UNIT-I
EMBEDDED COMPUTING

Contents at a glance:
✓ Introduction
✓ Complex systems and microprocessor
✓ The Embedded system design process
✓ Formalisms for system design
✓ Design examples

INTRODUCTION:
System Definition:

➢ A way of working, organizing or performing one or many tasks according to a fixed set of rules, program or plan.
➢ Also an arrangement in which all units assemble and work together according to a program or plan.
➢ Examples of Systems:
  • Time display system – A watch
  • Automatic cloth washing system – A washing machine

Embedded System Definitions:

➢ “An embedded system is a system that has software embedded into computer-hardware, which makes a system dedicated for an application (s) or specific part of an application or product or part of a larger system.”
  (Or)
➢ An embedded system is one that has dedicated purpose software embedded in computer hardware.
  (Or)
➢ It is a dedicated computer based system for an application(s) or product. It may be an independent system or a part of large system. Its software usually embeds into a ROM (Read Only Memory) or flash.”
  (Or)
➢ It is any device that includes a programmable computer but is not itself intended to be a general purpose computer.”

In simple words, Embedded System = (Hardware + Software) dedicated for a particular task with its own memory.

The components of embedded system hardware:

MICROPROCESSOR:

➢ Microprocessor is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output.
  or
➢ A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic device
that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to instructions, and provides result as output.

**MICROCONTROLLER:**

- A **microcontroller** (sometimes abbreviated μC, uC or MCU) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM.
- or
- CPUs with integrated memory or peripheral interfaces

**DIGITAL SIGNAL PROCESSOR:**

- Dedicated processors. A digital signal processor (DSP) is a specialized microprocessor (or a SIP block), with its architecture optimized for the operational needs of digital signal processing.

**IMAGE PROCESSOR:**

- An image processor, image processing engine, also called media processor, is a specialized digital signal processor (DSP) used for image processing in digital cameras, mobile phones or other devices.

**EMBEDDED COMPUTING SYSTEM:**

- An embedded system is a special-purpose system in which the computer is completely encapsulated by the device it controls. Unlike a general-purpose computer, such as a personal computer, an embedded system performs pre-defined tasks, usually with very specific requirements. Since the system is dedicated to a specific task, design engineers can optimize it, reducing the size and cost of the product.
- Some examples of embedded systems include ATMs, cell phones, printers, thermostats, calculators, and videogame consoles.
EMBEDDED COMPUTING SYSTEM DESIGN:

➢ To know about embedded computing system design process, first the purpose and uses of microprocessors should be known. Also we should know how microprocessors are used for control, user interface and signal processing etc.

THE CLASSIFICATION OF EMBEDDED SYSTEM IS BASED ON FOLLOWING CRITERIA’S:

1. On generation
2. On complexity & performance
3. On deterministic behaviour
4. On triggering

1. On generation:

(i) First generation (1G):
   ➢ Built around 8bit microprocessor & microcontroller.
   ➢ Simple in hardware circuit & firmware developed.
   Examples: Digital telephone keypads.

(ii) Second generation (2G):
   ➢ Built around 16-bit μp & 8-bit μc.
   ➢ They are more complex & powerful than 1G μp & μc.
   Examples: SCADA systems

(iii) Third generation (3G):
   ➢ Built around 32-bit μp & 16-bit μc.
   ➢ Concepts like Digital Signal Processors (DSPs), Application Specific Integrated Circuits (ASICs) solved.
   Examples: Robotics, Media, etc.

(iv) Fourth generation:
   ➢ Built around 64-bit μp & 32-bit μc.
   ➢ The concept of System on Chips (SoC), Multicore Processors evolved.
   ➢ Highly complex & very powerful.
   Examples: Smart Phones.

2. On complexity & performance

(i) Small-scale Embedded Systems:
   ◦ Simple in application need
   ◦ Performance not time-critical.
   ◦ Built around low performance & low cost 8 or 16 bit μp/μc.
   Example: an electronic toy

(ii) Medium-scale Embedded Systems:
   ◦ Slightly complex in hardware & firmware requirement.
   ◦ Built around medium performance & low cost 16 or 32 bit μp/μc.
   ◦ Usually contain operating system.
   Examples: Industrial machines.
(iii) Large-scale Embedded Systems:
- Highly complex hardware & firmware.
- Built around 32 or 64 bit RISC μp/μc or PLDs or Multicore Processors.
- Response is time-critical.
Examples: Mission critical applications.

