EMBEDDED COMPUTING SYSTEMS

Sub Code: 10CS72 **IA Marks :25**
Hrs/Week: 04 **Exam Hours :03**
Total Hrs: 52 **Exam Marks :100**

PART- A

UNIT – 1 **6 Hours**
Embedded Computing: Introduction, Complex Systems and Microprocessors, Embedded Systems Design Process, Formalism for System design Design Example: Model Train Controller.

UNIT – 2 **7 Hours**
Instruction Sets, CPUs: Preliminaries, ARM Processor, Programming Input and Output, Supervisor mode, Exceptions, Traps, Coprocessors, Memory Systems Mechanisms, CPU Performance, CPU Power Consumption. Design Example: Data Compressor.

UNIT – 3 **6 Hours**
Bus-Based Computer Systems: CPU Bus, Memory Devices, I/O devices, Component Interfacing, Designing with Microprocessor, Development and Debugging, System-Level Performance Analysis Design Example: Alarm Clock.

UNIT – 4 **7 Hours**
Program Design and Analysis: Components for embedded programs, Models of programs, Assembly, Linking and Loading, Basic Compilation Techniques, Program optimization, Program-Level performance analysis, Software performance optimization, Program-Level energy and power analysis, Analysis and optimization of program size, Program validation and testing. Design Example: Software modem.

PART- B

UNIT – 5 **6 Hours**
Real Time Operating System (RTOS) Based Design – 1: Basics of OS, Kernel, types of OSs, tasks, processes, Threads, Multitasking and Multiprocessing, Context switching, Scheduling Policies, Task Communication, Task Synchronization.

UNIT – 6 **6 Hours**
RTOS-Based Design - 2: Inter process Communication mechanisms, Evaluating OS performance, Choice of RTOS, Power Optimization. Design Example: Telephone Answering machine

UNIT – 7 **7 Hours**
Distributed Embedded Systems: Distributed Network Architectures, Networks for Embedded Systems: I2C Bus, CAN Bus, SHARC Link Ports, Ethernet, Myrinet, Internet, Network Based Design. Design Example:
Elevator Controller.

UNIT – 8 **7 Hours**
Embedded Systems Development Environment: The Integrated Development Environment, Types of File generated on Cross Compilation, Dis-assembler /Decompiler, Simulators, Emulators, and Debugging, Target Hardware Debugging.

Text Books:
1. Wayne Wolf: Computers as Components, Principles of Embedded Computing Systems Design, 2nd Edition, Elsevier, 2008.
2. Shibu K V: Introduction to Embedded Systems, Tata McGraw Hill, 2009
(Chapters 10, 13)
Reference Books:
1. James K. Peckol: Embedded Systems, A contemporary Design Tool, Wiley India, 2008
2. Tammy Neorgaard: Embedded Systems Architecture, Elsevier, 2005.

Table of Contents

UNIT 1

Embedded Computing

1.1 COMPLEX SYSTEMS AND MICROPROCESSORS

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 itself an embedded computing system, although PCs are often used to build embedded computing systems. But a fax machine or a clock built from a microprocessor is an embedded computing system.

1.1.1 Embedding Computers

A microprocessor is a single-chip CPU. Very large scale integration (VLSI) stet—the acronym is the name technology has allowed us to put a complete CPU on a single chip since 1970s, but those CPUs were very simple. The first microprocessor- the Intel 4004, was designed for an embedded application, namely, a calculator. The calculator was not a general-purpose computer—it merely provided basic arithmetic functions. However, Ted Hoff of Intel realized that a general-purpose computer programmed properly could implement the required function, and that the computer-on-a-chip could then be reprogrammed for use in other products as well. Since integrated circuit design was (and still is) an expensive and time- consuming process, the ability to reuse the hardware design by changing the software was a key breakthrough. The HP-35 was the first handheld calculator to perform transcendental functions [Whi72]. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor. How- ever, the ability to write programs to perform math rather than having to design digital circuits to perform operations like trigonometric functions was critical to the successful design of the calculator.

