that missing functionality is more easily recognized during the requirements elicitation process if some form of graphical aid, such as the SA context diagram, is available.

5.3.2 Object-Oriented Analysis and the Unified Modeling Language

As a viable alternative to the structured-analysis approach to developing software requirements specifications, we next consider using an object-oriented approach (Høydalsvik and Sindre, 1993). In contrast to procedural programming, which employs algorithmic procedures, object-oriented programming uses a structure of collaborating objects, in which each part performs its specialized processing by reacting to inputs from its immediate neighbors. There are various "flavors" of object-oriented analysis (OOA), each using its own toolsets. In the dominating approach discussed below, the system specification

phase begins with the representation of externally accessible functionality as *use cases* of the UML. Among practitioners, OOA is defined informally as "an analytical operation that uses UML diagrams" (Gelbard et al., 2010).

The UML approach, as a whole, is clearly more time-consuming to learn and more complicated to use than the SA/SD methods, since it (UML 2.2) has altogether 14 types of partially redundant diagrams divided into structural and behavioral categories (Miles and Hamilton, 2006):

1. *Structural*
   - Class diagram
   - Component diagram
   - Composite structure diagram
   - Deployment diagram
   - Object diagram
   - Package diagram
   - Profile diagram
2. *Behavioral*
   2.1. *General*
      - Activity diagram
      - State-machine diagram
      - Use-case diagram
   2.2. *Interaction*
      - Communication diagram
      - Interaction overview diagram
      - Sequence diagram
      - Timing diagram

All of these diagram types will be introduced in Chapter 6, while in the present discussion, we concentrate solely on those diagrams that are most relevant in the requirements engineering phase of a real-time software project.

Use cases are an essential artifact in object-oriented analysis and design and are described graphically using any of several techniques. The use-case diagram can be considered analogous to the context diagram in structured analysis in that it represents the interactions of the software application with its external environment. In the specification of an embedded system, this is also where overall timing constraints, sampling rates, and deadlines are often specified. Textual descriptions are used commonly to complement the use-case diagrams. A pragmatic discussion on creating appropriate use cases is available in Cockburn, (2001).

Use cases are represented graphically as ellipses, with the actors involved represented by stick figures, as can be seen in Figure 5.14. In that illustration, the use cases correspond to the five-level task structure proposed in Section 3.3.8. Generally speaking, it is often frustrating to decide on the level of detail

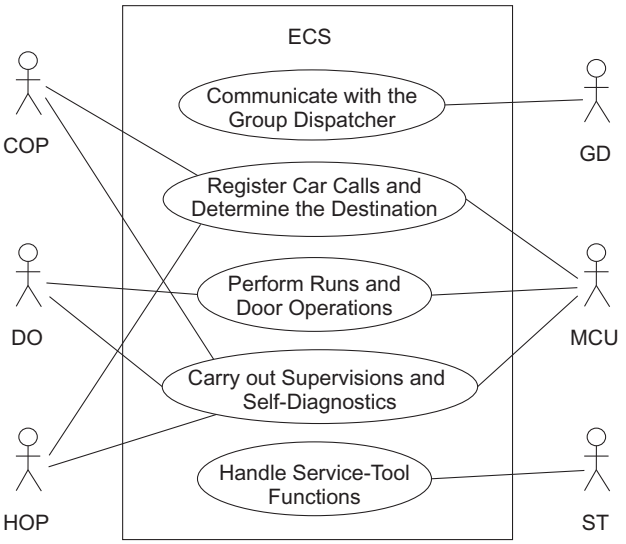


Figure 5.14. Use-case diagram for the elevator control system.

for a use case or try to understand what a particular use case consists of (Agarwal and Sinha, 2003). The lines drawn from the actor to the use case represent the communication between them. Each use case is actually a document that describes scenarios of operation of the system under consideration, as well as possible pre- and post-conditions and exceptions. In an iterative development process, these use cases will become increasingly refined and detailed as the analysis and design workflows progress. Next, interaction diagrams are created to describe the behaviors defined by each use case. In the first iteration, these diagrams depict the entire system as a black box, but once domain modeling has been completed, the black box is transformed into a collaboration of multiple objects. As an example, Figure 5.A3 in the case study in Section 5.7 illustrates the use-case diagram for the traffic-light control system.

Furthermore, an analysis class diagram presents the static structure of the system, system abstractions, and their relationships. It contains classes that represent entities with common characteristics, including attributes, operations, and associations that represent relationships between classes. The classes are depicted by rectangles, and the connection paths represent associations between classes. Classes require a name within the rectangle, whereas associations may not have an attached name. Moreover, the diamond attachment represents an aggregation relationship. If the diamond is filled, it is a dependent aggregation; otherwise it is independent, that is, the objects so aggregated can exist separately. Figure 5.A4 in the case study in Section 5.7 illustrates an analysis class diagram for the traffic-light control system. Class diagrams are used widely as a "cornerstone" of OOA.

The use of object-oriented approaches in real-time systems modeling provides numerous desired characteristics:

- Distributivity and concurrency
- Effective management of complexity
- Enhanced reusability
- Excellent traceability
- Improved understandability and maintainability
- Increased extensibility
- Modularity of design

Nevertheless, there are potential disadvantages when using an object-oriented approach with *time-critical* embedded systems, as discussed in Section 4.4.3.

A critical view on OOA and a discussion on specific weaknesses of using UML in the *analysis* phase are available in Gelbard et al. (2010). Their justified criticism concentrates on the observation that "UML representations have not been effective in large-scale projects for context and communication." They also argue that "OOA methodology lacks clarity and comprehensiveness." However, it is broadly recognized that the object-oriented approach strongly supports the *design* and *implementation* phases of (real-time) software development (Agarwal and Sinha, 2003).

5.3.3 Recommendations on Specification Approach

The preceding discussions illustrate typical challenges encountered by software engineers specifying real-time systems:

- Combining low-level hardware functionality and higher-level software and systems functionality at the same level of hierarchy.
- Mixing of descriptive and operational specifications.
- Omission of timing information.

It is not practical to prescribe here a single preferred technique, since it is well known that there is no "silver bullet" when it comes to software specification and design of a particular system. Therefore, each approach should be considered case-by-case on its specific merits. *Usability* of any technique has a crucial role in its initial acceptance and lifetime success. However, irrespective of the approach selected, real-time system modeling should incorporate the following best practices:

- Use uniform modeling techniques throughout the specification, for example, top-down decomposition together with structured analysis or object-oriented approaches.
- Separate operational specification from descriptive behavior.

- Use consistent levels of abstraction within models and conformance between levels of refinement across models.
- Model nonfunctional requirements as a part of the specification models, in particular, timing properties.
- Omit hardware–software partitioning in the specification phase (which is an aspect of design rather than analysis; a specification describes just *what* a real-time system must do, not how it will be done).

Finally, it should be noted that the borderline between analysis and design is usually hazy. The same applies to the other borderline between design and implementation, as well. And every organization can freely adjust those borderlines according to its needs and preferences.

5.4 THE REQUIREMENTS DOCUMENT

There are numerous ways to organize a software requirements specification (SRS), but the IEEE Std 830–1998 provides a sound template of what the SRS should look like (IEEE, 1998). The SRS can be seen as *a binding contract among designers*, *programmers*, *testers*, *and customers*, and it encompasses multiple paradigms or views for system design. The recommended design views include a combination of decomposition, dependency, interface, and detail descriptions. Together with boilerplate front matter, these form a standard template for software requirements specifications, which is shown in Figure 5.15. Sections 1 and 2 are self-evident; they provide front matter and introductory material for the SRS. The core of the SRS is, however, in the description sections, and their headings can be broken down further using, for example, structured analysis.

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions and Acronyms
   - 1.4 References
   - 1.5 Overview
2. Overall Description
   - 2.1 Product Perspective
   - 2.2 Product Functions
   - 2.3 User Characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and Dependencies
3. Specific Requirements

Appendices

Index

Figure 5.15. Recommended table of contents for the SRS from the IEEE Std 830–1998 (IEEE, 1998).

The IEEE Std 830 provides for several alternative (or complementary) means to represent the requirements specifications, aside from a function perspective. In particular, the software requirements can be organized by:

- System mode (e.g., normal, fireman service, and maintenance)
- User class (e.g., passenger, firefighter, and elevator technician)
- Objects (e.g., motor drive, car position sensors, and signaling devices)
- Feature (e.g., transport passengers from one floor to another)
- Stimulus (e.g., door contacts, push buttons, and safety sensors)
- Response (e.g., starting a floor-to-floor run upwards or downwards)
- Functional hierarchy (by common inputs, outputs or internal data access)
- Hybrid (combining two or more of the preceding)

5.4.1 Structuring and Composing Requirements

The text structure of the SRS can be depicted by the number of section identifiers at each hierarchical level. High-level requirements rarely have numbered sections below a depth of four (e.g., Section 3.2.1.5). Well-organized documents have typically a *pyramidal* structure to the requirements. Requirements with an *hourglass* structure, on the other hand, have too many administrative details, while *diamond*-structured requirements indicate subjects introduced at higher levels were addressed at different levels of detail (see Figure 5.16). Whatever approach is used in organizing the SRS, the IEEE Std 830 describes the characteristics of good requirements. Good requirements are:

- *Correct.* They must correctly describe the system behavior.
- *Unambiguous.* The requirements must be clear, not subject to multiple interpretations.

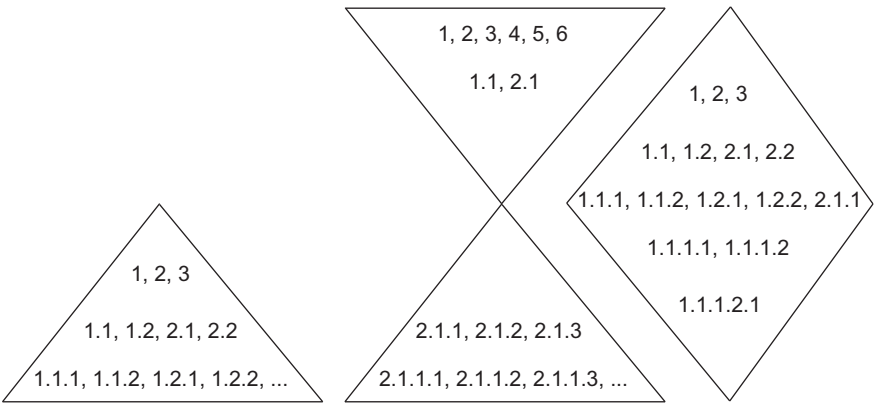


Figure 5.16. Triangle-, hourglass-, and diamond-shaped requirements structures.

- *Complete.* There must be no missing requirements.
- *Consistent.* No requirement may contradict another.
- *Ranked for importance and/or stability.* Designers will obtain guidance from ranked requirements when making trade-off decisions.
- *Verifiable.* A requirement that cannot be verified is a requirement that cannot be checked to have been met.
- *Modifiable.* The requirements need to be written in such a way so as to be easy to change.
- *Traceable.* The requirements provide a starting point for the backward/forward traceability chain.

To meet these criteria and to compose/edit clear requirements documentation, there are several best practices that the requirements engineer (or technical writer) can use. These include as follows:

- Use some standard format, and use it for all requirements.
- Use language in a consistent way and make sure that possible translations are exact in content.
- Use "shall" for essential requirements.
- Use "should" for desirable requirements.
- Use text highlighting to identify key parts of the requirements.
- Avoid the use of technical jargon unless it is warranted.

To illustrate this, consider the following five requirements:

1. "The system should be reliable."
2. "The system shall be modular."
3. "The system should be maintainable."
4. "The system shall be fast."
5. "The system shall be accurate."

These requirements are obviously *bad* for a number of reasons. None of them is verifiable; for instance, how are "reliability" and "modularity" supposed to be measured?

Next, consider a set of related requirements:

1. "The mean time between failures (MTBF) shall be at least 500 hours of continuous operation."
2. "The cyclomatic complexity of each program module shall be within the range of 10 to 40."
3. "The installing of any software update shall not take more than 15 minutes."

4. "Response times for all level-one operations shall be 250 ± 50 ms."
5. "The amplitude error in all estimated quantities shall be less than 3.5%."

These latter requirements are much better than the preceding ones. Each is measurable, because each makes some attempt to *quantify* the qualities that are desired: MTBF is a measure of reliability, cyclomatic complexity is a measure of modularity, updating time is a measure of maintainability, response time is a measure of speed, and amplitude error is a measure of accuracy.

5.4.2 Requirements Validation

Verification of the final software product means ensuring that the software is conforming to the SRS. It is akin to asking the question "Am I building the software as specified?" in that it requires satisfaction of all requirements.

Requirements validation, on the other hand, is tantamount to asking the question "Am I building the right software?" Too often, development projects deliver a fully functional real-time system that conforms to the SRS, only to discover that it is not what the customer really wanted. How could such an unfortunate outcome be prevented? Well, obviously, by strictly following a thorough requirements validation process, which is developed continually and systematically.

Performing such a requirements validation involves checking the following:

- *Validity.* Does the system provide functions that best (within the existing constraints) support the customer's needs?
- *Consistency.* Are there any requirements conflicts?
- *Completeness.* Are all functions required by the customer included?
- *Realism.* Can the requirements be implemented given available budget, time, and technology?
- *Verifiability.* Can the requirements be checked?

There are a number of ways of checking the software requirements specification for conformance to the IEEE standard's best practices and for ultimate validity. These mutually complementary approaches include (in alphabetic order):

- Automated consistency analysis
- Checking the consistency of a structured requirements description
- Comparing the requirements to those for a similar system
- Developing tests for requirements to check testability
- Prototyping
- Requirements reviews
- Systematic manual analysis of the requirements

- Test-case generation
- Using an executable model of the system to check requirements

Of these approaches, automated checking is the most desirable but, unfortunately, the least likely, because of the context sensitivity of natural languages, and the impossibility of verifying such cumbersome issues as requirements completeness. However, supporting tools can be developed to perform straightforward spelling and grammar checking (which may also indicate ambiguity and incompleteness), flagging of keywords that may be vague (e.g., "fast"), identification of missing requirements (e.g., search for the typical phrase "to be determined") and overly complex sentences (which can indicate unclear requirements).

Model checking is a formal technique that can be used to perform analysis of executable requirements specifications, even partial ones. The aim, however, is to find errors, not to prove correctness. One such methodology uses finite state machines to test for safety and liveness. The first step involves building a state model of the system (or one of its subsystems), for instance, using statecharts. Once this initial model is obtained, the state-space size is estimated in order to assess the potential for automated validation. Next, the state space is minimized by identifying possible equivalence classes and by exploiting symmetries and subclasses. Finally, a symbolic representation of the main features of the requirements is derived. This represents a behavioral, temporal logic structure that emulates the coarse-grain behavior of the system. For example, to check for fault tolerance, relevant faults are injected into the emulating model, and this model is exercised to identify possible problems (Schneider et al., 1998). Model checking represents, in some way, a high-level prototype of the requirements.

Automated requirements checking is used to assess certain qualities of requirements specifications, not to assess the correctness of the SRS. One example of such an approach is NASA's Automated Requirements Measurement (ARM) tool (Wyatt et al., 2003). Versatile tools, like ARM, use several requirements indicators at both a coarse-grain and fine-grain scale. Coarse-grain indicators include:

- Readability
- Size of requirements
- Specification depth
- Text structure

Fine-grain measures, on the other hand, look at the use of certain categories of words in the documents. Typical indicators are as follows:

- Imperatives
- Continuances

TABLE 5.6. Imperatives Found in Requirements Specifications and Their Purpose (Wilson, 1997)

Imperative	Purpose
Shall	Dictates provision of fundamental capability
Must	Establishes performance requirements or constraints
Must not	Establishes performance requirements or constraints
Is required to	Used in specifications statements written in passive voice
Are applicable	Used to include, by reference, standards, or other documentation as an addition to the requirements being specified
Responsible for	Used as an imperative for systems whose architectures are already defined
Will	Generally used to cite things that the operational or development environment are to provide to the capability being specified
Should	Not recommended for use

- Directives
- Options
- Weak phrases

Various imperatives are listed in Table 5.6.

Continuances follow an imperative and introduce the specification of requirements at a lower level. Continuances include such words/phrases as:

- "As follows"
- "Below"
- "Following"
- "In particular"
- "Listed"
- "Support"

Directives are words and phrases that point to illustrative information:

- "Depict"
- "Figure"
- "For example"
- "Such as"
- "Table"

Options give the developer latitude in satisfying the specifications, and include:

- "Can"
- "Could"

- "May"
- "Optionally"

Moreover, weak phrases, which should be avoided in the SRS, include:

- "Adequate"
- "As a minimum"
- "As applicable"
- "Be able to"
- "Be capable"
- "But not limited to"
- "Capability of"
- "Capability to"
- "Effective"
- "If practical"
- "Normal"
- "Provide for"
- "Timely"
- "To be determined"

These fine-grained measures can, minimally, be used to measure certain size quantities of the SRS, such as:

- Imperatives
- Lines of text
- Paragraphs
- Subjects (unique words following imperatives)

Useful numerical ratios can also be computed from these fine-grained measures, which can be used to judge the overall fitness of the software specification. Typical ratios are shown in Table 5.7.

TABLE 5.7. Numerical Ratios Derived from Software Requirements Specifications and Their Purpose

Ratio	Purpose
Imperatives to subjects	Indicates level of detail
Lines of text to imperatives	Indicates conciseness
Number of imperatives found at each document level	Counts the number of lower-level items that are introduced at a higher level by an imperative followed by a continuance
Specification depth to total lines of text	Indicates conciseness of the SRS

Readability statistics, similar to those used to measure writing level (or comprehension difficulty), can be used as a quality measure for the SRS. These statistics include:

- *Flesch Reading Easiness Index.* Total number of words/sentences and syllables/words.
- *Flesch–Kincaid Grade Level Index.* Flesch index converted to a grade level that is easier to judge (standard writing is seventh or eighth grade on the K–12 scale).
- *Coleman–Liau Grade Level Index.* Uses word-length in characters and sentence-length in words to determine the grade level.
- *Bormuth Grade Level Index.* Same as Coleman–Liau.

Any of these requirements metrics can be incorporated into a metrics management discipline, and if used consistently, constructively, and with good judgment, will improve the particular real-time system in the long run (as well as future real-time systems to be developed).

5.5 SUMMARY

Requirements engineering is the first phase of the software development process. And if it is not carried out properly, the software product may not fulfill the expectations and needs of customers—the resulting real-time system is simply "wrong" from the customers' point of view. In such an extreme scenario, it does not matter how well the system was designed or implemented. A decent recovery from inadequate requirements engineering can be costly and lead to considerable losses of revenue; some amount of redesign and reimplementation must definitely be done, and hence the product may come to market with a significant delay.

In spite of the critical role of requirements engineering, only a few undergraduate engineering programs stress the importance of this discipline. The majority of practicing engineers who perform requirements engineering are therefore educated on-the-job. Recently, however, some software programs are introducing requirements engineering as mandatory in the curriculum (Laplante, 2009). This should become more common in educational institutions around the world.

The large variety of existing specification methods (which ones to choose?) and the high cost of established CASE tools (can we afford them?) are tricky problems related to requirements engineering. Furthermore, it can be a major investment to a development organization to train its personnel to use the selected methods and acquired CASE tools effectively. For understandable reasons, it is desirable to have integrated CASE support during the entire software development process, but the license fees of complete CASE environments may be overly expensive for small and mid-size companies. A thor-

ough *cost–benefit* analysis is therefore required when selecting suitable methods and associated tools for an organization or project. Another important issue to consider is obviously whether the nontechnical customers will be able to comprehend the various diagrams (SA/SD or UML, for instance) in the requirements document.

As a rule of thumb, the number of specification methods used should be as low as possible but still adequate for the project at hand. Some combination of semiformal and formal methods is often advantageous when specifying real-time systems. While semiformal methods are typically of general-purpose character, formal methods like Petri nets are particularly useful when specifying communications protocols, for example. When tailoring a combination of semiformal and formal methods, the *usability* of individual methods and available CASE tools is of utmost importance.

Although it is a questionable stereotype, engineers are usually regarded as poor writers and communicators. On the other hand, the requirements document is an extensive written composition that is targeted not only to software developers but also to customers. Hence, it is important to use a sound template for organizing the software requirements specification. If no established in-house standard is yet in use, the IEEE Std 830–1998 offers a good framework for structuring the requirements document. In addition, it would be important to improve the technical writing proficiency of future practitioners already during the undergraduate studies. Nevertheless, it is challenging to integrate more *guided* writing opportunities to the usually overcrowded engineering curricula.

Finally, the main point of this chapter is that requirements engineering deserves more attention and systematic consideration, since it can be seen as one keystone toward sustainable success of software-development organizations.

5.6 EXERCISES

5.1. Estimate and justify the relative percentage of person months spent in each phase of the requirements engineering process (see Figure 5.1) for some embedded real-time system.

The instructor is encouraged to collect the estimates of all students and summarize the results (averages/medians/standard deviations)—*this is usually a fruitful starting point for a discussion or even a debate in class*.

5.2. Who should compose, analyze, and validate software requirements specifications?

5.3. Under what circumstances should software requirements specifications be changed? Who is authorizing such changes?

5.4. For an embedded system with which you are familiar, find three good requirements and three bad requirements in the software requirements

specification. Rewrite the bad requirements so that they comply with the IEEE Std 830–1998.

5.5. Give a concrete example of a case where it is beneficial to use a Mealy FSM instead of a Moore FSM for creating a formal part of software specifications.

5.6. The door-open button in Figure 5.2 is of momentary-pressure type. Hence, it is enough to press the button momentarily to open the door. However, when there is a fire in the building and the elevator is switched to *in-car fireman service*, the doors are not operated automatically anymore, but a firefighter operates the doors directly with door open/close buttons; no safety sensors or timeouts are in use. Besides, the door-open button is then of constant-pressure type. That is, the firefighter must press the button constantly until the door is fully open, otherwise the door will reclose swiftly (what is the motivation behind this functionality?). Redraw the finite state machine to fulfill the requirements of in-car fireman service.

5.7. For some traffic intersection in your neighborhood, draw a finite state machine that defines a common-sense control algorithm for the vehicle/pedestrian traffic lights.

5.8. Draw a statechart model of the control software for a simple digital camera. State clearly your assumptions regarding specific features of the camera.

5.9. Create a statechart with multiple orthogonal states (similar to that in Figure 5.6) for the elevator control system discussed in Section 3.3.8.

5.10. Use Petri nets instead of a finite state machine to represent the door-control subsystem of elevators depicted in Figure 5.2.

5.11. Using structured analysis, draw first a context diagram for a credit-card system described below. Then, go ahead and depict details of the functionality of the system. You are free to make assumptions as needed, but make sure that you have stated them clearly.

The credit-card system under consideration handles transactions for retail stores. For instance, a transaction might consist of buying a textbook from your favorite bookstore. Your data flow diagram(s) should include functions for retrieving and checking a credit-card record for a customer, approving and recording each transaction, and maintaining a log of transactions for each retail store. The system should maintain files of credit-card holders, current transactions, and accounts payable (approved transactions) for each store.

5.12. Draw a complete use case diagram for the credit-card system described in Exercise 5.11.

5.13. Consider a hospital's patient monitoring system. Each patient is connected to electronic instruments monitoring blood pressure, heart rate, and ECG. These monitoring instruments issue a binary signal indicating a STABLE (=0) or UNSTABLE (=1) condition. The results of each of these instruments are ORed together to form a signal called EMERGENCY. The EMERGENCY signals for each of the rooms (one patient per room) are then ORed together and sent to the nurse's workstation. If any instrument on any patient indicates an UNSTABLE condition, the emergency alarm is sounded and the nurse is urgently directed to the appropriate patient. Write a pseudo-code specification for such a system (define a simple pseudo-code syntax yourself).

5.14. What would be an appropriate combination of techniques to write software specifications for the:

(a) Semi-autonomous pasta sauce bottling system.

(b) Navigation unit for fighter aircraft.

(c) Airline reservation and booking system for local use.

5.15. What are the typical problems and ramifications of translating a formal requirements specification from one modeling technique (e.g., Petri nets) to another (e.g., statecharts)?

5.7 APPENDIX 1

CASE STUDY IN SOFTWARE REQUIREMENTS SPECIFICATION

The following is an excerpt from the Software Requirements Specification for a traffic-light control system. It embodies many of the elements discussed in this chapter in more detail, and provides a fully developed example of an object-oriented approach to requirements specification of a complex real-time system.

5.7.1 Introduction

Traffic controllers currently in use comprise simple timers that follow a fixed cycle to allow vehicle/pedestrian passage for a predetermined amount of time regardless of demand, actuated traffic controllers that allow passage by means of vehicle/pedestrian detection, and adaptive traffic controllers that determine traffic conditions in real-time by means of vehicle/pedestrian detection and respond accordingly in order to maintain the highest reasonable level of efficiency under varying conditions. The traffic controller described in this specification is capable of operating in all three of these modes.

5.7.1.1 Purpose This specification defines the software design requirements for an intersection control system for simple, four-way pedestrian/

vehicular traffic intersections. The specification is intended for use by end users, as well as software developers.

5.7.1.2 Scope This software package is part of a control system for pedestrian/vehicular traffic intersections that allows for (1) a fixed cycle mode, (2) an actuated mode, (3) a fully adaptive automatic mode, (4) a locally controlled manual mode, (5) a remotely controlled manual mode, and (6) an emergency preempt mode. In the fully adaptive automatic mode, a volume detection feature has been included so that the system is aware of changes in traffic patterns. Pushbutton fixtures are also included so the system can account for and respond to pedestrian traffic. The cycle is controlled by an adaptive algorithm that uses data from many inputs to achieve maximum throughput and acceptable wait times for both pedestrians and motorists. A preempting feature allows emergency vehicles to pass through the intersection in a safe and timely manner by altering the state of the signals and the cycle time.

5.7.1.3 Definitions, Acronyms, Abbreviations The following is a list of terms and their definitions as used in this document.

5.7.1.3.1 10-base T Physical connection formed by a twisted-pair as described in IEEE 802.3. Networking connection designed to transfer up to 10 megabits per second.

5.7.1.3.2 ADA Americans with Disabilities Act.

5.7.1.3.3 API Application Program Interface.

5.7.1.3.4 Approach Any one of the routes allowing access to an intersection.

5.7.1.3.5 Arterial Road A major traffic route or route used to gain access to a highway.

5.7.1.3.6 Aspect The physical appearance of an illuminated traffic standard.

5.7.1.3.7 Attribute Property of a class.

5.7.1.3.8 Cycle Time The time required to complete an entire rotation (cycle) of traffic signals at any one intersection.

5.7.1.3.9 Direct Route A route directly through the intersection that does not require the vehicle to turn.

5.7.1.3.10 DOT Department of Transportation.

5.7.1.3.11 Downstream The normal travel direction for vehicles.

5.7.1.3.12 Ethernet The most commonly used local area networking method as described in IEEE 802.3.

5.7.1.3.13 Intersection A system, including hardware and software, that regulates vehicle and pedestrian traffic where two or more major roads traverse. The class of intersection considered in this specification has only two roads.

5.7.1.3.14 Manual Override A device located at and physically connected to each intersection control system that allows traffic regulatory personnel to control the intersection manually.

5.7.1.3.15 Method Procedure within a class exhibiting an aspect of class behavior.

5.7.1.3.16 Message An event thrown from one code unit and caught by another.

5.7.1.3.17 Occupancy Loop A device used to detect the presence of vehicles in an approach or to count the passage of vehicles using an approach.

5.7.1.3.18 Offset The time difference between cycle start times at adjacent intersections. Applies only to coordinated intersection control, which is not covered by this specification.

5.7.1.3.19 Orthogonal Route A route through an intersection that requires a vehicle to turn.

5.7.1.3.20 Pedestrian Presence Detector A button console located on the corner of an intersection which gives pedestrians who wish to cross a street the ability to alert the intersection control system to their presence.

5.7.1.3.21 Pedestrian Traffic Standard Signals facing in the direction of pedestrian cross walks which have lighted indicators marked “Walk” and “Don’t Walk.”

5.7.1.3.22 Phase The state of an intersection. A particular period of the regulatory traffic pattern.

5.7.1.3.23 Remote Override A computer host that includes a software interface allowing a remote administrator to control the intersection remotely.

5.7.1.3.24 RTOS Real-time operating system.

5.7.1.3.25 Secondary Road A route that does not typically support high traffic volume or experiences less usage relative to another route.

5.7.1.3.26 SNMP (Simple Network Management Protocol) The de facto standard for inter-network management, defined by RFC 1157.

5.7.1.3.27 Split The duty cycle for a given phase, expressed as a decimal or percentage.

5.7.1.3.28 Vehicle Traffic Standard A traditional traffic signal with red, yellow, and green indicators.

5.7.1.3.29 Upstream Direction opposite to the normal direction of vehicle travel.

5.7.1.3.30 Vehicle Presence Detector See 5.7.1.3.17, Occupancy Loop.

5.7.1.3.31 WAN Wide area network.

5.7.1.4 Communications Standards

- 10 base-T Ethernet (IEEE 802.3)
- SNMP (RFC 1157)

5.7.1.5 Overview

5.7.2 Overall Description

5.7.2.1 Intersection Overview The intersection class to be controlled is illustrated in below Figure 5.A1.

The target class of intersection has the following characteristics:

1. Four-way crossing.
2. Roadway gradients and curvatures are small enough to be neglected.
3. No right-turn or left-turn lanes or right-turn and left-turn signals (note, however, that the intersection is wide enough to allow vehicles passing directly through to pass to the right of vehicles turning left).
4. Intersecting roads of different priorities (e.g., one road may be an arterial while the other may be a secondary road) or of equal priority.
5. Two vehicle traffic standards per approach: one suspended by overhead cable, the other mounted on a pedestal.
6. One pedestrian crosswalk per approach.

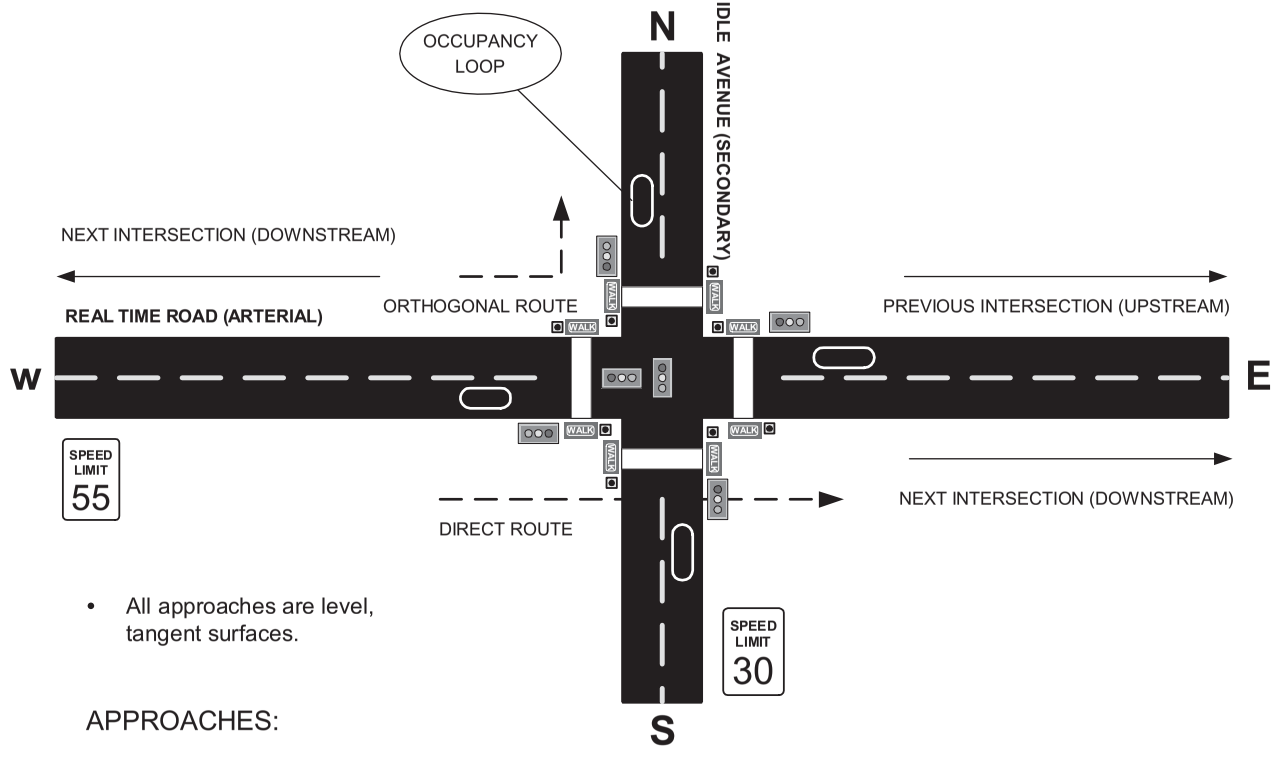


Figure 5.A1. Intersection topography.

7. Pedestrian traffic standards, pedestal mounted, on each side of each crosswalk.
8. Pedestrian presence detectors (pushbuttons) on each side of each crosswalk.
9. Stop-line vehicle presence detectors (loop detectors) in all approaches (one per approach) for detecting vehicle presence and for counting vehicles passing through the intersection.

5.7.2.2 Product Perspective

5.7.2.2.1 System Interfaces These are described in detail in the sections below.

5.7.2.2.2 User Interfaces

5.7.2.2.2.1 Pedestrians Pedestrian pushes button, generating service request to software and receives, in time, the "Walk" signal.

5.7.2.2.2.2 Motor Vehicles In ACTUATED mode, vehicle enters the intersection, generating service request to software and receives, in time, the "Okay to Proceed" signal.

In ADAPTIVE mode, vehicle passes over the loop detector, increasing the vehicle count, which, in turn, causes an adjustment in intersection timings.

5.7.2.2.2.3 Emergency Vehicle Emergency vehicle operator activates the "emergency vehicle override signal," generating priority service request to software and receives, in a preemptive time, the "Okay to proceed" signal.

5.7.2.2.2.4 Traffic Regulatory Personnel Traffic regulatory personnel will remove the manual override device from the control box and press buttons to control the intersection manually.

5.7.2.2.2.5 Remote Operator Remote operator uses a software control panel either to control the state of the intersection directly or to observe and manipulate the parameters and state of a specific intersection control system.

5.7.2.2.2.6 Maintainer Maintainer accesses system through Ethernet port to perform maintenance.

5.7.2.2.3 Hardware Interfaces The Intersection Control System hardware interfaces are summarized in Figure 5.A2 on the following page.

5.7.2.2.3.1 Major Hardware Components: Summary (Table 5.A1)

5.7.2.2.3.2 Wired Interfaces: Internal Hard-wired connections between the intersection controller and the following hardware components within the intersection controller enclosure are provided:

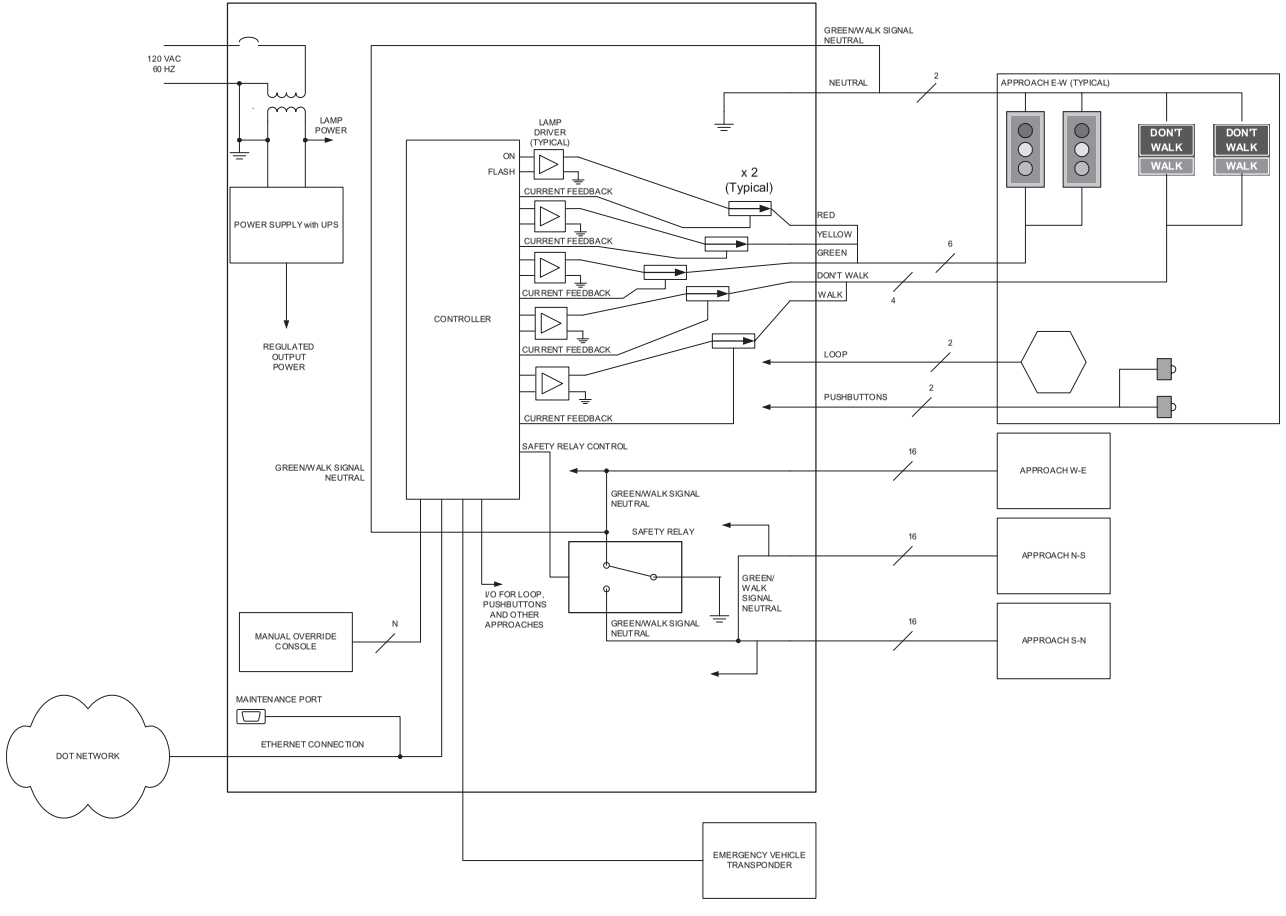


Figure 5.A2. Intersection controller hardware (not all details and interconnects shown).

TABLE 5.A1. Major Intersection Control System Hardware Components

Item	Description	Quantity
1	Intersection controller enclosure	1
1.1	Input circuit breaker	1
1.2	Input transformer	1
1.3	Input power supply with UPS	1
1.4	Intersection controller	1
1.5	Lamp driver	20
1.6	Lamp current sensor	40
1.7	Green signal safety relay	1
1.8	Manual override console	1
1.9	Vehicle presence detector interface unit (not shown in Figure 5.A2)	4
1.10	Pedestrian request detector interface unit (not shown in Figure 5.A2)	8
1.11	RJ-45 Ethernet connector—DOT network	1
1.12	RJ-45 Ethernet connector—maintenance	1
1.13	Enclosure wiring	A/R
2	Vehicle traffic standard—suspended	4
3	Vehicle traffic standard—pole mounted	4
4	Pedestrian traffic standard	8
5	Pedestrian request detector	8
6	Vehicle presence detector	4
7	Emergency vehicle transponder	1
8	Field wiring	A/R

1. Traffic standard lamp drivers (20)
2. Traffic standard lamp current sensors (40)
3. Vehicle presence detector interface units (4)
4. Pedestrian presence detector interface units (4)
5. Green signal safety relay (1)
6. Manual override console (1)
7. Maintenance connector (2; 10-base T twisted pair)

5.7.2.2.3.3 Wired Interfaces: External Hard-wired connections between the intersection control enclosure and the following external hardware components are provided:

1. Pedestrian presence detector
2. Pedestrian traffic standard
3. Vehicle presence detector
4. Vehicle traffic standard
5. Emergency vehicle transponder
6. DOT wide-area network (WAN)

5.7.2.2.3.4 Emergency Vehicle Transponder The emergency vehicle transponder is a radio frequency link between the intersection control system and the emergency vehicle override controller.

5.7.2.2.3.5 Ethernet Connection to DOT WAN Interaction between the software system and the remote operator console is conducted over a standard 10 base-T local area network. Each intersection control system is identified with a unique, statically assigned IP address.

5.7.2.2.4 Software Interfaces

5.7.2.2.4.1 Operating System The intersection controller interfaces to the RTOS via standard OS API calls.

5.7.2.2.4.2 Resource Managers Interfaces to hardware are handled by resource managers not specified in this SRS. Resource managers are assumed to have direct access to the object model defined here.

5.7.2.2.4.3 Software Control Panel The intersection control system must be able to interact with the software control panel to allow remote user access. This interface provides a remote user the ability to modify system parameters, perform maintenance functions, or assume manual control of the intersection. The standard protocol for this communication will be SNMP version 1.

5.7.2.2.5 Communications Interfaces The system will utilize TCP/IP's SNMP interface for inter-system communication.

5.7.2.2.6 Memory Constraints

5.7.2.2.6.1 Flash Memory Flash memory will be the memory media of choice for the system. The software will require no more than 32 M bytes of flash memory for RTOS, application program, and data.

5.7.2.2.6.2 RAM RAM will be used for application execution. The system shall not require more than 32 M bytes of RAM. Upon boot, the RTOS, application program and static data needed for execution will be copied from flash into the RAM.

5.7.2.2.7 Operations

1. Automatic, unattended operation (normal operation)
2. Local manual operation (through override console)
3. Remote manual operation (through WAN port)
4. Local observed operation (through maintenance port)
5. Remote observed operation (through WAN port)
6. Remote coordinated operation (option; through WAN port)

5.7.2.2.8 Site Adaptation Requirements This is summarized above in Section 5.7.2.1.

5.7.2.3 Product Functions The Intersection Control System provides the following functions:

1. Control of the intersection vehicle traffic standards
2. Control of the intersection pedestrian traffic standards
3. Collection and processing of traffic history from all approaches
4. Adaptive control of intersection timings in response to traffic flow
5. Actuated control of intersection in response to vehicle presence
6. Timed control of intersection in response to a fixed scheme
7. Handling of pedestrian crossing requests
8. Handling of emergency vehicle preemption
9. Intersection control in response to manual override commands
10. Intersection control in response to remote override commands
11. Management of traffic history and incident log databases
12. Handling of maintenance access requests from the maintenance port
13. Handling of maintenance access requests from the DOT WAN

5.7.2.4 User Characteristics

5.7.2.4.1 Pedestrians General population, including persons with disabilities.

5.7.2.4.2 Motor Vehicle Automobiles and trucks, depending on roadway use limitations.

5.7.2.4.3 Traffic Regulatory Personnel Authorized DOT, police, or other personnel trained in use of the Manual Override console. Must have key to the system enclosure.

5.7.2.4.4 System Administrators Authorized DOT personnel with training in the use of this system.

5.7.2.5 Constraints System Constraints include the following:

1. Regulatory policy (e.g., ADA)
2. DOT specifications
3. Local ordinances
4. Hardware limitations
5. Minimum time for pedestrian to cross
6. Minimum stopping distance for vehicles
7. Momentary power droops/outages
8. Interfaces to other applications

9. Audit functions
10. Higher-order language requirements (OO language supported by RTOS required)
11. Network protocols (e.g., SNMP)
12. Reliability requirements
13. Criticality of the application
14. Security considerations
15. Safety considerations

5.7.2.6 Assumptions and Dependencies

1. SI units are used for all physical quantities.
2. Commercially available RTOS is used.
3. Hardware interfaces have resource managers (drivers) already developed and available for integration with the software system specified here.
4. DOT WAN will use SNMP to communicate with intersection control system.
5. Watchdog circuitry forces safe default intersection state through hardware.

5.7.3 Specific Requirements

This section describes the basic functional elements of the intersection control system. In particular, the software object model is described in detail, with attributes and methods enumerated. External interfaces to users, hardware, and other software elements are described, and background on the adaptive algorithm to be used is provided.

5.7.3.1 External Interface Requirements

5.7.3.1.1 User Interfaces (Figure 5.A3)

1. Vehicle presence detector—user: motor vehicle
2. Pedestrian presence detector—user: pedestrian
3. Emergency vehicle override—user: emergency vehicle
4. Manual override—user: traffic control officer
5. Remote override—user: DOT officer
6. Maintenance interface—user: maintainer

5.7.3.1.2 Hardware Interfaces

1. Vehicle
2. Pedestrian crossing pushbutton
3. Traffic standard
4. Walk signal

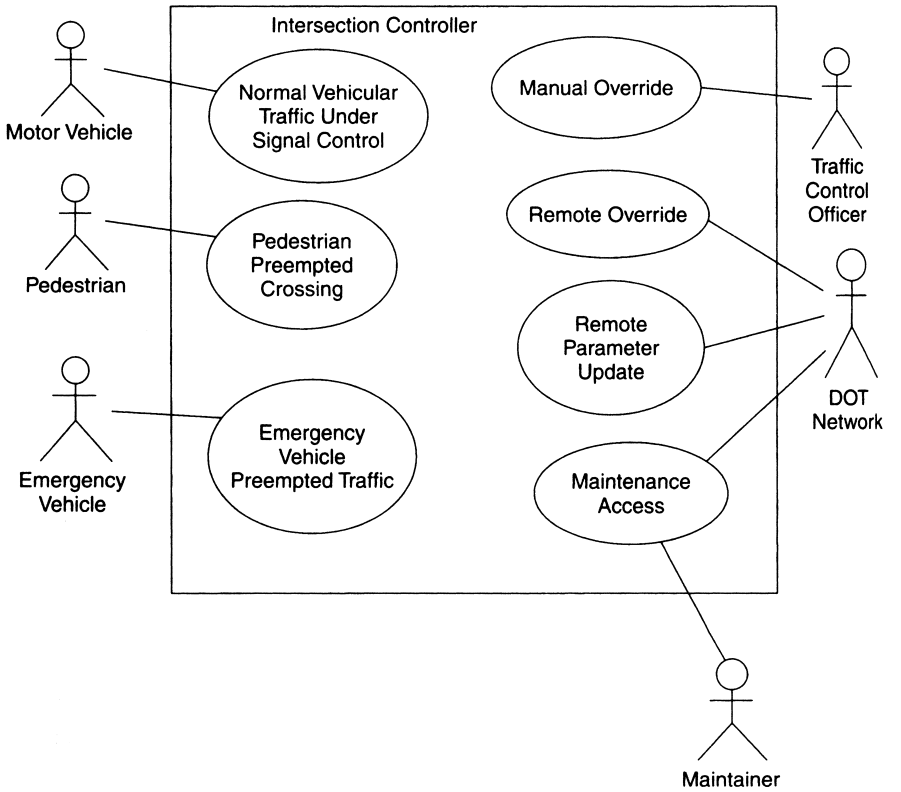


Figure 5.A3. Top-level use case diagram.

5. Hardware watchdog
6. Uninterruptible power supply

5.7.3.1.3 Software Interfaces

1. RTOS API calls
2. Hardware resource manager interfaces

5.7.3.1.4 Communications Interfaces

1. Interface to RTOS TCP/IP stack

5.7.3.2 Classes/Objects Figure 5.A4 depicts the classes constituting the intersection control system software application.

The Intersection Controller is responsible for managing the following functions:

1. Initialization
2. Instantiation of contained objects

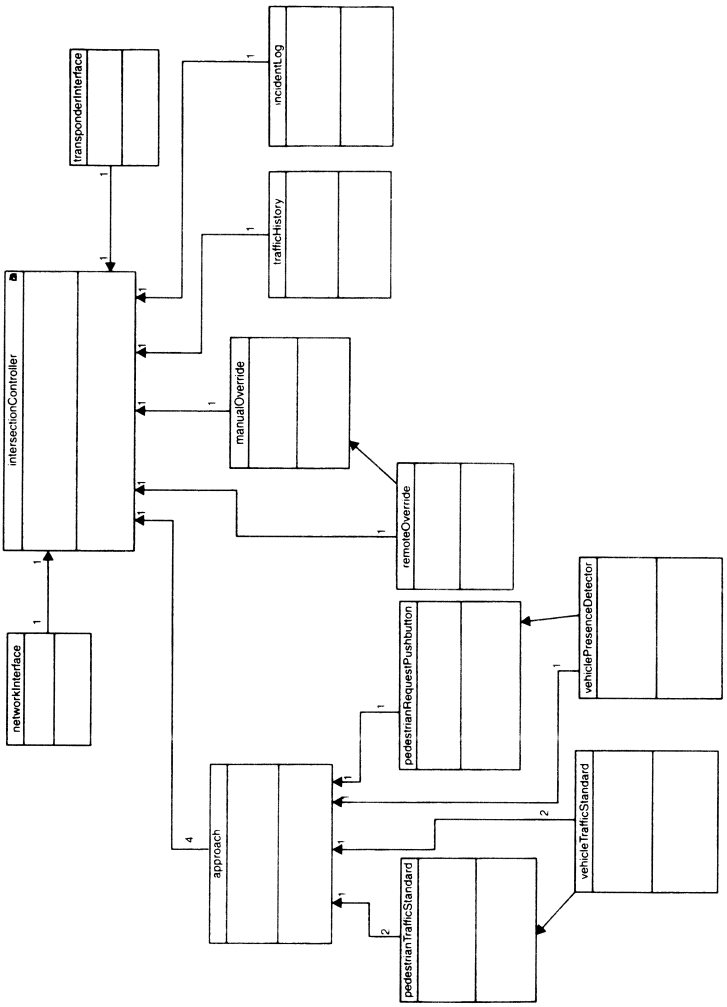


Figure 5.A4. Preliminary intersection controller class diagram.

3. Control of the intersection vehicle traffic standards
4. Control of the intersection pedestrian traffic standards
5. Collection and processing of traffic history from all approaches
6. Adaptive control of intersection timings in response to traffic flow
7. Actuated control of intersection in response to vehicle presence
8. Timed control of intersection in response to a fixed scheme
9. Handling of pedestrian crossing requests
10. Handling of emergency vehicle preemption
11. Intersection control in response to manual override commands
12. Intersection control in response to remote override commands
13. Management of traffic history and incident log databases
14. Handling of maintenance access requests from the maintenance port
15. Handling of maintenance access requests from the DOT WAN

Table 5.A2 below illustrates the attributes, methods, and events of the Intersection Controller class. Figure 5.A5 gives the statechart behavioral description for intersection control.

The corresponding traffic standard aspects are shown in Figure 5.A6.

5.7.3.2.1 Approach This is the programmatic representation of an intersection approach.

The Approach object is responsible for managing the following functions:

1. Instantiation of contained objects
2. Control of the traffic standards associated with the approach
3. Handling of pedestrian crossing events
4. Handling of loop detector entry and exit events
5. Tracking the vehicle count

Table 5.A3 below illustrates the attributes, methods, and events of the Approach class.

5.7.3.2.2 Pedestrian Traffic Standard This is the programmatic representation of a pedestrian crossing signal.

The Pedestrian Traffic Standard object is responsible for managing the following functions:

1. Displaying the commanded indication aspect from the Approach
2. Determining the indication actually displayed

Table 5.A4 below illustrates the attributes, methods, and events of the Pedestrian Traffic Standard class.

TABLE 5.A2. Intersection Controller Class

Intersection Controller		
	Name	Description
Attributes	Approaches	Array of Approach objects.
	Manual Override	Represents the Manual Override console.
	Remote Override	Represents the Remote Software console.
	Traffic History	Contains the traffic history for up to at least 7 days.
	Incident Log	Contains the incident log for up to at least 7 days.
	Network Interface	Object that provides an interface from the Network resource manager (driver) to the Intersection Controller object.
	Emergency Vehicle Interface	Object that provides an interface between the Emergency Vehicle transponder and the Intersection Controller object.
	Mode	Current operating mode of the Intersection Controller.
	Priority	Relative priority of the approaches.
	Cycle Time	Time to complete a full traversal of all intersection phases.
	Splits	Array of numbers defining the fraction of the cycle time allocated to each phase.
	Current Phase	Current intersection phase.
	Phase Time Remaining	Time remaining until the intersection moves to the next phase in the sequence.
	Commanded Green Signal Safety Relay State	Based on the Current Phase, this attribute holds the value required for the Green Signal Safety Relay resource manager, which is responsible for driving the relay.
	Detected Green Signal Safety Relay State	This holds the actual state of the Green Signal Safety Relay.
Methods	Initialize	
	Advance Phase	Advance the intersection phase to the next phase in the sequence.
	Calculate Cycle Parameters	Calculate the cycle time and splits for the next cycle based on traffic data.
Events	Phase Time Remaining Value Reaches 0	Fires when the Phase Time Remaining timer reaches 0.
	Override Activated	Fires when either the Manual Override or Remote override is activated.
	Override Canceled	Fires when Overrides are deactivated.
	Watchdog Timeout	Fires on a watchdog trip.
	Error	Fires when an error occurs. Takes the Error code as a parameter.

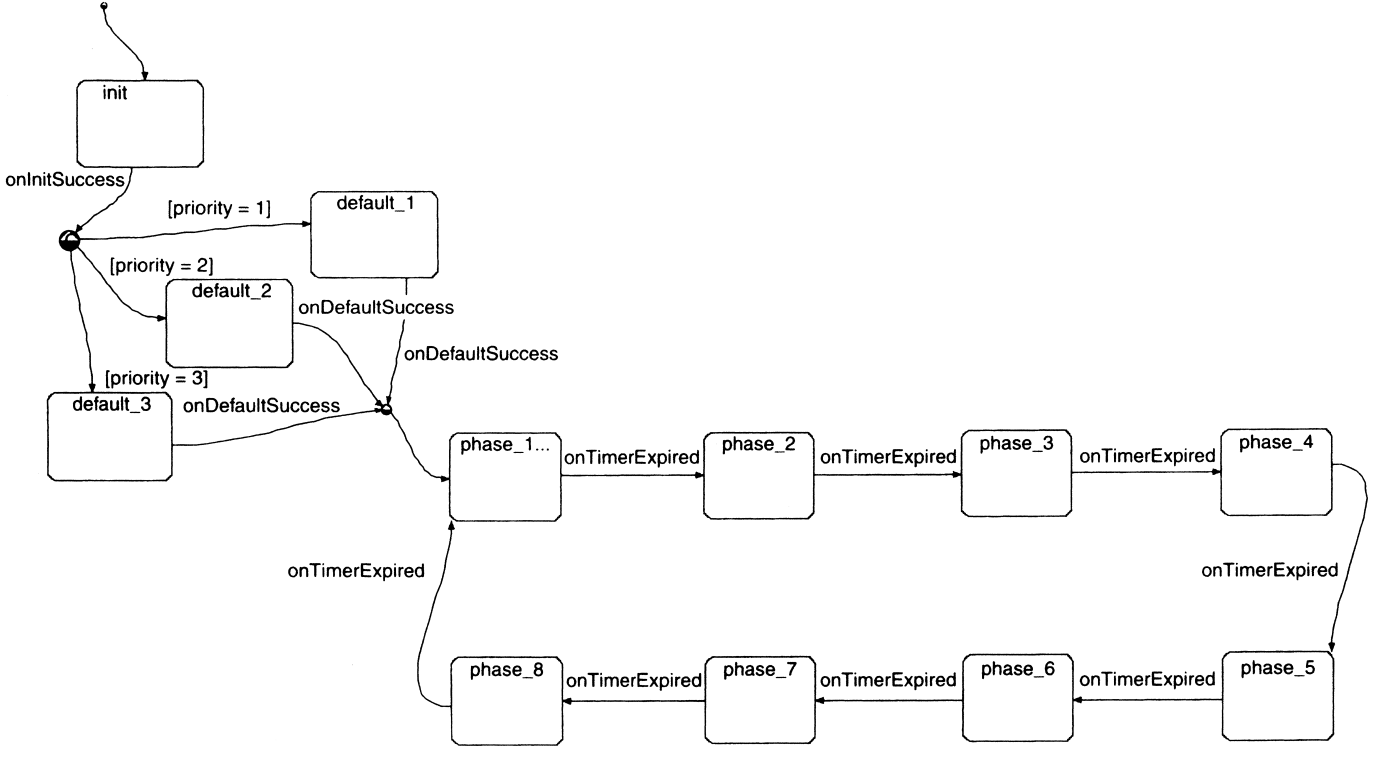


Figure 5.A5. Statechart for intersection controller phase sequence.

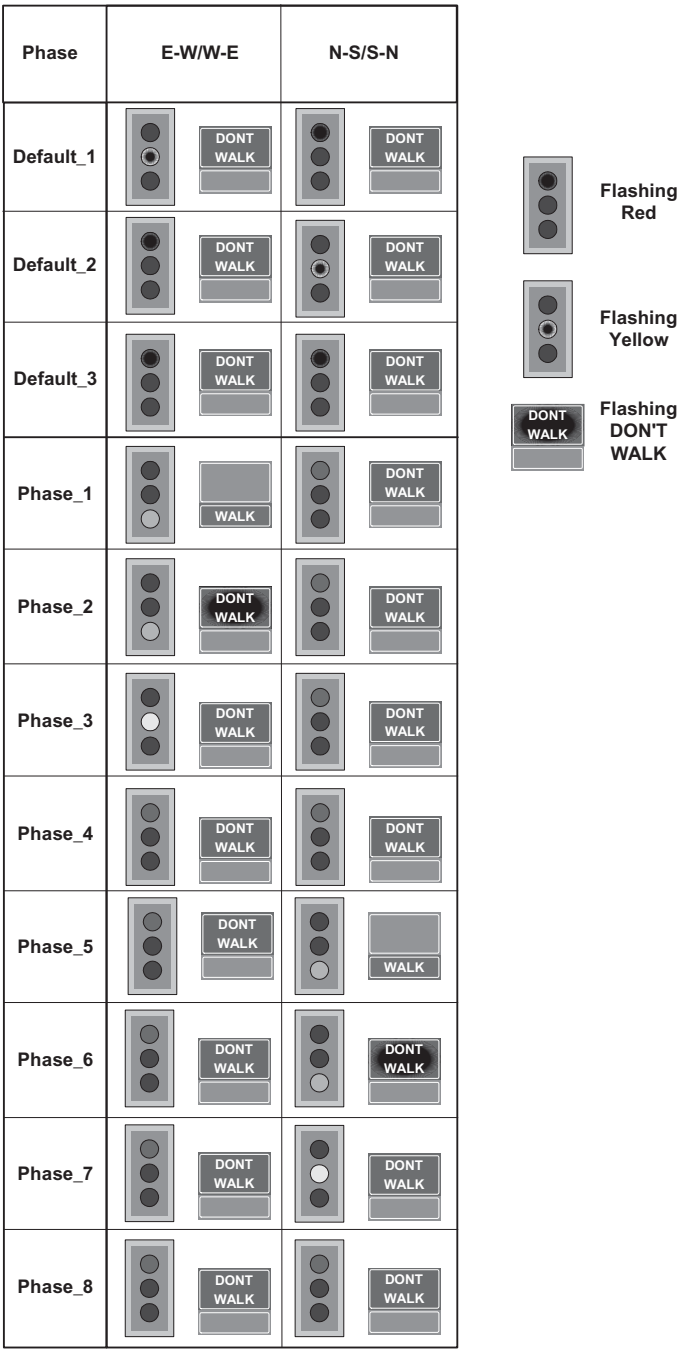


Figure 5.A6. Traffic standard aspects for each phase.

TABLE 5.A3. Approach Class

Approach		
	Name	Description
Attributes	Pedestrian Traffic Standard	Object representing the two pedestrian traffic standards associated with the approach.
	Vehicle Traffic Standards	Object representing the two vehicle traffic standards associated with the approach.
	Pedestrian Service Button	Object representing the two pedestrian service pushbuttons associated with the approach.
	Vehicle Presence Detector	Object representing the proximity detection loop, located at the stop line, associated with the approach.
	Vehicle Count	Count of vehicles passing through the approach.
	Indication	Array used to store the indications actually being displayed on all associated traffic standards.
	Current Aspect	Current commanded aspect corresponding to the Intersection Controller phase.
	Speed Limit	Value (in km/h) of the speed limit associated with the approach.
Methods	Set Aspect	Set the displayed aspect to the Commanded Aspect.
	Get Aspect	Get the actual displayed aspect based on signals from the current sensor hardware resource manager.
	Increment Count	Increase the vehicle count by 1.
	Reset Count	Reset the vehicle count to 0.
Events	Pedestrian Request	Fires when a pedestrian request has been made.
	Vehicle Entry	Fires when the loop detector detects vehicle entry.
	Vehicle Exit	Fires when the loop detector detects vehicle exit.

TABLE 5.A4. Pedestrian Traffic Standard Class

Pedestrian Traffic Standard		
	Name	Description
Attributes	Commanded Aspect	Commanded aspect from the Intersection Controller.
Methods	Set Indication	Set the displayed indication to the Commanded Indication.
	Get Indication	Get the actual displayed indication based on signals from the current sensor hardware resource manager.

TABLE 5.A5. Vehicle Traffic Standard Class

Vehicle Traffic Standard		
	Name	Description
Attributes	Commanded Aspect	Commanded aspect from the Intersection Controller.
Methods	Set Indication	Set the displayed indication to the Commanded Indication.
	Get Indication	Get the actual displayed indication based on signals from the current sensor hardware resource manager.

5.7.3.2.3 Vehicle Traffic Standard This is the programmatic representation of a vehicle traffic signal.

The Vehicle Traffic Standard object is responsible for managing the following functions:

1. Displaying the commanded aspect from the Intersection Controller.
2. Determining the aspect actually displayed.

Table 5.A5 above illustrates the attributes, methods, and events of the Vehicle Traffic Standard class.

5.7.3.2.4 Pedestrian Service Button This is an object representing the set of push-button consoles located on opposite sides of the crosswalk associated with an approach.

The Pedestrian Service Button object is responsible for managing the following functions:

1. Filtering of pushbutton service requests.
2. Generation of Pedestrian Service Request event.

Table 5.A6 below illustrates the attributes, methods, and events of the Pedestrian Service Button class.

5.7.3.2.5 Vehicle Presence Detector This is an object representing the proximity detection loop located near the stop line associated with an approach. The object class is based on the Pedestrian Service Button class.

The Vehicle Presence Detector object is responsible for managing the following functions:

1. Filtering of vehicle service requests (ACTUATED mode).
2. Generation of Vehicle Service Request event (ACTUATED mode).

TABLE 5.A6. Pedestrian Service Button Class

Pedestrian Service Button		
	Name	Description
Attributes	Request Masked	Indicates whether pedestrian service pushbutton signals should be ignored or processed.
	Request State	Indicates whether or not a pedestrian service request is active.
Methods	Set Request State	In response to a signal from the pushbutton hardware resource manager, determine whether or not to modify the Request State and raise an event.
	Reset Request State	Clear the Request State.
	Ignore Request State	Masks subsequent pedestrian button operations.
	Listen Request State	Respond to subsequent pedestrian button operations.
Events	Pedestrian Service Request	Indicates that a valid pedestrian service request is active.

TABLE 5.A7. Vehicle Presence Detector Class

Vehicle Presence Detector		
	Name	Description
Attributes	Request State	Indicates whether or not a vehicle service request is active (ACTUATED mode).
Methods	Set Request State	Set the Request State.
	Reset Request State	Clear the Request State.
Events	Vehicle Entry	Indicates that the detector loop is occupied.
	Vehicle Exit	Indicates that the detector loop is no longer occupied.

3. Maintenance of the vehicle count statistic (FIXED, ACTUATED and ADAPTIVE mode).

Table 5.A7 above illustrates the attributes, methods, and events of the Vehicle Presence Detector class.

5.7.3.2.6 Manual Override This is an object representing the set of push-buttons on the manual override console.

The Manual Override object is responsible for managing the following functions:

TABLE 5.A8. Manual Override Class

Manual Override		
	Name	Description
Attributes	None	None
Methods	None	None
Events	Override Activated	Fires when the override is activated.
	Override Canceled	Fires when the override is de-activated.
	Advance Phase	Fires in response to the ADVANCE button on the override console being pressed.

1. Triggering the appropriate mode change.
2. Generation and handling of events required to control intersection phase.

Table 5.A8 above illustrates the attributes, methods, and events of the Manual Override class.

5.7.3.2.7 Remote Override This is an object representing the commands available on the Remote Software console. Additionally, the object provides an interface for remote access to and update of intersection traffic data and cycle parameters for coordinated intersection control (option).

The Remote Override object is responsible for managing the following functions:

1. Triggering the appropriate mode change.
2. Generation and handling of events required to control intersection phase.

Table 5.A9 below illustrates the attributes, methods, and events of the Remote Override class.

5.7.3.2.8 Emergency Vehicle Interface This is an object that manages the wireless transponder interface to authorized emergency vehicles and accesses the Intersection Control object in order to display the correct traffic signals, allowing the emergency vehicle priority access to the intersection.

The Emergency Vehicle Interface object is responsible for managing the following functions:

1. Triggering the appropriate mode change.
2. Reception of emergency vehicle preemption requests.
3. Decryption and validation of emergency vehicle preemption requests.
4. Generation and handling of events required to control intersection phase.

TABLE 5.A9. Remote Override Class

Remote Override		
	Name	Description
Attributes	None	None
Methods	Process Command	Processes the events generated by the object, modifying the appropriate attribute or calling the appropriate method of the Intersection Controller object.
	Get Status	Retrieves the all parameter and other status data used as inputs to the Calculate Cycle Parameters adaptive control algorithm.
	Set Parameters	Sets the cycle timing parameters as calculated by the remote host.
Events	Override Activated	Fires when the override is activated.
	Override Canceled	Fires when the override is de-activated.
	Advance Phase	Fires in response to the ADVANCE command from the Remote Software console.

TABLE 5.A10. Emergency Vehicle Interface Class

Emergency Vehicle Interface		
	Name	Description
Attributes	None	None
Methods	None	None
Events	Preempt Activated	Fires when preemption is activated.
	Preempt Canceled	Fires when preemption is de-activated.
	Preempt Timeout	Fires when the preempt cancellation timeout interval expires.

Table 5.A10 above illustrates the attributes, methods, and events of the Emergency Vehicle Interface class.

5.7.3.2.9 Network Interface This is an object that manages communication via the Ethernet port.

The Network Interface object is responsible for managing the following functions:

1. Routing control messages to the appropriate objects.
2. Transferring traffic history and incident log data.
3. Management of maintenance operations.

Table 5.A11 below illustrates the attributes, methods, and events of the Network Interface class.

TABLE 5.A11. Network Interface Class

Network Interface		
	Name	Description
Attributes	None	
Methods	Process Message	Analyzes and routes network messages.
	Receive Message	Receives network messages.
	Send Message	Sends network messages.
Events	None	

TABLE 5.A12. Traffic History Class

Traffic History		
	Name	Description
Attributes	Record	An array of structures, each of which holds a single traffic history record.
	First Record	Index of the first active record.
	Last Record	Index of the record most recently added.
	Record Pointer	Index used to sequence through the Traffic History records.
Methods	Write Record	Writes a database record at the current position or at a specified position.
	Read Record	Reads a database record at the current position or at a specified position.
	Move Record Pointer	Moves record pointer as specified.
	Clear Database	Returns the database to an empty state.
Events	EOF	Fires when the last record is reached.
	Database Full	Fires when all allocated space for the database is used. Since the database is a FIFO structure, records will begin to be overwritten.

5.7.3.2.10 Traffic History This is an object that manages the stored traffic history.

The Traffic History object is responsible for managing the following functions:

1. Storage and retrieval of traffic history database records.
2. Clearing of traffic history in response to a command from a remote host.

Table 5.A12 above illustrates the attributes, methods, and events of the Traffic History class.

5.7.3.2.11 Incident Log This is an object that manages the stored incident log.

The Incident Log object is responsible for managing the following functions:

1. Storage and retrieval of incident log database records.
2. Clearing of incident in response to a command from a remote host.

Incidents are generated by the following events:

1. Error conditions.
2. Traffic History database full.
3. System resets.
4. Mode changes, including emergency vehicle preempts.
5. Maintenance actions, as updated by maintenance personnel through portable test equipment (laptop).

Table 5.A13 below illustrates the attributes, methods, and events of the Incident Log class.

5.7.3.3 Performance Requirements

5.7.3.3.1 Timing Requirements

5.7.3.3.1.1 Summary Table 5.A14 below provides a summary of all timing requirements.

This is illustrated in Figures 5.A7 and 5.A8 below.

TABLE 5.A13. Incident Log Class

Incident Log		
	Name	Description
Attributes	Record	An array of structures, each of which holds a single traffic history record.
	First record	Index of the first active record.
	Last record	Index of the record most recently added.
	Record pointer	Index used to sequence through the Traffic History records.
Methods	Write record	Writes a database record at the current position or at a specified position.
	Read record	Reads a database record at the current position or at a specified position.
	Move record pointer	Moves record pointer as specified.
	Clear database	Returns the database to an empty state.
Events	EOF	Fires when the last record is reached.
	Database full	Fires when all allocated space for the database is used. Since the database is a FIFO structure, records will begin to be overwritten.

TABLE 5.A14. Software Timing Requirements

ID	Designation	Applies to Mode(s)	Object/From Event	Object/To Response	Min Time (ms)	Max Time (ms)
1	Initialization	All	Hardware/Reset Signal	Intersection Controller/ Initialization Complete	–	4900
2	Set Default Phase	All	Intersection Controller/ Initialization Complete	All Traffic Standards/ Display of Commanded Phase	–	100
3	Start Normal Operation	ACTUATED ADAPTIVE TIMED	Intersection Controller/ Initialization Complete	All Traffic Standards/ Display of Phase 1	–	500
4	Advance Phase—Normal	ACTUATED ADAPTIVE TIMED	Intersection Controller/Phase Time Remaining Reaches 0	All Traffic Standards/ Display of Commanded Phase	–	100
5	Advance Phase—Local	LOCAL_MANUAL	Manual Override/Receipt of Advance Phase signal from Manual Override Panel	All Traffic Standards/ Display of Commanded Phase	–	100
6	Advance Phase—Remote	REMOTE_ MANUAL	Remote Override/Receipt of Advance Phase signal from Network Interface	All Traffic Standards/ Display of Commanded Phase	–	100
7	Calculate Cycle Parameters—Actuated	ACTUATED	Pedestrian Detector or Vehicle Detector/Pedestrian Request signal or Vehicle Request signal	Intersection Controller/ Cycle Time and Splits updated	–	100
8	Calculate Cycle Parameters—Adaptive	ADAPTIVE	Intersection Controller/Start of last phase in cycle	Intersection Controller/ Cycle Time and Splits updated	–	250
9	Critical Error—Display Defaults	All	Any/Critical Error	All Traffic Standards/ Display of Default State	–	50

(*Continued*)

TABLE 5.A14 (*Continued*)

ID	Designation	Applies to Mode(s)	Object/From Event	Object/To Response	Min Time (ms)	Max Time (ms)
10	Critical Error—Alarm	All	Any/Critical Error	Network Interface/Initiation of Alarm Transmission	–	1000
11	Critical Error—Reset	All	Any/Critical Error	Intersection Controller/System Reset	4500	5000
12	Write Error Log	All	Any/Any Error	Incident Log/Write Completed	–	500
13	Set Phase	ACTUATED ADAPTIVE MANUAL REMOTE	Intersection Controller/Advance phase	All Traffic Standards/Display Commanded phase	–	100
14	Get Phase	ACTUATED ADAPTIVE MANUAL REMOTE	Intersection Controller/Get phase	Intersection Controller/Displayed phase determined	2	150
15	Check Phase	ACTUATED ADAPTIVE MANUAL REMOTE	Intersection Controller/Check phase	Intersection Controller/Phase check status returned	–	10
16	Pedestrian Request Latching	ACTUATED ADAPTIVE TIMED	Resource Manager/Pedestrian Request signal	Pedestrian Detector/Latching of Pedestrian DetectorPending state	–	10
17	Pedestrian Request Reset	ACTUATED ADAPTIVE TIMED	Intersection Controller/completion of phase(s) during which pedestrian requests may be accepted	Pedestrian Detector/Clearing of Pedestrian DetectorPending state	–	100
18	Pedestrian Request Processing	ACTUATED ADAPTIVE TIMED	Pedestrian Detector/Latching of Pedestrian DetectorPending state	Intersection Controller/Updating of cycle time for next two (2) cycles; updating of all splits for next two (2) cycles.	–	100

19	Vehicle Entrance	FIXED ACTUATED ADAPTIVE	Resource Manager/Vehicle Entry signal	Vehicle Detector/Vehicle Entry state set	–	10
20	Vehicle Exit	FIXED ACTUATED ADAPTIVE	Resource Manager/Vehicle Exit signal	Vehicle Detector/Vehicle Entry state cleared	–	10
21	Vehicle Request Processing	FIXED ACTUATED ADAPTIVE	Vehicle Detector/Entry state set	Intersection Controller/ Process Vehicle Request	–	100
22	Vehicle Reset Request State	ACTUATED ADAPTIVE	Intersection Controller/Reset Vehicle Request state	Vehicle detector/Clear Vehicle Entry state	–	100
23	Vehicle Count Update	FIXED ACTUATED ADAPTIVE	Vehicle Detector/Entry State Cleared	Approach/Bump Count	–	50
24	Vehicle Count Fetch	FIXED ACTUATED ADAPTIVE	Intersection Controller/Get Count	Intersection Controller/ Count Returned	–	100
25	Vehicle Count Reset	FIXED ACTUATED ADAPTIVE	Intersection Controller/Phase Change	Approach/Reset Count	–	100
26	Get Cycle Parameters	REMOTE	Remote Override/Parameter Request	Network Interface/ Packet Ready to Send	–	100
27	Update Cycle Parameters	REMOTE	Remote Override/Parameter Update	Intersection Controller/ Parameters Updated	–	100
28	Process Message	EMERGENCY PREEMPT	Emergency Vehicle Interface/ Activate	Intersection Controller/ Mode Changed	–	200
29	Process Command	EMERGENCY PREEMPT	Emergency Vehicle Interface	Intersection Controller/ Emergency Vehicle Operations	–	100

(*Continued*)

TABLE 5.A14 (*Continued*)

ID	Designation	Applies to Mode(s)	Object/From Event	Object/To Response	Min Time (ms)	Max Time (ms)
30	Process Message	REMOTE	Remote Operations	Intersection Controller/ Network Interface	–	200
31	Fetch Database	REMOTE	Remote Operations	Intersection Controller/ Network Interface	–	1000
32	Add Record	FIXED ACTUATED ADAPTIVE	Intersection Controller	Traffic History	–	200
33	Clear Database	FIXED ACTUATED ADAPTIVE	Intersection Controller	Traffic History	–	200
34	Add Record	FIXED ACTUATED ADAPTIVE	Intersection Controller	Incident Log	–	200
35	Clear Database	FIXED ACTUATED ADAPTIVE	Intersection Controller	Incident Log	–	200

Notes: The timing requirements for vehicle detection are based on the following considerations:

Minimum vehicle length = 8 ft

Minimum following distance in motion = 4 ft

Loop width = 4 ft

Loop detects entrance with leading overlap = 2 ft

Loop detects exit with trailing overlap = 1 ft

Maximum vehicle speed = 65 mph (= 95.3 ft/s)

Vehicle speed for minimum gap time (see below) = 10 mph (= 14.67 ft/s)

Minimum presence pulse width = 9 ft/95.3 ft/s = 94.4 ms

Minimum gap time (time between exit and next vehicle entrance) = 3 ft/14.67 ft/s = 204.5 ms

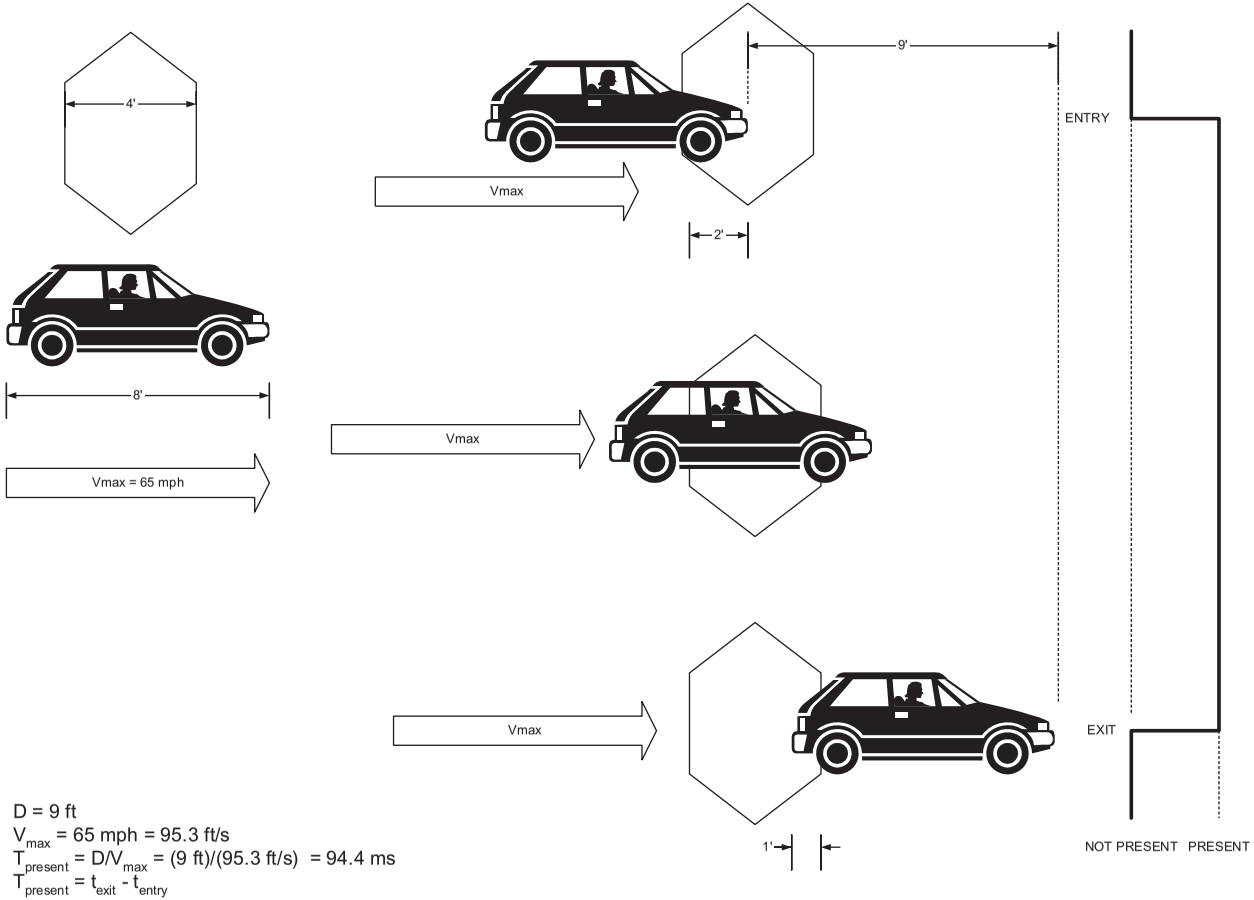


Figure 5.A7. Minimum presence pulse width.