3. on deterministic behavior

This classification is applicable for “Real Time” systems. The task execution behavior for an embedded system may be deterministic or non-deterministic. Based on execution behavior Real Time embedded systems are divided into two types
- Hard Real Time embedded systems
- Soft Real Time embedded systems

4 On triggering

Embedded systems which are “Reactive” in nature can be based on triggering. Reactive systems can be:
- Event triggered
- Time triggered

COMPLEX SYSTEM AND MICROPROCESSORS:

➢ Three main tasks or components in embedded system design:
  o Selecting and integrating hardware to give computer like functionalities
  o Dumping main application software generally into flash or ROM and the application software performs concurrently the number of tasks.
  o Integrating with a real time operating system (RTOS), this supervises the application software tasks running on the hardware and organizes the accesses to system resources according to priorities and timing constraints of tasks in the system.

Embedding Computers:

➢ Whirlwind, a computer designed at MIT in the late 1940s and early 1950s. Whirlwind was also the first computer designed to support real-time operation and was originally conceived as a mechanism for controlling an aircraft simulator. It was extremely large physically compared to today’s computers (e.g., it contained over 4,000 vacuum tubes).

➢ Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s. A microprocessor is a single-chip CPU. Very large scale integration (VLSI) technology allowed us to put a complete CPU on a single chip since 1970s, but those CPUs were very simple.

➢ In 1971 the first microprocessor the Intel 4004 invented by Ted Hoff, was designed for an embedded application, namely, a calculator. The calculator was not a general-purpose computer—it merely provided basic arithmetic functions. The HP-35 was the first handheld calculator to perform
transcendental functions. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor.

➢ **Automobile designers** started making use of the microprocessor soon after single-chip CPUs became available. The most important and sophisticated use of microprocessors in automobiles was to **control the engine**: determining when spark plugs fire, controlling the fuel/air mixture, and so on.

➢ **Microprocessors** are usually classified according to their word length.

  o An 8-bit **microcontroller** is designed for low-cost applications and includes on-board memory and I/O devices
  o 16-bit microcontroller is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory;
  o 32-bit **RISC** microprocessor offers very high performance for computation-intensive applications.

➢ **House Hold uses of microprocessor**:

  o The typical **microwave oven** has at least one microprocessor to control oven operation.
  o Many houses have **advanced thermostat systems**, which change the temperature level at various times during the day.
  o The **modern camera** is a prime example of the powerful features that can be added under microprocessor control.
  o **Digital Television** uses embedded processors

**APPLICATIONS OF EMBEDDED SYSTEMS IN VARIOUS SECTORS:**

*We can find applications of embedded systems in following sectors:*

- Daily Life Electronic appliances( Lift, Microwave Oven, Refrigerator, Washing Machine)
- Health Care( X-ray, ECG, Cardiograph, diseases diagnosis devices etc)
- Education (Laptop or desktop, projector, printer, calculator, lab equipments etc)
- Communication( Mobile phone, satellite, Modem, Network Hub, Router, Telephone, Fax)
- Security System( CC Camera, X ray Scanner, RFID System, Password protected door, Face detection)
- Entertainment( Television etc)
- Banking System( ATM etc)
- Automation
- Navigation
- Consumer Electronics: Camcorders, Cameras
- Household appliances: Washing machine, Refrigerator.
- Automotive industry: Anti-lock breaking system(ABS), engine control
- Home automation & security systems: Air conditioners, sprinklers, fire alarms.
- Telecom: Cellular phones, telephone switches.
- Computer peripherals: Printers, scanners.
- Computer networking systems: Network routers and switches.
- Healthcare: EEG, ECG machines.
• Banking & Retail: Automatic teller machines, point of sales.
• Card Readers: Barcode, smart card readers

EXAMPLE:

**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 (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.
➢ Let’s first look at the ABS. The purpose of an ABS is to temporarily release the brake on a wheel when it rotates too slowly—when a wheel stops turning, the car starts skidding and becomes hard to control. It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves as seen in the below diagram. The ABS system uses sensors on each wheel to measure the speed of the wheel. The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent the wheels from skidding.

➢ The ASC _ T system’s job is to control the engine power and the brake to improve the car’s stability. The ASC _ T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting.

*Characteristics of Embedded Computing Applications:*

  a. Complex Algorithms
  b. User Interface
  c. Real Time
  d. Multirate
e. Manufacturing Cost
f. Power

➢ **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 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.

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

➢ **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 behaviour. 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.

Costs of various sorts are also very important:

➢ **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 consumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

**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:
  o Microprocessors are a very efficient way to implement digital systems.
  o 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.