1.1.2 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 computing and embedded computing, but embedded applications must meet many other constraints as well.

- 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 opti- mize the performance of the car while minimizing pollution and fuel utilization.

- *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.

- *Real time:* Many embedded computing systems have to perform in real time—if the data is not 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 does not 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.

- *Manufacturing cost:* The total cost of building the 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 the hardware, since a larger power supply may be necessary. Energy con- sumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

1.1.3 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 can be extended to provide new features to keep up with rapidly changing markets.

1.1.4 The Physics of Software

Computing is a physical act. Although PCs have trained us to think about computers as purveyors of abstract information, those computers in fact do their work by moving electrons and doing work. This is the fundamental reason why programs take time to finish, why they consume energy, etc.

A prime subject of this book is what we might think of as the ***physics of software***. Software performance and energy consumption are very important prop- erties 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 have to optimize our programs by pushing around electrons. 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.

1.1.5 Challenges in Embedded Computing System Design

External constraints are one important source of difficulty in embedded system design. Let's consider some important problems that must be taken into account in embedded system design.

How much hardware do we need?

We have a great deal of control over the amount of computing power we apply to our problem. We cannot only select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more. Since we often must meet both performance deadlines and manufacturing cost constraints, the choice of hardware is important—too little hardware and the system fails to meet its deadlines, too much hardware and it becomes too expensive.

How do we meet deadlines?

The brute force way of meeting a deadline is to speed up the hardware so that the program runs faster. Of course, that makes the system more expensive.

It is also entirely possible that increasing the CPU clock rate may not make enough difference to execution time, since the program's speed may be limited by the memory system.

How do we minimize power consumption?

In battery-powered applications, power consumption is extremely important. Even in nonbattery applications, excessive power consumption can increase heat dis- sipation. One way to make a digital system consume less power is to make it un more slowly, but naively slowing down the system can obviously lead to missed deadlines. Careful design is required to slow down the noncritical parts of the machine for power consumption while still meeting necessary performance goals.

How do we design for upgradability?

The hardware platform may be used over several product generations, or for several different versions of a product in the same generation, with few or no changes. However, we want to be able to add features by changing software. How can we design a machine that will provide the required performance for software that we haven't yet written?

How Does it Really work ?

Reliability is always important when selling products—customers rightly expect that products they buy will work. Reliability is especially important in some appli- cations, such as safety-critical systems. If we wait until we have a running system and try to eliminate the bugs, we will be too late—we won't find enough bugs, it will be too expensive to fix them, and it will take too long as well. Another set of challenges comes from the characteristics of the components and systems them- selves. If workstation programming is like assembling a machine on a bench, then embedded system design is often more like working on a car—cramped, delicate, and difficult. Let's consider some ways in which the nature of embedded computing machines makes their design more difficult.

- *Complex testing:* Exercising an embedded system is generally more difficult than typing in some data. We may have to run a real machine in order to generate the proper data. The timing of data is often important, meaning that we cannot separate the testing of an embedded computer from the machine in which it is embedded.

- *Limited observability and controllability:* Embedded computing systems usually do not come with keyboards and screens. This makes it more difficult to see what is going on and to affect the system's operation. We may be forced to watch the values of electrical signals on the microprocessor bus, for example, to know what is going on inside the system. Moreover, in real-time applica- tions we may not be able to easily stop the system to see what is going on inside.

- *Restricted development environments:* The development environments for embedded systems (the tools used to develop software and hardware) are often much more limited than those available for PCs and workstations. We generally compile code on one type of machine, such as a PC, and download it onto the embedded system. To debug the code, we must usually rely on pro- grams that run on the PC or workstation and then look inside the embedded system.

1.1.6 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; the deadline is the time at which a computation must be finished. If the program does not produce the required output by the deadline, then the program does not work, even if the output that it eventually produces is functionally correct.

- *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 com- ponents that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.

- *Program:* Programs are very large and the CPU sees only a small window of the program at a 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.