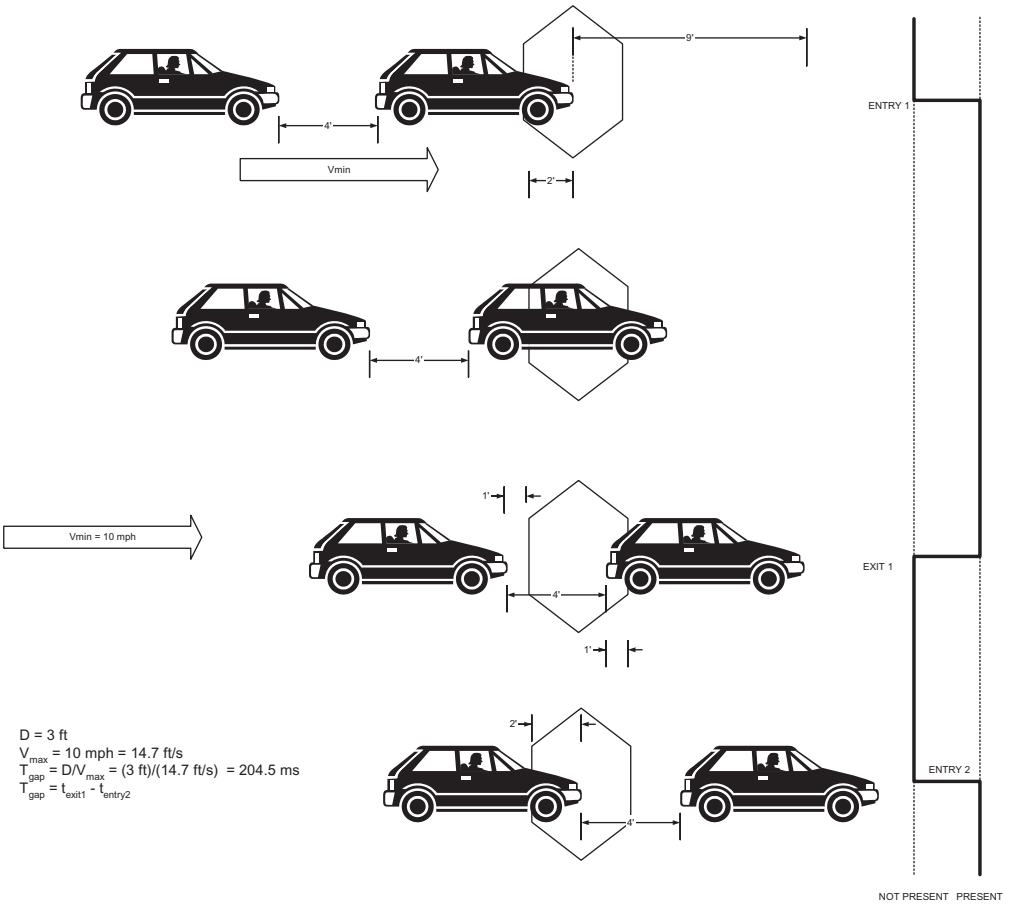


Figure 5.A8. Minimum gap pulse width.

It is necessary to determine the speed at which the minimum gap time between vehicles occurs in order to determine that time. The distance a following vehicle must cover in order to trigger the loop detector after the leading vehicle has exited is given by

$$D_{gap}(v) = \begin{cases} 4 \text{ ft}, v < 10 \text{ mph} \\ \dfrac{v^2}{2a} - \dfrac{\left(10 \text{ mph} \cdot \dfrac{88}{60}\right)^2}{2a} + 3 \text{ ft}, v \geq 10 \text{ mph} \end{cases}.$$

Given this, the gap time is given by

$$T_{\text{gap}}(v) = \frac{D_{\text{gap}}(v)}{v}, v > 0.$$

It can be shown that the minimum gap time occurs for $v = 10$ mph.

REFERENCES

R. Agarwal and A. P. Sinha, "Object-oriented modeling with UML: A study of developers' perceptions," *Communications of the ACM*, 46(9), pp. 248–256, 2003.

J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods," *IEEE Computer*, 28(4), pp. 56–63, 1995.

G. Bucci, M. Campanai, and P. Nesi, "Tools for specifying real-time systems," *Real-Time Systems*, 8(2/3), pp. 117–172, 1995.

A. Cockburn, *Writing Effective Use Cases*. Boston: Addison-Wesley, 2001.

T. De Marco, *Structured Analysis and System Specification*. New York: Yourdon Press, 1978.

R. Gelbard, D. Te'eni, and M. Sadeh, "Object-oriented analysis—is it just theory?" *IEEE Software*, 27(1), pp. 64–71, 2010.

H. Gomaa, "Extending the DARTS software design method to distributed real time applications," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, Kailua-Kona, HI, 1988, pp. 252–261.

D. Harel, "Statecharts in the making: A personal account," *Communications of the ACM*, 52(3), pp. 67–75, 2009.

G. M. Høydalsvik and G. Sindre, "On the purpose of object-oriented analysis," *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, 1993, pp. 240–255.

Institute of Electrical and Electronics Engineers, *IEEE Std 830–1998, Recommended Practice for Software Requirements Specification*. New York: IEEE Computer Society, 1998.

P. A. Laplante, *Software Engineering for Image Processing*. Boca Raton, FL: CRC Press, 2003.

P. A. Laplante, *Requirements Engineering for Software and Systems*. Boca Raton, FL: CRC Press, 2009.

S. Liu, *Formal Engineering for Industrial Software Development: Using the SOFL Method*. Berlin, Germany: Springer-Verlag, 2010.

A. Mazzeo, N. Mazzocca, S. Russo, and V. Vittorini, "A systematic approach to the Petri net based specification of concurrent systems," *Real-Time Systems*, 13(3), pp. 219–236, 1997.

R. Miles and K. Hamilton, *Learning UML 2.0*. Sebastopol, CA: O'Reilly Media, 2006.

S. Robertson and J. C. Robertson, *Requirements-Led Project Management: Discovering David's Slingshot*. New York: Addison-Wesley, 2005.

M. Samek, *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*, 2nd Edition. Burlington, MA: Newnes, 2009.

F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann, "Validating requirements for fault tolerant systems using model checking," *Proceedings of the 3rd IEEE International Conference on Requirements Engineering*, Colorado Springs, CO, 1998, pp. 4–13.

I. Sommerville, *Software Engineering*. New York: Addison-Wesley, 2000.

F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach*. Boca Raton, FL: CRC Press, 2006.

P. Ward and S. Mellor, *Structured Development for Real-Time Systems*. 1–3. New York: Yourdon Press, 1985.

W. Wilson, "Writing effective requirements specifications," in USAF *Software Technology Conference*, Salt Lake City, UT, 1997.

V. Wyatt, J. DiStefano, M. Chapman, and E. Aycoth, "A metrics based approach for identifying requirements risks," *Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop*, Greenbelt, MD, 2003, pp. 23–28.

E. Yourdon, *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

E. Yourdon, *Just Enough Structured Analysis*. New York: Yourdon Press, 2006 (free e-book available from http://www.yourdon.com/jesa/, last accessed August 17, 2011).

6

SOFTWARE DESIGN APPROACHES

Software design is a salient part of the entire software development process, which can be an individual subprocess of the high-level product development process. Furthermore, in embedded applications, the product development process may include concurrent hardware and subsystem development subprocesses, too. Software designers translate the problem-domain requirements document discussed in the previous chapter into physical models of the solution that are sufficient for straightforward implementation or programming. The resultant design document should be such that even an external programming consultant could implement the code with minimal interaction with the design team. Completeness of the design document is particularly important in globally distributed projects, where the requirements document might be created in the United States, the design document in Finland, and the implementation in India, for instance. Integrated CASE environments, which provide smooth transitions from the requirements engineering phase to the design phase and further to the implementation phase, should be used throughout the software development process. Besides, it is essential to use standardized/widespread modeling techniques, such as the SA/SD methods or the UML, to make the core documentation understandable for various collaborating teams.

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

The Institute of Electrical and Electronics Engineers (IEEE) Standard Dictionary of Electrical and Electronics Terms (IEEE Std 100–2000) describes the term "design" as follows: The process of defining the architecture, components, interfaces, and other characteristics of a system. Hence, during the software design phase, numerous decisions are made concerning responsibility assignment and fulfillment, system architecture and deployment, separation of concerns, as well as layering and modularization (Bernstein and Yuhas, 2005). Moreover, the computational algorithms and their numerical precision are specified in the design document; such important decisions are often supported by simulations or prototyping. The opportunity of design reuse also should be carefully considered. In our experience with typical real-time applications, the software design phase takes roughly the same amount of resources (in person months) as the requirements-engineering and programming phases together.

The desired qualities of real-time software as well as advantageous software engineering principles will be discussed in Sections 6.1 and 6.2, respectively. In addition, a mapping from these principles to qualities is sketched with a pragmatic discussion. As the procedural and object-oriented approaches exist in requirements engineering and programming phases, they both are naturally available in the design phase, as well. Therefore, we discuss the procedural design approach in Section 6.3 and the alternative object-oriented approach in Section 6.4. Both of these design sections are composed on real-world examples. Section 6.5 gives an evaluative overview on a sample of life cycle models that are currently used for the development of real-time software. The preceding sections on software design approaches are summarized in Section 6.6 with some suggestions. A carefully selected collection of stimulating exercises is provided in Section 6.7. Lastly, Section 6.8 contains a comprehensive case study on designing real-time software (the corresponding requirements document of this traffic-light control system is available in Section 5.7).

Some parts of this chapter have been adapted from Laplante (2003).

6.1 QUALITIES OF REAL-TIME SOFTWARE

Software systems and individual components can be characterized by a number of diverse qualities. *External* qualities are those that are observable by the user, such as performance and usability, and are of explicit interest to the end user. *Internal* qualities, on the other hand, are not observable by the user, but aid the software developers to achieve certain improvement in external qualities. For example, although the requirements and design documentation might never be seen by a typical user, their adequate quality is essential in achieving satisfactory external qualities. Such an external–internal distinction is a function of the software itself and the type of user involved.

While it is beneficial to know the software qualities and the motivations behind them, it is equally desirable to measure them objectively. Measuring

of these characteristics of software is necessary in enabling end users and designers to talk succinctly about the product, and for effective software process control and project management. More importantly, however, it is these qualities that shall be embodied in the real-time design.

6.1.1 Eight Qualities from Reliability to Verifiability

Reliability is a measure of whether a user can depend on the software (Teng and Pham, 2006). This quality can be informally defined in a number of ways. For instance, one definition might be simply "a system that a user can depend on." Other common characterizations of a reliable software system include:

- The system "stands the test of time."
- There is an absence of known errors that render the system useless.
- The system recovers "gracefully" from errors.
- The software is robust.

In particular, for real-time systems, other informal characterizations of reliability might include:

- Downtime is below a specified threshold.
- The accuracy of the system remains within a certain tolerance.
- Real-time performance requirements are met consistently.

While all of these informal characteristics are certainly desirable in real-time systems, they are difficult to measure or predict. Moreover, they are not true measures of reliability, but of various attributes of the software instead.

There is specialized literature on software reliability that defines this quality in terms of statistical behavior, that is, the probability that the software product will operate as expected over a specified time interval (Pham, 2000). These characterizations generally take the following approach. Let S be a software system, and let T be the time instant of system failure. Then the reliability of S at time t, denoted $r_s(t)$, or when there can be no confusion with other systems, $r(t)$, is the probability that T is greater than t; that is,

$$r(t) = P(T > t). \tag{6.1}$$

This is the probability that a software system will operate without failure for a specified period of time. In addition to the actual operating phase, also the testing phase may be included in the considered period.

A system with reliability function $r(t) = 1$ would never fail. However, it is unrealistic to have such an expectation with any real-world system. Instead, some reasonable goal, $r(t) < 1$, should be specified.

Example: Failure Probability Increases as a Function of Time

Consider the monitoring system of a nuclear power plant with the specified failure probability of no more than 10^{-9} per hour. This represents a reliability function of $r(t) = (0.999999999)^t$, where t is in hours. Note that as $t \to \infty$, $r(t) \to 0$. To illustrate, the failure probability, $q(t) = 1 - r(t)$, for various values of t is given in Table 6.1. Moreover, after 35 years of operation (306,600 hours)—still a reasonable time for nuclear power plants—the failure probability is approximately 0.0003.

Another way to characterize software reliability is in terms of a failure function or model. One failure function uses an exponential distribution where the abscissa is time and the ordinate represents the expected failure intensity at that time:

$$f(t) = \lambda / e^{\lambda t}, \quad t \geq 0. \tag{6.2}$$

Here the failure intensity is initially high, as would be expected in new software, since failures are detected more frequently during the testing phase. However, the number of failures would be expected to decrease with time during the operating phase, presumably as failures are uncovered and repaired (see Fig. 6.1). The factor λ is a system-dependent parameter that must be determined empirically.

TABLE 6.1. Failure probability as a Function of Operating Hours

t	10^0	10^1	10^2	10^3	10^4	10^5	10^6
$q(t)$	10^{-9}	$\approx 10^{-8}$	$\approx 10^{-7}$	$\approx 10^{-6}$	$\approx 10^{-5}$	$\approx 10^{-4}$	$\approx 10^{-3}$

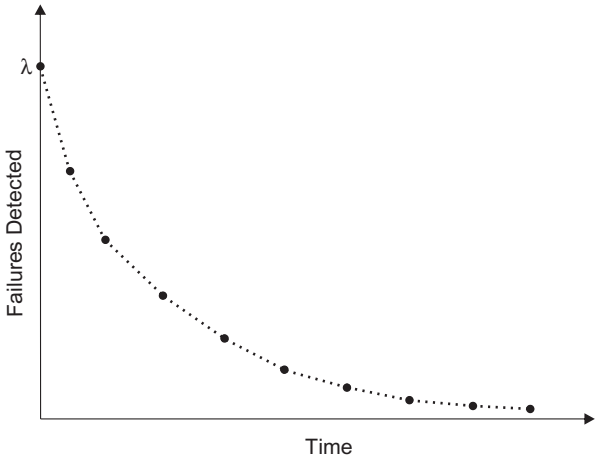


Figure 6.1. A model of failure represented by the exponential failure function (Laplante, 2003).

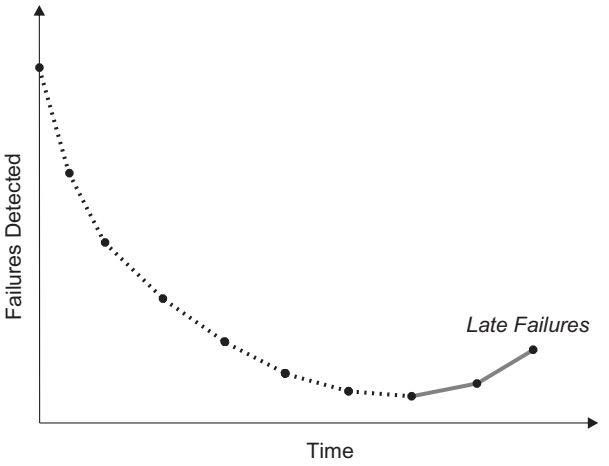


Figure 6.2. A software failure function represented by the bathtub curve (Laplante, 2003).

Another common failure model is given by the "bathtub curve" shown in Figure 6.2. Brooks notes that while this curve is widely used to describe the failure function of hardware and mechanical components, it might also be useful in describing the number of errors found in a software product (Brooks, 1995). This is particularly valid with embedded systems that have a long lifetime (even 10–30 years), and the software is updated (repaired/enhanced) numerous times during the lengthy period.

The interpretation of this failure function is apparent for hardware and mechanics: a certain number of product units will fail early due to manufacturing defects. Later, the failure intensity will increase again as the hardware/mechanics ages and wears out. But software does not wear out. Therefore, if software systems really seem to fail according to the bathtub curve, then there has to be some plausible explanation.

It is understandable that the largest number of errors will be found early in a software product's life cycle, just as the exponential failure model indicates. But why would the failure intensity increase much later? There are at least three possible explanations:

1. The failures are due to the effects of patching the software (making quick corrections to the code without designing them properly) for various reasons.
2. Late software failures are actually due to wearing of the underlying hardware or possible sensors/actuators.
3. As users master the basic software functions and begin to expose and strain advanced features, it is possible that certain inadequately tested functionality is eventually beginning to be used.

Empirical failure models are used commonly to make rough predictions of software failures during the entire operating phase. As the operating environments for the software may vary drastically in embedded applications, the randomness of a practical environment will affect the failure rate in an unpredictable way (Teng and Pham, 2006). Hence, the λ factor of Equation 6.2 should be a random variable.

Often, the traditional quality measures of mean time to first failure (MTFF) or mean time between failures (MTBF) are used to stipulate reliability in the software requirements specification. This approach to failure definition places great importance on the effective elicitation and specification of functional requirements, because the requirements also define the possible software failures.

Furthermore, real-time software execution is very sensitive to initial conditions and the external data driving it. What appear to be random failures are actually repeatable. The problem in finding and fixing these problems before a design is released, or even if the problem emerges once the embedded software is in use, is the difficulty of doing the detective work needed to discover first the particular conditions and second the data sequences that triggered the fault to become a failure. The longer a software system runs, the more likely it becomes that such a fault will be executed.

Correctness of software (Mills, 1992) is closely related to software reliability, and the terms may sometimes be used interchangeably. The fundamental difference is that even a minor deviation from the requirements is strictly considered a failure and hence means the software is incorrect. However, a system may still be deemed reliable if only minor deviations from the requirements are experienced. As widely known, such minor deviations are rather common in many software products, because typical software can only be tested partially, and often just a small proportion of the actual input space is explored statistically. In real-time systems, correctness incorporates both correctness of outputs, as well as deadline satisfaction, as discussed in Chapter 1.

Performance of software (Caprihan, 2006) is an explicit measure of some required behavior. A general methodology for measuring algorithmic performance is based on computational complexity theory (Goldreich, 2008). Alternatively, a simulation model of the real-time system might be built with the actual purpose of estimating performance. The most accurate approach, though, involves directly timing the behavior of the completed system with a logic analyzer or specific performance analysis tools.

Usability, which is often referred to as ease of use or user friendliness, is a measure of how easy and comfortable the software is for humans to use (Nielsen, 1993). This software quality is an elusive one. Properties that make an application user friendly to novice users are often very different from those desired by expert users or the software designers themselves. Demonstrative prototyping can increase the usability of a software system because, for instance, user interfaces can be evaluated and fine-tuned by a group of end users of the final product.

Usability is often difficult to quantify, although it may be easy to argue that some system is not usable. However, qualitative feedback from users and individual problem reports can be used in most cases for evaluating usability. Such general issues as user training time and readability of user documentation are possible measures of usability (Bernstein and Yuhas, 2005).

Interoperability refers to the ability of the software to coexist and cooperate with other relevant software. It is especially important in component-based software development, software reuse, and network-based software systems (Wileden and Kaplan, 1999). For example, in real-time applications, the software must be able to communicate with various devices using standard bus structures and protocols. Interoperability is usually straightforward to achieve if the decision to communicate is made before the software is designed—it is much more laborious to attain afterwards.

A concept related to interoperability is that of an open system (Dargan, 2005). An open system is an extensible collection of independently written applications that cooperate to function as an integrated system. Open systems differ from open source code, which is source code that is made available to the global user community for evolutionary improvement, extension, and correction, provided that the terms of the associated license are honored. An open system allows the addition of new functionality by independent parties through the use of standard interfaces whose detailed characteristics are published. Any applications developer can then take advantage of these interfaces, and thereby create software that can communicate using the interface. Open systems let different applications written by different organizations interoperate. For example, there are open standards for automotive (AUTOSAR, Automotive Open System Architecture), building automation (BAS, Building Automation System), and railway vehicle (IEEE Std 1473-L) systems. Interoperability can be measured in terms of compliance with relevant open system standards.

Maintainability is related to the anticipation of change that should guide the software engineer throughout the development project. A software system in which changes are relatively easy to make has a high level of maintainability; this is connected directly to the readability and understandability of the program code and associated documentation (Aggarwal et al., 2002). In the long run, design for change will significantly lower software life cycle costs and lead to an enhanced reputation for the software engineer, the software product, and the corresponding organization. Some embedded software products are maintained even for a few decades, and, therefore, the issue of maintainability is of particular importance in such cases.

Maintainability can be broken down into two contributing properties: evolvability and repairability. Evolvability is a measure of how easily the system can be changed to accommodate new features or modification of existing features. Furthermore, software is repairable if it allows for the fixing of all defects with a reasonable effort.

Measuring these qualities of software is not always easy or even possible, and often is based on anecdotal observation only. This means that changes and

the cost of making them should be tracked over time. Collecting such history data has a twofold purpose. First, the costs of maintenance can be compared with other similar systems for benchmarking and project management purposes. Second, the information can provide experiential learning that will help to improve the overall software development process, as well as the skills of software engineers.

Portability of software is a measure how easily the software can be made to run in different environments. Here, the term "environment" refers to the hardware platform on which the software runs, the real-time operating system used, or other system/application software with which the particular software is expected to interact. Because of the I/O-intensive hardware with which the software closely interacts, special care must be taken in making embedded software portable.

Hardware portability is achieved through a deliberate design strategy in which hardware-dependent code is confined to the fewest code units as possible (such as device drivers). This strategy can be achieved using either procedural or object-oriented programming languages and through structured or object-oriented design approaches. Both of these are discussed throughout the text.

On the other hand, portability of real-time operating systems or other system programs means usually the adoption of some standard application program interface (API) (Shinjo and Pu, 2005). This is commonly associated with potential overhead caused by the standards-prescribed interface. In this sense, portability may degrade the achievable real-time performance.

Also, portability is difficult to measure, other than through anecdotal observation. Person-months required to move the software to a new environment is a usual measure of this property. But this cannot be known before the actual moving effort.

Verifiability of software qualities refers to the degree to which various qualities, including all of those previously introduced, can be verified. In real-time systems, verifiability of deadline satisfaction (a form of performance) is of the utmost importance. This topic is discussed further in Chapter 7.

One common technique for increasing verifiability is through the insertion of special program code that is intended to monitor certain qualities, such as performance or correctness. Rigorous software engineering practices and the effective use of an appropriate programming language can also contribute to verifiability.

Measurement or prediction of software qualities is essential throughout the whole software life cycle. Therefore, this activity should be integrated seamlessly into the software development process. A summary of the software qualities just discussed and possible ways to measure them is given in Table 6.2.

Today, in the "embedded systems era," the emphasis on desirable software qualities has shifted gradually from *correctness* to *reliability* and *maintainability* (Aggarwal et al., 2002). A further emphasis is on the need to increase the

TABLE 6.2. Software Qualities and Possible Means for Measuring Them

Software Quality	Possible Measurement Approach
Reliability	Probabilistic measures, MTFF, MTBF, heuristic measures
Correctness	Probabilistic measures, MTFF, MTBF
Performance	Algorithmic complexity analysis, simulation, direct measurement
Usability	User feedback from surveys and problem reports
Interoperability	Compliance with relevant open standards
Maintainability	Anecdotal observation of resources spent
Portability	Anecdotal observation of resources spent
Verifiability	Insertion of special monitoring code

productivity of software developers, due to the growing complexity of software products and need for shorter time-to-market. The object-oriented design approach, to be discussed in Section 6.4, may help address this productivity challenge (Siok and Tian, 2008).

6.2 SOFTWARE ENGINEERING PRINCIPLES

Software engineering has been criticized for not having the same kind of theoretical foundation as older engineering disciplines, such as electrical, mechanical, or civil engineering. While it is true that only a few formulaic principles exist, there are several fundamental rules that form the basis of sound software engineering practice. The following subsection describes the most general and prevalent principles that are particularly applicable in the design and implementation phases of real-time software.

6.2.1 Seven Principles from Rigor and Formality to Traceability

Because software development is a creative human activity related to problem solving, there is an inherent tendency toward using informal *ad hoc* techniques in software specification, design, and coding. Nevertheless, a purely informal approach is contrary to "best software engineering practices." It should be pointed out, however, that the best practices are actually dependent on the application size as well as application type (Jones, 2010), and also on the size of the development organization (Jantunen, 2010).

Rigor in software engineering requires the use of mathematical techniques. *Formality*, on the other hand, is a higher form of rigor in which precise and unambiguous engineering approaches are used. In the case of real-time systems, strict formality would further require that there be an underlying algorithmic approach to the specification, design, coding, and documentation of the software. Due to insuperable difficulties in creating a pure algorithmic approach, semiformal and informal approaches are needed to complement

individual formal approaches. For instance, certain parts of the design document can be formal while most others are semiformal.

Separation of concerns is an effective divide-and-conquer strategy practiced by software engineers to manage miscellaneous problems related to complexity. There are various ways in which separation of concerns can be achieved. In terms of software design and coding, it is used in object-oriented design and in modularization of procedural code. Moreover, there may be separation in time, for example, developing an appropriate schedule for a collection of periodic computing tasks with different execution periods.

Yet another way of separating concerns is in dealing with individual software qualities. For instance, it may be helpful to address the fault tolerance of a system only while ignoring other qualities for some time. However, it must be remembered that many of the software qualities are actually interrelated, and it is often impossible to improve one without deteriorating another. Hence, a project-specific compromise is typically needed.

Modularity is commonly achieved by grouping together logically related elements, such as statements, procedures, variable declarations, and object attributes, in an increasingly fine-grained level of detail (see Fig. 6.3). Modular design involves the decomposition of software behavior in encapsulated software units, and can be achieved with both procedural and object-oriented

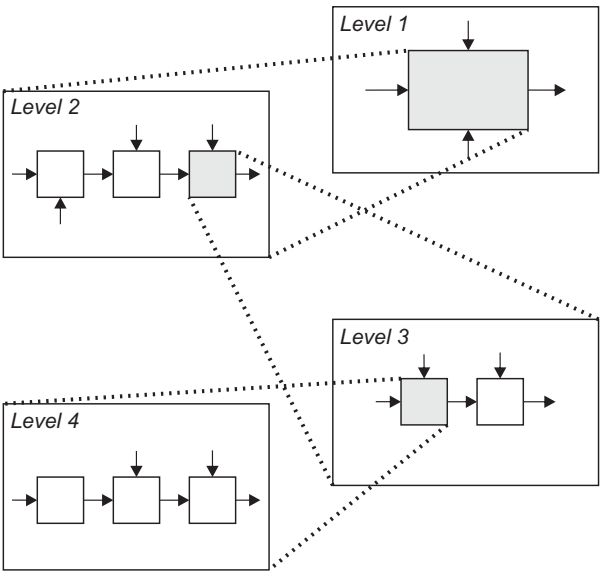


Figure 6.3. Modular decomposition of code units. The arrows represent inputs and outputs in the procedural paradigm. In the object-oriented paradigm, they represent associations. The boxes represent encapsulated data and procedures in the procedural paradigm. In the object-oriented paradigm, they represent classes (Laplante, 2003).

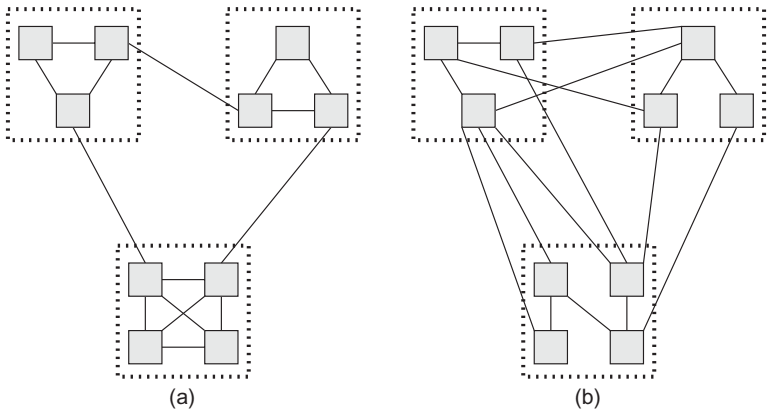


Figure 6.4. Software structures with (a) high cohesion and low coupling, and (b) low cohesion and high coupling. The inside squares represent statements or data; connecting lines indicate functional dependency.

programming languages. The main goal of modularity is high cohesion and low coupling of the software structure. With respect to the code units, cohesion represents intramodule connectivity and coupling represents intermodule connectivity. Cohesion and coupling can be illustrated as in Figure 6.4, which depicts software structures with high cohesion and low coupling (left), as well as low cohesion and high coupling (right). Cohesion relates to the relationship of the elements within a module. High cohesion implies that each module represents a single part of the problem solution. Therefore, if the system ever needs modification, then the part that needs to be modified exists in a single place, being easier and less error prone to change.

Constantine and Yourdon identified seven levels of cohesion in the order of increasing strength (Pressman, 2009):

1. ***Coincidental.*** Parts of a module are not related at all, but simply bundled into a single module.
2. ***Logical.*** Parts that perform similar tasks are put together in a joint module.
3. ***Temporal.*** Tasks that execute within the same time span are brought together.
4. ***Procedural.*** The elements of a module make up a single control sequence.
5. ***Communicational.*** All elements of a module act on the same area of a data structure.
6. ***Sequential.*** The output of one part in a module serves as input for another part.
7. ***Functional.*** Each part of a module is necessary for the execution of a single function.

This above list could be used when designing the contents of specific software modules; it brings valuable insight to the heuristic module-creation process. Modules should not be created solely by "grouping together logically related elements"—as is usually done. But there are multiple reasons to group individual elements together.

Coupling relates to the relationships between the modules themselves. There is a great benefit in reducing coupling so that changes made to one code unit do not propagate to others; they are said to be hidden. This principle of "information hiding," also known as Parnas partitioning, is the cornerstone of all software design and will be discussed in Section 6.3.1 (Parnas, 1979). Low coupling limits the effects of errors in a specific module (lower "ripple effect") and reduces the likelihood of data-integrity problems. In some cases, however, high coupling due to time-critical control structures may be necessary. For example, in most graphical user interfaces, control coupling is unavoidable, and indeed desirable.

Coupling has been characterized by six levels in the order of increasing strength:

1. ***None.*** All modules are completely unrelated.
2. ***Data.*** Every argument is either a simple argument or data structure in which all elements are used by the called module.
3. ***Stamp.*** When a data structure is passed from one module to another but that module operates on only some of the data elements of the whole structure.
4. ***Control.*** One module explicitly controls the logic of the other by passing an element of control to it.
5. ***Common.*** If two modules both have access to the same global data.
6. ***Content.*** One module directly references the contents of another.

To further illustrate both coupling and cohesion, consider the class structure diagram (object-oriented design approach) shown in Figure 6.5; the figure illustrates two interesting points. The first is the clear difference between the same system embodying low coupling and high cohesion versus high coupling and low cohesion. The second point is that the proper use of visual design techniques can positively influence the eventual design outcome.

Anticipation of change is another important principle in software design. As has been mentioned, software products are subject to frequent change either to support new hardware or software requirements or to repair defects. A high maintainability level of the software product is one of the hallmarks of outstanding commercial software.

Developers of embedded software know that their systems are subject to changes in hardware, algorithms, and even application. Therefore, these systems must be designed in such a way as to facilitate changes without degrading considerably the other desirable properties of the software. Anticipation of

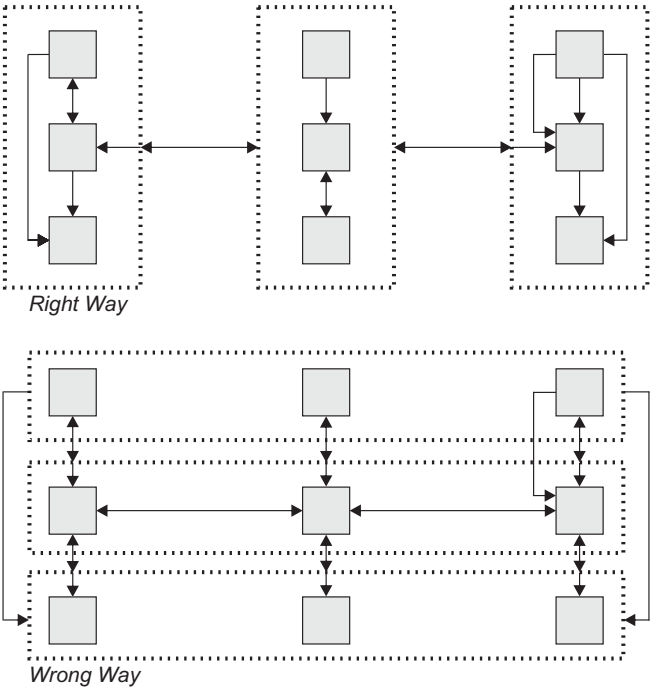


Figure 6.5. Coupling and cohesion. The right way: low coupling and high cohesion. The wrong way: high coupling and low cohesion.

change can be achieved in the software design through the adoption of an appropriate software life cycle model and corresponding design methodologies, as well as through appropriate project-management practices and associated training efforts.

In solving a problem, the principle of *generality* can be stated as the intent to look for a more general problem that may be hidden behind it. As an obvious example, designing an elevator control system for a low-end apartment building is less general than designing it to be adaptable to various hotels, offices, shopping centers, and apartment buildings.

Generality can be achieved through a diverse number of approaches associated with procedural and object-oriented paradigms. For example, Parnas' information hiding can be used with procedural languages. Extensive parameterization is another commonly used approach for providing generality to software. In object-oriented software, generalization is achieved by applying certain design principles and through the use of architectural and design patterns. Although generalized solutions may be more costly in terms of the problem at hand, in the long run, the extra costs of a generalized solution may be worthwhile. Nonetheless, these extra costs might affect real-time performance, which is always a difficult issue to handle. Moreover, a manager of a

specific development project might ask a relevant question: "Why should *this* project pay some costs of possible future projects in advance?" This is, indeed, a good question and should be addressed by the steering group of that particular project; there may be a conflict between short-term and long-term goals.

Incrementality involves a software-engineering approach in which progressively larger increments of the desired product are developed. Each increment provides additional functionality, which brings the unfinished product closer to the final one. Each increment also offers an opportunity for demonstration of the product to the customer for the purposes of gathering supplementary requirements and refining the look and feel of the product or its user interface, for example. In reality, however, some advanced sets of increments have even been delivered to the customer as "the product" due to sizeable delays in the development project. This usually leads to serious problems and shall be strictly avoided.

Traceability is concerned with the relationships between requirements, their sources, and the system design. Regardless of the life cycle model used, documentation and code traceability are truly important. A high level of traceability ensures that the software requirements flow down through the design and program code, and then can be traced back up at every stage of the development process. This would ensure, for instance, that a coding decision can be traced back to a design decision to satisfy a corresponding requirement.

Traceability is particularly important in embedded systems, because specific design and coding decisions are often made to satisfy rather unique hardware constraints that may not be directly associated with any higher-level requirement. Failure to provide a traceable path from such decisions through the requirements can lead to substantial difficulties in extending and maintaining the system.

Generally, traceability can be obtained by providing consistent links between all documentation and the software code. In particular, there should be links:

- From requirements to stakeholders who proposed these requirements.
- Between dependent requirements.
- From the requirements to the design.
- From the design to associated code segments.
- From requirements to the test plan.
- From the test plan to individual test cases.

One way to create these links is through the use of an appropriate numbering system throughout the documentation. For instance, a requirement numbered 3.1.1.2 would be linked to a design element with a similar number (the numbers do not have to be the same as long as the annotation in the document guarantees traceability). In practice, a traceability matrix is constructed to help cross reference the documentation and associated code elements (Table 6.3).

TABLE 6.3. A Traceability Matrix Sorted by Requirement Number

Requirement Number	Design Document Reference Number(s)	Test Plan Reference Number(s)	Code Unit Name(s)	Test Case Number(s)
3.1.1.1	3.1.1	3.1.1.1	Task_A	3.1.1.A
	3.2.4	3.2.4.1		3.1.1.B
		3.2.4.3		3.1.1.C
3.1.1.2	3.1.1	3.1.1.2	Task_B	3.1.1.A
				3.1.1.D
3.1.1.3	3.1.1.3	3.1.1.3	Task_C	3.1.1.B
				3.1.1.E

The matrix is constructed by listing the relevant software documents and the code units as columns, and then each software requirement in the rows. Traceability to the stakeholders related to certain requirements or to relevant standards and regulations could also be added as columns in Table 6.3.

Constructing the traceability matrix in a spreadsheet software package allows for providing multiple matrices sorted and cross-referenced by each column as needed. For example, a matrix sorted by test case numbers would be an appropriate appendix to the test plan. The traceability matrices are updated at each step in the software life cycle. For instance, the column for the code unit names (e.g., procedure names or object classes) would not be added until after the code is developed. A way to foster traceability between code units is through the use of data dictionaries, which are described later.

Finally, a mapping (*positive effect*) from the individual software-engineering principles, just discussed, to the desired software qualities of Table 6.2 is sketched in Table 6.4. Some of these mappings are explicit, while others are more implicit. Interestingly, the software quality of maintainability appears to be improvable by all the seven principles. Of the software engineering principles, modularity seems to be a particularly strong one, since it can improve all the software qualities except "usability" and "verifiability."

6.2.2 The Design Activity

The design activity is involved in identifying the components of the software design and their interfaces from the software requirements specification. The principal artifact of this activity is the Software Design Description (SDD). In the same way as the IEEE Std 830–1998 (discussed in Section 5.1.2) provides a sound framework for requirements engineering documents, a recently revised standard, IEEE Std 1016–2009, specifies requirements on the information content and organization for software design descriptions (IEEE, 2009). According to the standard, "SDD is a representation of a software design that is to be used for recording design information, addressing various design concerns, and communicating that information to the design's stakeholders."

TABLE 6.4. A Mapping Matrix from Software Engineering Principles to Software Qualities in Real-Time Applications

Principles/Qualities	Reliability	Correctness	Performance	Usability	Interoperability	Maintainability	Portability	Verifiability
Rigor and Formality	×	×	×			×	×	×
Separation of Concerns	×	×	×	×	×	×	×	×
Modularity	×	×	×		×	×	×	
Anticipation of Change					×	×	×	
Generality					×	×	×	
Incrementality	×	×		×		×		
Traceability	×	×				×	×	

During the design phase, a team of real-time systems engineers creates a detailed software design and acquires a formal acceptance for it. That involves the following tasks from the initial Architecture Design (Taylor et al., 2010) to the Final Design Review (Hadar and Hadar, 2007):

1. *Architecture Design*
   - Performing hardware/software trade-off analysis leading to hardware–software partitioning.
   - Making the determination between centralized or distributed processing schemes.
   - Designing interfaces to external components.
   - Designing interfaces between internal components.
2. *Control Design*
   - Determining concurrency of execution.
   - Designing principal control strategies.
3. *Data Design*
   - Determining storage, maintenance, and allocation strategy for data.
   - Designing database structures and handling routines.
4. *Functional Design*
   - Designing the start-up and shutdown processing.
   - Designing algorithms and functional processing.
   - Designing error processing and error-message handling.
   - Conducting performance analyses of critical functions.
5. *Physical Design*
   - Determining physical locations of software components and data.
6. *Test Design*
   - Designing any test software identified in test planning.
7. *Documentation Design*
   - Creating possible support documentation, such as the Operator's Manual, User's Manual, Programmer's Manual, and Application Notes.
8. *Intermediate Design Reviews* (→ internal acceptances)
   - Conducting internal design reviews.
9. *Detailed Design*
   - Developing the detailed design for all software components.
   - Developing the test cases and procedures to be used in the formal acceptance testing.
10. *Final Design Review* (→ organizational acceptance)
    - Documenting the software design in the form of the SDD.
    - Presenting the SDD at a formal design review for examination and criticism.

This is an intimidating set of substantial tasks that is further complicated by the fact that many of them must occur in parallel or be iterated several times. There is obviously no algorithm, per se, for conducting these tasks. Instead, it takes many years of practicing, learning from the experience of others, and good judgment to guide the software engineer heuristically through this maze of individual design tasks. In such effort, collective knowledge of a matured development organization would be of significant aid.

Two alternative methodologies, procedural and object-oriented design, which are related to structured analysis and object-oriented analysis, respectively, can be used to perform the design activities based on the software requirements specification. Both methodologies seek to arrive at a physical software model containing small, detailed components.

6.3 PROCEDURAL DESIGN APPROACH

Procedural design methodologies, like structured design, involve top-down and bottom-up approaches centered on procedural programming languages, such as the popular C language. The most common of these approaches utilize effective design decomposition via Parnas partitioning (Parnas, 1979).

6.3.1 Parnas Partitioning

Software partitioning into multiple software units with low external coupling and high internal cohesion can be achieved through the principle of *information hiding*. In this technique, a list of difficult design decisions or things that are likely to change is first prepared. Individual modules are then designated to hide the eventual implementation of each design decision or a specific feature from the rest of the system. Thus, only the functionality of each module is visible to other modules, not the method of implementation. Changes in these modules are therefore not likely to affect the rest of the system.

This form of functional decomposition is based on the notion that some aspects of a system are fundamental and remain constant, whereas others are somewhat arbitrary and likely to change. Moreover, it is those arbitrary aspects that often contain the most valuable design information. Arbitrary facts are hard to remember and usually require lengthy descriptions; hence, they are typical sources of documentation complexity.

The following five steps can be used to implement a good design that embodies information hiding:

1. Begin by characterizing the likely changes (consider different time horizons of the life cycle) and their effects.
2. Estimate the probabilities of each type of change.
3. Organize the software to confine likely and significant changes to a minimum amount of code.

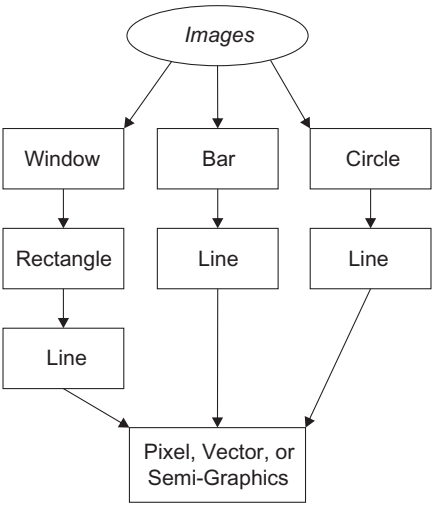


Figure 6.6. Parnas partitioning of graphics rendering software.

4. Provide an "abstract interface" that abstracts from the potential differences.
5. Implement "objects," that is, abstract data types and modules that hide changeable data and other structures.

These steps reduce intermodule coupling and increase intramodule cohesion. Parnas also indicated that although module design is easy to describe in textbooks, it is difficult to achieve in practice. He suggested that extensive real-world examples are needed to illustrate the point correctly (Parnas, 1979).

As an example, consider a portion of the display function of a graphics subsystem associated with an elevator monitoring system and depicted in hierarchical form in Figure 6.6. Such monitoring systems are used in supervision centers and can also be available in large lobbies for displaying the elevator traffic. It consists of color graphics that must be displayed (e.g., a representation of multiple elevator shafts, animated elevator cars, and registered calls) and are essentially composed from bars, rectangles, and circles. Different objects can naturally reside in different display windows. The actual implementation of bars, rectangles, and circles is based on the composition of line-drawing calls. Thus, line drawing is the most basic (hardware-dependent) function in this application. Whether the actual graphics controller is based on pixel, vector, or even semi-graphics does not matter; only the line-drawing routine with standard software interfaces needs to be changed. Hence, the hardware dependencies have been isolated to a single code unit.

Parnas partitioning hides the implementation details of software features, design decisions, low-level hardware drivers, and so forth, in order to limit the

scope of impact of future changes or corrections. Such a technique is especially applicable and useful in embedded systems; since they are so directly tied to hardware, it is important to partition and localize each implementation detail with a particular hardware interface. This approach allows easier modifications due to possible hardware interface changes, and minimizes the amount of code affected.

If in designing the software modules, increasing levels of detail are deferred until later (subordinate code units), then the software design approach is called *top-down*. If, on the other hand, the design detail is dealt with first and then increasing levels of abstraction are used to encapsulate those details, the approach is obviously *bottom-up*.

In Figure 6.6, it would be possible to design the software by first describing the characteristics of various components of the system and the functions that are to be performed on them, such as opening, sizing, and closing windows. Then the window functionality could be broken down into its constituent parts, such as rectangles and text. These could be subdivided even further, that is, all rectangles consist of lines, and so on. The top-down refinement continues until the lowest level of detail needed for code development has been reached.

Alternatively, it is possible to begin by encapsulating the details of the most volatile part of the system, the hardware implementation of a line or pixel, into a single code unit. Then working upward, increasing levels of abstraction are created until the system requirements are satisfied. This is a bottom-up approach to software design. In many real-world applications, however, the software design process contains both top-down and bottom-up sections.

6.3.2 Structured Design

Structured design (SD) is the companion methodology to structured analysis. It is a systematic approach concerned with the specification of the software architecture and involves a number of strategies, techniques, and tools. SD supports a comprehensive but easy-to-learn design process that is intended to provide high-quality software and minimized life cycle expenses, as well as to improve reliability, maintainability, portability, and overall performance of software products. Structured analysis (SA) is related to SD in the same way as a requirements representation is related to the software architecture, that is, the former is functional and flat, but the latter is modular and hierarchical.

The transition mechanisms from SA to SD are purely manual and involve substantial problem-solving effort in the analysis and trade-offs of alternative approaches. Normally, SD proceeds from SA in the following manner. Once the context diagram (CD) is first created, a hierarchical set of data flow diagrams (DFDs) is developed. DFDs are used to partition system functions and document that partitioning inside the specification. The first DFD, the level 0

diagram, illustrates the highest level of system abstraction. Subdividing processes to lower and lower levels until they are ready for detailed design renders further DFDs with successive levels of increasing detail. This heuristic decomposition process is called downward leveling, and it corresponds to the top-down design approach. Nevertheless, the bottom-up approach is also used commonly when developing DFDs. In that case, the composition process is called upward leveling. A problem-driven mixture of downward and upward leveling is preferred by most software designers (Yourdon, 1989).

In the CD (see Fig. 5.13), rectangles represent terminators that model the environment boundary. They are labeled with a noun phrase that describes the agent, device, or system from which data enters or to which it exits. Each process (or data transformation) depicted by a circle in CD/DFDs is labeled as a verb phrase describing the operation to be performed on the data, although it may be labeled with the name of a system or specific operation that manipulates the data as well. Solid arrow lines are used to connect terminators to processes and between processes to indicate the flow of data through the system. Each arrow line is labeled with a noun phrase that describes the data it carries. Moreover, parallel lines indicate data stores, which are labeled by a noun phrase naming the database, file, or repository where the system stores data (either simple data elements or a more complex data structure). A data store is passed to lower levels of hierarchy by connecting it with the corresponding process.

Each DFD should preferably have between five and nine processes (Yourdon, 1989). The descriptions for the lowest level processes are called process specifications, or P-SPECs, and are expressed in either decision tables or trees, pseudocode, or structured English, and are used to describe the detailed algorithms and operational logic of the actual program code. Yourdon stated that the purpose of structured English is "to strike a reasonable balance between the precision of a formal programming language and the casual informality and readability of the English language" (Yourdon, 1989). Figure 6.7 illustrates a typical evolution path from the context diagram through data flow diagrams to process specifications.

Example: Highest-Level DFD of the Elevator Control System

Consider again the elevator control system discussed in Section 3.3.8 and refer to its context diagram given in Figure 5.13. The associated level 0 DFD is shown in Figure 6.8. It contains five individual processes and three shared data stores ("global memory"). To create such a DFD, a thorough view/understanding of the elevator control system to be designed is developed gradually; hence, the resulting DFD is a refined outcome of a longish iterative process consisting of both top-down and bottom-up stages.

It should be noted that this DFD includes also a few control flows (dashed arrow lines), which are used to activate individual processes. These

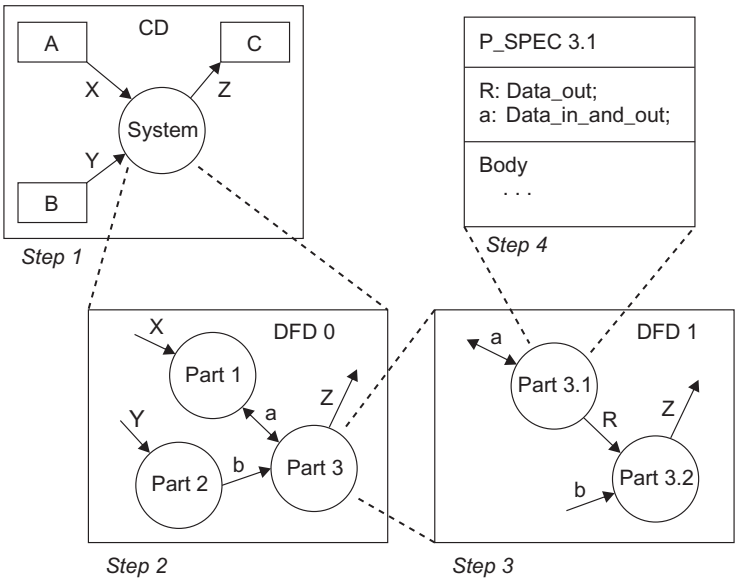


Figure 6.7. Evolution path from the context diagram to level 0 DFD to level 1 DFD, and finally to a P-SPEC.

activations are related to hardware interrupts and certain internal events, as outlined below:

1. ***Communications.*** Activated when the group dispatcher sends a request to communicate.
2. ***Update Destination.*** Periodic activation (75-ms timer interrupt).
3. ***Perform Runs.*** Activated primarily by Process 2 (also by the door and door zone interrupts) when there is a need to start a floor-to-floor run or to stop at the next possible floor (or perform some critical door control actions).
4. ***Supervise Operation.*** Periodic activation (500-ms timer interrupt).
5. ***Connect to Service Tool.*** Activated when an elevator technician presses some key of the service tool.

Notice that here the hardware interrupts were not included in the context diagram, but appear, for the first time, in this level 0 DFD.

To complement the DFDs, entity relationship diagrams (ERDs) are often used to define explicit relationships between stored data objects in the system. Hence, the entities of the ERD are modeling information concepts of the software application.

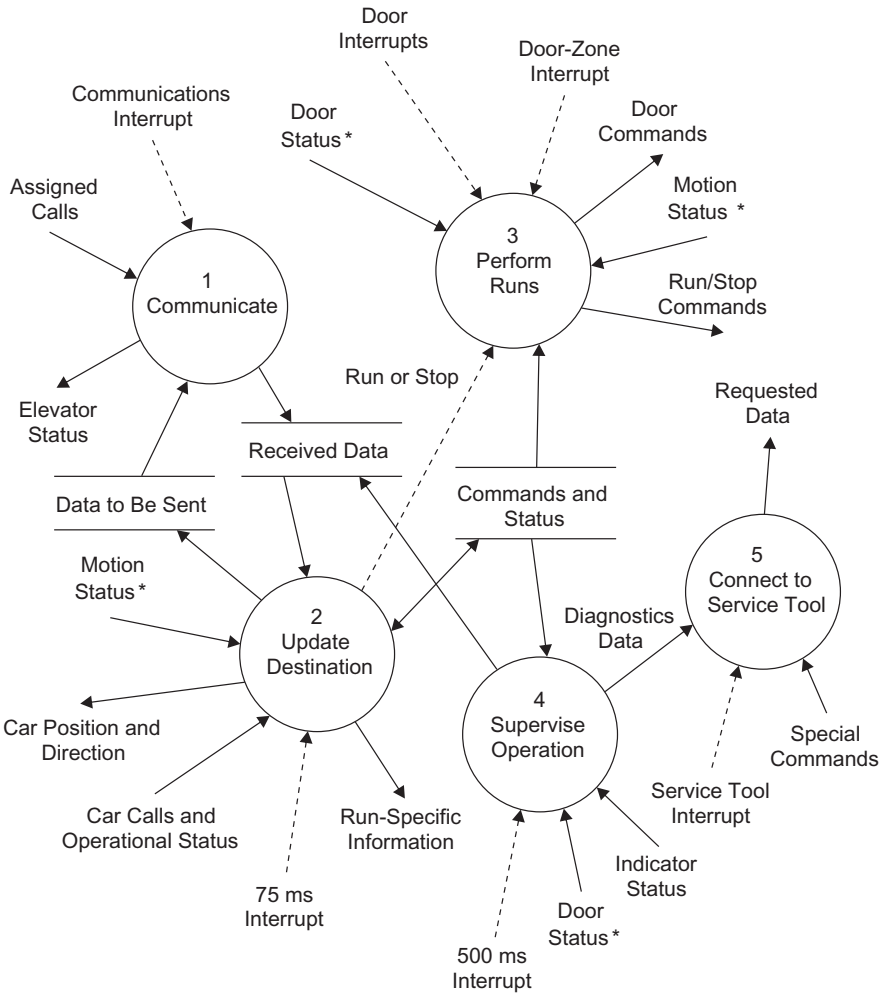


Figure 6.8. Level 0 DFD for the elevator control system. * This incoming data flow is connected to two processes.

Furthermore, a data dictionary (DD) is an essential component of the structured design, and includes entries for data flows, control flows, data stores, and buffers appearing in DFDs and control flow diagrams (to be discussed shortly). In addition, also the entries of ERDs should be included in the DD. Each entry is identified typically by its name, entry type, range, resolution, unit, location, and so forth. The data dictionary is organized alphabetically for ease of use. Other than that, there is no standard format, but every design element must have a descriptive entry in it. Most SA/SD CASE tools support the data-dictionary feature in addition to the diagrams mentioned above.

Example: A Sample Data-Dictionary Entry

For the elevator control system, one DD entry might appear as follows:

***Name*:**	Car call table
***Alias*:**	Car_calls
***Entry type*:**	Data store
***Description*:**	An integer vector containing the car call status for each possible destination floor
***Values*:**	"1" corresponds to "car call registered" and "0" represents "no car call," whereas other values are illegal
***Location*:**	Level 2.1 DFD

Additional "Location" information will be added as the program code is developed. In this way, data dictionaries help to provide traceability between design/code elements.

There are, however, apparent problems in using the standard structured analysis and structured design (SA/SD) to model *real-time* systems, including difficulty in modeling time dependencies and events. Consequently, concurrency is not adequately depictable using this form of SA/SD.

Another problem may arise already when creating the context diagram. Control flows are not easily translatable into code because they are hardware or operating-system dependent. In addition, such a control flow does not really make sense since there is no connectivity between portions of it, a condition known as "floating." As a representative example, the DFD of Figure 6.8 has altogether six floating control flows associated with hardware interrupts.

Details of the underlying hardware need to be known for further modeling of certain processes. For example, what happens if the communications hardware (interacts with Process 1) is changed? Or if another service tool with a different kind of keypad or display panel is taken in use (interacts with Process 5)? In such cases, the hardware-originated changes would need to propagate into the level 1 DFD for the corresponding process, any subsequent levels, and, ultimately, into the program code.

Making and tracking changes in structured design is fraught with danger, and hence requires special attention. Besides, a single change could mean that significant amounts of code would need to be rewritten, recompiled, and properly linked with the unchanged code to make the system work.

As expressed above, the standard SA/SD methodology is not well equipped for dealing with time, obviously, because it is a data-oriented and not a control-oriented approach. In order to address this shortcoming, the SA/SD method was extended by allowing for the addition of *control flow analysis*. This extension of SA/SD is called real-time SA/SD (SA/SD/RT). To accomplish this, the

following artifacts were added to the standard approach: dashed arrow lines to indicate the flow of control messages and dashed parallel lines indicating message buffers. More specifically, dashed arrow lines can be either triggering events, such as hardware interrupts, or specific control flows between processes. A control flow can carry a single message (such as "activate" or "deactivate"), or it can form a structure of multiple messages. A message buffer, on the other hand, is a data store that contains explicit control characteristics, since it can behave autonomously as a stack or queue. Furthermore, a dashed circle represents a control transformation in Ward-Mellor SA/SD/RT (Ward and Mellor, 1985), and it can be used conveniently to sequence data flow diagrams. For that purpose, Mealy-type finite state machines are commonly used to define the encapsulated state sequence and corresponding process activations.

The addition of the control artifacts allows, in principle, for the creation of a diagram containing solely control artifacts called a control flow diagram (CFD). These CFDs can be further decomposed into C-SPECs (control specifications), which can then be described by finite state machines. However, the control and data flow diagrams are usually combined as shown in Figure 6.8. The important relationship between the control and process models is depicted in Figure 6.9.

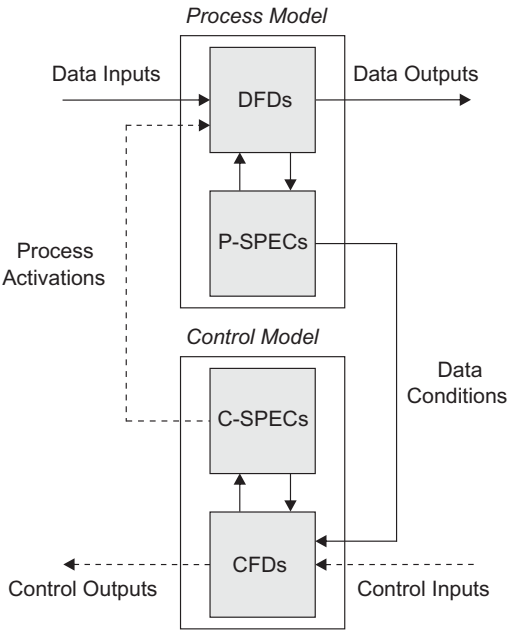


Figure 6.9. The relationship between control and process models (Laplante, 2003).

6.3.3 Design in Procedural Form Using Finite State Machines

One of the advantages of using finite state machines in the software requirements specification and later in the software design is that they can be easily (or even automatically) converted to code and test cases. For instance, consider the control of the elevator door. The tabular representation of the state transition function (see Table 5.2), which describes the system's high-level behavior rigorously, can be easily transformed into a design using the generic pseudocode shown in Figure 6.10. Each procedure associated with the possible door states (Open, Closing, Closed, Opening, Nudging, Fault C, and Fault O) will be structured code that can be viewed as executing in one of any number of possible states at every instant in time. This functionality can be described conveniently by the pseudocode shown in Figure 6.11.

```
typedef states:    (state 1,...,state n); {n is # of states}
        alphabet:  (input 1,...,input n);
        table_row: array [1..n] of states;

procedure move_forward; {advances FSM one state}

var
        state: states;
        input: alphabet;
        table: array [1..m] of table_row; {m is alphabet's size}

begin
        repeat
           get(input); {read one token from input stream}
           state := table[ord(input)] [state]; {next state}
           execute_process (state);
           until input = EOF;
end;
```

Figure 6.10. A generic pseudocode that can implement the behavior of finite state machines (Laplante, 2003).

```
procedure execute_process (state: states);

begin

    case state of
    state 1: process 1; {execute process 1}
    state 2: process 2; {execute process 2}
    ...
    state n: process n; {execute process n}

end;
```

Figure 6.11. Finite-state-machine code for executing a single operational process; each process can exist in multiple states, allowing partitioning of the program code into appropriate modules (Laplante, 2003).

Moreover, the pseudocodes given in Figures 6.10 and 6.11 can be easily translated to any procedural language or even to an object-oriented one. Alternatively, the system behavior can be described with a `case` statement or nested `if-then` statements such that, given the current state and receipt of a signal, a new state is assigned. The advantage of finite-state machine design over the `case` statement alternative is, of course, that the former is more flexible and compact.

6.4 OBJECT-ORIENTED DESIGN APPROACH

As discussed in Chapter 4, object-oriented programming languages are those characterized by data abstraction, inheritance, polymorphism, and messaging. Data abstraction through a variety of objects provides facilities for effective information hiding, or encapsulation and protected variation. Inheritance allows the software engineer to define new objects in terms of previously defined ones so that the new objects inherit properties. Function polymorphism allows the programmer to define operations that behave differently, depending on the type of object involved. Moreover, messaging allows objects to communicate and invoke the methods that they support.

Object-oriented languages provide a natural environment for information hiding through encapsulation. The state, data, and behavior of objects are encapsulated and accessed only via a published interface or certain private methods. For example, in the inertial measurement system (see Fig. 5.6), it would be appropriate to design a class called "accelerometer" with attributes describing its physical implementation and methods describing its output, compensation algorithm, and so forth.

Object-oriented design is a modern approach to systems design that views the system components as objects, as well as data processes, control processes, and data stores that are encapsulated within objects. Early forays into object-oriented design were led by aims to reuse some of the better features of structured methodologies, such as the data flow and entity relationship diagrams, by reinterpreting them in the context of object-oriented languages. This can be observed also in the popular unified modeling language (UML), which became standardized in the late nineties; the latest revision of the standard is UML 2.3 that was released in May 2010.

6.4.1 Advantages of Object Orientation

Over the last decade, the object-oriented framework has gained significant acceptance within the embedded-software community. The main advantages of applying object-oriented paradigms in real-time systems are the future extensibility and reuse that can be attained, and the relative ease of future changes. Also, the productivity of programmers is potentially improved through the use of object-oriented techniques. Most software systems are subject to

near-continuous change: requirements change, merge, emerge, and mutate; target languages, platforms, and architectures change; and the way the software is employed in practice changes, too. Larman pointed out that after the initial release of a typical software product, at least half of the effort and cost is spent in modification (Larman, 2002a). This calls for flexibility and places a considerable burden on the software design: How can systems that must support such widespread change be built without compromising quality measures? There are four basic principles of object-oriented software engineering that address this question, and they have been recognized collectively as *supporting reuse*.

First recorded by Meyer, the *open-closed principle* (OCP) states that classes should be open to extension, but closed to modification (Meyer, 2000). That is, it should be possible to extend the behavior of a class in response to new or changing requirements, but modification to the source code is not allowed. While these expectations may seem at odds (particularly to those whose background is primarily in procedural languages), the obvious key is abstraction. In object-oriented systems, a superclass can be created that is fixed, but can represent unbounded variation by subclassing. This aspect is clearly superior to structured approaches in making changes, for instance, in accelerometer compensation algorithms, which would require new function parameter lists and wholesale recompilation of all modules calling that code in the structured design.

While not a new idea, Beck gave a name to the principle that any aspect of a software system—be it an algorithm, a set of constants, documentation, or logic—should exist in one and only one place (Beck, 1999). This so-called *once-and-only-once principle* (OAOOP) isolates future changes, makes the system easier to comprehend and maintain, and through the low coupling and high cohesion that the principle instills, the reuse potential increases significantly (Beck, 1999). The encapsulation of state and behavior in objects, and the ability to inherit properties between classes, allows for the rigorous application of these ideas in an object-oriented system, but is difficult to implement in structured approaches. More importantly, in structured approaches, OAOOP needs to be breeched frequently for reasons of performance, reliability, availability, and, often, for security as well.

Furthermore, the *dependency inversion principle* (DIP) states that high-level modules should not depend upon low-level modules; both should depend upon abstractions. This can be reformulated: Abstractions should not depend upon details—details should depend upon abstractions. Martin introduced this idea as an extension to the OCP with reference to the proliferation of dependencies that exist between high- and low-level modules (Martin, 1996). For example, in a structured decomposition approach, the high-level procedures reference the lower-level procedures, but changes often occur at the lowest levels. This infers that high-level modules or procedures that should be unaffected by such detailed modifications may be affected due to these dependencies. Again, consider the case where the accelerometer characteristics

change and even though perhaps only one routine needs to be rewritten, the calling module(s) may need to be modified and recompiled, too. A preferable situation would be to reverse these dependencies, as is evident in the Liskov substitution principle (LSP). The intent here is to allow dynamic changes in the preprocessing scheme, which is achieved by ensuring that all the accelerometer objects conform to the same interface, and are therefore interchangeable.

Definition: Liskov Substitution Principle

> Liskov expressed the principle of the substitutivity of subclasses for their base classes as: If for each object o_1 of type S, there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T (Liskov, 1988).

This useful principle has led to the concept of type inheritance and is the basis of polymorphism in object-oriented systems, where instances of derived classes can be substituted for each other, provided they fulfill the obligations of a common superclass.

6.4.2 Design Patterns

Developing embedded software is hard, and developing truly reusable software is even harder. Competitive software designs should be specific to the current problem, but general enough to address potential future problems and requirements. Hence, there may arise a cost-related conflict between short-term and long-term goals. Experienced designers know not to solve every problem from first principles, but to reuse solutions encountered previously, that is, they find recurring patterns and use them as a basis for new designs. This is simply an embodiment of the principle of generality.

While object-oriented systems can be designed to be as rigid and resistant to extension and modification as in any other paradigm, object-orientation has the ability to include distinct design elements that can cater to future changes and extensions. These "design patterns" were first introduced to the mainstream of software engineering practice by Gamma, Helm, Johnson, and Vlissides, and are commonly referred to as the "Gang of Four" (GoF) patterns (Gamma et al., 1994).

The formal definition of a pattern varies throughout the literature. We will use the following informal definition throughout this text.

Definition: Pattern

> A pattern is a named problem–solution pair that can be applied in different contexts, with explicit advice on how to apply it in new situations.

Our presentation is concerned with three pattern types: *architectural patterns*, *design patterns*, and *idioms*. An architectural pattern occurs across subsystems; a design pattern occurs within a subsystem, but is independent of the programming language; and an idiom is a low-level pattern that is language specific (Horstmann, 2006).

In general, every pattern consists of four essential elements:

1. A name (such as "façade")
2. The problem to be solved (such as "provide a unified interface to a set of interfaces in a subsystem")
3. The solution to the problem
4. The consequences of the solution

More accurately, the problem describes when to apply the pattern in terms of specific design problems, such as how to represent algorithms as objects. The problem may describe class structures that are symptomatic of an inflexible design. Finally, the problem section might include conditions that must be met before it makes sense to apply the pattern.

The solution, on the other hand, describes the elements that make up the design, though it does not describe any concrete design or implementation. Rather, the solution provides how a general arrangement of objects and classes solves the problem. Consider, for instance, the previously mentioned GoF patterns. They describe 23 design patterns, each being either *creational*, *behavioral*, or *structural* in its intent (see Table 6.5). This table is provided for illustration only, and it is not our intention to describe any of these patterns in detail, since they are well documented elsewhere (Gamma et al., 1994). Some patterns have evolved specifically for real-time systems, and they provide various approaches to addressing the fundamental real-time scheduling, com-

TABLE 6.5. The Original Set of Design Patterns Popularized by the "Gang of Four" (Gamma et al., 1994)

Creational	Behavioral	Structural
Abstract factory	Chain of responsibility	Adapter
Builder	Command	Bridge
Factory method	Interpreter	Composite
Prototype	Iterator	Decorator
Singleton	Mediator	Façade
	Memento	Flyweight
	Observer	Proxy
	State	
	Strategy	
	Template method	
	Visitor	

munications, and synchronization problems, for example, Douglass (2003) and Schmidt et al. (2000).

Let us consider Douglass' real-time pattern set. Douglass groups his 48 patterns into six classes (Douglass, 2003):

1. Subsystem and component architecture
2. Concurrency
3. Memory
4. Resource
5. Distribution
6. Safety and reliability

As it turns out, we have discussed many of these patterns in Chapter 3 (but without mentioning the term "pattern"). The architecture patterns include the layered architecture that is so common to real-time operating systems (see Fig. 3.2), and the virtual machine which is the underlying architecture for Java. Of the concurrency patterns, many are based on solutions that we have already discussed, for instance, "round-robin," "static priority," "dynamic priority," and "cyclic executive." Various solutions for memory allocation, buffering, and garbage collection, are included in the memory patterns. The resource patterns, on the other hand, describe solutions to the critical-section problem through the use of semaphores, and the priority inheritance and priority ceiling protocols, among others. The distribution patterns deal with the problem of a synchronous control over a set of independent processes, and incorporates solutions found in other sets, such as the GoF's observer and proxy patterns. Finally, the safety and reliability patterns give solutions to improve fault tolerance and reliability through various types of redundancy, watchdog timers, and the like, many of which we will discuss in Chapter 8. Moreover, Douglass' pattern set includes many other solutions to real-time problems in a format that is quite accessible to the developer.

A comprehensive study on available pattern collections is provided by Henninger and Corrêa (2007). They pointed out: "As the number of patterns and diversity of pattern types continue to proliferate, pattern users and developers are faced with difficulties of understanding what patterns already exist and when, where, and how to use or reference them properly." This relevant concern is based on a careful survey, where altogether 170 software development-related pattern entities with more than 2200 patterns were identified and classified. A majority of those patterns is of architectural or design type. To avoid overlooking opportunities to utilize design patterns effectively, Briand et al. proposed a methodology for semiautomating the detection of areas within UML designs that are suitable candidates for the use of design patterns (Briand et al., 2006). Such methodologies, if just available in CASE environments with high usability, could advance the use of design patterns among practitioners.

6.4.3 Design Using the Unified Modeling Language

Today, the UML is widely accepted as the de facto standard for the specification and design of software-intensive systems using the object-oriented approach. By bringing together the "best-of-breed" in diverse specification techniques, the UML has become a sophisticated family of individual languages or diagram types, and users can choose which members of the family are suitable for their particular domain. Furthermore, complete UML models consist of a collection of diagrams, as well as accompanying textual and other documentation.

The UML is a graphical language based upon the premise that any system can be composed of communities of interacting entities. Various aspects of those entities and their interaction can be described using the original set of nine diagrams: activity, class, communication, component, deployment, sequence, state machine, object, and use case. Of these UML diagrams, five depict behavioral or dynamic views (activity, communication, sequence, state-machine, and use-case), while the remaining four are concerned with structural or static aspects. With respect to real-time systems, it is the behavioral diagrams that are of particular interest, since they define what must happen in the system under consideration. Many of those original diagrams are illustrated in the extensive design case study at the end of this chapter. The principal artifacts generated when using the UML as well as their relationships are depicted in Figure 6.12.

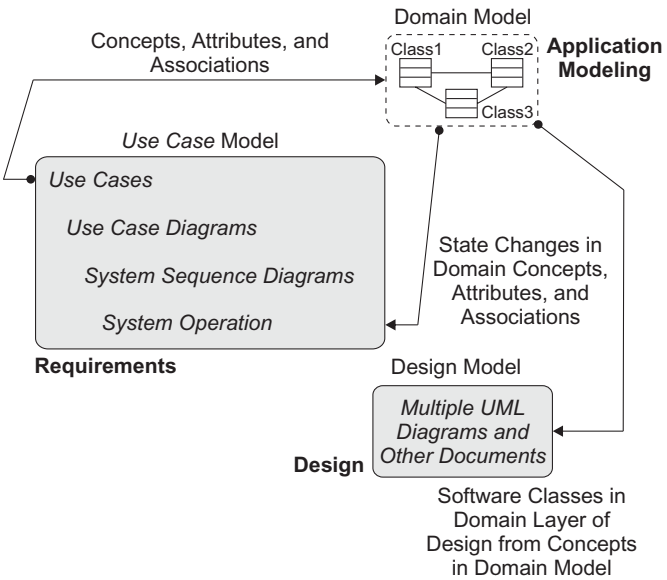


Figure 6.12. The role of UML in specification and design; adapted from Larman (2002b).

In addition to the nine diagrams mentioned above, the UML 2.2 (released in 2009) has five other diagrams (OMG Unified Modeling Language™ [OMG UML], 2009). Nonetheless, some of the numerous diagrams are partially redundant and used infrequently. All the 14 UML diagrams are introduced below in alphabetical order. For each of the diagrams, a suggested *Learning Priority* (*LP*) is given according to Ambler (2004); Ambler's *LP* has three possible levels: high, medium, and low. Although these suggestions are referring to the needs of "a business application developer," they give helpful guidelines also for real-time software developers.

***Activity Diagram* (Behavioral/General; *LP* = High):** Activity diagrams are closely related to the classical flowchart and are used for the same purpose, that is, to specify the flow of control. However, unlike flowcharts, they can model concurrent computational steps and the flow of objects as they move from state to state at different points in the flow of control. In fact, in UML 2.0 and later, the activity diagram was refashioned to be more similar to the Petri net, which is widely used in digital hardware design to conduct synchronization analysis and to identify deadlocks, race conditions, and dead states. Thus, activity diagrams are useful in modeling dynamic aspects of a real-time system.

***Class Diagram* (Structural; *LP* = High):** During system design, the class diagram defines the actual class attributes and methods implemented in an object-oriented programming language. Design pattern architectures are explored and physical requirements assessed during design. Design patterns provide guidance on how the defined class attributes, methods, and responsibilities should be assigned to objects. Physical requirements require the programmer to revisit the analysis class diagram, where new classes for the system requirements are defined. Figure 6.A10 in Appendix 1 at the end of this chapter is a design class diagram for the traffic-light control system.

***Communication Diagram* (Behavioral/Interaction; *LP* = Low):** Communication diagrams show the messages passed between objects through the basic associations between classes. In essence, they depict the dynamic behavior on static class diagrams. Communication diagrams are the most emphasized of UML interaction diagrams because of their clarity and expression of more information. The communication diagram contains classes, associations, and message flows between classes. Figures 6.A4–6.A9 in Appendix 1 at the end of the chapter are communication diagrams for the traffic-light control system.

***Component Diagram* (Structural; *LP* = Medium):** These diagrams are made up of components, interfaces, and relationships. Components represent preexisting entities. Interfaces represent the functionality of components that are directly available to the user, and relationships represent conceptual relationships between components (Holt, 2001).

Composite Structure Diagram **(Structural; *LP* = Low):** Composite structure diagrams define the internal structure of a class and also the immediate collaborations that are enabled by this structure.

Deployment Diagram **(Structural; *LP* = Medium):** Deployment diagrams consist of nodes representing real-world aspects (such as the hardware platform and execution environment) of a system, and links that show relationships between individual nodes.

Interaction Overview Diagram **(Behavioral/Interaction; *LP* = Low):** These diagrams provide an interaction overview, where nodes represent individual interaction diagrams (a subset of behavioral diagrams).

Object Diagram **(Structural; *LP* = Low):** Object diagrams realize part of the static model of a system and are closely related to class diagrams. They show the insides of things in the class diagrams, as well as their relationships. Moreover, they represent a model or "snapshot" of the partial or complete run-time system at a given point in time.

Package Diagram **(Structural; *LP* = Low):** These diagrams show how the software system is partitioned into logical packages by depicting the interdependencies among these packages.

Profile Diagram **(Structural; *LP* = low, Though Not Included in Ambler's Suggestions for UML 2.0):** A special kind of diagram that operates at the metamodel level (the metamodeling architecture is beyond the scope of this introduction).