Other reasons are

➢ Predesigned instruction set processor may in fact result in faster implementation of your application
than designing your own custom logic.

➢ But there are two factors that work together to make microprocessor-based designs fast.
   - First, microprocessors execute programs very efficiently. Modern RISC processors can execute
     one instruction per clock cycle most of the time and high performance processors can
     execute several instructions per cycle.
   - Second, microprocessor manufacturers spend a great deal of money to make their CPUs run
     very fast. With the slight changes designer can make the microprocessor to run at the highest
     possible speed.

➢ Microprocessors are **efficient utilizers of logic**
   ➢ Microprocessors can be used for many different algorithms simply by changing the program it executes.
   ➢ The microprocessors allow program design to be separated from the design of hardware on which
     programs will be running.

**Challenges in Embedded Computing System Design:**

  i. How much hardware do we need?
  ii. How do we meet deadlines?
  iii. How do we minimize power consumption?
  iv. How do we design for upgradability?
  v. Does it really work?
  vi. Complex testing
  vii. Limited observability and controllability
  viii. Restricted development environments

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 non battery applications, excessive power consumption can increase heat dissipation. One way to make a digital system consume less power is to make it run more slowly, 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.

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 applications. 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 more time.

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 applications 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 programs that run on the PC or workstation and then look inside the embedded system.

THE EMBEDDED SYSTEM DESIGN PROCESS

➢ The embedded system design process aimed at two objectives.
  o First, it will give us an introduction to the various steps in embedded system design
Second, it will allow us to consider the design methodology itself.

A design methodology is important for three reasons.

First, to ensure that we have done everything we need.

Second, it allows us to develop computer-aided design tools.

Third, it makes members of a design team to communicate easily.

Designing can be done in two ways. They are:

- Top down
- Bottom-up

Figure 1.1 summarizes the major steps in the embedded system design process. In this top–down view, we start from the system requirements. In bottom-up approach we start with components. 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.

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.

We 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. 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:

1. First, we gather an informal description from the customers known as requirements;
2. Second we refine the requirements into a specification that contains enough information to begin designing the system architecture.

Separating out requirements analysis and specification is often necessary because of the large gap between what the customers can describe about the system they want and what the architects need to design the system. Requirements may be functional or non functional.

Typical non functional requirements include:

• 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, 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 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
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 scanned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation.

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 towards understanding system requirements. As part of system design to analyze requirements, we will use a simple requirements methodology. Figure 1.2 shows a sample requirements form that can be filled out at the start of the project. Let’s consider the entries in the form:

- **Name:** This is simple but helpful. Giving a name to the project should tell 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? 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 to control physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time.
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.

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 that helps to take architectural decisions.

After writing the requirements, you should check them for internal consistency. 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 position. 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.

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. 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 sec.

- **Cost:** The selling cost 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 batteries.

### Requirements form for GPS moving map system:

<table>
<thead>
<tr>
<th>Name</th>
<th>GPS moving map</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Consumer-grade moving map for driving use</td>
</tr>
<tr>
<td><strong>Inputs</strong></td>
<td>Power button, two control buttons</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
<td>Back-lit LCD display 400 x 600</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td>Uses 5-receiver GPS system. Three user-selectable resolutions: always display current latitude and longitude</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Updates screen within 0.25 seconds upon movement</td>
</tr>
<tr>
<td><strong>Manufacturing cost</strong></td>
<td>$30</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>100mW</td>
</tr>
<tr>
<td><strong>Physical size and weight</strong></td>
<td>No more than 2” x 6” x 12 ounces</td>
</tr>
</tbody>
</table>

The selling price is four to five times the **cost of goods sold** (the total of all the component costs).

### 2. Specification:

- The specification is more precise—it serves as the contract between the customer and the architects.
- The specification must be carefully written so that it accurately reflects the customer’s requirements and that can be clearly followed during design.
- An unclear specification leads different types of problems.
- If the behaviour 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 satellite constellation.
  - 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.
3. **Architecture Design:**

- 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.
- To understand what an architectural description is, let’s look at sample architecture 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.

![Block diagram for the moving map.](image)

- 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.
- For more implementation details we should refine that system block diagram into two block diagrams:
  - Hardware block diagram (Hardware architecture)
  - Software block diagram (Software architecture)
- These two more refined block diagrams are shown in Figure 1.4.
- The hardware block diagram clearly shows that we have one central CPU surrounded by memory and I/O devices.
- We have chosen to use two memories:
  - A frame buffer for the pixels to be displayed
  - A separate program/data memory for general use by the CPU
- The software block diagram fairly closely follows the system block diagram.
- We have added a timer to control when we read the buttons on the user interface and render data onto the screen.
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 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.

