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.

This means that embedded computing system design is a useful skill for many types of product design. Automobiles, cell phones, and even household appliances make extensive use of microprocessors. Designers in many fields must be able to identify where microprocessors can be used, design a hardware platform with I/O devices that can support the required tasks, and implement software that performs the required processing.

Computer engineering, like mechanical design or thermodynamics, is a fundamental discipline that can be applied in many different domains. But of course, embedded computing system design does not stand alone.

Many of the challenges encountered in the design of an embedded computing system are not computer engineering for example, they may be mechanical or analog electrical problems. In this book we are primarily interested in the embedded computer itself, so we will concentrate on the hardware and software that enable the desired functions in the final product.

Embedding Computers

Computers have been embedded into applications since the earliest days of computing. One example is the 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.

Even though it was extremely large physically compared to today's computers (e.g., it contained over 4,000 vacuum tubes), its complete design from components to system was attuned to the needs of real-time embedded computing.

The utility of computers in replacing mechanical or human controllers was evident from the very beginning of the computer era for example, computers were proposed to control chemical processes in the late 1940s.

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. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor.

However, 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. 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. There was a trend toward electronics in automobiles in general electronic devices could be used to replace the mechanical distributor. But the big push toward microprocessor-based engine control came from two nearly simultaneous developments: The oil shock of the 1970s caused consumers to place much higher value on fuel economy, and fears of pollution resulted in laws restricting automobile engine emissions.

The combination of low fuel consumption and low emissions is very difficult to achieve; to meet these goals without compromising engine performance, automobile manufacturers turned to sophisticated control algorithms that could be implemented only with microprocessors.

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

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

Digital television makes extensive use of embedded processors. In some cases, specialized CPUs are designed to execute important algorithms an example is the CPU designed for audio processing in the SGS Thomson chip set for DirecTV [Lie98]. This processor is designed to efficiently implement programs for digital audio decoding. A programmable CPU was used rather than a hardwired unit for two reasons: First, it made the system easier to design and debug; and second, it allowed the possibility of upgrades and using the CPU for other purposes.

A high-end automobile may have 100 microprocessors, but even inexpensive cars today use 40 microprocessors. Some of these microprocessors do very simple things such as detect whether seat belts are in use. Others control critical functions such as the ignition and braking systems.

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.

Figure 1.1.1 shows the function of an Antilock Brake System. 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 following diagram. This hookup allows the ABS system to modulate the brakes in order to keep the wheels from locking.

The ABS system uses sensors on each wheel to measure 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.

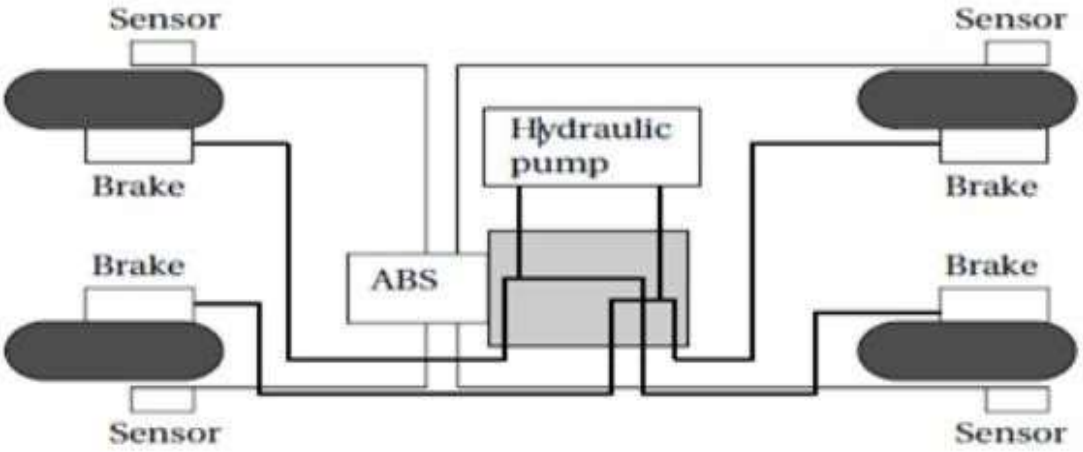


Figure 1.1.1 Antilock Brake System (ABS)

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

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

can be turned off by the driver, which can be important when operating with tire snow chains. The ABS and ASC+ T must clearly communicate because the ASC + T interacts with the brake system. Since the ABS was introduced several years earlier than the ASC + T, it was important to be able to interface ASC + T to the existing ABS module, as well as to other existing electronic modules.

