CHAPTER

Embedded Computing 1

CHAPTER POINTS

- Why we embed microprocessors in systems.
- What is difficult and unique about embedding computing?
- Real-time and low-power computing.
- Embedded computing as the foundation for Internet-of-Things (IoT) systems and cyber-physical systems (CPS).
- Design methodologies.
- Unified Modeling Language (UML).
- A guided tour of this book.

1.1 Introduction

For us to understand how to design embedded computing systems, we first need to understand the use cases—how and why microprocessors are used for control, user interface, signal processing, and many other tasks. The **microprocessor** has become so common that it is easy to forget how hard some things are to do without it.

We first review the various uses of microprocessors. We then review the major reasons why microprocessors are used in system design—delivering complex behaviors, fast design turnaround, and so on. Next, in Section 1.2, we walk through the design of a simple example to understand the major steps in designing an embedded computing system. Section 1.3 includes an in-depth look at techniques for specifying embedded systems; we use these specification techniques throughout the book. In Section 1.4, we use a model train controller as an example for applying these specification techniques. Section 1.5 provides a chapter-by-chapter tour of the book.

1.2 Complex systems and microprocessors

We tend to think of our laptop as a computer, but it is really one of many types of computer systems. A computer is a stored program machine that fetches and executes instructions from a memory device. We can attach different types of devices to the computer, load the memory with different types of software, and build many types of systems ranging from simple appliances to complex machines, like robots.

What is an **embedded computer system?** Loosely defined, it is any device that includes a programmable computer but is not itself intended to be a general-purpose computer. Thus a PC is not an embedded computing system. However, a thermometer built from a microprocessor is an embedded computing system.

This means that embedded computing system design is a useful skill for many types of products. Automobiles, medical devices, and even household appliances make extensive use of microprocessors. Designers in many fields must be able to identify where microprocessors can be used, design a hardware platform with I/O devices that can support the required tasks, and implement software that performs the required processing. Computer engineering, like mechanical design or thermodynamics, is a fundamental discipline that can be applied in many domains. However, embedded computing system design does not stand alone. Many of the challenges encountered in the design of an embedded computing system are not computer engineering. For example, they may be mechanical or analog electrical problems. In this book, we are primarily interested in the embedded computer itself; therefore we concentrate on the hardware and software that enable the desired functions in the final product.

1.2.1 Embedding computers

Microprocessors come in many levels of sophistication; they are usually classified by their word size. A **microcontroller** is a complete computer system on a chip: CPU, memory, I/O devices. An 8-bit microcontroller is designed for low-cost applications and includes on-board memory and I/O devices; a 16-bit microcontroller is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory; and a 32-bit or 64-bit reduced instruction set computer (**RISC**) microprocessor offers very high performance for computation-intensive applications.

Given the wide variety of microprocessor types available, it should be no surprise that microprocessors are used in many ways. There are many household uses of microprocessors and microcontrollers. The typical microwave oven has at least one microprocessor to control oven operation. Many houses have advanced thermostat systems that change the temperature level at various times during the day. The modern camera is a prime example of the powerful features that can be added under microprocessor control.

Digital televisions make extensive use of embedded processors. In some cases, specialized CPUs are designed to execute important algorithms, such as audio or video algorithms.

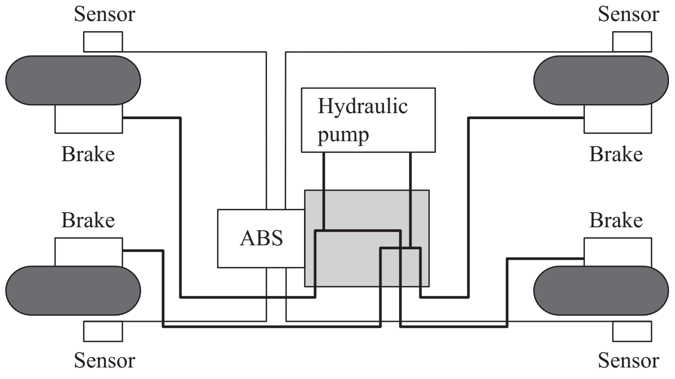
Many of today's cars operate with over 100 million lines of code [Zax12, Owe15]. The Ford F-150 operates with 150 million lines [Sar16]. A high-end automobile may have 100 microprocessors, but even inexpensive cars today use around 40. Some of these microprocessors do very simple things, such as detect whether seat belts are in use. Others control critical functions, such as ignition and braking systems. The big push toward microprocessor-based engine control came from two nearly simultaneous developments: The oil shock of the 1970s caused consumers to place much higher value on fuel economy, and fears of pollution resulted in laws restricting automobile engine emissions. The combination of low fuel consumption and low emissions is very difficult to achieve. To meet these goals without compromising engine performance, automobile manufacturers turned to sophisticated control algorithms that could be implemented only with microprocessors.

Braking is another important automotive subsystem that uses microprocessors to provide advanced capabilities. Design Example 1.1 describes some of the microprocessors used in BMW 850i.

Design Example 1.1: BMW 850i Brake and Stability Control System

The BMW 850i was introduced with a sophisticated system for controlling the wheels of the car. An antilock brake system (ABS) reduces skidding by pumping the brakes. An automatic stability control + traction (ASC+T) system intervenes with the engine during maneuvering to improve the car's stability. These systems actively control critical systems of the car; as control systems, they require inputs from and output to the automobile.

Consider the ABS. The purpose of an ABS is to temporarily release the brake on a wheel so that the wheel does not lose traction and skid. It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves, as shown in the accompanying diagram. This hookup allows the ABS system to modulate the brakes in order to keep the wheels from locking. The ABS system uses sensors on each wheel to measure its speed of rotation. The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent skidding.



The ASC+T system's job is to control the engine power and the brake to improve the car's stability during maneuvers. The ASC+T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting. The ASC+T can be turned off by the driver, which can be important when operating with tire snow chains.

The ABS and ASC+T must clearly communicate because the ASC+T interacts with the brake system. Since the ABS was introduced several years earlier than the ASC+T, it was important to be able to interface ASC+T to the existing ABS module and to other existing electronic modules. The engine and control management units include the electronically controlled throttle, digital engine management, and electronic transmission control. The ASC+T control unit has two microprocessors on two printed circuit boards: one of which concentrates on logic-relevant components and the other concentrates on performance-specific components.

1.2.2 Characteristics of embedded computing applications

Embedded computing is, in many ways, much more demanding than the sort of programs that you may have written for PCs or workstations. Functionality is important in both general-purpose and embedded computing, but embedded applications must meet many other constraints as well.

On the one hand, embedded computing systems must provide sophisticated functionality:

- *Complex algorithms:* The operations performed by the microprocessor may be very sophisticated. For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel consumption.
- *User interface*: Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options. The moving maps in Global Positioning System (GPS) navigation are good examples of sophisticated user interfaces.

To make things more difficult, embedded computing operations must often be performed to meet deadlines:

- *Real time*: Many embedded computing systems must perform in real time. Hence, if the data aren't ready by a certain deadline, the system breaks. In some cases, failure to meet a deadline is unsafe and can even endanger lives. In other cases, missing a deadline doesn't create safety problems but does create unhappy customers. Missed deadlines in printers, for example, can result in scrambled pages.
- *Multirate:* Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time. They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of **multirate** behavior. The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized. Failure to meet a deadline on either the audio or video portions spoils the perception of the entire presentation.

Constraints of various sorts are also very important:

- *Manufacturing cost*: The total cost of building a system is very important in many cases. Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required, and the types of I/O devices.
- *Power and energy*: Power consumption directly affects the cost of hardware, because a larger power supply may be necessary. Energy consumption affects battery life, which is important in many applications. Heat dissipation and thermal behavior are critical constraints in many computing systems.
- *Size*: Physical size is an important constraint in many systems.

Finally, most embedded computing systems are designed by small teams on tight deadlines. The use of small design teams for microprocessor-based systems is a self-fulfilling prophecy. The fact that systems can be built with microprocessors by only a few people invariably encourages management to assume that all microprocessor-based systems can be built by small teams. Tight deadlines are facts of life in today's internationally competitive environment. However, building a product using embedded software makes a lot of sense: Hardware and software can be debugged somewhat independently, and design revisions can be made much more quickly.

1.2.3 Why use microprocessors?

There are many ways to design a digital system: custom logic, field-programmable gate arrays (FPGAs), and so on. Why use microprocessors? There are two answers:

- Microprocessors are a very efficient way to implement digital systems.
- Microprocessors make it easier to design families of products that can be built to provide various feature sets at different price points and be extended to provide new features to keep up with rapidly changing markets.

CPUs are flexible

The paradox of digital design is that using a predesigned instruction set processor may in fact result in faster implementation of your application than designing your own custom logic. It is tempting to think that the overhead of fetching, decoding, and executing instructions is so high that it cannot be recouped.

CPUs are efficient

But there are two factors that work together to make microprocessor-based designs fast. First, microprocessors execute programs very efficiently. Modern processors can execute at least one instruction per clock cycle most of the time, and high-performance processors can execute several instructions per cycle. Although there is overhead that must be paid for interpreting instructions, it can often be hidden by clever utilization of parallelism within the CPU.

CPUs are highly optimized

Second, microprocessor manufacturers spend a great deal of money to make their CPUs run very fast. They hire large teams of designers to tweak every aspect of the microprocessor to make it run at the highest possible speed. Few products can justify the dozens or hundreds of computer architects and chip designers customarily employed in the design of a single microprocessor. Chips designed by small design teams are less likely to be as highly optimized for speed (or power) as are microprocessors. They also utilize the latest manufacturing technology.

It is also surprising but true that microprocessors are very efficient utilizers of logic. The generality of a microprocessor and the need for a separate memory may suggest that microprocessor-based designs are inherently much larger than custom logic designs. However, in many cases, the microprocessor is smaller when size is measured in units of logic gates. When special-purpose logic is designed for a particular function, it cannot be used for other functions. A microprocessor, on the other hand, can be used for many algorithms simply by changing the program it executes. Because so many modern systems make use of complex algorithms and user interfaces, we would generally have to design many custom logic blocks to implement all required functionality. Many of those blocks will often sit idle. For example, the processing logic may sit idle when user interface functions are performed. Implementing several functions on a single processor often makes much better use of the available hardware budget.

Programmability

Microprocessors provide substantial advantages and relatively small drawbacks. As a result, embedded computing is the best choice in a wide variety of systems. The programmability of microprocessors can be a substantial benefit during the design process. It allows program design to be separated (at least to some extent) from the design of the hardware on which programs will be run. While one team is designing the board that contains the microprocessor, I/O devices, memory, and so on, others can be writing programs.

Software as differentiator

Software is the principal differentiator in many products. In many product categories, competing products use one of a handful of chips as hardware platforms. Added software serves to differentiate these products by features.

How many platforms?

Why not use PCs for all embedded computing? Put another way, how many hardware **platforms** do we need for embedded computing systems? PCs are widely used and provide a very flexible programming environment. Components of PCs are, in fact, used in many embedded computing systems. However, several factors keep us from using the stock PC as a universal embedded computing platform: cost, physical size, and power consumption. A variety of embedded platforms has been developed for applications, such as automotive or motor control. Platforms, such as Arduino and Raspberry Pi, provide capable microcontrollers, a wide range of peripherals, and strong software development environments.

Real time

First, real-time performance requirements often drive us to different architectures. As we will see later in the book, real-time performance is often best achieved with multiprocessors. Heterogeneous multiprocessors are designed to match the characteristics of the application software that runs on them, providing performance improvements.

Low power and low cost

Second, low power and low cost also drive us away from PC architectures and toward multiprocessors. Personal computers are designed to satisfy a broad mix of computing requirements and to be very flexible. These features increase the complexity and price of the components. They also cause the processor and other components to use more energy to perform a given function. Custom embedded systems that are designed for an application, such as a cell phone, consume several orders of magnitude less power than PCs with equivalent computational performance, and they are considerably less expensive as well.

Physics of software

A central subject of this book is what we might call the **physics of software.** Computing is a physical act. Executing a program takes time and consumes energy. Software performance and energy consumption are very important properties when we are connecting our embedded computers to the real world. We need to understand the sources of performance and power consumption if we are to be able to design programs that meet our application's goals. Luckily, we don't need to optimize our programs by directly considering the flow of electrons in microelectronics. In many cases, we can make very high-level decisions about the structure of our programs to greatly improve their real-time performance and power consumption. As much as possible, we want to make computing abstractions work for us as we work on the physics of our software systems.

1.2.4 Embedded computing, IoT systems, and Cyber-Physical Systems

The discipline of embedded computing gives us a set of tools that we can use to design computer systems that interact with the physical world. Two styles of system design have emerged to support the wide range of applications in which computers interact with physical objects: Internet-of-Things (IoT) systems and cyber-physical systems (CPS).

IoT systems

An IoT system is a set of sensors, actuators, and computation units connected by a network. The network often has wireless links but may also include wired links. The IoT system monitors, analyzes, evaluates, and acts.

Cyber-Physical Systems

A cyber-physical system uses computers to build controllers, such as feedback control systems. A networked control system, in which a set of interfaces of a physical machine communicates over a bus with a CPU, is an important example of a cyber-physical system.

IoT and CPS

IoT and CPS are related but distinct, with a gray zone in between. One way to distinguish them is by sample rate: IoT systems tend to operate at lower sample rates, while cyber-physical systems run at higher sample rates. As a result, IoT systems are often more physically distributed, for example, a manufacturing plant; CPSs are more tightly coupled, such as an airplane or automobile.

Edge computing

IoT and CPSs are both examples of **edge computing**. When our computer must respond quickly to events in the physical world, we often don't have time to send queries to a remote data center and wait for a response. Computing at the edge must be responsive and energy efficient. Edge devices must also communicate with other devices at the edge as well as cloud computing resources.

1.2.5 Safety and security

Two trends make safety and security major concerns for embedded system designers. Embedded computers are increasingly found in safety-critical and other important systems that people use every day: cars, medical equipment, and so on. Many of these systems are connected to the Internet, either directly or indirectly, through maintenance devices or hosts. Connectivity makes embedded systems much more vulnerable to attack by malicious people. The danger of unsafe operation makes security problems even more important.