Sequence Diagram **(Behavioral/Interaction; *LP* = high):** sequence diagrams are composed of three basic elements: objects, links, and messages, which are exactly the same as for the communication diagram. However, the objects shown in a sequence diagram have a lifeline associated with them, which represents a logical timeline. The timeline is present whenever the object is active, and is illustrated graphically as a vertical line with logical time traveling down the line. The objects for the sequence diagram are shown going horizontally across the page and are shown staggered down the diagram depending on when they are created (Holt, 2001). Figure 6.A13 in Appendix 1 at the end of the chapter illustrates the sequence diagram for the traffic-light control system.

State Machine Diagram **(Behavioral/General; *LP* = Medium):** These diagrams are versatile statecharts, which define the possible states and the allowed state transitions of the system.

Timing Diagram **(Behavioral/Interaction; *LP* = Low):** Timing diagrams describe the critical timing constraints of the system.

Use-Case Diagram **(Behavioral/General; *LP* = Medium):** Use-case diagrams represent the specific interactions of the software application with its external environment, as well as possible dependencies between individual use cases.

The UML, even in its current form, does not provide complete facilities for the specification and analysis needs of real-time systems. However, since the UML is an evolving family of languages, there is no compelling reason for not adding to the family if a suitable language is found. Unfortunately, the majority of appropriate candidates are formal methods—specification languages with a sound mathematical background—and these are traditionally shunned by practitioners.

As stated earlier, the domain model (see Fig. 6.12) is created based upon the use cases, and, through further exploration of system behavior via the interaction diagrams, the domain model evolves systematically into the design class diagram. The construction of the domain model is, therefore, analogous to the analysis stage in SA/SD described earlier. In domain modeling, the central objective is to represent the real-world entities involved in the domain as concepts in the domain model. This is a key aspect of object-oriented systems and is seen as a significant advantage of the paradigm, since the resultant model is closer to reality than in alternative modeling approaches, including the SA/SD.

While most development in object-oriented design was initially done with little or no provision for real-time requirements, the UML 2.0 (released in 2005) with significant extensions for real-time applications improved the situation greatly (Miles and Hamilton, 2006).

6.4.4 Object-Oriented versus Procedural Approaches

The preceding observations beg the question of whether object-oriented design is more suitable than structured design for embedded real-time systems. Structured design and object-oriented design are often compared and contrasted, and, indeed, they are similar in certain ways. This is no surprise, since both have their roots in the pioneering work of Parnas and his predecessors (Parnas, 1972, 1979). Table 6.6 provides a qualitative comparison of these methodologies.

TABLE 6.6. A Side-by-Side Comparison of SA/SD and OOAD (UML) Approaches

System Components	SA/SD Functions	OOAD Objects
Data processes Control processes Data stores	Separated through internal decomposition	All encapsulated within objects
Characteristics	Hierarchy of composition Classification of functions Encapsulation of knowledge within functions	Inheritance of properties Classification of objects Encapsulation of knowledge within objects
User's viewpoint	Rather easy to learn and use	Much more difficult to learn and use
CASE tools	Widely available	Widely available
Volume of usage	Shrinking	Growing

Both structured and object-oriented analysis and design (OOAD) are full life cycle methodologies and use some similar tools and techniques. However, there are major differences as well. SA/SD describes the system from a functional perspective and separates data flows from the functions that transform them, while OOAD describes the system from the perspective of encapsulated entities that possess both function and form.

Additionally, object-oriented models include inheritance, while structured ones do not have such a useful characteristic. Although SA/SD has a definite hierarchical structure, this is a hierarchy of decomposition rather than heredity. Such a shortcoming leads to difficulties in maintaining and extending both the specification and design.

From the user's viewpoint, UML is more difficult to learn and use than SA/SD methods, although they both are supported by matured CASE tools. On the other hand, we see the use of UML growing steadily, while the use of SA/SD is shrinking correspondingly in new products. Notably, these trends are slower in real-time applications than with other kind of software.

An experimental rule-based framework for transforming SA/SD artifacts to UML was proposed by Fries (2006). It is targeted for evolving legacy software that was initially designed using the structured approach. The original data flow and entity relationship diagrams of SA/SD are converted semiautomatically to a use case diagram, sequence diagrams, and a class diagram of UML.

Consider three distinct viewpoints of a system: data, events, and actions. Events represent stimuli, such as various measurements in control systems, as in the case study at the end of this chapter. Actions are precise rules that are followed in computational algorithms, such as "compensate" and "calibrate" in the case of the inertial measurement system. The majority of early computer systems were focused on one, or at most two, of these complementary viewpoints. For instance, nonreal-time image processing systems were certainly data and action intensive, but did not encounter much in the way of events.

Real-time systems are usually data intensive, and hence would seem well suited to structured analysis. Nevertheless, real-time systems also include control information, which is not particularly well suited to structured design. It is likely that a real-time system is as much event or action based as it is data based, which makes it quite suitable for object-oriented techniques, too.

The purpose of this discussion is not to dismiss SA/SD, or even to conclude that it is better than OOAD in all cases. An overriding indicator of suitability of OOAD versus SA/SD to real-time systems is the nature of the application. A similar conclusion was made—not surprisingly—when procedural and object-oriented programming languages were compared in Chapter 4.

6.5 LIFE CYCLE MODELS

A systematic engineering approach to the specification, design, programming, testing, and maintenance of software is essential for maximizing the reliability

and maintainability of real-time systems, as well as for minimizing life cycle expenses. Therefore, software life cycle models form an integral part of any serious development and maintenance process for real-time systems; such models describe explicitly *what must be done* throughout the life cycle. The life cycle is considered to begin when the requirements engineering activities are commissioned and end when the particular software is no longer maintained by the responsible organization. This time period may vary from one year or so up to a few decades; and there are several life cycle models, which are practiced when developing and maintaining real-time software. These models include the classical waterfall model, the V-model, the spiral model, as well as a more recent collection of agile methodologies (Ruparelia, 2010). Nonetheless, most practiced life cycle models are actually hybrids; tailoring is commonly needed to create an appropriate compromise between strictly *sequential* and extensively *iterative* modeling approaches for a particular product and development organization.

Software life cycle models are intended to provide a solid and supportive framework leading to competitive software products within the available budget, personnel, and time frame. The word "competitive" refers here to an application- and environment-specific mixture of the desired software qualities discussed in Section 6.1. By using a well-defined life cycle model with thorough quality assurance procedures, it is possible to prevent the increasing and expensive late-failures period of the bathtub failure function (see Fig. 6.2) even in evolving embedded systems with a lengthy life span. In the following subsections, we will introduce a representative sample of sequential and iterative life cycle models and comment their strengths and weaknesses. All those models include at least a subset of the following fundamental activities:

- Requirements engineering
- Design
- Programming
- Testing
- Transfer to production
- Maintenance

6.5.1 Waterfall Model

The purely sequential waterfall (or cascade) model is the oldest software life cycle model, having its origins in the construction and manufacturing industries. It is based on the idealized assumptions that the requirements can be fixed on before starting the design phase, that the design can be fixed on before starting the programming phase, and so forth (see Fig. 6.13). Furthermore, there is typically a formal review between each phase, and one is allowed to advance to the following phase only when the preceding phase is finalized and

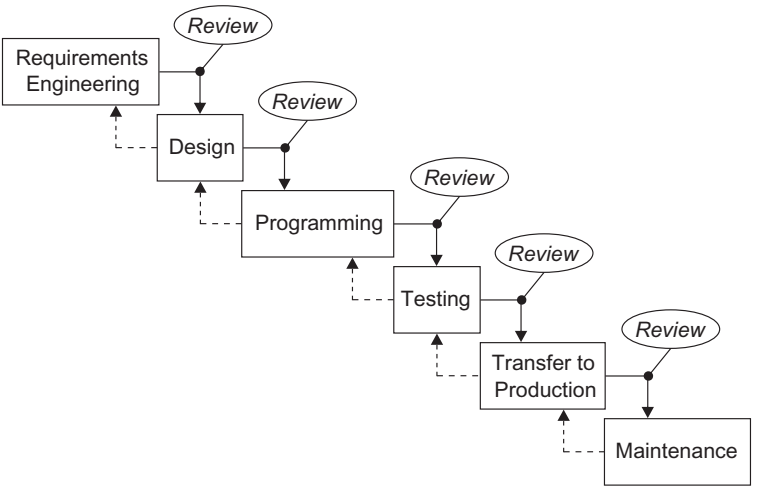


Figure 6.13. Sequential waterfall model with optional feedback enhancements.

approved. No feedback paths are provided for possible iteration in such an idealized scheme. In principle, it would be desirable to follow a feedforward model through the whole development process, but as there are multiple potential reasons why the requirements, design, or program code may need to be modified during a development project, the basic waterfall model has been enhanced to contain optional feedback paths (dashed lines in Fig. 6.13). These relaxing feedbacks make it possible to revisit preceding phases, for instance, to correct programming errors that were detected during tests. Although the enhanced waterfall model provides direct mechanisms for iteration, all the iterations are considered just as exceptions in the "waterfall philosophy." Moreover, quality assurance may be built seamlessly into the waterfall model; each phase can contain both the "*do* part" and the corresponding "*validate* part" (Ruparelia, 2010).

Since the introduction of the waterfall principles over five decades ago, various iterative models have appeared—and that seems to be the future trend, too. In iterative models, the development of all entities can continue throughout the life cycle. Nevertheless, according to a recent survey, a considerable majority (84%) of software projects were still developed according to the waterfall model (or one of its enhancements) and not using any of the modern iterative approaches (Gelbard et al., 2010).

As an important benefit, the cascade flow of the waterfall model makes it straightforward to even outsource individual development phases, since the firm documents, such as the approved Software Requirements Specification and the Software Design Description, are not expected to change later on. On the other hand, waterfall-type life cycle models do not support effectively

such projects that have evolving or changing requirements specifications. This is an increasingly critical issue, for example, when developing a clear but versatile user interface to navigate through evolving smartphone features and functions.

6.5.2 V-Model

The waterfall model has been enhanced not only by introducing simple feedbacks between consecutive phases but in other ways, too. One widely used enhancement is the V-model, where "V" describes both the graphical shape of the development flow and the central objectives related to "validation." Figure 6.14 depicts the V-model for software development; the left fork contains the requirements engineering and design phases in the same way as they are included in the enhanced waterfall model of Figure 6.13; the programming effort with module testing is at the bottom; and the right fork is devoted to quality assurance actions. These quality assurance actions form the heart of V-model, and they are based on close interaction between the symmetrical left and right forks. This means, for instance, that strategies and plans for system validation and integration tests are created already during the requirements engineering and design phases, respectively. Hence, it is ensured that every requirement as well as the design itself are strictly verifiable (Ruparelia, 2010). The foremost aim of the V-model is to tackle two obvious risks appearing in any software development project:

1. Does the integrated software correspond *exactly* to the design?
2. Is the overall system fulfilling *all* the requirements?

In the traditional waterfall model, the testing phase contains similar activities as the right fork of the V-model, but the V-model emphasizes their role

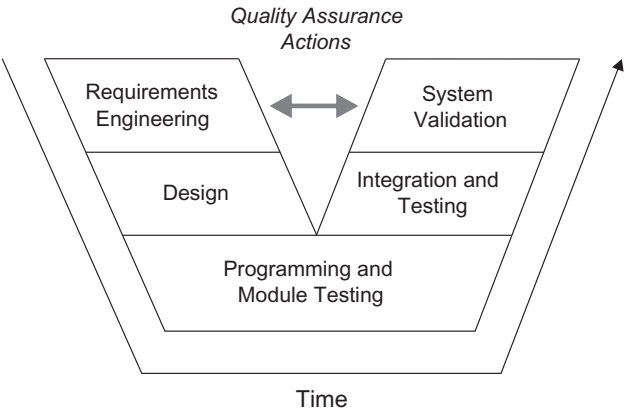


Figure 6.14. V-model with quality assurance activities.

throughout the development life cycle. It is a common practice to fine tune the presented model structure to correspond to the specific needs of a particular project; the five-phase structure of Figure 6.14 is just one example. The more complex a software system to be developed is, the longer the two forks become and hence contain more phases. In principle, the use of V-model is not dependent on the size of the software project. Furthermore, it can be used for hardware and mechanics development, as well.

6.5.3 Spiral Model

A particularly useful modification of the waterfall model, the spiral model (Boehm, 1986), has its orientation in risk analysis and intermediate prototyping. These are taking place cyclically until the phase of detailed design, which is then followed by a typical waterfall sequence (see Fig. 6.15). Each spiral cycle passes four quadrants, Q1–Q4, and ends up to a prototype that is validated, possibly with the stakeholders. The number of completed spiral cycles

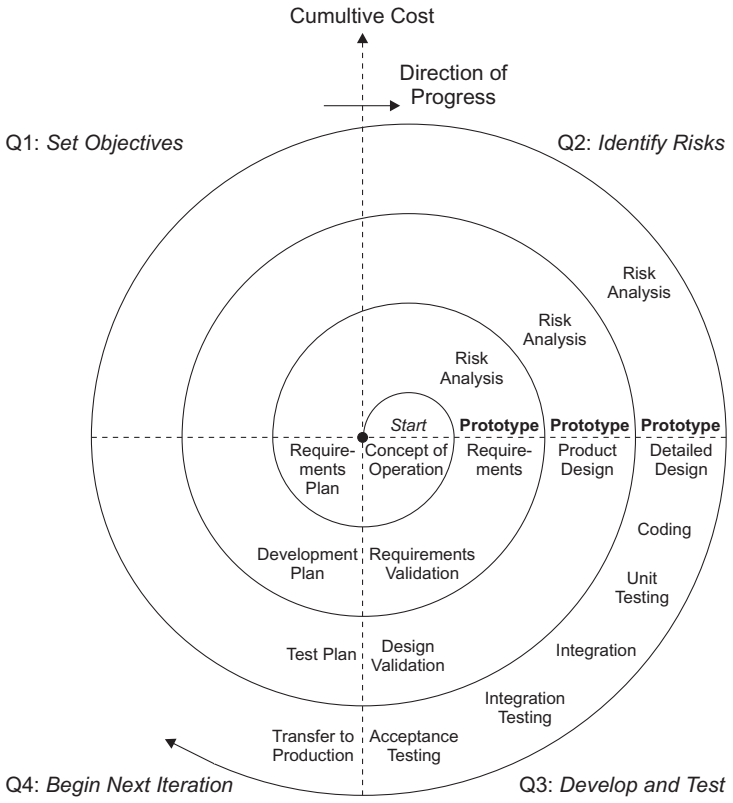


Figure 6.15. Spiral model with early prototyping efforts; adapted from Boehm (1986).

determines the cumulative cost of the development project. While the waterfall model is entirely specification driven, the spiral model is clearly a risk-driven approach.

Possible risks in a software development project are commonly related to the adequacy of available features, usability and user interface, real-time performance, externally furnished components (such as reused requirements specifications or partial designs), as well as various development issues (Boehm, 1991). Some of these risks may grow bigger if the software product is intended for a global market with significant cultural dynamics. Therefore, the importance of careful risk analysis is going to increase in the future. It should be noted, however, that the risk protection benefit of extensive prototyping can be costly. Besides, it is not always easy to identify critical risks, and hence development teams would benefit of risk identification and analysis training.

Also in the case of the spiral model, it is possible to adapt the model details to correspond to the needs of a particular project. Moreover, the use of the spiral model requires considerable effort by the project management. The philosophy of the spiral model can be stated as "start small, think big" (Ruparelia, 2010).

6.5.4 Agile Methodologies

Agile methodologies belong to a dynamic family of iterative and incremental software development strategies. While in the enhanced waterfall model of Figure 6.13, any iteration is considered as an undesired exception, agile methodologies are based on intentional iterations leading to incremental completion of the software under development. Hence, the underlying principle is far from that of the waterfall model, V-model, or spiral model, and it cannot be depicted with a static workflow diagram containing interconnected development activities. Figure 6.16 illustrates a flow of an imaginary software development project using an iterative agile strategy. At any iteration step, there are multiple merging development activities with varying effort volumes. Besides, a set of consecutive iterations can form a "mini project" that could provide a partial software release to the customer. Although agile methodologies are often deployed with a lack of rigid process, they can be, when correctly implemented, rigorous and thus suitable for embedded applications (Laplante, 2009).

There are several widely used agile methodologies, such as Crystal, Dynamic Systems Development Method, eXtreme Programming (XP), Feature-Driven Development, and Scrum, as well as a large number of *ad hoc* methodologies that claim to be agile (Laplante, 2009). Because they are relatively new, are light on documentation and formal process, and involve a high degree of experimentation early in the systems development process (when prototype hardware may be unavailable), agile methodologies are not frequently used in real-time and embedded systems development. Nevertheless, in many cases, where the true philosophy of agile development is embraced

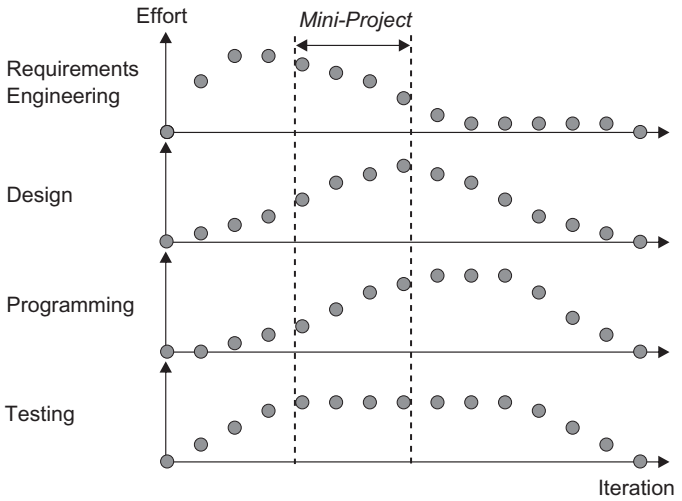


Figure 6.16. A sample flow of a software development project using iterative agile methodologies; adapted from Larman (2004).

and where the culture and application domain are appropriate, agile development can be the right development approach for real-time and embedded systems, as well.

It is beyond the scope of this book to describe any one agile methodology or to undertake a detailed analysis of when and how to use these approaches in real-time and embedded systems development. It is important, however, to understand and appreciate the philosophy of agile methodologies in order to see why they might be suitable for certain real-time applications. And to understand these approaches, it is essential to look at the Agile Manifesto and the explicit principles behind it. The following manifesto was introduced by a group of agility proponents in 2001 (Larman, 2004):

Definition: Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

That is, while there is value on the items on the right, we value items on the left more.

From the noble manifesto, a set of 12 principles were derived (Larman, 2004):

Definition: Agile Principles

P1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
P2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
P3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
P4. Business people and developers must work together daily throughout the project.
P5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
P6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
P7. Working software is the primary measure of progress.
P8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
P9. Continuous attention to technical excellence and good design enhances agility.
P10. Simplicity—the art of maximizing the amount of work done—is essential.
P11. The best architectures, requirements, and designs emerge from self-organizing teams.
P12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These principles, if practiced consistently, end up to a flexible what-to-do plan for the project, instead of a firm one. Moreover, the agile approach is human oriented as opposed to task orientation. The principle P5 is particularly interesting, because it enables the team members to utilize their "free will" (a powerful characteristic, which distinguishes human intelligence from advanced machine intelligence [Martinez, 2006]) instead of being controlled merely by policies, procedures, superiors, and so on.

Could agile methodologies even offer "the most suitable strategy" for all software development? We will answer this tempting question by referring synergistically to the "no free lunch" (NFL) theorems that Wolpert and Macready discussed in the context of optimization algorithms (Wolpert and Macready, 1997). They proved that improved performance for any optimization algorithm indicates a match between the structure of the algorithm and the structure of the problem at hand. Therefore, a general-purpose algorithm is never the most suitable one

for a specific problem, and the most suitable algorithm for a specific problem is not a general-purpose one. Intuitively reasoning, it could be possible to develop similar NFL theorems for software development strategies, as well; such an effort is, however, beyond the scope of this text. Nevertheless, if we apply our intuition freely, we can say that any general-purpose strategy is never the best one for all software development. For this reason, case-dependent tuning is advantageous when creating an appropriate life cycle model for a particular application and development environment. For instance, applying agile methodologies strictly to large software projects is difficult, since they stress the face-to-face communication and self-organizing teams that may not be possible to achieve due to large development groups and multiple geographical locations where the work is carried out (Ruparelia, 2010). Nonetheless, agile methodologies are undoubtedly usable in smaller-scale applications with changing or evolving requirements specifications.

Vignette: Are We Witnessing a Second Agile Development Period?

When examining the Agile Manifesto and the associated principles, we noticed that they contain numerous similarities to those informal software development strategies, which were used in the beginning of the embedded systems era—some three decades ago. At that time, industry was still inconsistent in the use of microprocessors; requirements specifications changed frequently; the entire software was not more than a few tens of kilobytes; development teams consisted typically of 2–3 highly motivated software engineers; and nobody in the team was a "guru," but all members were fairly inexperienced. Such initial conditions formed a fruitful basis for agile-type behavior to emerge by itself.

Based on the authors' subjective observations in a few development organizations—rather than any real survey—it can be argued that the *most successful* teams practiced up to 10 of the 12 agile principles: P1, P2, P4–P7, and P9–P12. Furthermore, all items of the manifesto were, more or less, common practices.

Is this just a coincidence, or is there truly something similar in the advanced products of tomorrow and the early embedded systems? No, it is probably not a coincidence, but the *frequently changing requirements and new technological opportunities* form common points of contact, which are as concrete with the future smartphone user interfaces as they were with the pioneering forest harvesters, for example. However, most of the agile principles were put aside for a couple of decades; since embedded software was growing rapidly in size, development teams became bigger and geographically distributed, and it was impossible to find an adequate number of well motivated and self-organizing individuals to support the embedded systems boom. These and other changes in the operating environment pushed the development organizations toward rigid life cycle models and strict project management practices.

Lastly, it should be pointed out that there are also other iterative software development approaches than the popular agile methodologies outlined above. A comprehensive discussion on agile and iterative development, from the manager's viewpoint, is available in Larman (2004).

6.6 SUMMARY

The purpose of software design is to create a sound mapping from the requirements document to an implementable design document. In general, there exist an infinite number of possible mappings. But which one of those is the desired one? To achieve a desirable mapping, we first need to define the term "desirable," which is related directly to a set of weighted software quality measures. These specific qualities (such as performance and maintainability) and usual ways to achieve them (such as modularity and generality) were discussed in the beginning of this chapter. In addition to software qualities, the term "desirable" is related necessarily to the development organization and environment, as well as the type and market of the software product. Thus, every design process includes actually a multi-objective optimization problem with considerable uncertainties; for instance, how should the different software qualities be weighted with respect to each other? The ultimate success of the software is largely dependent on the experience and skills of the design team to solve such problems.

There are two principal approaches to generate and document software designs: the procedural design approach and the object-oriented approach. It is useless to debate which approach is better and should hence be preferred generally. Instead, the selected approach must be justified by the concrete application needs and future visions of the development organization. Currently, many industrial companies with a long history of embedded systems development either have just switched to object-oriented techniques or are in the process of such a major transition. In large procedural-oriented organizations, this transition can be a laborious educational effort, since, for example, the fluent use of UML is more demanding than the use of SA/SD methods. The situation is apparently very different in young and small development organizations.

Software development and maintenance life cycle models have a central role in every serious development process. The purpose of strict "life cycle thinking" is to minimize the total expenses during the entire life cycle—not just the development expenses, as is traditionally done in the first place. However, while the life cycle thinking makes a lot of sense from the corporation or company point of view, it may be challenging to put into practice in large organizations, where the development expenses are often paid from "a different pocket" than the maintenance expenses. The situation is even more difficult when the lifespan of the software product is lengthy. Thus, the adoption of a life cycle model to cover both the development and maintenance phases is actually an executive-level decision.

The classical waterfall model or one of its enhancements have an established position in most development organizations. Nevertheless, as there is no single kind of software or development environment, there is a need to tune and evolve the existing life cycle models, or even create new life cycle philosophies. In the past decade, agile methodologies have gained interest and acceptance in applications where the requirements are changing frequently during the design and implementation phases. Agile methodologies provide an iterative and incremental alternative to the primarily sequential and rigid development life cycles. While these methodologies are shown to be effective in certain situations, they are no "silver bullet" for all. On the other hand, their position is clearly emerging in smaller-scale real-time systems (or subsystems) involved with novel technological opportunities.

In the future, such productivity issues as design reuse and (semi-)automatic design from requirements will continue to be important but challenging areas of research and development.

We want to close this chapter by Norman Maclean's captivating words: "Eventually, all things merge into one, and a river runs through it" (Maclean, 2001). These words have an obvious analogy to the design of embedded software.

6.7 EXERCISES

6.1. For whom should you, as a designer, prepare the software design description?

6.2. What are the primary reasons behind the current (and seemingly continuing) situation that there is no single, universally accepted approach for software design?

6.3. Why is it that the actual program code, even though it is an exact model of system behavior, is insufficient in serving as either a software requirements document or a software design document? In any case, pseudocode is used widely for such purposes.

6.4. How would you, as a software project manager, handle the confusing situation in which the software requirements specification contains numerous design-level details as well?

6.5. Why is it of utmost importance that the program code be traceable to the software design specification, and, in turn, to the software requirements specification? What are the possible consequences if it is not traceable?

6.6. A mapping from advantageous software engineering principles to desired software qualities is sketched in Table 6.4. Give specific explanations why "Modularity" is mapping to "Reliability," "Correctness,"

"Performance," "Interoperability," "Maintainability," and "Portability." Or, do you disagree on some of those suggested mappings?

6.7. What are the principal differences between procedural design approaches and object-oriented ones?

6.8. Why are procedural design approaches still practiced with many embedded systems—even with completely new products? What could be hindering the adoption of object-oriented approaches in those cases?

6.9. Using a data flow diagram, capture the data and functional requirements for monitoring the entry, exit, and traversal of aircraft in a busy airspace. Aircraft entering the space are sensed by the *Radar* input; the *Comm* input identifies aircraft that leave the space. The current contents of the space are maintained in the data area *AirspaceStatus*. A detailed log or history of the space usage is kept in the *AirspaceLog* storage. Air-traffic control personnel can request the display of the status of a particular aircraft through the *Controller* input.

6.10. Take the procedural design approach and create first the context diagram, and then the highest-level data flow and control-flow diagrams for an electronic lock in the laboratory door having the following requirements specifications:

- The lock has an integrated RFID card reader, and every registered user has a unique identification code.
- An accepted card is acknowledged by a green LED and a rejected one by a red LED.
- The lock will open when an adequate current is flowing through its control solenoid; otherwise, it remains locked.
- Information about registered users and their permitted entrance times is stored on a database of a remote workstation that manages all locks within the whole college building.
- Every successful and unsuccessful opening attempt is recorded on the database with the corresponding identification code, date, and time.
- Embedded controllers of individual locks in the building communicate with the common workstation through a wireless communications network.

You may define additional requirements yourself, if needed.

6.11. Perform a web search and find the reasons why the Unified Modeling Language (UML) was originally developed. What were the primary reasons why UML 2.0 appeared?

6.12. UML's use-case diagrams (see Fig. 5.14) are usually complemented by textual descriptions; what kind of information do they contain?

6.13. Redraw the use-case diagram of the elevator control system in Figure 5.14 for a maximally simplified single elevator, which is not a part of a multi-elevator bank.

6.14. Consider the following real-time systems:

(a) Elevator control system for simple home elevators.
(b) Core monitoring system of a nuclear power plant.
(c) Distributed airline reservations system for global use.

What design approach would you favor with each of them and why?

6.15. Consider the following embedded systems:

(a) Anti-lock braking system for buses.
(b) User interface of an evolving smartphone.
(c) Elevator monitoring system for domestic market.

What life cycle model would you prefer with each of them and why?

6.8 APPENDIX 1

CASE STUDY IN DESIGNING REAL-TIME SOFTWARE

To further illustrate the concepts of design, the Software Requirements Specification given in the case study of Section 5.7 is used to provide a corresponding object-oriented design for the traffic light control system. Some of the following figures have been referred to in the previous sections. This appendix serves to further explicate the object-oriented design process, many of its artifacts, and provides an instructive example of an object-oriented design document.

6.8.1 Introduction

Traffic controllers currently in use comprise simple timers that follow a fixed cycle to allow vehicle/pedestrian passage for a predetermined amount of time regardless of demand, actuated traffic controllers that allow passage by means of vehicle/pedestrian detection, and adaptive traffic controllers that determine traffic conditions in real time by means of vehicle/pedestrian detection and respond accordingly in order to maintain the highest reasonable level of efficiency under varying conditions. The traffic controller described in this design document is capable of operating in all three of these modes.

6.8.1.1 Purpose The purpose of this document is to provide a comprehensive set of software design guidelines to be used in the development phase of the application. This specification is intended for use by software developers.

6.8.1.2 Scope This software package is part of a control system for pedestrian/vehicular traffic intersections that allows for (1) a fixed cycle mode, (2) an actuated mode, (3) a fully adaptive automatic mode, (4) a locally controlled manual mode, (5) a remotely controlled manual mode, and (6) an emergency preempt mode. In the fully adaptive automatic mode, a volume detection feature has been included so that the system is aware of changes in traffic patterns. Pushbutton fixtures are also included so the system can account for and respond to pedestrian traffic. The cycle is controlled by an adaptive algorithm that uses data from many inputs to achieve maximum throughput and acceptable wait-times for both pedestrians and motorists. A preempting feature allows emergency vehicles to pass through the intersection in a safe and timely manner by altering the state of the signals and the cycle time.

This document follows the structure provided in the object-oriented SRS template found in IEEE Std 830–1998 and adopted in Section 5.7 rather than that defined in IEEE Std 1016–1998 due to the fact that, as acknowledged in the IEEE standard itself, IEEE Std 1016 is not suitable as a basis for representing object-oriented designs.

6.8.1.3 Definitions and Acronyms In addition to those given in Section 5.7, the following terms are defined here.

6.8.1.3.1 Accessor A method used to access a private attribute of an object.

6.8.1.3.2 Active Object An object that owns a thread and can initiate control activity. An instance of active class.

6.8.1.3.3 Collaboration A group of objects and messages between them that interact to perform a specific function.

6.8.1.3.4 Mutator A method used to modify a private attribute of an object.

6.8.1.4 Documentation Standards

- IEEE Std 830–1998
- IEEE Std 1016–1998

6.8.2 Overall Description

6.8.2.1 Intersection Overview The intersection class to be controlled is illustrated in Figure 6.A1. This figure has been repeated from Section 5.7.

The target class of intersection is described in detail in Section 5.7.

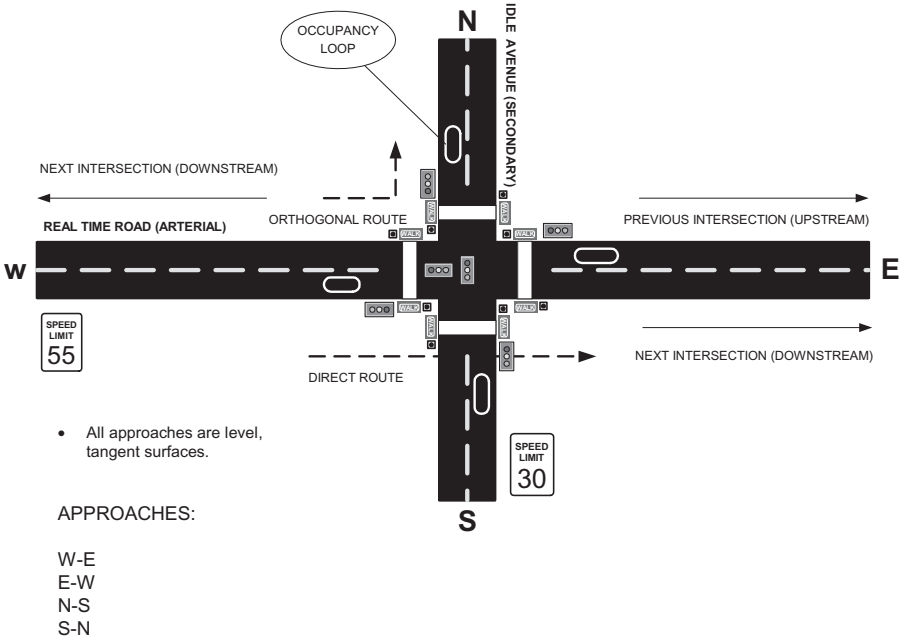


Figure 6.A1. Intersection topography.

6.8.2.2 Intersection Software Architecture The intersection controller software architecture consists of the major components shown in Figure 6.A2.

6.8.2.2.1 Real-Time Operating System (RTOS) The RTOS selected for the intersection controller is QNX Neutrino 6.2 for the iX86 family of processors.

6.8.2.2.2 Application Software Application software is written in C++ and is compiled using QNX Photon tools and the GNU GCC 2.95 compiler.

6.8.2.2.3 Resource Managers Resource managers are written in C++ using the QNX Driver Development Kit. Note that these have been developed by another team and so have not been covered in detail in this document.

6.8.3 Design Decomposition

This section provides a detailed object-oriented decomposition of the intersection controller software design. The decomposition is based on the use cases and preliminary class model described in Section 5.7.

The decomposition makes use of the unified modeling language (UML), supplemented by text descriptions, to define the details of the design. This representation provides the design views described in IEEE Std 1016 within the framework of object-oriented design, as shown in Table 6.A1.

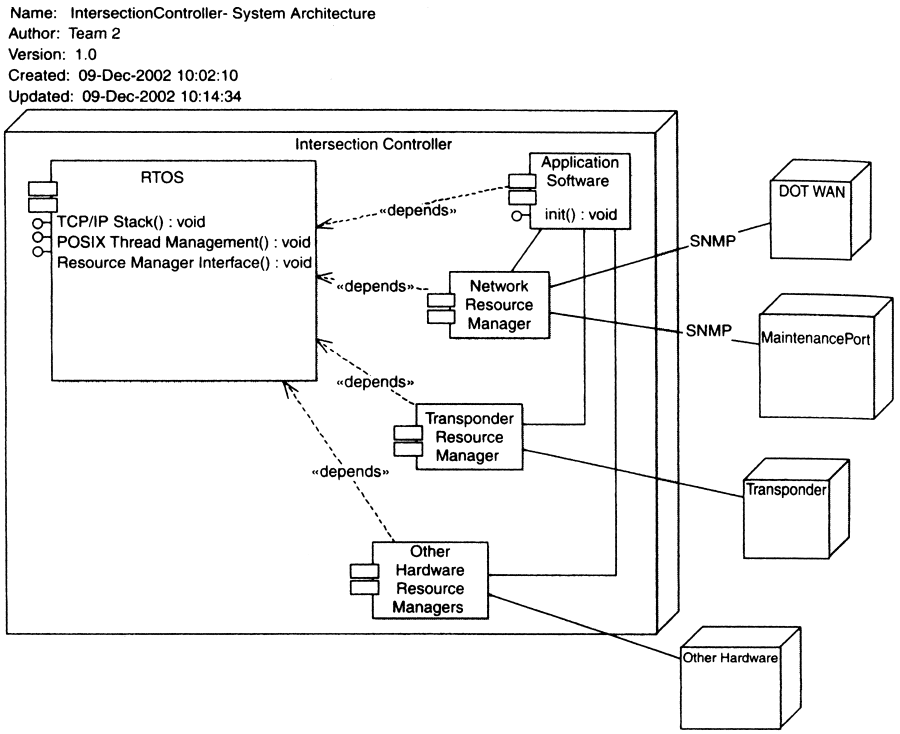


Figure 6.A2. Intersection controller software architecture.

TABLE 6.A1. IEEE Std 1016 Design Views

Design View	Represented By	SDD Reference
Decomposition view	Classes in class diagram	Figure 6.A10
Interrelationship view	Associations in class diagram	Figure 6.A10
Interface view	Collaboration diagrams	Figure 6.A4 through Figure 6.A9
Detailed view	Attribute and method details; behavioral diagrams	For each class

6.8.3.1 Major Software Functions (Collaborations) Based on the use case diagram provided in Section 5.7, the major functions of the intersection controller have been grouped into UML collaborations as shown in Figure 6.A3. Collaboration details are described in the following paragraphs.

6.8.3.1.1 Collaboration Messages The tables below provide a listing of the messages (method calls and events) passed between objects in each collaboration defined above. Messages with an "on…" prefix correspond to events.

Name: IntersectionController- Top-Level Users
Author: Team 2
Version: 1.0
Created: 30-Nov-2002 09:01:19
Updated: 03-Dec-2002 07:35:44

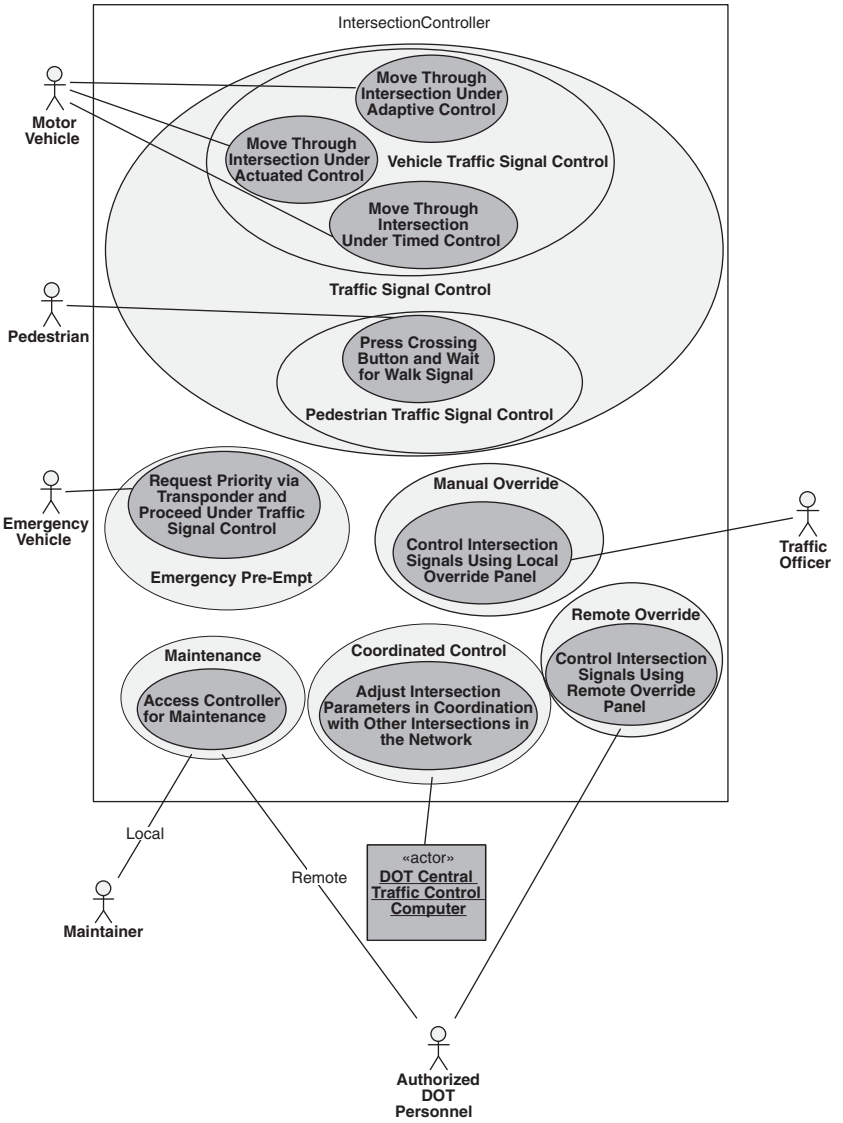


Figure 6.A3. Intersection controller collaborations.

6.8.3.1.1.1 TRAFFIC SIGNAL CONTROL (Table 6.A2)

6.8.3.1.1.2 EMERGENCY PREEMPT (Table 6.A3)

6.8.3.1.1.3 MANUAL OVERRIDE (Table 6.A4)

6.8.3.1.1.4 REMOTE OVERRIDE (Table 6.A5)

6.8.3.1.1.5 COORDINATED CONTROL (Table 6.A6)

6.8.3.1.1.6 MAINTENANCE (Table 6.A7)

6.8.3.1.2 Collaboration Diagrams The collaborations described above are depicted in Figures 6.A4–6.A9.

6.8.3.2 Class Model Figure 6.A10 depicts the classes constituting the intersection control system software application. The diagram reflects the preliminary class structure defined in Section 5.7, but with additional detail, and, in some cases, addition of classes and reallocation of responsibilities.

Classes corresponding to active objects (i.e., objects with their own thread of control) are shown in Figure 6.A10 with **bold** outlines. The active object instances are summarized in Table 6.A8.

6.8.3.3 Class Details

6.8.3.3.1 IntersectionController The IntersectionController class is responsible for managing the following functions:

1. Initialization.
2. Instantiation of contained objects.
3. Overall control of the intersection vehicle traffic standards.
4. Overall control of the intersection pedestrian traffic standards.
5. Collection and processing of traffic history from all approaches.
6. Adaptive control of intersection timings in response to traffic flow.
7. Actuated control of intersection in response to vehicle presence.
8. Timed control of intersection in response to a fixed scheme.
9. Overall handling of pedestrian crossing requests.
10. Handling of emergency vehicle pre-emption.
11. Intersection control in response to manual override commands.
12. Intersection control in response to remote override commands.
13. Management of traffic history and incident log databases.
14. Handling of maintenance access requests from the maintenance port.
15. Handling of maintenance access requests from the DOT WAN.

TABLE 6.A2. Intersection Controller—Traffic Signal Control Collaboration Messages

ID	Message	From Object	To Object
1	setAspect(Aspect)	m_Intersection Controller	m_Approach[0]
1.1	getAspect()	m_Intersection Controller	m_Approach[0]
1.2	getCount()	m_Intersection Controller	m_Approach[0]
2	ignoreState()	m_Approach[0]	m_PedestrianDetector[0]
2.1	watchState()	m_Approach[0]	m_PedestrianDetector[0]
2.2	resetState()	m_Approach[0]	m_PedestrianDetector[0]
3	onEntryStateSet(void)	m_VehicleDetector	m_Approach[0]
3.1	onEntryStateCleared(void)	m_VehicleDetector	m_Approach[0]
4	onPedestrianRequest()	m_PedestrianDetector[0]	m_Approach[0]
5	onPedestrianRequest()	m_Approach[0]	m_Intersection Controller
5.1	onVehicleEntry(int)	m_Approach[0]	m_Intersection Controller
6	setIndication(Indication)	m_Approach[0]	m_PedestrianTrafficStandard[0]
6.1	getIndication()	m_Approach[0]	m_PedestrianTrafficStandard[0]
7	setIndication(Indication)	m_Approach[0]	m_VehicleTrafficStandard[0]
7.1	getIndication()	m_Approach[0]	m_VehicleTrafficStandard[0]

TABLE 6.A3. Intersection Controller—Emergency Preempt Collaboration Messages

ID	Message	From Object	To Object
1	onActivate()	Emergency Vehicle Transponder	m_EmergencyPreempt
1.1	onDeactivate()	Emergency Vehicle Transponder	m_EmergencyPreempt
2	onPreemptRequest()	m_EmergencyPreempt	m_Intersection Controller
2.1	onPreemptCleared()	m_EmergencyPreempt	m_Intersection Controller

TABLE 6.A4. Intersection Controller—Manual Override Collaboration Messages

ID	Message	From Object	To Object
1	onActivate(OT)	Manual Control Panel	m_ManualOverride
1.1	onDeactivate()	Manual Control Panel	m_ManualOverride
2	onSetPhase()	Manual Control Panel	m_ManualOverride
3	onOverrideActivated(OT)	m_ManualOverride	m_Intersection Controller
3.1	onOverrideDeactivated(OT)	m_ManualOverride	m_Intersection Controller
4	setPhase()	m_ManualOverride	m_Intersection Controller

OT: OverrideType

TABLE 6.A5. Intersection Controller—Remote Override Collaboration Messages

ID	Message	From Object	To Object
1	onActivate(OT)	m_Network	m_RemoteOverride
1.1	onDeactivate(OT)	m_Network	m_RemoteOverride
2	onSetPhase()	m_Network	m_RemoteOverride
3	onOverrideActivated(OT)	m_RemoteOverride	m_Intersection Controller
3.1	onOverrideDeactivated(OT)	m_RemoteOverride	m_Intersection Controller
4	setPhase()	m_RemoteOverride	m_Intersection Controller
5	sendPacket(void*)	m_RemoteOverride	m_Network

OT: OverrideType

TABLE 6.A6. Intersection Controller—Coordinated Control Collaboration Messages

ID	Message	From Object	To Object
1	setMode(Mode)	m_RemoteOverride	m_Intersection Controller
2	setParameters()	m_RemoteOverride	m_Intersection Controller
3	getStatus()	m_RemoteOverride	m_Intersection Controller
4	onSetParameters (Parameters*)	m_Network	m_RemoteOverride
5	onGetStatus()	m_Network	m_RemoteOverride
6	sendPacket(void*)	m_RemoteOverride	m_Network

TABLE 6.A7. Intersection Controller—Maintenance Collaboration Messages

ID	Message	From Object	To Object
1	getStatus()	m_Maintenance	m_Intersection Controller
2	goFirst()	m_Maintenance	m_IncidentLog
2.1	read()	m_Maintenance	m_IncidentLog
2.2	goNext()	m_Maintenance	m_IncidentLog
2.3	isEOF()	m_Maintenance	m_IncidentLog
3	flush()	m_Maintenance	m_IncidentLog
4	getStatus()	m_Network	m_Maintenance
5	readDatabase(int)	m_Network	m_Maintenance
6	sendPacket(void*)	m_Maintenance	m_Network

Figure 6.A11 illustrates the attributes, methods, and events of the IntersectionController class.

6.8.3.3.1.1 INTERSECTIONCONTROLLER RELATIONSHIPS

- Association link from class *Status*
- Association link to class *PreEmpt*
- Association link to class *Network*
- Association link from class *PreEmpt*
- Association link to class *Database*
- Association link from class *Override*
- Association link to class *Mode*
- Association link from class *Maintenance*
- Association link to class *Database*
- Association link to class *Parameters*
- Association link to class *RemoteOverride*

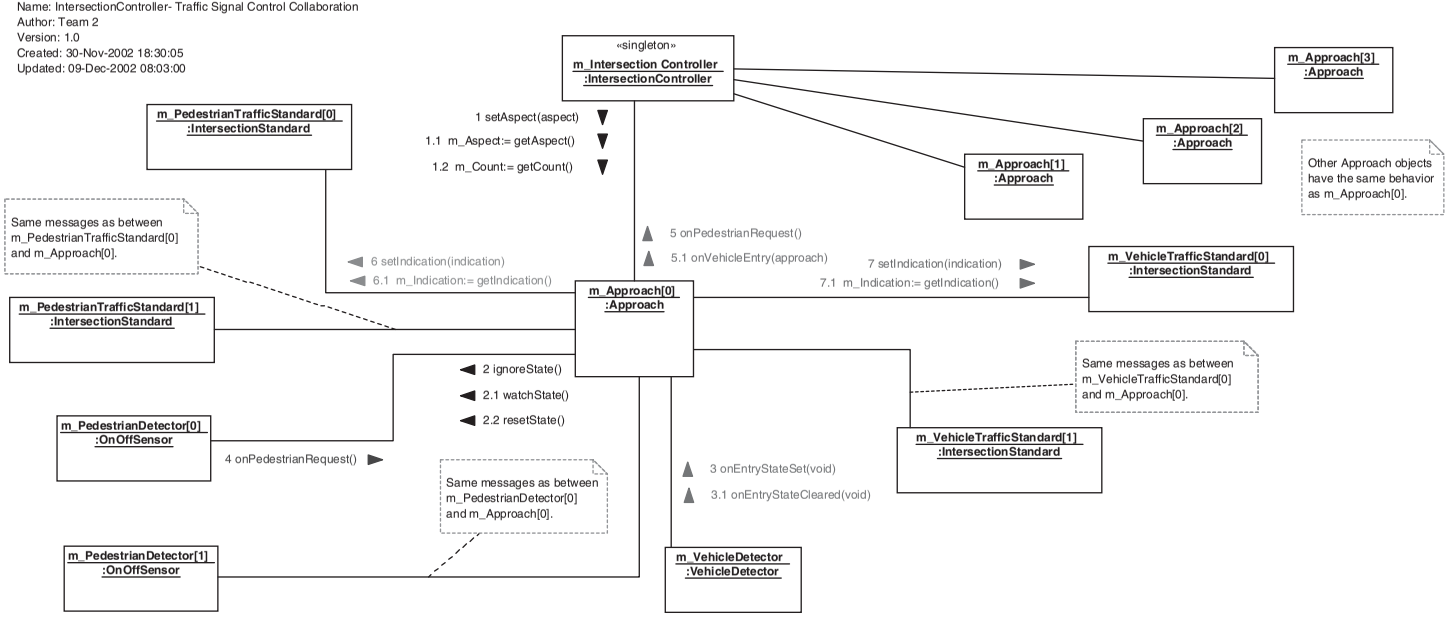


Figure 6.A4. Traffic signal control.

Name: IntersectionController- Emergency Preempt Collaboration
Author: Team 2
Version: 1.0
Created: 03-Dec-2002 14:56:09
Updated: 03-Dec-2002 15:08:11

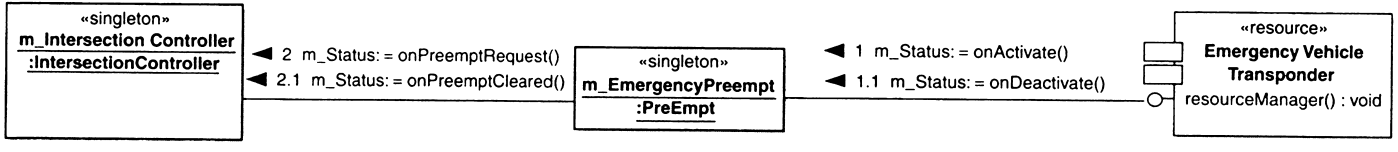


Figure 6.A5. Emergency preempt.

Name: IntersectionController- Manual Override Collaboration
Author : Team 2
Version : 1.0
Created : 03-Dec-2002 08:50:43
Updated : 05-Dec-2002 18:38:06

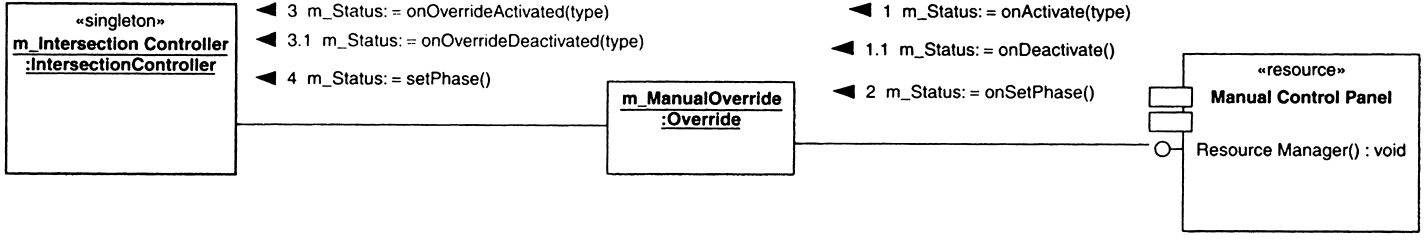


Figure 6.A6. Manual override.

Name: IntersectionController- Remote Override Collaboration
Author: Team 2
Version: 1.0
Created: 03-Dec-2002 11:55:19
Updated: 05-Dec-2002 18:46:02

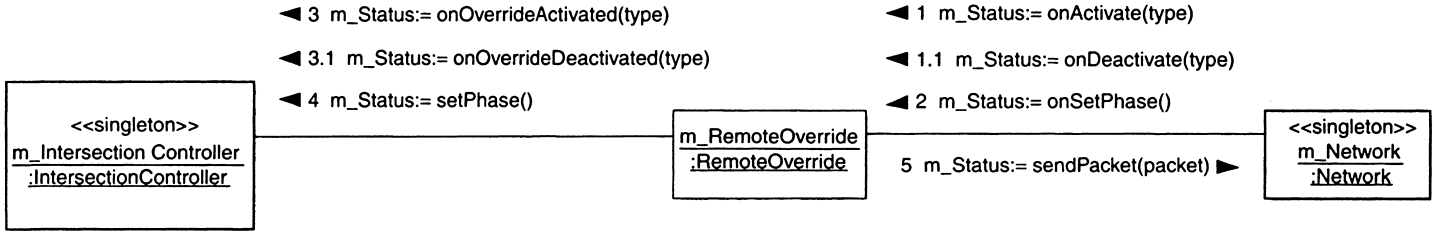


Figure 6.A7. Remote override.

Name: IntersectionController- Remote Override Collaboration
Author: Team 2
Version: 1.0
Created: 03-Dec-2002 11:55:19
Updated: 05-Dec-2002 18:46:02

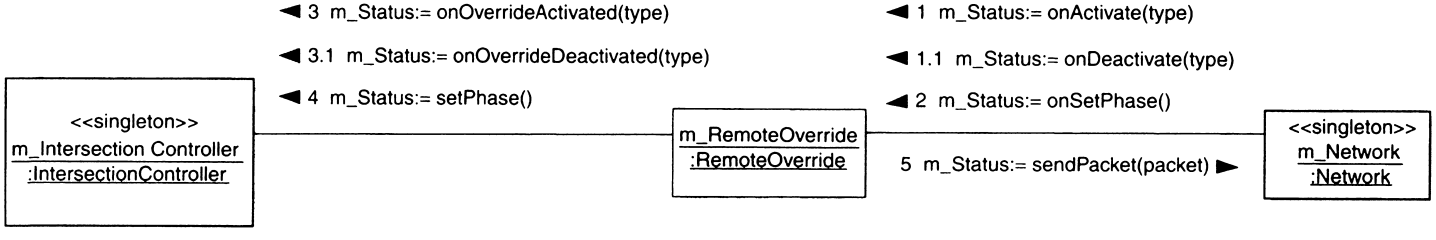


Figure 6.A8. Coordinated control.

Name: IntersectionController- Coordinated Control Collaboration
Author: Team 2
Version: 1.0
Created: 03-Dec-2002 15:11:01
Updated: 03-Dec-2002 15:27:15

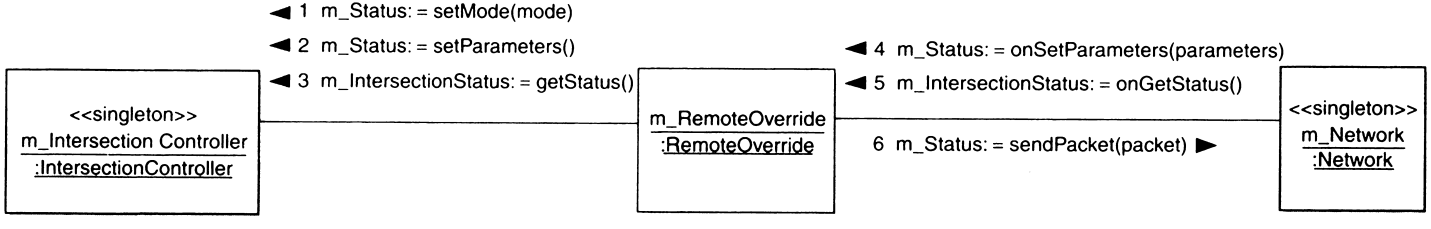


Figure 6.A9. Maintenance.

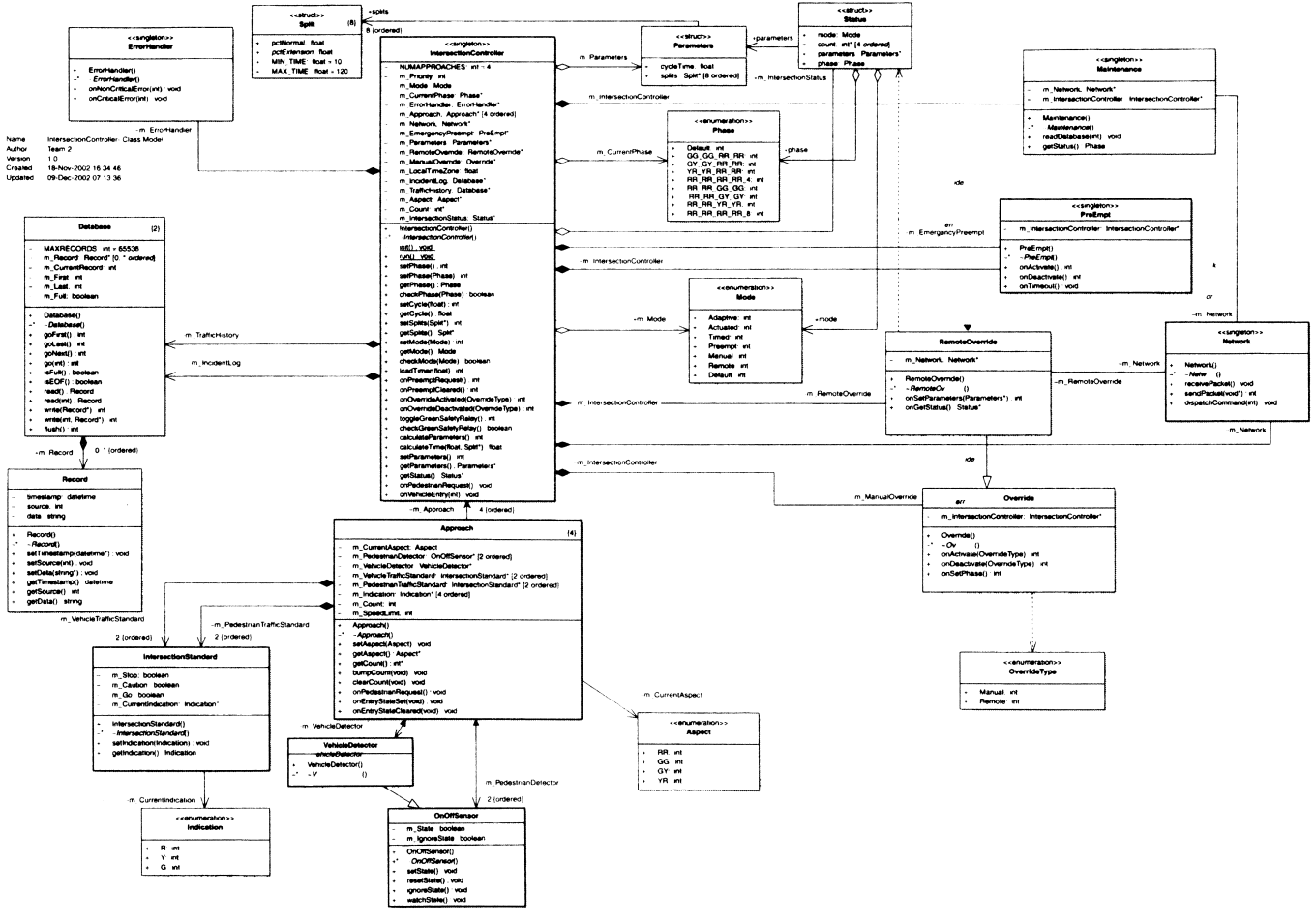


Figure 6.A10. Intersection controller class diagram.

TABLE 6.A8. Active Objects

Level	Object Name
1.	m_IntersectionController
1.1.	m_IntersectionController::m_Approach[0]
1.1.1.	m_IntersectionController::m_Approach[0]::m_VehicleTrafficStandard[0]
1.1.2.	m_IntersectionController::m_Approach[0]::m_VehicleTrafficStandard[1]
1.1.3.	m_IntersectionController::m_Approach[0]::m_VehicleTrafficStandard[0]
1.1.4.	m_IntersectionController::m_Approach[0]::m_PedestrianTrafficStandard[1]
1.1.5.	m_IntersectionController::m_Approach[0]::m_PedestrianDetector[0]
1.1.6.	m_IntersectionController::m_Approach[0]::m_PedestrianDetector[1]
1.1.7.	m_IntersectionController::m_Approach[0]::m_VehicleDetector
1.2.	m_IntersectionController::m_Approach[1]
1.2.1.	m_IntersectionController::m_Approach[1]::m_VehicleTrafficStandard[0]
1.2.2.	m_IntersectionController::m_Approach[1]::m_VehicleTrafficStandard[1]
1.2.3.	m_IntersectionController::m_Approach[1]::m_VehicleTrafficStandard[0]
1.2.4.	m_IntersectionController::m_Approach[1]::m_PedestrianTrafficStandard[1]
1.2.5.	m_IntersectionController::m_Approach[1]::m_PedestrianDetector[0]
1.2.6.	m_IntersectionController::m_Approach[1]::m_PedestrianDetector[1]
1.2.7.	m_IntersectionController::m_Approach[1]::m_VehicleDetector
1.3.	m_IntersectionController::m_Approach[2]
1.3.1.	m_IntersectionController::m_Approach[2]::m_VehicleTrafficStandard[0]
1.3.2.	m_IntersectionController::m_Approach[2]::m_VehicleTrafficStandard[1]
1.3.3.	m_IntersectionController::m_Approach[2]::m_VehicleTrafficStandard[0]
1.3.4.	m_IntersectionController::m_Approach[2]::m_PedestrianTrafficStandard[1]
1.3.5.	m_IntersectionController::m_Approach[2]::m_PedestrianDetector[0]
1.3.6.	m_IntersectionController::m_Approach[2]::m_PedestrianDetector[1]
1.3.7.	m_IntersectionController::m_Approach[2]::m_VehicleDetector
1.4.	m_IntersectionController::m_Approach[3]
1.4.1.	m_IntersectionController::m_Approach[3]::m_VehicleTrafficStandard[0]
1.4.2.	m_IntersectionController::m_Approach[3]::m_VehicleTrafficStandard[1]
1.4.3.	m_IntersectionController::m_Approach[3]::m_VehicleTrafficStandard[0]
1.4.4.	m_IntersectionController::m_Approach[3]::m_PedestrianTrafficStandard[1]
1.4.5.	m_IntersectionController::m_Approach[3]::m_PedestrianDetector[0]
1.4.6.	m_IntersectionController::m_Approach[3]::m_PedestrianDetector[1]
1.4.7.	m_IntersectionController::m_Approach[3]::m_VehicleDetector
2.	m_IntersectionController::m_Network
3.	m_IntersectionController::m_EmergencyPreempt

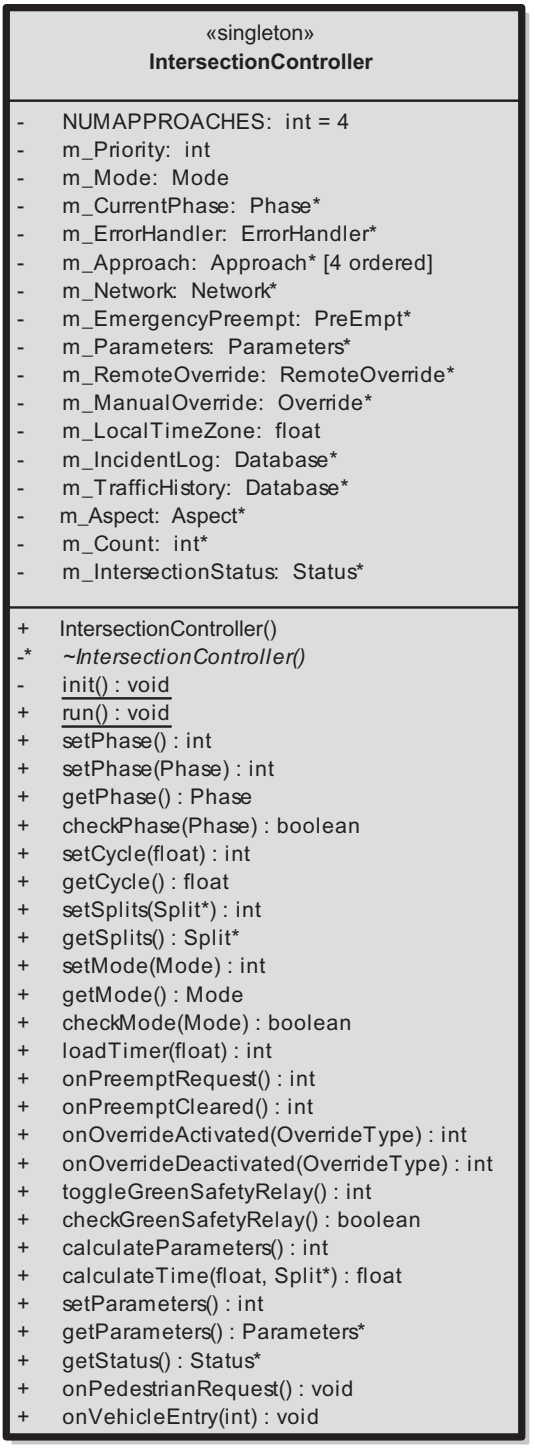


Figure 6.A11. Intersection controller class.

- Association link to class *Phase*
- Association link to class *ErrorHandler*
- Association link to class *Approach*

6.8.3.3.1.2 INTERSECTIONCONTROLLER ATTRIBUTES (Table 6.A9)

6.8.3.3.1.3 INTERSECTIONCONTROLLER METHODS (Table 6.A10)

6.8.3.3.1.4 INTERSECTIONCONTROLLER BEHAVIORAL DETAILS (Figs. 6.A12–6.A15)

6.8.3.3.2 Approach This is the programmatic representation of an individual entrance into the intersection.

The Approach class is responsible for managing the following functions:

1. Instantiation of contained objects.
2. Control of the traffic standards associated with the approach.
3. Handling of pedestrian crossing events.
4. Handling of loop detector entry and exit events.
5. Tracking the vehicle count.

Figure 6.A16 illustrates the attributes, methods, and events of the Approach class.

6.8.3.3.2.1 ASPECT RELATIONSHIPS

- Association link to class *IntersectionStandard*
- Association link to class *Aspect*
- Association link to class *IntersectionStandard*
- Association link to class *VehicleDetector*
- Association link to class *OnOffSensor*
- Association link from class *IntersectionController*

6.8.3.3.2.2 APPROACH ATTRIBUTES (Table 6.A11)

6.8.3.3.2.3 APPROACH METHODS (Table 6.A12)

6.8.3.3.3 IntersectionStandard Class (Pedestrian Traffic and Vehicle Traffic Standard) This is the programmatic representation of a traffic control signal.

The IntersectionStandard class is responsible for managing the following functions:

1. Displaying the commanded aspect from the Intersection Controller.
2. Determining the aspect actually displayed.
3. Checking for discrepancies between commanded and displayed aspects.
4. Raising an error event if there is an aspect discrepancy.

TABLE 6.A9. IntersectionController Class—Attributes

Attribute	Type	Notes
NUMAPPROACHES	private: *int*	Constant defining the number of approaches in the intersection.
m_Priority	private: *int*	Indicates the relative priority of the approaches. Values are as follows: 1. E-W/W-E approach pair has priority = 1. 2. N-S/S-N approach pair has priority = 2. 3. Both approach pairs have equal priority = 3. This attribute is used to determine which of the three default states should be set when the intersection initializes or is set to operate in Default mode either by an override command or by an error condition.
m_Mode	private: *Mode*	The object m_Mode, an instance of the Mode enumeration class, indicates the method currently being used to control the intersection. Valid values for this attribute are shown in the class diagram. The setPhase( ) method checks for changes in this value at the beginning of each cycle and changes the control scheme if required. Changes to Preempt, Manual, or Remote modes are handled by specific events; these events cause the control scheme to change immediately rather than at the beginning of the next cycle.
m_CurrentPhase	private: *Phase*	This is an enumeration of class Phase that also serves as an index into the m_Split array (since C++ automatically casts enumerated types as arrays where required) denoting which portion of the cycle is currently active. The Default phase is used during initialization and in response to override commands and critical system faults. Phases GG_GG_RR_RR (1) to RR_RR_RR_RR_8 (8) are used in normal operation.

(*Continued*)

TABLE 6.A9. (*Continued*)

Attribute	Type	Notes
m_ErrorHandler	private: *ErrorHandler*	Pointer to the m_ErrorHandler object.
m_Approach	private: *Approach*	This is an array of type Approach and a length of NUMAPPROACHES. This array represents each of the four entrances to an intersection. See the Approach class for more details. The m_Approach array is declared as follows: Approach m_Approach [NUMAPPROACHES] Where NUMAPPROACHES is a compile-time constant.
m_Network	private: *Network*	This object is the instance of the Network class that provides an abstraction layer between the network resource manager and the m_IntersectionController object.
m_EmergencyPreempt	private: *PreEmpt*	This is a pointer to the instance of the PreEmpt class that provides an abstraction layer between the emergency vehicle transponder resource manager and the m_ IntersectionController object.
m_Parameters	private: *Parameters*	Structure holding the intersection parameters, which are the cycle time and the splits array.
m_RemoteOverride	private: *RemoteOverride*	This is the instance of the RemoteOverride class representing the Remote Software console. This object abstracts requests made from the off-site software control panel from the main application.
m_ManualOverride	private: *Override*	This is the instance of the Override class representing the Manual Override console. The object serves as a broker, abstracting the main application from any requests made from the Manual Override console, which is located at the site of the traffic control system.

TABLE 6.A9. (*Continued*)

Attribute	Type	Notes
m_LocalTimeZone	private: *float*	Given as an offset in hours to UTC (GMT).
m_TrafficHistory	private: *Database*	This is the instance of the Database class that is used to log statistical data regarding traffic levels at the intersection being controlled. The data is stored in the system's flash memory store. See Section 5.7 for more information about the flash memory included in the system.
m_IncidentLog	private: *Database*	This object, which is another instance of the Database class, logs abnormal events observed by the system on the site of the intersection. Data recorded by this object will be stored in the system's flash memory store. See Section 5.7 for more information about the flash memory included in the system.
m_Aspect	private: *Aspect*	Detected Aspect from each m_Approach object; Aspect [4].
m_Count	private: *int*	Vehicle count from each m_Approach object; int [4].
m_IntersectionStatus	private: *Status*	

Figure 6.A17 illustrates the attributes, methods, and events of the IntersectionStandard class.

6.8.3.3.3.1 INTERSECTIONSTANDARD RELATIONSHIPS

- Association link from class *Approach*
- Association link to class *Indication*
- Association link from class *Approach*

6.8.3.3.3.2 INTERSECTIONSTANDARD ATTRIBUTES (Table 6.A13)

6.8.3.3.3.3 INTERSECTIONSTANDARD METHODS (Table 6.A14)

6.8.3.3.3.4 CORRESPONDENCE BETWEEN INDICATIONS AND ACTUAL DISPLAYED SIGNALS Since this class is used for both the Vehicle and Pedestrian Traffic Standard objects, it is necessary to define the relationship between the attribute values and the actual displayed signal; this is shown in Table 6.A15.

TABLE 6.A10. Intersection Controller Class—Methods

Method	Type	Notes
IntersectionController ()	public:	Constructor.
~IntersectionController ()	private abstract:	Destructor.
init ()	private static: *void*	This is the first code unit executed when the equipment becomes active. This function performs the following basic tasks: Tests memory and hardware. Gathers all environmental information (initial mode, priority, approach parameters). Sets all the components of the intersection to their default states. Starts the first cycle in normal mode.
run ()	public static: *void*	
setPhase ()	public: *int*	Moves the intersection to the next phase in the cycle. This method is invoked in response to the following events: Phase timer reaches 0 (in Actuated, Fixed, and Adaptive modes). Remote Override *onSetPhase(void)* event fired (in Remote mode). Manual Override *onSetPhase(void)* event fired (in Manual mode). The following tasks are performed by this method: Changes the *m_CurrentPhase* attribute according to the assignment operation *m_CurrentPhase = (m_CurrentPhase++) mod 9*. Changes the state of the Green Signal Safety Relay as required by the new value of *m_CurrentPhase*. Checks the state of the Green Signal Safety Relay and raises an error if there is a discrepancy. Manipulates the attributes of the *m_Approach* objects as required by the new Current Phase. Calculates the phase time as *calculateTime(m_Cycle, m_Splits [m_CurrentPhase])*.

TABLE 6.A10. (*Continued*)

Method	Type	Notes
		Loads the Phase Time Remaining timer with the calculated phase time by invoking *loadTimer(calculateTime (m_Cycle, m_Splits[m_CurrentPhase]))*. Checks that the phase setting is displayed properly by the approaches and raises an error if there is a discrepancy.
setPhase (*Phase*)	public: *int*	param: phase [Phase - in] Moves the intersection to the specified phase.
getPhase ()	public: *Phase*	Determines the displayed intersection phase by querying all Aspect objects and determining their aspects. Used by the *checkPhase* method
checkPhase (*Phase*)	public: *boolean*	param: phase [Phase - in] Returns True if the displayed phase agrees with the commanded phase (passed as a parameter), False otherwise.
setCycle (*float*)	public: *int*	param: time [float - in] Mutator for the cycle time attribute.
getCycle ()	public: *float*	Accessor for the cycle time attribute.
setSplits (*Split**)	public: *int*	param: splits [Split* - inout] Mutator for the splits attribute.
getSplits ()	public: *Split**	Accessor for the splits attribute.
setMode (*Mode*)	public: *int*	param: mode [Mode - in] Mutator for the attribute m_Mode.
getMode ()	public: *Mode*	Accessor for the attribute m_Mode.
checkMode (*Mode*)	public: *boolean*	param: mode [Mode - in]
loadTimer (*float*)	public: *int*	param: time [float - in] Loads the phase timer (utilizing OS timer services) with the phase time, specified as a parameter.
onPreemptRequest ()	public: *int*	Emergency preempt request event from the m_EmergencyPreempt object. This method performs the following tasks: 1. Saves the current value of *m_Mode*. 2. Sets the mode to Preempt. 3. Sets the intersection phase to allow the emergency vehicle to pass safely under traffic signal control.

(*Continued*)

TABLE 6.A10. (*Continued*)

Method	Type	Notes
onPreemptCleared ()	public: *int*	Event that terminates preempted operation and returns the intersection to normal operating mode. This method performs the following tasks: 1. Restores the previous mode. 2. Sets the intersection to the default state. 3. Returns the intersection to normal operation.
onOverrideActivated (*OverrideType*)	public: *int*	param: type [OverrideType - in] Overrides activation event from either the *m_ManualOverride* or *m_RemoteOverride* object. The parameter type indicates which override is involved. This method performs the following tasks: 1. Saves the current value of *m_Mode*. 2. Sets the mode to Manual or Remote, depending on the value of parameter type. 3. Sets the intersection to the Default phase.
onOverrideDeactivated (*OverrideType*)	public: *int*	param: type [OverrideType - in] Override cancellation event from either the *m_ManualOverride* or *m_RemoteOverride* object. The parameter type indicates which override is involved. This method performs the following tasks: 1. Restore the previous value of *m_Mode*. 2. Set the intersection to the Default phase. 3. Returns the intersection to normal operation.
toggleGreenSafetyRelay ()	public: *int*	Toggles the state of the Green Safety Relay.
checkGreenSafetyRelay ()	public: *boolean*	Checks that the Green Safety Relay is in the proper state for the active intersection phase.
calculateParameters ()	public: *int*	Adaptive algorithm for determining intersection timing parameters for the next cycle.
calculateTime (*float, Split**)	public: *float*	param: cycle [float - in] param: split [Split* - in] Used to calculate the actual phase time from the values of *m_Parameters.cycleTime* and *m_Parameters.splits*.

TABLE 6.A10. (*Continued*)

Method	Type	Notes
setParameters ()	public: *int*	Mutator for the intersection timing parameters.
getParameters ()	public: *Parameters**	Accessor for the intersection timing parameters.
getStatus ()	public: *Status**	Method used to access the overall status of the intersection.
onPedestrianRequest ()	public: *void*	Event triggered by a valid pedestrian crossing request.
onVehicleEntry (*int*)	public: *void*	param: approach [int - in] Event triggered by a vehicle entering the vehicle detection loop.

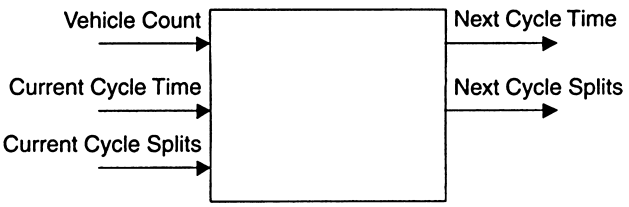


Figure 6.A12. Black box representation of adaptive algorithm.

6.8.3.3.4 OnOffSensor This class represents the pedestrian crossing request pushbuttons located on opposite sides of the crosswalk associated with an approach.

Objects of the OnOffSensor class are responsible for managing the following functions:

1. Filtering of pushbutton service requests.
2. Generation of Pedestrian Service Request event.

Figure 6.A18 below illustrates the attributes, methods, and events of the OnOffSensor class.