The engine and control management units include the electronically controlled throttle, digital engine management, and electronic transmission control. The ASC + T control unit has two microprocessors on two printed circuit boards, one of which concentrates on logic-relevant components and the other on performance-specific components.

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.

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

Complex algorithms: The operations performed by the microprocessor may be very sophisticated. For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel 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*** 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.

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.

CONSUMER ELECTRONICS ARCHITECTURE

Although some predict the complete convergence of all consumer electronic functions into a single device, much as the personal computer now relies on a common platform, we still have a variety of devices with different functions. However, consumer electronics devices have converged over the past decade around a set of common features that are supported by common architectural features. Not all devices have all features, depending on the way the device is to be used, but most devices select features from a common menu. Similarly, there is no single platform for consumer electronics devices, but the architectures in use are organized around some common themes. This convergence is possible because these devices implement a few basic types of functions in various combinations: multimedia, communications, and data storage and management. The style of multimedia or communications may vary, and different devices may use different formats, but this causes variations in hardware and software components within the basic architectural templates. In this section we will look at general features of consumer electronics devices; in the following sections we will study a few devices in more detail.

Use Cases and Requirements

Consumer electronics devices provide several types of services in different combinations:

- *Multimedia:* The media may be audio, still images, or video (which includes both motion pictures and audio). These multimedia objects are generally stored in compressed form and must be uncompressed to be played (audio playback, video viewing, etc.). A large and growing number of standards have been developed for multimedia compression:MP3, Dolby Digital(TM), etc. for audio; JPEG for still images; MPEG-2, MPEG-4, H.264, etc. for video.
- *Data storage and management:* Because people want to select what multimedia objects they save or play, data storage goes hand-in-hand with multimedia capture and display. Many devices provide PC-compatible file systems so that data can be shared more easily.

- *Communications:* Communications may be relatively simple, such as a USB interface to a host computer. The communications link may also be more sophisticated, such as an Ethernet port or a cellular telephone link.

Consumer electronics devices must meet several types of strict nonfunctional requirements as well. Many devices are battery-operated, which means that they must operate under strict energy budgets. A typical battery for a portable device provides only about 75mW, which must support not only the processors and digital electronics but also the display, radio, etc. Consumer electronics must also be very inexpensive. A typical primary processing chip must sell in the neighborhood of $10. These devices must also provide very high performance sophisticated networking and multimedia compression require huge amounts of computation. Let's consider some basic use cases of some basic operations. Figure 1.8.1 shows a use case for selecting and playing a multimedia object (an audio clip, a picture, etc.). Selecting an object makes use of both the user interface and the file system. Playing also makes use of the file system as well as the decoding subsystem and I/O subsystem. Figure 1.8.2 shows a use case for connecting to a client. The connection may be either over a local connection like USB or over the Internet. While some operations may be performed locally on the client device, most of the work is done on the host system while the connection is established.