Security

Security relates to a system's ability to prevent malicious attacks. Security was originally applied to information processing systems, such as banking systems. In these cases, we are concerned with the data stored in the computer system. Stuxnet which we discuss in Section 7.6.1, shows that computer security vulnerabilities can open the door to attacks that can compromise the physical plant of a CPS, not just its information. Like dependability, security has several related concepts. **Integrity**

refers to the data's proper values; an attacker should not be able to change values. **Privacy** refers to the unauthorized release of data.

Safety

Safety relates to the way in which energy is released or controlled [Koo10]. Breaches in security can cause embedded computers to operate a physical machine improperly, causing a safety problem. Poor system design can also cause similar safety problems. Safety is a vital concern for any computer connected to physical devices, whether the threats come from malicious external activity, poor design, or improper use.

Safe and secure systems

Safety and security are related but distinct concepts [Wol18]. An insecure system may not necessarily present a safety problem, but the combination of these two dangers is particularly potent and is a novel aspect of embedded computing and CPSs. Security is traditionally associated with IT systems. A typical security breach does not directly endanger anyone's safety. Unauthorized disclosure of private information may, for example, allow the victim of a breach to be threatened, but a typical credit card database breach doesn't directly cause safety problems. Similarly, the safety of a traditional mechanical system is not directly related to any information about that system. Today, the situation is profoundly different; we need safe and secure systems. A poorly designed car can allow an attacker to install software and take over its operation. It may even cause the car to drive to a location at which the driver can be assaulted.

Safety and security technologies

We need to deploy a combination of technologies to ensure the safety and security of our embedded systems:

- **Cryptography** provides mathematical tools, such as encryption, which allow us to protect information. We discuss cryptography in Section 4.9.
- **Security protocols** make use of cryptography to provide functions, such as authenticating the source of a piece of software or the integrity of our system configuration. We discuss security protocols in Section 5.11.
- **Safe and secure hardware architectures** are required to ensure that our cryptographic operations and security protocols are not compromised by adversaries and that operational errors are caught early. We discuss safe and secure hardware architectures in Sections 3.8 and 4.9.

1.2.6 Challenges in embedded computing system design

We have high expectations of modern engineered devices and systems. Old-fashioned mechanical devices don't provide the user experience that is expected in the 21st century.

First, we expect our devices to be capable. They must be feature-rich, providing many options on what they can do and how to do them.

They must be very efficient. Battery-powered devices must go a long time between charges. They must be physically small, even though they are capable. They must be inexpensive, and they must be reliable. They must work as expected without failure for long periods. If they do break, they must do so safely, and they should adjust themselves as they operate to maintain their accuracy and effectiveness.

What barriers prevent us from taking advantage of the opportunities provided by microparocessors? In order to leverage the power of embedded computers, we must master several technical challenges.

Real-time and concurrent execution

Embedded computers and software need to operate in real time. The world doesn't stop for the computer to catch up. Computer hardware and software must be designed so that they respond promptly. Real-time operation often requires concurrency—the ability to perform several tasks at once. Concurrent software is difficult to design but necessary to support the natural concurrency of many mechanical systems.

Low-power embedded computing

Many embedded computer systems must also operate as low-power devices; they must efficiently use electrical energy. Battery-powered devices need to draw as little power from the battery as possible to maximize battery life. Even machines connected to the electrical grid must be efficient. Electric power costs money, and the heat generated by electric power consumption causes a new set of problems.

Safety and security

Embedded computing systems must operate safely and securely. Although information security is an important characteristic of all computer systems, the nature of embedded computing means that new types of security problems must be solved.

Fundamental techniques

Luckily, we have several tools at our disposal to help us solve these challenges and unlock the potential of embedded computing.

Performance analysis helps us to ensure that our embedded computer systems meet their real-time requirements. We can analyze our software and the computer hardware on which it runs to determine how long is required to execute the given function. If the software is too slow, we have tools at our disposal to help identify the problems and propose solutions.

Similarly, **power** analysis helps us ensure that our embedded computer systems operate at acceptable levels of power consumption.

Design methodology (the steps we follow to turn our ideas into finished products) are very important to the success of an embedded system design project. Methodologies help us refine a design, identify challenges, and identify good ways to solve those challenges.

1.2.7 Performance of embedded computing systems

Performance in general-purpose computing

When we talk about the performance of PC programs, what do we really mean? Most programmers have a vague notion of performance. They want their program to run fast enough, and they may be worried about the asymptotic complexity of their program. Most general-purpose programmers use no tools designed to help them improve the performance of their programs.

Performance in embedded computing

Embedded system designers, in contrast, have a very clear performance goal in mind; their program must meet its **deadline.** At the heart of embedded computing is **real-time computing**, which is the science and art of programming to deadlines. The program receives its input data, and the deadline is the time at which a computation must be finished. If the program doesn't produce the required output by the deadline, then the program doesn't work, even if the output that it eventually produces is functionally correct.

Understanding real-time performance

This notion of deadline-driven programming is at once simple and demanding. It isn't easy to determine whether a large, complex program running on a sophisticated microprocessor will meet its deadline. We need tools to help us analyze the real-time performance of embedded systems; we also must adopt programming disciplines and styles that make it possible to analyze these programs.

To understand the real-time behavior of an embedded computing system, we must analyze the system at several levels of abstraction. As we move through this book, we will work our way up from the lowest layers that describe components of the system up through the highest layers that describe the complete system. Those layers include:

- *CPU*: The CPU clearly influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.
- *Platform*: The platform includes the bus and I/O devices. The platform components that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.
- *Program*: The CPU sees only a small window of a program at any given time. We must consider the structure of the entire program to determine its overall behavior.
- *Task*: We generally run several programs simultaneously on a CPU, creating a **multitasking system.** The tasks interact with each other in ways that have profound implications for performance.
- *Multiprocessor*: Many embedded systems have more than one processor; they may include multiple programmable CPUs as well as accelerators. Once again, the interaction between these processors adds yet more complexity to the analysis of overall system performance.

1.3 The embedded system design process

This section provides an overview of the embedded system design process aimed at two objectives. First, it introduces the various steps in embedded system design before we delve into them in more detail. Second, it allows us to consider the design **methodology** itself. A design methodology is important for three reasons. First, it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as optimizing **performance** or functional testing. Second, it allows us to develop computer-aided design tools. Developing a single program that takes in a concept for an embedded system and produces a completed design would be a daunting task. However, by first breaking the process into manageable steps, we can work on automating (or at least semiautomating) the steps one at a time. Third, a design methodology makes it much easier for members of a design team to communicate. By defining the overall process, team members can more easily understand what they are supposed to do, what they should receive from other team members at certain times, and what they are to hand off when they complete their assigned steps. Because most embedded systems are designed by teams, coordination is perhaps the most important role of a well-defined design methodology.

Fig. 1.1 summarizes the major steps in the embedded system design process. In this top-down view, we start with the system **requirements analysis** to capture some basic elements of the system**.** In the next step, **specification**, we create a more detailed, complete description of what we want. However, the specification states only how the system behaves, not how it is built. The details of the system's internals begin to take shape when we develop the architecture, which gives the system structure in terms of large components. Once we know the components we need, we can design them, including both software modules and any specialized hardware. Based on those components, we can finally build a complete system.

In this section, we consider design from the **top down**. We begin with the most abstract description of the system and conclude with concrete details. The alternative is a **bottom-up** view in which we start with components to build a system. Bottom-up design steps are shown in Fig. 1.1 as dashed-line arrows. We need bottom-up design because we do not have perfect insight into how later stages of the design process will turn out. Decisions at one stage of design are based upon estimates of what will happen later: How fast can we make a particular function run? How much memory will we need? How much system bus capacity do we need? If our estimates are

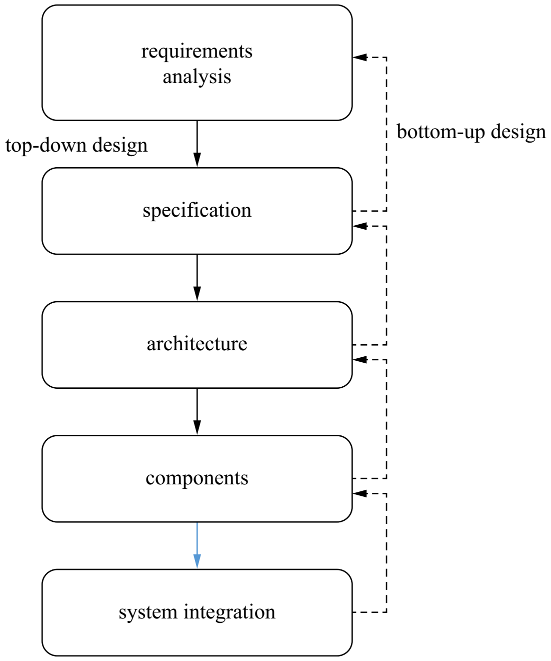


FIGURE 1.1

Major levels of abstraction in the design process.

inadequate, we may have to backtrack and amend our original decisions to take the new facts into account. In general, the less experience we have with the design of similar systems, the more we will have to rely on bottom-up design information to help us refine the system.

The steps in the design process are only one axis along which we can view embedded system design. We also need to consider the major goals of the design:

- manufacturing cost
- performance (throughput, latency, and deadlines)
- power consumption

We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail:

- We must *analyze* the design at each step to determine how we can meet the specifications.
- We must then *refine* the design to add detail.
- We must verify the design to ensure that it still meets all system goals, such as cost and speed.

1.3.1 Requirements

Clearly, before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components.

A requirement is a need for the system; a specification is a complete set of requirements for the system. We proceed in two phases. First, we gather an informal description from the customers in a process known as requirements analysis. We then flesh out the requirements into a specification and a complete description of the requirements that contains enough information to begin designing the system architecture.

Requirements and specifications

Separating requirements analysis and specification is often necessary owing to the large gap between what the customers can describe about the system they want and what the architects need to design the system. Consumers of embedded systems are usually not embedded systems or product designers. Their understanding of the system is based on how they envision user interactions with the system. They may have unrealistic expectations as to what can be done within their budgets, and they may also express their desires in a language very different from system architects' jargon. Capturing a consistent set of requirements from the customer and then massaging those requirements into a more formal specification is a structured way to manage the process of translating from the consumer's language to that of the designer's.

Requirements may be **functional** or **nonfunctional.** We must of course capture the basic functions of the embedded system, but functional description is often not sufficient. Typical nonfunctional requirements include:

- *Performance*: The speed of the system is often a major consideration both for the usability of the system and its ultimate cost. As we have noted, performance may be a combination of soft performance metrics, such as approximate time to perform a user-level function, and hard deadlines by which a particular operation must be completed.

- *Cost*: The target cost or purchase price of a system is almost always a consideration. Cost typically has two major components: **manufacturing cost**, which includes the cost of components and assembly, and **nonrecurring engineering (NRE)** costs, which include the personnel, tooling, and other costs of designing the system.
- *Physical size and weight*: The physical aspects of the final system can vary greatly, depending upon the application. An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict limitations on weight. A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.
- *Power consumption*: Power, of course, is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life. However, the customer is unlikely to be able to describe the allowable wattage.

Validating requirements

Validating a set of requirements is ultimately a psychological task because it requires understanding both what people want and how they communicate those needs. One good way to refine at least the user interface portion of a system's requirements is to build a **mock-up.** The mock-up may use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation. It should give the customer a good idea of how the system will be used and how the user can react to it. Physical, nonfunctional models of devices can also give customers a better idea of characteristics, such as size and weight.

Simple requirements form

Requirements analysis for big systems can be complex and time consuming. However, capturing a relatively small amount of information in a clear, simple format is a good start toward understanding system requirements. To introduce the discipline of requirements analysis as part of system design, we use a simple requirements methodology.

Fig. 1.2 shows a sample **requirements form** that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of a system. Let's consider the entries in the form:

- *Name*: This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people, but can also crystallize the purpose of the machine.
- *Purpose*: This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.
- *Inputs and outputs*: These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:
  - *Types of data*: Analog electronic signals? Digital data? Mechanical inputs?
  - *Data characteristics*: Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?

Name	GPS moving map
Purpose	
Inputs	
Outputs	
Functions	
Performance	
Manufacturing cost	
Power	
Physical size and weight	

FIGURE 1.2 Sample requirements form.

  - *Types of I/O devices*: Buttons? Analog/digital converters? Video displays?
  - *Functions*: This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?
- *Performance*: Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time frame. It is essential that the performance requirements be identified early because they must be carefully measured during implementation to ensure that the system works properly.
- *Manufacturing cost*: This includes primarily the cost of the hardware components. Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range. Cost has a substantial influence on architecture: A machine that is meant to sell at $25 most likely has a very different internal structure than a $1000 system.
- *Power*: Similarly, you may have only a rough idea of how much power the system can consume, but a little information can go a long way. Typically, the most important decision is whether the machine will be battery powered or plugged into the wall. Battery-powered machines must be much more careful about how they spend energy.
- *Physical size and weight*: You should give some indication of the physical size of the system to help guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel-mounted voice recorder.

A more thorough requirements analysis for a large system might use a form simlar to Fig. 1.2 as a summary of the longer requirements document. After an introductory section containing this form, a longer requirements document could include details on each of the items mentioned in the introduction. For example, each individual feature described in the introduction in a single sentence may be described in detail in a section of the specification.

Use cases

Another important means of understanding requirements is a set of **use cases**, which are descriptions of the system's use by actors. A use case describes how human users or other machines interact with the system. A basic use case diagram shows the set of possible actors—either people or other machines—and the operations they may perform. In some cases, cyber-physical systems may require a more thorough description of the use case, including a sequence diagram, which shows the series of operations that the user will perform. A set of use cases that describe typical usage scenarios often helps to clarify what the system needs to do. Analyzing the use cases for the system helps designers better understand its requirements.