6.8.3.3.4.1 ONOFFSENSOR RELATIONSHIPS

- Association link from class *Approach*
- Generalization link from class *VehicleDetector*

6.8.3.3.4.2 ONOFFSENSOR ATTRIBUTES (Table 6.A16)

6.8.3.3.4.3 ONOFFSENSOR METHODS (Table 6.A17)

6.8.3.3.4.4 ONOFFSENSOR BEHAVIORAL DETAILS (Figs. 6.A19 and 6.A20)

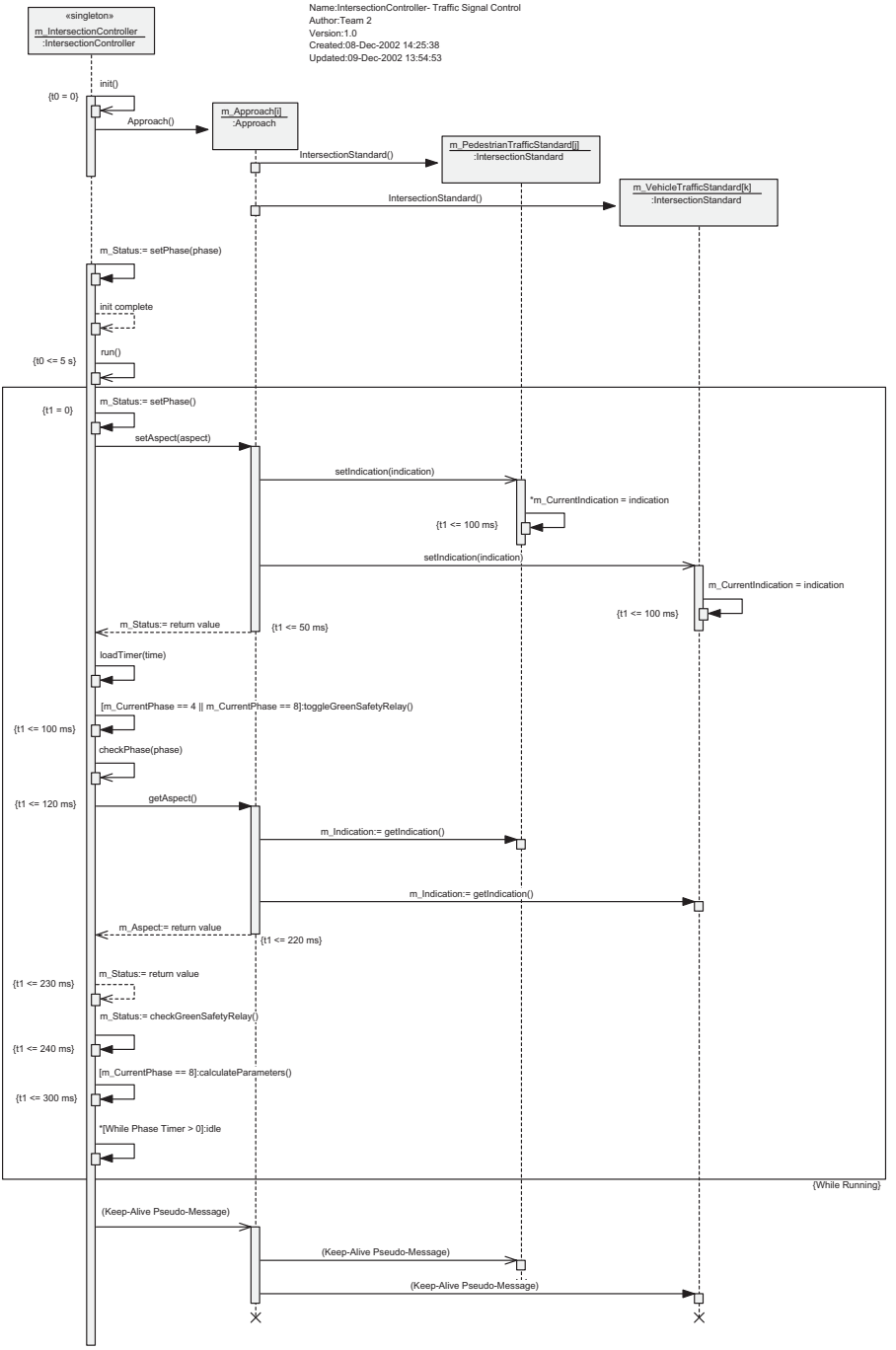


Figure 6.A13. Traffic signal control sequence diagram.

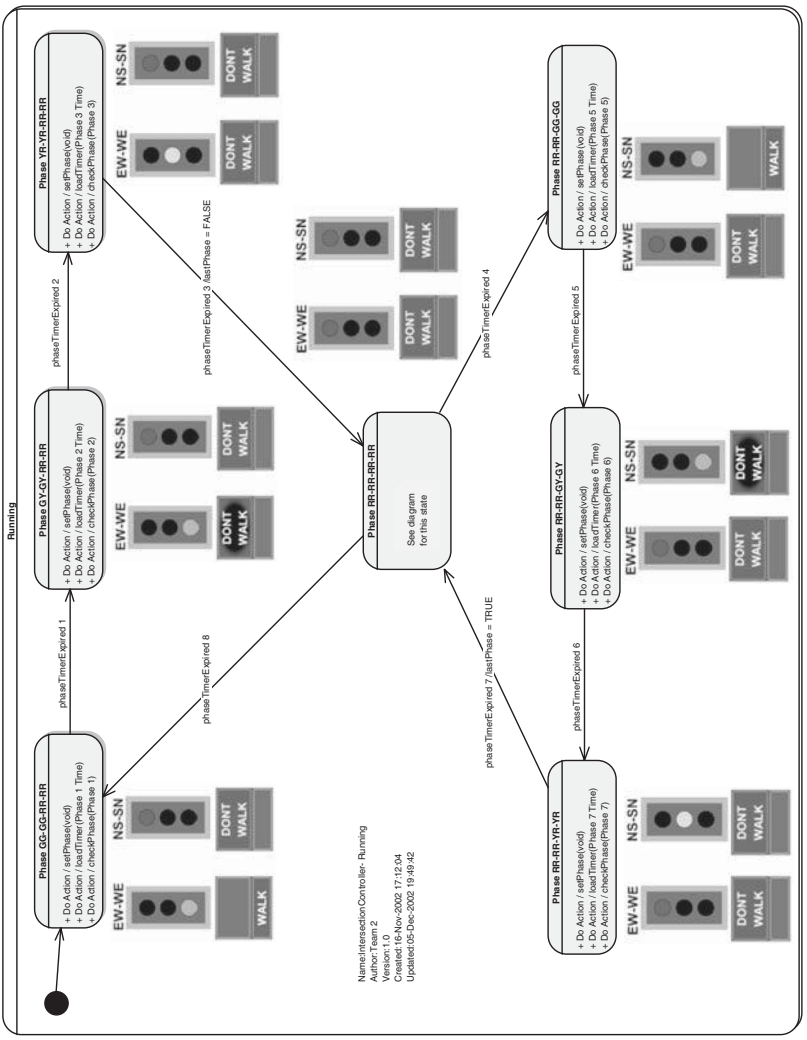


Figure 6.A14. Statechart for intersection controller phase sequence.

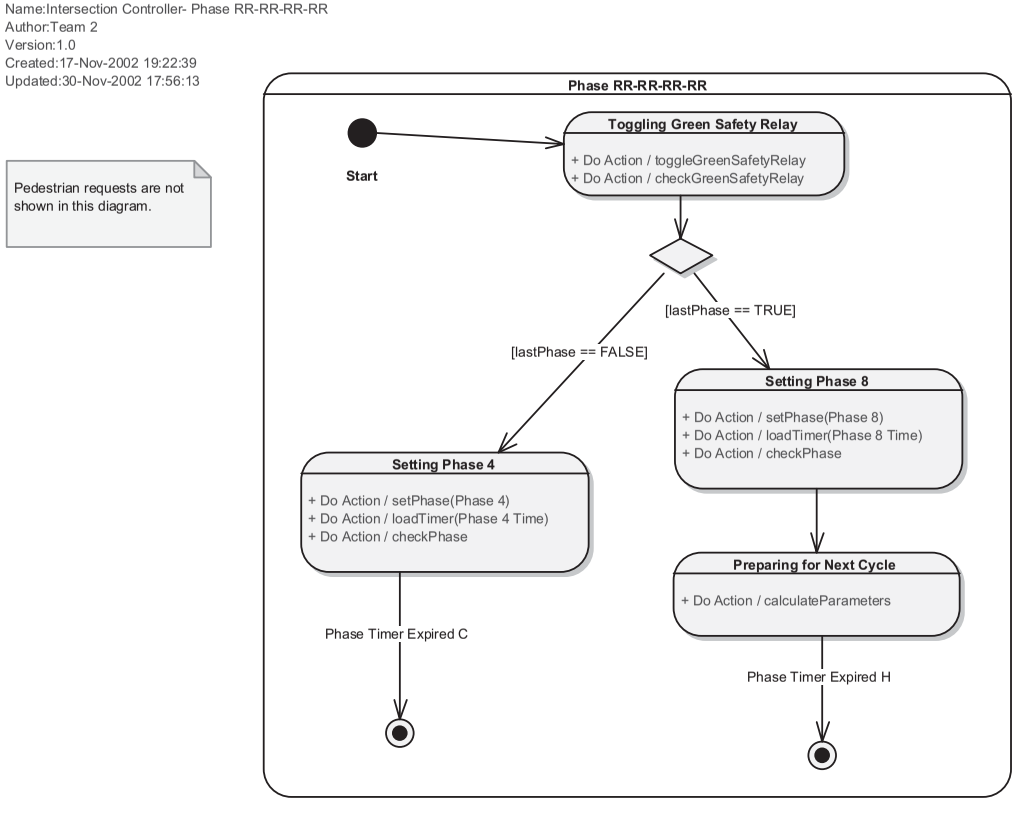


Figure 6.A15. Statechart for phases 4 and 8.

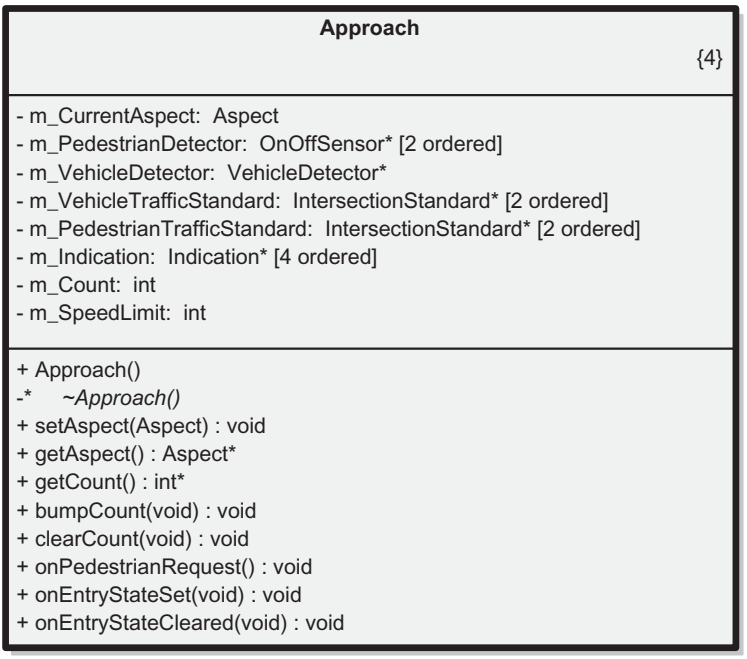


Figure 6.A16. Approach class.

6.8.3.3.5 VehicleDetector This class represents the proximity detection loop located near the stop line associated with an approach. The class is based on the OnOffSensor class.

The Vehicle Presence Detector object is responsible for managing the following functions:

1. Filtering of vehicle service requests (ACTUATED mode).
2. Generation of Vehicle Service Request event (ACTUATED mode).
3. Maintenance of the vehicle count statistic (FIXED, ACTUATED and ADAPTIVE mode).

Figure 6.A21 illustrates the attributes, methods, and events of the VehicleDetector class.

6.8.3.3.5.1 VEHICLE DETECTOR RELATIONSHIPS

- Association link from class *Approach*
- Generalization link to class *OnOffSensor*

6.8.3.3.5.2 VEHICLEDETECTOR ATTRIBUTES Inherited from superclass.

6.8.3.3.5.3 VEHICLEDETECTOR METHODS Inherited from superclass. Overridden methods are described in Table 6.A18.

TABLE 6.A11. Approach Class—Attributes

Attribute	Type	Notes
m_CurrentAspect	private: *Aspect*	Current Approach aspect corresponding to the current intersection phase.
m_PedestrianDetector	private: *OnOffSensor**	Pointer to an array of objects of the *OnOffSensor* class, which provide an abstraction layer for the pedestrian crossing request pushbuttons.
m_VehicleDetector	private: *VehicleDetector**	Pointer to an object of class *VehicleDetector* (superclass of *OnOffSensor*), providing an abstraction layer for the vehicle detection loop.
m_VehicleTrafficStandard	private: *IntersectionStandard*	Pointer to an array of *IntersectionStandard* objects representing the vehicle traffic standards associated with the approach.
m_PedestrianTrafficStandard	private: *IntersectionStandard**	Pointer to an array of *IntersectionStandard* objects representing the pedestrian traffic standards associated with the approach.
m_Indication	private: *Indication**	Pointer to an array of *Indication* objects; used to store the indication values obtained from associated traffic standards.
m_Count	private: *int*	Used to count the number of vehicles passing through the approach.
m_SpeedLimit	private: *int*	Speed limit (in km/h) associated with the approach.

TABLE 6.A12. Approach Class—Methods

Method	Type	Notes
Approach ()	public:	Constructor.
~Approach ()	private abstract:	Destructor.
setAspect (*Aspect*)	public: *void*	param: aspect [Aspect - in] Mutator for attribute *m_CurrentAspect*.
getAspect ()	public: *Aspect**	Accessor used to fetch the aspect actually being displayed by the set of approach traffic standards.
getCount ()	public: *int**	Accessor for the *m_Count* attribute.
bumpCount (*void*)	public: *void*	Method called to increment the attribute *m_Count* by 1.
clearCount (*void*)	public: *void*	Method called to set the attribute *m_Count* to 0.
onPedestrianRequest ()	public: *void*	Event triggered by a valid pedestrian crossing request from one of the pedestrian request pushbuttons associated with the approach.
onEntryStateSet (*void*)	public: *void*	Event triggered when the vehicle detector attribute *m_State* is set.
onEntryStateCleared (*void*)	public: *void*	Event triggered when the vehicle detector attribute *m_State* is cleared.

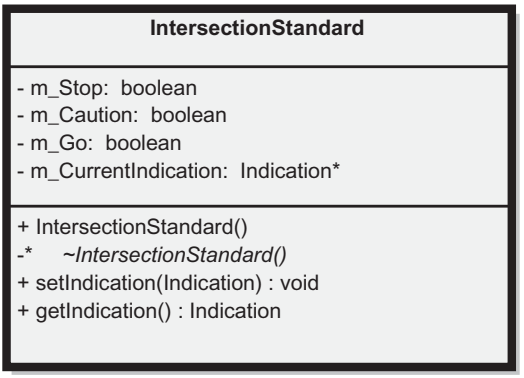


Figure 6.A17. IntersectionStandard class.

TABLE 6.A13. Intersection Standard Class—Attributes

Attribute	Type	Notes
m_Stop	private: *boolean*	A Boolean value indicating that the signal is commanded to show a Stop signal (corresponding to an Indication value of R).
m_Caution	private: *boolean*	A Boolean value indicating that the signal is commanded to show a Caution signal (corresponding to an Indication value of Y).
m_Go	private: *boolean*	A Boolean value indicating that the signal is commanded to show a Go signal (corresponding to an Indication value of G).
m_CurrentIndication	private: *Indication*	An instance of the Indication enumerated class indicating the current traffic signal to be displayed.

TABLE 6.A14. Intersection Standard Class—Methods

Method	Type	Notes
IntersectionStandard ()	public:	Constructor.
~IntersectionStandard ()	private abstract:	Destructor.
setIndication (*Indication*)	public: *void*	param: indication [Indication - in] Mutator for the m_CurrentIndication attribute. The method performs the following: 1. Check whether the commanded aspect is valid. If not, raise an error. 2. If the commanded aspect is valid, display it.
getIndication ()	public: *Indication*	Accessor for determining the value of the indication actually being displayed.

TABLE 6.A15. Attribute and Signal Correspondence

m_CurrerntIndication	m_Stop	m_Caution	m_Go	Vehicle Standard	Pedestrian Standard
R	True	False	False	Red	DON'T WALK
Y	False	True	False	Amber	Flashing DON'T WALK
G	False	False	True	Green	WALK

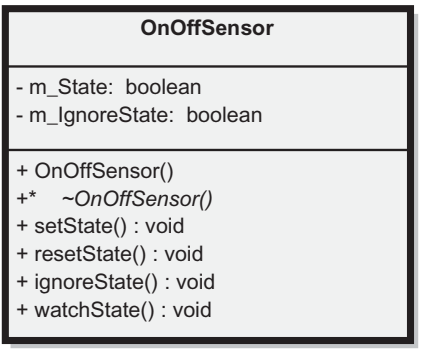


Figure 6.A18. OnOffSensor class.

TABLE 6.A16. OnOff Sensor Class—Attributes

Attribute	Type	Notes
m_State	private: *boolean*	Indicates whether or not a valid pedestrian service request has been made since the last time the value was reset.
m_IgnoreState	private: *boolean*	A value that indicates whether subsequent pedestrian service requests should raise an event or simply be ignored.

TABLE 6.A17. OnOff Sensor Class—Methods

Method	Type	Notes
OnOffSensor ()	public:	Constructor.
~OnOffSensor ()	public abstract:	Destructor.
setState ()	public: *void*	Sets the object's m_State attribute to True indicating that a pedestrian service request is pending.
resetState ()	public: *void*	Sets the object's state attribute to False to indicate that any previous pedestrian service requests have been completed.
ignoreState ()	public: *void*	Sets the object's m_IgnoreState attribute to True indicating that subsequent pedestrian requests are to be ignored.
watchState ()	public: *void*	Sets the object's m_IgnoreState attribute to False indicating that subsequent pedestrian requests are to be processed.

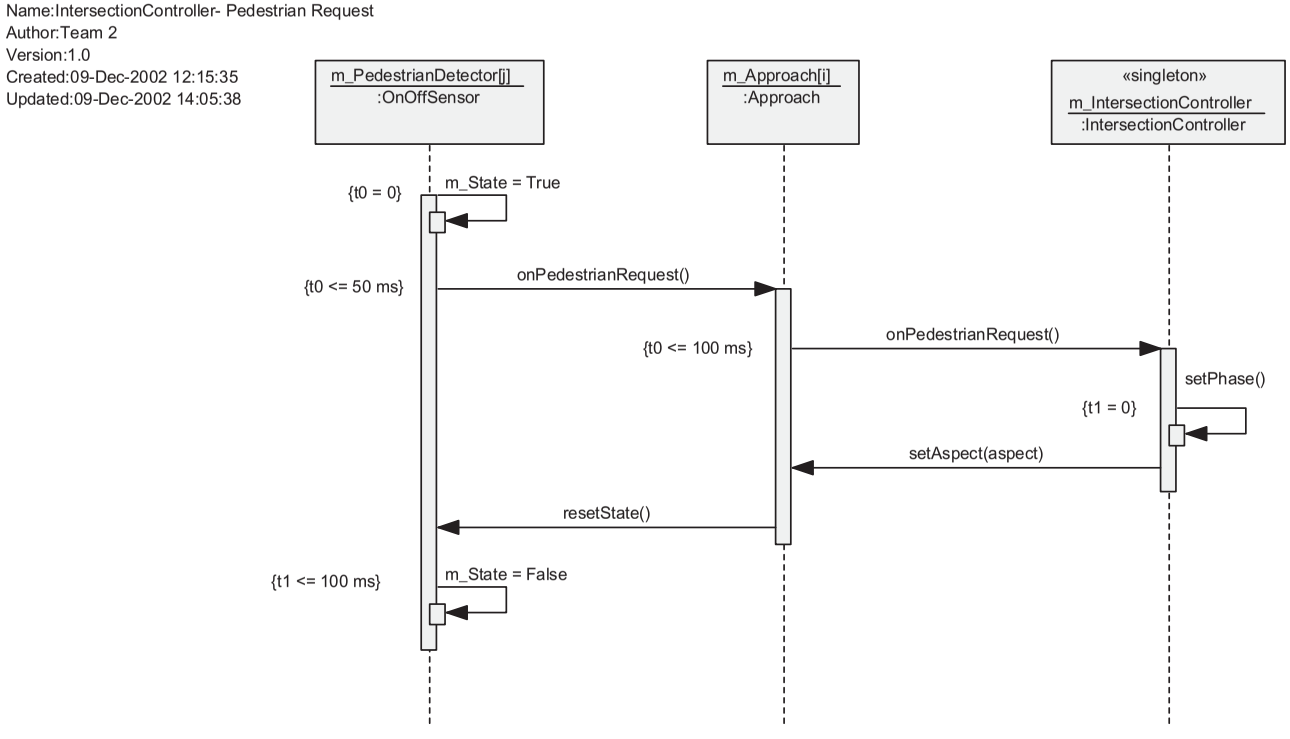


Figure 6.A19. OnOffSensor sequence diagram.

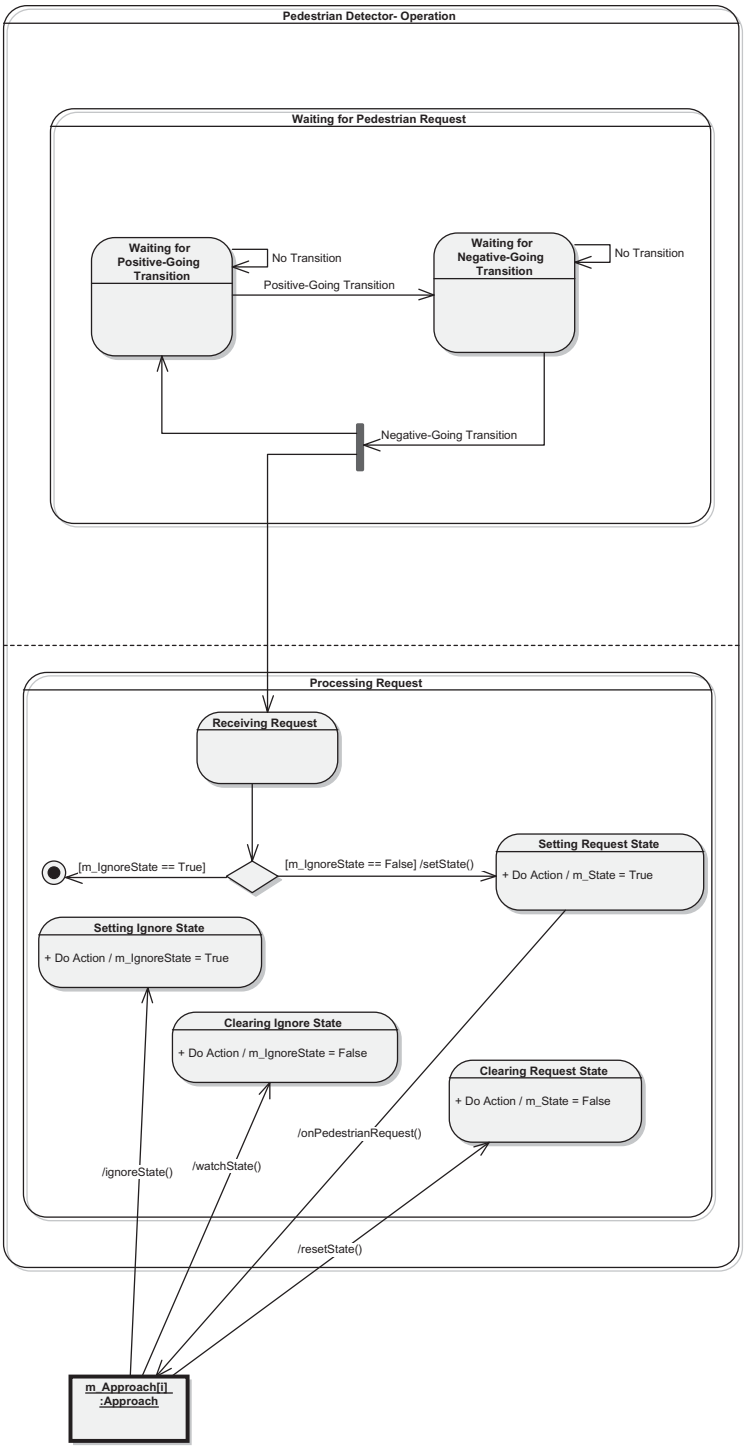


Figure 6.A20. OnOffSensor statechart.

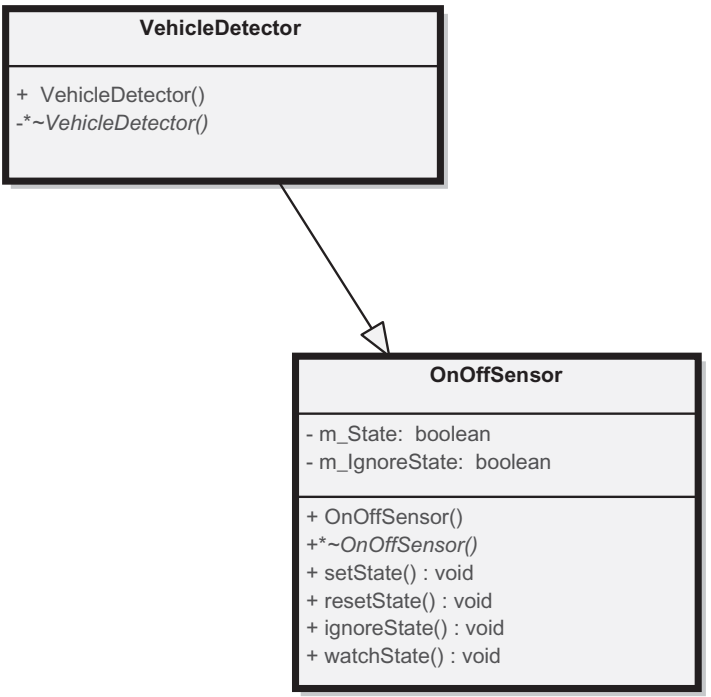


Figure 6.A21. VehicleDetector class.

TABLE 6.A18. VehicleDetector Class—Attributes

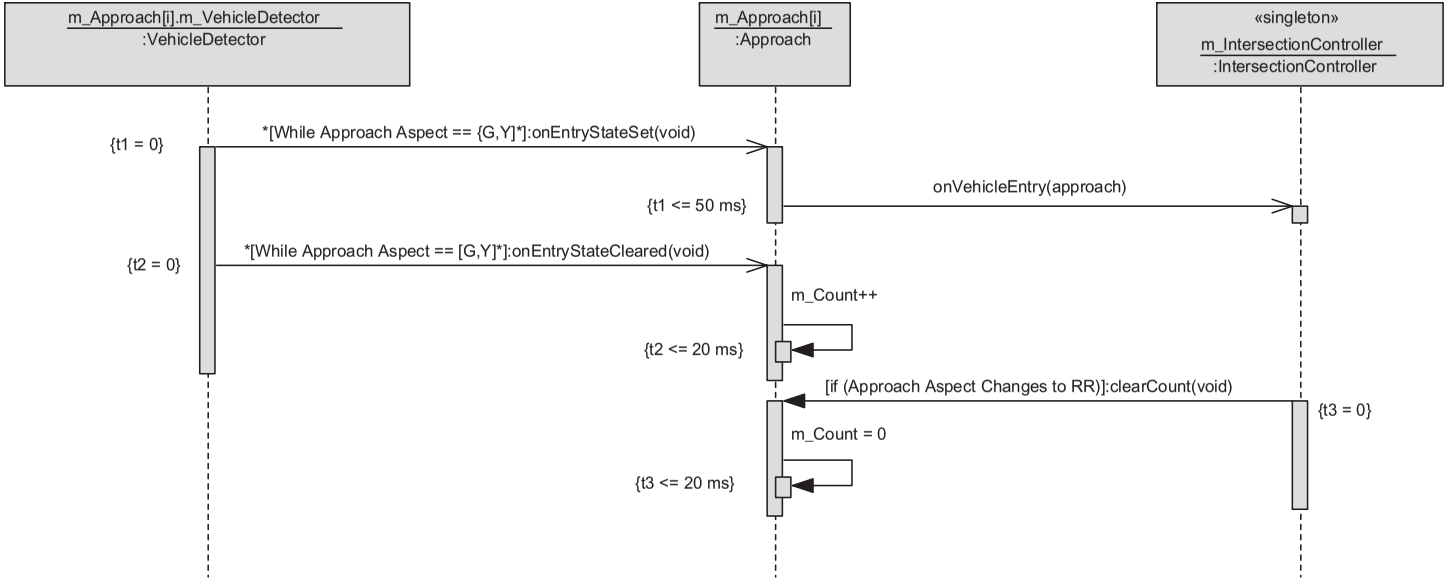
Method	Type	Notes
VehicleDetector ()	public:	Constructor.
~VehicleDetector ()	private abstract:	Destructor.
setState ()	public: *void*	Sets the *m_State* attribute and triggers the *onVehicleEntry* event.
resetState ()	public: *void*	Clears the *m_State* attribute and triggers the *onVehicleExit* event.

6.8.3.3.5.4 VEHICLEDETECTOR BEHAVIORAL DETAILS (Figs. 6.A22 and 6.A23)

6.8.3.3.6 Override This class represents the set of pushbuttons on the manual override console (Fig. 6.A24).

6.8.3.3.6.1 OVERRIDE RELATIONSHIPS

- Dependency link to class *OverrideType*
- Association link to class *IntersectionController*
- Generalization link from class *RemoteOverride*



Name:IntersectionController- Vehicle Detector
Author:Team 2
Version:1.0
Created:07-Dec-2002 21:54:19
Updated:09-Dec-2002 13:58:06

Figure 6.A22. VehicleDetector class sequence diagram.

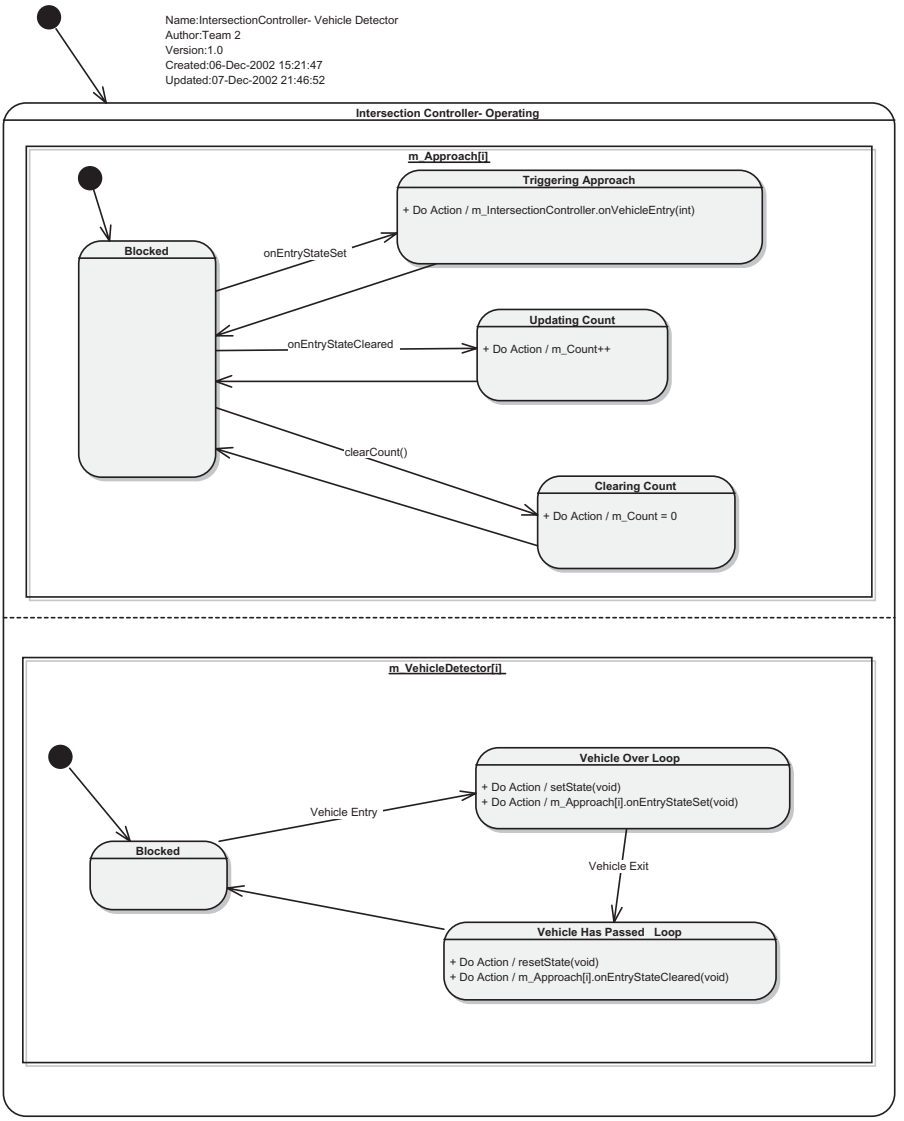


Figure 6.A23. VehicleDetector class statechart.

6.8.3.3.6.2 OVERRIDE ATTRIBUTES (Table 6.A19)

6.8.3.3.6.3 OVERRIDE METHODS (Table 6.A20)

6.8.3.3.6.4 OVERRIDE BEHAVIORAL DETAILS (Fig. 6.A25)

6.8.3.3.7 RemoteOverride This class represents the commands available on the Remote Software console. Additionally, the class provides an interface for

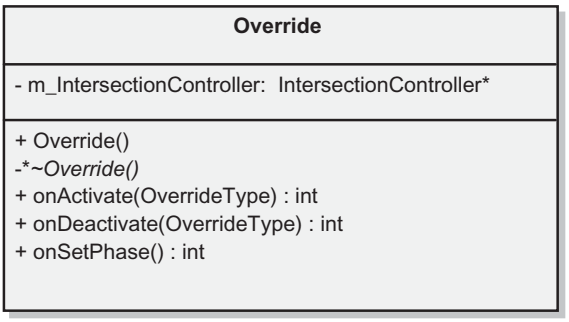


Figure 6.A24. Override class.

TABLE 6.A19. Override Class—Attributes

Attribute	Type	Notes
m_ IntersectionController	private: *IntersectionController*	Pointer to the m_IntersectionContoller object.

TABLE 6.A20. Override Class—Methods

Method	Type	Notes
Override ()	public:	Constructor.
~Override ()	private abstract:	Destructor.
onActivate (*OverrideType*)	public: *int*	param: type [OverrideType - in] Event triggered by receipt of an activation command from the local override console.
onDeactivate (*OverrideType*)	public: *int*	param: type [OverrideType - in] Event triggered by receipt of a deactivation command from the local override console.
onSetPhase ()	public: *int*	Event triggered by receipt of an advance phase command from the local override console.

remote access to and update of intersection traffic data and cycle parameters for coordinated intersection control (option).

The RemoteOverride class is responsible for managing the following functions:

1. Triggering the appropriate mode change.
2. Generation and handling of events required to control intersection phase.

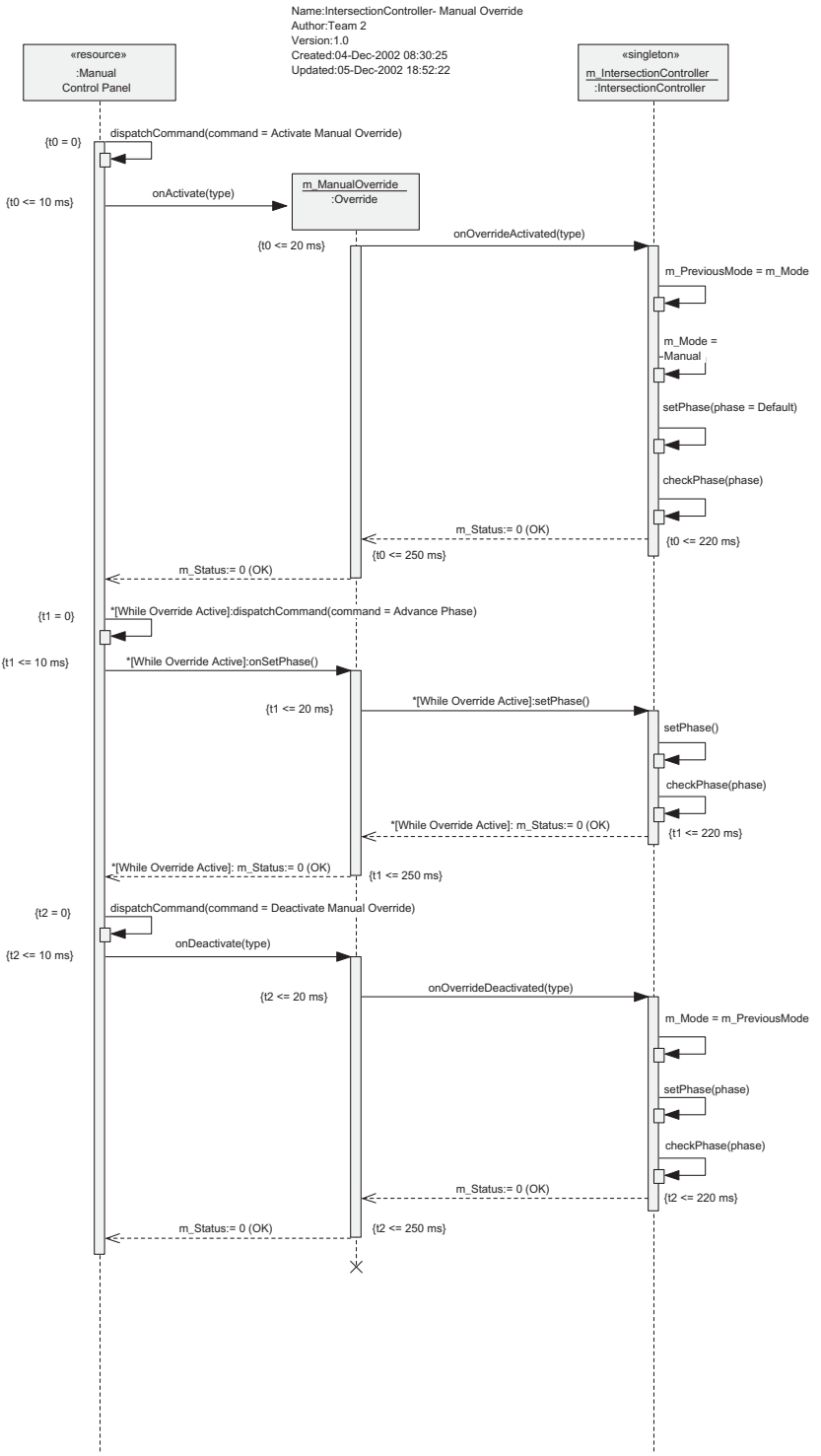


Figure 6.A25. Override class sequence diagram.

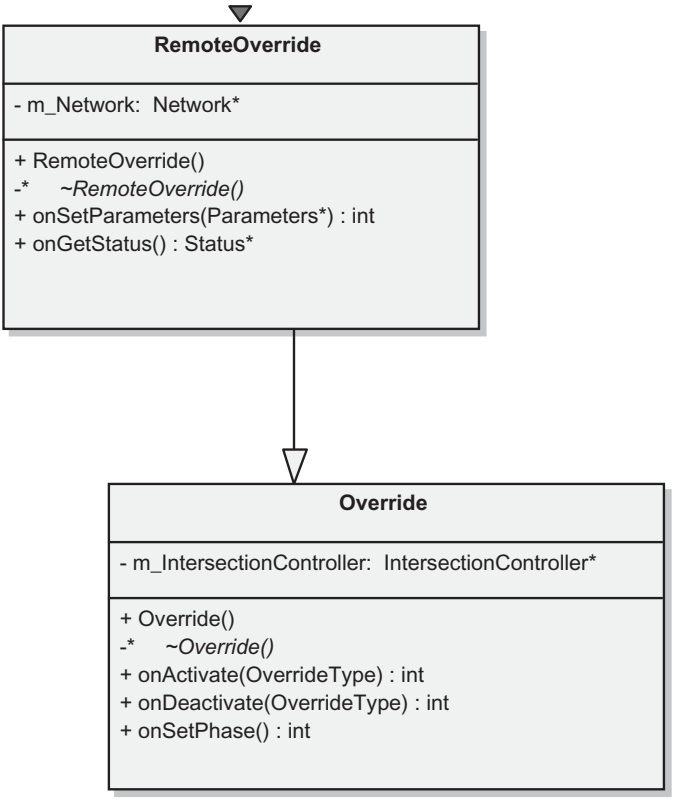


Figure 6.A26. Remote override class.

3. Acting as a substitute for the Calculate Cycle Parameters method of the Intersection Control object (in coordinated mode, not covered by this specification).

Figure 6.A26 illustrates the attributes, methods, and events of the Remote Override class.

6.8.3.3.7.1 REMOTE OVERRIDE RELATIONSHIPS

- Dependency link to class *Status*
- Generalization link to class *Override*
- Association link from class *IntersectionController*
- Association link to class *Network*

6.8.3.3.7.2 REMOTEOVERRIDE ATTRIBUTES In addition to those inherited from the superclass Override, RemoteOverride attributes are as follows (Table 6.A21):

TABLE 6.A21. RemoteOverride Class—Attributes

Attribute	Type	Notes
m_Network	private: *Network*	Pointer to the m_Network object.

TABLE 6.A22. RemoteOverride Class—Methods

Method	Type	Notes
RemoteOverride ()	public:	Constructor.
~RemoteOverride ()	private abstract:	Destructor.
onSetParameters (*Parameters**)	public: *int*	param: parameters [Parameters* - in] Event triggered under coordinated control; used to set the intersection timing parameters under remote control. Completes within 100 ms.
onGetStatus ()	public: *Status**	Event triggered under coordinated control; used to get the intersection timing parameters under remote control. Completes within 100 ms.

6.8.3.3.7.3 REMOTEOVERRIDE METHODS In addition to those inherited from the superclass Override, RemoteOverride methods are as follows (Table 6.A22):

6.8.3.3.7.4 REMOTEOVERRIDE BEHAVIORAL DETAILS Behavior of the RemoteOverride class is identical to that of the Override class for methods inherited from the superclass.

6.8.3.3.8 PreEmpt This class manages the wireless transponder interface to authorized emergency vehicles and accesses the m_IntersectionControl object in order to display the correct traffic signals, allowing the emergency vehicle priority access to the intersection.

The PreEmpt class is responsible for managing the following functions:

1. Triggering the appropriate mode change.
2. Reception of emergency vehicle preemption requests.
3. Decryption and validation of emergency vehicle preemption requests.
4. Generation and handling of events required to control intersection phase.

Figure 6.A27 illustrates the attributes, methods, and events of the PreEmpt class.

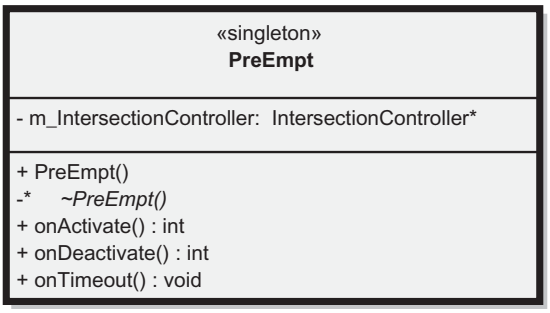


Figure 6.A27. PreEmpt class.

TABLE 6.A23. PreEmpt Class—Attributes

Attribute	Type	Notes
m_IntersectionController	private: *IntersectionController*	Pointer to the m_Intersection controller object.

TABLE 6.A24. PreEmpt Class—Methods

Method	Type	Notes
PreEmpt ()	public:	Constructor.
~PreEmpt ()	private abstract:	Destructor.
onActivate ()	public: *int*	Event triggered by receipt of an activate signal from the emergency vehicle transponder.
onDeactivate ()	public: *int*	Event triggered by receipt of a deactivate signal from the emergency vehicle transponder.
onTimeout ()	public: *void*	Event triggered if a deactivate signal is not received after the timeout interval has elapsed.

6.8.3.3.8.1 PREEMPT RELATIONSHIPS

- Association link from class *IntersectionController*
- Association link to class *IntersectionController*

6.8.3.3.8.2 PREEMPT ATTRIBUTES (Table 6.A23)

6.8.3.3.8.3 PREEMPT METHODS (Table 6.A24)

6.8.3.3.8.4 PREEMPT BEHAVIORAL DETAILS (Fig. 6.A28)

6.8.3.3.9 Network This class manages communication via the Ethernet port.

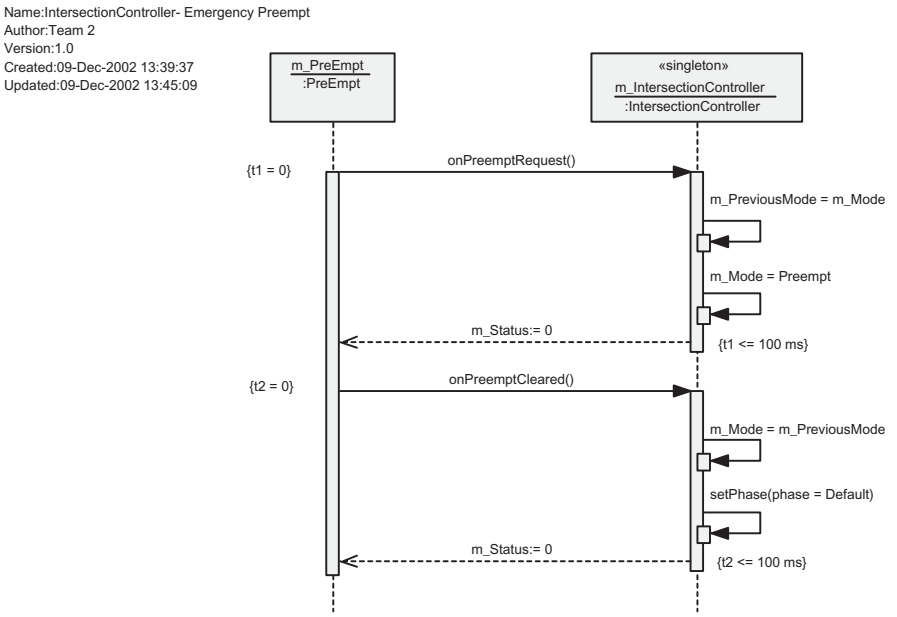


Figure 6.A28. PreEmpt sequence diagram.

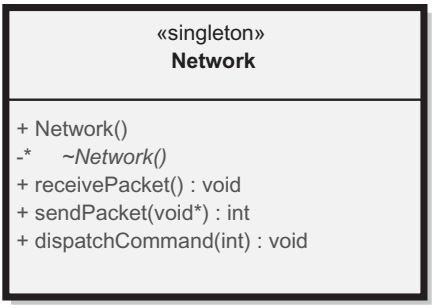


Figure 6.A29. Network class.

Figure 6.A29 illustrates the attributes, methods, and events of the Network Interface class.

6.8.3.3.9.1 NETWORK RELATIONSHIPS

- Association link from class *IntersectionController*
- Association link from class *Maintenance*
- Association link from class *RemoteOverride*

6.8.3.3.9.2 NETWORK METHODS (Table 6.A25)

TABLE 6.A25. Network Class—Methods

Method	Type	Notes
Network ()	public:	Constructor.
~Network ()	private abstract:	Destructor.
receivePacket ()	public: *void*	Method responsible for receiving network SNMP packets.
sendPacket (*void**)	public: *int*	param: packet [void* - in] Method responsible for sending network SNMP packets.
dispatchCommand (*int*)	public: *void*	param: command [int - in] Interprets the received SNMP packet and invokes the appropriate method in response.

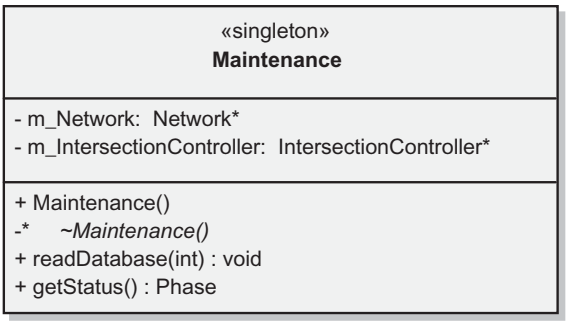


Figure 6.A30. Maintenance class.

6.8.3.3.10 Maintenance This class provides a maintenance interface to the intersection controller, accessible either from the local maintenance Ethernet port or the DOT WAN.

The Maintenance class is responsible for managing the following functions:

1. Retrieval of database information.
2. Retrieval of current intersection controller status. (Fig. 6.A30)

6.8.3.3.10.1 MAINTENANCE RELATIONSHIPS

- Association link to class *IntersectionController*
- Association link to class *Network*

6.8.3.3.10.2 MAINTENANCE ATTRIBUTES (Table 6.A26)

6.8.3.3.10.3 MAINTENANCE METHODS (Table 6.A27)

TABLE 6.A26. Maintenance Class—Attributes

Attribute	Type	Notes
m_Network	private: *Network*	Pointer to the *m_Network* object.
m_IntersectionController	private: *IntersectionController*	Pointer to the *m_IntersectionController* object.

TABLE 6.A27. Maintenance Class—Methods

Method	Type	Notes
Maintenance ()	public:	Constructor.
~Maintenance ()	private abstract:	Destructor.
readDatabase (*int*)	public: *void*	param: database [int - in] Method to read the contents of the database specified by the parameter *database*.
getStatus ()	public: *Phase*	Method to get the intersection status.

6.8.3.3.11 Database (Traffic History; Incident Log) Instances of this class are used to store the Traffic History and the Incident Log for the intersection being controlled.

The Traffic History object is responsible for managing the following functions:

1. Storage and retrieval of traffic history database records.
2. Clearing of traffic history in response to a command from a remote host.

Figure 6.A31 illustrates the attributes, methods, and events of the Traffic History class.

6.8.3.3.11.1 DATABASE RELATIONSHIPS

- Association link to class *Record*
- Association link from class *IntersectionController*
- Association link from class *IntersectionController*

6.8.3.3.11.2 DATABASE ATTRIBUTES (Table 6.A28)

6.8.3.3.11.3 DATABASE METHODS (Table 6.A29)

6.8.3.3.12 Record This class defines the attributes and methods used by records contained in object instances of the Database class (Fig. 6.A32).

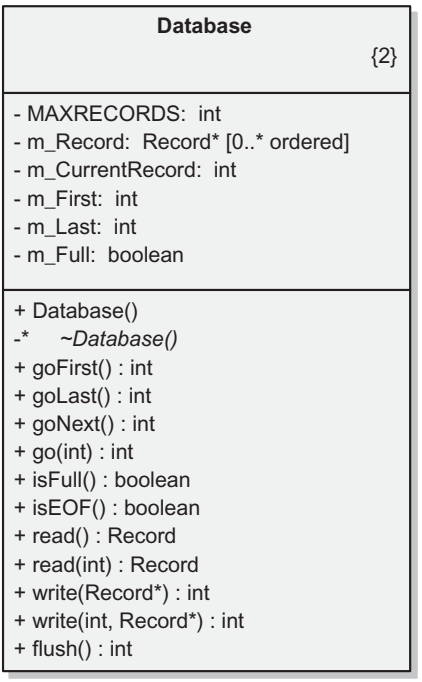


Figure 6.A31. Database class.

TABLE 6.A28. Database Class—Attributes

Attribute	Type	Notes
MAXRECORDS	private: *int*	Constant defining the maximum number of records permitted.
m_Record	private: *Record*	Pointer to database records, which are of type Record.
m_CurrentRecord	private: *int*	Position (index) of current record.
m_First	private: *int*	Position (index) of first (least recent) record in FIFO database structure.
m_Last	private: *int*	Position (index) of last (most recent) record in FIFO database structure.
m_Full	private: *boolean*	True if data is being overwritten.

TABLE 6.A29. Database Class—Methods

Method	Type	Notes
Database ()	public:	Constructor.
~Database ()	private abstract:	Destructor.
goFirst ()	public: *int*	Moves cursor to first (least recent) record. Completes in 40 ms.
goLast ()	public: *int*	Moves cursor to last (most recent) record. Completes in 40 ms.
goNext ()	public: *int*	Moves cursor to the next record. Completes in 40 ms.
Go (*int*)	public: *int*	param: record [int - in] Move cursor to the specified record. Completes in 40 ms.
isFull ()	public: *boolean*	True if the database is full. Subsequent writes will overwrite oldest data (FIFO).
isEOF ()	public: *boolean*	True when the cursor is at the last record.
Read ()	public: *Record*	Reads record at current position. Completes in 10 ms.
Read (*int*)	public: *Record*	param: position [int - in] Reads record at specified position; updates current record to specified position. Completes in 50 ms.
Write (*Record**)	public: *int*	param: record [Record* - inout] Adds new record to end of database. If isFull() is True, data will be overwritten. Completes in 50 ms.
Write (*int, Record**)	public: *int*	param: position [int - in] param: record [Record* - inout] Overwrites record at specified position; updates current record to specified position. Completes in 50 ms.
Flush ()	public: *int*	Clears all records by setting first and last logical record positions to zero; moves cursor to first physical record position. Completes in 200 ms.

6.8.3.3.12.1 RECORD RELATIONSHIPS

- Association link from class *Database*

6.8.3.3.12.2 RECORD ATTRIBUTES (Fig. 6.A33)

6.8.3.3.12.3 RECORD METHODS (Fig. 6.A34)

6.8.3.3.13 ErrorHandler This class handles all errors generated by the application. All errors are generated by the IntersectionController class in response either to internal errors or error returns from method calls.

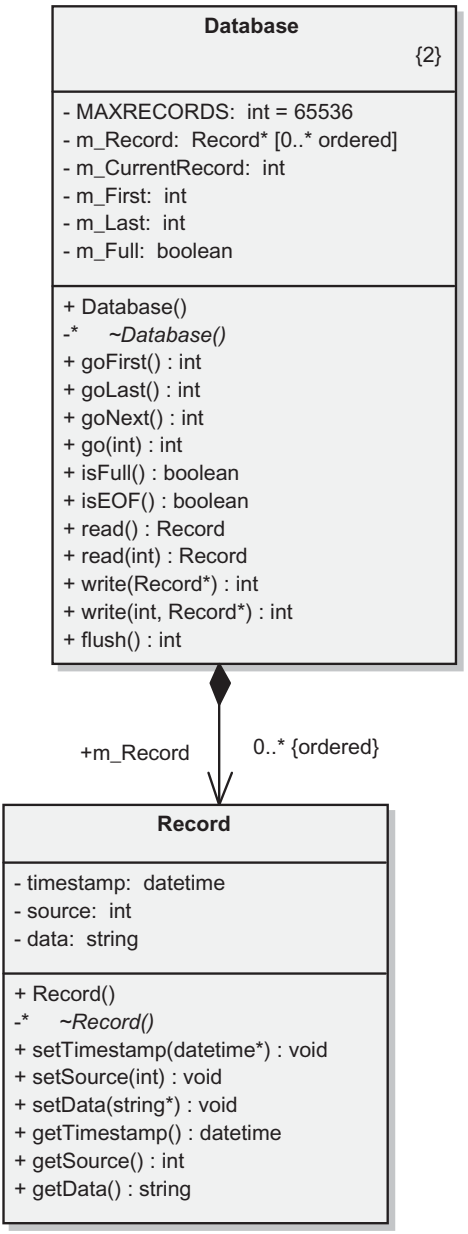


Figure 6.A32. Record class.

Attribute	Type	Notes
timestamp	private: *datetime*	Date and time of the incident or traffic history entry.
source	private: *int*	Integer value representing the object that is the source of the database record.
data	private: *string*	String of bytes containing the actual data.

Figure 6.A33. Record class—attributes.

Method	Type	Notes
Record ()	public:	Constructor.
~Record ()	private abstract:	Destructor.
setTimestamp (*datetime*)	public: *void*	param: timestamp [datetime - inout] Mutator for m_Timestamp attribute.
setSource (*int*)	public: *void*	param: source [int - in]
setData (*string*)	public: *void*	param: data [string - inout] Mutator for m_Data attribute.
getTimestamp ()	public: *datetime*	Accessor for m_Timestamp attribute.
getSource ()	public: *int*	Accessor for m_Source attribute.
getData ()	public: *string*	Accessor for m_Data attribute.

Figure 6.A34. Record class—methods.

6.8.3.3.13.1 ERRORHANDLER RELATIONSHIPS

- Association link from class *IntersectionController*

6.8.3.3.13.2 ERRORHANDLER METHODS (Fig. 6.A35)

6.8.3.3.13.3 ERRORHANDLER BEHAVIORAL DETAILS (Figs. 6.A36 and 6.A37)

6.8.3.3.14 Support Classes These comprise the structures and enumerated classes used to define attributes in the classes detailed above.

6.8.3.3.14.1 SPLIT (Fig. 6.A38)
Percentage of cycle time per phase. Comprises the nominal phase time plus the calculated extension due to traffic volume.

The values are determined as follows:

1. In FIXED mode, the nominal times are used (i.e., the extensions are set to zero).
2. In ACTUATED mode, the extensions contain fixed values at the start of each cycle. These values are modified in response to Vehicle Entry and Pedestrian Request events.
3. In ADAPTIVE mode, the extensions are updated prior to the start of each cycle as determined by the calculateParameters( ) method of the m_IntersectionController object.

6.8.3.3.14.1.1 Split Relationships

- Association link from class *Parameters*

6.8.3.3.14.2 PARAMETERS (Fig. 6.A39)

6.8.3.3.14.2.1 Parameters Relationships

- Association link from class *Status*
- Association link to class *Split*
- Association link from class *IntersectionController*

6.8.3.3.14.3 STATUS (Fig. 6.A40)

6.8.3.3.14.3.1 Status Relationships

- Association link to class *Parameters*
- Association link to class *IntersectionController*
- Dependency link from class *RemoteOverride*
- Association link to class *Mode*
- Association link to class *Phase*

Method	Type	Notes
ErrorHandler ()	public:	Constructor.
~ErrorHandler ()	private abstract:	Destructor.
onNonCriticalError (*int*)	public: *void*	param: error [int - in] Logs the error incident and resumes normal operation.
onCriticalError (*int*)	public: *void*	param: error [int - in] Attempts to set the intersection to the default phase. If unsuccessful, attempts a reset. If this fails or the error occurs again immediately after reset, the watchdog timer will override software error handling. Logs the error and sends a network message to the DOT central office via the DOT WAN.

Figure 6.A35. ErrorHandler class—methods.

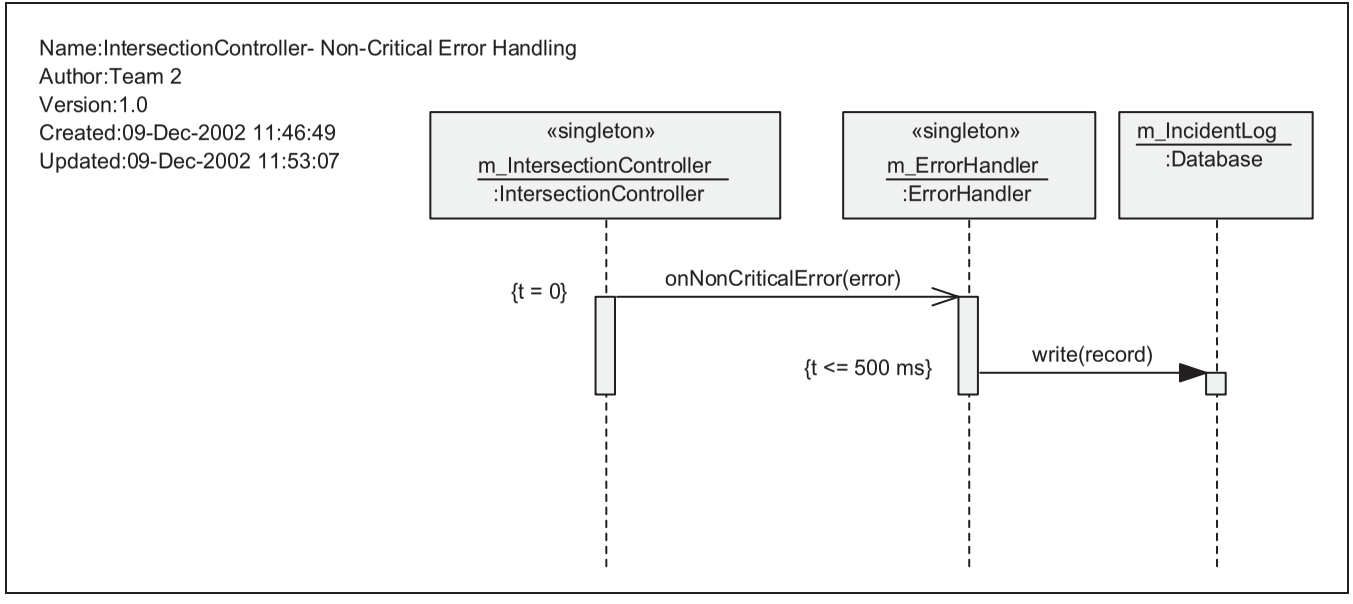


Figure 6.A36. Noncritical error sequence diagram.

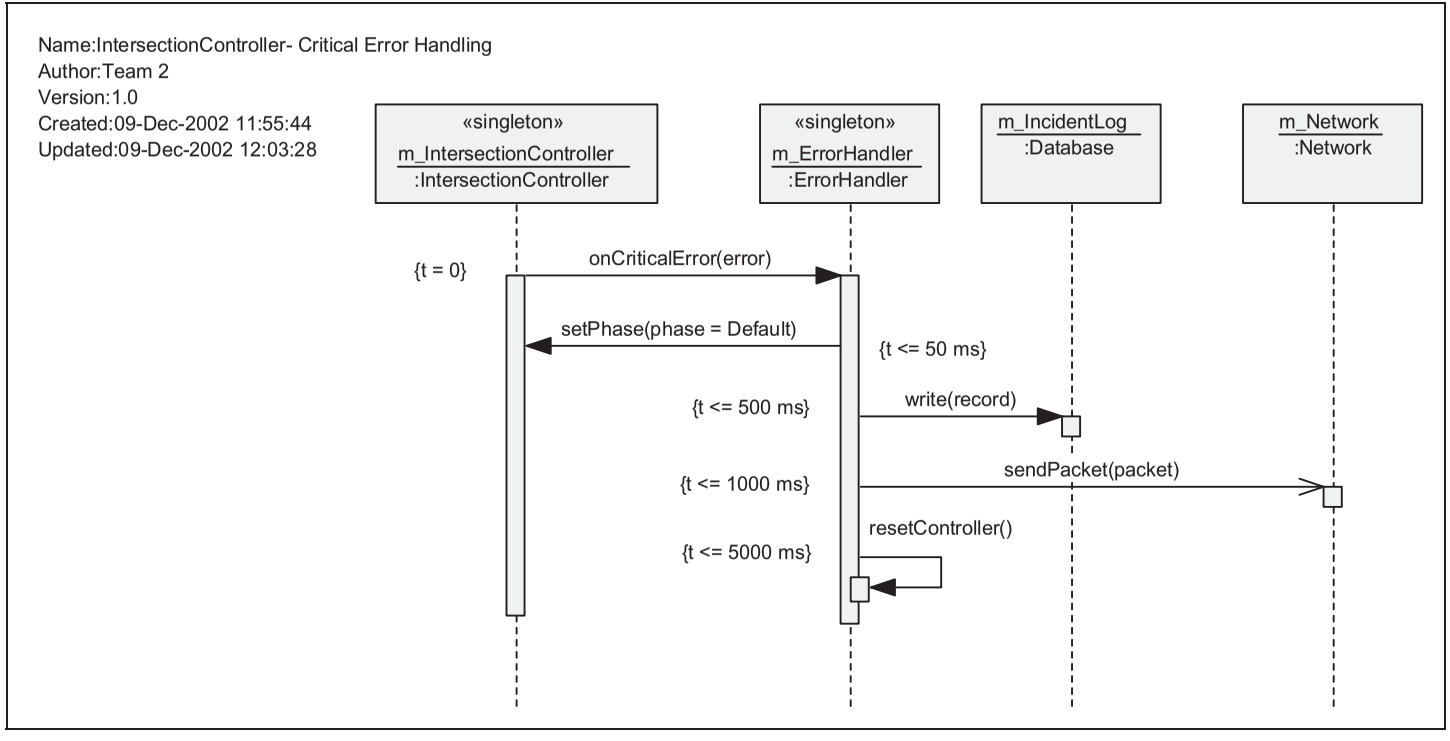


Figure 6.A37. Critical error sequence diagram.

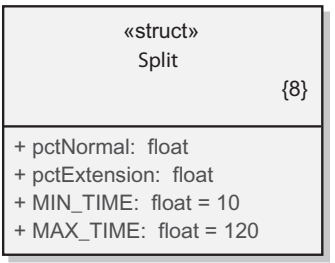


Figure 6.A38. Split class.

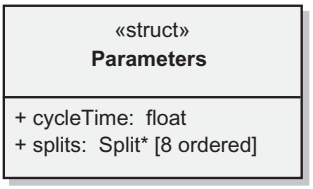


Figure 6.A39. Parameters class.

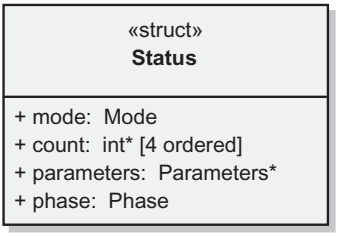


Figure 6.A40. Status class.

6.8.3.3.14.4 PHASE (Fig. 6.A41)

6.8.3.3.14.4.1 Phase Relationships

- Association link from class *IntersectionController*
- Association link from class *Status*

6.8.3.3.14.5 ASPECT (Fig. 6.A42)

6.8.3.3.14.5.1 Aspect Relationships

- Association link from class *Approach*

6.8.3.3.14.6 INDICATION (Fig. 6.A43)

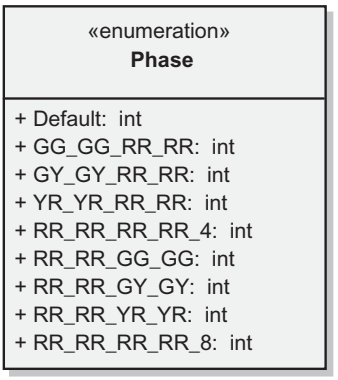


Figure 6.A41. Phase class.

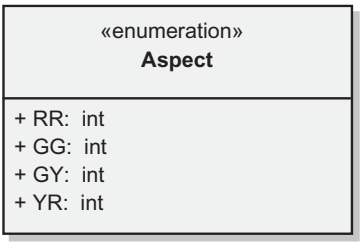


Figure 6.A42. Aspect class.

6.8.3.3.14.6.1 Indication Relationships

- Association link from class *IntersectionStandard*

6.8.3.3.14.7 MODE (Fig. 6.A44)

6.8.3.3.14.7.1 Mode Relationships

- Association link from class *Status*
- Association link from class *IntersectionController*

6.8.3.3.14.8 OVERRIDE TYPE (Fig. 6.A45)

6.8.3.3.14.8.1 OverrideType Relationships

- Dependency link from class *Override*

6.8.4 Requirements Traceability

Tables 6.A30–6.A32 illustrate SDD compliance with the SRS requirements.

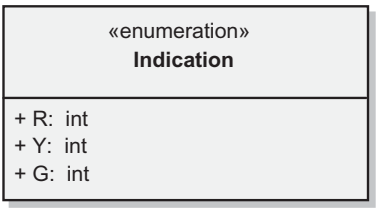


Figure 6.A43. Indication class.

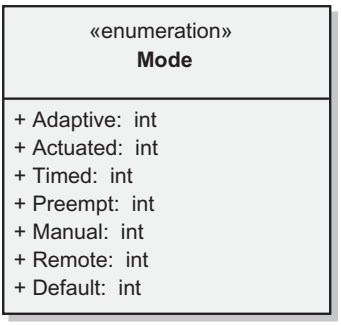


Figure 6.A44. Mode class.

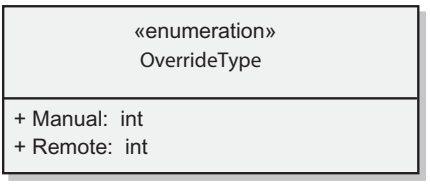


Figure 6.A45. OverrideType class.

TABLE 6.A30. Architectural Requirements

Section Reference for SRS Requirement	SDD Section Demonstrating Compliance	Comments
.2.5(10)	.2.2.2	Application software to be written in high-order OO language; C++ selected
.2.6(2)	.2.2.1	Commercial RTOS
.2.6(3)	.2.2.3	Resource managers

Note: All the SRS section references have an additional prefix "5.7" and all the SDD references have a prefix "6.8". For ease of reading, these are not shown.

TABLE 6.A31. Functional Requirements

Section Reference for SRS Requirement	SDD Section Demonstrating Compliance	Comments
.2.6(1)	.3.3.2.2	SI units; speed limit is in km/h
.3.1.1, Figure 5.A3	.3.1, Figure 6.A3	Use cases and collaborations
.3.2, Figure 5.A4	.3.2, Figure 6.A10	Class model
.3.2.1	.3.1, .3.3.1	Requirements for Intersection Controller class
.3.2.2	.3.1, .3.3.2	Requirements for Approach class
.3.2.3	.3.1, .3.3.3	Requirements for Pedestrian Traffic Standard class
.3.2.4	.3.1, .3.3.3	Requirements for Vehicle Traffic Standard class
.3.2.5	.3.1, .3.3.4	Requirements for Pedestrian Service Button class
.3.2.6	.3.1, .3.3.5	Requirements for Vehicle Presence Detector class
.3.2.7	.3.1, .3.3.6	Requirements for Manual Override class
.3.2.8	.3.1, .3.3.7	Requirements for Remote Override class
.3.2.9	.3.1, .3.3.8	Requirements for Emergency Vehicle Interface class
.3.2.10	.3.1, .3.3.9, .3.3.10	Requirements for Network Interface class
.3.2.11	.3.1, .3.3.11	Requirements for Traffic History class
.3.2.12	.3.1, .3.3.11	Requirements for Incident Log class

Note: All the SRS section references have an additional prefix "5.7" and all the SDD references have a prefix "6.8". For ease of reading, these are not shown.

TABLE 6.A32. Timing Requirements

Section Reference for SRS Requirement	SDD Section Demonstrating Compliance	Comments
.3.3.1.1, Table 5.A14 (1)	.3.3.1.4, Figure 6.A13	Initialization
.3.3.1.1, Table 5.A14 (2)	.3.3.1.4, Figure 6.A13	Set default phase
.3.3.1.1, Table 5.A14 (3)	.3.3.1.4, Figure 6.A13	Start normal operation
.3.3.1.1, Table 5.A14 (4)	.3.3.1.4, Figure 6.A13	Advance phase—normal
.3.3.1.1, Table 5.A14 (5)	.3.3.6.4, Figure 6.A25	Advance phase—local
.3.3.1.1, Table 5.A14 (6)	.3.3.6.4, Figure 6.A25	Advance phase—remote
.3.3.1.1, Table 5.A14 (7)	.3.3.1.4, Figure 6.A13	Calculate cycle parameters—actuated
.3.3.1.1, Table 5.A14 (8)	.3.3.1.4, Figure 6.A13	Calculate cycle parameters—adaptive
.3.3.1.1, Table 5.A14 (9)	.3.3.13.3, Figure 6.A37	Critical error—display defaults
.3.3.1.1, Table 5.A14 (10)	.3.3.13.3, Figure 6.A37	Critical error—alarm
.3.3.1.1, Table 5.A14 (11)	.3.3.13.3, Figure 6.A37	Critical error—reset
.3.3.1.1, Table 5.A14 (12)	.3.3.13.3, Figure 6.A36, Figure 6.A37	Write error Log
.3.3.1.1, Table 5.A14 (13)	.3.3.1.4, Figure 6.A13	Set phase
.3.3.1.1, Table 5.A14 (14)	.3.3.1.4, Figure 6.A13	Get phase
.3.3.1.1, Table 5.A14 (15)	.3.3.1.4, Figure 6.A13	Check phase
.3.3.1.1, Table 5.A14 (16)	.3.3.4.4, Figure 6.A19	Pedestrian request latching
.3.3.1.1, Table 5.A14 (17)	.3.3.4.4, Figure 6.A19	Pedestrian request reset
.3.3.1.1, Table 5.A14 (18)	.3.3.4.4, Figure 6.A19	Pedestrian request processing
.3.3.1.1, Table 5.A14 (19)	.3.3.5.4, Figure 6.A22	Vehicle entrance
.3.3.1.1, Table 5.A14 (20)	.3.3.5.4, Figure 6.A22	Vehicle exit
.3.3.1.1, Table 5.A14 (21)	.3.3.5.4, Figure 6.A22	Vehicle request processing
.3.3.1.1, Table 5.A14 (22)	.3.3.5.4, Figure 6.A22	Vehicle reset request state
.3.3.1.1, Table 5.A14 (23)	.3.3.5.4, Figure 6.A22	Vehicle count update
.3.3.1.1, Table 5.A14 (24)	.3.3.1.4, Figure 6.A13	Vehicle count fetch
.3.3.1.1, Table 5.A14 (25)	.3.3.5.4, Figure 6.A22	Vehicle count reset
.3.3.1.1, Table 5.A14 (26)	.3.3.7.3	Get cycle parameters
.3.3.1.1, Table 5.A14 (27)	.3.3.7.3	Update cycle parameters
.3.3.1.1, Table 5.A14 (28)	.3.3.8.4, Figure 6.A28	Process message
.3.3.1.1, Table 5.A14 (29)	.3.3.8.4, Figure 6.A28	Process command
.3.3.1.1, Table 5.A14 (30)	.3.3.8.4, Figure 6.A28	Process message
.3.3.1.1, Table 5.A14 (31)	.3.3.11.3, Table 6.A29	Fetch database
.3.3.1.1, Table 5.A14 (32)	.3.3.11.3, Table 6.A29	Add record
.3.3.1.1, Table 5.A14 (33)	.3.3.11.3, Table 6.A29	Clear database
.3.3.1.1, Table 5.A14 (34)	.3.3.11.3, Table 6.A29	Add record
.3.3.1.1, Table 5.A14 (35)	.3.3.11.3, Table 6.A29	Clear database

Note: All the SRS section references have an additional prefix “5.7” and all the SDD references have a prefix “6.8”. For ease of reading, these are not shown.

REFERENCES

K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," *Proceedings of the Annual Reliability and Maintainability Symposium*, Seattle, WA, 2002, pp. 235–241.

S. W. Ambler, *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3rd Edition. New York: Cambridge University Press, 2004.

K. Beck, *Extreme Programming Explained: Embrace Change*. New York: Addison-Wesley, 1999.

L. Bernstein and C. M. Yuhas, *Trustworthy Systems through Quantitative Software Engineering*. Hoboken, NJ: Wiley-Interscience, 2005.

B. W. Boehm, "A spiral model of software development and enhancement," *ACM SIGSOFT Software Engineering Notes*, 11(4), pp. 14–24, 1986.

B. W. Boehm, "Software risk management: Principles and practices," *IEEE Software*, 8(1), pp. 32–41, 1991.

L. C. Briand, Y. Labiche, and A. Sauve, "Guiding the application of design patterns based on UML models," *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, Philadelphia, PA, 2006, pp. 234–243.

F. Brooks, *The Mythical Man-Month*, 2nd Edition. New York: Addison-Wesley, 1995.

G. Caprihan, "Managing software performance in the globally distributed software development paradigm," *Proceedings of the IEEE International Conference on Global Software Engineering*, Florianopolis, Brazil, 2006, pp. 83–91.

P. A. Dargan, *Open Systems and Standards for Software Product Development*. Norwood, MA: Artech House, 2005.

B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley, 2003.

T. P. Fries, "A framework for transforming structured analysis and design artifacts to UML," *Proceedings of the 24th Annual ACM International Conference on Design of Communication*, Myrtle Beach, SC, 2006, pp. 105–112.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. New York: Addison-Wesley, 1994.

R. Gelbard, D. Te'eni, and M. Sadeh, "Object-oriented analysis—Is it just theory?" *IEEE Software*, 27(1), pp. 64–71, 2010.

O. Goldreich, *Computational Complexity: A Conceptual Perspective*. New York: Cambridge University Press, 2008.

E. Hadar and I. Hadar, "Effective preparation for design review: Using UML arrow checklist leveraged on the gurus' knowledge," *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Montreal, Canada, 2007, pp. 955–962.

S. Henninger and V. Corrêa, "Software pattern communities: Current practices and challenges," *Proceedings of the 14th Conference on Pattern Languages of Programs*, Monticello, IL, 2007, Article no. 14.

