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.

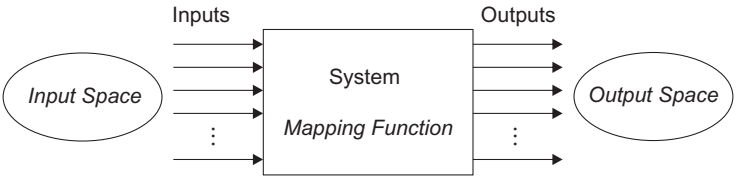

Figure 1.1. A general system with inputs and outputs.

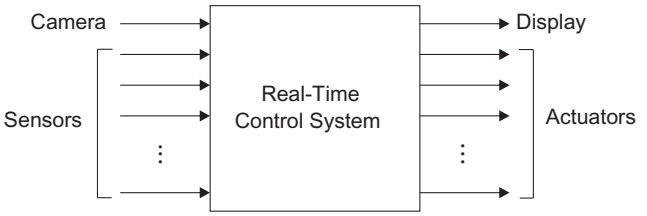

Figure 1.2. A real-time control system including inputs from a camera and multiple sensors, as well as outputs to a display and multiple actuators.

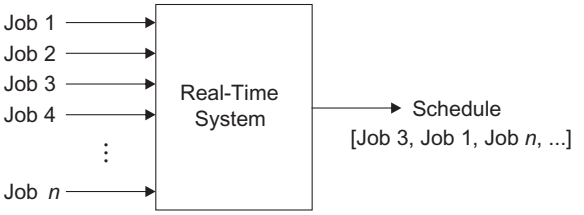

Figure 1.3. A classic representation of a real-time system as a sequence of schedulable jobs.

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

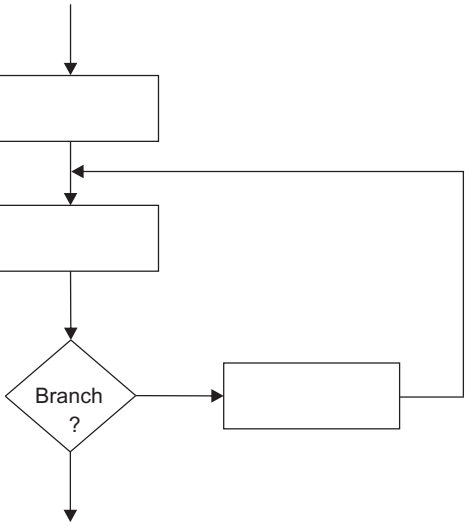

Figure 1.4. A partial program flowchart showing a conditional branch as a change in flow of control.

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.

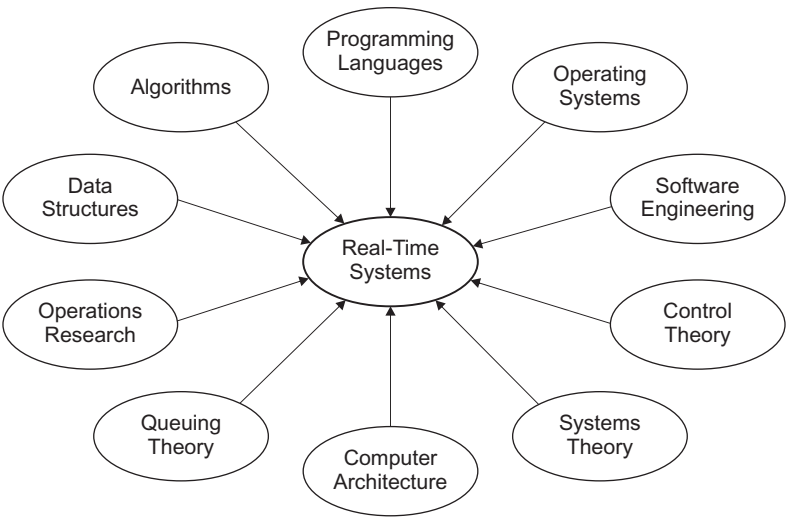

Figure 1.5. A variety of disciplines that affect real-time systems engineering.

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*.

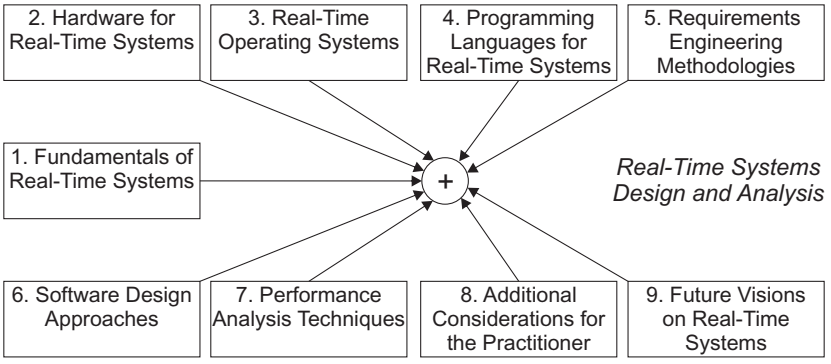

Figure 1.7. Composition of this unique text from nine complementary chapters.

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.