Internal consistency of requirements

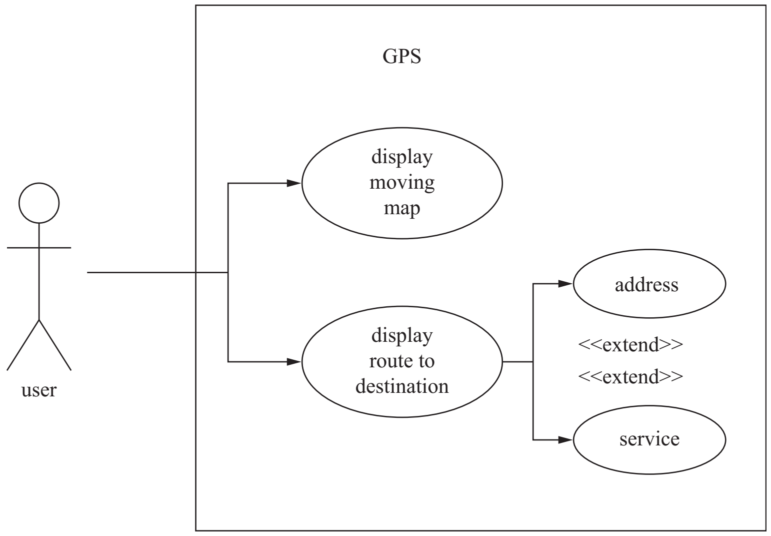
After writing the requirements, you should check them for internal consistency: Did you forget to assign a function to an input or output? Did you consider all the modes in which you want the system to operate? Did you place an unrealistic number of features into a battery-powered, low-cost machine?

To practice the capture of system requirements, Example 1.1 creates the requirements for a GPS moving map system.

Example 1.1: Requirements Analysis of a GPS Moving Map

The moving map is a small device, an alternative to a smartphone map, which displays for the user a map of the terrain around the user's current position; the map display changes as the user and the map device change position. The device can also show directions to a destination on the map. The moving map obtains its position from the GPS, a satellite-based navigation system. GPS moving maps are often used for hiking or as accessories for cars.

A simple use case diagram shows that the user may interact with the driver.



The user has two main ways to use the GPS unit: simply display a map that moves with position, and asking the device to show a route to a desired destination. That basic functionality can be extended in two ways: giving an address for the destination or selecting from a list of possible services (gas, restaurant, and so on).

What requirements might we have for our GPS moving map? Here is an initial list:

- *Functionality*: This system is designed for highway driving and similar uses, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.
- *User interface*: The screen should have at least 400 ×600 pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed, to allow the user to make selections to control the system.
- *Performance*: The map should scroll smoothly with screen updates at least 10 frames per second. Upon power-up, a display should take no more than 1 s to appear, and the system should be able to verify its position and display the current map within 15 s.
- *Cost*: The selling cost (street price) of the unit should be no more than $50. Selling price in turn helps to determine the allowable values for manufacturing cost. For example, if we use a rule-of-thumb in which the street cost is 4× that of the manufacturing cost, the cost to build this device should be no more than $12.50. Selling price also influences the amount of money that should be spent on nonrecurring engineering costs. High NRE costs will not be recoverable if the profit on each device is low.
- *Physical size and weight*: The device should fit comfortably in the palm of the hand.
- *Power consumption*: The device should run for at least 8 h on four AA batteries with at least 30 min of those 8 h comprising operation with the screen on.

Notice that many of these requirements are not specified in engineering units. For example, physical size is measured relative to a hand, not in centimeters. Although these requirements must ultimately be translated into something that can be used by the designers, keeping a record of what the customer wants can help resolve questions about the specification that may crop up later during design.

Based on this discussion, let's write a requirements chart for our moving map system.

Name	GPS moving map
Purpose	Consumer-grade moving map for driving use
Inputs	Power button, two control buttons
Outputs	Back-lit LCD display 400×600
Functions	Uses a five-receiver GPS system; three user-selectable resolutions; always displays current latitude and longitude
Performance	Updates screen within 0.25 s upon movement
Manufacturing cost	$12.50
Power	100 mW
Physical size and weight	No more than 2"× 6", 12 oz

This chart adds some requirements in engineering terms that will be of use to the designers. For example, it provides actual dimensions of the device. The manufacturing cost was derived from the selling price by using a simple rule-of-thumb: The selling price is four to five times the **cost of goods sold** (the total of all the component costs).

1.3.2 Specification

The specification is more precise and complete; it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it fully reflects the system's requirements and does so in a way that can be clearly followed during design.

Specification is probably the least familiar phase of this methodology for neophyte designers, but it is essential to creating working systems with a minimum of designer effort. Designers who lack a clear idea of what they want to build when they begin typically make faulty assumptions early in the process that aren't obvious until they have a working system. At that point, the only solution is to take the machine apart, throw away some of it, and start again. Not only does this take a lot of extra time, but the result is also very likely to be inelegant, kludgy, and bug-ridden.

The specification should be understandable enough so that someone can verify that it meets the system requirements and overall expectations of the customer. It should also be unambiguous enough that designers know what they need to build. Designers can run into several types of problems caused by unclear specifications. If the behavior of some feature in a particular situation is unclear from the specification, the designer may implement the wrong functionality. If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

A specification of the GPS system would include several components:

- data received from the GPS satellites;
- map data;
- user interface;
- operations that must be performed to satisfy customer requests;
- background actions required to keep the system running, such as operating the GPS receiver.

UML, a language for describing specifications, will be introduced in the next section. We will practice writing specifications in each chapter as we work through example system designs. We also study specification techniques in more detail in Chapter 7.

1.3.3 Architecture design

The specification does not say how the system does things, only what the system does. Describing how the system implements those functions is the purpose of the architecture. The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture. The creation of the architecture is the first phase of what many designers think of as design.

To understand what an architectural description is, let's look at a sample architecture for the moving map of Example 1.1. Fig. 1.3 shows a sample system architecture in the form of a **block diagram** that shows major operations and data flows among them. This block diagram is still quite abstract; we haven't yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on. The diagram does, however, go a long way toward describing how to implement the functions described in the specification. We clearly see, for example, that we need to search the topographic database and to render (draw) the results for the display. We have chosen to separate those functions so that we can potentially do them in parallel, which includes performing rendering separately from searching the database to help us update the screen more fluidly.

Only after we have designed an initial architecture that is not biased toward too many implementation details should we refine the system block diagram into two

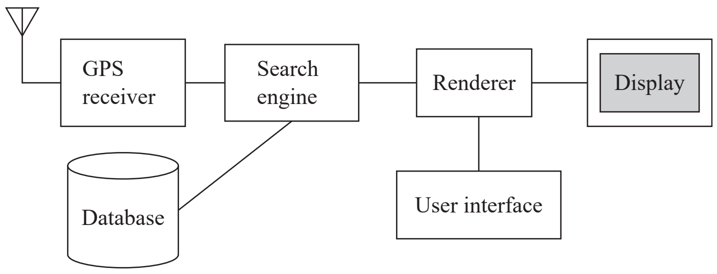


FIGURE 1.3 Block diagram for the moving map.

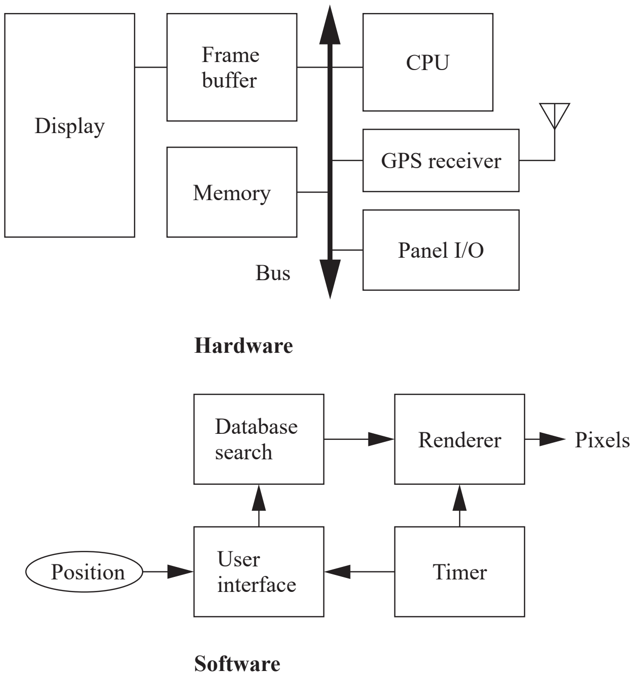


FIGURE 1.4 Hardware and software architectures for the moving map.

block diagrams: one for hardware and another for software. These refined block diagrams are shown in Fig. 1.4. The hardware block diagram clearly shows that we have one CPU surrounded by memory and I/O devices. In particular, we have chosen to use two memories: a frame buffer for the pixels to be displayed and a separate program/data memory for general use by the CPU. The software block diagram follows the system block diagram, but we have added a timer to control when we read the buttons on the user interface and render data onto the screen. To have a truly complete architectural description, we require more detail, such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed.

Architectural descriptions must be designed to satisfy both functional and nonfunctional requirements. Not only must all the required functions be present, but we must meet cost, speed, power, and other nonfunctional constraints. Starting out with a system architecture and refining that to hardware and software architectures is one good way to ensure that we meet all specifications: We can concentrate on the functional elements in the system block diagram and then consider the nonfunctional constraints when creating the hardware and software architectures.

How do we know that our hardware and software architectures in fact meet constraints on speed, cost, and so on? We must somehow be able to estimate the properties of the components of the block diagrams, such as the search and rendering functions in the moving map system. Accurate estimation derives in part from experience, both general design experience and particular experience with similar systems. However, we can sometimes create simplified models to help us make more accurate estimates. Sound estimates of all nonfunctional constraints during the architecture phase are crucial because decisions based on bad data will show up during the final phases of design, indicating that we did not, in fact, meet the specification.

1.3.4 Designing hardware and software components

The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification. The components will in general include both hardware—FPGAs, boards, and so on—and software modules.

Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other components. In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component. We can also make use of standard software modules. One good example is the topographic database. Standard topographic databases exist, and you probably want to use standard routines to access the database. Not only are the data in a predefined format, but they are highly compressed to save storage. Using standard software for these access functions not only saves design time, but it may also give us a faster implementation for specialized functions, such as the data decompression phase.

You will have to design some components yourself. Even if you are using only standard integrated circuits, you may be required to design the printed circuit board that connects them. You will probably have to do a lot of custom programming as well. When creating these embedded software modules, you must of course make use of your expertise to ensure that the system runs properly in real time and that it doesn't take up more memory space than is allowed. The power consumption of the moving map software example is particularly important. You may need to be very careful about how you read and write memory to minimize power. For example, because memory accesses are a major source of power consumption, memory transactions must be carefully planned to avoid reading the same data several times.

1.3.5 System integration

Only after the components are built do we have the satisfaction of putting them together and seeing a working system. Of course, this phase usually consists of a lot more than just plugging everything together and standing back. Bugs are typically found during system integration, and good planning can help us find bugs quickly. By building up the system in phases and running properly chosen tests, we can often find bugs more easily. If we debug only a few modules at a time, we are more likely to uncover simple bugs and will be able to easily recognize them. Only by fixing the simple bugs early will we be able to uncover the more complex or obscure bugs that can be identified only by giving the system a hard workout. We need to ensure during the architectural and component design phases that we make it as easy as possible to assemble the system in phases and test functions independently.

System integration is difficult because it usually uncovers problems. It is often hard to observe the system in sufficient detail to determine exactly what is wrong. The debugging facilities for embedded systems are usually much more limited than what you would find on desktop systems. As a result, determining why things don't work correctly and how they can be fixed is a challenge. Careful attention to inserting appropriate debugging facilities during design can help ease system integration problems, but the nature of embedded computing means that this phase will always be a challenge.

1.3.6 Formalisms for system design

As mentioned in the previous section, we perform a number of different design tasks at different levels of abstraction throughout this book: creating requirements and specifications, architecting the system, designing the code, and designing the tests. It is often helpful to conceptualize these tasks as diagrams. Luckily, there is a visual language that can be used to capture all these design tasks: the Unified Modeling Language [Boo99, Pil05]. UML was designed to be useful at many levels of abstraction in the design process. It is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction.

UML is an **object-oriented** modeling language. Object-oriented design emphasizes two concepts of importance:

- It encourages the design to be described as a number of interacting objects, rather than as a few large monolithic blocks of code.
- At least some of those objects will correspond to real pieces of software or hardware in the system. We can also use UML to model the outside world that interacts with our system, in which case the objects may correspond to people or other machines. It is sometimes important to implement something we think of at a high level as a single object using several distinct pieces of code or to otherwise break up the object correspondence in the implementation. However, thinking of the design in terms of actual objects helps us understand the natural structure of the system.

Object-oriented (often abbreviated as OO) specifications can be seen in two complementary ways:

- It allows a system to be described in a way that closely models real-world objects and their interactions.
- It provides a basic set of primitives that can be used to describe systems with particular attributes, irrespective of the relationships of those systems' components to real-world objects.

Object-oriented design vs. programming

Both views are useful. At a minimum, object-oriented specification is a set of linguistic mechanisms. In many cases, it is useful to describe a system in terms of real-world analogs. However, performance, cost, and so on may dictate that we change the specification to be different in some ways from the real-world elements we are trying to model and implement. In this case, the object-oriented specification mechanisms are still useful.

What is the relationship between an object-oriented specification and an object-oriented programming language, such as C++ [Str97]? A specification language may not be executable, but both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

UML is a large language, and covering all of it is beyond the scope of this book. In this section, we introduce only a few basic concepts. In later chapters, as we need a few more UML concepts, we introduce them to the basic modeling elements introduced here. Because UML is so rich, there are many graphical elements in a UML diagram. It is important to be careful to use the correct drawing to describe something. For instance, UML distinguishes between arrows with open and filled-in arrowheads, as well as solid and broken lines. As you become more familiar with the language, uses of the graphical primitives will become more natural to you.

We also won't take a strict object-oriented approach. We may not always use objects for certain elements of a design; in some cases, such as when taking particular aspects of the implementation into account, it may make sense to use another design style. However, object-oriented design is widely applicable, and no designer can be considered design literate without understanding it.

1.3.7 Structural description

By **structural descriptions** of the basic components of the system, we will learn how to describe how these components act in the next section. The principal component of an object-oriented design is, naturally enough, the **object.** An object includes a set of **attributes** that define its internal state. When implemented in a programming language, these attributes usually become variables or constants held in a data structure. In some cases, we will add the type of the attribute after the attribute name for clarity, but we do not always have to specify a type for an attribute. An object describing a display, such as a CRT screen, is shown in the UML notation in Fig. 1.5. The text in the folded-cornerpage icon is a **note;** it does not correspond to an object in the system and only serves as a comment. The attribute is, in this case, an array of pixels that hold the contents of the display. The object is identified in two ways: It has a unique name, and it is a member of a **class.** The name is underlined to show that this is a description of an object and not of a class.