J. Holt, *UML for Systems Engineering*. London, UK: IEE, 2001.

C. Horstmann, *Object-Oriented Design and Patterns*, 2nd Edition. Hoboken, NJ: John Wiley & Sons, 2006.

Institute of Electrical and Electronics Engineers, *IEEE Std 1016–2009, IEEE Standard for Information Technology—Systems Design—Software Design Descriptions*. New York: IEEE Computer Society, 2009.

S. Jantunen, "Exploring software engineering practices in small and medium-sized organizations," *Proceedings of the ICSE Workshop on Cooperative Aspects of Software Engineering*, Cape Town, South Africa, 2010, pp. 96–101.

C. Jones, *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. New York: McGraw-Hill, 2010.

P. A. Laplante, *Software Engineering for Image Processing*. Boca Raton, FL: CRC Press, 2003.

P. A. Laplante, *Requirements Engineering for Software and Systems*. Boca Raton, FL: CRC Press, 2009.

C. Larman, "Tutorial: Mastering design patterns," *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, 2002a, p. 704.

C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd Edition. Englewood Cliffs, NJ: Prentice-Hall, 2002b.

C. Larman, *Agile and Iterative Development: A Manager's Guide*. Boston: Pearson Education, 2004.

B. Liskov, "Data abstraction and hierarchy," *SIGPLAN Notices*, 23(5), pp. 17–34, 1988.

N. Maclean, *A River Runs through It and Other Stories*, 25th Anniversary Edition. Chicago, IL: The University of Chicago Press, 2001, p. 104.

R. C. Martin, "The dependency inversion principle," *C++ Report*, 8(6) pp. 61–66, 1996.

T. Martinez, "Computer-based intelligence: Where is it going?" *Opening Talk in Panel Discussion*, *IEEE Mountain Workshop on Adaptive and Learning Systems*, Logan, UT, July 26, 2006.

B. Meyer, *Object-Oriented Software Construction*, 2nd Edition. Englewood Cliffs, NJ: Prentice-Hall, 2000.

R. Miles and K. Hamilton, *Learning UML 2.0*. Sebastopol, CA: O'Reilly Media, 2006.

H. D. Mills, "Certifying the correctness of software," *Proceedings of the 25th Hawaii International Conference on System Science*, Kauai, HI, 1992, vol. 2, pp. 373–381.

J. Nielsen, *Usability Engineering*. New York: Academic Press, 1993.

OMG Unified Modeling Language™ (OMG UML), "Superstructure, Version 2.2," 2009, p. 686. Available at http://www.omg.org/spec/UML/2.2/, last accessed August 17, 2011.

D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, 15(12), pp. 1053–1058, 1972.

D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Transactions on Software Engineering*, 5(2), pp. 128–138, 1979.

H. Pham, *Software Reliability*. New York: Springer, 2000.

R. S. Pressman, *Software Engineering: A Practitioners Approach*, 7th International Edition. New York: McGraw-Hill, 2009.

N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, 35(3), pp. 8–13, 2010.

D. C. Schmidt, M. Stal, H. Robert, and F. Bushmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons, 2000.

Y. Shinjo and C. Pu, "Achieving efficiency and portability in systems software: A case study on POSIX-compliant multithread programs," *IEEE Transactions on Software Engineering*, 31(9), pp. 785–800, 2005.

M. F. Siok and J. Tian, "Empirical study of embedded software quality and productivity," *Proceedings of the 10th High Assurance Systems Engineering Symposium*, Plano, TX, 2008, pp. 313–320.

R. N. Taylor, N. Medvidović, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ: John Wiley & Sons, 2010.

X. Teng and H. Pham, "A new methodology for predicting software reliability in the random field environment," *IEEE Transactions on Reliability*, 55(3), pp. 458–468, 2006.

P. Ward and S. Mellor, *Structured Development for Real-Time Systems*, vols. 1–3. New York: Yourdon Press, 1985.

J. C. Wileden and A. Kaplan, "Software interoperability: Principles and practice," *Proceedings of the International Conference on Software Engineering*, Los Angeles, CA, 1999, pp. 675–676.

D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, 1(1), pp. 67–82, 1997.

E. Yourdon, *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

7

PERFORMANCE ANALYSIS TECHNIQUES

Performance analysis activities can occur in all phases of the software development life cycle (Liu, 2009). While it is natural to analyze performance measures in the testing phase when the individual software components have been integrated together (and possibly with the embedded hardware platform), indicative predictive performance analysis is often needed already in the design and programming phases—even in the requirements engineering phase.

In the testing phase, it is practical to *measure* the performance either in a real operating environment or, at least, in some simulated environment. Extensive measurements provide the most fruitful basis for performance analysis. However, there are often needs to analyze the performance of critical algorithms, achievable response times, and task schedulability before the complete real-time system is available for direct measuring. In such cases, specific performance measures are usually *predicted* or *estimated* using the existing collective knowledge related to similar software products, performing system-level simulations with selected algorithms, doing some instruction-level analysis for program modules, applying theoretical principles and simple laws for parts of the real-time system, and so forth. Nonetheless, a precise and reliable performance analysis of embedded systems is practically impossible without direct measurements from the completed system. And even then, the measurements should be analyzed carefully using statistical methodologies. For

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

instance, a response time might have a non-negligible variance instead of being strictly deterministic; this characteristic is connected to the real-time punctuality defined in Chapter 1.

Of all the places where theory and practice seldom coincide, none is probably more obvious than performance analysis. For all the scientific research on real-time performance analysis, those that have built real-world systems know that reality has the annoying habit of getting in the way of theoretical results. Approximate formulas that ignore resource contention, presume overly simplified hardware, or make the assumption of zero context switch time are only of limited practical use. This criticism, nevertheless, does not mean that theoretical analysis is useless or that there are no useful theoretical results. It only means that there are far less realistic, cookbook-type approaches than would be desired by practitioners. The same observation also applies to other approximate methodologies used for predicting or estimating the performance of real-time systems.

Some system performance optimization may be needed as a consequence of any performance analysis. Performance optimization aims for improving a specific measurable quality in such a way that it would eventually fulfill the requirements specification. This is an important point: performance optimization should be performed solely if there is an explicit demand for it; any optimization effort for just the sake of optimization (often referred to as "gold plating") will cause unnecessary expense and possible schedule delays.

Real-time performance analysis based on simplifying estimation approaches is introduced in Section 7.1. These straightforward techniques provide a handy toolset for limited performance analyses, and the toolset is fully usable even before it is possible to perform direct measurements. Section 7.2 gives a pragmatic discussion on the use of classical queuing theory for analyzing real-time systems. Its applicability to buffer-size calculation and response-time modeling is illustrated with a few examples. Furthermore, input/output (I/O) performance issues are considered in Section 7.3 with an emphasis on buffer-size calculation. This section highlights the common performance bottleneck presented by device I/O access. In Section 7.4, a focused analysis of memory utilization in real-time systems is presented. The chapter is summarized in Section 7.5, with some thoughts on performance optimization, too. Finally, a set of exercises is provided for both self-study and class usage in Section 7.6.

Some parts of this chapter have been adapted from Laplante (2003).

7.1 REAL-TIME PERFORMANCE ANALYSIS

7.1.1 Theoretical Preliminaries

In computational complexity theory (Arora and Barak, 2009), the complexity class P is the class of problems that can be solved by an algorithm that runs in polynomial time on a deterministic computing machine. On the other hand, the complexity class NP (non-polynomial) is the class of all problems that

cannot be solved in polynomial time by any deterministic machine, although a candidate solution can be verified to be correct by a polynomial-time algorithm. A particular decision-making or recognition problem is said to be *NP-complete* if it belongs to the class NP and all other problems in NP are polynomial transformable to it. Moreover, a problem is *NP-hard* if all problems in NP are polynomial transformable to that problem, but it has not been shown that the specific problem belongs to the class NP.

For example, the particular *Boolean satisfiability problem*, which arose during requirements consistency checking in Chapter 5, is NP-complete. The general Boolean satisfiability problem (termed "*N*-Sat") is NP-complete. However, the Boolean satisfiability problem for systems involving two Boolean variables (termed "2-Sat") or three Boolean variables (termed "3-Sat") is in P, and there are tools available for solving such problems. Still, it is easy to imagine that the most interesting problems are *N*-Sat type problems. NP-complete problems in real-time systems tend to be those relating to resource allocation, which is exactly the situation that occurs in multitask scheduling. This unfortunate fact does not bode well for the solution of real-time scheduling problems, as will be discussed shortly.

The remarkable challenges in finding optimal solutions for real-time scheduling problems can be seen in nearly four decades of real-time systems research. Unfortunately, most important problems in real-time scheduling require either excessive practical constraints to be managed or are NP-complete or even NP-hard. Below is a representative sampling from the literature as summarized in Stankovic et al. (1995):

1. When there are mutual exclusion constraints, it is impossible to find a totally online optimal run-time scheduler.
2. The problem of deciding whether it is possible to schedule a set of periodic tasks that use only semaphores to enforce mutual exclusion is NP-hard.
3. The multiprocessor scheduling problem with two processors, no resources, arbitrary partial-order relations, and every task having a 1-unit computation time is polynomial. A partial-order relation indicates that any task can call itself; if task A calls task B, then the reverse is not possible; but if task A calls task B and task B calls task C, then task A can call task C.
4. The multiprocessor scheduling problem with two processors, no resources, independent tasks, and arbitrary task computation times is NP-complete.
5. The multiprocessor scheduling problem with two processors, no resources, independent tasks, arbitrary partial order, and task computation times of either 1 or 2 units of time is NP-complete.
6. The multiprocessor scheduling problem with two processors, one resource, a forest partial order (partial order on each processor), and the computation time of every task equal to 1 unit is NP-complete.

7. The multiprocessor scheduling problem with three or more processors, one resource, all independent tasks, and each computation time of every task equal to 1 unit is NP-complete.
8. Earliest deadline scheduling is not optimal in the multiprocessing case.
9. For two or more processors, no deadline-scheduling algorithm can be optimal without complete *a priori* knowledge of deadlines, computation times, and task start times.

Hence, it turns out that most multiprocessor scheduling problems are in NP. However, for deterministic scheduling, this is not a serious problem because a polynomial scheduling algorithm can be used to develop an optimal schedule if the specific problem is not NP-complete (Stankovic et al., 1995). In such cases, heuristic search techniques can be applied. These offline techniques typically just need to find competitive schedules, not any optimal ones. And this is what practicing engineers do when workable theories do not exist—engineering judgment must prevail.

7.1.2 Arguments Related to Parallelization

Amdahl's Law is a classical argument regarding the effectiveness of parallelization that can be achieved by a parallel computer system (Amdahl, 1967). Today, this fundamental law is somewhat timely even in real-time systems, because of the growing usage of multi-core processors (or "chip multiprocessors") with an increasing number of parallel on-chip cores (Hill and Marty, 2008). These multi-core platforms are used in significant quantities, for instance, in cell phone exchanges, which are mostly firm real-time systems. Nevertheless, the current usage of multi-core processors in such applications still resembles the use of multiple independent uniprocessors, instead of utilizing true parallelization between the available cores.

Definition: Amdahl's Law

Amdahl stated that for a constant problem size, the incremental speedup approaches zero as the number of processor elements grows (Amdahl, 1967). This observation highlights a severe constraint for parallelism in terms of speedup as merely a software property, not a hardware one.

Formally, let N be the number of equal processors available for parallel processing. Let S be the fraction of program code that is of serial nature, that is, it cannot be parallelized at all ($0 \leq S \leq 1$). A usual reason why a portion of code cannot be parallelized is a firm sequence of operations, each depending on the result of the previous operation. Thus, $(1 - S)$ is the fraction of code that can be parallelized. The achievable speedup is then determined as the ratio of the code before allocation to the parallel processors to the ratio of that afterwards:

$$\text{Speedup}_{\text{Amdahl}} = \frac{S+(1-S)}{S+\dfrac{(1-S)}{N}} = \frac{1}{\dfrac{1}{N}+\left(1-\dfrac{1}{N}\right)S} \tag{7.1}$$

Clearly, for $S = 0$ linear speedup can be obtained as a function of the number of processors. But for $S > 0$, perfect speedup is no more possible due to the disturbing sequential component. In such cases, the speedup is saturating to a limit value:

$$\lim_{N\to\infty} \text{Speedup}_{\text{Amdahl}} = 1/S \tag{7.2}$$

Amdahl's law is cited as an argument against parallel systems and, in particular, against massively parallel processors. For example, it can be stated that "there will always be a part of the computation which is inherently sequential, (and) no matter how much you speed up the remaining 90%, the computation as a whole will never speed up by more than a factor of 10. The processors working on the 90% that can be done in parallel will end up waiting for the single processor to finish the sequential 10% of the task" (Hillis, 1998). But from the practical point of view, Amdahl's pessimistic argument is actually flawed. The key assumption of Amdahl's law is that the problem size remains *constant*; and then at some point there is a diminishing increment of return for speeding up the computation. Problem sizes, however, tend to scale with the size of a parallel system. Therefore, parallel computer systems, which are bigger in number of processors, are used to solve larger (more demanding) problems than uniprocessor systems. And this is true both in scientific number crunching as well as in advanced real-time systems with multi-core processors.

Amdahl's law stymied the field of parallel and massively parallel computers for many years, creating an insoluble problem that limited the efficiency and application of parallelism to various problems. The skeptics of parallelism took Amdahl's law as the insurmountable bottleneck to any kind of practical parallelism, which ultimately impacted on real-time systems as well. Fortunately, later research provided new insights into Amdahl's law and its relation to large-scale parallelism.

Two decades after the introduction of Amdahl's law, Gustafson demonstrated with a 1024-processor system at Sandia National Laboratories that the key presumption in Amdahl's Law is clearly inappropriate for massive parallelism (Gustafson, 1988). He found that the problem size scales generally with the number of processors or with a more powerful processor, instead of remaining constant as presumed by Amdahl. However, what is remaining more or less constant is the used (or acceptable) computing time.

Gustafson's empirical results demonstrated that the parallel or vector part of a program scales, indeed, with the problem size. Nonetheless, inherent times for vector start-up, program loading, serial bottlenecks, and I/O that make up

the serial component of the run do not usually grow with the problem size (Gustafson, 1988).

Definition: Gustafson's Law

If the firmly serial code fragment, S, and the parallelized fragment, $(1 - S)$, are processed by a parallel computer system with N equal processors, then the achievable speedup is given as:

$$\text{Speedup}_{\text{Gustafson}} = N - (N - 1)S \tag{7.3}$$

Comparing the bar charts of Equations 7.1 and 7.3 in the example case of $S = 0.5$ (see Fig. 7.1), it can be concluded that Gustafson provides a more optimistic picture of speedup due to parallelism than does Amdahl. In Gustafson's practical view, it is easier to achieve parallel efficiency than is implied by Amdahl's law (Gustafson, 1988). Moreover, the speedup of Equation 7.3 does not saturate as N approaches infinity.

A different take on the flaw of Amdahl's Law can be observed as "a more efficient way to use a parallel computer is to have each processor perform similar work, but on a different section of the data . . . where large computations are concerned this method works surprisingly well" (Hillis, 1998). Doing the same task but on a different range of data circumvents an underlying presumption in Amdahl's law, that is "the assumption that a fixed portion of the computation . . . must be sequential. This estimate sounds plausible, but it turns out not to be true of most computations" (Hillis, 1998).

Lastly, the current "multi-core era" could be viewed as a partial consequence of Gustafson's law. Nonetheless, it remains truly challenging to parallelize real-time software effectively and divide the computing load dynamically to multiple cores.

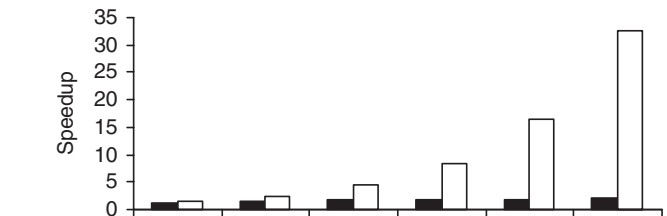


Figure 7.1. Gustafson's unbound speedup compared with Amdahl's saturating speedup ($\rightarrow 2$) when 50% of code is suitable for parallelization.

7.1.3 Execution Time Estimation from Program Code

It is common to analyze real-time systems *a priori* to see if they will meet their critical deadlines. Unfortunately, in a practical sense, this is rarely possible exactly due to the NP-completeness of most scheduling problems and severe constraints imposed by common synchronization mechanisms. Nevertheless, it is possible to get a handle on the system's behavior through approximate analysis. The first step in performing any kind of schedulability analysis is to predict, estimate, or measure the execution time of essential code units.

The need to know the execution time of certain program modules and the overall system time loading before implementation is important from both engineering and project-management perspectives. Not only are CPU utilization requirements expressed as specific design goals, but also knowing them beforehand is important in selecting, for instance, the embedded processor platform. During the programming and testing phases, estimation of CPU utilization is needed to recognize those problematic code units that are particularly slow or whose response times are inadequate. Several methods can be used to determine module execution times and the CPU utilization factor.

Most measures of real-time performance require an execution time estimate, e_i, for each parallel task. The most accurate method for obtaining the execution time of completed code is to use a logic analyzer that is described in Chapter 8. One advantage of this direct approach is that all hardware latencies, as well as other system delays and uncertainties, are taken into account. The drawback in using the logic-analyzer approach is that the entire system or subsystem must be completely programmed and the target hardware available. Hence, the logic analyzer is usually employed solely in the final stages of programming, during testing, and especially during system integration.

When a logic analyzer is not available, the code execution time can be estimated by examining the compiler output and counting machine language instructions either manually or using automated tools. This technique also requires that the code be written, a reasonable sketch of the final code exists, or highly similar systems are available for indicative analysis. The approach simply involves tracing the worst-case execution path through the code, identifying the machine language instructions along the way, and accumulating their execution times.

Another possible method for code execution timing uses the system clock (generated by a timer), which is read before and after executing the particular program code. The time difference can then be used to determine the actual time of execution. This straightforward technique, however, is only viable when the sequence of code to be timed is sufficiently time-consuming relative to the consecutive timer calls.

When it is too early for the logic analyzer, or if one is not available, instruction counting is a practical method for determining time loading. In this approximate approach, the actual CPU-specific instruction times are needed. They can be obtained from the manufacturer's datasheets by timing the

specific instructions using a simulator, or simply by educated guessing. In addition, read/write access times and the number of possible wait states for each memory operation are needed as well.

Example: Instruction Counting Approach

Consider the inertial measurement system discussed earlier in this text. A certain program module converts raw sensor pulses into the actual acceleration components that are later compensated for temperature and other effects. The module is just to decide if the aircraft is still on the ground, in which case only a small acceleration reading for each of the XYZ components is allowed (represented by the symbolic constant `PRE_TAKE`). Now, consider a time-loading analysis for the corresponding C code.

```
#define SCALE 0.01    /* scaling factor */
#define PRE_TAKE 0.1 /* maximum allowable */
void accelerometer (unsigned x, unsigned y, unsigned
z, float *ax, float *ay, float *az, unsigned on_ground,
unsigned *signal)
{
 /* convert pulses to xyz accelerations */
 *ax = (float) x*SCALE;
 *ay = (float) y*SCALE;
 *az = (float) z*SCALE;
 if(on_ground)
    if(*ax > PRE_TAKE ||
       *ay > PRE_TAKE ||
       *az > PRE_TAKE)
       /* no more on the ground: set a bit */
       *signal = *signal | 0x0001;
}
```

These C-language instructions with the compiled assembly language instructions are shown in the following listing for convenient execution-path tracing. Generic assembly language for a two-address machine is assumed. The assembler and compiler directives have been omitted (along with some data-allocation pseudo operations) for clarity and since they do not impact the time loading.

The assembly instructions beginning with "F" are floating-point instructions that require 5 μs. And the FLOAT instruction converts an integer to floating-point format. All other instructions are of integer type and require 0.6 μs.

```
/* convert pulses to xyz accelerations */
*ax = (float) x*SCALE;
     LOAD       R1,&x
```

```
        FLOAT       R1
        FMULT       R1,&SCALE
        FSTORE      R1,&ax
*ay = (float) y*SCALE;
        LOAD        R1,&y
        FLOAT       R1
        FMULT       R1,&SCALE
        FSTORE      R1,&ay
*az = (float) z*SCALE;
        LOAD        R1,&z
        FLOAT       R1
        FMULT       R1,&SCALE
        FSTORE      R1,&az
if(on_ground)
        LOAD        R1,&on_ground
        CMP         R1,0
        JE          @2
if(*ax > PRE_TAKE ||
 *ay > PRE_TAKE ||
 *az > PRE_TAKE)
        FLOAD       R1,&ax
        FCMP        R1,&PRE_TAKE
        JLE         @2
        FLOAD       R1,&ay
        FCMP        R1,&PRE_TAKE
        JLE         @2
        FLOAD       R1,&az
        FCMP        R1,&PRE_TAKE
        JLE         @2
@1:
/* no more on the ground: set a bit */
*signal = *signal | 0x0001;
        LOAD        R1,&signal
        OR          R1,1
        STORE       R1,&signal
@2:
```

Tracing the worst-case execution path and counting the instructions shows that there are 12 integer (7.2 μs) and 15 floating-point (75 μs) instructions for a total execution time of 82.2 μs. Since this sequence of code runs in a 5-ms cycle, the corresponding time-loading is only 82.2/5000 ≈ 1.6%.

In the previous example, we assumed a nonpipelined CPU architecture for simplicity. However, in the next example, we calculate the best- and worst-case execution times (BCET and WCET) for another sequence of assembly code,

first without assuming an instruction pipeline, and then for a three-stage instruction pipeline.

Example: Instruction Counting in Nonpipelined and Pipelined CPU Platforms

Consider the following assembly-language code with 12 numbered instructions:

```
1.          LOAD   R1,&a ; load contents of "a" to R1
2.          LOAD   R2,&a ; load contents of "a" to R2
3.          TEST   R1,R2 ; compare R1 and R2
4.          JNE    @L1   ; go to @L1 if R1 and R2 are not equal
5.          ADD    R1,R2 ; R1 = R1 + R2
6.          TEST   R1,R2 ; compare R1 and R2
7.          JGE    @L2   ; go to @L2 if R1 ≥ R2
8.          JMP    @L3   ; go to @L3 unconditionally
9.    @L1   ADD    R1,R2 ; R1 = R1 + R2
10.         JMP    @L3   ; go to @L3 unconditionally
11.   @L2   ADD    R1,R2 ; R1 = R1 + R2
12.   @L3   SUB    R2,R3 ; R2 = R2 − R3
```

Now, calculate the following estimates:

1. The best- and worst-case execution times (nonpipelined).
2. The best- and worst-case execution times (pipelined).

First, identify all the possible execution paths (A*i* denotes "assembly instruction number *i*"):

Path 1: A1–A4, A9–A10, A12

Path 2: A1–A7, A11–A12

Path 3: A1–A8, A12

Hence, Path 1 includes 7 instructions @ 0.6 μs each → 4.2 μs. Paths 2 and 3 include both 9 instructions @ 0.6 μs each → 5.4 μs. These are the BCET and WCET for this code sequence, respectively.

For the second case, assume that a three-stage pipeline consisting of fetch (F), decode (D), and execute (E) stages is in use and that each stage takes 0.6 μs/3 = 0.2 μs. Here, it is necessary to simulate the contents of the instruction pipeline for each of the three execution paths, flushing the pipeline when required.

For Path 1, the pipeline execution trace is given in Figure 7.2. At the bottom of the trace, time is shown in multiples of 0.2 μs; this yields a total

execution time of 2.6 μs. For Path 2, the pipeline trace looks correspondingly as depicted in Figure 7.3. This represents a total execution time of 2.6 μs. Furthermore, for Path 3, the execution trace is shown in Figure 7.4. Also this path yields a total execution time of 2.6 μs. Thus, the BCET and WCET happen to be equal. This is just a coincidence, and, in general, there is naturally some difference between them.

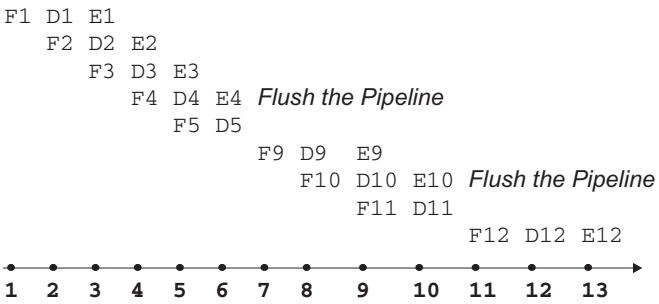


Figure 7.2. Pipeline simulation trace for Path 1.

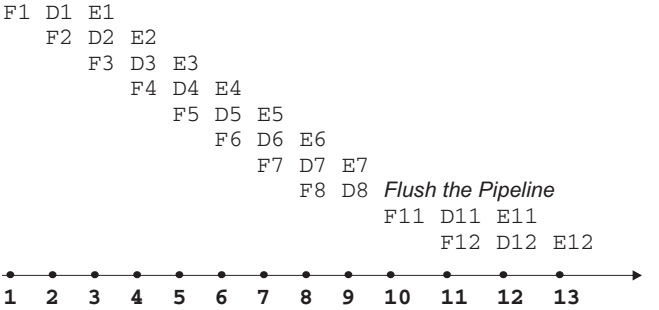


Figure 7.3. Pipeline simulation trace for Path 2.

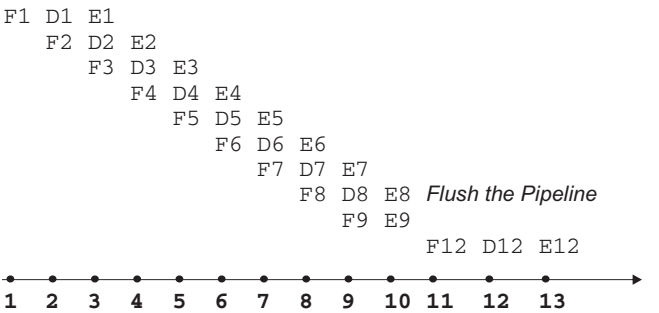


Figure 7.4. Pipeline simulation trace for Path 3.

The laborious process of instruction counting could be automated if a parser is written for the target assembly language that can resolve branching. Besides, commercial performance analysis software is available for execution profiling.

In addition, the determination of instruction execution times is also dependent on memory access times and wait states, which can vary depending on the source region of the instruction code or data in memory. Some organizations that frequently design real-time systems on a variety of CPU platforms use special simulation software to estimate instruction execution times and CPU throughput. With these simulators, users can typically input the CPU type, memory speeds for different address ranges, as well as the instruction mix, and calculate total instruction times and throughput.

Moreover, sections of code can be timed conveniently by reading the system clock before and after the execution of the code. The time difference is then used to determine the actual execution time. Of course, if the code sequence under examination takes only a few microseconds or so, it is recommended to execute the code several thousand times in a loop. This will help to reduce inaccuracies introduced by the granularity of the system clock. When such looping is applied, it is necessary to calculate the additional time spent in the empty loop structure and subtract it from the total.

Example: Timing Accuracy with a 60-kHz System Clock

Suppose 2000 repetitions of the program code under interest take 450 ms with the clock granularity of 16.67 μs. Hence, the execution time measurement has a high accuracy as follows:

$$\text{Accuracy} = \frac{16.67 \cdot 10^{-6}}{450 \cdot 10^{-3}} \cdot 100\% \approx \pm 0.0037\%.$$

The following C code can be used to time a single high-level language instruction or a series of instructions. The number of iterations needed can be varied depending on how short the code to be timed is; the shorter the code, the more iterations should naturally be used to get an adequate accuracy. Here, `current_clock_time()` is a system function that returns the current time, and `function_to_be_timed()` is where the actual code to be timed should be placed.

```
#include system.h
unsigned long timer(void)
{
 unsigned long time0, time1, time2, time3, i, j,
 loop_time, total_time
 iteration = 1000000L;
```

```
 time0 = current_clock_time(); /* read time now */
 for (j=1; j<=iteration; j++); /* run empty loop */
 time1 = current_clock_time(); /* read time now */
 loop_time = time1-time0;      /* empty loop time */
 time2 = current_clock_time(); /* read time now */
 for (i=1; i<=iteration; i++)  /* time function */
    function_to_be_timed();
 time3 = current_clock_time(); /* read time now */
 total_time = (time3-time2-loop_time)/iteration;
 return total_time;            /* function's time */
}
```

7.1.4 Analysis of Polled-Loop and Coroutine Systems

The response time for a *polled-loop system* consists of three essential components: the cumulative hardware delays involved in setting the software flag by some external device; the time for the polled loop to test the flag; and the time needed to process the event associated with the flag (see Fig. 7.5). The first delay component is typically on the order of nanoseconds and can usually be ignored. On the other hand, the time to check the flag and jump to the handler routine can be several microseconds. And the time to process the event related to the flag depends on the task involved (anyhow larger than the two preceding delays). Hence, calculation of a response time for polled loops is straightforward.

The above case assumes that sufficient processing time is available between consecutive events. However, if events begin to overlap each other, that is, if a new event is initiated while a previous one is still being processed, then the response time is becoming worse. In general, if t_F is the time needed to check the flag and t_P is the time to process the event, including resetting the flag (and ignoring the time needed by the external device to set the flag), then the response time for the Nth overlapping event is bounded by

$$Bound = N(t_F + t_P). \tag{7.4}$$

In practice, some limit is placed on N, that is, the number of events that are allowed to overlap. Nonetheless, overlapping events may not be desirable at all in certain applications.

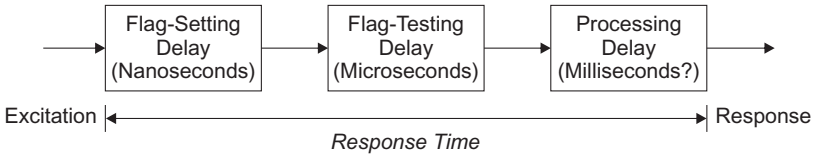


Figure 7.5. Delay components of polled-loop response time.

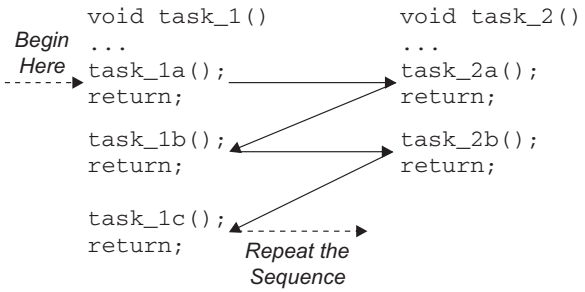


Figure 7.6. Tracing the execution path in a two-task coroutine system. A central dispatcher calls `task_1()` and `task_2()` by turns, and a `switch` statement in each task (not shown) steps the phase-driven code.

Furthermore, the absence of interrupts in a *coroutine system* makes the determination of response time rather easy, and the time is simply obtained by tracing the worst-case execution path through all tasks (see Fig. 7.6). In such case, the execution time of each phase must first be determined using one of the approaches discussed above.

7.1.5 Analysis of Round-Robin Systems

Assume that a round-robin system has n tasks in the ready queue, no new ones arrive after the scheduling starts, and none terminates prematurely. The task release times are arbitrary—in other words, although all tasks are ready for execution at the same time, the order of execution is not specifically predesigned, but is still fixed. Further assume that all the tasks have the maximum end-to-end execution time of c time units. This assumption might first appear overly unrealistic. Nevertheless, suppose that each task, τ_i, has a different maximum execution time, c_i; then letting $c = \max\{c_1, \ldots, c_n\}$ yields a reasonable *upper bound* for the system performance and allows the use of this simple model.

Now, let the constant timeslice for each task be q time units. If any task completes before the end of its time quantum, in practice, that slack time would be assigned to the next ready task in the queue. However, for simplicity of the analysis, we assume here that the possible slack times are not utilized at all. This does not hurt the analysis seriously because only an upper bound is desired, not an exact response time solution.

Ideally, each task would get $1/n$ of the available CPU time in slices of q time units, and would wait no longer than $(n-1)q$ time units until its next time up. Since each task requires at most $\lceil c/q \rceil$ time units to complete (where $\lceil \cdot \rceil$ represents the "ceiling" function, which yields the smallest integer greater than the quantity inside the half brackets), the waiting time will be $(n-1)q\lceil c/q \rceil$. Thus, the worst-case time from readiness to completion for any task (also

known as turnaround time), denoted T, is the waiting time plus the undisturbed time to complete, c, or

$$T = (n-1)q\lceil c/q \rceil + c. \tag{7.5}$$

Example: Turnaround Time Calculation without Context Switching Overhead

First, suppose that there is only one task with a maximum execution time of 500 ms, and that the time quantum is 100 ms. Hence, $n = 1$, $c = 500$, $q = 100$, and

$$T = (1-1)\cdot 100\cdot\lceil 500/100\rceil + 500 = 500 \text{ ms},$$

which is the duration of five time quanta as expected.

Next, suppose there are five equally important tasks with a maximum execution time of 500 ms. The time quantum is still 100 ms. Thus, $n = 5$, $c = 500$, $q = 100$, which yields correspondingly

$$T = (5-1)\cdot 100\cdot\lceil 500/100\rceil + 500 = 2500 \text{ ms}.$$

This result is intuitively agreeable, since it would be expected that five consecutive tasks of 500 ms each would take altogether 2500 ms end-to-end to complete.

Furthermore, assume that there is a *non-negligible* context switching overhead, o, associated with task switching. Each task still waits no longer than $(n-1)q$ until its next time quantum, but there is an inherent overhead of $n \cdot o$ time units each time around for context switching. Again, each task requires at most $\lceil c/q \rceil$ time quanta to complete. An additional assumption is that there is an initial "context switch" to load the first time around. Therefore, the worst-case turnaround time for any task is now at most

$$T = [(n-1)q + n\cdot o]\lceil c/q \rceil + c \tag{7.6}$$

Example: Turnaround-Time Calculation with Context Switching Overhead

First, suppose that there is one task with a maximum execution time of 500 ms. The time quantum is now 40 ms, and the context switch time is 1 ms. Hence, $n = 1$, $c = 500$, $q = 40$, $o = 1$. So,

$$T = [(1-1)\cdot 40 + 1\cdot 1]\lceil 500/40\rceil + 500 = 513 \text{ ms},$$

which is expected, since the context switch to serve the round-robin clock interrupt costs 1 ms each time for the 13 times it occurs.

Next, suppose there are six equally important tasks, each with a maximum execution time of 600 ms, the time quantum is 40 ms, and every context switch costs 2 ms. Thus, $n = 6$, $c = 600$, $q = 40$, $o = 2$. Then,

$$T = [(6-1)\cdot 40 + 6\cdot 2]\lceil 600/40 \rceil + 600 = 3780 \text{ ms},$$

which again is agreeable, because one would expect six tasks of 600 ms in duration to already take 3600 ms without any context switching costs.

In terms of the time quantum, it is desirable that $q < c$ to achieve fair behavior for the round-robin system. On the other hand, if q is very large, the round-robin algorithm is in fact the first-come, first-served algorithm, in that each task will execute to its completion in the order of arrival and within the very large time quantum.

The approximate technique just described is also applicable for cooperative multitasking analysis or any kind of fair cyclic scheduling with non-negligible context switching costs.

7.1.6 Analysis of Fixed-Period Systems

In general, plain utilization-based analysis is not accurate and provides satisfactory bounds just for a highly simplified task model. Therefore, a necessary and sufficient condition for schedulability based on worst-case response-time calculation is presented below.

For the highest-priority task, the worst-case response time will evidently be equal to its own execution time. However, other tasks running on the real-time system are subjected to interference caused by execution of higher-priority tasks. For any task τ_i with an execution time of e_i time units, the response time, R_i, is given as

$$R_i = e_i + I_i, \tag{7.7}$$

where I_i is the maximum possible delay in execution (caused by higher priority tasks) that task τ_i is going to experience in any time interval $[t, t + R_i)$. At the most critical time instant, which is the instant when all higher-priority tasks are released along with task τ_i, I_i will have its maximum contribution.

Consider a task τ_j of higher priority than τ_i. Within the interval $[0, R_i)$, the release time of τ_j will be $\lceil R_i/p_j \rceil$, where p_j is the execution period of τ_j. Each release of task τ_j is going to contribute to the amount of interference τ_i is going to suffer, and is expressed as:

$$\text{Maximum interference} = \lceil R_i/p_j \rceil e_j. \tag{7.8}$$

Each task of higher priority is interfering with task τ_i. Hence,

$$I_i = \sum_{j \in HPR(i)} \lceil R_i/p_j \rceil e \tag{7.9}$$

where $HPR(i)$ is the set of higher-priority tasks with respect to τ_i. Substituting this I_i into Equation 7.7 yields

$$R_i = e_i + \sum_{j \in HPR(i)} \lceil R_i/p_j \rceil e_j. \tag{7.10}$$

Due to the inconvenient ceiling function, it is difficult to solve for R_i directly. Without getting into details, a neat recursive solution is provided, where the equation for calculating R_i is evaluated iteratively by rewriting it as a recurrence relation

$$R_i^{n+1} = e_i + \sum_{j \in HPR(i)} \lceil R_i^n/p_j \rceil e_j. \tag{7.11}$$

where R_i^n is the result of the nth iteration.

When using the recurrence relation to find response times, it is necessary to compute consecutive values of R_i^{n+1} iteratively until the first value of m is found such that $R_i^{m+1} = R_i^m$. This R_i^m is then the desired response time, R_i. It is important to note that if the recursive equation does not have a solution, then the value of R_i^{n+1} will continue to grow, as in the overloaded case when a task set has a CPU utilization factor greater than 100%.

Example: Response Time Calculation in a Rate-Monotonic Case

To illustrate the response time analysis for a fixed-priority scheduling scheme, consider a task set to be scheduled rate monotonically, as shown below:

$$\tau_1: \quad e_1 = 3, p_1 = 9$$
$$\tau_2: \quad e_2 = 4, p_2 = 12$$
$$\tau_3: \quad e_3 = 2, p_3 = 18$$

For every task set, it is always a good practice to calculate first the CPU utilization factor, U of Equation 1.2, to make sure that the real-time system is not overloaded. Here,

$$U = \sum_{i=1}^{3} e_i/p_i = \frac{3}{9} + \frac{4}{12} + \frac{2}{18} \approx 0.72.$$

According to the linguistic classification of Chapter 1, 72% belongs to the "questionable" utilization zone of 70–82%, which is well below overloading.

The highest priority task, τ_1, will naturally have a response time equal to its execution time, so $R_1 = 3$. Moreover, the medium priority task, τ_2, will have its response time iterated using Equation 7.11. First, let $R_2^0 = 4$, and then two recursive values following R_2^0 are derived as:

$$R_2^1 = 4 + \lceil 4/9 \rceil \cdot 3 = 7$$
$$R_2^2 = 4 + \lceil 7/9 \rceil \cdot 3 = 7$$

The equality $R_2^1 = R_2^2$ implies that $R_2 = 7$. Similarly, the response time of the lowest priority task, τ_3, is calculated as follows. First, $R_3^0 = 2$, and two recursive values are again obtained from Equation 7.11:

$$R_3^1 = 2 + \lceil 2/9 \rceil \cdot 3 + \lceil 2/12 \rceil \cdot 4 = 9$$
$$R_3^2 = 2 + \lceil 9/9 \rceil \cdot 3 + \lceil 9/12 \rceil \cdot 4 = 9$$

As $R_3^1 = R_3^2$, the response time $R_3 = 9$.

7.1.7 Analysis of Nonperiodic Systems

In practice, a real-time system having one or more aperiodic or sporadic cycles could be modeled as a rate-monotonic system, but with the nonperiodic tasks approximated as having a period equal to their worst-case expected interarrival time. However, if this rough approximation leads to unacceptably high utilizations, it may be possible to use some heuristic analysis approach instead. Queuing theory (Gross et al., 2008) could also be helpful in this regard. Certain important results from queuing theory are discussed later.

The calculation of response times for interrupt-driven systems is dependent on a variety of factors, including interrupt latency, scheduling/dispatching times, and context switch times. Determination of context save/restore times is carried out similarly as execution time estimation for any application code. The scheduling time is negligible when the CPU uses an interrupt controller supporting multiple interrupts. When a single interrupt is supported in conjunction with an interrupt controller, it can be timed using straightforward instruction counting.

Interrupt latency is a component of response time, and is the (varying) period between when a device requests an interrupt and when the first instruction for the associated interrupt service routine executes. In the design of a real-time system, it is necessary to consider what the worst-case interrupt latency can be. Typically, such an uncommon situation would occur when all possible interrupts in the system are requested simultaneously. The number of tasks also contributes to the worst-case latency, because a real-time operating system needs to disable interrupts while it is processing lists of blocked or waiting tasks. If the real-time software contains a large number of parallel tasks, it is necessary to perform some latency analysis to verify that the operat-

ing system is not disabling interrupts for an unacceptably long time. Nevertheless, in hard real-time systems, it is always good to keep the number of tasks as low as practical.

Another contributor to interrupt latency is the time needed to complete execution of the particular machine language instruction that was interrupted. Hence, it is necessary to find the worst-case execution time of every machine language instruction by measurement, simulation, or manufacturer's data-sheets. The instruction with the longest execution time in the program code will maximize the contribution to interrupt latency if it has just begun executing when the interrupt request arrives.

For instance, suppose in a certain 32-bit microcontroller, all fixed-point instructions take 2 μs, floating-point instructions take 10 μs, and special instructions, such as trigonometric functions, take 50 μs. The real-time software under consideration is known to have only one arc-tangent instruction, but its contribution to interrupt latency can be as high as 50 μs. Nonetheless, the probability of executing the specific arc-tangent instruction just when an interrupt occurs is obviously very low. The latency caused by instruction completion is often overlooked, possibly resulting in unexplained sporadic problems in hard and firm real-time systems.

Deliberate disabling of interrupts by the real-time software can create substantial interrupt latency, and hence it must be included in the overall latency estimation, too. Interrupts are disabled for a number of reasons, including protection of critical regions, buffering routines, and context switching. But it is recommended to avoid interrupt disabling when possible and to minimize the length of periods when they have to be disabled. As a rule of thumb, no application software should have the right to disable interrupts, but interrupt disabling is allowed solely in system software.

Instruction and data caches, instruction pipelines, and direct memory access (DMA), all designed to improve *average* computing performance, destroy determinism and thus make prediction of real-time performance troublesome. In the case of an instruction cache, for example, it is uncertain whether the requested instruction is in the cache. Where it is being fetched from has a significant effect on the execution time of that instruction. Besides, to bring the missing instruction into the cache, a time-consuming replacement algorithm must be applied. Therefore, to carry out a strict worst-case performance analysis, it must be pessimistically assumed that every instruction is not fetched from cache but from the slower main memory instead. This assumption has a very deleterious effect on the predicted performance. Similarly, in the case of pipelines, one must assume that at every possible opportunity, the pipeline needs to be flushed. Finally, when DMA is used in the real-time system, it must be assumed that cycle stealing is occurring at every opportunity, thus inflating instruction fetch times.

Do these special cases all mean that the widely used architectural enhancements render a computer system effectively unanalyzable for real-time performance? Unfortunately, yes, because the traditional worst-case analysis

leads to impractically pessimistic outcomes due to long-tailed execution time distributions. There is, indeed, a nonzero probability that an avalanche of cache misses, pipeline flushes, and cycle stealing would occur when executing a particular code sequence. By making experiential assumptions about the impact of these statistically appearing effects, however, an indicative estimation of performance is still possible.

To cope more effectively with the "destroyed determinism" dilemma, it could be beneficial to create probabilistic performance models for caches, pipelines, and DMA for execution time analysis (Liang and Mitra, 2008). Bernat et al. introduced the notion of *probabilistic hard real-time systems* (Bernat et al., 2002). Such systems should definitely meet all the required deadlines, but it is sufficient to have a probabilistic guarantee very close to 100% instead of an absolute guarantee. This practical relaxation reduces drastically the worst-case execution times to be considered, for instance, in schedulability analysis. Nonetheless, it remains problematic to use the advanced CPU and memory architectures in hard real-time systems.

7.2 APPLICATIONS OF QUEUING THEORY

The classic queuing problem in applied statistics involves one or more *producer* processes called servers and one or more *consumer* processes called customers (Gross et al., 2008). Queuing theory has been applied to the analysis of real-time systems this way since the mid-1960s (Martin, 1967). However, it seems to be mostly omitted from the recent real-time literature.

A standard notation for a queuing system is a three tuple, such as M/M/1 (Gross et al., 2008). The first component describes the probability distribution for the time between arrivals of customers, the second is the probability distribution of time needed to service each customer, and the third is the number of available servers. The letter "M" is customarily used to represent exponentially distributed interarrival or service times.

In a real-time system, the first component of the tuple could represent the probability distribution of the interarrival time for a certain interrupt request. The second component would then be the probability distribution of the time needed to service that interrupt. And the third component would be unity for a uniprocessor system and an integer >1 for multiprocessing systems. The well-known properties of this queuing model can be used, for instance, to predict mean service times for tasks in a real-time system.

7.2.1 Single-Server Queue Model

The simplest queuing model is the M/M/1 queue, which represents a single-server system (see Fig. 7.7) with a Poisson arrival distribution (exponential interarrival times for the customers or interrupt requests with mean $1/\lambda$), exponential service or processing time with mean $1/\mu$, and $1/\lambda > 1/\mu$. Moreover,

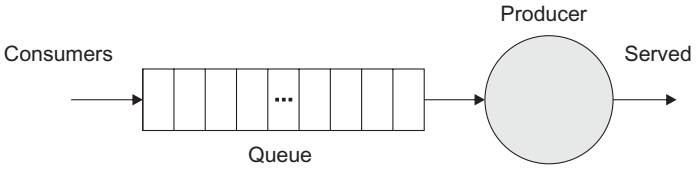


Figure 7.7. A simple single-server queuing model for analyzing real-time systems.

the queue length and the number of possible customers are assumed infinite. As suggested before, this model can be used effectively to model certain aspects of real-time systems; it is particularly useful because the theory is well established, and hence several important results are immediately available (Kleinrock, 1975). For example, let N be the number of customers in the queue. Letting $\rho = \lambda/\mu$, then the expected number of customers in the queue in such a single-server system is

$$\bar{N} = \frac{\rho}{1-\rho}, \tag{7.12}$$

with the corresponding variance

$$\sigma_N^2 = \frac{\rho}{(1-\rho)^2}. \tag{7.13}$$

The mean time a customer spends in the entire system (a typical performance measure) can be expressed as

$$T = \frac{1/\mu}{1-\rho}. \tag{7.14}$$

In addition, a random variable Y for the time spent in the system has the exponential probability distribution

$$s(y) = \mu(1-\rho)e^{-\mu(1-\rho)y}. \tag{7.15}$$

with $y \geq 0$.

Finally, it can be shown that the probability that at least k customers are in the queue simultaneously is

$$\Pr[\geq k \text{ in system}] = \rho^k. \tag{7.16}$$

In the M/M/1 model, the probability of exceeding a certain number of customers in the system decreases geometrically. If interrupt requests are considered customers in a real-time system, then two such requests in the system at the

same time (a time-overloaded condition) have a considerably greater probability of occurrence than three or more simultaneous requests. Thus, building robust systems that can tolerate a single time overload condition will contribute significantly to system reliability, while worrying about multiple time overload conditions is usually pointless. The following subsections describe how the M/M/1 queue can be used conveniently in the analysis of real-time systems.

7.2.2 Arrival and Processing Rates

Consider an M/M/l queuing system in which the customer represents an interrupt request of a certain type and the server represents the particular processing required for that request. In this uniprocessor model, a waiter in the queue represents a time-overloaded condition. Because of the nature of the arrival and processing times, this condition could occur in practice. Suppose, however, that the arrival or processing times can vary. Varying the arrival rate, which is represented by the parameter λ, could be accomplished by modifying the hardware or altering the actual process causing the interrupt. Changing the processing rate, represented by the parameter μ, could be achieved by code optimization or changing the CPU. In any case, fixing one of these two parameters, and selecting the second parameter in such a way as to reduce the probability that more than one interrupt will be in the system simultaneously, will ensure that time overloading cannot occur within a specific confidence interval. This is illustrated in the following two examples.

Example: Mean Processing Time Calculation

Suppose 1/λ, the mean inter-arrival time between interrupt requests, is known to be 10 ms. It is desired to find the mean processing time, 1/μ, necessary to guarantee that the maximum probability of time overloading is 1%.

By using Equation 7.16, we obtain:

$$\Pr[\geq 2 \text{ in system}] = \left(\frac{\lambda}{\mu}\right)^2 \leq 0.01$$

which can be solved for 1/μ as follows:

$$\frac{1}{\mu} \leq \frac{\sqrt{0.01}}{\lambda} \leq 1 \text{ ms.}$$

Thus, the mean processing time, 1/μ, should be no more than 1 ms to guarantee with 99% confidence that time overloading cannot occur.

Example: Mean Inter-Arrival Time Calculation

Next, presume the service time, $1/\mu$, is known to be 5 ms. Here, it is desired to find the mean interarrival time for interrupts, $1/\lambda$, to guarantee that the probability of time-overloading is not more than 1%.

Again,

$$\Pr[\geq 2 \text{ in system}] = \left(\frac{\lambda}{\mu}\right)^2 \leq 0.01,$$

which is now solved for $1/\lambda$:

$$\frac{1}{\lambda} \geq \frac{1}{\mu\sqrt{0.01}} \geq 50 \text{ ms}.$$

Hence, the mean interarrival time between two interrupt requests should be at least 50 ms to guarantee only a 1% risk of time overloading.

Obviously, the context switching time and blocking due to possible semaphore waits are not incorporated in these approximate analyses. Nevertheless, this straightforward approach can be useful in exploring the feasibility of a real-time system with aperiodic or sporadic interrupts, in particular.

7.2.3 Buffer Size Calculation

The M/M/1 queue model can also be used for buffer-size calculations by portraying the "customers" as data being placed in a buffer. The "service time" is the time needed to pick up the buffered data by some server process. In such case, the basic properties of M/M/1 queues are used to calculate the expected buffer size needed to hold the data using Equation 7.12, and the mean time a datum spends in the system (or datum's age) using Equation 7.14. This is shown in the following example.

Example: Expected Number of Data Items and Their Mean Age

Assume a process produces data with an interarrival rate given by the exponential distribution $\lambda = 4e^{-4t}$, and data is consumed by a process at another rate given by the exponential distribution $\mu = 5e^{-4t}$.

To calculate the expected number of data items in the buffer, we use Equation 7.12:

$$\bar{N} = \frac{\lambda/\mu}{1-\lambda/\mu} = \frac{4/5}{1-4/5} = 4.$$

Standard deviation gives a useful indication on the statistical confidence of the mean value, and it can be calculated by taking a square root of the corresponding variance, σ_N^2, which is first determined using Equation 7.13. Thus,

$$\sigma_N = \sqrt{\frac{\lambda/\mu}{(1-\lambda/\mu)^2}} = \frac{\sqrt{4/5}}{1-4/5} \approx 4.5.$$

This is a notably large standard deviation compared to the mean value, and leads to the wide margin of 4 ± 4.5 for the number of data elements in the buffer.

In addition, the mean age of the data items in the buffer can be found by using Equation 7.14:

$$T = \frac{1/\mu}{1-\lambda/\mu} = \frac{1/5}{1-4/5} = 1 \text{ s}.$$

7.2.4 Response Time Modeling

The mean response time for a process handling an interrupt request in the absence of other competing processes can also be computed if an M/M/1 model is assumed. In this case, Equation 7.14 is used to determine the mean time spent in the system by an interrupt request (the mean response time) as illustrated below.

Example: Mean Response Time and Its Probability Distribution

Suppose a process, which serves a sporadic interrupt that occurs with an inter-arrival time given by the exponential distribution function with mean $1/\lambda = 5$ ms. The process handles the interrupt in an amount of time determined by another exponential function with mean $1/\mu = 3$ ms.

Now, the mean response time for this interrupt request is determined by Equation 7.14:

$$T = \frac{1/\mu}{1-\lambda/\mu} = \frac{3}{1-\frac{1/5}{1/3}} = 7.5 \text{ ms}.$$

A probability distribution for the random variable, Y, determining the mean response time can be found by using Equation 7.15:

$$s(y) = \frac{1}{3}(1-3/5)e^{-(1/3)(1-3/5)y} = \frac{2e^{-2y/15}}{15}.$$

Note that the expected response time will be deleteriously affected if the mean interrupt rate is greater than the mean service rate.

7.2.5 Other Results from Queuing Theory

The simple M/M/1 queue can be used also in a variety of other ways to model real-time systems. The only requirements are that the producer be modeled as a Poisson process and that the consumption time be exponential. Although the theoretical model assumes an infinite-length queue, confidence intervals can be fixed appropriately for modeling practical finite-length queues.

Furthermore, consumer–producer systems that can be modeled to match other queue models can benefit from the well-known results there. For example, an M/G/1 queue with Poisson arrival (exponential interarrival) and *general* service time probability distributions could be used. Other results cover the general arrival as well as service densities (Gross et al., 2008). Relationships involving balking consumers, those that leave the queue, can be used to represent rejected spurious interrupts or time overloads.

An important result in queuing theory, *Little's law*, has also some application in performance prediction of real-time systems. This law, which appeared in 1961, is expressed below (Kleinrock, 1975).

Definition: Little's Law

The expected number of consumers in a queuing system, $\overline{N}_{co}$, is equal to the mean arrival rate of the consumers to that system, $\overline{r}_{ar}$, multiplied by the mean time spent in the system, $\overline{t}_{sp}$:

$$\overline{N}_{co} = \overline{r}_{ar}\overline{t}_{sp}. \tag{7.17}$$

If altogether n producers are available, then we can generalize Little's law as

$$\overline{N}_{co} = \sum_{i=1}^{n} \overline{r}_{i,ar}\overline{t}_{i,sp}, \tag{7.18}$$

where $\overline{r}_{i,ar}$ is the mean arrival rate for consumers to producer i, and $\overline{t}_{i,sp}$ is the corresponding mean service time.

What makes this law significant is that the outcome is independent of any definite probability distributions related to the underlying scenario. Moreover, viewing each task as a producer and interrupt arrivals as consumers, Little's law is, actually, Equation 1.2 for CPU utilization with substitutions $e_i = \overline{t}_{i,sp}$ and $1/p_i = \overline{r}_{i,ar}$.

Example: Expected Number of Consumers versus Time Loading

Presume a real-time system is known to have three periodic interrupts occurring at 10, 20, and 100 ms and a sporadic interrupt that is known to occur in average every 1000 ms. The average processing times for these interrupts are 3, 8, 25, and 30 ms, respectively.

Then, by Little's law, the expected number of consumers in the queue (or time-loading) is

$$\bar{N}_{co} = \frac{3}{10} + \frac{8}{20} + \frac{25}{100} + \frac{30}{1000} = 0.98.$$

This result is equal to the one obtained by using Equation 1.2 for CPU utilization with the substitutions defined above.

Another useful result of queuing theory is the *Erlang loss formula* (ELF), which dates back to 1917 (Kleinrock, 1975). Originally, the term "Erlang" refers to a unit used in telephone systems as a statistical measure of service load on switching equipment. Erlang represents a time-average of the number of concurrent telephone calls handled by the switching equipment. Nevertheless, an analogous scenario exists precisely in real-time systems when considering the service of interrupts by a number of processes.

Definition: Erlang Loss Formula

Assume there are m producers and a variable number of consumers. Each newly arriving consumer is serviced by a producer, unless all producers are busy (a potential blocking condition). In this case, the consumer is simply lost. If it is assumed that the average service time of producers is $1/\mu$, and the average interarrival time of consumer is $1/\lambda$, then the probability that all producers are busy is given by

$$P_{\text{busy}} = \frac{(\mu/\lambda)^m / m!}{\sum_{k=0}^{m} (\mu/\lambda)^k / k!}. \tag{7.19}$$

This P_{busy} can be seen as an explicit measure of the *quality of service*, where $P_{\text{busy}} = 0$ corresponds to the ideal condition.

Example: The Use of ELF in Analyzing Real-Time Systems

Applying the ELF of Equation 7.19 to the previous real-time example (where producer = process and consumer = interrupt) gives $m = 4$, $\lambda = 1/282.5$, and $\mu = 1/16.5$; then

$$P_{busy} = \frac{\left(\frac{282.5}{16.5}\right)^4 \Big/ 24}{1 + \left(\frac{282.5}{16.5}\right) + \left(\frac{282.5}{16.5}\right)^2 \Big/ 2 + \left(\frac{282.5}{16.5}\right)^3 \Big/ 6 + \left(\frac{282.5}{16.5}\right)^4 \Big/ 24} \approx 0.78.$$

Hence, there is a probability of 78% for time overloading due to simultaneous interrupts. Considering the mean time-loading factor of 98% ("dangerous") from the previous example, this result seems reasonable. In Chapter 3, we learned from the rate-monotonic theory that a time-loading factor below 69% is sufficient (but not necessary) to guarantee that no overloads occur with any number of tasks. Besides, a nominal time-loading factor of 60% ("safe") is used commonly as a design parameter for cell phone exchanges, for instance.

7.3 INPUT/OUTPUT PERFORMANCE

One performance area that varies greatly owing to device dependencies is the bottleneck presented by hard-disk and device I/O access. In many firm and soft real-time systems, disk I/O is the single greatest contributor to performance degradation. Therefore, hard disks are typically avoided in hard real-time systems. Or, at least, their usage is limited to certain "soft" periods that are less time critical. Moreover, when analyzing a system's performance through straightforward instruction counting, it is very difficult to account for disk device access times. In most cases, the recommended approach is to assume worst-case access times for all device I/O and include them in performance estimations.

Furthermore, when a real-time system participates in some form of a communications network—a fieldbus network or a local area network—loading of the network can seriously affect the real-time performance and make estimation of that performance difficult. Therefore, it is practical to estimate the performance of the system assuming first that the communications network is in the best possible state (i.e., has no other users). Later on, direct measurements of performance can be taken under varying conditions of loading, and a performance curve can be generated. This empirical analysis should be complemented with appropriate statistical methodologies.

7.3.1 Buffer Size Calculation for Time-Invariant Bursts

A buffer is a set of consecutive memory locations that provide temporary storage for data that are being input or output or are being passed between two individual tasks. The use of linear and ring buffers in real-time systems was discussed in Chapter 3.

Assume that the data are being sent for some finite time called a burst period. If the data are produced at a rate of $P(t)$ and can be consumed at a rate of $C(t)$, where $C(t) < P(t)$, for a burst period of T, what is the size of the

buffer needed to prevent any data from being lost? In the trivial case when both $P(t)$ and $C(t)$ are constant, denoted P and C, respectively, and when the consumption rate C is greater than or equal to the production rate P, then no buffer is needed since the system can always consume data faster than they can be produced. However, if $C < P$, then an overflow will eventually occur. To calculate the buffer size needed to avoid any overflow for a burst of period T, note that the total data produced is $P \cdot T$, and the total data consumed within that period is $C \cdot T$. Thus, there is an excess of $(P - C)T$ data units. This is how much data must be stored in the buffer. Hence, the required buffer size, B, can be calculated as

$$B = (P - C)T. \tag{7.20}$$

Example: Time-Invariant Bursts

Suppose a data acquisition unit is providing data to a real-time computer via DMA at 100 K bytes/s in bursts of 0.1 second duration occurring every 5 seconds. The computer is capable of processing the data at 10 K bytes/s. What is the minimum buffer size required? Using Equation 7.20 yields:

$$B = (102400 - 10240)\,\text{bytes/s} \cdot 0.1\ \text{s} = 9216\ \text{bytes}.$$

Handling data that occur in bursts with Equation 7.20 is possible solely if the buffer can always be emptied before another burst occurs. Emptying the buffer in the previous case will take only 0.9 second, which provides a sufficient time margin before the next expected data burst.

If data bursts occur too frequently, then buffer overflow will necessarily take place. In such case, the real-time system becomes unstable, and either upgrading the processor (hardware/software) or slowing down the production process is necessary to solve the problem.

7.3.2 Buffer Size Calculation for Time-Variant Bursts

It is often not adequate to assume that burst periods are fixed; they may frequently be variable. Suppose that a task produces data at a rate given by the real-valued function $P(t)$. Further suppose that another task consumes or uses the data produced by the first task at a rate determined by the real-valued function $C(t)$. The data are produced during a finite burst period $T = t_2 - t_1$, where t_2 and t_1 $(t_2 > t_1)$ represent the finish and start times of the data burst, respectively. Then the buffer size needed at time t_2 can be expressed as

$$B(t_2) = \int_{t_1}^{t_2} [P(t) - C(t)]\,dt. \tag{7.21}$$

Example: Time-Variant Bursts

Assume, a task produces data at a rate (in bytes/s) that is determined by the function $P(t)$ having a discontinuous derivative:

$$P(t)=\begin{cases}10000t & 0\le t\le 1\\ 10000(2-t) & 1<t\le 2,\\ 0 & t>2\end{cases}$$

with t representing the (non-negative) burst time. In addition, the data are consumed by a task at a rate determined by another function:

$$C(t)=\begin{cases}10000(t/4) & 0\le t\le 2\\ 10000(1-t/4) & 2<t\le 4.\\ 0 & t>4\end{cases}$$

Now, if the burst period is known to be 1.6 seconds (from $t_1 = 0$ to $t_2 = 1.6$), what is the necessary buffer size? Applying Equation 7.21 yields,

$$\begin{aligned}B(1.6)&=\int_0^{1.6}[P(t)-C(t)]dt\\ &=10000\int_0^1(t-t/4)dt+10000\int_1^{1.6}[(2-t)-t/4]dt\\ &=10000\left(3t^2/8\Big|_0^1+2t\Big|_1^{1.6}-5t^2/8\Big|_1^{1.6}\right)=6000\text{ bytes}\end{aligned}.$$

Furthermore, if the burst ending time is determined by a real-valued function $u(t)$, where t is the burst starting time, then for a burst starting at t_1 and ending at $t_2 = u(t_1)$, the necessary buffer size at time t_2 is

$$B(t_2)=\int_{t_1}^{u(t_1)}[P(t)-C(t)]dt. \tag{7.22}$$

Example: Random Burst Period

In the previous example, if the data burst ends at a time instant t_2 determined by the Gaussian bell function, then $u(t_1)$ of Equation 7.22 can be expressed as

$$u(t_1)=\frac{1}{\sqrt{2\pi}}e^{-(t_1-2)^2/2}.$$

Now, presume the burst starts at time $t_1 = 0$, then it will end at the time instant $u(0) = 0.053991$. Recalculation of the buffer size yields

$$\begin{aligned} B(0.053991) &= \int_0^{0.053991} [P(t) - C(t)]\,dt \\ &= 10000 \int_0^{0.053991} (t - t/4)\,dt \\ &= 10000\,(3/8)t^2 \Big|_0^{0.053991} \approx 10.9 \text{ bytes} \Rightarrow 11 \text{ bytes} \end{aligned}$$

Determining *when* the maximum buffer size is needed is easily done by graphing the consumer, $C(t)$, and producer, $P(t)$, function curves, and then inspecting them to identify when the difference in the areas under the curves is having its maximum.

7.4 ANALYSIS OF MEMORY REQUIREMENTS

With memory becoming continuously denser and cheaper, memory utilization analysis has become less of a concern also in many real-time applications. Still, efficient use of memory is particularly important in small embedded systems, and, for instance, in aerospace applications where savings in size, power consumption, and cost are highly desirable. In a small embedded system, all the available memory can reside inside a microcontroller, and, hence, there is no way to extend it. On the other hand, in larger systems, it may be possible to enhance the memory by simply changing the memory components, for instance, from 512 K byte Flash chips to 4 M byte ones.

7.4.1 Memory Utilization Analysis

The total memory utilization in a real-time system is the sum of individual memory utilizations for all memory areas. Suppose that a memory map (see Fig. 2.7 for a typical memory map) consists of the following four areas:

1. Program
2. Stack
3. Data
4. Parameters

Then the total memory utilization, $M_r \in [0, 1]$, is calculated as

$$M_T = M_{PG} \cdot P_{PG} + M_{ST} \cdot P_{ST} + M_{DT} \cdot P_{DT} + M_{PM} \cdot P_{PM}, \tag{7.23}$$

where M_{PG}, M_{ST}, M_{DT}, and M_{PM} represent the memory utilization for the program, stack, data, and parameters areas, respectively; and P_{PG}, P_{ST}, P_{DT}, and P_{PM} are fractions of the total memory allocated for those memory areas, respectively. Possible memory-mapped I/O and DMA memory are not included in the following memory-utilization equation, since they are fixed in hardware. Thus, memory utilization is calculated by dividing the number of used locations in a particular memory area by the number of available memory locations in that area:

$$M_A = \frac{U_A}{T_A}, \quad A \in \{T, PG, ST, DT, PM\}, \tag{7.24}$$

where U_A is the number of locations used in memory area A, and T_A is the total number of available memory locations in that area. The limit value for T_A is obviously determined by the hardware platform, but the actual value of U_A is provided by the linker/locator program. Nonetheless, in the case of stack, the value of U_{ST} is dependent on multiple factors, such as the real-time operating system used, the depth of nested procedure calls, the use of local variables, and the number of simultaneous interrupts. Therefore, the estimation of adequate T_{ST} must be done with utmost care, and it is recommended to leave reasonable safety margins between stack and other areas to prevent sporadic stack overflows.

Although the program instructions may be stored in RAM instead of ROM for increased fetching speed and possible modifiability, all global variables are stored in RAM. While the size of the available RAM area is determined at system design time, the loading factor for this area is not known until the application programs have been completed.

Example: Total Memory Utilization

Suppose, a soft real-time system has 64 M bytes of program memory that is loaded at 75%, 16 M bytes of data memory that is loaded at 25%, and 8 M bytes of stack area that is loaded at 50%. All these memory-loading figures represent the corresponding worst-case values. Besides, there is no separate parameters area in this particular memory configuration. Thus, the total memory utilization can be calculated by Equation 7.23

$$M_T = \overbrace{0.75 \cdot \frac{64}{88}}^{\text{Program}} + \overbrace{0.25 \cdot \frac{16}{88}}^{\text{Stack}} + \overbrace{0.5 \cdot \frac{8}{88}}^{\text{Data}} \approx 0.64.$$

Lastly, it should be emphasized that even if the total memory utilization is well below 100%, if any of the memory areas has utilization greater than 100%, then the real-time system cannot operate properly.

TABLE 7.1. Memory Specifications of Low-Cost (*LC*) and High-Performance (*HP*) Motor Drive Products

Drive Id.	Memory Type	Memory Size	Purpose
LC	ROM	64 K bytes	Program
	RAM	2 K bytes	Stack and data
	EEPROM	"Tiny"	Parameters
HP	ROM	"Small"	Booting program
	Flash	512 K bytes	Program (storage) and parameters
	RAM	384 K bytes	Stack, data, and program (execution)

In the previous example, the imaginary soft real-time system had totally 88 M bytes of memory. However, the memory needs of embedded control systems are usually much lower. This is illustrated in Table 7.1, where two electric motor drives are considered—a *low-cost* motor drive (with a 16-bit CPU) and a *high-performance* one (with a 24-bit CPU). These memory requirements are typical for similar embedded systems.

A survey of memory behavior of embedded software is provided by Wolf and Kandemir (2003). That survey is somewhat unique, since it considers the memory system from the software viewpoint. They point out that "in many cases, the memory system is the primary limitation on the performance and power consumption of the embedded software." And what makes the situation complicated for system designers is the interdependence between performance and power consumption. For instance, it is hard to maximize the performance and simultaneously minimize the power consumption in battery-powered embedded systems.

7.4.2 Optimizing Memory Usage

In modern computer systems, memory constraints are not as troublesome as they once were. Nevertheless, in embedded applications or in legacy systems (those that are being reused), often the real-time systems engineer is faced with strict restrictions on the amount of memory available for program storage or for scratch-pad calculations, dynamic allocation, and so forth. Since there exists commonly a fundamental trade-off between memory usage and CPU utilization, when it is desired to optimize for memory usage, it is necessary to trade computing performance to save memory. For example, to calculate a trigonometric function accurately using a lengthy series expansion is a CPU-intensive approach, while a large look-up table would be a memory-intensive solution. And a medium-size look-up table with linear interpolation could provide a practical compromise between those two extremes. These implementation issues are discussed further in Chapter 8.

Moreover, it is important to match the real-time processing algorithms to the underlying computer architecture. For instance, it is necessary to recognize the effects of such features as cache size (memory hierarchy) and pipeline

characteristics (internal parallelism) in hard and firm real-time applications. In the case of cache size, any time-critical algorithm could be tailored to maximize the cache hit ratio, and hence minimize the effective memory access time. In the case of pipeline characteristics, on the other hand, increasing the code's locality of reference can reduce the amount of deleterious pipeline flushing. A pragmatic discussion on the worst-case execution time problem for real-time systems with careful considerations on caches and pipelines is provided by Wilhelm et al. (altogether 15 coauthors) in Wilhelm et al. (2008).

As mentioned above, memory utilization is less of a problem today than it has been in the past, but occasionally a severely constrained system needs to be designed in which the available main memory is small in relation to the program size. Besides, it is expected that this situation will arise more frequently in the future, as ubiquitous and mobile computing applications call for very compact processors with small memories. Most of the approaches developed to reduce memory utilization date from a time when memory was at a premium and might violate the principles of good software engineering.

Memory utilization in one area can be reduced at the expense of another. For example, all variables that are local to procedures increase the loading in the stack area of memory, whereas global variables appear in the data area. By forcing variables to be either local or global, relief can be purchased in one area of memory at the expense of the other, thus making it possible to balance the individual memory utilizations.

In addition, intermediate result calculations that are computed explicitly require a variable either in the stack or the data area, depending on whether it is local or global. The intermediate value could be forced into a work register instead by omitting the intermediate calculation. Nonetheless, such a "forcing" is dependent on the used programming language and the code optimization abilities of the compiler.

Memory fragmentation does not impact memory utilization directly, but it can produce effects resembling memory overloading. In this case, although sufficient memory is available, it is not contiguous. Although memory-compaction schemes were discussed in Chapter 3 and it was noted that they are not desirable in real-time systems, they may be necessary in serious cases of memory overutilization. Nevertheless, this applies to soft real-time systems only.

7.5 SUMMARY

Performance analysis is not a separate stage in the software life cycle, but it is necessary to analyze the performance of software tasks or even the entire real-time system in all stages of the life cycle. Every performance analysis action can be seen as a consequence of the definition: "A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timelines"—particularly the "timelines" part of this definition. Hence,

performance analysis is often focused on predicting, estimating, or measuring specific execution times, interrupt latencies, response times, and so on. For these purposes, a collection of approximate as well as more rigorous methodologies and tools are needed.

The approximate techniques include miscellaneous theoretical models, straightforward instruction-counting approaches with pipeline- and cache-related simplifications, plain task models for estimating worst-case bounds, etc. Moreover, the use of queuing theory offers effective means for analyzing the behavior of buffer structures and real-time systems with aperiodic or sporadic events, for instance. But all these techniques are more or less approximate, and the advanced CPU and memory architectures do anyway destroy the determinism in real-time systems. So, what is the value of such imprecise tools for the practitioner?

Well, performance prediction/estimation is the best the practitioner can do when it is not yet possible to make direct measurements or execution profiling from the completed real-time system (followed by a careful statistical analysis) in a realistic operating environment. Approximate results can provide useful *insight* and *upper bounds* for critical timelines. Such complementary information could be utilized when making specific design decisions, and, particularly, for early recognition of problematic areas in real-time software. In addition, the use of approximate performance analysis tools is helpful when educating software and systems engineers, since it is easy to illustrate the effects of system parameters for a certain performance measure by simple quantitative techniques.

The I/O performance of embedded systems is of great importance, because embedded computers typically have intensive time-critical interaction with their operating environment. In general, communications networks have a growing role in real-time applications. Nevertheless, fieldbus and local area networks can be seen as significant sources of uncertainty due to varying loading conditions, which affect the achievable response times and their punctuality in distributed systems. To obtain meaningful performance estimates in a network environment, realistic statistical models for network traffic should be used; otherwise, there is a threat to either over- or under-estimate some response times drastically. Essentially, the best method would be to make direct performance measurements in a real operating environment with true traffic conditions. Furthermore, it might be practical to implement parallel fieldbus networks when the nature of transferred data is varying from high-priority control commands (short message frames) to low-priority operational statistics (long message frames), for example. In that case, one lightly loaded network could be reserved for high-priority messages and another network for all lower-priority traffic.

Memory performance is the other general performance class, and it can be divided into two subclasses: memory speed and memory size. While the bottleneck of memory speed is commonly relieved with hierarchical memory systems, the potential problem of memory size is usually handled case by case. In

nonreal-time computing, the problem of memory size is less common; but in small embedded systems, there is frequently a necessity to perform memory-size optimization due to severe constraints set by ubiquitous systems, wireless sensor networks, and other mobile units. Therefore, the memory size and power consumption constraints should be addressed in all stages of the software development project—they are not merely hardware issues.

Performance analysis is often followed by performance optimization actions. Whenever something is "optimized," the (multi-objective) cost function—although it may be partly qualitative—should be explicitly defined. Unfortunately, this healthy approach is not always the standard practice; too often, the implicit cost functions applied are focused on a single objective leading to disappointing, suboptimal, and sometimes disastrous results. Practical issues related to performance optimization are discussed in Chapter 8.

7.6 EXERCISES

7.1. Show that there is no such value $\tilde{S}$ for the serial code fragment S that would yield $\text{Speedup}_{\text{Amlahl}} = \text{Speedup}_{\text{Gustafson}}$ with $0 < \tilde{S} < 1$ and $N > 1$.

7.2. A polled-loop system checks a binary status signal every 100 μs. Testing the signal and vectoring to the corresponding interrupt processing routine take 15 μs. If it takes 625 μs to serve the interrupt, what is the minimum response time for this polled interrupt? And what is the maximum response time?

7.3. Consider a foreground/background system that has three task cycles: 10, 40, and 1000 ms. If the worst-case task completion times have been estimated as 4, 12, and 98 ms, respectively, what is the CPU utilization factor of the whole system?

7.4. An intelligent node of a distributed control system has four tasks, τ_1–τ_4 (rate-monotonic priorities), with the corresponding execution periods $p_1 = 10$ ms, $p_2 = 100$ ms, $p_3 = 500$ ms, and $p_4 = 1000$ ms. The execution times are $e_1 = 2$ ms, $e_2 = 15$ ms, $e_3 = 100$ ms, and $e_4 = 10$ ms, respectively. However, task τ_1 is a critical control loop, whose execution period affects directly to the achievable control performance. Hence, in principle, that execution period should be as short as possible. What is the minimum execution period for τ_1 ($= p_{1,\text{min}}$), if the maximum allowed CPU utilization factor is 0.91 ("dangerous")?

7.5. What is the worst-case response time for the background task in a foreground/background system in which the background task requires 100 ms to complete, the single foreground task executes every 50 ms and requires 25 ms to complete, and context switching takes no more than 100 μs?

7.6. Consider a preemptive priority system. The three tasks in the system, time needed to complete, and priority are given below:

Task Id.	Time Needed (ms)	Priority (1 is highest)
τ_1	40	3
τ_2	20	1
τ_3	30	2

If the tasks arrive in the order τ_1, τ_2, τ_3, what is the time needed to complete each task?

7.7. A preemptive foreground/background system has three interrupt-driven task cycles, described below (with context switch time ignored):

Task Id.	Task Cycle	Time Needed (ms)	Priority (1 is highest)
τ_1	10 ms	4	1
τ_2	20 ms	5	3
τ_3	40 ms	10	2
τ_4	Background	5	n/a

(a) Draw an execution time line for this system.

(b) What is the CPU utilization factor?

(c) Considering the context switch time to be 1 ms, redraw the execution time line for this system.

(d) What is the CPU utilization factor with the context switch time included?

7.8. A producer generates data at 1 byte per 200 ns in bursts of 64 K bytes. A consumer, on the other hand, can read the data in 32-bit words, but only at a rate of 1 word every 2 μs. Calculate the minimum buffer size required to avoid overflow, assuming there is enough time between successive data bursts to empty the buffer.

7.9. Show that when the producer and consumer tasks have constant rates, then Equation 7.21 becomes Equation 7.20.

7.10. A producer task is known to be able to process data at a rate that is exponentially distributed with average service time of 3 ms per datum. What is the maximum allowable average data rate if the probability of collision is to be no more than 0.1%? Assume that the data arrive at intervals that are exponentially distributed.

7.11. A computer in a soft real-time system has instructions that require two bus cycles, one to fetch the instruction and another to fetch the data. Each bus cycle takes 250 ns and each instruction takes 500 ns (i.e., the internal processing time is assumed negligible). The computer has a hard disk with 16,512 byte sectors per track. Disk rotation time is 8.092 ms. To what percentage of its normal speed is the computer degraded during

DMA transfer, if each cycle-stealing DMA operation takes one bus cycle? Consider two cases: 16-bit bus transfer and 32-bit bus transfer.