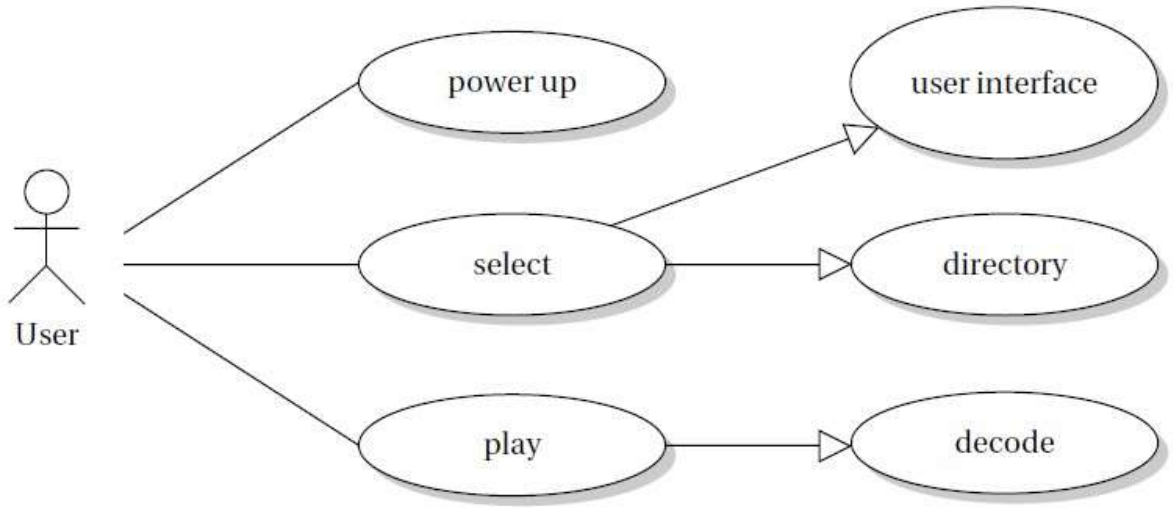


Figure 1.8.1 Use case for playing multimedia

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

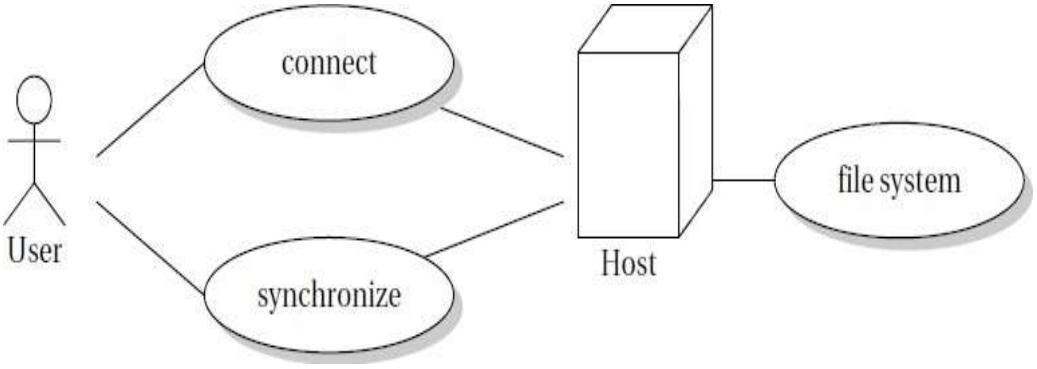


Figure 1.8.2 Use case of synchronizing with a host system

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

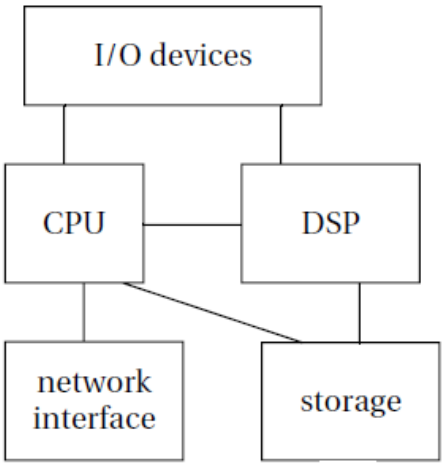


Figure 1.8.3 Functional architecture of a generic consumer electronics device

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Platforms and Operating Systems

Given these types of usage scenarios, we can deduce a few basic characteristics of the underlying architecture of these devices. Figure 1.8.3 shows a functional block diagram of a typical device. The storage system provides bulk, permanent storage. The network interface may provide a simple USB connection or a full-blown Internet connection.

Multiprocessor architectures are common in many consumer multimedia devices. Figure 1.8.3 shows a two-processor architecture; if more computation is required, more DSPs and CPUs may be added. The RISC CPU runs the operating system, runs the user interface, maintains the file system, etc. The DSP performs signal processing. The DSP may be programmable in some systems; in other cases, it may be one or more hardwired accelerators.

The operating system that runs on the CPU must maintain processes and the file system. Processes are necessary to provide concurrency, for example, the user wants to be able to push a button while the device is playing back audio. Depending on the complexity of the device, the operating system may not need to create tasks dynamically. If all tasks can be created using initialization code, the operating system can be made smaller and simpler.