Classes as types

A class is a form of type definition. All objects derived from the same class have the same attributes, although their attributes may have different values. A class also defines the **operations** that determine how the object interacts with the rest of the world. In a programming language, the operations would become pieces of code used to manipulate the object. The UML description of the *Display* class is shown in Fig. 1.6. The class has the name that we saw used in the *d1* object because *d1* is an instance of class *Display*. The *Display* class defines the *pixels* attribute seen in

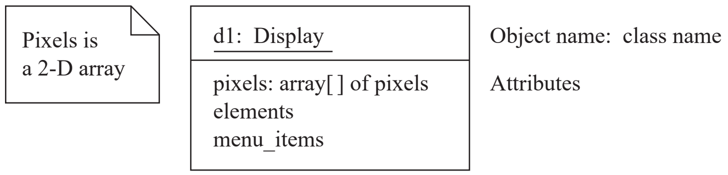


FIGURE 1.5 An object in UML notation.

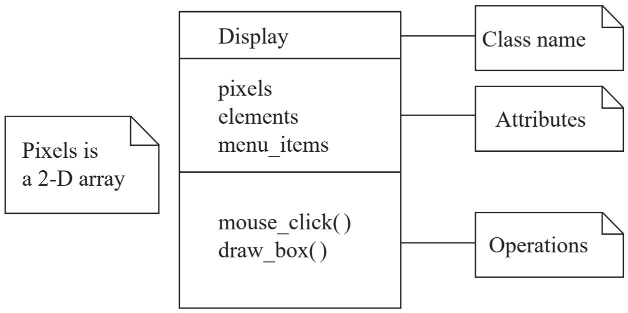


FIGURE 1.6 A class in UML notation.

the object. When we instantiate the class of an object, that object will have its own memory so that different objects of the same class have their own values for the attributes. Other classes can examine and modify class attributes; if we must do something more complex than use the attribute directly, we define a behavior to perform that function.

A class defines both the **interface** for a particular type of object and that object's **implementation.** When we use an object, we do not directly manipulate its attributes; we can only read or modify the object's state through the operations that define the interface to the object. The implementation includes both the attributes and whatever code is used to implement the operations. As long as we do not change the behavior of the object seen at the interface, we can change the implementation as much as we want. This lets us improve the system by, for example, speeding up an operation or reducing the amount of memory required without requiring changes to anything else that uses the object.

Choose your interface properly

Clearly, the choice of an interface is a very important decision in object-oriented design. The proper interface must provide ways to access the object's state and update the state because we cannot directly see the attributes. We need to make the object's interface general enough so that we can make full use of its capabilities. However, excessive generality often makes the object large and slow. Big, complex interfaces also make the class definition difficult for designers to understand and use properly.

There are several types of **relationships** that can exist between objects and classes:

- **Association** occurs between objects that communicate with each other but have no ownership relationship between them.

- **Aggregation** describes a complex object made of smaller objects.
- **Composition** is a type of aggregation in which the owner does not allow access to the component objects.
- **Generalization** allows us to define one class in terms of another.

The elements of a UML class or object do not necessarily directly correspond to statements in a programming language. If the UML is intended to describe something more abstract than a program, there may be a significant gap between the contents of the UML and a program implementing it. The attributes of an object do not necessarily reflect variables in the object. An attribute is some value that reflects the current state of the object. In the program implementation, that value could be computed from some other internal variables. The behaviors of the object would, at a higher-level specification, reflect the basic things that can be done with an object. Implementing all these features may require breaking up a behavior into several smaller behaviors. For example, you should initialize the object before you start to change its internal state.

Derived classes

UML, like most object-oriented languages, allows us to define one class in terms of another. An example is shown in Fig. 1.7, where we **derive** two types of displays. The first, *BW_display*, describes a black-and-white display. This does not require us to add new attributes or operations, but we can specialize both to work on one-bit pixels. The second, *Color_map_display*, uses a graphic device known as a color map to allow

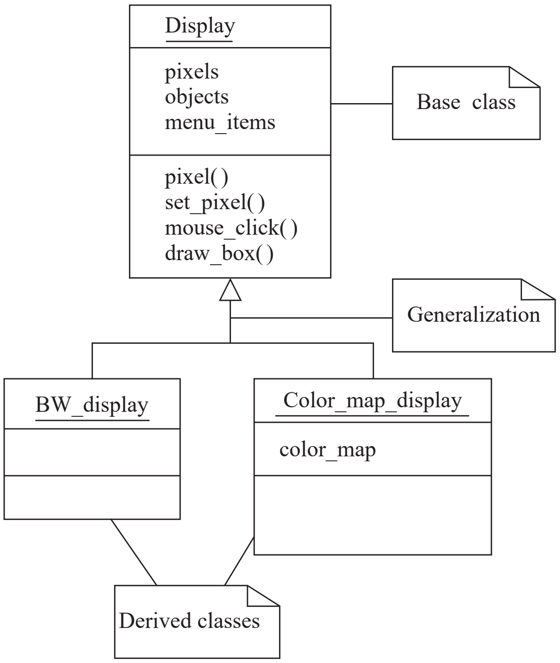


FIGURE 1.7 Derived classes as a form of generalization in UML notation.

the user to select from many available colors, even with a few bits per pixel. This class defines a *color_map* attribute that determines how pixel values are mapped onto display colors. A **derived class** inherits all the attributes and operations from its **base class.** In this class, *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class. This relation is transitive. If *Display* were derived from another class, both *BW_display* and *Color_map_display* would inherit all the attributes and operations of *Display*'s base class as well. Inheritance has two purposes. It, of course, allows us to succinctly describe one class that shares some characteristics with another class. More importantly, it captures those relationships between classes and documents them. If we ever need to change any of the classes, knowledge of the class structure helps us determine the reach of changes. For example, should the change affect only *Color_map_display* objects, or should it change all *Display* objects?

Generalization and inheritance

UML considers inheritance to be one form of generalization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead. Both *BW_display* and *Color_map_display* are specific versions of *Display*, so *Display* generalizes both. UML also allows us to define **multiple inheritance**, in which a class is derived from more than one base class. Most object-oriented programming languages support multiple inheritance as well. An example of multiple inheritance is shown in Fig. 1.8; we have omitted the details of the classes' attributes and operations for simplicity. In this case, we have created a *Multimedia_display* class by combining the *Display* class with a *Speaker* class for sound. The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*. Because multiple inheritance causes the sizes of the attribute set and operations to expand so quickly, it should be used with care.

A **link** describes a relationship between objects; association is to link as class is to object. We need links because objects often do not stand-alone; associations let us

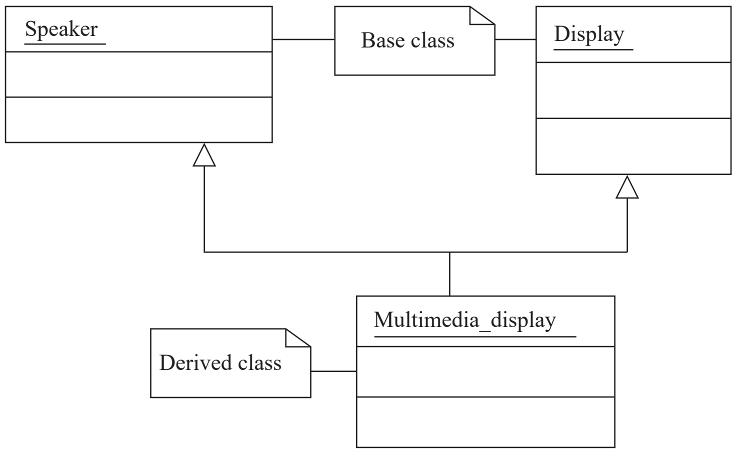


FIGURE 1.8 Multiple inheritance in UML notation.

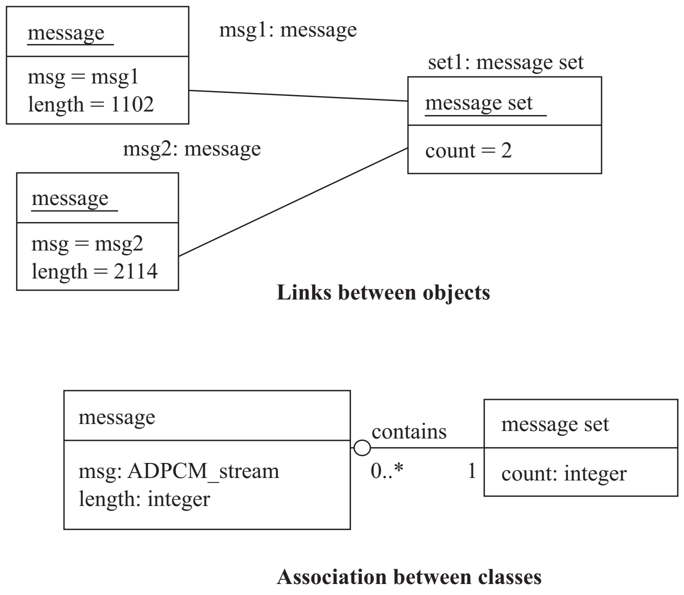


FIGURE 1.9 Links and associations.

capture type information about these links. Fig. 1.9 shows examples of links and an association. When we consider the actual objects in the system, there is a set of messages that keeps track of the current number of active messages (two in this example) and points to the active messages. In this case, the link defines the *contains* relation. When generalized into classes, we define an association between the message set class and the message class. The association is drawn as a line between the two labeled with the name of the association: *contains*. The ball and the number at the message class end indicate that the message set may include zero or more message objects. Sometimes we may want to attach data to the links themselves; we can specify this in the association by attaching a class-like box to the association's edge, which holds the association's data.

Typically, we find that we use a certain combination of elements in an object or class many times. We can give these patterns names, which are called **stereotypes** in UML. A stereotype name is written in the form ≪signal≫.

1.3.8 Behavioral description

We must specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a **state machine.** Fig. 1.10 shows the UML states; the transition between two states is shown by a skeleton arrow.

These state machines do not rely on the operation of a clock as in hardware; rather, changes from one state to another are triggered by the occurrence of **events.** An event

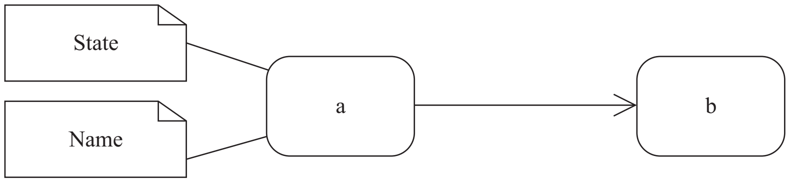


FIGURE 1.10

A state and transition in UML notation.

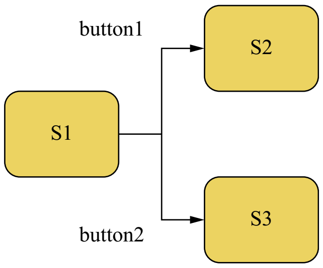


FIGURE 1.11 Events in UML state machines.

is some type of action. Fig. 1.11 illustrates this aspect of UML state machines: The machine transitions from S1 to S2 or S3 only when button1 or button2 is pressed. The event may originate outside the system, such as the button press. It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine.

UML defines several special types of events, as illustrated in Fig. 1.12:

- A **signal** is an asynchronous occurrence. It is defined in UML by an object that is labeled as a «signal». The object in the diagram serves as a declaration of the event's existence. Because it is an object, a signal may have parameters that are passed to the signal's receiver.
- A **call event** follows the model of a procedure call in a programming language.
- A **time-out event** causes the machine to leave a state after a certain amount of time. The label tm(time-value) on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism.

We show the occurrence of all types of signals in a UML diagram in the same way as a label on a transition.

Let's consider a simple state machine specification to understand the semantics of UML state machines. A state machine for the operation of the display is shown in Fig. 1.13. The start and stop states are special states that help us organize the flow of the state machine. The states in the state machine represent different conceptual

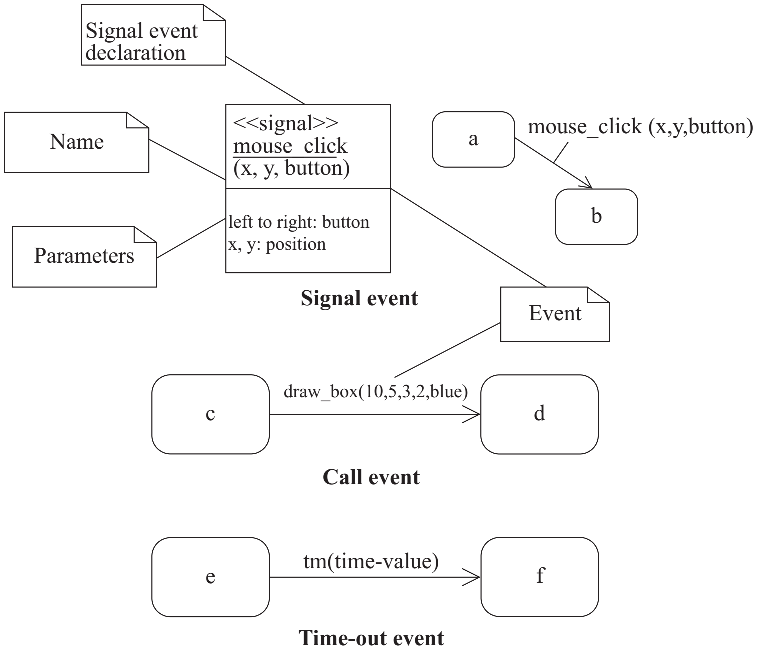


FIGURE 1.12 Signal, call, and time-out events in UML notation.

operations. In some cases, we take conditional transitions out of states based on inputs or the results of some computation done in the state. In other cases, we make an unconditional transition to the next state. Both the unconditional and conditional transitions make use of the call event. Splitting a complex operation into several

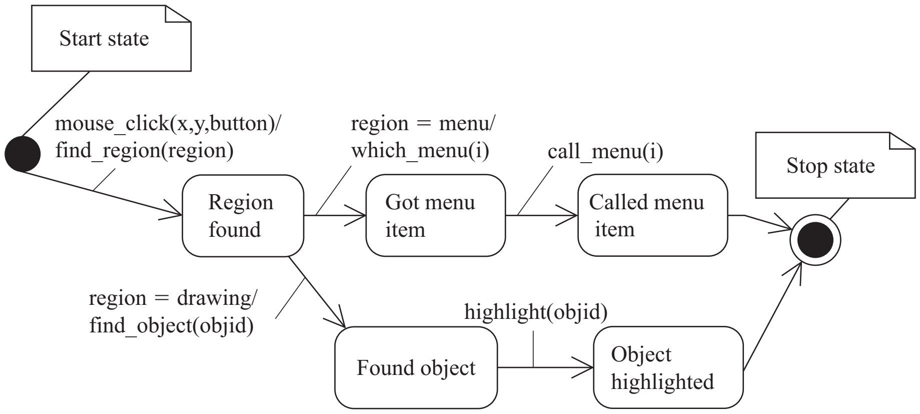


FIGURE 1.13 A state machine specification in UML notation.

states helps document the required steps, much as subroutines can be used to structure code.