4. Designing Hardware and Software Components

- The architectural description tells us what components we need.
- In general the components will 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—the data in a predefined format and it is highly compressed to save storage.
- Using standard software for these access functions not only saves us design time.
5. System Integration:

- Putting hardware and software components together will give complete working system.
- Bugs are typically found during system integration, and good planning can help us to find the bugs quickly.
- If we debug only a few modules at a time, we are more likely to uncover the simple bugs and able to easily recognize them.
- 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 work correctly and how they can be fixed is a challenge in itself.

4. FORMALISMS FOR SYSTEM DESIGN

- We perform a number of different design tasks at different levels of abstraction: creating requirements and specifications, architecting the system, designing code, and designing tests. It is often helpful to conceptualize these tasks in diagrams.
- The Unified Modeling Language (UML). UML was designed to be useful at many levels of abstraction in the design process. UML is an object-oriented modeling language.
- The design in terms of actual objects helps us to understand the natural structure of the system.

- Object-oriented 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.
- What is the relationship between an object-oriented specification and an object-oriented programming language?
- A specification language may not be executable. But both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

Structural Description:

- By structural description, we mean the basic components of the system.
- The principal component of an object-oriented design is 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 UML notation in Figure 1.5.
➢ The text in the folded-corner page 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 holds 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.
➢ 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.

![Figure 1.5](image)

**Figure 1.5**
An object in UML notation.

➢ 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;
➢ A class defines both the interface for a particular type of object and that object’s implementation.
➢ 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.
**Derived class:**

- *Unified Modeling Language*, 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 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.
- A **derived class** inherits all the attributes and operations from its **base class**.
- Here *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.

![UML Diagram](image)

**FIGURE 1.7**

Derived classes as a form of generalization in UML.

- Inheritance has two purposes.
  - It allows us to describe one class that shares some characteristics with another class.
  - It captures those relationships between classes and documents them

- *Unified Modeling Language* 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 of them.

**Multiple inheritances:**

- In which a class is derived from more than one base class.
- An example of multiple inheritances is shown in Figure 1.8; 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**.

![Multiple inheritance UML diagram](image)

**FIGURE 1.8**

Multiple inheritance in UML.

**Link:**

- A **link** describes a relationship between objects; association is to link as class is to object.
- Fig1.9 shows an example of links and an association.

![Links between objects](image)

**FIGURE 1.9**

Links and 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 set may include zero or more message objects.

**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*.

- Fig1.10 shows UML states; the transition between two states is shown by 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*.

<table>
<thead>
<tr>
<th>State</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

**FIGURE 1.10**

A state and transition in UML.

- An event is some type of action. Events are divided into two categories. They are:
  - External events: The event may originate outside the system, such as a user pressing a button.
  - Internal events: It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine.

- We will concentrate on the following three types of events defined by UML, as illustrated in figure 1.11(signal and call event) and (Time out event)
  - 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.
Unconditional and conditional transitions:

- 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.
- 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 Fig1.12. The start and stop states are special states that help us to organize the flow of the state machine.
Sequence diagram:

- 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 Fig 1.13.
- A sequence diagram is somewhat similar to 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. 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.

**DESIGN EXAMPLE: MODEL TRAIN CONTROLLER:**

➢ The model train controller, which is shown in the below figure.

i. The user sends messages to the train with the control box attached to the tracks.

ii. The control box may have familiar controls such as throttle, emergency stop button and so on.

iii. Since train receives its electrical power from the track, the control box can send a signal to the train over the track by modulating the power supply voltage.

iv. As shown Fig1.14, the control panel sends packet over the tracks to the receiver on the train. 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.

**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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Model train controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Control speed of up to eight model trains</td>
</tr>
<tr>
<td>Inputs</td>
<td>Throttle, inertia setting, emergency stop, train number</td>
</tr>
<tr>
<td>Outputs</td>
<td>Train control signal</td>
</tr>
<tr>
<td>Functions</td>
<td>Set engine speed based upon inertia settings, Respond to emergency stop</td>
</tr>
<tr>
<td>Performance</td>
<td>Can update train speed at least 10 times per second</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>$50</td>
</tr>
<tr>
<td>Power</td>
<td>10W (plugs into walls)</td>
</tr>
<tr>
<td>Physical size and weight</td>
<td>Console should be comfortable for two hands, approximate size of standard keyboard. Weight less than 2 pounds</td>
</tr>
</tbody>
</table>
CONCEPTUAL SPECIFICATION OF MODEL TRAIN CONTROLLER:

1. Objects: Console, Train
3. Console: panel, formatter, transmitter
4. Train: receiver, controller, motor interface

➢ The conceptual specification allows us to understand the system little better. Writing of conceptual specification will help us to write a detailed specification. Defining the messages will help us understand the functionality of the components. The set of commands that we can use to implement the requirements placed on the system.