Flash File Systems

Many consumer electronics devices use flash memory for mass storage. Flash memory is a type of semiconductor memory that, unlike DRAM or SRAM, provides permanent storage. Values are stored in the flash memory cell as electric charge using a specialized capacitor that can store the charge for years. The flash memory cell does not require an external power supply to maintain its value. Furthermore, the memory can be written electrically and, unlike previous generations of electrically-erasable semiconductor memory, can be written using standard power supply voltages and so does not need to be disconnected during programming.

Disk drives, which use rotating magnetic platters, are the most common form of mass storage in PCs. Disk drives have some advantages: they are much cheaper than flash memory (at this writing, disk storage costs $0.50 per gigabyte, while flash memory is slightly less than $50/gigabyte) and they have much greater capacity. But disk drives also consume more power than flash storage. When devices need a moderate amount of storage, they often use flash memory. The file system of a device is typically shared with a PC. In many cases the memory device is read directly by the PC through a flash card reader or a USB port.

The device must therefore maintain a PC-compatible file system, using the same directory structure, file names, etc. as are used on a PC. However, flash memory has one important limitation that must be taken into account. Writing a flash memory cell causes mechanical stress that eventually wears out the cell. Today’s flash memories can reliably be written a million times but at some point they will fail. While a million write cycles may sound like enough to ensure that the memory will never wear out, creating a single file may require many write operations, particularly to the part of the memory that stores the directory information.

A wear-leveling flash file system manages the use of flash memory locations to equalize wear while maintaining compatibility with existing file systems. A simple model of a standard file system has two layers: the bottom layer handles physical reads and writes on the storage device; the top layer provides a logical view of the file system. A flash file system imposes an intermediate layer that allows the logical-to-physical mapping of files to be changed. This layer keeps track of how frequently different sections of the flash memory have been written and allocates data to equalize wear. It may also move the location of the directory structure while the file system is operating. Because the directory system receives the most wear, keeping it in one place may cause part of the memory to wear out before the rest, unnecessarily reducing the useful life of the memory device. Several flash file systems have been developed, such as Yet Another Flash Filing System (YAFFS).

DESIGN EXAMPLE: 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 1.3.1 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 1.3.1, 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.

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

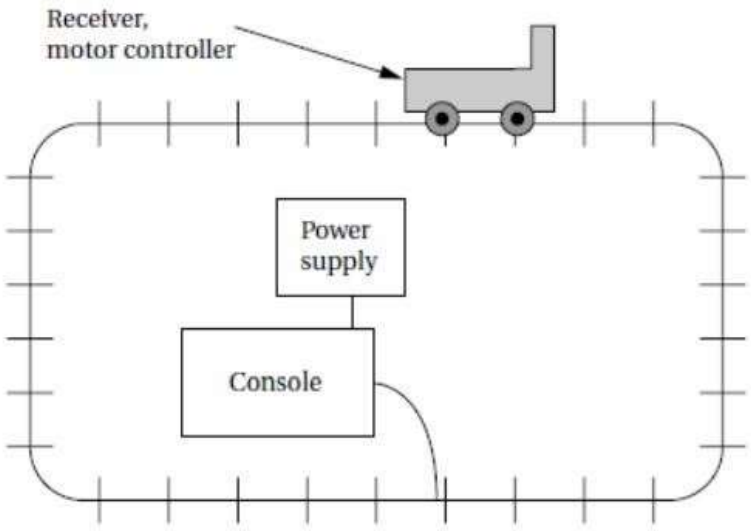
Requirements

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

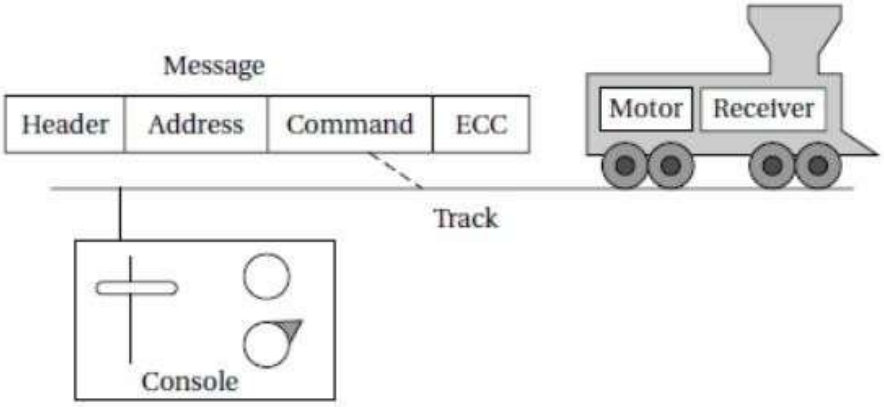
- The console shall be able to control up to eight trains on a single track.
- The speed of each train shall be controllable by a throttle to at least 63 different levels in each direction (forward and reverse).
- There shall be an inertia control that shall allow the user to adjust the responsiveness of the train to commanded changes in speed. Higher inertia means that the train responds