It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved. In this case, we can create a **sequence diagram**, which is often used to describe use cases. A sequence diagram is like a hardware timing diagram, although the time flows vertically in a sequence diagram, whereas time typically flows horizontally in a timing diagram. The sequence diagram is designed to show a particular scenario or choice of events; it is not convenient for showing several mutually exclusive possibilities.

An example of a mouse click and its associated actions is shown in Fig. 1.14. The mouse click occurs on the menu region. Processing includes three objects shown at the top of the diagram. Extending below each object is its *lifeline*, a dashed line that shows how long the object is alive. In this case, all objects remain alive for the entire sequence, but in other cases, objects may be created or destroyed during processing. The boxes along the lifelines show the *focus of control* in the sequence when the object is actively processing. In this case, the mouse object is active only long enough to create the *mouse_click* event. The display object remains in play longer; it then uses call events to invoke the menu object twice: once to determine which menu item was selected and again to execute the menu call. The find_region() call is internal to the display object, so it does not appear as an event in the diagram.

1.4 Design example: Model train controller

In order to learn how to use UML to model systems, we specify a simple system, a **model train controller**. Model trains run on tracks and often represent scale models of full-sized trains. Simple trains either run at a constant speed or use a simple speed control. Modern model trains can make use of digital train controllers that send

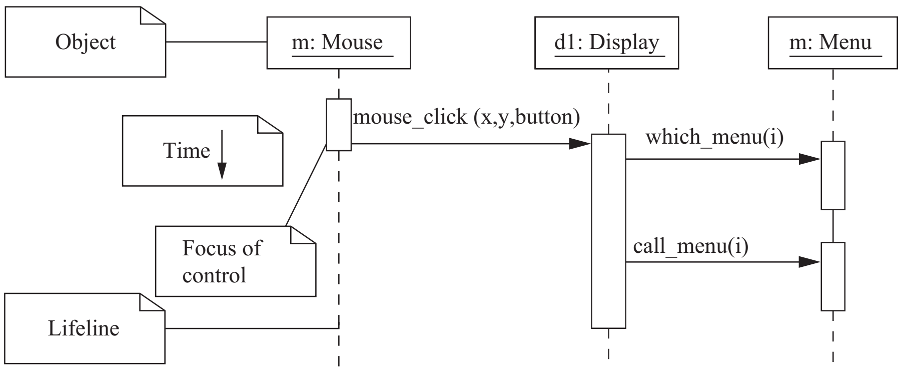


FIGURE 1.14 A sequence diagram in UML notation.

messages along the tracks to the train engine. Train controllers not only allow sophisticated control of the train—various speeds and reverse—but they also allow several trains to run on the track at different speeds. The train controller allows the hobbyist to recreate much more sophisticated train environments.

Fig. 1.15 shows a train controller in the context of a model train layout. Several trains may be on the track at the same time, each with its own address. The controller may also control other parts of the layout, such as switches that select which track a train will follow at the switch point. The user sends messages to a train through a control box attached to the tracks. The control box may have familiar controls, such as a throttle, an emergency stop button, and so on. Because the train receives its electrical power from the two rails of the track, the control box can send signals to the train over the tracks by modulating the power-supply voltage. As shown in the figure, the control panel sends packets over the tracks to the receiver on the train. The train includes analog electronics to sense the bits being transmitted and a control system to set the train motor's speed and direction based on those commands. Each packet includes an address so that the console can control several trains on the same track. The packet also includes an error correction code (ECC) to guard against transmission errors. This is a one-way communication system; the model train cannot send status or acknowledgments back to the user.

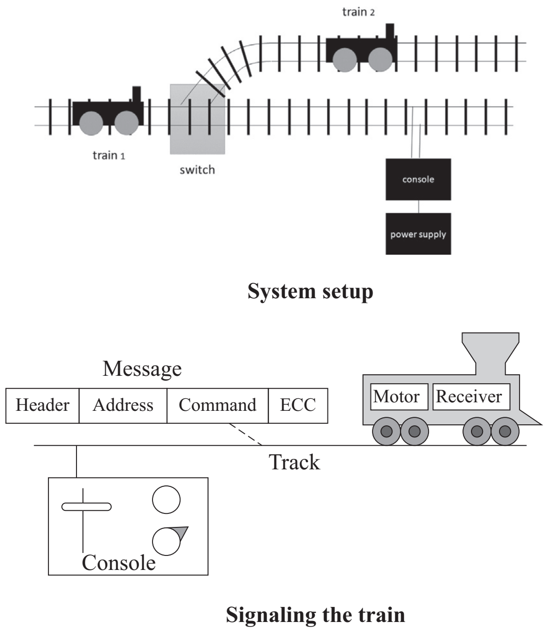


FIGURE 1.15 A model train control system.

We start by analyzing the requirements for the train control system. We base our system on a real standard developed for model trains. We then develop two specifications: a simple, high-level specification and a more detailed one.

1.4.1 Requirements

Before we can create a system specification, we should understand the requirements. Here is a basic set of requirements for the system:

- The console shall be able to control up to eight trains on a single track.
- The speed of each train shall be controllable by a throttle to at least 63 different levels in each direction (forward and reverse).
- There shall be an inertia control that shall allow the user to adjust the responsiveness of the train to commanded changes in speed. Higher inertia means that the train responds more slowly to a change in the throttle, simulating the inertia of a large train. The inertia control will provide at least eight different levels.
- There shall be an emergency stop button.
- An error detection scheme will be used to transmit messages.

We can put the requirements into our chart format:

Name	Model train controller
Purpose	Control speed of up to eight model trains
Inputs	Throttle, inertia setting, emergency stop, train number
Outputs	Train control signals
Functions	Set engine speed based upon inertia settings; respond to emergency stop
Performance	Can update train speed at least 10 times per second
Manufacturing cost	$50
Power	10 W (plugs into wall)
Physical size and weight	Console should be comfortable for two hands, approximate size of standard keyboard; weight less than 2 lb.

We will develop our system using a widely used standard for model train control. We could develop our own train control system from scratch, but basing our system upon a standard has several advantages in this case. It reduces the amount of work we must do, and it allows us to use a wide variety of existing trains and other pieces of equipment.

1.4.2 Digital Command Control (DCC)

The DCC standard (http://www.nmra.org/standards/DCC/standards_rps/DCCStds.html) was created by the National Model Railroad Association to support interoperable digitally controlled model trains. Hobbyists started building homebrew digital control systems in the 1970s, and Marklin developed its own digital control system in the 1980s. DCC was created to provide a standard that could be built by any manufacturer so that hobbyists could mix and match components from multiple vendors.

DCC documents

The DCC standard is given in two documents:

- Standard S-9.1, the DCC Electrical Standard, defines how bits are encoded on the rails for transmission.
- Standard S-9.2, the DCC Communication Standard, defines the packets that carry information.

Any DCC-conforming device must meet these specifications. DCC also provides several recommended practices. These are not strictly required, but they provide some hints to manufacturers and users as to how to best use DCC.

The DCC standard does not specify many aspects of a DCC train system. It doesn't define the control panel, the type of microprocessor used, the programming language to be used, or many other aspects of a real model train system. The standard concentrates on those aspects of system design that are necessary for interoperability. Overstandardization, or specifying elements that don't really need to be standardized, only makes the standard less attractive and harder to implement.

DCC Electrical Standard

The Electrical Standard deals with voltages and currents on the track. Although the electrical engineering aspects of this part of the specification are beyond the scope of the book, we briefly discuss the data encoding here. The standard must be carefully designed because the main function of the track is to carry power to the locomotives. The signal encoding system should not interfere with power transmission either to DCC or non-DCC locomotives. A key requirement is that the data signal should not change the average power supply voltage on the train's rails.

The data signal swings between two voltages around the power-supply voltage. As shown in Fig. 1.16, bits are encoded in the time between transitions, not by voltage levels. A zero is at least 100 s, whereas a one is nominally 58 s. The specification

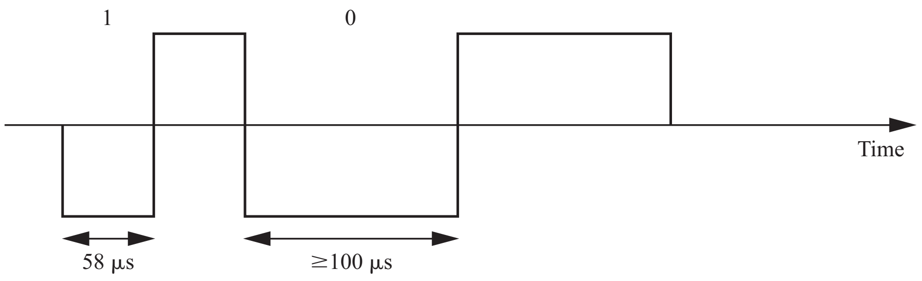


FIGURE 1.16 Bit encoding in digital command and control.

also gives the allowable variations in bit times that a conforming DCC receiver must be able to tolerate.

The standard also describes other electrical properties of the system, such as allowable transition times for signals.

DCC Communication Standard

The DCC Communication Standard describes how bits are combined into packets and the meaning of some important packets. Some packet types are left undefined in the standard, but typical uses are given in the recommended practices documents.

We can write the basic packet format as a regular expression:

$$PSA(sD) + E \qquad \text{(Eq. 1.1)}$$

In this regular expression,

- P is the preamble, which is a sequence of at least ten 1 bits. The command station should send at least 14 of these 1 bits, some of which may be corrupted during transmission.
- S is the packet start bit. It is a 0 bit.
- A is an address data byte that gives the address of the unit, with the most significant bit of the address transmitted first. An address is 8 bits long. The addresses, 00000000, 11111110, and 11111111, are reserved.
- s is the data byte start bit, which, like the packet start bit, is 0.
- D is the data byte, which includes 8 bits. A data byte may contain an address, instruction data, or error correction information.
- E is a packet end bit, which is a 1 bit.

A packet includes one or more data byte start bit/data byte combinations. Note that the address data byte is a specific type of data byte.

Baseline packet

A **baseline packet** is the minimum packet that must be accepted by all DCC implementations. More complex packets are given in a recommended practice document. A baseline packet has three data bytes: An address data byte gives the intended receiver of the packet; the instruction data byte provides a basic instruction; and an error correction data byte is used to detect and correct transmission errors.

The instruction data byte carries several pieces of information. Bits 0—3 provide a 4-bit speed value. Bit 4 has an additional speed bit, which is interpreted as the least significant speed bit. Bit 5 gives direction, with one for forward and zero for reverse. Bits 7—8 are set as 01 to indicate that this instruction provides speed and direction.

The error correction data byte is the bitwise exclusive OR of the address and instruction data bytes.

Packet transmission rates

The standard says that the command unit should send packets frequently because a packet may be corrupted. Packets should be separated by at least 5 ms.

1.4.3 Conceptual specification

DCC specifies some important aspects of the system, particularly those that allow equipment to interoperate. However, DCC deliberately does not specify everything about a model train control system. We must round out our specification with details that complement the DCC spec. A **conceptual specification** allows us to understand the system a little better. We use the experience gained by writing the conceptual specification to help us write a detailed specification to be given to a system architect. This specification doesn't correspond to what any commercial DCC controllers do, but it is simple enough to allow us to cover some basic concepts in system design.

Commands

A train control system turns **commands** into **packets.** A command comes from the command unit while a packet is transmitted over the rails. Commands and packets may not be generated in a 1:1 ratio. In fact, the DCC standard says that command units should resend packets in case a packet is dropped during transmission. Fig. 1.17 shows a generic command class and several specific commands derived from that base class. *Estop* (emergency stop) requires no parameters, whereas *Set-speed* and *Set-inertia* do.

We now need to model the train control system itself. There are clearly two major subsystems: the command unit and the train-board component. Each of these subsystems has its own internal structure. The basic relationship between them is illustrated in Fig. 1.18. This figure shows a UML **collaboration diagram**. We could have used another type of figure, such as a class or object diagram, but we want to emphasize the transmit/receive relationship between these major subsystems. The command unit and receiver are each represented by objects; the command unit sends a sequence of packets to the train's receiver, as illustrated by the arrow. The notation on the arrow provides both the type of message sent and its sequence in a flow of messages;

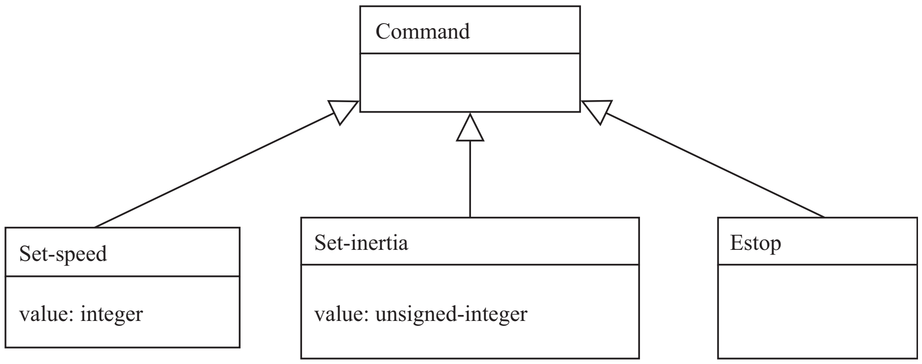


FIGURE 1.17 Class diagram for the train controller commands.