7.12. Which characteristics of reduced instruction set computer (RISC) architectures tend to reduce the total interrupt latency as compared to complex instruction set computer (CISC) architectures (see Chapter 2 for RISC and CISC)?

REFERENCES

G. M. Amdahl, "Validity of the single-processor approach to achieving large-scale computing capabilities," *Proceedings of the AFIPS Spring Joint Computer Conference*, Atlantic City, NJ, 1967, vol. 30, pp. 483–485.

S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. New York: Cambridge University Press, 2009.

G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, Austin, TX, 2002, pp. 279–288.

D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris, *Fundamentals of Queuing Theory*, 4th Edition. Hoboken, NJ: John Wiley & Sons, 2008.

J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, 31(5), pp. 532–533, 1988.

M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, 41(7), pp. 33–38, 2008.

D. Hillis, *The Pattern on the Stone*. New York: Basic Books, 1998.

L. Kleinrock, *Queuing Systems, Volume 1: Theory*. New York: John Wiley & Sons, 1975.

P. A. Laplante, *Software Engineering for Image Processing*. Boca Raton, FL: CRC Press, 2003.

Y. Liang and T. Mitra, "Cache modeling in probabilistic execution time analysis," *Proceedings of the 45th ACM/IEEE Design Automation Conference*, Anaheim, CA, 2008, pp. 319–324.

H. H. Liu, *Software Performance and Scalability: A Quantitative Approach*. Hoboken, NJ: Wiley-Interscience, 2009.

J. Martin, *Design of Real-Time Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1967.

J. A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo, "Implications of classical scheduling results for real-time systems," *IEEE Computer*, 28(6), pp. 16–25, 1995.

R. Wilhelm et al., "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, 7(3), pp. 1–53, 2008.

W. Wolf and M. Kandemir, "Memory system optimization of embedded software," *Proceedings of the IEEE*, 91(1), pp. 165–182, 2003.

8

ADDITIONAL CONSIDERATIONS FOR THE PRACTITIONER

In addition to the fundamental hardware and software technologies, a variety of engineering methodologies is needed when developing real-time systems. While principal real-time system technologies were discussed in Chapters 2–4, Chapters 5–7 covered essential system development methodologies. Now we could ask, is that *all* that the practitioner needs during a software development project? And the answer is obviously "no"—there is, indeed, a heterogeneous collection of techniques and tools that complement the fundamental technologies and methodologies in real-time systems engineering. This pragmatic chapter is devoted to a carefully selected sample of these complementary considerations.

Total system cost is an important factor in software development projects; therefore, it is desirable to have a reliable overall effort estimate available as early as possible. An accurate effort estimate is critical for managing resource allocation and scheduling throughout the development life cycle. Various software metrics and experiential cost models can be used for predicting the progress and costs of a project. Any meaningful prediction should rely on the experience and insight gained in similar software projects (preferably within the same organization). Thus, the knowledge-driven parameters of general cost models could evolve in time leading to continuously improving cost estimates—at least in principle. Sections 8.1 and 8.2 present some commonly used software metrics and so-called constructive cost models, respectively.

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

Furthermore, identifying and managing uncertainty is a standard part of the engineering of complex systems. When dealing with real-time systems, however, special forms of uncertainty create even greater challenges. What are the reasons behind this phenomenon? Well, the obvious answer is that real-time systems must add the assurance of temporal correctness to the already demanding tasks of sensor interfacing and actuator control (Laplante, 2004). But the problem is actually more complicated than that. Hence, it is very important to be aware of the different forms of multidimensional uncertainty in real-time systems. The broad uncertainty issue is addressed in Section 8.3.

Autonomous embedded systems should remain operational for lengthy periods without any intervention by maintenance or service personnel. This is usually achieved by thorough reliability engineering, using high-quality components and subsystems, carrying out extensive testing efforts, and so forth. Testing in different phases of the development project provides effective means for improving the initial reliability of real-time systems. Nevertheless, both autonomous and safety-related systems, such as elevators, planetary rovers, aircraft and nuclear power plants, are commonly also designed to have certain fault tolerance. Fault tolerance is needed to ensure that the real-time system remains functional after some critical fault occurs. In such cases, the initial reliability is not considered adequate without fault tolerance extensions, which can be implemented, for instance, through hardware redundancy, various error-correction capabilities, or functional robustness against missed deadlines. A practical discussion of fault-tolerant embedded systems and a variety of testing schemes is given in Sections 8.4 and 8.5, respectively.

Moreover, there are several performance optimization techniques for time-critical program code. Although floating-point arithmetic and a complete suite of mathematical functions are routinely available when writing simulation and design software, this is not the normal situation when programming embedded systems. It is sometimes the case that only fixed-point (integer) arithmetic is available in the CPU's native instruction set, with special functions computed using series expansions or look-up tables—even multiplication and division instructions might be missing. Thus, a considerable portion of the practitioner's design and programming effort may be needed for "patching" the limitations of the embedded computing platform. Section 8.6 provides an introduction to some performance optimization techniques used in embedded applications.

Finally, a contemplative summary of this chapter is given in Section 8.7. And Section 8.8 contains an instructive collection of exercises on *Additional Considerations for the Practitioner*.

Some parts of this chapter have been adapted from Laplante (2003).

8.1 METRICS IN SOFTWARE ENGINEERING

Empirical software metrics are utilized for real-time systems development in several ways. Certain metrics can be used even during requirements engineer-

ing to assist in resource and cost estimation. Another typical application for software metrics is for benchmarking. For example, if some organization has a collection of successfully completed real-time systems available, then computing metrics for those systems yields a standard set of measurable characteristics with which to compare future systems. Many metrics can also be used for testing in the sense of measuring desirable properties of the real-time software and setting specific limits on the bounds of those criteria.

Of course, metrics can also be used to track project progress. In fact, some companies reward employees based on the amount of software developed per day as measured by some of the metrics to be introduced shortly. Furthermore, software metrics can be used during the testing phase and for debugging purposes to help focus on likely sources of errors. A pragmatic discussion on the broad field of metrics and measuring in software engineering is available in Abran et al. (2003).

8.1.1 Lines of Source Code

The most obvious characteristic of software that can be measured is the number of lines of finished source code. Measured as thousands of lines of code (KLOC), the KLOC metric is often referred to as delivered source instructions (DSI) or noncommented source-code statements (NCSS). That is, a count of executable program instructions, excluding comment clauses, header files, formatting statements, macros, and anything that does not show up as executable code after compilation or cause allocation of memory. Another related metric is source lines of code (SLOC), the major difference being that a single source line of code may span several lines. For instance, an `if-then-else` statement would be a single SLOC, but multiple delivered source instructions.

While the KLOC metric essentially measures the weight of a printout of the source code, thinking in these terms makes it likely that the usefulness of KLOC will be unjustifiably dismissed as supercilious. But is it not likely that 1000 lines of program code are going to have more errors than 100 lines of code? And would it not take longer to develop the latter than the former? Naturally, the answer is dependent on how *complex* the particular code is.

One of the main disadvantages of using lines of source code as a metric is that it can only be measured *after* the code has been written. While it can be estimated beforehand and during software development based on knowledge from similar projects, this is far less accurate than measuring the already available code. Nevertheless, KLOC is a widely used metric, and in most cases is better than measuring nothing. Moreover, many other metrics are fundamentally derived from lines of code. For example, a closely related metric is delta KLOC. The delta KLOC measures how KLOC changes over a fixed period of time. Such a difference measure is useful, perhaps, in the sense that as a project nears the end of code development, delta KLOC would be expected to reduce correspondingly.

8.1.2 Cyclomatic Complexity

A valid criticism of the KLOC metric is that it does not take into account the complexity of the software involved. For instance, 1000 lines of `printf` statements probably have fewer initial defects than 100 lines of a real-time kernel.

To attempt to measure software complexity, cyclomatic complexity was introduced by McCabe to measure program flow-of-control (McCabe, 1976). This concept fits well with procedural programming, but not necessarily with object-oriented programming, though there are adaptations for use with the latter (Coppick and Cheatham, 1992). In any case, this metric has two primary uses:

1. To indicate escalating complexity in a module as it is coded, and, therefore, assist the programmers in determining the appropriate size of their modules.
2. To determine the upper bound on the number of tests that must be designed and performed.

The cyclomatic complexity is based on determining the number of linearly independent paths in a program module, suggesting that the complexity increases with this number, while reliability decreases likewise.

To compute the metric, the following procedure is followed. Consider the flow graph of a program where the *nodes* represent program segments and *edges* represent independent paths. Let e be the number of edges and n be the number of nodes. Form the cyclomatic complexity, C, as follows:

$$C = e - n + 2. \tag{8.1}$$

This is the most generally accepted form for cyclomatic complexity.

To get a sense of the relationship between program flow for some basic code structures and cyclomatic complexity, refer to Figure 8.1. Here, for example, a sequence of instructions has one edge, two nodes, and hence a complexity of $C = 1$. This is intuitively pleasing, as nothing could be less complex than a simple sequence. On the other hand, the particular `case` structure shown in Figure 8.1 has six edges and five nodes with $C = 3$. The higher value for C is consistent with the notion that a `case` statement with three alternative paths is somewhat more complex than a simple sequence of instructions.

Example: Cyclomatic Complexity of a Hard Real-Time Application

Consider a segment of program code extracted from the gyro compensation code for the inertial measurement system. The procedure calls between modules **a**, **b**, **c**, **d**, **e**, and **f** are depicted in Figure 8.2. Here $e = 9$, $n = 6$, and thus the cyclomatic complexity of $C = 5$.

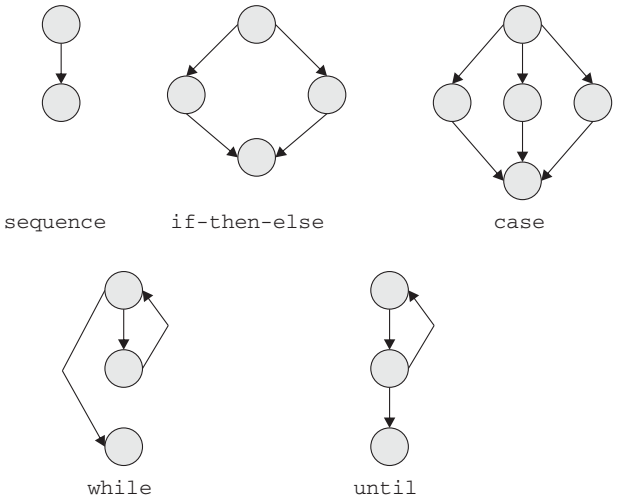


Figure 8.1. Correspondence of language statements and flow graphs; adapted from Pressman (2000).

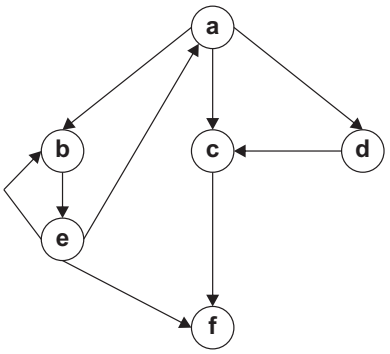


Figure 8.2. Flow graph for gyro compensation code of the inertial measurement system (Laplante, 2003).

Computation of cyclomatic complexity could be done straightforwardly during compilation by analyzing the internal tree structure generated by the parser. However, commercial tools are available to perform this analysis conveniently.

8.1.3 Halstead's Metrics

One of the drawbacks of cyclomatic complexity is that it measures complexity as a function of control flow. But complexity can also exist internally in the way the programming language is used. Halstead's metrics (Halstead, 1977)

measure the information content, or how intensively the programming language is used; and the different metrics are computed as shown below:

1. Find n_1; this is essentially the number of distinct, syntactic `begin-end` pairs (or their equivalents), called "operators."
2. Find n_2, the number of distinct statements (or "operands"). A statement is determined by the syntax of the programming language; for instance, a line terminated by a semicolon is a statement in C language.
3. Count N_1, the total number of occurrences of n_1 in the program.
4. Count N_2, the total number of occurrences of n_2 in the program.

From these basic statistics, the following metrics can now be computed. The program *vocabulary*, n, is defined as

$$n = n_1 + n_2. \tag{8.2}$$

The program *length*, N, is defined as

$$N = N_1 + N_2. \tag{8.3}$$

The program *volume*, V, is defined as

$$V = N \log_2 n. \tag{8.4}$$

The *potential volume* of the program, V^*, is defined as

$$V^* = (2 + n_2) \log_2 (2 + n_2). \tag{8.5}$$

The program *level*, L, is defined as

$$L = V^* / V. \tag{8.6}$$

where L is a measure of the level of abstraction of the program. It is believed that increasing this number will increase software reliability. Nonetheless, there exists no general proof of such a correlation.

Another Halstead metric measures the amount of mental effort required in the development of the code. The programming *effort*, E, is defined as

$$E = V / L. \tag{8.7}$$

Again, decreasing the effort level is believed to increase reliability, as well as ease of implementation. In practice, the program length, N, can be estimated easily, and hence is useful in cost and resource estimation. This length is also a measure of the "complexity" of the program in terms of language usage, and thus can be used to estimate defect rates, too.

Halstead's metrics, though dating back over three decades, are still widely used, and software tools are available to completely automate their determination. Besides, Halstead's metrics can be applied both to program code and to requirements specifications, by adapting the definitions of "operator" and "operand" accordingly. In this way, comparative statistics can be generated from the software requirements specification. Halstead's metrics have also been used for related applications, such as identifying whether two programs are identical except for naming changes; something that is useful in plagiarism detection or software patent infringement.

8.1.4 Function Points

Function points were introduced in the late 1970s as an alternative to metrics based on the simple source line count (Seibt, 1987). The basis of function points is that as more powerful programming languages are developed, the number of source lines necessary to perform a given function decreases. Paradoxically, however, the blind cost/KLOC measure indicates a reduction in productivity, as the fixed costs of programming remain largely unchanged.

One solution is to measure the functionality of software via the number of interfaces between modules and subsystems in programs or entire systems. A significant advantage of the function point metric is that it can be calculated *before* any coding occurs based solely on the design description.

The following five characteristics for each software module, subsystem, or system represent its function points:

1. Number of inputs (I)
2. Number of outputs (O)
3. Number of user inquiries (Q)
4. Number of files used (F)
5. Number of external interfaces (X)

Next, consider empirical weighting factors for each characteristic that reflect their relative difficulty in implementation. For example, one set of weighting factors for a particular kind of system might yield the function point (FP) formula:

$$FP = 4I + 4O + 5Q + 10F + 7X. \tag{8.8}$$

The weights given in Equation 8.8 could be adjusted experientially to take into account factors, such as the particular application domain and software developers' experience. For instance, if W_i are the weighting factors, F_j are the "complexity adjustment factors," and A_i are the item counts, then FP is defined as:

$$FP = \left(\sum_i A_i W_i \right) \cdot \left(0.65 + 0.01 \sum_j F_j \right). \tag{8.9}$$

Intuitively, the higher *FP*, the more difficult the software system is to implement.

The complexity factor adjustments can be further adapted for different application domains, such as embedded and other real-time systems. To determine the complexity factor adjustments, a set of 14 standard questions are answered by the software engineer(s) with numerical responses from a scale from 0 to 5, where:

0 = No influence
1 = Incidental
2 = Moderate
3 = Average
4 = Significant
5 = Essential

For example, in the inertial measurement system suppose the engineering team was queried, and the following interrogatory and resulting answers to the questions were obtained:

Q1. Does the system require reliable backup and recovery?
A1. "Yes, this is a critical system; assign a 4."
Q2. Are data communications required?
A2. "Yes, there is communication between various components of the system over the MIL–STD–1553 serial data bus; therefore, assign a 5."
Q3. Are there distributed processing functions?
A3. "Yes, assign a 5."
Q4. Is performance critical?
A4. "Absolutely, this is a hard real-time system; hence, assign a 5."
Q5. Will the system run in an existing, heavily utilized operational environment?
A5. "In this case, yes; assign a 5."
Q6. Does the system require on-line data entry?
A6. "Yes, via multiple sensors; thus, assign a 4."
Q7. Does the on-line data entry require the input transactions to be built over multiple screens or operations?
A7. "Yes it does; assign a 4."
Q8. Are the master files updated on-line?
A8. "Yes they are; therefore, assign a 5."

Q9. Are the inputs, outputs, files, or inquiries complex?

A9. "Yes, they involve comparatively complex sensor inputs; assign a 4."

Q10. Is the internal processing complex?

A10. "Clearly it is, the compensation and other algorithms are nontrivial; hence, assign a 4."

Q11. Is the code designed to be reusable?

A11. "Yes, there are high upfront development costs and multiple applications have to be supported for this investment to pay off; assign a 4."

Q12. Are the conversion and installation included in the design?

A12. "In this case, yes; thus, assign a 5."

Q13. Is the system designed for multiple installations in different organizations?

A13. "Not organizations, but in different applications, and therefore this must be a highly flexible system; assign a 5."

Q14. Is the application designed to facilitate change and ease of use by the user?

A14. "Yes, absolutely; hence, assign a 5."

These quantified answers are summarized in Table 8.1. Then applying Equation 8.9 yields:

$$FP = \left(\sum_i A_i W_i\right) \cdot (0.65 + 0.01 \cdot [6 \cdot 4 + 8 \cdot 5]) = \left(\sum_i A_i W_i\right) \cdot 1.29.$$

TABLE 8.1. Quantified Answers, A1–A14, to the Complexity Factor Questionnaire for the Inertial Measurement System

Answer	0	1	2	3	4	5
A1					×	
A2						×
A3						×
A4						×
A5						×
A6					×	
A7					×	
A8						×
A9					×	
A10					×	
A11					×	
A12						×
A13						×
A14						×
Total #	0	0	0	0	6	8

Now, suppose that it was determined from the Software Requirements Specification that the item counts were as follows:

$$A_1 = I = 5$$
$$A_2 = U = 7$$
$$A_3 = Q = 8$$
$$A_4 = F = 5$$
$$A_5 = X = 5$$

Using the weighting factors from Equation 8.8 and an additional one for A_5:

$$W_1 = 4$$
$$W_2 = 4$$
$$W_3 = 5$$
$$W_4 = 10$$
$$W_5 = 7$$

Including these into Equation 8.9, yields

$$FP = (5 \cdot 4 + 7 \cdot 4 + 8 \cdot 5 + 5 \cdot 10 + 5 \cdot 7) \cdot 1.29 \approx 223.$$

For the purposes of comparison, and as a project management tool, function points have been mapped to the relative lines of source code in particular programming languages (Jones, 1995). Such mappings are shown in Table 8.2. For instance, it is intuitively acceptable that it would take many more lines (+150%) of assembly language code to express a certain functionality than it would take when using a high-level language like C. In the case of the inertial measurement system, with $FP = 223$, it is expected that about 28.5 thousand lines of code would be needed to implement the functionality. In turn, it should take many less (–50%) to express that same functionality in a more abstract language such as C++. The same observations that apply to programming might also apply to maintenance, as well as to the reliability of software.

TABLE 8.2. Programming Language and Lines of Code per Function Point; Adapted from (Jones, 1998)

Programming Language	Lines of Code/Function Point
Assembly	320
C	128
C++	64

Real-time applications such as the inertial measurement system are highly complex and hence they have many complexity factors rated at "5," whereas in other kinds of systems, such as database applications, these factors would be much lower. This is an explicit statement about the difficulty in developing and maintaining code for embedded real-time systems versus non-embedded ones.

The function point metric was developed for use in business information processing, and not in embedded systems. Nevertheless, a special form of function points is used widely in real-time systems, especially in large-scale real-time databases, multimedia applications, and Internet support (see the following subsection). These systems are data driven and often behave like the large-scale transaction-based systems for which function points were originally developed.

The International Function Point Users Group (http://www.ifpug.org/, last accessed August 23, 2011) maintains a Web database of weighting factors and function point values for a variety of application domains. These can be used for comparison.

8.1.5 Feature Points

Feature points are an extension of function points developed by Software Productivity Research Inc., in 1986. Feature points address the fact that the function point metric was developed for business information systems and hence is not particularly applicable to real-time systems, such as mobile communications or industrial process control. The motivation is that these systems exhibit high levels of algorithmic complexity, but relatively sparse inputs and outputs.

The feature-point metric is computed in a similar manner to the function point, except that a new term for the number of algorithms, A_6 with weighting factor W_6, is added to Equation 8.9. Besides, "A" for "algorithms" is added to Equation 8.8.

For example, suppose that the item counts are the same as in Equation 8.8 with $A = 7$, and the empirical weightings are correspondingly:

$$W_1 = 3$$
$$W_2 = 4$$
$$W_3 = 5$$
$$W_4 = 4$$
$$W_5 = 7$$
$$W_6 = 7$$

Then the feature-point metric, FP^+, is

$$FP^{+} = 3I + 4O + 5Q + 4F + 7X + 7A. \tag{8.10}$$

As another example, consider the inertial measurement system. Using the same item counts as computed before, suppose that the item count for algorithms, $A = 10$. Now using the same complexity adjustment factor, FP^{+} would be computed as follows:

$$FP^{+} = (5 \cdot 3 + 7 \cdot 4 + 8 \cdot 5 + 10 \cdot 4 + 5 \cdot 7 + 10 \cdot 7) \cdot 1.29 \approx 294.$$

If the system were to be written in C language, it could be estimated that approximately 37.6 thousand lines of code would be needed (use Table 8.2 and substitute $FP^{+} \rightarrow FP$), a clearly more pessimistic estimate than that computed earlier using the function point metric.

8.1.6 Metrics for Object-Oriented Software

While any of the previously discussed metrics can be used with object-oriented code, particularly with respect to the code within methods, other metrics are better suited for this setting (Coppick and Cheatham, 1992). For instance, some of the metrics that have been used include:

- A weighted count of methods per class
- The depth of inheritance tree
- The number of children in the inheritance tree
- The coupling between object classes
- The lack of cohesion in methods

As with any metrics, the key to obtaining benefits is consistency.

8.1.7 Criticism against Software Metrics

Many software engineers object to the use of empirical metrics in one or all of the ways that have been described. Several counterarguments to the use of metrics have been stated, for example, that they can be misused or that they are a costly and an unnecessary distraction. For instance, metrics related to the raw number of code lines imply that the more powerful the language, the less productive the programmer appears. Hence, obsessing with code production based on lines of code can be seen as a meaningless endeavor.

Metrics can also be misused through carelessness, which can lead to bad decision making. Finally, metrics can be misused in the sense that they are abused to "prove a point." For example, if a project manager wishes to assert that a particular member of the software team is "incompetent," he could frivolously base his assertion on the lines of code produced per day without accounting for other factors at all.

Another objection is that measuring the correlation effects of a single metric without clearly understanding the causality is dangerous. For instance, while there are numerous studies suggesting that lowering the cyclomatic complexity leads to more reliable software, there just is no objective way to know why. Obviously, the arguments about the complexity of well-written code versus "spaghetti code" apply, but there is just no way to show the causal relationship. Therefore, the opponents of metrics might argue that if in a study of several companies, it was shown that software written by engineers who always wore blue shirts had statistically significant fewer defects in their code, companies should start requiring a dress code of blue shirts! This illustration is a hyperbole, but the point of correlation versus causality is made clear. While it is possible that in many cases these objections may be valid, software metrics can be either useful or harmful, depending on how they are used (or abused).

The objections raised about software metrics, however, suggest that best practices need to be used in conjunction with metrics. These include establishing the purpose, scope, and scale of the metrics. In addition, any serious metrics program needs to be incorporated into the project management plan by setting solid measurement objectives, defining appropriate procedures, and performing measurements throughout the software life cycle. Besides, it is important to create a positive team culture where honest measurement and collection of data is encouraged and rewarded.

8.2 PREDICTIVE COST MODELING

Resource and cost estimation are imperative issues in any software development project. One of the most widely used and appreciated resource estimation tools is Boehm's algorithmic COCOMO, first introduced in 1981 (Boehm, 1981). COCOMO is an acronym for *Constructive Cost Model*, and it is a predictive model. This predictive nature makes it possible to obtain meaningful resource estimates already early in the software development life cycle. There are three forms of the original COCOMO 81: *basic*, *intermediate*, and *detailed*, as well as the more recently released COCOMO II (Boehm et al., 2000).

8.2.1 Basic COCOMO 81

The basic COCOMO 81 is based on the simple KLOC metric (thousands of lines of code). In short, for a given piece of software, the development effort applied (in person months), PM, to complete the software is a nonlinear function of L, the KLOC measure, and two empirical parameters, a and b, which will be explained shortly. The effort equation for the basic COCOMO 81 can thus be expressed as:

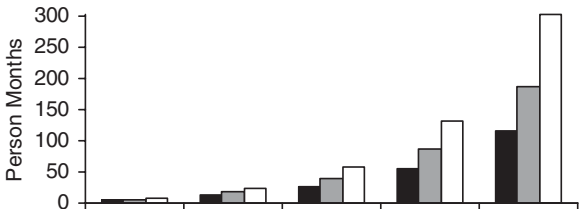
$$PM = a \cdot L^b, \tag{8.11}$$

where the parameters a and b are empirical functions of the type of software system to be developed, and they are determined from extensive data collected from representative projects. For example, if the software system is *organic*, that is, one that is not heavily embedded in the hardware, then the following parameter values are used: $a = 2.4$ and $b = 1.05$. On the other hand, if the system is considered *semidetached*, that is, partially embedded, then these values should be used instead: $a = 3.0$ and $b = 1.12$. Finally, if the system is truly *embedded*, that is, intimately tied to the underlying hardware (like the inertial measurement system), then the following parameters are used: $a = 3.6$ and $b = 1.20$. Note that the exponent b for the embedded alternative is the highest ($b = 1.20 > 1.12 > 1.05$), leading to the largest effort to complete for an equal number of lines of code. The remarkable effect of software type on the person-month estimates for a few KLOC values is depicted in Figure 8.3.

Recall that for the inertial measurement system, using feature points, 37.6 thousand lines of C code were estimated. Nonetheless, if we use this estimate in Equation 8.11, the *technical complexity* becomes actually double counted as the exponent 1.20 is based on essentially the same parameter set as the technical complexity factor 1.29 that was used earlier for calculating the feature points. Therefore, we have to scale down the feature points, 294/1.29 = 228, and use the corresponding estimate of 29.2 thousand lines of C code (see Table 8.2). Finally, an effort estimate is obtained using Equation 8.11:

$$PM = 3.6 \cdot (29.2)^{1.20} \approx 206.4 \text{ person months.}$$

The basic COCOMO 81 also provides a formula for estimating the calendar time (in months) to develop the whole software, DT, when having the corresponding PM available. For this purpose, two other empirical parameters, c and d, are introduced. The parameter $c = 2.5$ is independent of the type of the software, while d has values 0.38, 0.35, and 0.32 for organic, semidetached, and



	2	5	10	20	40
■ Organic	5.0	13.0	26.9	55.8	115.4
▩ Semidetached	6.5	18.2	39.5	86.0	186.8
□ Embedded	8.3	24.8	57.1	131.1	301.1

KLOC

Figure 8.3. Person-month, PM, estimates for different types of software as a function of the KLOC measure, $L \in \{2, 5, 10, 20, 40\}$.

embedded software, respectively. Now, the development time can be determined as:

$$DT = c \cdot PM^d. \tag{8.12}$$

By continuing with the inertial measurement system example, we can next calculate the estimated number of months the project would take using Equation 8.12:

$$DT = 2.5 \cdot (206.4)^{0.32} \approx 13.8 \text{ months}.$$

From the estimated PM and DT values, the number of software engineers required, SE, can now be determined as follows:

$$SE = PM/DT. \tag{8.13}$$

Equation 8.13 gives for the inertial measurement system: $SE = 206.4/13.8 \approx 15$ persons. Hence, 15 software engineers are needed to complete this demanding software project in about 14 calendar months (assuming 152 effective working hours per month). It should be emphasized, however, that the basic COCOMO is solely intended for making rough initial estimations of project costs and resources. Miyazaki and Mori evaluated COCOMO 81 with a set of real-world project data and concluded that the original COCOMO clearly overestimated the efforts required to develop software in their environment (Miyazaki and Mori, 1985).

8.2.2 Intermediate and Detailed COCOMO 81

The intermediate and detailed COCOMO 81 dictates the kinds of adjustments used to improve the modeling accuracy. Consider the intermediate model, for instance. Once the effort estimate for the basic model is computed based on the appropriate parameters and number of lines of code, further adjustments can be made based on additional factors. In this case, for example, if the lines of code to be produced consist of design-modified code, code-modified code, and integration-modified code rather than completely new code, a linear combination of these relative percentages is used to create an adaptation-adjustment factor, as will be discussed below.

Adjustments are then made to PM based on two sets of factors, the adaptation adjustment factor and the effort adjustment factor. The former is a measure of the kind and proportion of program code that is to be used in the system, namely, design modified, code modified, or integration modified. And the adaptation-adjustment factor, A, is given correspondingly:

$$\begin{aligned} A = {} & 100 - 0.4 \cdot (\%\text{ design modified}) - 0.3 \cdot (\%\text{ code modified}) - \\ & 0.3 \cdot (\%\text{ integration modified}). \end{aligned} \tag{8.14}$$

For totally new software components, $A = 100$, since there is no reused code. On the other hand, if all of the code is reused as *design modified*, then $A = 60$. The percentages of design-, code-, and integration-modified code reused do not have to add up to 100 unless all of the code has been reused in some manner. For example, if 10% of the code is reused as design modified, 15% is reused as code modified and 20% as integration modified, then $A = 100 - 0.4 \cdot 10 - 0.3 \cdot 15 - 0.3 \cdot 20 = 85.5$.

Next, an adjusted value for the number of code lines, L', is obtained as:

$$L' = L \cdot A/100. \tag{8.15}$$

You can see how as A varies, it reflects the advantages of reuse in the effective adjusted lines of code count, for instance, if $A = 90$ in Equation 8.15, then $L' = L \cdot 0.9$. This L' is now used in Equation 8.11 in place of the original L.

A tuned version of the effort-adjustment factor can be made to the prior adjusted number of code lines, L', based on a variety of case-dependent attributes, including:

- *Hardware* attributes, such as performance constraints
- *Personnel* attributes, such as applications experience
- *Product* attributes, such as the required reliability
- *Project* attributes, such as the CASE tools used

Each of these attributes is assigned a number (typical values from 0.8 to 1.5) depending on an assessment that rates the attributes on a relative scale. Then, a straightforward linear combination of the attribute numbers is formed based on the particular software type. This provides another adjustment factor, call it E. Hence, the second adjustment leading to the effort-adjusted number of code lines, L'', is made based on the formula:

$$L'' = E \cdot L'. \tag{8.16}$$

This finally yields to the enhanced effort equation:

$$PM'' = a \cdot (L'')^b. \tag{8.17}$$

Furthermore, the detailed model differs from the intermediate model in that tailored effort-adjustment factors are used for each phase of the software life cycle.

COCOMO is widely recognized and respected as a project management tool. It is useful even if the background of the empirical model is not really understood. COCOMO software is commercially available, and easy-to-use resource/cost calculators can be found on the Web for free usage.

One drawback to COCOMO 81, however, is that it does not take into account the leveraging effect of various productivity tools. Moreover, the cost

model bases its estimation almost entirely on the number of lines of code, not on actual program attributes, which is something that feature points do. Feature points and function points, however, can be converted easily to lines of code using standard conversion tables, such as Table 8.2.

8.2.3 COCOMO II

COCOMO II is a major revision of COCOMO 81 that was introduced in 2000 to deal with some of the original version's obvious shortcomings (Boehm et al., 2000). The newer model helps to accommodate more expressive programming languages, as well as advanced software generation tools that tend to produce more code with essentially the same human effort.

In addition, in COCOMO II, some of the more important factors that contribute to a project's expected duration and cost are included as new scale drivers. These scale drivers are used to modify the exponent b in the fundamental effort equation:

- Architectural/risk resolution
- Development flexibility
- Process maturity
- Project novelty
- Team cohesion

The scale drivers of project novelty and development flexibility, for instance, describe many of the same attributes found in the adjustment factors of the original COCOMO 81.

It is beyond the scope of this real-time systems text to discuss COCOMO or its use in more detail. As with any metric or model, it must be used carefully and be based on insight and experience. Nevertheless, using such a well-proven cost model is certainly better than using none at all.

A recent overview of the main contributions on software cost and resource estimation over the past four decades is available in Boehm and Valerdi (2008). That article provides an insightful discussion on the evolution of COCOMO and other significant models.

8.3 UNCERTAINTY IN REAL-TIME SYSTEMS

Over the past three decades, the focus of embedded systems engineering has evolved from simply meeting the performance goals to *design for uncertainty*. In this section, the diverse nature of uncertainty in real-time systems is examined. Our emphasis is on the identification of uncertainty in software through "tell-tale" behaviors and "code smells." Besides, practical techniques for managing, mitigating, or even eliminating the uncertainty are given.

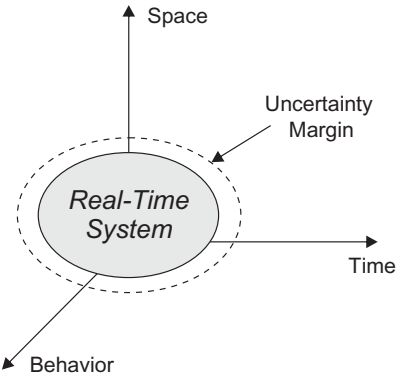


Figure 8.4. A real-time system with uncertainty can be viewed as having dimensions of time, space, and behavior, and some uncertainty margin in all these dimensions.

This section is adapted from Laplante (2004).

8.3.1 The Three Dimensions of Uncertainty

Uncertainty exists in real-time systems along three principal dimensions: *time*, *space*, and *behavior* as illustrated in Figure 8.4. If we try to reduce uncertainty in any one of these dimensions, at least one of the other two will suffer increased uncertainty. This empirical observation is analogous to the well-known Heisenberg Uncertainty Principle in quantum mechanics.

Definition: Heisenberg Uncertainty Principle

The precise position and momentum of a particle cannot be known simultaneously; trying to be more certain about one comes at the expense of increased uncertainty in the other.

By definition, real-time systems have a requirement of timeliness, and it is largely the unpredictability of response times that makes real-time systems so challenging to design and analyze. Quantifying the uncertainty of time is the focus of mainstream real-time systems research and is epitomized by the primary useful result of such research, the rate-monotonic (RM) theorem (discussed in Chapter 3). The RM theorem states that the optimal scheduling algorithm for a set of periodic preemptive priority tasks on a single processor is to assign the task priorities so that the higher the rate of execution, the higher the priority. Nonetheless, this theorem holds only when aperiodic/sporadic tasks, mutual exclusion, and resource contention are excluded. When those realistic elements are included, the RM theorem breaks down and uncertainty in response times begins to grow.

Returning to Figure 8.4, the space dimension deals with the physical resources that the real-time system manages, including its memory. Like Heisenberg uncertainty, trying to improve the certainty in one dimension comes at the cost of uncertainty in another. For example, trying to bring uncertain behavior under control costs space (more memory or hardware) or computation time. On the other hand, occasional "cheating" on timeliness perhaps by prematurely terminating some iterative calculation can lead to uncertain behavior.

8.3.2 Sources of Uncertainty

Let us look more closely at the illustration of Figure 8.4 and examine the potential sources of uncertainty, particularly along the axis of behavior. To do this, we view the real-time system as a state-based transformation of a set of inputs and current state into a new state and corresponding set of outputs. Each of these essential elements can incorporate uncertainty. For instance, uncertainty can be found in any of the inputs to the system. Similarly, the state of the system might be uncertain at any time, resulting in loss of control (but every control system must be stable all of the time). The transition from one state to another can also be nondeterministic, that is, uncertain. Moreover, the outputs to the operating environment are not always predictable in poorly designed or misused real-time systems. Finally, the real-time system must usually interact with an uncertain environment.

Environmental uncertainty can stem from many sources. For example, it can be caused by a chaotic system under control (SUC). Chaotic systems are those in which small changes in inputs lead to radically changed state behavior and outputs. Hence, chaotic systems present significant challenges to the real-time systems engineer. Complicating the situation is the fact that corrupted outputs from the real-time system to the SUC can even cause a stable SUC to appear to be chaotic. The classic inverted pendulum or cart-and-pole control problems create environmental uncertainty through their inherent instability, for instance.

Another form of environmental uncertainty arises from carelessly defined, incomplete, or inconsistent software requirements. If such uncertainties are left unattended, they will lead to insidious uncertainty in the realized system.

In addition, a further kind of uncertainty arises from the physical environment, for example, due to single event upsets in space or various peculiarities of military battlefield conditions. Of course, controlling the physical environment is generally impossible. Therefore, any real-time system must be constructed to be sufficiently tolerant (or robust) to environmental influences.

A kind of environmental uncertainty results also from the testing process. Testing should be used to try to control all sources of uncertainty and verify the coping mechanisms. However, testing itself is inherently uncertain. For instance, is the test coverage adequate, or is the test strategy applied the correct one? There is no way to know the answers for certain. In fact, just

because a real-time system has passed some test suite does not mean that the system would be 100% defect free.

Uncertainty of the data input into the real-time system is often due to malfunctioning devices, disturbances, noise, data acquisition errors, and so on. Some of the bad inputs, however, might be the result of problems with the operating environment and not the system under control. In any case, the real-time systems engineer must never trust the external inputs to the system—they need to be sanity checked, verified, or filtered before use.

The outputs of the real-time system can also be corrupted by device malfunctioning and data conversion errors, presenting uncertainty to the system under control. Besides, the real-time system could then receive corrupted inputs from the system under control, and this corruption is actually based on its own corrupted outputs.

At any time for a given set of inputs and current state, we should be able to predict the next state of the real-time system correctly. In uncertain situations, this is not always possible. For example, undesired jumping of the program counter due to such causes as single event upsets, pointer misuse, or "phantom" interrupts can lead to uncertainty of state. Since we cannot open up the "black box" and look inside, we can never be certain of the integrity of any state.

Behavioral uncertainty as a whole is a wide class of uncertainty that incorporates timing and scheduling problems, the uncertainty of component behavior, and the uncertainty of the programming language being used.

Uncertainty in time arises from the fact that the time it takes for the system to make a transition from an input set and current state to the output set and new state is not necessarily deterministic. Here, we are faced with a dilemma: most task-scheduling problems are NP-complete or NP-hard, and thus not yielding to straightforward solutions.

Moreover, uncertainty of component behavior in off-the-shelf or legacy hardware/software is a reality that must frequently be addressed. There are techniques, however, that can help to reduce this form of uncertainty. For instance, if the source code is available, fault-injection could be used to examine the software component. In this experimental technique, deliberate faults are created in the software at critical points to see how the faults propagate throughout the code (Voas and McGraw, 1998). Other approaches would incorporate rigorous testing of the off-the-shelf or legacy components and not rely on the possibly available second-hand information.

Another form of behavioral uncertainty is caused by the use of programming languages and their compilers. For example, in object-oriented languages, composition is preferred to inheritance. Yet the former yields more uncertain behavior and is difficult to test. In both the object-oriented and procedural conventions of programming, unbounded recursion, dead and unreachable code, unbounded `while` loops, and left-in debug statements, can all lead to uncertain behavior. Clearly, knowing how compiler and run-time support code behave is critical in controlling such uncertainty.

8.3.3 Identifying Uncertainty

The nondeterministic behavior of real-time systems may be painfully visible. Bizarre outputs, hung systems, missed interrupts, and sporadic deadlocks are all symptoms of uncertainty. The problem is that it is not always clear if the uncertainty lies in the environment, input, output, state, or system behavior. One potential technique for identifying the source of the uncertainty is through code smells. A code smell is a term that refers to a somewhat subjective indicator of poor design or coding style (Mäntylä et al., 2004). More specifically, the term relates to observable signs that suggest the need for refactoring—a behavior preserving code transformation enacted to improve some feature of the software, which is evidenced by the code smell.

A traditional code smell that hints at behavioral uncertainty involves timing delays implemented as `while` or other loops. These *software delays* rely on the computational cost of the loop construct plus the execution time of the body code to achieve a specific delay. Here, the problem is that if the underlying computer architecture or characteristics of instruction execution change, then the delay length is inadvertently altered, leading to timing uncertainty. The obvious solution would be to use some reliable timing mechanism provided by the real-time operating system (RTOS).

Another sign of uncertainty is a *dubious constraint*. This particular code smell involves response time constraints that have a questionable or nonattributable origin. In some cases, real-time systems have specific deadlines that are based on nothing more than guessing or on some forgotten and since eliminated requirement. The refactoring is to discover the true reason for such a constraint. If the origin can be determined, then the constraint may be relaxable.

The *speculative generality* code smell relates to hooks and special cases, which are built into the code to handle things that are not currently required (it is uncertain that they are needed). Real-time systems should not contain any "what–if" code, since it can lead to testing anomalies and possibly unreachable code.

Furthermore, the *tell-tale comment* code smell involves comments that are excessive or tend to explicate the code beyond a reasonable level. Explicating comments are often an indicator of a serious problem. Comments that explicitly acknowledge uncertainty, such as "do not remove this piece of code," or "if you remove this statement the code does not work, I do not know why" (and we have seen these in real, "industrial-strength" code) are natural alarms for concern. These kinds of nonprofessional comments indicate that there are probably hidden timing problems. In any case, the refactoring involves rewriting the code so that such vague comments become unnecessary.

In addition to the above code smells, three typical smell indicators for object-oriented programming, that is, *large class*, *long parameter list*, and *duplicate code*, were suggested for automatic code analysis in Mäntylä et al. (2004).

8.3.4 Dealing with Uncertainty

Uncertainty in real-time software, if properly managed, can be reduced over time, but if left unattended will most likely grow. We have already explored the reduction of uncertainty through refactoring of code indicated by bad smells. There are other useful techniques, too.

Environmental uncertainty due to poor requirements could be managed by consistency checking and goals-based requirements analysis. In this context, goals are high-level objectives of business, organization, or system; and a requirement specifies how a goal should be accomplished by the proposed real-time system. Other formal methods could be helpful, as well for the same purpose (Hinchey and Bowen, 1999).

For uncertain inputs, typical solutions include the use of averaging, median filters, Kalman filters, data fusion, as well as roll backing and use of recovery blocks. These general techniques can also be used to control the uncertainty of outputs.

In the case of state-based uncertainty, in addition to the refactorings already mentioned, model checking and black-box recorders can be helpful. Model checking is a formal method that uses finite state machines to verify the state behavior (Chandra et al., 2002). A software black box, on the other hand, is a run-time tool that uses checkpointing to record functional transitions (Elbaum and Munson, 2000). The recorded transitions are used for postmission analysis to determine the likeliest sequence of execution that led to a particular failure.

Another form of execution time uncertainty may arise due to gradual build up of various truncation and round-off errors (the running software ages). These can be managed by stopping and restarting the system regularly. Such a blunt technique is called rejuvenation (Bernstein and Yuhas, 2005); however, it should be used cautiously.

Uncertainty is a pervasive and persistent quality of real-time systems. The total elimination of uncertainty is practically impossible, because of the complex nature of the systems under control as well as the uncertain operating environments (Littlewood, 1994). But rather than admit defeat, a proactive approach to mitigating uncertainty is needed. Such an approach starts with acknowledging uncertainty's existence and then identifying its primary causes so that an effective mitigation strategy can be designed. Each mitigation strategy should preferably be a custom-designed solution. Table 8.3 summarizes the different kinds of uncertainty in real-time systems, their typical signs, possible causes, as well as potential solutions to the underlying problem.

8.4 DESIGN FOR FAULT TOLERANCE

Fault tolerance in real-time systems is the tendency to continue functioning in the presence of hardware or software failures (Koren and Krishna, 2007). Sometimes, it may be necessary to reduce the quality of functioning to a minimum acceptable level due to a sensor failure, for instance. In real-time

TABLE 8.3. Summary of Kinds of Uncertainty in Real-Time Systems with Typical Signs, Possible Causes, and Potential Solutions (Laplante, 2004)

Kinds of Uncertainty	Typical Sign(s)	Possible Cause(s)	Potential Solution(s)
Environmental			
System under control	Bizarre inputs or outputs	Nature of application; faulty hardware	Use fault-tolerant design
Operating environment	Bizarre inputs or outputs	Humidity, temperature, or electromagnetic interferences	Use fault-tolerant design and implementation approaches
Requirements	Sparse requirements; numerous "to be determined"	Inconsistent or incomplete requirements	Goal-based requirements analysis; formal consistency checking
Testing	System that passes tests fails in the field	Poor testing regimen or incomplete coverage	Improve testing process
Input	Strange behavior; explicating comments	Unstable input sources; defective hardware	Averaging, median filter, Kalman filter, data fusion, rollback and recovery blocks
Output	Strange behavior; explicating comments	Defective hardware; corrupted inputs from system under control	Averaging, median filter, Kalman filter, data fusion, rollback and recovery blocks
State	Strange behavior; explicating comments	Program counter jumping, pointer errors, phantom interrupts	Model checking, software black boxes, interrupt service routines for all interrupts
Behavioral			
Timing and schedulability	Dubious constraints; missed deadlines; explicating comments	Speculative generality, delays as loops, or failed off-the-shelf components	Model checking; use RTOS provided timing facilities; fault injection
Language	Explicating comments	Compiler-induced errors	Verify the compiler; improve coding techniques
Off-the-shelf components	Missed deadlines; inexplicable failure	Poorly tested software or hardware; falsely advertised claims	Fault injection; software black boxes

systems, fault tolerance includes also such design choices that transform hard real-time deadlines into softer ones. These are encountered in interrupt-driven systems, which can provide for detecting and reacting to a missed deadline.

Fault tolerance designed to upgrade the initial reliability in embedded systems can be classified as either *spatial* or *temporal*. Spatial fault-tolerance includes methods involving redundant hardware and/or software solutions, whereas temporal fault-tolerance involves miscellaneous techniques that allow for tolerating missed deadlines. Of these two, temporal fault tolerance is more difficult (often impossible) to achieve, since it requires careful algorithm design.

8.4.1 Spatial Fault-Tolerance

The reliability of real-time systems can usually be increased using some form of spatial fault-tolerance based on *redundant hardware*. In one typical scheme, two or more pairs of redundant hardware devices provide inputs to the system. Each device compares its output to its companion. If the results are unequal, the pair declares itself in error, and their outputs are ignored. An alternative is to use a third device to determine which of the other two is correct. In either case, the unavoidable penalty is increased cost, space, and power requirements.

Various voting schemes (Bass et al., 1997) can also be used in software to increase algorithm robustness. Often like inputs are processed from more than one source and reduced to some sort of best estimate of the actual value. For example, an aircraft's position can be determined via information from satellite positioning systems, inertial navigation data, and ground information. A composite of these complementary readings is then made using data fusion techniques (Varshney, 1997).

Furthermore, it is important to build redundancy solely in such parts of the real-time system, which are *known* to be significant sources of catastrophic faults. It is just wasting money and resources to implement fault tolerance in parts that are not likely to have faults, although those faults, if they occurred, would be catastrophic. This issue is discussed in the following vignette.

Vignette: Fault-Tolerance But No Faults

Consider the elevator bank control system of Figure 3.17. From that figure, it is easy to point out three areas, which are particularly susceptible to critical faults:

F1. Interface from the Hall-Call Buttons to Group Dispatcher; faults in this interface isolate some or all hall calls from call allocation.

F2. Communications link between the Group Dispatcher and individual Elevator Controllers; faults in the serial link make it impossible to allocate registered hall calls to one or all elevators.

F3. The Group Dispatcher itself; should the dispatcher computer fail, then the whole call-allocation process will terminate abruptly.

Hence, in the worst scenario, these faults could lead to a catastrophic situation where no passengers are serviced by the elevator bank. Imagine a morning traffic peak when a huge number of employees are entering a 40-story office building within an hour or so, and no elevators are servicing one of the three zones (low-, mid-, or high-rise)!

In lobbies with multiple elevators on two sides, the potential hall-call interface problem (F1) is handled routinely by including redundant sets of up and down buttons on both sides of the lobby. Moreover, these two sets are interfaced to the group dispatcher via independent I/O channels. If one of the interfaces becomes faulty, the other set of call buttons is still operational.

Following similar thinking without much further analysis, fault tolerance through redundancy was provided for the communications-link and group-dispatcher faults (F2 and F3), too. This enhancement is illustrated in Figure 8.5, where a backup communications link as well as a backup group dispatcher are included (Ovaska, 1998). These redundant hardware/software extensions were carefully designed and implemented, and the successful real-time system was in production for several years. So, what is the point of this vignette?

Well, the term "fault-tolerance" consists of two essential parts: the "fault" causing a failure and "tolerance" against its effects. While the elevator system of Figure 8.5 is tolerant against faults F1–F3, the point here is that these faults were later recognized to be practically missing. The vast majority of recorded hardware faults appeared in the I/O section interfacing to the external operating environment of the elevator controllers. On the other hand, computer faults and faults in the internal communications link had an insignificant probability of occurrence in this particular product. Thus, the backup components represented development, material, assembly, and testing expenses, but did not offer any explicit benefits.

Our conclusion is that before any fault-tolerance enhancements are planned, the application-dependent issues of fault probability and severity should be assessed thoroughly and objectively.

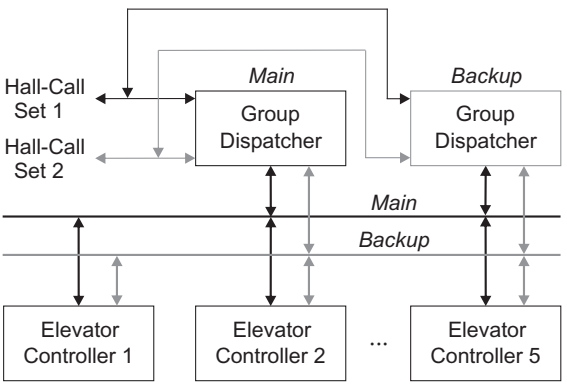


Figure 8.5. Fault-tolerant version of the elevator bank control system.

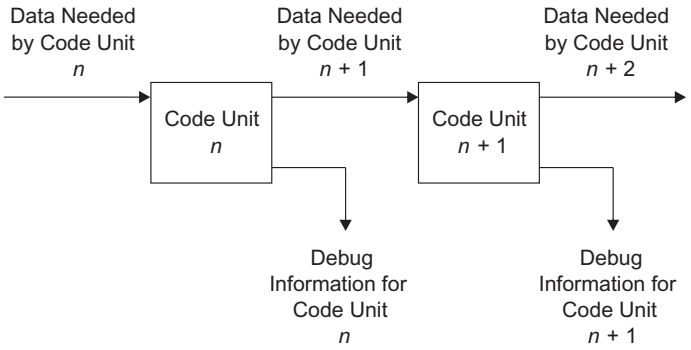


Figure 8.6. Checkpoint implementation (Laplante, 2003).

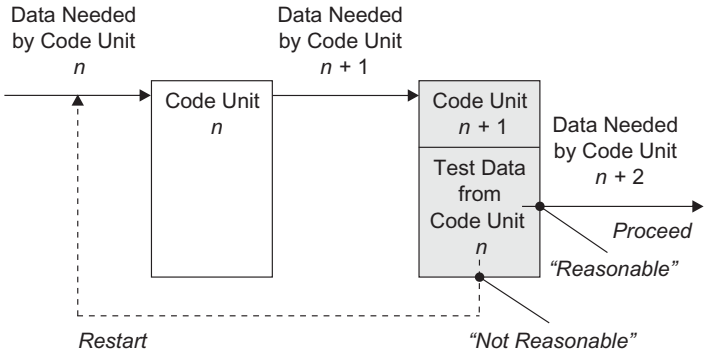


Figure 8.7. Recovery block implementation (Laplante, 2003).

Another way to increase fault tolerance is to use *software checkpoints* (Saglietti, 1990). In this scheme, intermediate results are written to memory at fixed locations in program code for diagnostic purposes (see Fig. 8.6). These special locations, called checkpoints, can be used both during system verification and during system operation. If the checkpoints are used only during testing, then this additional code is known as a test probe. On the other hand, test probes can introduce subtle timing problems for real-time systems.

Fault-tolerance can be further increased by using checkpoints in conjunction with predetermined reset points in software. These reset points mark *recovery blocks* in the real-time software. At the end of each recovery block, the corresponding checkpoints are tested for "reasonableness." If the results are not reasonable, then processing resumes with a prior recovery block (see Fig. 8.7). The point, of course, is that some hardware device (or another process that is independent of the one in question) has provided faulty inputs to the block. By repeating the processing in the block, with presumably valid input data, the "soft" error will not be repeated.

In the process block approach, each recovery block represents a redundant parallel process to the block being tested. Although this strategy increases system reliability, it can have a severe impact on real-time performance because of the overhead added by the checkpoint and repetition of the processing in a block.

8.4.2 Software Black Boxes

The software black box is related to checkpoints and is used in certain mission-critical systems to recover data to analyze the cause of disasters to prevent future ones. The objective of a software black box is to determine the sequence of events that led to the software failure for the purpose of identifying the faulty program code (Elbaum and Munson, 2000). The software black-box recorder is essentially a checkpoint that records and stores behavioral data during program execution while attempting to minimize any impact on that execution.

The execution of program functionalities results in a sequence of module transitions such that the real-time system can be described as modules and their interaction. When software is running, it passes control from one module to the next. Exchanging control from one module to another is considered a transition. Call graphs can be created from these transitions graphically using an $N \times N$ matrix, where N represents the number of modules in a system or subsystem.

When each module is called, each transition is recorded in a matrix, incrementing the associated element in a transition frequency matrix. From this, *a posteriori* probability-of-transition matrix can be derived that records the likeliness that a transition will occur. The transition frequency and transition matrices indicate the number of observed transitions and the probability that some sequence is missing in these data.

Analysis can begin after the system has failed and the software black box has been recovered. The software black box decoder generates possible functional scenarios based on the execution frequencies found in the transition matrix. The generation process attempts to map the modules in the execution sequence to specific functionalities, which allows for the isolation of the likely cause of failure.

8.4.3 *N*-Version Programming

In virtually any complex system, such a state can be entered where the system is rendered ineffective or locks up. This is usually due to some untested flow-of-control in the software for which there is no escape. That is to say that event determinism has been violated.

In order to reduce the likelihood of this sort of catastrophic error, redundant processors are sometimes added to the real-time system. These processors are coded to the same specifications, but by different programming teams.

It is therefore unlikely that more than one of the systems could lock up under the same circumstances. Since each of the redundant systems usually resets a watchdog timer, it quickly becomes obvious when one of them is locked up, because it fails to reset its individual timer. The other processors in the system can then ignore this processor, and the overall system continues to function. This technique is called *N*-version programming (Teng and Pham, 2002), and it has been used successfully in a number of projects developing mission-critical systems, including the space shuttle's general-purpose computer. Nevertheless, Parnas showed in the case history of Ontario Hydro that even independent programming teams might produce correlated results (Hoffman and Weiss, 2001).

The redundant processors can use a voting scheme to decide on outputs, or, more often, there are two processors, a master and a slave. The master processor is online and produces the actual outputs to the system under control, whereas the slave processor shadows the master offline. If the slave detects that the master has become hung up, then the slave takes over the mastership and goes on-line.

8.4.4 Built-in-Test Software

Built-in-test software (BITS) can enhance fault-tolerance by providing online diagnostics data of the underlying hardware for further processing by the software. BITS is especially important in embedded systems. For instance, if an I/O channel is functioning incorrectly as determined by its onboard test circuitry, the software may be able to shut off that channel and redirect the I/O to another channel. Although BITS is an important part of embedded systems, it may add considerably to the worst-case time-loading analysis. This must be considered when selecting BITS and when interpreting the CPU utilization contributions that result from the additional software.

In a critical embedded system, the health of the CPU should be checked regularly. A set of carefully constructed tests can be performed to verify the efficacy of its instruction set in all addressing modes. Such a comprehensive test suite is time consuming and thus should be relegated to background processing. Interrupts should be disabled during each subtest to protect the data being used.

Nonetheless, there is a "catch-22" involved in using the CPU to test itself. If, for example, the CPU detects an error in its instruction set, can it be trusted? On the other hand, if the CPU does not detect an error that is actually present, then this, too, is a paradox. Such a contradiction should not be a reason for omitting the CPU instruction set test, because in any case, the detected error is due to some failure either in the test itself or in the underlying hardware.

All types of memory, including nonvolatile memories, can be corrupted via electrostatic discharge, power surging, vibration, or other physical means. This damage can manifest either as a permutation of data stored in a memory cell or as permanent damage to the cell. Corruption of both RAM and ROM by

randomly encountered charge particles is a particular problem in space. These single-event upsets do not usually happen on earth, because either the magnetosphere deflects the offending particle, or the mean free path of the particle is not sufficient to reach the surface of earth.

Damage to the contents of memory is called a *soft error*, whereas damage to the cell itself is a *hard error*. The embedded systems engineer is particularly interested in techniques that could detect an upset to a memory cell and then correct it.

The contents of ROM are often checked by comparing its original checksum to a newly calculated one. The original checksum, which is usually a simple (overflowing) binary addition of all program-code memory locations, is computed at link time and stored in a specific location in ROM. The new checksum can be recomputed in a slow cycle or background processing, and compared against the original checksum. Any deviation should be reported as a memory error.

Simple checksums are not a reliable form of error checking with large memories, since errors to an even number of locations can result in error cancellation. For instance, an error to bit 4 of two different memory locations may cancel out in the overall checksum, resulting in no error being detected. In addition, although an error may be reported, the location of the error in memory remains unknown.

A more reliable method for checking ROM-type memory uses a cyclic redundancy code (CRC). The CRC treats the contents of memory as a long stream of bits, and each of these bits as the binary coefficient of a message polynomial (Moon, 2005). A second binary polynomial of much lower order, (such as 16) called the generator polynomial, is divided (modulo-2) into the message, producing a quotient and a remainder. Before dividing, the message polynomial is appended with a zero bit for every term (with a zero or unity coefficient) in the generator polynomial. The remainder from the modulo-2 division of the zero-padded message is the CRC check value, and the quotient is discarded. The widely applied CRC-16 (CCITT) generator polynomial is

$$X^{16} + X^{12} + X^5 + 1, \tag{8.18}$$

whereas the alternative CRC-16 (ANSI) generator polynomial is

$$X^{16} + X^{15} + X^2 + 1. \tag{8.19}$$

These CRCs can detect all 1-bit errors and virtually all multiple-bit errors. The source of the error, however, cannot be pinpointed. For example, suppose a ROM consists of 64 K of 16-bit wide memory. The CRC-16 of Equation 8.19 is to be employed to check the validity of the memory contents. Here, the full memory contents represent a polynomial of order $65{,}536 \cdot 16 = 1{,}048{,}576$ at most. Whether the polynomial starts from high or low memory does not matter as long as consistency is maintained. After appending the polynomial with 16

zeros due to the use of CRC-16, the polynomial is at most of order 1,048,592. This so-called message polynomial is then divided by the generator polynomial $X^{16} + X^{15} + X^2 + 1$, producing a quotient, which is discarded, and the remainder, which is the desired CRC value to be saved.

Because of the volatile nature of RAM, simple checksums or CRCs are not viable. One way of protecting against errors to memory is to equip it with extra bits used to implement some Hamming code (Moon, 2005). Depending on the number of extra bits, known as the syndrome, errors to one or more bits can be detected and even corrected. Such effective coding schemes can be used with ROM, as well.

Integrated circuits that implement Hamming code error detection and correction (EDC) are available commercially. During a normal memory fetch or save, the data must pass through the EDC chip before going into or out of memory. Besides, the chip compares the data against the check bits and makes corrections if necessary. The chip also sets a readable flag, which indicates that either a single- or multiple-bit error was detected. Realize, however, that the error is not corrected in memory during a read cycle, so if the same erroneous data is fetched again, it must be corrected again. When data is stored in memory, however, the correct check bits for the data are computed and stored along with the data, thereby fixing any errors. This process is called RAM scrubbing (Mariani and Boschi, 2005).

In RAM scrubbing, the contents of a RAM location are simply read and written back. The error detection and correction occurs on the system bus, and the data to be corrected is reloaded into an intermediate register. Upon writing the data back to the memory location, the correct data and syndrome are stored. Thus, the error is corrected in memory, as well as on the bus. RAM scrubbing is used, for instance, in the Space Shuttle's inertial measurement unit (Laplante, 1993). The obvious disadvantages of EDC are that additional memory is needed for the scheme (6 bits for every 16 bits), and an access-time penalty of about 50 ns per access is incurred if an error correction is made. Finally, multiple-bit errors cannot be corrected.

In the absence of EDC hardware—as is usual in most embedded systems—straightforward techniques can be used to verify the integrity of RAM-type memory. These tests are usually run upon initialization, but they could also be implemented in slow cycles if interrupts are appropriately disabled. It is typically desired to exercise the address and data buses as well as the memory cells. This is accomplished by writing and then reading back certain bit patterns to every memory location. The bit patterns are carefully selected so that any stuck-at faults, as well as possible cross talk between wires, can be detected. Bus wires do not always reside alongside by bit location, however, so that various crosstalk situations may arise.

In embedded systems, A/D converters, D/A converters, analog and digital multiplexers, digital I/O, and the like may need to be tested after every power-up and also continually. Such interface modules can have built-in watchdog timer circuitry to indicate that the device is still online. The software can check

for watchdog timer overflows and either reset the corresponding device or indicate a specific failure.

8.4.5 Spurious and Missed Interrupts

Extraneous and unwanted interrupts not due to time loading are called spurious (or "phantom") interrupts. Such interrupts can destroy algorithmic integrity and cause runtime stack overflows or system crashes. Spurious interrupts are caused by noisy hardware, power surges, electrostatic discharges, or single-event upsets. Missed interrupts can be caused in similar ways. In either case, hard real-time deadlines can be compromised, leading to system failure. It is the goal, therefore, to transform these hard errors into some kind of tolerable soft errors.

Spurious interrupts can be tolerated by using redundant interrupt hardware in conjunction with a voting scheme. Similarly, the device issuing the interrupt can issue a redundant check, such as using direct memory access (DMA) to send a confirming flag. Upon receiving the interrupt, the handler routine checks the redundant flag. If the flag is set, the interrupt is legitimate. The handler should then clear the flag. If the flag is not set, the interrupt is bogus and the handler routine should exit quickly and in an orderly fashion. The additional overhead of checking the redundant flag is minimal relative to the benefit derived. Of course, extra stack space should be allocated to allow for at least one spurious interrupt per cycle to avoid stack overflow. Stack overflow caused by repeated spurious interrupts is called a "death spiral."

Missed interrupts are more difficult to deal with. Software watchdog timers can be constructed that must be set or reset by the task in question. Tasks running at a higher priority or at a faster rate can check these memory locations to ensure that they are being accessed properly. If not, the dead task can be restarted or an error indicated. The surest method for sustaining integrity in the face of missed interrupts is through the design of robust algorithms, but that wide topic is beyond the scope of this text.

8.5 SOFTWARE TESTING AND SYSTEMS INTEGRATION

There is more than a subtle difference between the common terms bug, defect, fault, and failure. Use of "bug" is, in fact, discouraged, since it somehow implies that an error crept into the program through no one's action, which is, of course, not true. The preferred term for an error in requirement, design, or program code is either "error" or "defect." Furthermore, the manifestation of a defect during the operation of the software system is called a fault. And a fault that causes the software system to fail to meet one of its requirements is a failure.

Verification and validation of the software are crucial phases of the development process. Verification determines whether the outcomes of a given phase of the software development process fulfill the requirements established

during the previous phase. Thus, verification answers the question, "Am I building the software as specified?"

Validation, on the other hand, determines the correctness of the final software with respect to the user's explicit needs and requirements. Hence, validation answers the question, "Am I building the right software?"

Testing is the execution of a program or partial program with known inputs (excitations) and outputs (responses) that are both predicted and observed for the purpose of finding faults or deviations from the requirements.

Although effective testing is supposed to flush out errors, this is just one of its purposes. The other is to increase trust in the software system. Perhaps once, software testing was thought of as intended to remove *all* errors as will be seen in the following vignette. But testing can only detect the presence of errors, not the absence of them; therefore, *it can never be known when all errors have been detected*. Instead, testing must increase faith in the system, even though it still may contain undetected faults, by ensuring that the software meets its requirements. This objective places emphasis on solid design techniques and a well-developed requirements document. Moreover, a formal test plan must be created that provides criteria used in deciding whether the system has satisfied the requirements.

Vignette: Remove All Errors or Get Fired!

A state-of-the-art embedded control system was delivered to a beta customer. Due to schedule problems, the software was not thoroughly tested, and almost daily, a new error was found by the users of the embedded system. Understandably, the customer was unhappy. One evening, the regional manager who had sold the system to that customer called the software engineer in charge. The regional manager wanted to know "when the last error would be corrected." And the software engineer answered truthfully "I do not know and nobody will ever know." This blunt answer made the regional manager so upset that he immediately called to the vice president of engineering and requested him to fire such an impudent engineer.

Fortunately, the rate of detected errors began to decline steeply, and, in a few weeks, the customer stopped complaining. Hence, the faith in the new software had reached a satisfactory level—although the last error was probably never detected. In addition, the intrepid software engineer was not fired, but he surely learned to put more effort on testing in his future projects.

An in-depth treatment of software testing is available in Patton (2006). Moreover, five thought-invoking views on software testing and industry needs were provided by Glass et al. (2006).

8.5.1 Testing Techniques

There is a wide range of testing techniques for unit- and system-level testing, as well as for integration testing. Some techniques may be interchangeable,

while others are not. Any one of these testing techniques can be either insufficient or not computationally feasible for real-time systems. Therefore, some combination of multiple techniques is usually employed. Recently, commercial and open-source user-guided test case generators have emerged. These tools, such as XUnit (Meszaros, 2007), can greatly facilitate many of the testing strategies to be discussed shortly.

Several methods can be used to test individual modules or code units. These techniques can be used by the unit author or by an independent test team to exercise each code unit in the system. The same techniques can also be applied to subsystems, that is, collections of modules related to the same function.

In *black box testing*, only inputs and outputs of the code unit are considered; how the outputs are generated based on a particular set of inputs is totally ignored. Such a technique, being independent of the implementation of the module, can be applied to any number of modules with the same functionality. But this technique does not provide any insight into the programmer's skills in implementing the module. Consequently, dead or unreachable code cannot be detected.

For each module, a number of test cases need to be generated. This number depends on the number of inputs, the functionality of the module, and so forth. If a module fails to pass a single module-level test, then the detected error must be repaired, and all previous module-level test cases are rerun to prevent the repair from causing other errors.

Some widely used black-box testing techniques include:

- Exhaustive testing
- Boundary-value testing
- Random test-case generation
- Worst-case testing

An important aspect of using black-box testing techniques is that clearly defined interfaces to the software modules are required. This places additional emphasis on the application of Parnas partitioning principles (discussed in Chapter 6) to module design.

Brute-force or *exhaustive testing* involves presenting each code unit with every possible input combination. Exhaustive testing works well in the case of a small number of inputs, each with a limited input range, for example, a code unit that evaluates a small number of Boolean inputs. A major problem with exhaustive testing, however, is the combinatorial explosion in the number of test cases. For instance, for the program code that will deal with raw accelerometer data, altogether $2^{16} \cdot 2^{16} \cdot 2^{16} = 2^{48}$ test cases would be required (three 16-bit acceleration components, a_x, a_y and a_z), which is not reasonable.

Corner-case or *boundary-value testing* solves the problem of combinatorial explosion by testing just a tiny subset of the input combinations identified as meaningful "boundaries" of the input space. For example, consider a code unit with five different inputs, each of which is a 16-bit signed integer. Approaching

the testing of this code unit using exhaustive testing would require $2^{16} \cdot 2^{16} \cdot 2^{16} \cdot 2^{16} \cdot 2^{16} = 2^{80}$ test cases. However, if the test inputs are restricted to every combination of the minimum, maximum, and mean values for each input, then the test set would consist of $3^5 = 243$ test cases, which is a reasonable number. A test set of this size can be handled easily with automatic test case generation.

Random test case generation, or statistically based testing, can be used for both unit- and system-level testing. This kind of testing involves subjecting the code unit to numerous randomly generated test cases over some period of time. The purpose of this approach is to simulate execution of the software under virtually realistic conditions.

The randomly generated test cases are based on determining the underlying statistics of the expected inputs. Such basic statistics are usually collected by expert users of similar systems, or, if none exist, by educated guessing. The theory is that system reliability will be enhanced if prolonged usage of the software system can be simulated in a controlled environment. The major drawback of such a technique is that the underlying probability distribution functions for the input variables may be unavailable or incorrect. Besides, randomly generated test cases are likely to miss special conditions with low probability of occurrence.

Pathological-case or *worst-case testing* deals with those test scenarios that might be considered highly unusual or even unlikely. It is often the case that these exceptional cases are exactly those for which the code is likely to be poorly designed, and therefore, to fail. For instance, in the inertial measurement system, while it might be highly unlikely that the system will achieve the maximum acceleration that can be represented in a 16-bit scaled number, this worst case still needs to be tested.

Of course, there are many other forms of black-box testing, including equivalence class testing, all-pairs testing, and decision table based testing (Jorgensen, 2008).

An obvious disadvantage of black-box testing is that it does not recognize unreachable or dead code. In addition, it may not test all of the flow paths in the module. Another way to look at this is that black-box testing only tests what is expected to happen, not what was not intended. Clear-box, glass-box, or *white-box testing* techniques can be used to deal with this problem. The fundamental difference between black- and white-box testing is illustrated in Figure 8.8.

Whereas black-box tests are data driven, white-box tests are logic driven, that is, they are designed to exercise all paths in the code unit. For example, in the nuclear plant monitoring system, all error paths would need to be tested, including those pathological situations that deal with simultaneous or multiple failures.

White-box testing also has the advantage that it can discover those code paths that cannot be executed. Such unreachable code is undesirable because it is likely a sign that the underlying logic is incorrect, because it wastes memory space, and since it might inadvertently be executed in the case of the corruption of the CPU's program counter.

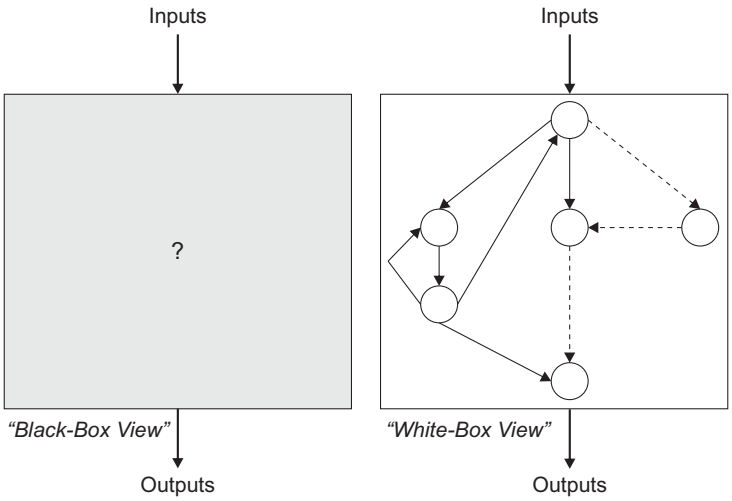


Figure 8.8. Black- and white-box views of a software module for the tester.

Code inspections or group walkthroughs are a kind of white-box testing in which code is inspected line-by-line by a group of experienced programmers. Carefully organized and conducted walkthroughs have been shown to be much more effective than traditional testing techniques.

In code inspections, the author of some collection of software modules presents each line of code to a competent review group, which can detect errors as well as discover ways for improving the implementation. This audit also provides possible control of the coding standards. Finally, unreachable code can be discovered, too.

Formal program proving is another kind of white-box testing using formal methods in which the code is treated as a theorem and some form of calculus is used to prove that the program is correct.

A program is said to be *partially correct* if it produces the correct output for each input when it terminates. It is said to be *correct* if it is partially correct and it always terminates. Hence, to verify a program is correct, partial correctness must be demonstrated first, and then it must be demonstrated that the program terminates.

To illustrate formal program verification, consider the following example. It is casual since some of the rigorous mathematics are omitted for ease of understanding.

Example: Formal Program Verification

Consider a function to compute the power a^b, where a is a floating-point number and b is a nonnegative integer (type and range checking are omitted from the verification, since it is expected that these are performed by the run-time library).

```
float power(float real, unsigned b)
{
 if (b==0)
   return 1;
 else
   return a*power(a,b-1); /* recursion */
}
```

In a real-time sense, it is important to show that this program always terminates, that is, unbounded recursion does not occur. To show this, note that `b` is a loop invariant, and that `b` is a monotonically decremented integer. Hence, `b` will eventually become `0`, which is the explicit termination condition.

Next, to demonstrate partial correctness, note that

$$a^b = \prod_{i=1}^{b} a, \quad b \in Z^+ \text{ and } b \neq 0$$
$$a^b = 1, \qquad\qquad b = 0$$

Recognizing that the program under verification is called `b` times (recursively) through the `else` condition and only once through the `if` condition, yields the equality shown.

Hence, the program is correct.

In its rigorous form, formal verification requires a higher level of mathematical sophistication and is appropriate, generally, only for limited, mission-critical situations because of the intensity of analysis activity.

Furthermore, a testing process that complements object-oriented design and programming can significantly increase the programmer's productivity, software quality, as well as reuse potential. There are three principal issues in *testing object-oriented software*:

1. Testing the base class
2. Testing external code that uses a base class
3. Dealing with inheritance and dynamic binding

Without inheritance, testing object-oriented code is not very different from simply testing abstract data types. Each object has some data structure, such as an array, and a set of member functions to operate. There are also member functions to operate on the object. These member functions are tested like any other function using black-box or white-box techniques.

In a good object-oriented design, there should be a well-defined inheritance structure. Therefore, most of the tests from the base class can be used for testing the derived class, and only a small amount of retesting of the derived class is required. On the other hand, if the inheritance structure is bad, for

instance, if there is inheritance of implementation (where code is used from the base class), then additional testing will be necessary. Hence, the price of using inheritance poorly is having to retest all of the inherited code. Finally, dynamic binding requires that all cases have to be tested for each binding possibility.

Effective testing is guided by information about likely sources of error. The combination of polymorphism, inheritance, and encapsulation is unique to object-oriented languages, presenting such opportunities for an error that do not exist in procedural programming languages. Here, the main rule is that if a class is used in a new context, then it should be tested as if it were new.

Further guidelines for testing object-oriented software are available in McGregor and Sykes (2001).

Test-first coding (or test-driven design) is a code production approach normally associated with eXtreme Programming (English, 2002). In test-first coding, the test cases are designed by the software engineer who will eventually write the code. The advantage of this approach is that it forces the software engineer to think about the code in a different way that involves focusing on "breaking down" the software. Those who use this technique report that, while it is sometimes difficult to change their way of thinking, once the test cases have been designed, it is actually easier to write the program code, and debugging becomes much easier because the unit-level test cases have already been written. Test-first coding is not really a testing technique, it is a design and analysis approach, and it does not obviate the need for testing.

As it turns out, cyclomatic complexity measures the number of linearly independent paths through the code, and hence, provides an indication of the minimum number of test cases needed to exercise every code path. To determine the linearly independent paths, McCabe developed an algorithmic procedure, called the *baseline method*, to determine a set of basis paths (Emergy and Mitchell, 1989).

First, a clever construction is followed to force the complexity graph to look like a vector space by defining the notions of scalar multiplication and addition along paths. Then basis vectors for this space are determined. The method proceeds with the selection of a baseline path, which should correspond to some "ordinary" case of program execution along one of the basis vector paths. McCabe advises choosing a path with as many decision nodes as possible. Next, the baseline path is retraced, and in turn, each decision is reversed, that is, when a node of outdegree of greater than two is reached, a different path must be taken. Continuing in this way until all possibilities are exhausted, it generates a set of paths representing the entire test set (Jorgensen, 2008). For example, consider Figure 8.2. In this case, the cyclomatic complexity was computed to be 5, indicating that there are five linearly independent test cases. Tracing through the graph, the first path is **adcf**. And following McCabe's procedure yields the other four paths **acf**, **abef**, **abeb**, and **abea**.

Function and feature points can also be used to determine the minimum number of test cases needed for adequate coverage. The International Function

Point Users Group (http://www.ifpug.org/) indicates that there is a strong relationship between the number of test cases, software defects, and function points. Accordingly, the number of acceptance test cases can be estimated by multiplying the number of function points by 1.2, which is the factor suggested by McCabe. For instance, if a project consists of 200 function points, then 240 test cases would be needed.

An experimental framework for comparing software testing techniques from an industrial perspective was proposed by Eldh et al. (2006).

8.5.2 Debugging Approaches

In real-time systems, testing methods often affect the systems that they test. When this is considered harmful, nonintrusive testing should be used. For example, when bypassing code during debugging, do not use conditional branching. Conditional branching affects timing and can introduce subtle timing problems. Conditional compilation, on the other hand, is more appropriate in these instances. In conditional compilation, selected code is included only if a particular compiler directive is set, and hence it does not affect timing in the production version.