➢ The system console controls the train by sending messages on to the tracks. The transmissions are packetized: each packet includes an address and a message. A typical sequence of train control commands is shown as a UML sequence diagram.

![UML sequence diagram](image)

Fig: A UML sequence diagram for a typical sequence of train control commands

➢ The focus of the control bars shows both the console and receiver run continuously. The packets can be sent at any time—there is no global clock controlling when the console sends and the train receives, we do not have to worry about detecting collisions among the packets.

➢ Set-inertia message will send infrequently. Most of the message commands are speed commands. When a train receives speed command, it will speed up and slow down the train smoothly at rate determined by the set-inertia command.

➢ An emergency stop command may be received, which causes the train receiver to immediately shut down the train motor.

➢ We can model the **commands in UML** with two level class hierarchy as shown in the Fig1.16. Here we have one base class command, there are three sub classes set-speed, set-inertia, Estop, derived from base class. One for each specific type of command.
We now need to model the train control system itself. There are clearly two major subsystems: the control-box and the train board component. Each of these subsystems has its own internal structure.

The figure 1.17 shows relationship between console and receiver (ignores role of track):

The figure 1.18 shows:

1. Console class roles:
   - **Panel:** Describes the console front panel, which contains analog knobs and interface hardware to interface to the digital parts of the system.
   - **Formatter:** It knows how to read the panel knobs and creates bit stream for message.
• **Transmitter**: Send the message along the track.

• **Knobs* describes the actual analog knobs, buttons, and levers on the control panel.**

• **Sender* describes the analog electronics that send bits along the track.**

➢ **Train class roles:**

• **Receiver**: It knows how to turn the analog signal on the track into digital form.

• **Controller**: Interprets received commands and figures out how to control the motor.

• **Motor interface**: Generates the analog signals required to control the motor.

➢ **We define two classes to represent analog components:**

  o **Detector* detects analog signals on the track and converts them into digital form.**

  o **Pulser* turns digital commands into the analog signals required to control the motor speed.**

**DETAILED SPECIFICATION:**

➢ Conceptual specification that defines the basic classes, let’s refine it to create a more detailed specification. We won’t make a complete specification. But we will add details to the class. We can
now fill in the details of the conceptual specification. Sketching out the spec first helps us understand the basic relationships in the system.

➢ We need to define the analog components in a little more detail because there characteristics will strongly influence the formatter and controller. Fig1.19 shows a little more detail than Fig 1.18, It includes attributes and behavior 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 emergency-stop.

➢ The Sender and Detector classes are relatively simple: They simply put out and pick up a bit, respectively.

![Knobs and Pulser classes](image)

**FIGURE 1.19**

Classes describing analog physical objects in the train control system.

➢ To understand the Pulser class, let’s consider how we actually control the train motor’s speed. As shown in Figure 1.20, 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.

![Pulse-width modulation diagram](image)

**FIGURE 1.20**

Controlling motor speed by pulse-width modulation.
Figure 1.21 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;
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.
The Motor-interface defines an attribute for speed that can be set by other classes.
The Transmitter and Receiver classes are shown in Figure 1.22. 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.
The Formatter class is shown in Figure 1.23. The formatter holds the current control settings for all of 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.
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 Figure 1.24.

- The figure shows two changes to the knob settings: first to the throttle, inertia, or emergency stop; 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 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.

➢ The state diagram for a very simple version of the operate behavior of the Formatter class is shown in Figure 1.25.

➢ 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.

![State diagram for the formatter operate behavior.](image1)

➢ Figure 1.26 shows a state diagram for the panel-active behavior.

![State diagram for the panel-active behavior.](image2)
➢ The definition of the train’s Controller class is shown in Figure 1.27
➢ 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, direction, and inertia settings as necessary.

![Controller class diagram](image1)

**FIGURE 1.27**
Class diagram for theController class.

➢ A specification for operate is shown in Figure 1.28.

![State diagram for Controller operate behavior](image2)

**FIGURE 1.28**
State diagram for the Controller operate behavior.

➢ The operation of the Controller class during the reception of a set-speed command is illustrated in Figure 1.29.