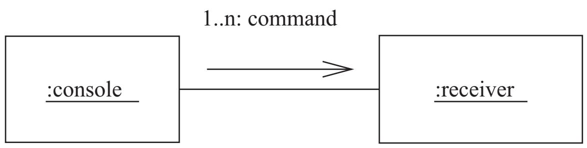


FIGURE 1.18 UML collaboration diagram for major subsystems of the train controller system.

because the console sends all the messages, we have numbered the arrow's messages as 1…n. Those messages are, of course, carried over the track. Because the track isn't a computer component and is purely passive, it does not appear in the diagram. However, it would be perfectly legitimate to model the track in the collaboration diagram, and in some situations, it may be wise to model such nontraditional components. For example, if we are worried about what happens when the track breaks, modeling the tracks would help us identify failure modes and recovery mechanisms.

Let's break down the command unit and receiver into their major components. The console needs to perform three functions: read the state of the front panel on the command unit, format messages, and transmit messages. The train receiver must also perform three major functions: receive the message, interpret the message (taking into account the current speed, inertia setting, and so on), as well as actually controlling the motor. In this case, let's use a class diagram to represent the design; we could also use an object diagram if we wished. The UML class diagram is shown in Fig. 1.19. It shows the console class using three classes, one for each of its major components. These classes must define some behaviors, but for the moment, we concentrate on the basic characteristics of these classes:

- The *Console* class describes the command unit's front panel, which contains the analog knobs and hardware to interface to the digital parts of the system.

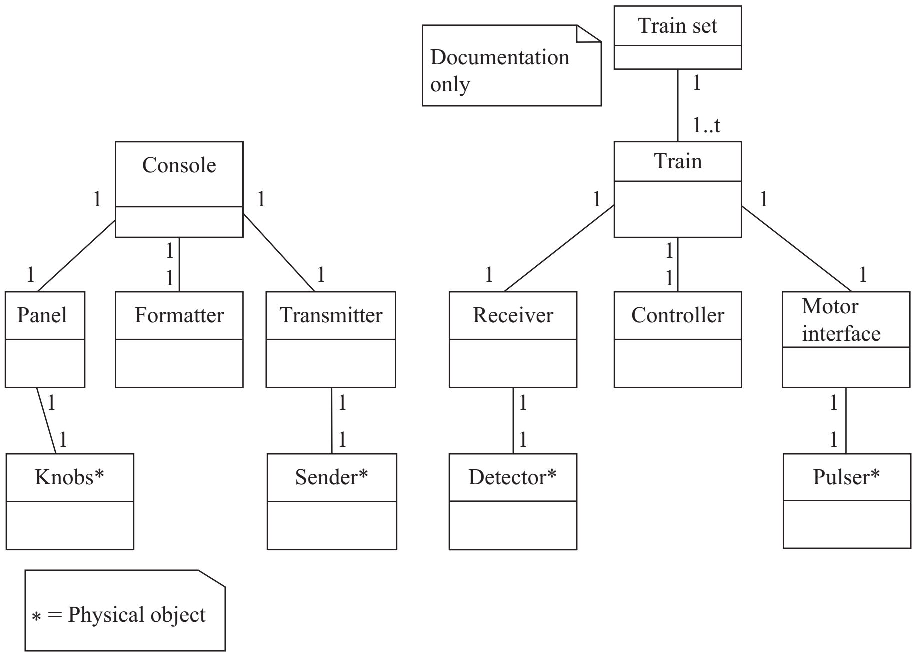


FIGURE 1.19 A UML class diagram for the train controller showing the composition of the subsystems.

- The *Formatter* class includes behaviors that know how to read the panel knobs and create a bit stream for the required message.
- The *Transmitter* class interfaces analog electronics to send the message along the track.

There will be one instance of the *Console* class and one instance of each of the component classes, as shown by the numeric values at each end of the relationship links. We also show some special classes that represent analog components, ending the name of each with an asterisk:

- *Knobs** describes the actual analog knobs, buttons, and levers on the control panel.
- *Sender** describes the analog electronics that send bits along the track.

Likewise, the *Train* makes use of three other classes that define its components:

- The *Receiver* class knows how to turn the analog signals on the track into digital form.
- The *Controller* class includes behaviors that interpret the commands and figure out how to control the motor.
- The *Motor interface* class defines how to generate the analog signals required to control the motor.

We define two classes to represent analog components:

- *Detector** detects analog signals on the track and converts them into digital form.
- *Pulser** turns digital commands into the analog signals required to control the motor speed.

We also define a special class, *Train set*, to help us remember that the system can handle multiple trains. The values on the relationship edge show that one train set can have *t* trains. We would not actually implement the train set class, but it does serve as useful documentation of the existence of multiple receivers.

Fig. 1.20 shows a sequence diagram for the simple use of DCC with two trains. The console is first set to select train 1 and then to set the speed. The console is then set to select train 2 and set its speed.

1.4.4 Detailed specification

Now that we have a conceptual specification that defines the basic classes, let's refine it to create a more detailed specification. We won't create a complete specification, but we will add detail to the classes and look at some of the major decisions in the specification process to get a better handle on how to write good specifications.

At this point, we need to define the analog components in a little more detail because their characteristics will strongly influence the *Formatter* and *Controller*. Fig. 1.21 shows a class diagram for these classes; this diagram shows a little more detail than Fig. 1.19 because it includes attributes and behaviors of these classes. The *Panel* has three knobs: *train* number (which train is currently being controlled), *speed* (which can be positive or negative), and *inertia*. It also has one button for

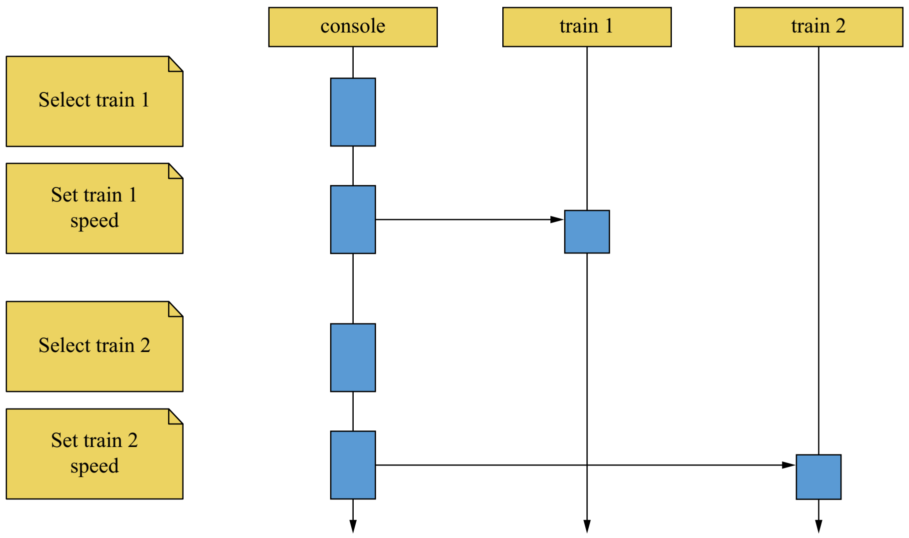


FIGURE 1.20 Use case for setting speed of two trains.

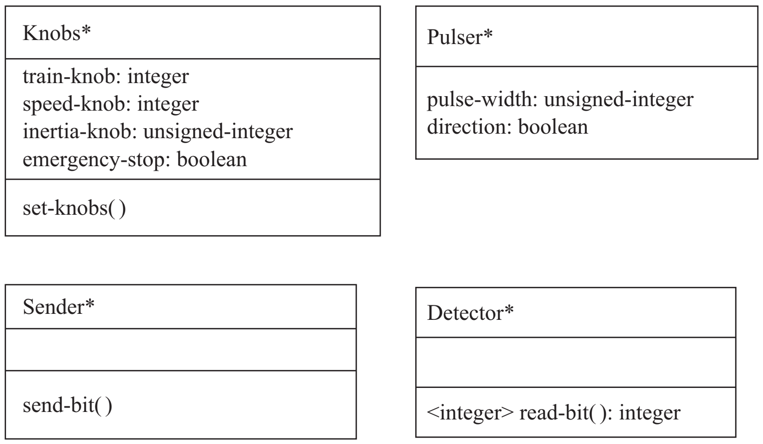


FIGURE 1.21 Classes describing analog physical objects in the train control system.

emergency-stop. When we change the train number setting, we also want to reset the other controls to the proper values for that train so that the previous train's control settings are not used to change the current train's settings. To do this, *Knobs* must provide a *set-knobs* behavior that allows the rest of the system to modify the knob settings. If we wanted or needed to model the user, we would expand on this class definition to provide methods that a user object would call to specify these parameters. The motor system takes its motor commands in two parts. The *Sender* and *Detector* classes are relatively simple: They simply put out and pick up a bit, respectively.

To understand the *Pulser* class, let's consider how we control the train motor's speed. As shown in Fig. 1.22, the speed of electric motors is commonly controlled using pulse-width modulation: Power is applied in a pulse for a fraction of some fixed interval, with the fraction of the time that power is applied determining the speed. The digital interface to the motor system specifies that pulse width is an integer, with the maximum value being maximum engine speed. A separate binary value controls direction. Note that the motor control takes an unsigned speed with a separate direction, while the panel specifies speed as a signed integer, with negative speeds corresponding to reverse.

Fig. 1.23 shows the classes for the panel and motor interfaces. These classes form the software interfaces to their respective physical devices. The *Panel* class defines a behavior for each of the controls on the panel. We have chosen not to define an internal variable for each control because their values can be read directly from the physical device, but a given implementation may choose to use internal variables. The *new-settings* behavior uses the *set-knobs* behavior of the *Knobs** class to change the knobs settings whenever the train number setting is changed. *Motor-interface* defines an attribute for speed that can be set by other classes. As we will see in a moment, the controller's job is to incrementally adjust the motor's speed to provide smooth acceleration and deceleration.

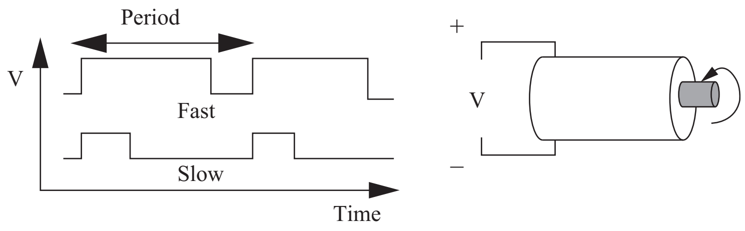


FIGURE 1.22 Controlling motor speed by pulse-width modulation.

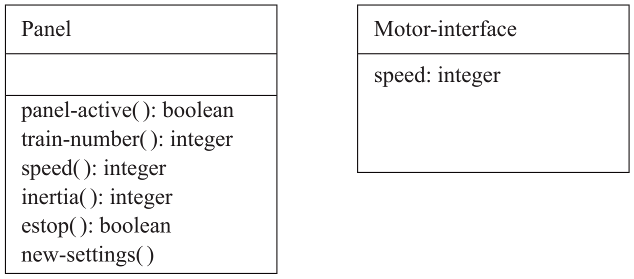


FIGURE 1.23

Class diagram for the panel and motor interface.

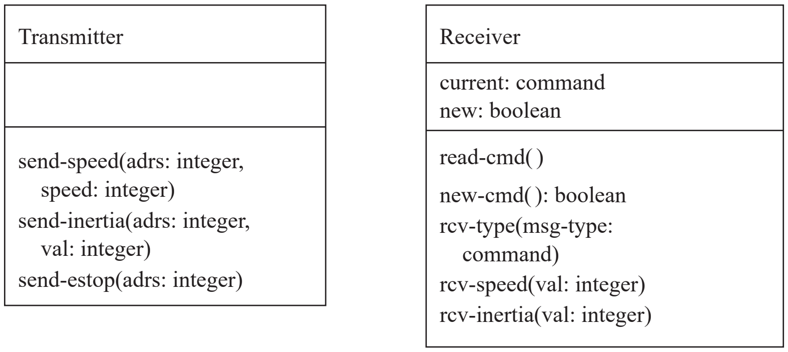

Transmitter
send-speed(adrs: integer, speed: integer)
send-inertia(adrs: integer, val: integer)
send-estop(adrs: integer)
Receiver
current: command
new: boolean
read-cmd( )
new-cmd( ): boolean
rcv-type(msg-type: command)
rcv-speed(val: integer)
rcv-inertia(val: integer)

FIGURE 1.24 Class diagram for the *Transmitter* and *Receiver.*

The *Transmitter* and *Receiver* classes are shown in Fig. 1.24. They provide the software interface to the physical devices that send and receive bits along the track. The *Transmitter* provides a distinct behavior for each type of message that can be sent. It internally takes care of formatting the message. The *Receiver* class provides a *read-cmd* behavior to read a message off the tracks. We can assume for now that the receiver object allows this behavior to run continuously to monitor the tracks and intercept the next command. (We consider how to model such continuously running behavior as processes in Chapter 6.) We use an internal variable, *current*, to hold the current command. Another variable, *new*, holds a flag showing when the command has been processed. Separate behaviors let us read out the parameters for each type of command; these messages also reset the new flag to show that the command has been processed. We don't need a separate behavior for an *Estop* message because it has no parameters; knowing the type of message is sufficient.

Now that we have specified the subsystems around the formatter and controller, it is easier to see what sorts of interfaces these two subsystems may need.

The *Formatter* class is shown in Fig. 1.25. The formatter holds the current control settings for all the trains. The *send-command* method is a utility function that serves as the interface to the transmitter. The *operate* function performs the basic actions for the object. At this point, we only need a simple specification, which states that the formatter repeatedly reads the panel, determines whether any settings have changed, and sends out the appropriate messages. The *panel-active* behavior returns true whenever the panel's values do not correspond to the current values.