1.2 THE EMBEDDED SYSTEM DESIGN PROCESS

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 perform- ing functional tests. Second, it allows us to develop computer-aided design tools. Developing a single program that takes in a concept for an embedded system and emits a completed design would be a daunting task, but 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. Since most embedded systems are designed by teams, coordination is perhaps the most important role of a well-defined design methodology.

specification, we create a more detailed description of what we want. But 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 those components, including both software modules and any specialized hardware we need. Based on those components, we can finally build a complete system.

In this section we will consider design from the ***top–down***—we will 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 the figure 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 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

But 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 (both overall speed and deadlines); and

- 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.

- And we must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

1.2.1 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. We generally proceed in two phases: First, we gather an informal description from the customers known as requirements, and we refine the requirements into a specification that contains enough information to begin designing the system architecture

- *Performance:* The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost. As we have noted, perfor- mance 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 for the system is almost always a consideration. Cost typically has two major components: ***manufacturing cost*** includes the cost of components and assembly; ***nonrecurring engineering*** **(NRE)** costs include the personnel 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—the customer is unlikely to be able to describe the allowable wattage.

Validating a set of requirements is ultimately a psychological task since 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. But 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.

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 the 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?

— *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 since they must be carefully measured during implementation to ensure that the system works properly.

- *Manufacturing cost:* This includes primarily the cost of the hardware compo- nents. 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 $10 most likely has a very different internal structure than a $100 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 similar to Figure 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.

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 handheld device that 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 posi- tion. The moving map obtains its position from the GPS, a satellite-based navigation system. The moving map display might look something like the following figure.

- *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. Upon power-up, a display should take no more than one second 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 $100.

■ *Physical size and weight:* The device should fit comfortably in the palm of the hand.

■ *Power consumption:* The device should run for at least eight hours on four AA

batteries.

1.2.2 Specification

The specification is more precise—it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it accurately reflects the customer'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 neo- phyte 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 obvi- ous 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, the resulting system is also very likely to be inelegant, kludgey, and bug-ridden.

The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer. It should also be unambiguous enough that designers know what they need to build. Designers

1.2.3 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 archi- tecture for the moving map of Example 1.1. Figure 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 have not yet specified which oper- ations 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 (i.e., draw) the results for the display. We have chosen to separate those functions so that we can potentially do them in parallel—performing rendering separately from searching the database may help us update the screen more fluidly.

1.2.4 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 spec- ification. 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 com- ponents. 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 stan- dard routines to access the database—not only is the data in a predefined format, but it is highly compressed to save storage. Using standard software for these access functions not only saves us design time, but it may give us a faster implementation for specialized functions such as the data decompression phase.

1.2.5 System Integration

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 do not stet work correctly and how they can be fixed is a challenge in itself.

1.3 FORMALISMS FOR SYSTEM DESIGN

As mentioned in the last 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 code,and designing tests. It is often helpful to conceptualize these tasks in diagrams. Luckily, there is a visual language that can be used to capture all these design

tasks: the ***Unified Modeling Language (UML)*** [Boo99, Pil05]. UML was designed to be useful at many levels of abstraction in the design process. UML 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. We will see precisely what we mean by an object in just a moment, but object-oriented design emphasizes two concepts of importance:

- It encourages the design to be described as a number of interacting objerather than 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 implementations.

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

- Object-oriented specification allows a system to be described in a way that closely models real-world objects and their interactions.

- Object-oriented specification 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.

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.

Unified Modeling Language (UML)—the acronym is the name is a large lan- guage, 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, and solid and broken lines. As you become more familiar with the language, uses of the graphical primitives will become more natural to you.

1.3.1 Structural Description

By ***structural description***, we mean 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 A class is a form of type definition—all objects derived from the same class have the same characteristics, although their attributes may have different values. A class defines the attributes that an object may have. It 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 Figure 1.6. The class has the name that we saw used in the $d1$ object since $d1$ is an instance of class *Display*. The *Display* class defines the *pixels* attribute seen in the object; remember that when we instantiate the class 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 have to 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 opera- tions 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.

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.