Programs can be affected by syntactic or logic errors. Syntactic or syntax errors arise from the failure to satisfy the rules of the programming language. A good compiler will always detect syntax errors, although the way that it reports an error often can be misleading. For instance, in a C program a missing `}` may not be detected until many lines after it should have appeared. Besides, some compilers only report vaguely "syntax error" rather than, for example, "missing `}`."

In logic errors, the program code adheres to the rules of the language, but the algorithm that is implemented is somehow wrong. Logic errors are more difficult to diagnose because the compiler cannot detect them. Nevertheless, a few basic rules may help you to find and eliminate logic errors:

- Document the program carefully and appropriately. Each nontrivial line of code should include an explanatory comment. In the course of commenting, logic errors may be detected.
- Where a symbolic debugger is available, use steps, traces, breakpoints, skips, and so on to isolate the logic error.
- Use automated testing where possible. Open-source test generators are available, for instance, the XUnit family (Meszaros, 2007), which includes JUnit for Java and CUnit for C++. These tools help generate test cases and are used for ongoing unit and regression testing of components or classes.
- In the case of a plain command-line environment, such as Unix/Linux, use print statements to output intermediate results at checkpoints in the code. This may help detect logic errors.

- In case of an error, comment out necessary portions of the code until the program compiles and runs. Add in the commented-out code, one feature at a time, checking to see that the program still compiles and runs. When the program either does not compile or runs incorrectly, the last code increment is involved in the error.

Finding and eliminating errors effectively in real-time systems is as much art as it is science, and the software engineer develops these intuitive skills gradually over time with practice. In many cases, code audits or walkthroughs can be particularly helpful in finding logic errors.

Source-level debuggers are software tools that provide the ability to step through code at either an assembly or high-level language level. They are extremely useful in module-level testing. However, they are less useful in system-level debugging, because the real-time aspect of the system is necessarily affected or even disabled.

Debuggers can be obtained as part of compiler support packages or in conjunction with logic analyzers. For example, `sdb` is a generic name for a symbolic debugger associated with Unix and Linux. `sdb` allows the software engineer to single step through the source language code and view the results of each step.

In order to use the symbolic debugger, the source code must be compiled with a particular option set. This has the effect of including special run-time code that interacts with the debugger. Once the code has been compiled for debugging, then it can be executed "normally." For instance, in the Unix/Linux environment, the program can be started normally from the `sdb` debugger at any point by typing certain commands at the command prompt. Nonetheless, it is often more useful to single step through the source code. Lines of code are displayed and executed one at a time by using the step command. If the executed statement is an output statement, it will output to the screen accordingly. If the statement is an input statement, it will await user input. All other statements execute as usually. At any point in the single-stepping process, individual variables can be examined or set. There are many other features of `sdb`, such as breakpoint setting, which are common in all debuggers. In more sophisticated operating environments, a graphical user interface is provided, but essentially, these tools provide the same functionality.

Very often when debugging a new program, the Unix operating system will abort execution and indicate that a core dump has occurred. This is a signal that some fault has occurred. A core dump creates a rather large file named `core`, which is often removed before proceeding with the debugging. But `core` contains some valuable debugging information, especially when used in conjunction with `sdb`. For example, `core` contains the last line of the program that was executed and the contents of the function-call stack at the time of the fault. `sdb` can be used to single step up to the point of the core dump to identify its cause. Later on, breakpoints can be used to quickly come up to this particular line of code.

A logical approach to debugging both software and hardware is presented by Agans (2002), where he suggests the nine "bug-finding rules," R1–R9, for the practitioner:

R1. "Understand the system."
R2. "Make it fail."
R3. "Quit thinking and look."
R4. "Divide and conquer."
R5. "Change one thing at a time."
R6. "Keep an audit trail."
R7. "Check the plug."
R8. "Get a fresh view."
R9. "If you didn't fix it, it ain't fixed."

These general rules provide a useful checklist for anybody involved with debugging.

8.5.3 System-Level Testing

Once individual modules have been tested, then all subsystems and the entire system need to be tested. In larger systems, the process can be broken down into a series of subsystem tests, and then a test of the complete system.

System testing treats the software system as a black box so that one or more of the black-box testing techniques can be applied. System-level testing occurs after all modules pass their unit test. At this point, the coding team hands the software over to the testing team for validation. If an error occurs during system-level testing, the error must first be repaired. Then every test case involving the changed module must be rerun, and all previous system-level tests must be passed in succession. The collection of system test cases is commonly called a system test suite.

Burn-in testing is a type of system-level testing that seeks to flush out those failures appearing early in the life of the real-time system, and thus to improve the reliability of the delivered software product. System-level testing may be followed by alpha testing, which is a type of validation consisting of internal distribution and exercise of the software. This testing is followed by beta testing, where preliminary versions of validated software are distributed to friendly customers, who test the software under actual use. Later in the life cycle of the software, whenever corrections or enhancements are added, regression testing is mandatory. Figure 8.9 shows the various phases of a typical testing hierarchy.

Regression testing, which can also be performed at the module level, is used to validate the modified software against the old set of test cases that has already been passed. Any new test case needed for the enhancements are then added to the test suite, and the software is validated as if it were a new product.

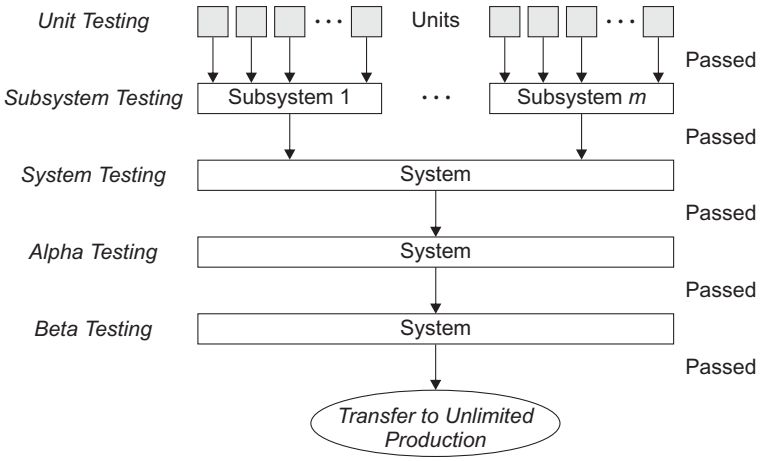


Figure 8.9. Hierarchical bottom-up sequence of software testing phases.

Regression testing is also an integral part of integration testing as new modules are added to the tested subsystem.

The principal tenet of *cleanroom software development* is that given sufficient time and with care, error-free software can be written. Cleanroom software development relies heavily on group walkthroughs, code inspections, and formal program verification. It is taken for granted that software specifications exist that are sufficient to completely describe the system. In this approach, the development team itself is not allowed to test any code as it is being developed. Rather, syntax checkers, group walkthroughs, code inspections, and formal verifications are used to ensure code integrity. Statistically based testing is then applied at various stages of software implementation by a separate test team. This technique produces documentation and program code that are more reliable and maintainable, as well as easier to test than other development methods.

The program is developed by slowly "growing" features into the code, starting with some baseline of functionality. At each milestone, an independent test team checks the code against a set of randomly generated test cases based on a set of statistics describing the frequency of use for each feature specified in the requirements. This group tests the code incrementally at predetermined milestones, and either accepts or returns it to the development team for correction. Once a functional milestone has been reached, the development team adds to the "clean" code, using the same techniques as before. Thus, like an onion skin, new layers of functionality are added to the software system until it completely satisfies the requirements.

In another type of testing, *stress testing*, the software system is subjected to a large disturbance in the inputs (for instance, a large burst of interrupts), followed by smaller disturbances spread out over a longer period of time. One

objective of this kind of testing is to see how the system fails, either gracefully or catastrophically.

Stress testing can also be useful in dealing with particular cases and conditions where the system is under heavy load. For example, in testing for memory or processor utilization in conjunction with other application and operating system resources, stress testing can be used to determine whether performance is acceptable. An effective way to stress test, for instance, is to generate a configurable number of tasks in a test program and subject the software to them. Running such tests for long periods of time also has the benefit of checking for possible memory leaks (such as stack overflows).

One of the challenges in testing real-time systems is dealing with *partially implemented systems*. Many of the problems that arise are similar to those found in dealing with prototype hardware. There are straightforward strategies involving creating stubs and special drivers to deal with missing components at the interface. Commercial and open-source test generators can be helpful in these cases. But the strategies involved for testing real-time systems are nontrivial.

Lastly, the *test plan* should follow the requirement to document item by item, providing criteria that are used to judge whether the required item has been met. A set of test cases is then written, which is used to measure the criteria set out in the test plan. Writing such test cases can be difficult when a complicated user interface is part of the requirements. The test plan includes criteria for testing the real-time software on a module-by-module or unit level, as well as on subsystem and system levels.

8.5.4 Systems Integration

Integration is the process of combining partial functionality to form the complete system functionality. Because real-time systems are usually embedded, the integration process involves both multiple software units and hardware. Each of these parts potentially has been developed by different teams or individuals within the project organization. Although it is presumed that the parts have been rigorously tested and verified separately, the overall behavior of the embedded system, and conformance to most of the software requirements, cannot be tested until the system is wholly integrated. Software integration is further complicated when both software and hardware are new. In such situations, it may be hard to identify whether a particular fault is caused by a software or hardware error.

The software integration phase has the most uncertain schedule and is typically the cause of project cost overruns. Moreover, the stage has been set for failure or success at this phase, by the specification, design, implementation, and testing practices used throughout the software development life cycle. Hence, by the time of software integration, it may be very difficult to identify and fix problems. Indeed, many modern programming practices were devised to ensure arrival at this stage with the fewest errors in the source code. For

example, lightweight agile methodologies, such as eXtreme Programming, tend to reduce these kinds of problems (English, 2002).

Fitting the pieces of the software system together from its individual components is a tricky process, especially for embedded systems. Parameter mismatching, variable name mistyping, and calling sequence errors are some of the typical problems encountered during system integration.

The system unification process consists of linking together the tested software modules drawn in an orderly fashion from the source-code library. During the linking process, errors are likely to occur that relate to unresolved external symbols, memory assignment violations, page link errors, and the like. These problems must, of course, be resolved. Once resolved, the loadable code or load module, can be downloaded from the development environment to the target platform. This is achieved in a variety of ways, depending on the system architecture. In any case, once the load module has been created and loaded into the target platform, testing of timing as well as hardware/software interaction can begin.

Final system testing of embedded systems can be a truly demanding process, often requiring weeks. During system validation, a careful test log must be kept, indicating the test case number, results, and disposition. Table 8.4 is a sample of such a test log for the elevator control system. If a system test fails, it is imperative, once the problem has been identified and presumably corrected, that all affected tests be rerun. These include:

- All module-level test cases for any module that has been changed
- All related subsystem-level test cases
- All system-level test cases

Even though the module-level test cases and previous (sub)system-level test cases have been passed, it is imperative that these be rerun to ensure that no side effects have been introduced during error repair.

As mentioned before, it is not always easy to identify sources of error during a system test. Fortunately, a number of hardware and software tools are available to assist in the validation of embedded systems. Versatile testing tools pave the way for ultimate success—especially in deeply embedded systems.

An *oscilloscope* is not regarded as a software-debugging tool, but it is useful in embedded software environments. Oscilloscopes can be used for validating

TABLE 8.4. Sample Test Log for Elevator Control System

Test Number	Reference Requirements Number	Test Name	Status	Date	Tester
MO27	3.4.1	Attendant service	Pass	11/3/10	S.J.O.
MO28	3.4.2	Independent service	Pass	11/4/10	P.A.L.
MO29	3.4.3	Fireman service	Fail	11/4/10	S.J.O.

interrupt integrity, discrete signal issuance and receipt, and for monitoring clocks.

The *logic analyzer* is an important tool for debugging embedded software. It can be used to capture data or events, to measure individual instruction times, or to time sections of code. Moreover, the availability of programmable logic analyzers with integrated debugging environments has further enhanced the capabilities of the systems integrator.

Advanced logic analyzers include built-in disassemblers for effective debugging, as well as performance analysis and even code profiling features. These integrated environments make the identification of performance bottlenecks particularly convenient.

No matter how elaborate, all logic analyzers have the same basic functionality. This is shown in Figure 8.10. The logic analyzer is connected to the system under test by connecting probes that sit directly on the address and data buses. A clock probe connects to the memory-access synchronization clock. Upon each memory access, the corresponding address and data are captured by the logic analyzer and stored in buffers for transfer to the logic analyzer's main memory, from which they can be processed for display. Using the logic analyzer, the software engineer can capture specific memory locations and data for the purposes of timing or for verifying execution of a specific code segment. The logic analyzer can be used to time accurately an individual machine-language instruction, segments of code, or an entire task.

During module-level debugging and systems integration of embedded software/hardware, the abilities to single-step the CPU, set the program counter, and insert into and read from memory is extremely important. These capabilities in conjunction with the symbolic debugger are keys to successful

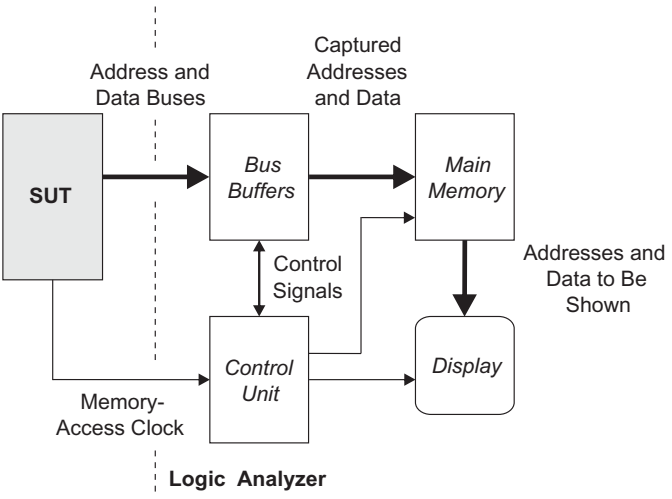


Figure 8.10. A logic analyzer connected to the system under test (SUT).

integration of real-time systems. In an embedded environment, however, this capability is provided by an *in-circuit emulator* (ICE). In-circuit emulation uses special hardware in conjunction with software to emulate the target CPU while providing the aforementioned debugger features. Typically, the ICE plugs into the chip carrier or board slot normally occupied by the CPU. In addition, external wires may connect to an emulation system. Access to the emulator is provided directly or via a workstation.

In-circuit emulators are useful for single-stepping through critical portions of code. In-circuit emulators are not typically useful in timing tests, however, because subtle timing changes can be introduced by the emulator hardware.

When integrating and debugging embedded systems, complementary *software simulators* are often needed to stand in for hardware or inputs that do not exist or that are not readily available, for instance, to generate simulated accelerometer or gyro readings where real ones are unavailable at the time. The author of the simulator program has a task that is by no means easy, since the software must be written to mimic exactly the hardware specification, especially in timing characteristics. Moreover, the simulator must be thoroughly tested. Nevertheless, many real-time systems have been successfully validated and integrated with software simulators, only to fail when connected to the actual hardware environment.

A deliberate approach must be used when performing systems integration to ensure system integrity. Failure to do so can lead to cost escalation and frustration. Software integration approaches are largely based on experience and insight. The following represents a viable strategy for software integration.

In any real-time operating system (RTOS), it is important to ensure that all tasks in the system are being scheduled and dispatched properly. Thus, the first goal in integrating the embedded system is to ensure that each task is running at its prescribed rate, and that context is saved and restored correctly. This is done without performing any application functions within those tasks; the application functions are added later.

As discussed before, a logic analyzer is particularly useful in verifying cycle rates by setting the triggers on the starting location of each of the tasks involved. During debugging, it is most helpful to establish the fact that cyclic processes are being called at the appropriate rates. Until the system cycles properly, the application code associated with each of the tasks should not be added. The success of this method depends on the fact that *one change at a time* is made to the system so that when the system becomes corrupted, the problem can be isolated with a reasonable effort.

The overall approach is depicted in Figure 8.11, and it involves establishing a baseline of plain RTOS components (no application functions). This ensures that timer interrupts are being handled properly, and that all cycles are running at their prescribed rates, without worry about interference from application code. Once the baseline is successfully established, small sections of application code are added and the cycle rates verified. If an error is detected, it is

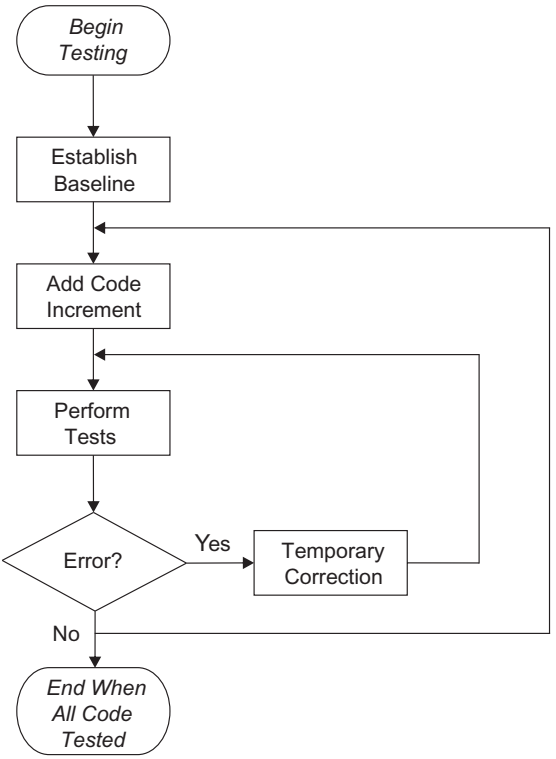


Figure 8.11. A straightforward integration strategy (Laplante, 2003).

corrected temporarily (to save time) when possible. If the correction (or "patch") succeeds in restoring the cycle rates properly, then more code is added. This ensures that the real-time system is grown incrementally, with an appropriate baseline at each stage of the integration. Such an approach represents a smoothly phased integration process with regression testing after each phase.

8.5.5 Testing Patterns and Exploratory Testing

This discussion is adapted from Laplante (2009).

There are various traditionally organized patterns (or problem–solution pairs) for testing software. But all of the available software testing pattern catalogues are language specific (e.g., Thomas et al., 2004) or focused on unit testing (e.g., Meszaros, 2007), and we know of no testing pattern languages that specifically focus on the problem of testing real-time systems. What is needed is a set of general-purpose patterns for testing mission critical, real-time systems postintegration. In this regard, exploratory testing is of great value. Exploratory testing seeks to uncover the various sources of uncertainty

described in Section 8.3 and helps to organize a set of abstract testing paradigms that can greatly assist in developing large-scale test cases.

First described by Kaner, exploratory testing is a guided, *ad hoc* technique that incorporates simultaneous learning, test design, and test execution (Kaner, 1988). Although almost exclusively applied to GUI (graphical user interface) testing in commercial applications, exploratory testing is the kind of testing that is often conducted in embedded systems to augment traditional, scripted testing approaches.

Traditional software and systems testing involves techniques that rely on scripted, or context-based testing. That is, for each test, the test engineer defines a particular initial state for the system (including the environment state), a set of inputs to the system, and a set of expected outputs for the test. In exploratory testing, however, testing is conducted in an almost ad hoc manner; the tester simply *uses* the system in certain ways, and then records any anomalies that he encounters. Exploratory testing is "almost ad hoc" because the usage is actually guided by a behavior pattern driven by a particular posture that is adopted by the tester.

Consider the following tourism metaphor: scripted-based testing involves a traveler who follows a planned itinerary. Exploratory testing involves a traveler who uses his own instincts and personal agenda to guide his explorations. Continuing with the metaphor, we may have many types of traveler personalities, and these personalities influence the journey of each traveler. Whittaker describes many such journeys in his book (Whittaker, 2009). For instance, he notes that in big cities, the local people avoid tourist traps. The software analogy is the set of features avoided by expert users. In the "Historical District Tour," then, testers deliberately explore those features that the experts would avoid. In the "Hotel District Tour," testers test the functionality of the code that is running behind the scenes when the software is ostensibly at rest. Although Whittaker enumerates a number of tours that would be appropriate for security testing or embedded systems (the "Saboteur's Tour" and "Seedy District Tour;" think about the implications), there is ample room for other tours specifically related to embedded and other real-time systems.

One way to develop a set of useful exploratory tests for real-time systems is to consider the sources of systems uncertainty (Bach, 2004), and then create specific tours that uncover these uncertainties. These explorations can be converted into use, and misuse cases for the purpose of operationalizing the tests, and for regression testing. Consider the following examples.

Example: Environmental Explorations

Environmental explorations simulate uncertainty in the environment in which the system is to be operating. These uncertainties may emanate from operating system anomalies, as in the Mars Pathfinder Mission, or from operational domain disturbances, such as a power surge or a violent storm.

Hence, there needs to be a family of explorations that uncovers these types of problems. Let us call these kinds of explorations a "Bad Weather Tour." To facilitate such explorations, simulations need to be created that can model any kind of adverse operating environment condition that can be imagined. Experience can serve as a guide in these situations.

Example: Input Explorations

Input uncertainties, such as spurious or missed interrupts, anomalous data, and deliberately poisoned data, can lead to a cascading series of failures that overload the system. Typically, an abandonment of the single fault assumption is needed to overwhelm any built-in fault-tolerance mechanisms. Such tests are called "Murphy's Tour," because anything that could go wrong is made to go wrong.

Example: Output Explorations

A software control system can deliver grossly or subtly defective output to the system under control, causing the system response to be perturbed further. The aberrant feedback loop can eventually cause failure in the control system. A set of exploration tests, called "Magic Mystery Tours" (after the Beatles' album of the same name), need to simulate every variant of this scenario.

Example: State Explorations

Internal faults, such as jumping program counter, due to a number of possible scenarios, can lead to uncertainty of program state that is difficult to diagnose, and nearly impossible to recover from. Because these kinds of failures lead to erratic, almost drunken behavior, we call these explorations "Fear and Loathing in Las Vegas Tours" (after the famed counter-culture book by Hunter Thompson).

Example: Behavioral Explorations

There is a broad class of timing and scheduling problems that are the hallmark of real-time systems. But these kinds of problems are very commonly tested and diagnosed using traditional scripted, context-driven testing, so no new exploratory tests are offered at this time.

Example: Language Explorations

Many compilers produce code in ways that appear to be nonlinear. For example, removing a single line of source code can fundamentally change

the compiler output thereafter. The effects of these changes can lead to insidious timing problems and undesirable side effects. Therefore, a set of explorations are needed to test the compiler and other systems programs involved in the production of the executable code (debuggers, linkers, loaders, and so forth). A series of exploratory tests, called "Shakeout Tours" (after the term used for a debugging first voyage for some transportation craft), are needed to uncover these potential problems.

Example: Commercial Off-the-Shelf Explorations

These explorations seek to uncover problems in software furnished by third parties, such as commercial vendors, or open source software. Traditional testing of these components needs to be done. Because "trust but verify" is the hallmark of this testing, we call these "Reagan's Tours" (with respect to nuclear arms verification, President Ronald Reagan declared that it was best to "trust, but verify").

Though never given these colorful names, Laplante and his colleagues used these explorations extensively in testing various embedded systems for avionics applications, including the Space Shuttle Inertial Measurement Unit, satellite systems, and other navigation systems. Work is ongoing to classify and socialize these exploratory tests for other kinds of real-time systems.

8.6 PERFORMANCE OPTIMIZATION TECHNIQUES

Identifying wasteful computation is a preliminary step in reducing code execution time, and hence, the CPU utilization factor. Many traditional approaches employed in compiler optimization (see Chapter 4) can be applied for this purpose, but various other methods have evolved that are specifically oriented toward embedded systems. A small sample of common performance optimization techniques is discussed in the following three subsections.

Moreover, all real-time processing should be done, in principle, at the slowest rate that *can be tolerated*. Checking a discrete temperature for a large lecture hall at faster than 1 second may be wasteful, for room temperature cannot change quickly owing to thermal inertia. In the nuclear power plant, on the other hand, dedicated sensors are used to monitor the core temperature continuously and issue a high-priority service request if an over-temperature is detected.

8.6.1 Scaled Numbers for Faster Execution

In virtually all computers, integer operations are faster than floating-point ones. This fact can be exploited by converting floating-point algorithms into scaled integer algorithms. In the so-called scaled numbers, the least significant

bit (LSB) of an integer variable is assigned a real-number scale factor. Scaled numbers can be added and subtracted together as well as multiplied and divided by a constant—but not by another scaled number. The results are then converted to floating point only at the last computing step, thus saving processing time.

Example: Scaled Acceleration Samples

Suppose an analog-to-digital converter is converting accelerometer data. If the least significant bit of the two's complement 16-bit integer has value $a = 0.0001$ m/s^2, then any acceleration can be represented up to the maximum value of $(2^{15} - 1) \cdot 0.0001$ m/s^2 = 3.2767 m/s^2. The 16-bit number 0000 0000 0000 1101, for instance, represents an acceleration of $13 \cdot 0.0001$ m/s^2 = 0.0013 m/s^2.

A common practice is to convert an integer number x into its floating-point equivalent by $x \cdot a$ and then proceed to use it in calculations directly with other converted numbers; for example, $d = x \cdot a - y \cdot a$, where $y \cdot a$ is a similarly converted floating-point number. Instead, the calculation could be performed in integer form first and then converted to floating point: $d = (x - y) \cdot a$—this would undoubtedly save some time.

For algorithms involving numerous additions and subtractions of like data, scaled numbers can introduce significant time savings. Note, however, that multiplication and division by another scaled number cannot be performed on a scaled number, as those operations would change the scale factor. Besides, accuracy is generally sacrificed by excessive use of scaled numbers. Therefore, a careful error analysis is needed when using scaled numbers for implementing complicated or numerically sensitive algorithms.

Another type of scaled number is based on the property that adding 180° to any angle is analogous to taking its two's complement. This technique, called binary angular measure (BAM) works as follows. Consider the LSB of an n-bit word to be $2^{-(n-1)} \cdot 180°$ with the most significant bit (MSB) of 180°. The range of any angle θ represented this way is $0° \le \theta \le 360° - 2^{-(n-1)} \cdot 180°$. A 16-bit BAM word is shown in Figure 8.12. For better accuracy, BAM can be extended to multiple words. Each n-bit word has a maximum value of:

$$\left(2 - 2^{-(n-1)}\right) \cdot 180° = 360° - \text{LSB}, \tag{8.20}$$

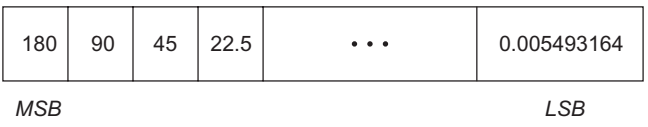


Figure 8.12. A 16-bit BAM word.

with granularity

$$2^{-(n-1)} \cdot 180^\circ = \text{LSB}. \tag{8.21}$$

Now, consider the 16-bit BAM word:

0000 0000 1010 0110

It corresponds to the angle $166 \cdot 2^{-15} \cdot 180^\circ \approx 0.9119^\circ$.

BAM words can be added and subtracted together, as well as multiplied and divided by constants as if they were unsigned integers, and converted at the last stage of the algorithm to produce floating-point results. It is easy to show that the overflow condition for BAM numbers presents no problem as the angle simply wraps around to 0°. BAM is frequently used in navigation software, robotic control, and in conjunction with digitizing imaging devices.

8.6.2 Look-Up Tables for Functions

A further variation of the scaled-number concept uses a stored table of precalculated function values at fixed intervals. Such a table, called a look-up table (Bateman and Yates, 1988), allows for the computation of continuous functions using mostly fixed-point arithmetic.

Let $f(x)$ be a continuous real-valued function and let Δx be the interval size. Suppose it is desired to store n values of $f(x)$ over the range $x \in [x_0, x_0 + (n-1) \cdot \Delta x]$ in an array of scaled integers. Corresponding values for the derivative, $f'(x)$, may also be stored in the table for faster interpolation as will be shown below. The choice of Δx represents a trade-off between the size of the table and the desired resolution of the function. A generic look-up table is given in Table 8.5. Such a table can be used for the interpolation of $f(\hat{x})$, where $x < \hat{x} < x + \Delta x$, by the well-known interpolation formula:

$$f(\hat{x}) = f(x) + (\hat{x} - x) \cdot \overbrace{\frac{f(x+\Delta x) - f(x)}{\Delta x}}^{\text{Derivative when } \Delta x \to 0}. \tag{8.22}$$

This calculation is done using integer instructions except for the final multiplication by the factor $(\hat{x} - x)/\Delta x$ and conversion to floating point. The

TABLE 8.5. A Generic Function Look-Up Table Containing Both the Function and Its Derivative

x	$f(x)$	$f'(x)$
x_0	$f(x_0)$	$f'(x_0)$
$x_0 + \Delta x$	$f(x_0 + \Delta x)$	$f'(x_0 + \Delta x)$
$x_0 + 2\Delta x$	$f(x_0 + 2\Delta x)$	$f'(x_0 + 2\Delta x)$
⋮	⋮	⋮
$x_0 + (n-1)\Delta x$	$f(x_0 + (n-1)\Delta x)$	$f'(x_0 + (n-1)\Delta x)$

accuracy of Equation 8.22 improves obviously when $\Delta x \to 0$. If $f'(x)$ is also stored in the look-up table, then the interpolation formula reduces to:

$$f(\hat{x}) = f(x) + (\hat{x} - x) \cdot f'(x). \tag{8.23}$$

This clearly improves the execution time of the interpolation algorithm (increasing memory space is traded with decreasing execution time).

The main advantage in using look-up tables, of course, is speed. If a table value is found directly and no interpolation is needed, then the approach is much faster than any corresponding series expansion.

Look-up tables are widely used in the implementation of continuous functions, such as the sine, cosine, and tangent functions, as well as their inverses. Because trigonometric functions are used frequently, for example, in conjunction with the discrete Fourier transform (DFT) and discrete cosine transform (DCT), look-up tables can provide considerable speed-up in real-time signal and image processing applications.

In some real-time applications, partial results can be given in order to meet a critical deadline. In cases where software routines are needed to provide mathematical support, complex algorithms are often employed to produce the desired calculation. For instance, a Taylor-series expansion (perhaps using a look-up table for function derivatives) could be terminated prematurely, at a loss of accuracy, but with improved real-time performance. Techniques involving early truncation of a series expansion in order to meet deadlines are called imprecise computation. Imprecise computation may be difficult to apply, however, because it is not usually easy to determine the processing that can be discarded and its overall effects.

8.6.3 Real-Time Device Drivers

In general, a real-time device driver is a piece of system software that forms a high-level interface between the hardware platform and application software, and may use the functionality of the real-time operating system (RTOS) to accomplish this task effectively. It has three main purposes, which all have a more or less explicit connection to real-time performance:

1. To provide an efficient and reliable interface to hardware devices, and achieve minimum input/output overhead by carefully designed and implemented driver functions.
2. To hide the device-specific details from application programmers, and hence improve the programmers' productivity.
3. To isolate a particular hardware platform and devices from the application software, and thus enhance the portability and reusability of software.

A device-driver simplifies the use of real-time devices, such as peripheral interface adapters, data acquisition hardware, wireless network interfaces, and

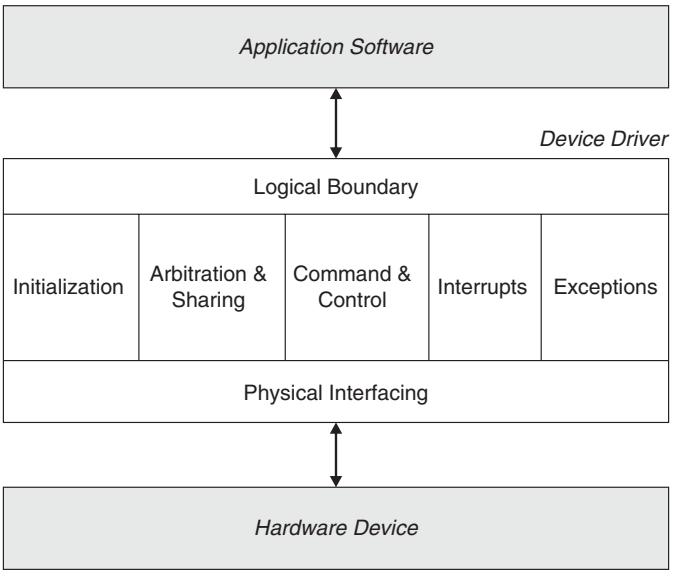


Figure 8.13. Typical functions of a real-time device driver that is interacting closely with the RTOS.

so forth. While the use of peripheral interface adapters is straightforward, the programming of communications network adapters usually goes beyond the scope of engineers developing software for some embedded application. Hence, the device drivers for sophisticated real-time devices are often acquired from the manufacturer of the hardware device, who knows all the functional details of the particular hardware device. Moreover, a device driver may be operating system specific.

Real-time device drivers manage a variety of important functions, which include (see Fig. 8.13):

- Initialization of the device
- Logical (possibly standard) boundary to the application software
- Physical interfacing to the particular device hardware
- Resource arbitration and sharing
- Chip-level command/control sequences
- Interrupt servicing
- Exception handling

In addition, virtual device drivers can be used to emulate the functionality of specific hardware devices during the process of software testing when the real devices are not yet available. Such a virtual driver mimics the behavior of some

physical device, and thus provides a realistic environment for testing real-time software.

Sertić et al. discussed the use of UML for designing a device driver for a real-time Linux environment (Sertić et al., 2003). Both static and run-time models of the device-driver's behavior were developed, including a model of the interrupt handler. Furthermore, the implemented device driver was verified to be computationally efficient and reliable, and it is used in a data-communications application. The proposed design approach is applicable to other device drivers, as well.

To conclude, any real-time device driver is usually such a critical unit of software that the effort used to optimize its performance can be easily justified.

8.7 SUMMARY

After several solid chapters on the design and analysis of real-time systems, this integrative chapter provided a rich composition of additional considerations for the practitioner. Actually, one characteristic that makes the field of real-time systems engineering so fascinating is the multi-dimensionality of the whole field; there are, indeed, many attractive areas to become specialized with. And nobody can likely claim to master them all. This also means that a competitive development team needs members with partially similar and partially different educational focuses, who could then complement each other and form a dynamic group that is more than a sum of its members.

The hidden or passive role of software metrics and associated cost modeling techniques is going to enhance to be enhanced even in smaller organizations, since the cost-consciousness and need for careful project planning are continuously increasing. This trend is advanced by the growing role (and size) of software in embedded systems, as well as the globalizing nature of software development projects. For instance, future smartphones will be increasingly "software products," and even mid-size companies will likely have international software development teams.

Uncertainty management has always been an important topic within embedded systems engineering. Nonetheless, this importance is getting new flavor when autonomous systems, ubiquitous computing, and massively distributed wireless solutions are growing in production volumes. They will surely create fresh challenges for the embedded software developer, too. Such uncertainty challenges are largely related to the issues of software reliability and fault tolerance. Hence, there is a need for a senior-level course on "uncertainty management and fault tolerance in wireless distributed systems." Such a *software-biased* course could be targeted for both computer-engineering and computer-science students.

The demand for high system reliability is extending steadily from high-end real-time systems toward low-end ones. In the foreseeable future, govern-

ments, societies, organizations, and individuals will all be increasingly dependent on real-time computing. This somewhat "blue-eyed" dependency could be seen as a threat, and thus it calls for fault-tolerant solutions.

Software testing is another area that has a significant position in the success of future embedded systems. This significance arises from the growing number and global distribution of embedded products, as well as from the growing size of embedded software. To keep the time-to-market measure acceptable and simultaneously ensure that the software maintenance phase will be satisfactory, cost-wise, it is crucial to increase the use of automatic test case generators, automatic testing environments, as well as design for testability. In addition, carefully developed virtual hardware platforms together with virtual device drivers are necessary to improve the confidence level of early testing phases.

Due to the continuing validity of the Moore's law, the instruction throughput of embedded processors is increasing at a remarkable rate. This obviously reduces the need for traditional performance optimization approaches in all but the most time-critical or expense-sensitive applications. On the other hand, the emerging use of multi-core processors in embedded applications for achieving *true concurrency* of multiple tasks calls for novel performance optimization techniques. Therefore, the area of performance optimization is in the process of stepping into something rather unknown.

Finally, in spite of all the architectural advancements, it is not likely that the availability of hardware-based floating-point support is going to increase in medium- and low-end embedded platforms. This keeps the importance of scaled numbers, look-up tables, and other traditional performance optimization techniques at a steady level.

8.8 EXERCISES

8.1. Research the use of the cyclomatic complexity metric in real-time systems by performing a thorough Web search. How would you conclude your findings?

8.2. Recalculate the cyclomatic complexity metric for the `if-then-else`, `while`, and `until` structures depicted in Figure 8.1.

8.3. Recalculate the *FP* (function point) metric for the inertial measurement system using a set of weightings that assumes that significant off-the-shelf software (say 70%) is to be used. Make assumptions about which factors will be most influenced by the off-the-shelf software. How many lines of C++ code do you estimate you will need?

8.4. Do the same as Exercise 8.3, except recalculate now the FP^{+} (feature point) metric. How many lines of C++ code do you estimate will be needed?

8.5. In *N*-version programming, the different programming teams code independently from the same set of specifications. Discuss the possible disadvantages of this approach.

8.6. Describe the effect (if any) of the BITS and reliability schemes (a)–(d), without appropriately disabling interrupts. How should interrupts be disabled?

(a) CPU instruction set test
(b) CRC calculation for ROM
(c) RAM pattern tests
(d) RAM scrubbing

8.7. Suppose a real-time computer system has 16-bit data and address buses. What test patterns are necessary and sufficient to test the address and data lines, as well as the RAM cells?

8.8. Write a module in the programming language of your choice that generates a CRC check value for a range of 16-bit memory. The modules should take as input the starting and ending addresses of the range, and output the 16-bit check value. Use either CRC-16 (CCITT) or CRC-16 (ANSI) as your generator polynomial.

8.9. A software module is to take as inputs four signed 16-bit integers and produce two outputs, the sum and average. How many test cases would be needed for an exhaustive testing scheme? How many would be needed if just the minimum, maximum, and average values for each input were to be used?

8.10. How much could testing and test case/suite generation be automated in practice? What are the roadblocks to automating a test suite? In programming languages like Java?

8.11. For the example systems discussed throughout this book:

(a) Airline reservation system
(b) Elevator control system
(c) Inertial measurement system
(d) Nuclear monitoring system

which testing methods would you prefer and why?

8.12. An elevator bank monitoring system shows the clock time in hours and minutes on multiple displays. The time is generated by a programmable 16-bit timer, which uses a 50-kHz clock signal for calculating seconds. These seconds are then accumulated to minutes and further to hours by software. However, the users of monitors were initially complaining that the clock time is advancing or lagging up to 7 minutes in a month.

A field engineer made a survey of the problem and noticed that the magnitude of advance or lag remains practically constant on each site, but is dependent on the individual monitoring computer. Based on these observations, a solution for the annoying problem was developed and taken in use in the final testing stage before the complete monitoring system leaves the elevator factory.

What kind of approach would you take with this problem if no hardware modifications were allowed? Hint: the monitoring computer has free parameter space in a Flash memory that can be accessed by a service tool.

8.13. Create a compact look-up table (floating-point numbers with three decimal places) for the tangent function in increments of 1°. Be sure to take advantage of symmetry.

8.14. Suppose x is a 16-bit BAM word representing the angle 225°, and y is another 16-bit BAM word representing 157.5°. Using binary arithmetic, show that $x + y = 22.5°$.

8.15. What are the advantages and disadvantages of writing a BAM object class in an object-oriented language like C++?

REFERENCES

A. Abran, A. Sellami, and W. Suryn, "Metrology, measurement and metrics in software engineering," *Proceedings of the 9th IEEE International Software Metrics Symposium*, Sydney, Australia, 2003, pp. 2–11.

D. J. Agans, *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. New York: AMACOM, 2002.

J. Bach, "Exploratory testing," in *The Testing Practitioner*, 2nd Edition. E. van Veenendaal (Ed.), Den Bosch, The Netherlands: UTN Publishers, 2004, pp. 253–265.

J. M. Bass, G. Latif-Shabgahi, and S. Bennett, "Experimental comparison of voting algorithms in cases of disagreement," *Proceedings of the 23rd Euromicro Conference*, Budapest, Hungary, 1997, pp. 516–523.

A. Bateman and W. Yates, *Digital Signal Processing Design*. London, UK: Pitman, 1988.

L. Bernstein and C. M. Yuhas, *Trustworthy Systems through Quantitative Software Engineering*. Hoboken, NJ: Wiley-Interscience, 2005.

B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

B. W. Boehm et al., *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice-Hall, 2000.

B. W. Boehm and R. Valerdi, "Achievements and challenges in COCOMO-based software resource estimation," *IEEE Software*, 25(5), pp. 74–83, 2008.

S. Chandra, P. Godefroid, and C. Palm, "Software model checking in practice: An industrial case study," *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, 2002, pp. 431–441.

J. C. Coppick and T. J. Cheatham, "Software metrics for object-oriented systems," *Proceedings of the ACM Annual Computer Science Conference*, Kansas City, MO, 1992, pp. 317–322.

S. Elbaum and J. C. Munson, "Investigating software failures with a software black box," in *Proceedings of the IEEE Aerospace Conference*, Big Sky, MT, 2000, vol. 4, pp. 547–566.

S. Eldh, H. Hansson, P. Sasikumar, A. Pettersson, and D. Sundmark, "A framework for comparing efficiency, effectiveness and applicability of software testing techniques," *Proceedings of the Testing: Academic & Industrial Conference—Practice and Research Techniques*, Windsor, UK, 2006, pp. 159–170.

K. O. Emergy and B. K. Mitchell, "Multi-level software testing based on cyclomatic complexity," *Proceedings of the IEEE National Aerospace and Electronics Conference*, Dayton, OH, 1989, vol. 2, pp. 500–507.

A. English, "Extreme programming, it's worth a look," *IT Professional*, 4(3), pp. 48–50, 2002.

R. L. Glass, R. Collard, A. Bertolino, J. Bach, and C. Kaner, "Software testing and industry needs," *IEEE Software*, 23(4), pp. 55–57, 2006.

M. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

M. G. Hinchey and J. P. Bowen (Eds.), *Industrial-Strength Formal Methods in Practice*. London, UK: Springer-Verlag, 1999.

D. M. Hoffman and D. M. Weiss (Eds.), *Software Fundamentals: Collected Papers by David L. Parnas*. New York: Addison-Wesley, 2001.

C. Jones, "Backfiring: Converting lines of code to function points," *IEEE Computer*, 28(11), pp. 87–88, 1995.

C. Jones, *Estimating Software Costs*. New York: McGraw-Hill, 1998.

P. Jorgensen, *Software Testing: A Craftsman's Approach*, 3rd Edition. Boca Raton, FL: CRC Press, 2008.

C. Kaner, *Testing Computer Software*. Blue Ridge Summit, PA: TAB Professional & Reference Books, 1988.

I. Koren and C. M. Krishna, *Fault Tolerant Systems*. San Francisco, CA: Morgan Kaufmann, 2007.

P. A. Laplante, "Fault-tolerant control of real-time systems in the presence of single event upsets," *Control Engineering Practice*, 1(5), pp. 9–16, 1993.

P. A. Laplante, *Software Engineering for Image Processing*. Boca Raton, FL: CRC Press, 2003.

P. A. Laplante, "The certainty of uncertainty in real-time systems," *IEEE Instrumentation & Measurement Magazine*, 7(4), pp. 44–50, 2004.

P. A. Laplante, "Exploratory testing for mission critical, real-time, and embedded systems," *IEEE Reliability Society Annual Technical Report*, 2009. Available at http://paris.utdallas.edu/IEEE-RS-ATR/document/2009/2009–08.pdf, last accessed August 17, 2011.

B. Littlewood, "Learning to live with uncertainty in our software," *Proceedings of the 2nd International Software Metrics Symposium*, London, UK, 1994, pp. 2–8.

M. V. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad smells—humans as code critics," *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Chicago, IL, 2004, pp. 399–408.

R. Mariani and G. Boschi, "Scrubbing and partitioning for protection of memory systems," in *Proceedings of the 11th IEEE International On-Line Testing Symposium*, St-Raphael, France, 2005, pp. 195–196.

T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, 2(4), pp. 308–320, 1976.

J. D. McGregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*. Upper Saddle River, NJ: Addison-Wesley Professional, 2001.

G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ: Addison-Wesley, 2007.

Y. Miyazaki and K. Mori, "COCOMO evaluation and tailoring," *Proceedings of the 8th International Conference on Software Engineering*, London, UK, 1985, pp. 292–299.

T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. Hoboken, NJ: Wiley-Interscience, 2005.

S. J. Ovaska, "Evolutionary modernization of large elevator groups: Toward intelligent mechatronics," *Mechatronics*, 8(1), pp. 37–46, 1998.

R. Patton, *Software Testing*, 2nd Edition. Indianapolis, IN: Sams Publishing, 2006.

R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th Edition. New York: McGraw-Hill, 2000.

F. Saglietti, "Location of checkpoints in fault-tolerant software," *Proceedings of the 5th Jerusalem Conference on Information Technology*, Jerusalem, Israel, 1990, pp. 270–277.

D. Seibt, "Function point method: Characteristics, implementation and application experiences," *Angewandte Informatik*, 29(1), pp. 3–11, 1987.

H. Sertić, F. Rus, and R. Rac, "UML for real-time device driver development," *Proceedings of the 7th International Conference on Telecommunications*, Zagreb, Croatia, 2003, vol. 2, pp. 631–636.

X. Teng and H. Pham, "A software-reliability growth model for N-version programming systems," *IEEE Transactions on Reliability*, 51(3), pp. 311–321, 2002.

J. Thomas, M. Young, K. Brown, and A. Glover, *Java Testing Patterns*. Hoboken, NJ: John Wiley & Sons, 2004.

P. K. Varshney, "Multisensor data fusion," *Electronics & Communications Engineering Journal*, 9(6), pp. 245–253, 1997.

J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs against Errors*. New York: John Wiley & Sons, 1998.

J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Boston: Addison Wesley, 2009.

9

FUTURE VISIONS ON REAL-TIME SYSTEMS

Forecasting is always a risky endeavor, as we have frequently observed, for example, when trying to plan our outdoor activities relying on professional weather forecasts that eventually turn out to be incorrect—it is raining heavily although it was supposed to be sunny. Technology forecasting is especially difficult, particularly if we try to forecast too far into the future. Nevertheless, different kinds of technology forecasts are created by business consultants, research engineers, and scientists. These forecasts are typically used for supporting decision-making processes and for strategic planning purposes. The fundamental methodologies behind forecasting are the use of analogies, extrapolation, and modeling, as well as their various combinations (Makridakis et al., 1998). Conversely, vision creation on the development of some technology is a *speculative* form of technology forecasting that might rely on the established forecasting methodologies but is strongly supported by human intuition—even imagination. Such technology visions are sometimes used in place of actual forecasts or in parallel to complement them, since forecasts do anyhow contain uncertainty.

In this chapter, we provide *selective visions* of the evolution and advancement of real-time systems. The time span of our visions is up to year 2040; hence, some of the visions are necessarily "blue-sky" visions, while others are more "feet-on-the-ground" type. But what is the motivation behind this

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

chapter? Well, our primary intent is to create a stimulating basis for class debates, panel discussions, literature search assignments, and so forth, which could be used as a final stage of a university course on *Real-Time Systems Design*. Besides, these visions are helpful for the practitioner who needs to gain understanding of prospective real-time technologies. It may be easy to disagree with some of our visions, but it is certainly educational to find specific arguments and reasoning to support the disagreement as objectively as possible. The visions we present are derived from our collective insight informed by the most current literature on real-time systems technologies. It must be emphasized, however, that the literature cited in this chapter is a subjectively collected sample—not the byproduct of any comprehensive literature survey. Our insight, on the other hand, has matured during the exciting three decades that we have been involved with real-time systems, particularly the embedded ones.

At a recent European Futurist Conference, some leading futurologists expressed their prognoses (or visions) for year 2020. Below is a condensed list of such prognoses, which have an obvious connection to future real-time systems (Talwar and Pearson, 2010):

- "Augmented reality will be much more mature and a familiar part of our lives."
- "Our interaction with machines will inevitably need to become more 'natural' through the dramatic increase in the use of indirect channels of communication—making machines sensitive to biometric data [and gestures] from which emotional and contextual information can be derived."
- "10 Terabits of computer memory (roughly the equivalent of the memory space in a single human brain) will probably cost just $1000."
- "At the end of each day, I spend 20 minutes reviewing and annotating the downloads from my personal data chips that captured every conversation I had and every image I saw."

We believe that all of these envisioned futures should be reality by 2020.

While the previous visions were expressed by futurologists, the SICE (The Society of Instrument and Control Engineers, Japan) Trans-Division Technology Committee on Embedded Systems has created a roadmap of embedded control systems (Funabashi et al., 2009). Their roadmap covers the years 2015, 2025, and 2035, and includes the following principal milestones:

- 2015: Distributed embedded control systems with multi-core processors and advanced networks.
- 2015: Practical modeling methods from function specification to implementation.
- 2015: Automatic verification methods for control software.
- 2015: Collaboration among enterprises by model-based development methods.

- 2025: Automatic generation of small-scale control software from high-level specifications.
- 2025: Remote maintenance of control software.
- 2035: Automatic software generation from high-level specifications for networked embedded systems.
- 2035: Evolution and self-organizing mechanisms for developing new products.

In addition, Funabashi et al. defined the term "ubiquitous intelligence" as an approach toward *post*-embedded systems, which consists of embedded control systems that are networked with each other and also to computing clouds (Funabashi et al., 2009).

The prognoses of the selected group of international futurologists as well as the thoughtful milestones of the Japanese roadmap provide guidance for our future visions on real-time systems, as will be seen shortly.

Our visions on real-time systems have five dimensions (which, however, are not fully orthogonal): real-time hardware, real-time operating systems, real-time programming languages, real-time systems engineering, and real-time applications. And these distinct vision elements are presented in Sections 9.1–9.5, respectively. Moreover, Section 9.6 summarizes the chapter and Section 9.7 provides a collection of future-vision exercises for class usage. We weave a common thread throughout the chapter by building explicit connections between consecutive sections. Thus, this chapter is more than a collection of individual visions on real-time systems—it is an integrated whole that will hopefully leave the perceptive reader with much to consider.

Furthermore, as any vision of long-term technology development contains more or less uncertainty, it is suggested that the instructor and students would create their individual *confidence pentacles* for the five vision elements of Sections 9.1–9.5 (Exercise 9.6). A vision confidence pentacle has an axis for every vision element, and each axis corresponds to the evaluator's confidence on the contents of the particular section: "1" represents full confidence and "0" no confidence at all (Sick and Ovaska, 2007). Figure 9.1 illustrates such a confidence pentacle with arbitrary confidence values. The likely differences between the instructor's and students' pentacles could form a fruitful basis for classroom discussions.

9.1 VISION: REAL-TIME HARDWARE

The exponential progress of integrated circuit technology has followed Moore's law for more than four decades. In 1965, Gordon Moore predicted that the density of transistors doubles every 2 years. And today, we can have several millions of transistors on a single processor chip. Moore's law can be expressed mathematically as (Powell, 2008):

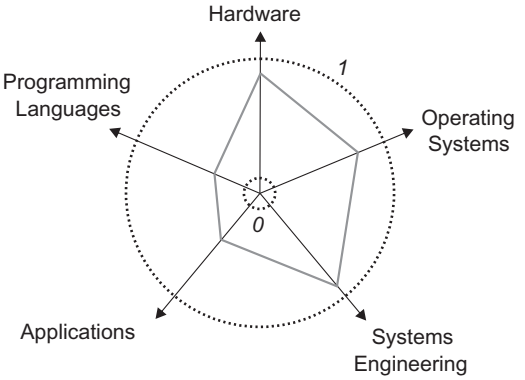


Figure 9.1. An example of the vision confidence pentacle.

$$\hat{N}_2 = N_1 2^{[(y_2 - y_1)/2]}. \tag{9.1}$$

where N_1 is the known number of transistors in year y_1, and $\hat{N}_2$ is the predicted number of transistors in year y_2. To give some perspective for our visions on real-time hardware, let us calculate $\hat{N}_2$ for years 2020, 2030, and 2040, and use a normalized value for the initial number of transistors in 2011, that is $N_1 = 1$. The corresponding values for $\hat{N}_2$ are ≈23, ≈724, and ≈23171, respectively. These numbers will significantly scale up the current "several millions" of transistors, if Moore's law continues to be valid. On the other hand, Powell estimated that the physical quantum limit to Moore's law would be reached in 2036 (Powell, 2008)—but that distant year is close to the end of our visioning period.

A principal concern with the future billion-transistor integrated circuits is their high power consumption leading to severe heat problems (Gea-Banacloche and Kish, 2005). This creates obvious needs to optimize the clock frequency (and operating voltage) in every block of an integrated processor system, because the power consumption is linearly proportional to the applied clock frequency. Hence, different functional blocks will run at different rates, which are no more than "adequate" for the particular function.

Error-tolerant computing is seen as another (but rather "revolutionary") approach that could relieve the power consumption problem in future processor chips (Lammers, 2010). It is based on relaxed thinking that most internal errors that occur during the instruction execution process will be corrected (power is consumed) but small errors could be ignored (power is saved). This somewhat probabilistic approach would necessarily drive computing results away from determinism; and that is something we cannot tolerate in hard and firm real-time systems. Nonetheless, in certain soft real-time systems, such as massive graphics processing, it could be acceptable. Augmented/virtual reality, as well as mobile data-logging applications, are potential users of error-tolerant computing.

Although embedded processors have never been on the leading edge with respect to the number of transistors on a single chip, there will be enormous opportunities to innovate new embedded processor systems as long as Moore's law continues to be valid. These opportunities will be discussed below as our hardware visions.

9.1.1 Heterogeneous Soft Multi-Cores

In 2004, Paul Otellini, the President and CEO of Intel, announced that his company would dedicate "all of our future product designs to multi-core environments" (Patterson, 2010). Although Intel is no longer a major player in the embedded processor field, this shaking announcement was a global starting point for a new era in embedded processor research and development, as well. Since that, a few commercial and many experimental multi-core architectures have emerged for embedded systems use (Levy and Conte, 2009).

Multi-core processor architectures offer the possibility for *true concurrency* in executing multiple tasks (with equal priorities) in a real-time system. In principle, every individual task could run on its own CPU core and thus have all the processing capacity of that core available. However, if the CPU cores on a single chip were equal, this would lead to very inefficient use of computing resources (and excessive power consumption). Since some software tasks are computationally light while others are heavier, and some have a long execution cycle while others have a shorter one, it would be advantageous to have a multi-core processor with an application-specific set of *heterogeneous* CPU cores that could be assigned to individual tasks and run at various voltage settings based on their explicit needs (Hashemi and Ghiasi, 2010).

But there are diverse hard real-time applications, and, hence, also diverse needs for the composition of a heterogeneous multi-core chip. For instance, one eight-task application could need a multi-core chip with four 8-bit RISC microcontroller cores, two 32-bit RISC cores, and two 16-bit digital signal processor cores. Moreover, another four-task system could require just three 8-bit RISC microcontroller cores and one 24-bit digital signal processor core. Naturally, it would not be feasible to have a large variety of heterogeneous multi-core chips available as *standard* components.

The traditional solution to this "variety problem" is an application-specific integrated circuit (ASIC), which could be used for implementing heterogeneous multi-core architectures tailored for a specific application. Nonetheless, the expensive design and manufacturing processes, as well as long turnaround times, make ASICs undesired for most applications. Fortunately, large field-programmable gate arrays (FPGAs) provide an attractive implementation alternative for heterogeneous multi-core processors. It is clearly visible that programmable (or reconfigurable) substrates are replacing the dedicated ASICs rather fast in a broad range of applications (Hashemi and Ghiasi, 2010), and with the continuously increasing number of logic elements, FPGAs can be used for implementing heterogeneous *soft* multi-cores. The individual soft

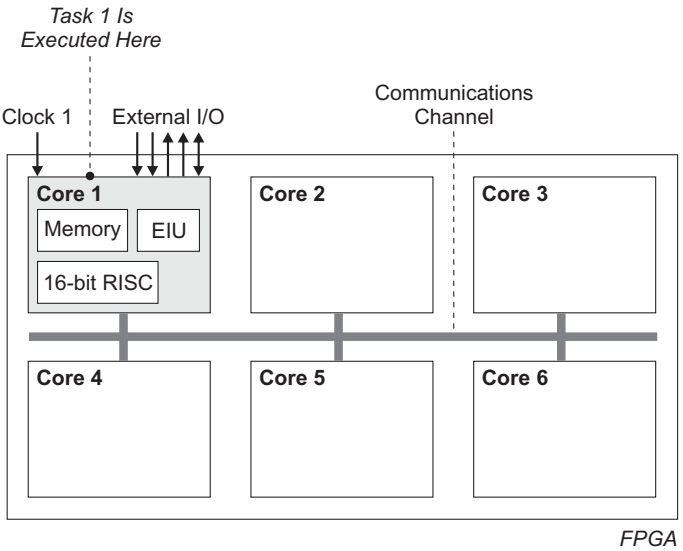


Figure 9.2. A heterogeneous soft multi-core architecture for future embedded systems (only Core 1 is shown with internal details for clarity).

CPU cores are implemented cost-effectively utilizing basic FPGA logic elements only, instead of using any special higher-level elements (except memory blocks) (Elkateeb and Mandepanda, 2009); this guides the different FPGA manufacturers toward standardization.

A heterogeneous soft multi-core (HSMC) architecture for embedded applications is depicted in Figure 9.2. The single FPGA circuit contains six CPU cores, a high-speed communications channel between the cores (or "network-on-chip"), core-specific external interface unit (EIU), as well as core-local memories. Moreover, the clock frequencies of individual CPU cores are optimized for the exact needs of the task executed; this leads to simple reduction of power consumption. While the dedicated processor designs are directly composed of transistors, logic gates, and standard cells, the future "soft components" in reconfigurable platforms are such high-level functions as CPU cores, inter-core communications channels, memories, and so on—the level of abstraction will step up remarkably keeping the design effort manageable. In hard real-time embedded applications, the number of required CPU cores (or parallel tasks) is typically no more than 10–15.

In mission-critical applications (such as the Mars Exploration Rover in Fig. 1.6), reconfigurable soft-core systems could even be self-repairing as discussed in Laplante (2005). The semi-automatic design and configuring process of future HSMC compositions will be discussed in Section 9.3.

9.1.2 Architectural Issues with Individual Soft Cores

After proposing a composition of multiple heterogeneous soft cores as an application-specific platform for embedded hard real-time systems, we next discuss the architectural issues related to the individual soft cores within the composition.

It is generally known that instruction queues/pipelines and instruction/data caches create challenges for the analysis of real-time systems since they make it difficult to estimate the upper and lower bounds of execution times for individual tasks. A main problem is that straightforward combinations of module-level execution time measurements—the usual manner in which performance analysis is conducted—may lead to overly pessimistic worst-case execution time estimates for the whole task (Wilhelm et al., 2009). And these problems are not only related to the difficulty of performance analysis, but both pipelines and caches cause uncertainty to response times; hence, real-time punctuality becomes deteriorated.

In hard real-time systems with heterogeneous soft multi-cores, it is possible to eliminate cache memories and have a flat memory architecture instead. This is feasible because every task runs on its own CPU core, and the clock frequency of the core does not usually need to be at the limits of the underlying FPGA hardware; the CPU–memory latency gap does not exist in such an environment. This architectural feature makes the individual soft-core blocks simpler, since no cache controller is needed. Furthermore, as every CPU core has a private memory, there is no memory-interleaving latency either in this multi-core platform.

Pipelines, superpipelines, superscalar architectural features, and out-of-order execution are all used to improve the instruction throughput of modern processors. While they do improve the average execution time of instructions, they also introduce a remarkable uncertainty to worst-case execution times. Therefore, following the recommendations of Wilhelm et al. (Wilhelm et al., 2009), we prefer a short (3–5 stages) compositional pipeline that makes the timing analysis straightforward. As they point out, with fully timing compositional architectures, the analysis can safely follow local worst-case paths only.

Finally, the soft CPU cores have RISC-like instruction sets without any speculative features, register-rich data paths, memory-mapped input/output (I/O), and Harvard architecture. And it should be adequate to have configurable soft cores with a few performance levels available for each relevant FPGA platform, such as:

- 8-bit low-performance RISC
- 16-bit medium-performance RISC
- 32-bit high-performance RISC
- 16-bit digital signal processor
- 32-bit digital signal processor

These soft-core blocks can be configured flexibly with respect to the availability of floating-point arithmetic, amount and type of memory, and the available external interface units.

9.1.3 More Advanced Fieldbus Networks and Simpler Distributed Nodes

In the early years of real-time systems, automation and control computing in large-scale plants (Kamiya, 2004) was centralized: a single computer unit collected measurements, executed elementary signal processing and control algorithms, and delivered commands for actuators, as well as references for local analog controllers. This required significant amounts of parallel wiring in industrial environments (chemical factories, paper mills, electric power plants, etc.) with high levels of electromagnetic interferences. And the wiring was particularly problematic when analog signals were transferred for lengthy distances. As we know, these problems were tackled effectively by implementing distributed control systems, which use fieldbus networks for wiring-efficient connections between individual measurement units, controllers, actuators, and monitors.

Furthermore, the distributed control approach is not just for saving wiring expenses and space, but it provides an opportunity to implement *machine intelligence* in the distributed units. Nevertheless, there is also another driver for distributing intelligence around the whole control system—to make it possible to manage hard real-time constraints in such a cooperative environment, where the shared communications network creates considerable uncertainties for node-to-node message delivery times. This is still the situation with most fieldbus systems.

In the future, the fieldbus networks will become much faster due to the increasing use of optical cabling instead of copper wires. Besides, messages will be delivered in chunks of only a few bytes with light-weight simplified protocols. This makes it possible to reduce the latencies and their uncertainties when delivering time-critical messages. With these advanced networks, it is possible to move some of the distributed intelligence "back" to the *central computing unit* that is normally handling the supervisory control tasks. The motivation behind such a move is to simplify the numerous distributed nodes and thus make their remote maintenance and updating easier.

The powerful central computing unit is further connected to a *computing cloud* (Luo, 2010) over the Internet. This cloud-computing infrastructure is relying on a regional data center that provides on-demand services for the plant supervision center, local service crews, product maintenance groups, research and development teams, operative management, and so forth. The required raw data is collected from the distributed units per need basis, but all postprocessing, fault prognosis and diagnosis, optimization of process parameters, as well as service-specific user interfaces, are provided by the servers of the computing cloud. And all this information and derived knowledge could even be accessed by smartphones using a web browser or possible augmented reality features.

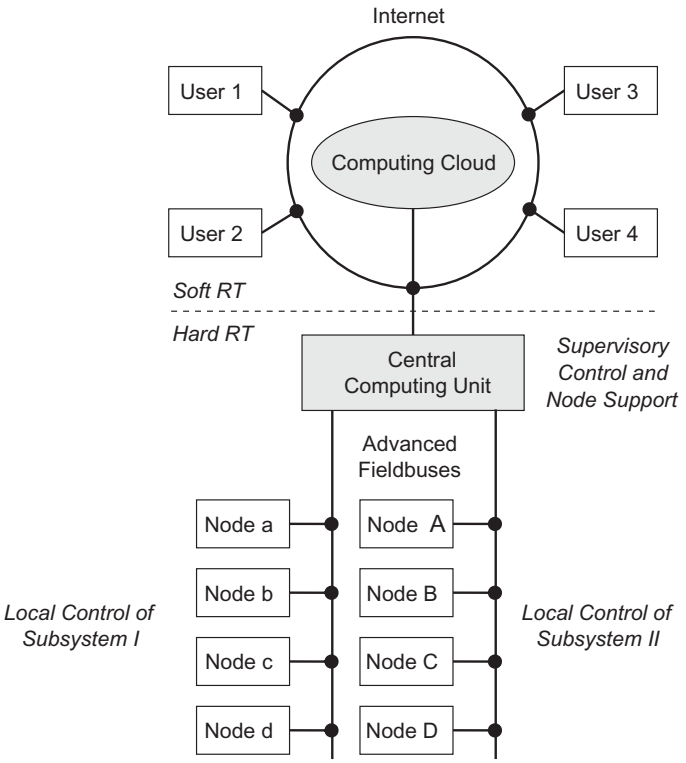


Figure 9.3. A distributed control architecture for future large-scale plants.

Figure 9.3 illustrates a distributed control architecture for large-scale plants, which is based on simple distributed nodes, advanced fieldbus networks, a central computing unit, Internet connection, and a computing cloud. This dual-level architecture is also applicable besides industrial plants. The hard real-time functions are processed strictly within the fieldbus framework, while the soft real-time services are provided by the cloud. In addition, both of these entities may handle firm real-time tasks—depending on the nature of the particular application.

As a general comment, before taking any complex control architecture in use, it is important to perform a careful sensitivity analysis to ensure the robustness of the new architecture to changes (Racu et al., 2008). Sensitivity analysis is necessary to identify how well the new architecture can accommodate updates and later modifications, for instance.

9.2 VISION: REAL-TIME OPERATING SYSTEMS

Since currently available multi-core processors are not feasible for multitasking use in hard real-time applications with strict mission or safety concerns

(Wolf et al., 2010), we proposed the straightforward architecture of heterogeneous soft multi-cores—which is highly deterministic—earlier in this chapter. Next, we will outline a plain real-time operating system for that reconfigurable environment. The service-oriented operating system is very simple, because it does not need to perform any intra-core or inter-core scheduling/dispatching, as will be seen below.

9.2.1 One Coordinating System Task and Multiple Isolated Application Tasks

The HSMC architecture provides a dedicated CPU core for every task that runs in isolation. Actually, this approach could be interpreted as a distributed multi-processor system on a single chip. Hence, the needed RTOS functionality is focused merely on reliable synchronization and intertask communication. These critical services are executed within a single system task that interacts with the application tasks through the high-speed communications channel depicted in Figure 9.2. All application tasks handle their local scheduling and dispatching themselves. In this context, scheduling/dispatching means the timely activation of application programs due to local hardware interrupts and system events related to synchronization or intertask communication. When an application program is waiting for some hardware interrupt or a specific system event, a background program (non real-time) is running and executing built-in self-tests for the hardware platform—this is a natural alternative to simple idling. Thus, every CPU core contains a foreground–background system.

Synchronization is performed using *mutex locks* or other forms of semaphores for protecting critical resources, such as shared data-acquisition channels and external communications networks. Application tasks request mutexes from the system task, which allocates them when available. If the desired mutex is not immediately available for the requesting task, the task puts itself to a wait state, and when the system task finally provides an event corresponding to the availability of the mutex, the waiting application task will be continued. And eventually, the mutex lock will be released for other application tasks. The system task handles simultaneous requests from application tasks using a priority or round-robin scheme, or some combination of them (depending on the application). Moreover, when common resources are accessed, a time-bounded handling is guaranteed by the system task; the usage time of every shared resource is limited to a particular duration. Hence, the worst-case execution time analysis is always possible. No such condition as priority inversion will ever occur, since true task concurrency is practiced with independent cores.

Intertask communication is carried out using task-local message buffers, which are "connected to each other" by the system task. That is, if Task 1 wants to send a message to Task 2, it fills its local message buffer and informs the system task that there is a message for Task 2. Next, the contents of the message buffer are transferred over the high-speed communications channel

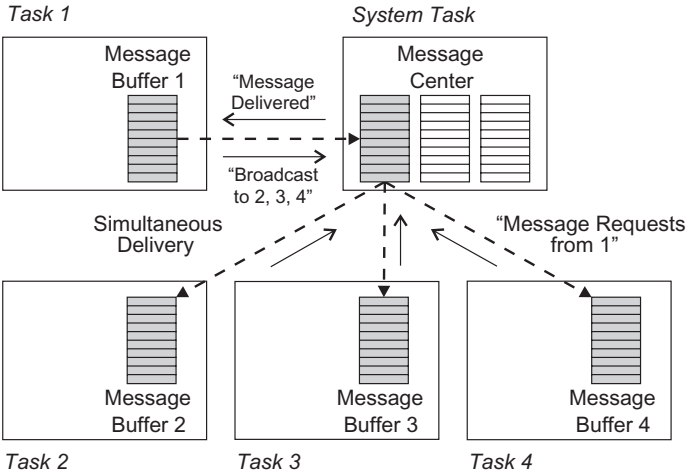


Figure 9.4. An example of coordinated message broadcasting from Task 1 to Tasks 2–4.

to the *message center* of the system task. Finally, after Task 2 has informed the system task that it is waiting for a message from Task 1, the system task delivers the message buffer to Task 2. In addition, the system task sends a "message delivered" notification to Task 1. If some message is not requested by the addressed destination task within a specified time period, an "undelivered message" warning will be sent to the source task. Besides, if an expected message is not made available by the associated source task within another specified time period, an "unavailable message" warning will be sent to the destination task. It is also possible to broadcast messages simultaneously to multiple tasks using the same principle, as illustrated in Figure 9.4.

In addition to the synchronization and intertask communication services, the system task may provide also other utility services, such as an accurate real-time clock and calendar.

This simple real-time operating system and the straightforward processor architecture together are responses to the increasing complexity of hard real-time systems. The deterministic and easily manageable computing platform makes it possible for research and development teams to concentrate more effectively on the design of applications, as will be discussed in Section 9.4.

9.2.2 Small, Platform Independent Virtual Machines

Let us leave the special HSMC architecture for a while and take a look at a more traditional approach. Recent architectural/hardware advances have produced a number of experimental small embedded processors with more processing power and memory and lower power than previous generations, and all this with a relatively small footprint. But modern virtual machine support for

these environments may still require too much memory. For example, the ROM size for Java Standard Edition for Embedded Systems is approximately 30 M bytes, and the .NET "Compact Framework" is around 5 M bytes. Nevertheless, for small embedded environments, a virtual machine with a ROM size in the 256 K-byte range would be desirable to fit into the smallest devices.

Even when a virtual machine that is small enough is developed, the architecture of a typical real-time system is not designed to manage the load of a virtual machine. Hence, a new architectural paradigm for a small microcontroller platform is needed that can support both real-time processing and a virtual machine (Davis, 2011). In addition, an appropriate microkernel architecture that can run on the virtual machine will need to be developed.

9.3 VISION: REAL-TIME PROGRAMMING LANGUAGES

Over the past few decades, specialized real-time programming languages have been introduced every now and then, but they have not obtained any mainstream role in embedded software development. The embedded systems industry is rather satisfied with the currently available procedural and object-oriented languages. So, why would the situation change in the visionable future?

There are two principal needs that cannot be fulfilled properly with the existing programming languages under the ongoing transition to multi-core platforms:

1. The need to improve the *productivity* of programmers
2. The need to continue to have *platform-independent* code

The first need is related to the general trends that the complexity of embedded systems is growing steadily, and the product development cycles are becoming shorter. And the latter one is a consequence of the emergence and evolution of diverse multi-core architectures. How could we develop software for a specific multi-core processor that would be portable to another multi-core or even a single-core environment? This is a particularly critical question if the embedded software has a long life; there is not yet architectural convergence in the multi-core processor field, and the processors that will be available by the end of this decade will be different from existing ones. Therefore, industrial companies that have taken multi-core platforms for their embedded products (so far, soft or firm real-time systems only) are currently using the multi-cores as multiple single-cores to reduce their dependency on the particular parallel architecture. Besides, by following such a conservative approach, there is not any need to have a special multi-core operating system either. Our heterogeneous soft multi-core approach that was introduced in Sections 9.1 and 9.2 can be seen as an extension of such a pragmatic principle.

When the semiconductor industry switched from making processors run faster to putting more of them on a single chip, no clear notion was given how

such devices would, in general, be programmed (Patterson, 2010); hence, to get rid of a hardware bottleneck, a programming bottleneck was essentially introduced. The programmers of multi-core processors are facing the classical parallel programming challenges, such as sequential dependencies, load balancing, memory sharing, and synchronization. In a recent Workshop of Computer Architecture Research Directions, two recognized scientists, David August and Keshav Pingali, had a moderated debate around the fundamental question "Do applications programmers need to write explicitly parallel programs?" (Arvind et al., 2010). After a lively debate, no ultimate agreement was concluded for the *explicit* versus *implicit* issue—both of the parallel programming models have their pros and cons, but the implicit approach is obviously preferable. Furthermore, as the productivity of programmers and the shortage of skilled programmers are current problems in the expanding field of embedded systems development (even without considering needs for parallel programming experts), it is practical to favor the implicit parallel programming model. In implicit parallel programming, it is the responsibility of the compiler—not the programmer—to identify and utilize the opportunities for parallelism. And that should be the way how embedded software is programmed in the multi-core era. The new architectures should not make the practitioner's programming task any harder or destroy the portability of code between different platforms.

9.3.1 The UML++ as a Future "Programming Language"

To make the programming process more efficient, it is necessary to raise the level of abstraction above the current programming languages. Thus, we propose that the traditional programming languages would "merge" with the universal modeling language (UML) and form the *UML++*. This imaginary name was selected both to distinguish the future modeling and programming language from the current UML and to emphasize that it has evolved, indeed, from the UML. The UML–UML++ relationship has an analogy to the C–C++ association. UML forms a sound basis for a "parallel programming language," since its objects are considered as parallel entities, and a single object entity exhibits itself as a concurrent activity. Such a step-up in the level of programmer's abstraction will mean that code generation has to become automatic, and, hence, programmers will shift their efforts from code writing to the design of real-time systems. And with the heterogeneous soft multi-core architecture, the outcome is the design of real-time *systems*—not just real-time software—because there is the necessity to compose and configure the application-specific processor as well. All this is done using the UML++, which needs to have strict formalism in the full subset of diagrams that are intended for specifying hard real-time systems. Thus, the UML++ Design Engine is able to map the high-level behavioral description of a real-time system semi-automatically to the multiple soft cores of the heterogeneous processor platform, which is synthesized as a set of configurable CPU cores with varying

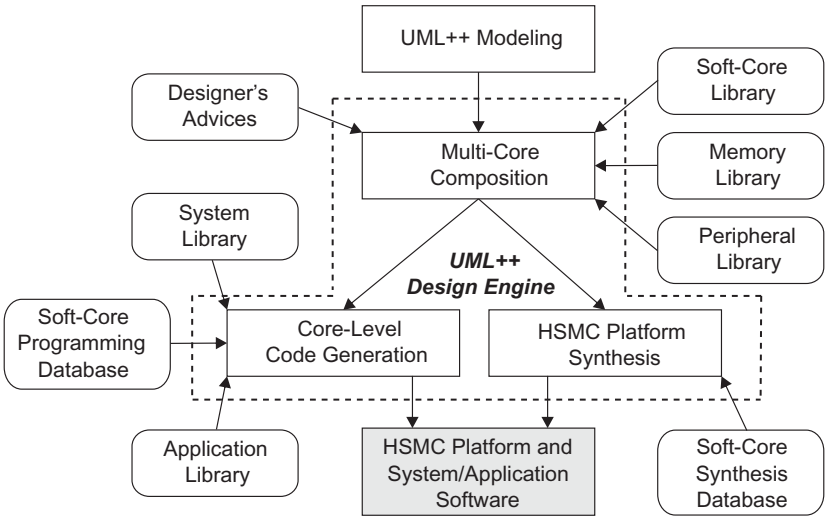


Figure 9.5. The procedure of generating an application-specific HSMC platform and the corresponding program code for the individual soft cores using the UML++ design engine.

levels of performance and functionality. In addition, the program code is generated automatically for each of the soft cores from the UML++ descriptions. This procedure is sketched in Figure 9.5.

But currently, UML 2.0 is considered overly complex and difficult to learn/use among many practitioners; would that not hinder the acceptance of UML++? Not necessarily, if the *basic* UML notation is made familiar to "everybody" already in middle and high schools and throughout their college education. This could be feasible, since the graphic notation techniques used in math, science, and engineering courses could be replaced consistently with the basic UML notation in associated textbooks (currently, diverse block and flow diagrams are used depending on the context). Spinellis called this approach "UML everywhere," and he pointed out "after a short learning period, we'll all be able to concentrate on how our diagrams can best convey our ideas, rather than on inventing new notations" (Spinellis, 2010). In this way, the effort in taking the advanced UML++ in use by future practitioners could be moderate.

To conclude, the unambiguous behavioral modeling part of UML++ would be the future "real-time programming language" (and much more) for HSMC architectures. An initial step toward that direction was taken in Arpinen et al. (2006). However, before all the required features are well established and in mainstream use within the embedded systems community, we are close to the year 2040. This lengthy adoption period is understandable when we take into account the methodology-wise conservative nature of embedded systems

industry. And while it is easy to argue that such an automatic code generation scheme is inefficient, we can safely respond that the continuing validity of Moore's law will override such considerations in our visioning period. This straightforward reasoning has an analogy to the painful but successful transitions from the assembly language to procedural languages and further from procedural languages to object-oriented ones in hard real-time systems.