The role of the formatter during the panel's operation is illustrated by the sequence diagram of Fig. 1.26. The figure shows two changes to the knob settings: first to the throttle, inertia, or emergency stop and then to the train number. The panel is called periodically by the formatter to determine if any control settings have changed. If a setting has changed for the current train, the formatter decides to send a command, issuing a *send-command* behavior to cause the transmitter to send the bits. Because transmission is serial, it takes a noticeable amount of time for the transmitter to finish a command; in the meantime, the formatter continues to check the panel's control

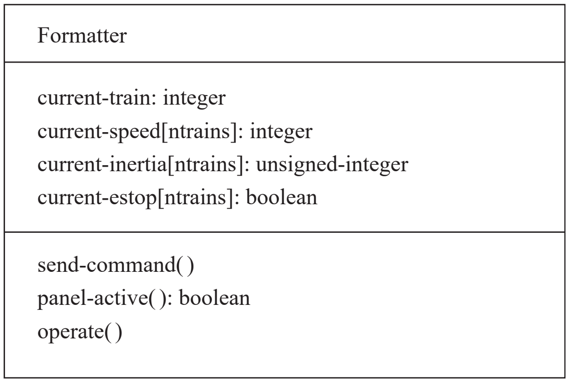

Formatter
current-train: integer
current-speed[ntrains]: integer
current-inertia[ntrains]: unsigned-integer
current-estop[ntrains]: boolean
send-command( )
panel-active( ): boolean
operate( )

FIGURE 1.25 Class diagram for the *Formatter* class.

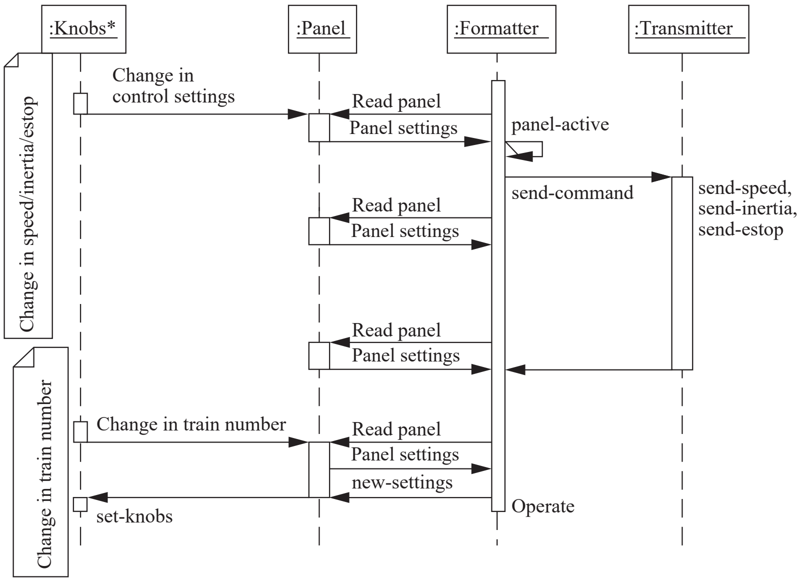


FIGURE 1.26 Sequence diagram for transmitting a control input.

settings. If the train number has changed, the formatter must cause the knob settings to be reset to the proper values for the new train.

We have not yet specified the operation of any of the behaviors. We define what a behavior does by writing a state diagram. The state diagram for a very simple version of the *operate* behavior of the *Formatter* class is shown in Fig. 1.27. This behavior watches the panel for activity: If the train number changes, it updates the panel display; otherwise, it causes the required message to be sent. Fig. 1.28 shows a state diagram for the *panel-active* behavior.

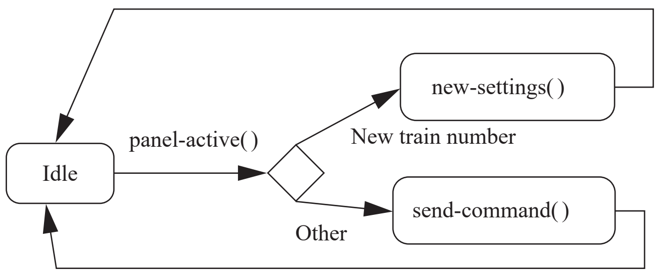


FIGURE 1.27 State diagram for the formatter *operate* behavior.

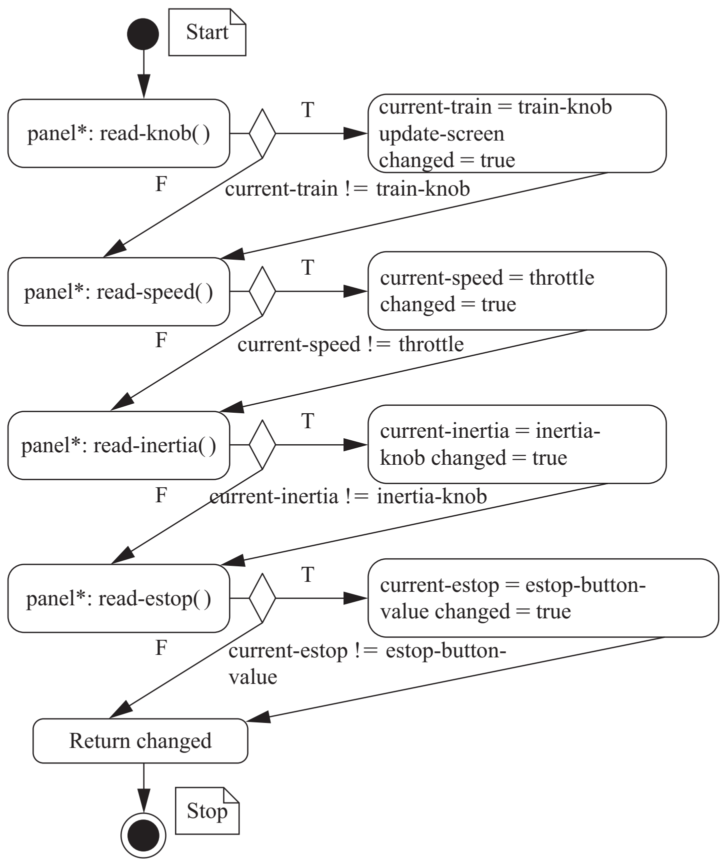

Start
panel*: read-knob( )
T
current-train = train-knob
update-screen
changed = true
F
current-train != train-knob
panel*: read-speed( )
T
current-speed = throttle
changed = true
F
current-speed != throttle
panel*: read-inertia( )
T
current-inertia = inertia-knob changed = true
F
current-inertia != inertia-knob
panel*: read-estop( )
T
current-estop = estop-button-value changed = true
F
current-estop != estop-button-value
Return changed
Stop

FIGURE 1.28 State diagram for the panel-activate behavior.

The definition of the train's *Controller* class is shown in Fig. 1.29. The *operate* behavior is called by the receiver when it gets a new command; *operate* looks at the contents of the message and uses the *issue-command* behavior to change the speed,

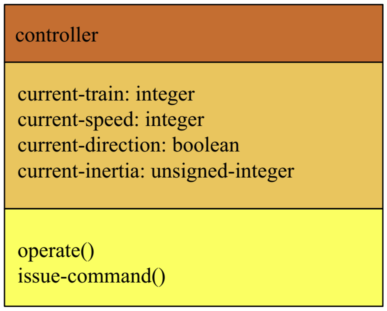


FIGURE 1.29 Class diagram for the *Controller* class.

direction, and inertia settings, as necessary. A specification for *operate* is shown in Fig. 1.30.

The operation of the *Controller* class during the reception of a *set-speed* command is illustrated in Fig. 1.31. The *Controller*'s *operate* behavior must execute several behaviors to determine the nature of the message. Once the speed command has been parsed, it must send a sequence of commands to the motor to smoothly change the train's speed.

Refining command classes

It is also a good idea to refine our notion of a command. These changes result from the need to build a potentially upward-compatible system. If the messages were entirely internal, we would have more freedom in specifying messages that we could use during architectural design. However, because these messages must work with a variety of trains and we may want to add more commands in a later version of the system, we need to specify the basic features of messages for compatibility. There are three important issues. First, we need to specify the number of bits used to determine the message type. We chose three bits because that gives us five unused message codes. Second, we need to include information about the length of the data fields, which is determined by the resolution for speeds and inertia set by the requirements. Third,

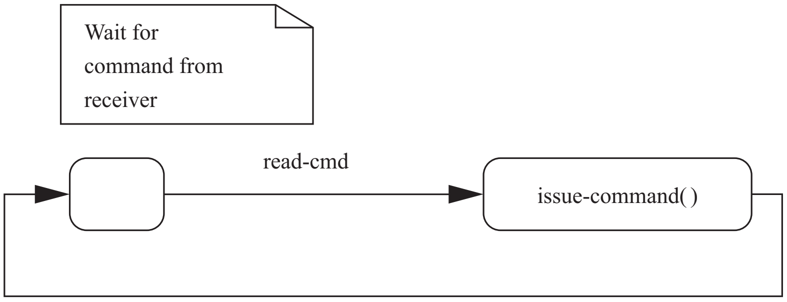


FIGURE 1.30 State diagram for the *Controller*'s *operate* behavior.

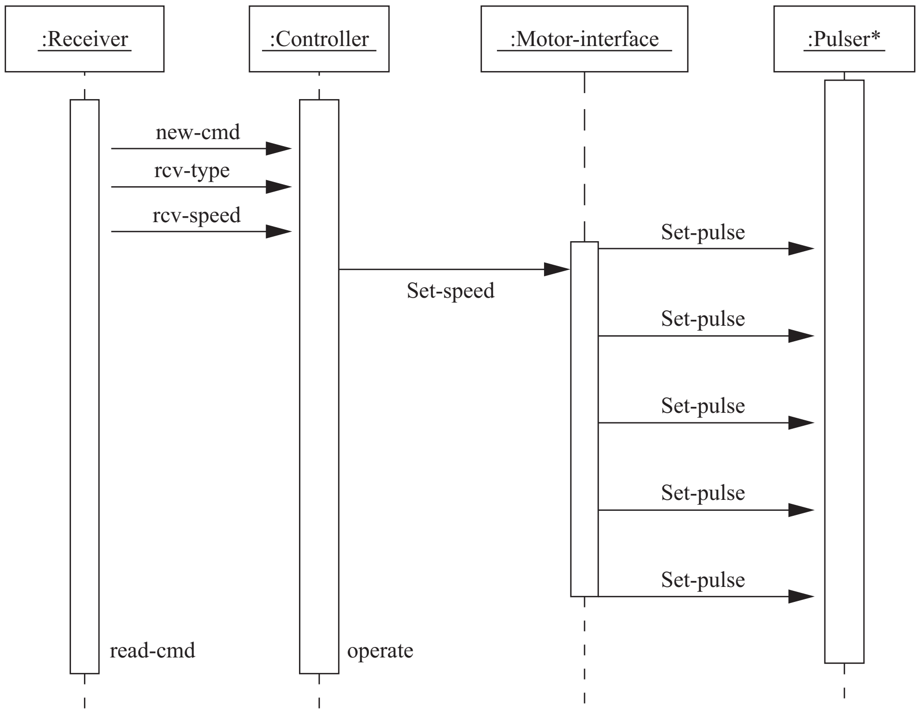


FIGURE 1.31 Sequence diagram for a *set-speed* command received by the train.

we need to specify the error detection mechanism; we choose to use a single-parity bit. We can update the classes to provide this extra information, as shown in Fig. 1.32.

1.4.5 Lessons learned

This example illustrates some general concepts. First, standards are important. We often cannot avoid working with standards, but standards often save us work and allow us to make use of components designed by others. Second, specifying a system isn't easy. You often learn a lot about the system you are trying to build by writing a specification. Third, specification invariably requires making some choices that may influence the implementation. Good system designers use their experience and intuition to guide them when these kinds of choices must be made.

1.5 A guided tour of this book

The most efficient way to learn all the necessary concepts is to move from the bottom up. This book is arranged so that you learn about the properties of components and build toward more complex systems and a more complete view of the system design

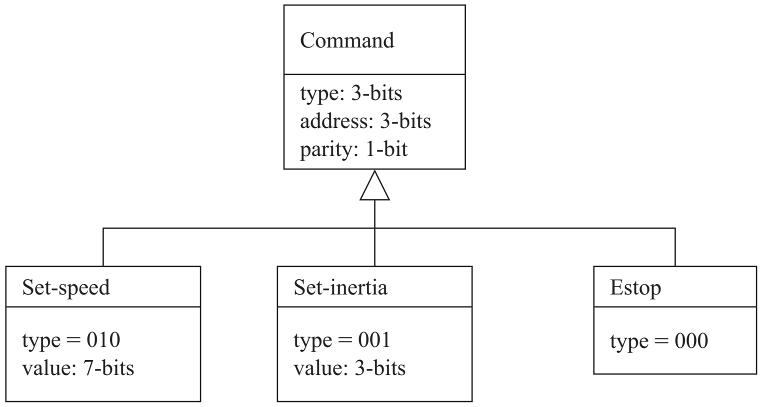


FIGURE 1.32

Refined class diagram for the train controller commands.

process. Veteran designers have learned enough bottom-up knowledge from experience to know how to use a top-down approach to designing a system, but when learning things for the first time, the bottom-up approach allows you to build more sophisticated concepts based on lower-level ideas.

We use several organizational devices throughout the book to help you. Application Examples focus on a particular end-use application and how it relates to embedded system design. We also make use of Programming Examples to describe software designs. In addition to these, most chapters use significant system design examples to demonstrate the major concepts of the chapter.

Each chapter includes questions that are intended to be answered on paper as homework assignments. The chapters also include lab exercises. These are more open ended and are intended to suggest activities that can be performed in the lab to help illuminate various concepts in the chapter.

Throughout the book, we use several CPUs as examples: the advanced RISC machine (ARM) processor, the Texas Instruments C55x digital signal processor (DSP), the PIC16F, and the Texas Instruments C64x. All are well-known microprocessors used in many embedded applications. Using real microprocessors helps make concepts more concrete. However, our aim is to learn concepts that can be applied to all sorts of microprocessors. Although microprocessors will evolve over time (Warhol's Law of Computer Architecture [Wol92] states that every microprocessor architecture will be the price/performance leader for 15 min), the concepts of embedded system design are fundamental and long term.