more slowly to a change in the throttle, simulating the inertia of a large train. The inertia control will provide at least eight different levels.

- There shall be an emergency stop button.
- An error detection scheme will be used to transmit messages.



System setup



Signaling the train

Figure 1.3.1 Model Train Control System

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

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.

We can put the requirements into chart format:

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

DCC

The **Digital Command Control (DCC)** was created by the National Model Railroad Association to support interoperable digitally-controlled model trains.

Hobbyists started building homebrew digital control systems in the 1970s and Marklin developed its own digital control system in the 1980s. DCC was created to provide a standard that could be built by any manufacturer so that hobbyists could mix and match components from multiple vendors.

The DCC standard is given in two documents:

Standard S-9.1, the DCC Electrical Standard, defines how bits are encoded on the rails for transmission.

Standard S-9.2, the DCC Communication Standard, defines the packets that carry information.

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

The DCC standard does not specify many aspects of a DCC train system. It doesn't define the control panel, the type of microprocessor used, the programming language to be used, or many other aspects of a real model train system.

The standard concentrates on those aspects of system design that are necessary for interoperability. Over standardization, 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 voltage. As shown in Figure 1.3.2, bits are encoded in the time between transitions, not by voltage levels. A 0 is at least 100 ms while a 1 is nominally 58ms.

The durations 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.

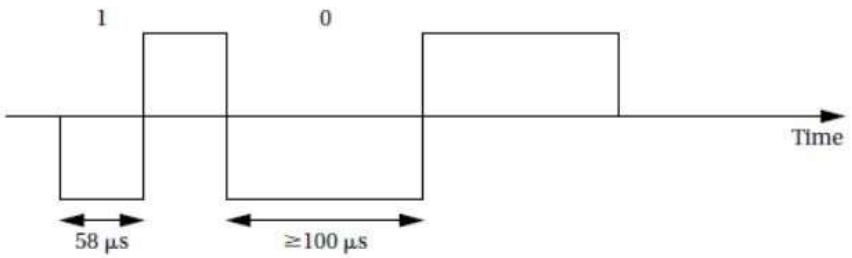


Figure 1.3.2 Bit Encoding in DCC

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Some packet types are left undefined in the standard but typical uses are given in Recommended Practices documents. We can write the basic packet format as a regular expression:

PSA (sD) + E..... (1.1)

In this regular expression:

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 document. 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 instruction; and an error correction data byte is used to detect and correct transmission errors.

The instruction data byte carries several pieces of information. Bits 0–3 provide a 4-bit speed value. Bit 4 has an additional speed bit, which is interpreted as the least significant speed bit. Bit 5 gives direction, with 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 data byte 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.

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 specification 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 specification 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 1.3.3. Each of these subsystems has its own internal structure.

The basic relationship between them is illustrated in Figure 1.3.4. 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 represented by objects; the command unit sends a sequence of packets to the train's receiver, as illustrated by the arrow.

The notation on the arrow provides both the type of message sent and its sequence in a flow of messages; 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 diagrams.

For example, if we are worried about what happens when the track breaks, modeling the tracks would help us identify failure modes and possible recovery mechanisms.