Unified Modeling Language, like most object-oriented languages, allows us to define one class in terms of another. An example is shown in Figure 1.7, where we ***derive*** two particular 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 the user to select from a behaviors—for example, large number of available colors even with a small number of 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. Even more important, 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?

Unified Modeling Language considers inheritance to be one form of general- ization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead. Both *BW_display* and *Color*

versions of *Display*, so *Display* generalizes both of them. 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 Figure 1.8; we have omit- ted 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 capture type information about these links. 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, namely, *contains*. The ball and the number at the message class end indicate that the message 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***

1.3.2 Behavioral Description

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

These state machines will 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***.

- 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—

Let's consider a simple state machine specification to understand the semantics of UML state machines. A state machine for an operation of the display is shown in Figure 1.12. The start and stop states are special states that help us to organize the flow of the state machine. The states in the state machine represent different conceptual 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 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, like the one for a mouse click scenario shown in Figure 1.13. A ***sequence diagram*** is somewhat similar to a hardware timing diagram, although the time flows verti- cally 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 a number of mutually exclusive possibil- ities. In this case, the sequence shows what happens when a mouse click is 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 the 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,

that is, 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 in turn uses call events to invoke the menu object twice: once to determine which menu item was selected and again to actually 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 MODEL TRAIN CONTROLLER

In order to learn how to use UML to model systems, we will specify a simple system, a model train controller, which is illustrated in Figure 1.14. The user sends messages to the train with a control box attached to the tracks. The control box may have familiar controls such as a throttle, emergency stop button, and so on. Since 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 commands back to the user.

1.4.1 Requirements

Before we can create a system specification, we have to understand the require- ments. 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 respon- siveness 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.

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
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)
size and weight	Console should be comfortable for two hands, approx- imate size of standard keyboard; weight 2 pounds

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 have to do and it allows us to use a wide variety of existing trains and other pieces of equipment.

1.4.2 DCC

The **Digital Command Control (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. Hob- byists 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.

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 lan- guage 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 interoper- ability. Overstandardization, or specifying elements that do not really need to be standardized, only makes the standard less attractive and harder to implement.

The Electrical Standard deals with voltages and currents on the track. While the electrical engineering aspects of this part of the specification are beyond the scope of the book, we will 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 DC value of the rails.

The data signal swings between two voltages around the power supply volt- age. As shown in Figure 1.15, bits are encoded in the time between transitions, not by voltage levels. A 0 is at least 100 s while a 1 is nominally 58 s. The dura- tions of the high (above nominal voltage) and low (below nominal voltage) parts of a bit are equal to keep the DC value constant. The specification 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.

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 Recommended Practices documents.

- *P* is the preamble, which is a sequence of at least 10 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 eight bits long. The addresses 00000000, 11111110, and 11111111 are reserved.

- *s* is the data byte start bit, which, like the packet start bit, is a 0.

- *D* is the data byte, which includes eight 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.

A ***baseline packet*** is the minimum packet that must be accepted by all DCC implementations. More complex packets are given in a Recommended Practice doc- ument. A baseline packet has three data bytes: an address data byte that gives the intended receiver of the packet; the instruction data byte provides a basic instruc- tion; 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 1 for forward and 0 for reverse. Bits

7–8 are set at 01 to indicate that this instruction provides speed and direction.

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

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

1.4.3 Conceptual Specification

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

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-to-1 ratio. In fact, the DCC standard says that command units should resend packets in case a packet is dropped during transmission.

We now need to model the train control system itself. There are clearly two major subsystems: the command unit and the train-board component as shown in Figure 1.16. Each of these subsystems has its own internal structure. The basic relationship between them is illustrated in Figure 1.17. This figure shows a UML ***collaboration diagram***; we could have used another type of figure, such as a class or object diagram, but we wanted to emphasize the transmit/receive relationship between these major subsystems. The command unit and receiver are each rep- resented 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; since 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. Since the track is not 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 in the specification dia- grams. For example, if we are worried about what happens when the track breaks,

- *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 figures 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.