1.5.1 Chapter 2: Instruction sets

In Chapter 2, we begin our study of microprocessors by concentrating on **instruction sets.** The chapter covers the instruction sets of the ARM, Microchip PIC16F, TI C55x, and TI C64x microprocessors in separate sections. All these microprocessors are very different. Understanding all details of both is not strictly necessary to the design of embedded systems. However, comparing them does provide some interesting lessons in instruction set architectures.

Understanding some aspects of the instruction set is important both for concreteness and for seeing how architectural features can affect performance and other system attributes. However, many mechanisms, such as caches and memory management, can be understood in general before we go on to detail how they are implemented in these three microprocessors.

We do not introduce a design example in this chapter; it is difficult to build even a simple working system without understanding other aspects of the CPU that will be introduced in Chapter 3. However, understanding instruction sets helps us to understand problems, such as execution speed and code size, that we study throughout the book.

1.5.2 Chapter 3: CPUs

Chapter 3 rounds out our discussion of microprocessors by focusing on the following important mechanisms that are not part of the instruction set itself:

- We introduce the fundamental mechanisms of **input** and **output**, including interrupts.
- We also study the **cache** and **memory management unit.**

We also begin to consider how CPU hardware affects important characteristics of program execution. Program performance and power consumption are very important parameters in embedded system design. An understanding of how architectural aspects, such as pipelining and caching affect these system characteristics is a foundation for analyzing and optimizing programs in later chapters.

Our study of program performance begins with instruction-level performance. The basics of pipeline and cache timing serve as the foundation for our studies of larger program units.

We use as an example a simple data compression unit, concentrating on the programming of the core compression algorithm.

1.5.3 Chapter 4: Computing platforms

Chapter 4 looks at the combined hardware and software platform for embedded computing. The microprocessor is very important, but only part of a system that includes memory, I/O devices, and low-level software. We need to understand the basic characteristics of the platform before we move on to build sophisticated systems.

The basic embedded computing platform includes a microprocessor, I/O hardware, I/O driver software, and memory. Application-specific software and hardware can be added to this platform to turn it into an embedded computing platform. The microprocessor is at the center of both the hardware and software structure of the embedded computing system. The CPU controls the bus that connects to memory and I/O devices; the CPU also runs software that talks to the devices. In particular, I/O is central to embedded computing. Many aspects of I/O are not typically studied in modern computer architecture courses, so we need to master the basic concepts of input and output before we can design embedded systems.

Chapter 4 covers several important aspects of the platform:

- We study in detail how the CPU talks to memory and devices using the microprocessor **bus.**
- Based on our knowledge of bus operation, we study the structure of the **memory system** and types of **memory components.**
- We look at basic techniques for embedded system **design** and **debugging.**
- We study system-level performance analysis to see how bus and memory transactions affect the execution time of systems.

We illustrate these principles using two design examples. We use an alarm clock as a simple example of an appliance built on an embedded computing platform. We consider a jet engine controller as a more sophisticated example of a microprocessor plus bus used as an embedded computing platform.

1.5.4 Chapter 5: Program design and analysis

Chapter 5 moves on to the software side of the computer system to understand how complex sequences of operations are executed as programs. Given the challenges of embedded programming—meeting strict performance goals, minimizing program size, reducing power consumption—this is an especially important topic. We build upon the fundamentals of computer architecture to understand how to design embedded programs.

- We develop some basic software components—data structures and their associated routines—that are useful in embedded software.
- As part of our study of the relationship between programs and instructions, we introduce a model for high-level language programs known as the **control/data flow graph (CDFG).** We use the CDFG model extensively to help us analyze and optimize programs.
- Because embedded programs are increasingly written in higher-level languages, we look at the processes for compiling, assembling, and linking to understand how high-level language programs are translated into instructions and data. Some of the discussion surveys basic techniques for translating high-level language programs. We also spend time on compilation techniques designed specifically to meet embedded system challenges.
- We develop techniques for the **performance analysis** of programs. It is difficult to determine the speed of a program simply by examining its source code. We learn how to use a combination of the source code, its assembly language

implementation, and expected data inputs to analyze program execution time. We also study some basic techniques for optimizing program performance.

- An important topic related to performance analysis is **power analysis.** We build on performance analysis methods to learn how to estimate the power consumption of programs.
- It is critical that the programs that we design function correctly. The CDFG and techniques we have learned for performance analysis are related to techniques for **testing programs.** We develop techniques that can methodically develop a set of tests for a program in order to excise bugs.

At this point, we can consider the performance of a complete program. We introduce the concept of worst-case execution time as a basic measure of program execution time.

We use two design examples in Chapter 5. The first is a simple software modem. A modem translates between the digital world of the microprocessor and the analog transmission scheme of the telephone network. Rather than using analog electronics to build a modem, we can use a microprocessor and special-purpose software. Because the modem has strict real-time deadlines, this example lets us exercise our knowledge of the microprocessor and of program analysis. The second is a digital still camera, which is considerably more complex than the modem, both in the algorithms it must execute and the variety of tasks it must perform.

1.5.5 Chapter 6: Processes and operating systems

Chapter 6 builds on our knowledge of programs to study a special type of software component, the **process**, and operating systems that use processes to create systems. A process is an execution of a program; an embedded system may have several processes running concurrently. A separate **real-time operating system (RTOS)** controls when the processes run on the CPU. Processes are important to embedded system design because they help us juggle multiple events happening at the same time. A real-time embedded system that is designed without processes usually ends up as a mess of spaghetti code that doesn't operate properly.

We study the basic concepts of processes and process-based design in this chapter:

- We begin by introducing the **process abstraction.** A process is defined by a combination of the program being executed and the current state of the program. We learn how to switch contexts between processes.
- In order to make use of processes, we must be able to **schedule** them. We discuss process priorities and how they can be used to guide scheduling.
- We cover the fundamentals of **interprocess communication**, including the various styles of communication and how they can be implemented. We also look at the various uses of these interprocess communication mechanisms in systems.
- The real-time operating system (RTOS) is the software component that implements process abstraction and scheduling. We study how RTOSs implement schedules, how programs interface to the operating system, and how we can

evaluate the performance of systems built from RTOSs. We also survey some examples of RTOSs.

Tasks introduce a new level of complexity to performance analysis. Our study of real-time scheduling provides an important foundation for the study of multitasking systems.

Chapter 6 analyzes an engine control unit for an automobile. This unit must control the fuel injectors and spark plugs of an engine based upon a set of inputs. Relatively complex formulas govern its behavior, all of which must be evaluated in real time. The deadlines for these tasks vary over several orders of magnitude.

1.5.6 Chapter 7: System design techniques

Chapter 7 studies the design of large, complex embedded systems. We introduce important concepts that are essential for the successful completion of large embedded system projects, and we use those techniques to help us integrate the knowledge obtained throughout the book.

This chapter delves into several topics related to large-scale embedded system design:

- We revisit the topic of **design methodologies.** Based on our more detailed knowledge of embedded system design, we can better understand the role of methodology and the possible variations in methodologies.
- We study system **requirements analysis and specification methods.** Proper specifications become increasingly important as system complexity grows. More formal specification techniques help us capture intent clearly, consistently, and unambiguously. System analysis methodologies provide a framework for understanding specifications and assessing their completeness.
- We look at some **system modeling** methods that allow us to capture a design at several levels of abstraction.
- We consider **system analysis** and the **design of architectures** that meet our functional and nonfunctional requirements.
- We look at **quality assurance** techniques. The program testing techniques covered in Chapter 5 are a good foundation but may not scale easily to complex systems. Additional methods are required to ensure that we exercise complex systems to shake out bugs.
- We consider **safety** and **security** as aspects of **dependability.**

1.5.7 Chapter 8: Internet-of-Things

The Internet-of-Things has emerged as a key application area for embedded computing. We look at the range of applications covered by IoT. We study wireless networks used to connect to IoT devices. We review the basics of databases that are used to tie together devices into IoT systems. As a design example, we consider a smart home with a variety of sensors.

1.5.9 Chapter 9: Automotive and aerospace systems

Automobiles and airplanes are important examples of networked control systems and of safety-critical embedded systems. We will see how cars and airplanes are operated using multiple networks and many communicating processors of various types. We look at the structure of some important networks used in distributed embedded computing, such as the controller area network widely used in cars and the inter-integrated circuit network for consumer electronics. We consider safety and security in vehicles.

1.5.10 Chapter 10: Embedded multiprocessors

The final chapter is an advanced topic that may be left out of initial studies and introductory classes without losing the basic principles underlying embedded computing. Many embedded systems are multiprocessors—computer systems with more than one processing element. The multiprocessor may use CPUs and DSPs; it may also include nonprogrammable elements known as **accelerators.** Multiprocessors are often more energy efficient and less expensive than platforms that try to do all the required computing on one big CPU. We look at the architectures of multiprocessors as well as the programming challenges of single-chip multiprocessors.

The chapter analyzes an accelerator for use in a video compression system. Digital video requires performing a huge number of operations in real time; video also requires large volumes of data transfers. As such, it provides a good way to study not only the design of the accelerator itself, but also how it fits into the overall system.

1.6 Summary

Embedded microprocessors are everywhere. Microprocessors allow sophisticated algorithms and user interfaces to be added inexpensively to an amazing variety of products. Microprocessors also help reduce design complexity and time by separating out hardware and software design. Embedded system design is much more complex than programming PCs because we must meet multiple design constraints, including performance, cost, and so on. In the remainder of this book, we build a set of techniques from the bottom up that will allow us to conceive, design, and implement sophisticated microprocessor-based systems.

What we learned

- Embedded computing can be fun. It can also be difficult thanks to the combination of complex functionality and strict constraints that we must satisfy.
- Trying to hack together a complex embedded system probably won't work. You need to master a number of skills and understand the design process.

- Your system must meet certain functional requirements, such as features. It may also have to perform tasks to meet deadlines, limit its power consumption, be of a certain size, or meet other nonfunctional requirements.
- A hierarchical design process takes the design through several levels of abstraction. You may need to do both top-down and bottom-up design.
- We use UML to describe designs at several levels of abstraction.
- This book takes a bottom-up view of embedded system design.

Further reading

Koopman [Koo10] describes in detail the phases of embedded computing system development. Spasov [Spa99] describes how 68HC11 microcontrollers are used in Canon EOS cameras. Douglass [Dou98] gives a good introduction to UML for embedded systems. Other foundational books on object-oriented design include Rumbaugh et al. [Rum91], Booch [Boo91], Shlaer and Mellor [Shl92], and Selic et al. [Sel94]. Bruce Schneier's book *Applied Cryptography* [Sch96] is an outstanding reference on the field.

Questions

Q1-1 Briefly describe the distinction between requirements and specifications.

Q1-2 Give an example of a requirement on a smart voice command speaker.

Q1-3 Give an example of a requirement on a smartphone camera.

Q1-4 How could a security breach on a commercial airliner's Wi-Fi network result in a safety problem for the airplane?

Q1-5 Given an example of a specification on a smart speaker, giving both type of specification and any required values. Take your example from an existing product and identify that product.

Q1-6 Given an example of a specification on a smartphone camera, giving both type of specification and any required values. Take your example from an existing product and identify that product.

Q1-7 Briefly describe the distinction between specification and architecture.

Q1-8 At what stage of the design methodology would we determine what type of CPU to use?

Q1-9 At what stage of the design methodology would we choose a programming language?

Q1-10 Should an embedded computing system include software designed in more than one programming language? Justify your answer.

Q1-11 At what stage of the design methodology would we test our design for functional correctness?

Q1-12 Compare and contrast top-down and bottom-up design.

Q1-13 Give an example of a design problem that is best solved using top-down techniques.

Q1-14 Give an example of a design problem that is best solved using bottom-up techniques.

Q1-15 Provide a concrete example of how bottom-up information from the software programming phase of design may be useful in refining the architectural design.

Q1-16 Give a concrete example of how bottom-up information from I/O device hardware design may be useful in refining the architectural design.

Q1-17 Create a UML state diagram for the *issue-command()* behavior of the *Controller* class of Fig. 1.27.

Q1-18 Show how a *Set-speed* command flows through the refined class structure described in Fig. 1.18, moving from a change on the front panel to the required changes on the train:
a. Show it in the form of a collaboration diagram.
b. Show it in the form of a sequence diagram.

Q1-19 Show how a *Set-inertia* command flows through the refined class structure described in Fig. 1.18, moving from a change on the front panel to the required changes on the train:
a. Show it in the form of a collaboration diagram.
b. Show it in the form of a sequence diagram.

Q1-20 Show how an *Estop* command flows through the refined class structure described in Fig. 1.18, moving from a change on the front panel to the required changes on the train:
a. Show it in the form of a collaboration diagram.
b. Show it in the form of a sequence diagram.

Q1-21 Draw a state diagram for a behavior that sends the command bits on the track. The machine should generate the address, generate the correct message type, include the parameters, and generate the ECC.

Q1-22 Draw a state diagram for a behavior that parses the bits received by the train. The machine should check the address, determine the message type, read the parameters, and check the ECC.

Q1-23 Draw a state diagram for the train receiver class.

Q1-24 Draw a class diagram for the classes required in a basic microwave oven. The system should be able to set the microwave power level between one and nine and time a cooking run up to 59 min and 59 s in 1-s increments. Include classes for the physical interfaces to the front panel, door latch, and microwave unit.

Q1-25 Draw a collaboration diagram for the microwave oven of Q1-23. The diagram should show the flow of messages when the user first sets the power level to seven, then sets the timer to 2:30, and then runs the oven.

Lab exercises

L1-1 How would you measure the execution speed of a program running on a microprocessor? You may not always have a system clock available to measure time. To experiment, write a piece of code that performs some function that takes a small but measurable amount of time, such as a matrix algebra function. Compile and load the code onto a microprocessor, and then try to observe the behavior of the code on the microprocessor's pins.

L1-2 Complete the detailed specification of the train controller that was started in Section 1.4. Show all the required classes. Specify the behaviors for those classes. Use object diagrams to show the instantiated objects in the complete system. Develop at least one sequence diagram to show system operation.

L1-3 Develop a requirements description for an interesting device. The device may be a household appliance, a computer peripheral, or whatever you wish.

L1-4 Write a specification for an interesting device in UML. Try to use a variety of UML diagrams, including class diagrams, object diagrams, sequence diagrams, and so on.