9.4 VISION: REAL-TIME SYSTEMS ENGINEERING

Formal methods have always given hope that better ways of building reliable real-time systems easier, faster, and cheaper could be found. But this promise has never really been fulfilled, largely because of difficulties in modeling temporal behavior.

Currently, there are problems in using formal methods as the panacea for real-time systems design and analysis. There is no standard methodology—in fact, there seem to be too many formal methods available. There also seems to be a dearth of real success stories in building successful real-time systems exclusively using formal methods. And when these success stories are reported, they are hailed as some kind of exceptional case. If formal methods really are the answer, then reports of their successful use in real-time systems should not be news—they should be commonplace.

Software engineering pioneer David Parnas recently made this case and decried the current state of applied formal methods. "It is our job to improve these methods, not sell them. Good methods, properly explained, sell themselves. Our present methods don't sell beyond the first trial" (Parnas, 2010). Parnas goes on to argue for a different paradigm for the use of formal methods and for simplicity and more universality in notation.

In the previous section, we proposed the UML++, which is a next-generation modeling language to be used for the design and implementation of real-time software for single- and (heterogeneous) multi-core environments. The formal UML++ model of an embedded system is platform independent, since all platform dependencies are hidden inside the *UML++ Design Engine* depicted in Figure 9.5. These platform dependencies are related both to the application-specific multi-core composition, as well as to the actual FPGA circuit in use. It should be emphasized that the software design and implementation activities are highly inseparable within the UML++ framework; thus, the final UML++ model is constructed iteratively, and the incremental process is supported by high-level simulation tools for verifying the model increments as they appear. This model verification is performed semi-automatically.

9.4.1 Automatic Verification of Software

The software for the HSMC platform is generated automatically from the UML++ model (see Fig. 9.5). In parallel with such a major computation effort,

also the necessary test cases and associated test suites are generated. Furthermore, the embedded software is verified automatically at three different levels:

1. *Object Level*: Each object within a single CPU core.
2. *Core Level*: Each CPU core within the multi-core processor.
3. *System Level*: All CPU cores together.

To be able to carry out adequate tests for the levels 2 and 3 of real-time software, a versatile library of configurable simulation programs is needed for providing the essential application environment. The biggest challenges toward *fully automatic* software verification are related to these simulated application environments; it is apparent that they have to be generated semi-automatically throughout our visioning period.

9.4.2 Conservative Requirements Engineering

The role of requirements engineering must be emphasized, since it is of growing importance to develop the "right product" at the "right time" for truly global markets with tough competition and shortening life cycles. Depending on the type of the product to be developed, the requirements engineering process can either be incremental and closely integrated to the UML++ design/implementation phase or be a separate phase that is completed before the UML++ modeling begins at all. The former approach has clear similarities to agile methodologies, while the latter one is supporting a sequential life cycle model. And both of these are definitely needed, because there are various types of embedded products, as well as development environments with different characteristics to manage.

While the design/implementation is carried out entirely using the single UML++ tool, the requirements engineering activities continue to be supported by multiple tools. This is due to the diversity of stakeholders. They are not going to have a "common language" during this visioning period; the methodological gap between marketing and engineering people remains too wide.

9.4.3 Distance Collaboration in Software Projects

In the future, embedded software development projects will be distributed increasingly in multiple physical locations, which may reside even in different continents. This is due to the globalization boom that has made the large corporations and also smaller companies distribute their research and development (R&D) units. The parent organization can be located in Europe or the United States, but there may be local R&D groups in countries such as Brazil, China, and India.

To establish a solid foundation for effective cooperation between the geographically distributed groups, it is necessary to have common modeling tools

in use within the whole project team. A model-centered approach to software engineering collaboration is beneficial due to its structuring effects (Whitehead, 2007).

Today, e-mail and certain web-based applications are used routinely in most development projects as the preferred means for inter-group communication. The amount of physical travel by real-time software engineers is continuously decreasing; it is time-consuming, expensive, and environmentally unsound. Nonetheless, based on our international R&D experience, a well-prepared physical meeting of a project team cannot be fully substituted by current video-conferencing techniques. The state of video conferencing is still too primitive, and the divisive "they-and-us" arrangement disturbs natural interaction.

Whitehead presented a relevant roadmap on collaboration in software engineering (Whitehead, 2007). He indicated that the future communications and presence technologies will offer novel opportunities for fading the physical distances between cooperating engineering groups. Particularly, the advancing virtual reality environments of 3D games (and virtual environments like Second Life) could provide a reasonable virtual environment for design reviews, regular project meetings, and even for daily coffee breaks. In that way, the cohesion of the entire project team would improve and the harmful "they-and-us" thinking could diminish.

It is expected that the innovative game industry will develop such virtual- or augmented-reality environments during this visioning period that could also provide a "quantum leap" for natural and effective interaction between distributed software engineering groups.

9.4.4 Drag-and-Drop Systems

Components for building real-time systems abound in open-source repositories, and many of these are quite robust and are currently being used in industrial-strength real-time systems. But for the most critical applications, new engineering paradigms are needed to allow for easy identification, validation, verification and assembly of these components from a number of sources (e.g., open source, commercial off-the-shelf, and in-house reuse).

We envision a new generation of drag-and-drop component-based software engineering tools and associated capability to support round-tripping engineering (forward and reverse engineering from specification to code). Progenitors of these kinds of systems abound, but none has the kind of robustness and provable correctness that is needed for the most critical applications.

9.5 VISION: REAL-TIME APPLICATIONS

Since the beginning of the embedded systems era, numerous embedded applications have been introduced; many of them have survived over years and

have an established position in our lives, while others have quietly disappeared. The introduction of new embedded applications will undoubtedly continue during the visioning period—and beyond that. In this section, we discuss some future embedded applications that could have a considerable impact in our personal lives and the society around us.

Large numbers of small, communicating real-time computers found in smartphones, wearable clothing, appliances, vending machines, and so forth promise to transform the way human beings interact with their environment. We are only moments away from the following scenarios:

- Embedded microcontrollers in your clothes communicate with the washing machine in order to set up the appropriate wash cycles.
- Sensors reading perspiration and body temperature adjust the environment in your home or car.
- Your refrigerator and smart pantry can take inventory of the contents and prepare a shopping list for you based on your dietary preferences, recently prepared meals, and upcoming holidays.

All of the envisioned advances of Sections 9.1–9.4 will enable new families of applications that promise amazing functionality. Understandably, we can discuss only a small sample of these possibilities.

9.5.1 Local Networks of Collaborating Real-Time Systems

Local communications networks involving collaborating real-time systems are needed for advanced smart homes and smart buildings. In addition to numerous conveniences (e.g., environmental control), there are many safety and security applications for the elderly, disabled, or very young in a smart home. For example, such a system could track if a resident has wandered outside of a safe zone or has been immobile for too long. Connecting the system with vital sign and other status monitoring equipment can provide important information for tracking and maintaining the health and wellness of any inhabitant of a private home or public facility, such as a hospital, school, or retirement facility.

There is a rich variety of entertainment and comfort applications, too. For instance, virtual wall art and music and climate control that adapt to the tastes and desires of the most proximate individual in the room of our home. As a matter of fact, Bill Gates envisioned these applications of RFID technology in his own home and then realized them (Gates, 1995). These same adaptations could next be implemented in various public spaces, such as schools, libraries, and hospitals. And the same advancements that will be found in smart homes can scale up to smart buildings that interact with internal components (e.g., elevators, HVAC, and lighting), users, and the environment (Snoonian, 2003).

Robots present the ultimate challenge for real-time systems engineers because they are a blend of image processing, artificial intelligence, and elec-

tromechanical systems all communicating over a fieldbus network. But we can expect to see more, and robot-based applications in industrial and residential settings with a likelihood of realistic, humanoid robots becoming commonplace by 2040. Even today, many homes worldwide have an autonomous robotic vacuum cleaner, which is able to navigate a living space while vacuuming the floor and rugs.

9.5.2 Wide Networks of Collaborating Real-Time Systems

Perhaps the most exciting applications for real-time systems involve large numbers of collaborating systems over a wide area. Typical applications involve intelligent transportation systems, the smart grid, and coordinated infrastructure systems.

Intelligent transportation systems are those in which individual vehicles interact with each other and traffic monitoring and controlling equipment, emergency vehicles, pedestrians, and other real-time components (Ma et al., 2009).

A smart grid, on the other hand, is "an automated, widely distributed energy delivery network [that is] characterized by a two-way flow of electricity and information and will be capable of monitoring everything from power plants to customer preferences to individual appliances. It incorporates into the grid the benefits of distributed computing and communications to deliver real-time information and enable the near-instantaneous balance of supply and demand at the device level" (U.S. Department of Energy, Office of Electricity Delivery and Energy Reliability, 2008).

Furthermore, secure systems for infrastructure, such as power grid, water processing, telecommunications systems, and so forth, interact to identify complex threat vectors, such as *cyberpandemics*. A cyberpandemic is a "massive disruption of computing services that can trigger second- and third-order failures or malfunctions in computing and non-computing systems worldwide" (Laplante et al., 2009).

All of these systems currently exist in rudimentary forms, but fully capable, robust, and fault-tolerant solutions will be available within the next two decades. All kinds of interesting problems are presented by these systems for real-time engineers. However, many of these problems are scaled up versions of the kinds of challenges that we have studied throughout this text.

9.5.3 Biometric Identification Device with Remote Access

Another interesting class of applications is based on a novel biometric identification (Ricanek et al., 2010) device with remote access (BIDRA). BIDRA is an intelligent device that virtually everybody could own and have continually available, or maybe it will be integrated with the future smartphone. It is able to perform reliable and robust biometric identification of its genuine owner, and the identification procedure is repeated aperiodically to make sure

that the person who holds the BIDRA device is really its owner and nobody else. Moreover, BIDRA can interact securely with its near environment as well as be accessed over the Internet using a wireless communications interface.

These principal features make BIDRA a solution to multiple problems, which are somehow related to secure identification of individuals. Today, we all have multiple high-strength passwords that make it possible for us to use various systems and services with different levels of privacy and security. But anybody could use those systems/services with our identity if he just had our password. And, unfortunately, this scenario is happening all the time, because there is always a possibility of gaining access to somebody else's password by illegal or at least questionable means. BIDRA could conveniently be used in place of all passwords, and the *advanced biometric identification technique* would guarantee a high level of personal security. So, the annoying problem of passwords becomes solved.

Furthermore, BIDRA could be used for configuring different environments automatically to match the personal preferences of the corresponding individual. For example, when the individual enters his automobile that may also be used by other family members, the automobile recognizes him sitting behind the steering wheel and adjusts the steering wheel, driver's seat, mirrors, interior temperature, and preferred radio channels according to the predefined preferences obtained from BIDRA. BIDRA would also serve as a general key to multiple electronic locks (home, office, automobile, etc.)—no other keys are needed anymore.

In addition, BIDRA could be used for effortless automobile parking in a public parking garage. A remote BIDRA reader recognizes that an automobile that is driven by Person X is entering the parking hall and records the arrival time. Later on, when the automobile driven by the same individual exits the garage, the remote reader records the departure time. No on-site payment is needed in such a default case, but the payment would be charged directly from the bank account, which number is provided by BIDRA.

Automatic overspeed detection is used increasingly on highways and streets in many countries. There is typically an inductive sensor under the road surface that is used for measuring the speed of passing vehicles, and a rugged camera that takes a digital image of every overspeeding vehicle. Eventually, some manual processing is needed before a penalty invoice is mailed to the owner of the vehicle. BIDRA could greatly simplify the process of overspeed "ticketing." In place of those expensive roadside cameras, only a simple BIDRA reader with wireless Internet access is needed. When a vehicle passes such a reader and an overspeed is detected, the penalty invoice can automatically be e-mailed to the address provided by the BIDRA of the driver, or even charged directly from the driver's bank account. In some countries, certain fines are dependent on the driver's annual income, which is also known by BIDRA. Besides, no vehicle can be taken in use without a valid driver's license, and this information is stored in BIDRA, as well.

This free visioning process could be continued further, but the examples already presented show the huge potential that BIDRA would offer. But what would happen if a BIDRA device were lost? No worry, every BIDRA contains a satellite navigation receiver that is used to determine BIDRA's location with a high accuracy. And it is possible to access a lost BIDRA device through a wireless Internet connection to find out the precise coordinates where it currently resides. In principle, it would also be possible to track every BIDRA device all the time by some authorized person or organization—if just allowed.

9.5.4 Are There Any Threats behind High-Speed Wireless Communications?

The term "wireless" was mentioned three times in the short BIDRA introduction, and it appears that our *digitized society* is going toward faster and faster wireless networks. "Wireless is handy and wireless is everywhere," but there may be serious threats behind such thinking.

Since the beginning of the cell phone era, there has been a debate whether the transmitted high frequencies or microwaves can cause brain cancer. Nonetheless, there is no scientific proof that cell phones would be dangerous for their users' health. On the other hand, there is ongoing discussion on the possible effects of cell phones and base stations of wireless communications systems to humans' hormonal excretion. Disturbances in specific hormonal excretion can lead, for instance, to depression or alcoholism. Moreover, there are at least suspicions that high-speed wireless communications systems can disrupt the navigation abilities of certain insects, such as honeybees.

Persistent long-term research is needed to confirm or reject those and other similar hypotheses related to the biological effects of wireless communications (Valberg et al., 2007). In parallel with such research efforts, however, the used microwave frequencies are continually increasing as higher data rates are achieved for wireless Internet use.

If the biological threats turn out to be true, that would have a significant influence to the future real-time systems, too. Another, purely technical concern is that the public frequency bands (which require no explicit operator licenses) will become overly crowded due to the excessive use of wireless sensor networks in future embedded applications.

9.6 SUMMARY

Future Visions on Real-Time Systems is a truly challenging topic that could be presented in numerous different ways, depending on the chosen viewpoint. Instead of aiming to provide a comprehensive treatment of this wide topic, we decided to focus on a carefully selected sample of specific subjects that could form an educational basis for class discussions and related assignments at the end of a practical course on real-time systems.

The continuing development of hardware technologies maintains a solid basis for the advancement of real-time systems. Large and reconfigurable (but cost-effective) FPGA platforms, together with the proposed heterogeneous soft multi-core architecture, make it possible to implement hard real-time systems with true task concurrency. The possibility of assigning an individual CPU core for each software task makes the overall software structure straightforward and hence easier to maintain. And this is important, since the size of applications software is going to grow considerably in future embedded systems.

Moreover, the role of real-time operating systems is somewhat different in HSMC environments than in traditional multitasking, because every software task runs in isolation on its private CPU core. No centralized scheduling/dispatching is needed, but the RTOS is mainly providing punctual synchronization and intertask communication services for cooperating tasks. This deterministic scheme is much simpler than those multi-core operating systems that handle both intra-core and inter-core scheduling/dispatching, as well as dynamic load balancing.

Furthermore, the programming of HSMC processors is carried out at a higher level of abstraction than provided by the existing procedural and object-oriented languages. We call the new "real-time programming language" UML++. However, the UML++ is not just a programming language, but an evolved entity of the present UML with enhanced levels of formality and real-time support. The incrementally designed and verified behavioral models are used by the UML++ Design Engine both for automatic composition of the application-specific HSMC platform and for automatic code generation.

It is no more practical to test the future applications software manually, but automatic test case and test suite generation is followed by automatic test execution. With the HSMC platform, the embedded software is tested automatically at three hierarchical levels: object level, core level, and system level.

Effective collaboration between software engineers is going to be an increasingly important issue in embedded software development. This is due to the growing complexity of real-time software and the globalization of software development activities. To make the inter-group collaboration more natural and efficient, it could be possible to use the future virtual- and augmented-reality environments of 3D games as collaboration environments between geographically distributed software engineering groups.

One of the principal problems of the digitalizing society is how to identify the users of various systems/services reliably but conveniently. Currently employed password schemes are approaching the end of their utility for obvious reasons. Hence, we proposed the biometric identification device with remote access to be used in virtually all future applications where secure identification of individuals is needed. It uses a robust biometric technique for identifying the genuine owner of the BIDRA. A few potential applications of the BIDRA were envisioned and discussed. It is likely that such an identification device with flexible wireless communications and self-locating abilities will become available before the end of our visioning period.

Future applications of real-time systems are exciting, but most of the challenges facing the realization of these systems are the same problems that have faced real-time systems engineers for more than 50 years. While many solutions to old problems can be scaled up, new complexities are introduced as the capabilities of systems improve over time. Besides, applications development will create new theoretical problems to be solved and require new and better software engineering techniques so that the applications can be effectively realized.

A new edition of a textbook is like a new spring in nature.

9.7 EXERCISES

9.1. Find at least three advantages and three disadvantages of the HSMC architecture (see Fig. 9.2). Based on those findings, evaluate your confidence (scale 0–1) on this particular hardware vision. By which decade, if ever, would you think that such an approach would be practical and in wide use?

Class assignment: Compare the answers of individual students and create a collective answer for the class.

9.2. What are the benefits, if any, that a computing cloud (see Fig. 9.3) could offer for a distributed control system of large-scale plants compared with traditional solutions to similar requirements?

Class discussion: Is it feasible to combine Internet-based cloud computing with such a distributed control system having hard and firm real-time constraints?

9.3. How could the physical communications channel, used in Figure 9.4 for transferring messages between different CPU cores, be implemented in practice? What are the pros and cons of those alternative implementations?

Class discussion: Could the physical communications channel become a bottleneck for the synchronization and intertask communication services offered by the system task?

9.4. What are the main challenges—or even obstacles—behind the suggested design and implementation procedure of Figure 9.5? Based on the identified challenges, evaluate your confidence (scale 0–1) on that particular vision.

Class assignment: Compare the answers of individual students and create jointly a single collective answer of the whole class.

9.5. The proposed BIDRA device seems to offer a vast variety of application opportunities. List five possible applications of BIDRA that are not

mentioned in the text. Which practical issues could hinder the development or wide spreading of BIDRA?

Class debate: The instructor moderates a prepared debate between two teams. First, the class is divided to three teams; one of the teams is favoring the BIDRA concept, one is against it, and one team is evaluating the presented arguments. What are the objective conclusions of the evaluation team?

9.6. Based on your personal vision confidence pentacle (see Fig. 9.1), identify the strongest and weakest vision element (or dimension) expressed in this chapter. Take the possible multiple visions within a single vision element as a whole when drawing the pentacle.

Class assignment: Compare the pentacles of individual students and create jointly a single collective pentacle of the entire class.

9.7. What are the three most important visions on real-time systems that are missing from this chapter? Explain why those visions are of particular importance.

Class discussion: Evaluate the additional visions of individual students and create the class' top 3 of additional visions (possibly by voting).

9.8. Virtual machines, such as the Java Virtual Machine (JVM), can make the applications software easily portable to diverse hardware platforms. Would it be feasible to consider a similar virtual machine model also for the HSMC architecture (see Figs. 9.2 and 9.5), which is intended for hard real-time systems? What advantages and disadvantages would such a scheme offer in this case?

Class discussion: What are the main challenges in developing and implementing small and temporally predictable virtual machines for embedded applications?

REFERENCES

T. Arpinen, P. Kukkala, E. Salminen, M. Hännikäinen, and T. D. Hämäläinen, "Configurable multiprocessor platform with RTOS for distributed execution of UML 2.0 designed applications," *Proceedings of the Design, Automation and Test in Europe*, Munich, Germany, 2006, vol. 1.

D. A. Arvind, K. Pingali, D. Chiou, R. Sendag, and J. J. Yi, "Programming multicores: Do applications programmers need to write explicitly parallel programs?" *IEEE Micro*, 30(3), pp. 19–33, 2010.

R. Davis, Evaluating virtual machines for use on a small embedded real time microcontroller platform. Doctoral Dissertation, Colorado Technical University, 2011.

A. Elkateeb and A. Mandepanda, "Embedded soft processor for sensor networks," *Proceedings of the International Conference on Network-Based Information Systems*, Indianapolis, IN, 2009, pp. 268–272.

M. Funabashi, T. Kawabe, and A. Nagashima, "Towards post embedded systems era," *Proceedings of the ICROS-SICE International Joint Conference*, Fukuoka, Japan, 2009, pp. 466–469.

B. Gates, *The Road Ahead*. New York: Penguin Books, 1995.

J. Gea-Banacloche and L. B. Kish, "Future directions in electronic computing and information processing," *Proceedings of the IEEE*, 93(10), pp. 1858–1863, 2005.

M. Hashemi and S. Ghiasi, "Versatile task assignment for heterogeneous soft dual-processor platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3), pp. 414–425, 2010.

A. Kamiya, "General model for large-scale plant application," in *Computationally Intelligent Hybrid Systems: The Fusion of Soft Computing and Hard Computing*, S. J. Ovaska (Ed.), Hoboken, NJ: Wiley-Interscience, 2004, pp. 35–55.

D. Lammers, "The era of error-tolerant computing," *IEEE Spectrum*, 47(11), p. 15, 2010.

P. A. Laplante, "Computing requirements for self-repairing space systems," *Journal of Aerospace Computing, Information, and Communication*, 2(3), pp. 154–169, 2005.

P. Laplante, B. Michael, and J. Voas, "Cyberpandemics: History, inevitability, response," *IEEE Security and Privacy*, 7(1), pp. 63–67, 2009.

M. Levy and T. M. Conte, "Embedded multicore processors and systems," *IEEE Micro*, 29(3), pp. 7–9, 2009.

Y. Luo, "Network I/O virtualization for cloud computing," *IT Professional*, 12(5), pp. 36–41, 2010.

Y. Ma, M. Chowdhury, A. Sadek, and M. Jeihani, "Real-time highway traffic condition assessment framework using vehicle–infrastructure integration (VII) with artificial intelligence (AI)," *IEEE Transactions on Intelligent Transportation Systems*, 10(4), pp. 615–627, 2009.

S. Makridakis, S. C. Wheelwright, and R. J. Hyndman, *Forecasting: Methods and Applications*, 3rd Edition. New York: John Wiley & Sons, 1998.

D. L. Parnas, "Really rethinking 'formal methods'," *IEEE Computer*, 43(1), pp. 28–34, 2010.

D. Patterson, "The trouble with multi-core," *IEEE Spectrum*, 47(7), pp. 28–32, and 53, 2010.

J. R. Powell, "The quantum limit to Moore's law," *Proceedings of the IEEE*, 96(8), pp. 1247–1248, 2008.

R. Racu, A. Hamann, and R. Ernst, "Sensitivity analysis of complex embedded real-time systems," *Real-Time Systems*, 39(1), pp. 31–72, 2008.

K. Ricanek, M. Savvides, D. L. Woodard, and G. Dozier, "Unconstrained biometric identification: Emerging technologies," *IEEE Computer*, 43(2), pp. 56–62, 2010.

B. Sick and S. J. Ovaska, "Fusion of soft and hard computing: Multi-dimensional categorization of computationally intelligent hybrid systems," *Neural Computing and Applications*, 16(2), pp. 125–137, 2007.

D. Snoonian, "Smart buildings," *IEEE Spectrum*, 40(8), pp. 18–23, 2003.

D. Spinellis, "UML everywhere," *IEEE Software*, 27(5), pp. 90–91, 2010.

R. Talwar and I. Pearson, "The world in 2020 [General Forecasts]," *Engineering and Technology*, 5(1), pp. 21–24, 2010.

U.S. Department of Energy, Office of Electricity Delivery and Energy Reliability, "The smart grid: An introduction," 2008. Available at http://energy.gov/sites/prod/files/oeprod/DocumentsandMedia/DOE_SG_Book_Single_Pages%281%29.pdf, last accessed August 17, 2011.

P. A. Valberg, T. E. van Deventer, and M. H. Repacholi, "Workgroup report: Base stations and wireless networks—radiofrequency (RF) exposures and health consequences," *Environmental Health Perspectives*, 115(3), pp. 416–424, 2007.

J. Whitehead, "Collaboration in software engineering: A roadmap," *Proceedings of the Future of Software Engineering*, Minneapolis, MN, 2007, pp. 214–225.

R. Wilhelm et al., "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7), pp. 966–978, 2009.

J. Wolf et al., "RTOS support for parallel execution of hard real-time applications on the MERASA multi-core processor," *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Carmona, Spain, 2010, pp. 193–201.

GLOSSARY

Many of these terms have been adapted from P. A. Laplante (Editor-in-Chief), *The Dictionary of Computer Science*, *Engineering*, *and Technology*. Boca Raton, FL: CRC Press, 2001.

Abstract class: A superclass that has no direct instances.

Abstract data type: A programming language construct where a user defines his own data type along with the requisite operation that can be applied to it.

Accept operation: Operation on a mailbox that is similar to the pend operation, except that if no data are available, the task returns immediately from the call with a condition code rather than suspending.

Access time: The interval between when data are requested from the memory cell and when they are actually available on the bus (read operation). Analogously applicable with the write operation.

Accumulator: A special-purpose register used with arithmetic and logic instructions in certain processor architectures.

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

Actual parameter: The named variable passed to a procedure or subroutine.

Adaptive programming: A lightweight programming methodology that offers a series of frameworks to apply adaptive principles and encourage collaboration to obtain easily evolvable code.

Address bus: The collection of parallel wires needed to address individual memory cells.

Agile programming: A lightweight programming methodology that is divided into four principal activities—planning, designing, coding, and testing—all performed iteratively.

Algorithm: A systematic and precise, step-by-step procedure for solving a certain problem or accomplishing a task, for instance, converting a particular kind of input data to a particular kind of output data. An algorithm can be executed by a computer.

Alpha testing: A type of validation consisting of internal distribution and exercise of the developed software.

ALU: *See* arithmetic logic unit.

Analog-to-digital conversion: The process of sampling and converting analog (continuous amplitude and time) signals into digital (discrete amplitude and time) ones.

Anonymous variable: A hidden variable created by the compiler to facilitate call-by-value parameter passing.

Application program: Program to perform tasks and solve problems related to some specific application.

Argument: An address or data that is passed to a procedure or function call as a typical way of communicating across procedure/function boundaries.

Arithmetic Logic Unit (ALU): CPU's internal unit that performs arithmetic and logic operations.

Arithmetic operation: Anyone of the following operations: addition, subtraction, multiplication, and division.

Artifact: Any by-product of the software development process including program code and documentation.

Assembler: A computer program that translates an assembly code text file to an object file suitable for linking.

Assembly code: A program written in assembly language.

Assembly language: The set of symbolic (mnemonic) equivalents to the machine language instruction set.

Asynchronous event: An event that is not synchronous to the applied clock signal.

Atomic instruction: An instruction that cannot be interrupted.

Attribute: A named property of a class that describes a value held by each object of the class.

Attribute multiplicity: The possible number of values for an object–attribute combination.

Background: Noninterrupt-driven tasks in foreground/background systems.

BAM: *See* binary angular measure.

Bathtub curve: A graph describing the phenomenon that in hardware components most errors occur either very early or very late in the lifecycle of the component. Similar thinking is applicable to software as well.

Benchmark: Standard tests that are used to compare the performance of computers, processors, circuits, or algorithms.

Beta testing: A type of system test where preliminary versions of validated software are distributed to friendly customers, who test the new software under actual use.

Binary Angular Measure (BAM): An n-bit scaled number where the least significant bit is $2^{-(n-1)} \cdot 180°$.

Binary semaphore: A semaphore that can take on one of two possible values.

BITS: *See* built-in-test software.

Black-box testing: A testing methodology where only the inputs and outputs of the software unit are considered. How the outputs are generated inside the unit is ignored.

Blocked: The condition experienced by tasks that are waiting for the occurrence of some event.

Branch instruction: An instruction used to modify the instruction execution sequence of the CPU. The transfer of control to another sequence of instructions may be unconditional or conditional based on the result of a previous instruction.

Branch prediction: A mechanism in advanced CPUs used to predict the outcome of conditional branch instructions prior to their execution.

Breakpoint: An instruction address at which a debugger is instructed to suspend the execution of a program. Or a critical point in a program, at which execution can be conditionally stopped to allow examination if the program variables contain the correct values and/or other manipulation of data.

Breakpoint instruction: A debugging instruction provided through hardware support in most processors. When a program hits a breakpoint, specified actions occur that save the state of the program, and then switch to another program that allows the user to examine the stored state.

Broadcast communication: In statecharts, a technique that allows for transitions to occur in more than one orthogonal system simultaneously.

Buffer: A temporary data storage area used to interface between, for example, a fast device and a slower task servicing that device.

Built-in-Test Software (BITS): Special software used to perform self-testing. Online BITS assures testing concurrently with normal operation, while offline BITS suspends normal operation.

Burn-in testing: Testing technique that seeks to flush out those failures that appear early in the life cycle of the part and thus improve the reliability of the delivered product.

Burst period: The time over which data are being passed into a buffer.

Bus: The set of parallel wires that connect the CPU and main memory. The bus is used to transfer memory addresses and exchange data between the CPU and main memory in binary-encoded form. The width of the bus is determined by the number of bits or wires provided for the binary code. Usually, the address and data wires are referred to as the address bus and data bus, respectively.

Bus arbitration: The process of ensuring that only one device at a time can place data on the bus.

Bus contention: Condition in which two or more devices attempt to gain control of the bus simultaneously.

Bus cycle: A complete memory read or write operation; from addressing to successful data delivery.

Bus grant: A signal provided by the DMA controller to a device, indicating that is has exclusive rights to the bus.

Bus timeout: A condition whereby a device making a DMA request does not receive a bus grant before some specified time.

Busy wait: In polled-loop systems, the process of testing the flag without success.

Cache: *See* memory caching.

Cache hit ratio: The percentage of memory accesses in which a requested instruction or data are actually in the cache memory.

Call-by-address: *See* call-by-reference.

Call-by-reference: Parameter passing mechanism in which the address of the parameter is passed by the calling routine to the called procedure so that it can be altered there. *Also* known as call-by-address.

Call-by-value: Parameter passing mechanism in which the value of the actual parameter in the subroutine or function call is copied into the procedure's formal parameter.

Calling tree: *See* structure chart.

CAN: A fieldbus network used widely in automotive and machine automation applications.

Capability: An object that contains both a pointer to another object and a set of access permissions that specify the modes of access permitted to the associated object from a process that holds the capability.

CASE: Computer-aided software engineering.

Catastrophic error: An error that renders the system useless or has severe consequences.

Central Processing Unit (CPU): In a computer, it provides fetching, decoding, and executing machine-language instructions, reading operands from the main memory, and writing results to the main memory.

CFD: *See* control flow diagram.

Chain reaction: In statecharts, a group of sequential events where the nth event is triggered by the $(n - 1)$th event.

Checkpoint: The instant in the history of execution at which a consistent version of the system's state is saved so that if a later event causes potential difficulties, the system can be restarted from the state that had been saved at the checkpoint.

Checksum: A value used to determine if a block of data has changed. The checksum is formed by adding all of the data values in the block together. After that, the checksum is usually inserted to the end of the data block.

Circular queue: *See* ring buffer.

CISC: *See* complex instruction set computer.

Class: A group of objects with similar attributes, behavior, and relationships to other objects.

Class definitions: Object declarations along with the methods associated with them.

Clear-box testing: *See* white-box testing.

COCOMO: A constructive cost model developed by Boehm, which is one of the most widely used resource estimation tools.

Code inspection: *See* group walkthrough.

Coding: The process of programming, generating program code in a specific language. Or the process of translating data from some representation form into a different one by using a set of rules or tables.

Collision: Condition in which one device already has control of the bus when another obtains access. Also, simultaneous use of a critical resource.

Compaction: The process of compressing fragmented memory so that it is no longer fragmented but continuous.

Compiler: A program that translates a high-level language program into an executable machine-language program or other lower-level form, such as assembly language.

Complex Instruction Set Computer (CISC): Processor architecture characterized by a large, microcoded instruction set with numerous addressing modes.

Computational intelligence: *See* soft computing.

Compute-bound: Computations in which the number of operations is large in comparison with the number of executed I/O instructions.

Computer simulation: Execution of computer programs that allows one to model the important aspects of the behavior of the specific system under study.

Concrete class: A class that can have direct instances.

Condition code register: CPU's internal register used to implement a conditional transfer, such as a conditional branch instruction.

Conditional instruction: An instruction that performs its function only if a certain condition is met.

Conditional transfer: A change of the program counter based on the result of a test.

Configuration: Operation in which a set of parameters is imposed for defining the operating conditions or mode.

Constant folding: A code optimization technique that involves precomputing constants at compile time.

Context: The minimum information that is needed in order to save a currently executing task so that it can be resumed.

Context switching: The process of saving and restoring sufficient information for a real-time task so that it can be resumed after being interrupted.

Contiguous file allocation: The process of forcing all allocated file sectors to follow one another on the hard disk or other mass memory.

Continuous random variable: A random variable with a continuous sample space.

Control Flow Diagram (CFD): A real-time extension to data flow diagrams that shows the flow of control signals through the system.

Control specifications: In data flow diagrams, a finite state machine in diagrammatic or tabular representation.

Control Unit (CU): CPU's internal device that synchronizes the entire fetch–execute cycle.

Cooperative multitasking system: A scheme in which two or more tasks are divided into states or phases, determined by a finite state machine. Calls to a central dispatcher are made after each phase is complete.

Coprocessor: A second specialized CPU used to extend the machine language instruction set of the main CPU. For instance, a floating-point coprocessor.

Coroutine system: *See* cooperative multitasking system.

Correctness: A property in which the software does not deviate from the requirements specification. Sometimes used synonymously with reliability, but correctness requires a stricter adherence to the requirements.

Correlated data: *See* time-relative data.

Counting semaphore: A semaphore than can take on two or more values. *Also* called a general semaphore.

CPU: *See* central processing unit.

CPU utilization: A measure of the percentage of nonidle processing.

CRC: *See* cyclic redundancy code.

Critical region: Code segment that interacts with a serially reusable resource.

Crystal: A lightweight programming methodology that empowers the development team to define the development process and refine it in subsequent iterations until it is stable.

CU: *See* control unit.

Cycle stealing: A situation in which an ongoing DMA access precludes the CPU from accessing the bus.

Cyclic Redundancy Code (CRC): A mathematical method for checking the correctness of memory contents or received data, which is superior to a simple checksum.

Cycling: The process whereby all tasks are being appropriately scheduled, although no actual processing is occurring.

Cyclomatic complexity: A measure of a software complexity devised by McCabe.

Daemon: A device server that does not run explicitly, but rather lies dormant waiting for specific condition(s) to occur.

Dangerous allocation: Any memory allocation that can preclude system determinism.

Data bus: Bus used to carry data between the various components in the computer system.

Data dependency: The normal situation in which the data that an instruction uses or produces depends upon the data used or produced by other instructions such that the instructions must be executed in a specific order to obtain the correct results.

Data Flow Diagram (DFD): A structured analysis tool for modeling software systems.

Data structure: A particular way of organizing a group of data, usually optimized for efficient storage, fast search, fast retrieval, and/or fast modification.

Data-oriented methodology: An application development methodology that considers data as the main focus of activities.

Dead code: *See* unreachable code.

Deadlock: A catastrophic situation that can arise when tasks are competing for the same set of two or more serially reusable resources. *Also* called a deadly embrace.

Deadly embrace: *See* deadlock.

Death spiral: Stack overflow caused by repeated spurious interrupts.

Debug: To find and remove errors from hardware or software.

Debug port: The facility to switch the processor from run mode into probe mode to access its debug and general registers.

Debugger: A program that allows interactive analysis of a running program by allowing the user to pause execution of the running program and examine its variables and path of execution at any point. Or a program that aids in debugging.

Debugging: Locating and correcting errors in a hardware circuit or a computer program. Or determining the exact nature and location of a program error and fixing the error.

Decode: The process of isolating the opcode field of a machine language instruction and determining the corresponding address in CPU's micromemory.

Default: The value or status that is assumed unless otherwise specified.

Defect: The preferred term for an error in a requirement, design, or code. *See* also fault and failure.

Demand page system: Technique where program segments are permitted to be loaded in noncontiguous memory, as they are requested in fixed-size chunks.

Density: In computer memory, the number of bits per unit area.

Dependability: System feature that combines such concepts as reliability, safety, maintainability, performance, and testability.

De-referencing: The process in which the actual locations of the parameters that are passed using call-by-value are determined.

Deterministic system: A system where for each possible state and each set of inputs, a unique set of outputs and next state of the system can always be determined.

Digital Signal Processor (DSP): An application-specific processor that is tailored for the specific needs of digital signal processing algorithms.

Digital-to-analog conversion: The process of converting digital (discrete amplitude and time) signals into analog (continuous amplitude and time) ones.

Direct Memory Access (DMA): A scheme in which access to the computer's memory bus is afforded temporarily to other devices in the system without the intervention of the CPU.

Direct mode instruction: Instruction in which the operand is the data contained at the address specified in the address field of the instruction.

Disassembler: A computer program that can take an executable image (a stream of machine-language instructions) and convert it back into assembly code.

Discrete signals: Logic lines used to control devices.

Discriminator: An enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization.

Dispatcher: The part of the real-time kernel that performs the necessary bookkeeping to start a task.

Distributed computing: An environment in which multiple computers are networked together, and the resources from more than one computer are available to a user or application.

Distributed real-time systems: A collection of interconnected, self-contained processors.

DMA: *See* direct memory access.

DMA controller: Device that performs bus arbitration.

Dormant state: In the task-control block (TCB) model, the state of a task that is unavailable to the operating system.

Double buffering: A technique using two swappable buffers where one is filled while the data in the other is being used.

Double-indirect mode: A memory addressing scheme similar to indirect mode, but with another level of indirection.

DRAM: Dynamic random-access memory. *See* also dynamic memory.

DSI: Delivered source instructions. *See* KLOC.

DSP: Digital signal processing. *Also* digital signal processor.

Dynamic memory: Random-access memory that uses a capacitor to store logic ones and zeros, and that must be refreshed periodically to restore the charge lost due to capacitive discharge.

Dynamic priority system: A multitasking system in which the priorities to tasks can change. Contrast with fixed priority system.

Dynamic Systems Development Method (DSDM): A lightweight programming methodology conceived as a methodology for rapid application development. DSDM relies on a set of principles that include empowered teams, frequent deliverables, incremental development, and integrated testing.

Effort: One of Halstead's metrics (see Chapter 8).

Embedded software: Real-time software that is part of an embedded system. Embedded software integrates an operating system with specific drivers and application software.

Embedded system: A real-time computing machine contained in a device whose purpose is not to be a computer. For example, the computers in automobiles and household appliances are all embedded computers.

Emulator: The firmware that simulates a given machine architecture. Also a device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

Encapsulation: Property of a program that describes the complete integration of data with a legal process relating to the data.

Entity Relationship Diagram (ERD): A diagram that describes the important entities in a system and the ways in which they are interrelated.

Enumeration: A list of permitted values.

Environment: A set of objects outside the system whose attributes affect and is affected by the behavior of the system.

Event: Any occurrence that results in a change in the state of a real-time system.

Event determinism: When the next states and outputs of the system are known for each set of inputs that trigger events.

Event flag: Synchronization mechanism provided by certain programming languages.

Exception: An error or other special condition that arises during program execution.

Exception handler: Code used to process exceptions.

Execute: Process of sequencing through the steps in micromemory or hard-wired state machine corresponding to a particular machine language instruction.

Executing state: In the task control block model, the state of a task that is currently running.

Executive: *See* kernel.

External fragmentation: When main memory becomes checkered with unused but available partitions.

eXtreme Programming (XP): A lightweight programming methodology based on 12 practices including pair programming, test first coding, having the customer on site, and frequent refactoring. eXtreme programming is, perhaps, the most prescriptive of the lightweight (agile) methodologies.

Failed system: A system that cannot satisfy one or more of the requirements listed in the formal system specification.

Failure: Manifestation of an error at system level. It relates to execution of wrong actions, nonexecution of correct actions, severe performance degradation, and so forth.

Failure function: An analytical or empirical function describing the probability that a system fails at time t.

Fault: The appearance of a defect during the operation of a software system.

Fault prevention: Any technique or process that attempts to eliminate the possibility of having a failure occur in a hardware device or software routine.

Fault tolerance: Correct execution of a specified function in a system, provided by redundancy, despite existing faults. The redundancy provides the information needed to negate the effects of faults.

Feature-driven development: A lightweight model-driven, short-iteration process built around the feature, a unit of work that has meaning for the client and developer and is small enough to be completed quickly.

Feature points: An extension of function points to cope with systems having a high level of algorithmic complexity. This software metric is also suitable for embedded environments.

Fetch: The process of retrieving a machine language instruction from main memory and placing it in the instruction register.

Fetch–execute cycle: The process of continuously fetching and executing machine language instructions from the main memory.

Field Programmable Gate Array (FPGA): A digital integrated circuit, which is designed so that it can be configured (or "programmed") for specific applications after manufacturing.

Fieldbus: A communications network intended for automation and control applications.

File fragmentation: Analogous to memory fragmentation, but occurring within files, with the same associated problems.

Finite State Automaton (FSA): *See* finite state machine.

Finite State Machine (FSM): A mathematical model of a machine consisting of a set of inputs, a set of states, and a transition function that describes the next state given the current state and an input event. *Also* known as finite state automaton and state diagram.

Firing: In Petri nets or in certain multiprocessor architectures, when a process performs its prescribed function.

Firm real-time system: A real-time system that can fail to meet a few deadlines without system failure.

Firmware: Small system programs, which are used to control specific hardware devices. Firmware is stored in a nonvolatile memory that is usually unalterable.

Fixed priority system: A multitasking system in which the task priorities cannot be changed. Contrast with dynamic priority system.

Fixed-rate system: A system in which interrupts occur only at fixed rates.

Floating-point number: A term describing the computer's representation of a real-valued number.

Flowchart: A traditional graphic representation of an algorithm or a program in using named functional blocks, decision evaluators, and I/O symbols interconnected by directional arrows that indicate the flow of processing.

Flush: In pipelined CPU architectures, the act of emptying the instruction pipeline when branching occurs.

Foreground: A collection of interrupt-driven or real-time tasks in foreground/background systems.

Formal parameter: The dummy variable used in the description of a procedure or subroutine.

Forward error recovery: A technique of continuing processing by skipping a few faulty states (applicable to some real-time systems in which occasional missed or wrong responses are tolerable).

FPGA: *See* field programmable gate array.

Framework: A skeletal structure of a program that requires further elaboration.

FSA: Finite state automaton. *See* finite state machine.

FSM: *See* finite state machine.

Function points: A widely used software metric in nonembedded environments. Function points measure the number of interfaces between modules and subsystems in programs or systems.

Function test: A check for correct device operation generally by truth table verification.

Functional decomposition: The division of tasks into modules according to their functionality.

Functional requirements: Those system features that can be directly verified by executing the program.

Garbage: An object or a set of objects that can no longer be accessed, typically because all pointers that direct accesses to the object or set have been eliminated.

Garbage collector: A software run-time system component that periodically scans dynamically allocated storage and reclaims allocated storage that is no longer in use.

General register: CPU's internal memory that is addressable in the address field of certain machine-language instructions.

General semaphore: *See* counting semaphore.

Generalization: The relationship between a class and one or more variations of that class.

Generator polynomial: The modulo-2 divisor of the message polynomial in CRC.

Global variable: Any variable that is within the scope of all modules of the software system.

Group walkthrough: A kind of white-box testing in which a number of persons inspect the code line-by-line with the unit author.

Hamming code: An effective coding technique used to detect and correct errors in computer memory.

Hard error: Physical (unrepairable) damage to a memory cell.

Hard real-time system: A real-time system in which missing even one deadline results in system failure.

Hazard: A momentary output error that occurs in a logic circuit because of input signal propagation along different delay paths in the circuit.

Heterogeneous: Having dissimilar hardware/software components in a system.

Host: A computer that is the one responsible for performing a certain computation or function.

Hybrid system: A system in which interrupts occur both at fixed rates and sporadically.

ICE: *See* in-circuit emulator.

Immediate-mode instruction: An instruction in which the operand is an integer-valued number.

Implied-mode instruction: An instruction involving one or more specific memory locations or registers that are implicitly defined in the operation performed by the instruction.

Imprecise computation: Techniques involving early termination of an iterative computation in order to meet deadlines.

In-Circuit Emulator (ICE): A device that replaces the processor and provides the functions of the processor plus various testing and debugging functions.

Incrementality: A software development approach in which progressively larger increments of the desired product are developed.

Indirect-mode instruction: Instruction where the operand field is a memory location containing the address of the address of the operand.

Induction variable: A variable in a loop that is incremented or decremented by some constant.

Information hiding: A program design principle that makes available to a function just the data it needs—everything else is hidden.

Inheritance: In object orientation, the possibility for different data types to share the same code.

Initialize: To place a hardware system in a known state, for instance, at power-up. Or to store the correct initial data in a data item, for example, filling an array with zero values before it is used.

Input space: The set of all possible input combinations to a system.

Instance: An occurrence of a class.

Instruction issue: The sending of an instruction to CPU's functional units for execution.

Instruction register: CPU's internal register that holds the instruction pointed to by the contents of the program counter.

Instruction set: The instruction set of a processor is the collection of all the machine-language instructions available to the programmer (or a high-level language compiler).

Integration: The process of uniting hardware/software modules from different sources to form the overall system.

Internal fragmentation: Condition that occurs in fixed-partition schemes when, for instance, a processor requires 1 K byte of memory, while only 2 K byte partitions are available.

Interoperability: Software quality that refers to the ability of the software system to coexist and cooperate with other systems.

Interpreter: A computer program that translates and immediately performs intended operations of the source statements of a high-level language program.

Interrupt: An input to a processor that signals the occurrence of an asynchronous event.

Interrupt controller: A device that provides additional interrupt handling capability to a CPU.

Interrupt handler: A predefined subprogram that is executed when an interrupt occurs. *Also* known as an interrupt service routine.

Interrupt handler location: Memory location containing the starting address of an interrupt handler routine. The program counter is automatically loaded with its address when the particular interrupt occurs.

Interrupt latency: The delay between when an interrupt request occurs and when the CPU begins reacting to it.

Interrupt register: A register containing a bit map of all pending (latched) interrupts.

Interrupt return location: Memory location (usually in the stack memory) where the content of the program counter is saved when the CPU processes an interrupt.

Interrupt Service Routine (ISR): *See* interrupt handler.

Interrupt vector: Register that contains the identity of the (highest-priority) interrupt request. The interrupt vector is sent to the CPU by the device whose interrupt request was just acknowledged by the CPU.

Intrinsic function: A macro where the actual function call is replaced by corresponding in-line code.

Jackson chart: A popular form of structure chart that provides for conditional branching.

Kalman filter: A mathematical construct used, for instance, to combine measurements of the same quantity from different sources.

KDSI: *See* KLOC.

Kernel: The smallest portion of the operating system that provides for task scheduling and dispatching only.

Kernel preemption: A method used in real-time Unix that provides preemption points in calls to kernel functions to allow them to be interruptible.

Key: In a mailbox, the data that are passed as a flag used to protect a critical region.

KLOC: A software metric measuring thousands of lines of source code (not counting comments and nonexecutable statements). *Also* known as thousands of delivered source instructions KDSI) and noncommented source code statements (NCSS).

Latency: A measure of time delay experienced in a real-time system.

Least Recently Used (LRU) rule: The best nonpredictive memory-page replacement algorithm.

Legacy system: Applications that are in a maintenance phase but are not ready for retirement.

Leveling: In data flow diagrams, the process of redrawing a diagram at a finer level of detail.

Library: A set of precompiled routines that may be linked with a program at compile time or loaded at load time or dynamically at run time.

Lightweight programming methodology: Any programming methodology that is adaptive rather than predictive and emphasizes people rather than processes. *Also* known as agile programming.

Link: The postcompilation process in which individual object modules are placed together and cross-module references resolved.

Linker: A computer program that takes one or more object files, assembles them into blocks that are to fit into particular regions in memory, and resolves all references to other segments of a program and to libraries of precompiled program units.

Little's law: Rule from queuing theory stating that the average number of customers in a queuing system is equal to the average arrival rate of the customers to that system times the average time spent in the system.

Live variable: A variable that can be used subsequently in the program.

Livelock: Another term for task starvation.

Load module: Executable code that can be readily loaded into the machine.

Locality-of-reference: The notion that if you examine a list of recently executed program instructions, you will see that most of the instructions are localized to within a small number of addresses.

Lock-up: When a system enters a state in which it is rendered ineffective.

Logic analyzer: A sophisticated instrument that can be used to read, store, and display signals from individual circuits, circuit boards, or hardware systems.

Logical operation: A machine-language instruction that performs Boolean operations, such as AND, OR, and XOR.

Look-up table: An arithmetic technique that uses precalculated tables for function values and may rely on mathematical definition of the derivative to interpolate these functions quickly.

Loop invariant optimization: The process of placing computations outside a loop that do not need to be performed within the loop.

Loop invariant removal: A code optimization technique that involves removing code that does not change inside a looping sequence.

Loop jamming: An optimization technique that involves combining multiple loops within the control of one loop variable.

Loop unrolling: A code optimization technique that involves expanding a loop so that loop overhead is completely removed.

Loosely coupled system: A software system that can run on other hardware with the rewrite of no more than a few modules (perhaps device drivers).

LRU: *See* least recently used rule.

Machine code: The machine format of a compiled executable, in which individual instructions are represented in binary notation.

Machine language: The set of legal instructions to a CPU, expressed in binary notation.

Macro: *See* macroinstruction.

Macroinstruction: A native machine-language instruction.

Macroprogram: A sequence of macroinstructions.

Mailbox: An intertask communication device consisting of a memory location and two operations—post and pend—that can be performed on it.

Main memory: Memory that is directly addressable by the CPU.

Maintainability: A software quality that is a measure of how easily the system can be evolved to accommodate new features, or changed to repair errors.

Maintenance: The changes made on a system to fix errors, to support new requirements, or to make it more efficient.

Major cycle: The largest sequence of repeating processes in cyclic or periodic systems.

MAR: *See* memory address register.

Mask register: A register that contains a bit map either enabling or disabling specific interrupts.

Master processor: The online processor in a master/slave configuration.

MDR: *See* memory data register.

Mealy finite state machine: A finite state machine with outputs during transitions.

Memory Address Register (MAR): Register that holds the address of the memory location to be acted on.

Memory caching: A technique in which frequently used segments of main memory are stored in a faster and smaller bank of memory, called a cache, which is local to the CPU.

Memory Data Register (MDR): Register that holds the data to be written to or that is read from the memory location held in the MAR.

Memory leak: It occurs when a task consumes temporary memory, but is unable to release it back to the real-time operating system after it is no more needed.

Memory-loading: The percentage of usable memory that is being used.

Memory locking: In a real-time system, the process of locking all or certain parts of a task into memory to reduce the overhead involved in paging, and thus make the execution times more predictable.

Memory-mapped I/O: An input/output scheme where reading or writing involves executing a load or store instruction on a pseudomemory address mapped to the device. Contrast with DMA and programmed I/O.

Memory reference instruction: An instruction that communicates with memory, writing to it (store) or reading from it (load).

Message exchange: *See* mailbox.

Message polynomial: Used in CRC.

Metadata: Data that describes other data.

Methods: In object-oriented systems, functions that can be performed on objects.

Microcode: A stream of low-level operations that are executed as a result of a single macroinstruction being executed.

Microcontroller: A single-chip computer system, which contains a CPU, some memory, and I/O ports.

Microinstructions: *See* microcode.

Microkernel: A kernel that provides for task scheduling and dispatching only.

Micromemory: CPU's internal memory that holds the individual microcodes corresponding to macroinstructions.

Microprogram: Sequence of microcode stored in the micromemory.

Minor cycle: A sequence of repeating processes in cyclic or periodic systems.

Mixed listing: A printout that combines the high-level language instructions with the corresponding assembly language code.

Mixed system: A system in which interrupts occur both at fixed rates and sporadically.

Modularity: Design principle that calls for design of small, self-contained code units.

Moore finite state machine: *See* finite state machine.

Multi-core processor: A processor that is composed of two or more CPUs, which are independent but share some memory.

Multiplexer (MUX): An analog or digital device used to route multiple lines onto fewer lines.

Multiprocessing operating system: An operating system where more than one processor is available to provide for simultaneity. Contrast with multitasking operating system.

Multiprocessor: A computer system that has more than one internal processor capable of operating collectively on a computation. Normally associated with those systems where the individual processors can access a common main memory.

Multitasking operating system: An operating system that provides sufficient functionality to allow multiple tasks to run on a single processor so that the illusion of simultaneity is created. Contrast with multiprocessing operating system.

Mutex: A common name for a semaphore variable.

MUX: *See* multiplexer.

NCSS: Noncommented source statements. *See* KLOC.

Nested subroutine: A subroutine called by another subroutine.

Nonfunctional requirements: System requirements that cannot be tested simply by program execution.

Nonvolatile memory: Memory whose contents are preserved upon removing power.

Non von Neumann architecture: An architecture that does not use the stored-program serial fetch–execute cycle.

No-op: A macroinstruction that does not change the state of the CPU but just advances the program counter.

NP-complete problem: A decision problem that is a seemingly intractable problem for which the only known solutions are exponential functions of the problem size and which can be transformed to all other NP-complete problems. Compare with NP-hard problem.

NP-hard problem: A decision problem that is similar to an NP-complete problem, except that for the NP-hard problem it cannot be shown to be transformable to all other NP-complete problems.

N-version programming: A programming technique used to reduce the likelihood of system lock-up by using redundant processors, each running software that has been coded to the same specifications by different teams.

Nucleus: *See* kernel.

Null: A special value denoting that an attribute value is unknown or not applicable.

Object: An instance of a class definition.

Object code: A file comprising a compiled description of a program segment.

Object-oriented: The organization of software into discrete objects that encapsulate both the data structure and behavior.

Object-oriented analysis: A method of analysis that estimates requirements from the perspective of the classes and objects found in the problem domain.

Object-oriented design: A design methodology viewing a system as a collection of objects with messages passed from object to object.

Object-oriented language: A programming language that provides constructs that encourage a high degree of information hiding and data abstraction.

Object-oriented methodology: An application development methodology that uses a top-down approach based on the decomposition of a system in a collection of objects communicating via messages.

Object-oriented programming: A programming style using languages that support abstract data types, inheritance, function polymorphism, and messaging.

Object type: The type of an object determines the set of allowable operations that can be performed on the object. This information can be encoded in a "tag" associated with the object, can be found along an access path reaching to the object, or can be determined by the compiler that inserts "correct" instructions to manipulate the object in a manner consistent with its type.

Opcode: Starting address of the microcode of a machine language instruction stored in micromemory.

Open source code: Source code that is made available to the user community for moderate improvement and correction.

Open system: An extensible collection of independently written applications that cooperate to function as an integrated system.

Operating system: A set of programs that manages the operations of a computer. It oversees the interaction between the hardware and software and provides a set of services to system users.

Operation: Specification of one or a set of computations on the specified source operands placing the results in the specified destination operands.

Organic system: A system that is not embedded.

Orthogonal product: In statecharts, a process that depicts concurrent tasks that run in isolation.

Output dependency: The situation when two sequential instructions in a program write to the same location. To obtain the desired result, the second instruction must write to the location after the first instruction.

Output space: The set of all possible output combinations for a system.

Overlay: Dependent code and data sections used in overlaying.

Overlaying: A technique that allows a single program to be larger than the allowable user space.

Overloading: Principle according to which operations bearing the same name apply to arguments of different data type.

Page: Fixed-size chunk used in demand-paged memory systems.

Page fault: An exception that occurs when a memory reference is made to a location within a page not loaded in main memory.

Page frame: *See* page.

Page stealing: When a page is to be loaded into main memory, and no free pages are found, then some page frame must be written out or swapped to disk to make room.

Page table: A collection of pointers to pages used to allow noncontiguous allocation of page frames in demand paging.

Pair programming: A technique in which two persons write code together.

Parnas partitioning: *See* information hiding.

Pattern: A named problem–solution pair that can be applied in new contexts, with advice on how to apply it in novel situations.

PC: *See* program counter.

PDL: *See* program design language.

Peephole optimization: A code optimization technique where a small window of assembly language or machine code is compared against known patterns that yield optimization opportunities.

Pend operation: Operation of removing data from a mailbox. If data are not available, the task performing the pend suspends itself until the data become available.

Performance: A measure of the software's capability of meeting certain functional constraints such as timing or output precision.

Petri net: A mathematical/pictorial system description technique.

Phantom interrupt: *See* spurious interrupt.

Phase-driven code: *See* state-driven code.

PIC: Priority interrupt controller. *Also* known as interrupt controller.

Ping-pong buffering: *See* double buffering.

Pipeline: For example, an intertask communication mechanism provided in some operating systems. *See* also pipelining.

Pipelining: A technique used to increase CPU's instruction throughput that relies on the fact that fetching the instruction is only one part of the fetch–execute cycle, and that is can overlap with different parts of the fetch–execute cycle for other instructions.

PIU: Peripheral interface unit.

PL/I: A procedural programming language that was introduced in the 1960s. It was the model for the first high-level programming languages that were developed in the late 1970s for microprocessors (such as MPL, PL/M, and PL/Z).

Polled loop system: A real-time system in which a single and repetitive test instruction is used to test a flag, which indicates that some event has occurred.

Polymorphism: In object-oriented programming, polymorphism allows the programmer to create a single function that operates on different objects, depending on the type of object involved.

Portability: A quality in which the software can easily run in different hardware and operating system environments.

Post operation: Operation that places data in a mailbox.

Power bus: The collection of wires used to distribute power to the various components of computer systems.

Power on self-test: A series of diagnostic tests performed by a computer when it powers on.

Pragma: In certain programming languages, a pseudo-op that allows assembly code to be placed in line with the high-level language code.

Preempt: A condition that occurs when a higher-priority task interrupts a lower-priority task.

Preemptive-priority system: A system that uses preemption schemes instead of round-robin or first-come, first-served scheduling.

Primary memory: *See* main memory.

Priority inversion: A condition that occurs when a medium-priority task is executing while a high-priority task is waiting for a shared resource from a low-priority task.

Procedure: A self-contained code sequence designed to be reexecuted from different places in a main program or another procedure.

Process: The context, consisting of allocated memory, open files, and network connections, in which an operating system places a running program.

Process control block: An area of memory containing information about the context of an executing program.

Program Counter (PC): A CPU register containing the address of the next macroinstruction to be executed.

Program Design Language (PDL): A type of abstract high-order language used in system specification.

Programmed I/O: Transferring data to or from a peripheral device by running a program that executes specific computer instructions or commands to control the transfer. An alternative is to transfer data using DMA.

Propagation delay: The contribution to interrupt and other latencies due to limitations in switching speeds of digital devices and in the transit time of electrons across wires.

Protection fault: An error condition detected by the address mapper when the type of request is not permitted by the object's access code.

Prototype: A mock-up of a software system often used during the design phase.

Prototyping: Building an engineering model of all or part of a system to verify that the concept works.

Pseudocode: A technique for specifying the logic of a program in an English-like language. Pseudocode does not have to follow strict syntax rules and can be read by anyone who understands programming logic.

Pseudo-exhaustive testing: A testing technique that relies on various forms of hardware/software segmentation and application of exhaustive test patterns to these segments.

Pseudo-operation: In assembly language, an operation code that is an instruction to the assembler rather than a machine-language instruction.

Pseudorandom testing: A testing technique based on pseudorandomly generated test patterns. The test comprehensiveness is adapted to the required level of fault coverage.

Pure procedure: A procedure that does not modify itself during its own execution. The instructions of a pure procedure can be stored in a read-only portion of the memory and can be accessed by multiple tasks.

Race condition: A situation where multiple tasks access and manipulate shared data with the outcome dependent on the relative timing of these tasks.

Raise: Mechanism used to initiate a software interrupt in certain programming languages, such as C.

RAM scrubbing: A technique used in memory configurations that include error detection and correction chips. Such a technique, which reduces the chance of multiple-bit errors occurring, is needed because in some configurations, memory errors are corrected on the bus and not in memory itself. The corrected memory data then need to be written back to memory.

Random testing: The process of testing using a set of pseudorandomly generated test patterns.

Random variable: An integer- or real-valued variable whose values are not predictable but random.

Rate-monotonic system: A fixed-rate, preemptive, prioritized real-time system where the task priorities are assigned so that the higher the execution rate, the higher the priority.

Reactive system: A system that has some essential interaction with its environment.

Read/write line: A logic control line that is set to logic 0 during each memory write and to logic l during memory read.

Ready state: In the task-control block model, the state of those tasks that are ready to execute, but are not executing since some higher priority task is executing.

Real-time: Refers to systems whose correctness depends not only on outputs but the timeliness of those outputs as well. Failure to meet one or more of the deadlines can result in system failure.

Real-time computing: Support for operating environments in which response time to an event must occur within a predetermined amount of time. Real-time systems may be categorized into hard, firm, and soft real-time.

Recovery: Action that restores the state of a task to an earlier configuration after it has been determined that the system has entered a state that does not correspond to the desired functional behavior. For correct functional behavior, the states of all tasks should be restored in a manner consistent with each other and with the conditions within communication links or message channels.

Recovery block: Section of code that may terminate in checkpoints. If the check fails, processing can resume at the beginning of a recovery block.

Recursion: The process whereby a program calls itself.

Recursive procedure: A procedure that can be called by itself or by another program that it has called; effectively, a single task can have several executions of the same procedure alive at the same time. Recursion provides one means of defining special functions, such as the factorial function.

Reduced Instruction Set Computer (RISC): CPU architecture usually characterized by a compact instruction set with limited addressing modes and hardwired (as opposed to microcoded) macroinstructions.

Reduction in strength: A code optimization technique that uses the fastest macroinstruction available to accomplish a given calculation.

Redundancy: The use of parallel or serial components in a system to reduce the probability of failure. Similarly, referring to an increase in the number of components that can interchangeably perform the same function in a system. Redundancy can increase the system reliability.

Reentrant: Term describing a program that uses concurrently exactly the same executable code in memory for more than one invocation of the program rather than separate copies of a program for each invocation. The read and write operations must be timed so that the correct results are

always available and the results produced by one invocation are not overwritten by another.

Reentrant procedure: A procedure that can be used by several concurrently running tasks in a multitasking system.

Refactoring: To perform a behavior-preserving code transformation.

Register-direct mode: A memory-addressing scheme similar to direct mode except the operand is a CPU register and not an explicit address.

Register-indirect mode: A memory-addressing scheme similar to indirect mode, except the operand address is kept in a register rather than in another memory address.

Regression testing: A testing methodology used to validate modified software against an earlier set of test cases that have already been passed.

Reliability: The probability that a component or system will function without any failure over a specified time period, under stated conditions.

Requirements analysis: A phase of software development life cycle in which the high-level (or business) requirements for a software product are defined and documented.

Response time: The time between the presentation of a set of inputs to a software system and the appearance of all the associated outputs.

Reusability: The possibility to use or easily adapt the hardware or software developed for a specific system to build other systems. Reusability is a property of module design that permits and supports reuse.

Reuse: Program modules are reused when they are copied from one program and used in another.

Reverse engineering: The reverse analysis of an old application to conform to a new methodology.

Ring buffer: A first-in, first-out list in which simultaneous input and output to the list is achieved by keeping separate head and tail pointers. Data are loaded at the tail and read from the head.

RISC: *See* reduced instruction set computer.

Robustness: A software quality that measures the software's tolerance to exceptional situations, for example, an input out of range.

Root: In overlaying memory management, the portion of memory containing the overlay manager and code common to all overlay segments, such as math libraries.

Round-robin system: A system in which several tasks are executed sequentially to completion, often in conjunction with a cyclic executive.

Round-robin system with timeslicing: A system in which each executable task is assigned a fixed time quantum called a time slice in which to execute. A clock is used to initiate an interrupt at a rate corresponding to the time slice.

RTOS: Real-time operating system.

RTSJ: The real-time specification of Java.

SA: *See* structured analysis.

Safety: The probability that a system will either perform its functions correctly or will discontinue its functions in a well-defined and safe manner.

Safety-critical system: A system that is intended to handle unexpected, dangerous events.

Sampling rate: The rate at which an analog input signal is converted to digital form.

Scale factor: A technique used to simulate floating-point operations by assigning an implicit noninteger value to the least significant bit (LSB) of an integer.

Scaled number: A performance optimization technique where the least significant bit (LSB) of an integer variable is assigned a real number scale factor.

Schedulability analysis: The compile-time prediction of tasks' execution time performance.

Scheduler: The part of the real-time kernel that determines which task will execute.

Scratch pad memory: CPU's internal memory used for intermediate results.

Screen signature: The CRC of a screen memory.

Scrum: A lightweight programming methodology based on the empirical process control model, the name is a reference to the point in a rugby match where the opposing teams line up in a tight and contentious formation. Scrum programming relies on self-directed teams and dispenses with much advanced planning, task definition, and management reporting.

SD: *See* structured design.

Secondary memory: Memory that is characterized by long-term storage devices, such as hard disks and Flash cards, which are not part of the physical address space of the CPU.

Segment: A disjoint processing element in instruction pipelining. *Also* called a stage.

Self-modifying code: A program using a machine instruction that changes the stored binary pattern of another machine instruction in order to create a different instruction that will be executed subsequently. This is by no means a recommended practice.

Self-test: A functional test that a module, either hardware or software, runs upon itself.

Self-test and repair: A fault-tolerant technique based on a functional unit's active redundancy, spare switching, and reconfiguration.

Semaphore: A special variable type used for protecting critical regions.

Semaphore primitives: The two fundamental operations that can be performed on a semaphore, namely, wait and signal.

Semidetached system: *See* loosely coupled system.

Serially reusable resource: A resource that can only be used by one task at a time and that must be used to completion.

Server: A task used to manage multiple requests to a serially reusable resource.

SEU: *See* single-event upset.

Signal operation: Operation on a semaphore that essentially releases the resource protected by the semaphore.

Single-Event Upset (SEU): Alteration of memory contents due to charged particles present in space, or in the presence of a nuclear event.

Slave processor: The off-line processor in a master–slave configuration.

SLOC: *See* source lines of code.

Soft computing: An association of computing methodologies centering on fuzzy logic, artificial neural networks, and evolutionary computation. Each of these methodologies provides complementary and synergistic reasoning and searching methods to solve complex, real-world problems. *Also* known as computational intelligence.

Soft error: Repairable alternation of the contents of a memory cell.

Soft real-time system: A real-time system in which failure to meet deadlines results in performance degradation but not necessarily a system failure.

Software: A systematic composition of macroinstructions.

Software design: A phase of software development lifecycle that maps what the system is supposed to do into how the system will do it in a particular hardware/software configuration.

Software development lifecycle: A way to divide the work that takes place in the development of an application.

Software engineering: Systematic development, operation, maintenance, and retirement of software.

Software evolution: The process that adapts the software to changes of the environment where it is used.

Software interrupt: A machine language instruction that initiates an interrupt function. Software interrupts are often used for system calls, because they can be executed from anywhere in memory and the CPU provides the necessary return address handling.

Software reengineering: The reverse analysis of an old application to conform to a new methodology.

Software reliability: The probability that a software system will not fail before some time t, under certain conditions.

Source code: Software code that is written in a form or language meant to be understood by programmers. Must be translated (or compiled) to object code in order to run on a computer.

Source Lines of Code (SLOC): A metric that measures the number of executable program instructions; one SLOC may span several lines, for instance, as in an if-then-else statement.

Spatial fault tolerance: Methods involving redundant hardware or software components.

Specification: A statement of the design or development requirements to be satisfied by a system or product.

Speculative execution: An instruction execution technique in which instructions are executed without regard to data dependencies.

Spin lock: Another name for the wait semaphore operation.

Sporadic system: A system with all interrupts occurring sporadically.

Spurious interrupt: An extraneous and unwanted interrupt. *Also* known as a phantom interrupt.

SRAM: *See* static random-access memory.

Stack: A first-in, last-out data structure.

Stack machine: Computer architecture in which the instructions are centered on an internal memory store called a stack, and an accumulator.

Stage: *See* segment.

Starvation: A condition that occurs when a task is not being serviced frequently enough.

State diagram: A diagram showing the conditions (states) that can exist in a logic system and what signals are required to go from one state to another state.

State-driven code: Program code based on a finite state machine.

Static Random-Access Memory (SRAM): Random access memory that does not need to be recharged (or refreshed) periodically.

Statistically based testing: Technique that uses an underlying probability distribution function for each system input to generate random test cases.

Stress testing: A type of testing wherein the system is subjected to a large disturbance in the inputs (e.g., a large burst of interrupts), followed by smaller disturbances spread out over a longer period of time.

Structure chart: Graphical design tool used to partition system functionality.

Structured Analysis (SA): A graphical methodology for systems analysis.

Structured Design (SD): A graphical methodology for systems design, which is related to structured analysis.

Subclass: A class that adds specific attributes, behavior, and relationships for a generalization.

Subroutine: A group of instructions written to perform a specific task, independent of a main program and can be accessed by a program or another subroutine to perform the task.

Superclass: A class that holds common attributes, behavior, and relationships for generalization.

Suspended state: In the task control block model, those tasks that are waiting on a particular resource, and thus are not ready. Also called blocked state.

Swapping: The simplest scheme that allows the operating system to allocate main memory to two tasks simultaneously.

Switch bounce: The physical phenomenon that an electromechanical switch cannot change logic states instantaneously without short-term oscillation between these states.

Synchronous: An operation or multiple operations that are controlled or synchronized by a clocking signal.

Synchronous data: *See* time-relative data.

Synchronous event: An event that occurs at predictable times in the flow-of-control.

Syndrome bits: The extra bits needed to implement a Hamming code.

Syntax: The part of a formal definition of a programming language that specifies legal combinations of symbols that make up statements in the language.

System: An entity that when presented with a set of inputs produces corresponding outputs.

System integration: A phase of the software development lifecycle during which a software product is integrated into its operational environment.

System program: Software used to manage the resources of a computer.

System unification: A process consisting of linking together the testing software modules in an orderly fashion.

Systems engineering: An approach to the overall lifecycle evolution of a product or system. Generally, the systems engineering process comprises a number of phases. There are three essential phases in any systems engineering lifecycle: formulation of requirements and specifications, design and development of the system or product, and deployment of the system. Each of these basic phases can be further expanded.

Task Control Block (TCB): A collection of data associated with a task including context, process code (or a pointer to it), and other necessary information.

TCB: *See* task control block.

Temporal determinism: A condition that occurs when the response time for each set of outputs is known in a deterministic system.

Temporal fault tolerance: Techniques that allow for tolerating missed deadlines.

Test-and-set instruction: A macroinstruction that can atomically test and then set a particular memory address to some value.

Test first coding: A software engineering technique in which the code's unit test cases are written by the programmer before the actual code is written.

Test pattern: An input vector designed in such a way that the faulty output is different from the fault-free output.

Test probe: A checkpoint used only during testing.

Test suite: A collection of test cases.

Testability: The measure of the ease with which a system can be tested.

Testing: A phase of software development life cycle during which the application is exercised for the purpose of finding errors.

Thrashing: Very high paging activity.

Throughput: A measure of the number of macroinstructions per second that can be processed based on some predetermined instruction mix.

Time loading: The percentage of "useful" processing the computer is doing. *Also* known as the utilization factor.

Time overloaded: A system that is 100% or more time loaded.

Time-relative data: A collection of data that must be time correlated.

Timeslice: A fixed time quantum used to limit execution time in round-robin systems.

Timing error: An error in a system due to faulty time relationships between some of its constituents.

Traceability: A software property that is concerned with the relationships between requirements, their sources, and the system design.

Tracing: In software engineering, the process of capturing a stream of instructions, referred to as the trace, for later analysis.

Transceiver: A transmitter/receiver hybrid device.

Trap: Internal interrupt caused by the execution of a certain operation, such as a divide by zero.

UML: *See* Unified modeling language.

Unconditional branch: A "jump" instruction that causes a transfer of control to another address without regard to the state of any condition flags.

Unified Modeling Language (UML): A collection of modeling tools for object-oriented representation of software and other enterprises.

Unified Process Model (UPM): A process model that uses an object-oriented approach by modeling a family of related software processes using the UML as a notation.

Unit: A software module.

Unreachable code: Code that can never be reached in the normal flow of control.

UPM: *See* unified process model.

Usability: A property of software detailing the ease in which it can be used.

User space: Memory not required by the operating system.

Utilization factor: *See* time loading.

Validation: A review to establish the quality of a software product for its operational purpose.

Verifiability: A software property in which its other properties (e.g., portability and usability) can be verified easily.

Version control software: A system that manages the access to the various software components from the software library.

Very Long Instruction Word (VLIW) computer: A computer that implements a form of parallelism by combining microinstructions to exploit redundant CPU components.

Virtual machine: A task on a multitasking computer that behaves as if it were a standalone computer and not part of a larger system.

VLIW: *See* very long instruction word computer.

Void: Empty data type in C language. For example, when used as a function return type, `void` means that the function does not return any value.

Volatile memory: Memory in which the contents will be lost if power is removed.

von Neumann architecture: A CPU employing a serial fetch–execute process.

von Neumann bottleneck: A situation in which the serial fetch and execution of instructions limits the overall execution speed.

WBS: *See* work breakdown structure.

Wait-and-hold condition: The situation in which a task acquires a resource and then does not relinquish it until it can acquire another resource.

Wait operation: Operation on a semaphore that essentially locks the resource protected by the semaphore, or prevents the requesting task from proceeding if the resource is already locked.

Wait state: Additional clock cycles used to synchronize macroinstruction execution with the access time of memory.

Watchdog timer: A device that must be reset periodically or a discrete "alarm" signal is issued.

White-box testing: Logic-driven testing designed to exercise all paths in the software module. Same as clear-box testing.

Work Breakdown Structure (WBS): A hierarchically decomposed listing of tasks.

ABOUT THE AUTHORS

Dr. Phillip A. Laplante is a Professor of Software Engineering and a member of the Graduate Faculty at The Pennsylvania State University. Before joining Penn State, he was a Professor and senior academic administrator at several other colleges and universities.

Prior to his academic career, Dr. Laplante was a software engineer and project manager working on avionics, computer-aided design, and software test systems. He has nearly 30 years of experience in building, studying, and teaching real-time systems. His travels have taken him to NASA, UPS, Lockheed Martin, the Canadian and Australian Defense Forces, MIT's Charles Stark Draper Lab, and many other places. His practical and theoretical knowledge of real-time systems has been enhanced during these visits and in interactions with hundreds of students from Boeing, Motorola, Siemens, and other major companies.

Dr. Laplante has authored or edited 25 books (including three technical dictionaries and the *Encyclopedia of Software Engineering*) and has published more than 150 scholarly papers. He also co-founded the journal *Real-Time Imaging*, which he edited for five years, and serves as editor-in-chief for three book series.

Dr. Laplante received his B.S., M.Eng., and Ph.D. degrees in computer science, electrical engineering, and computer science, respectively, from Stevens

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.

Institute of Technology, and an MBA from the University of Colorado. He is a licensed professional engineer in the Commonwealth of Pennsylvania and a Certified Software Development Professional. He is a Fellow of both the IEEE and SPIE for his achievements in real-time systems and real-time imaging research and education.

Dr. Seppo J. Ovaska is a Professor in the School of Electrical Engineering at Aalto University, Finland. His current research focuses on computationally intelligent systems and their applications. Dr. Ovaska is a prolific author, having published more than 100 peer-reviewed journal articles, 155 conference publications, and 10 book chapters. In addition, he has edited the pioneering book *Computationally Intelligent Hybrid Systems: The Fusion of Soft Computing and Hard Computing* (Wiley-Interscience, 2004), and holds nine patents in the area of high-rise elevator systems.

Dr. Ovaska received a D.Sc. degree in electrical engineering from the Tampere University of Technology, Finland. He earned an Lic.Sc. degree in computer science and engineering from the Helsinki University of Technology, Finland, and an M.Sc. in electrical engineering from the Tampere University of Technology. During the academic year 2006–2007, he served as a Visiting Professor of Electrical and Computer Engineering at Utah State University. Dr. Ovaska has taught university courses in computer architectures, embedded microprocessor systems, microcomputer hardware and software, microcomputer systems programming, and real-time systems design. Prior to his academic career, he held software engineering, research, and R&D management positions, both in Finland and Kentucky.

Dr. Ovaska has served as a guest editor for the prestigious *Proceedings of the IEEE*. Besides, he was an elected member of the board of governors, IEEE Systems, Man, and Cybernetics (SMC) Society. He is a recipient of two Outstanding Contribution Awards, as well as the Most Active SMC Technical Committee Award of the IEEE SMC Society.

INDEX

Note: Page numbers in *italics* refer to figures, those in **bold** to tables.

Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition.
Phillip A. Laplante and Seppo J. Ovaska.