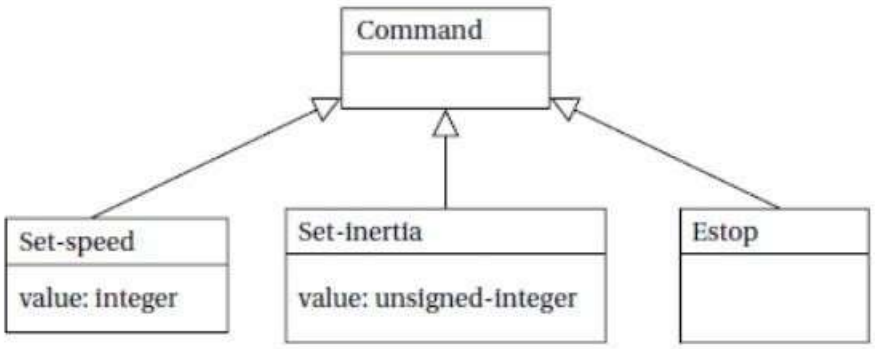


Figure 1.3.3 Class Diagram for the Train Controller Messages

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

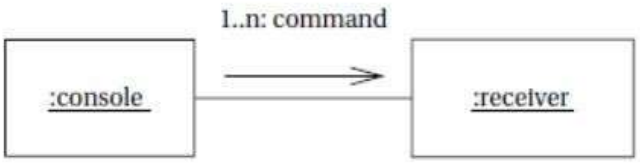


Figure 1.3.4 UML Collaboration Diagram for major Subsystems of the Train Controller System

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Let's break down the command unit and receiver into their major components. The console needs to perform three functions: read the state of the front panel on the command unit, format messages, and transmit messages.

The train receiver must also perform three major functions: receive the message, interpret the message (taking into account the current speed, inertia setting, etc.), and actually control the motor. In this case, let's use a class diagram to represent the design; we could also use an object diagram if we wished.

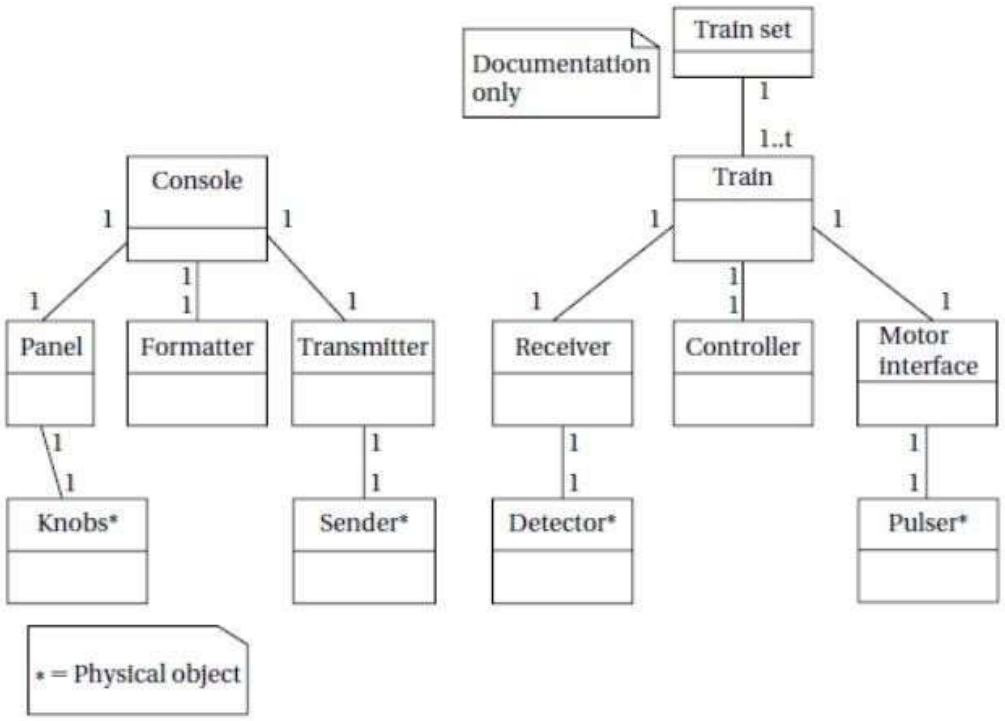


Figure 1.3.4 UML Class Diagram for the Train Controller showing the Composition of the Subsystems

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

The UML class diagram is shown in Figure 1.3.5. It shows the console class using three classes, one for each of its major components. These classes must define some behaviors, but for the moment we will concentrate on the basic characteristics of these classes:

- The *Console* class describes the command unit's front panel, which contains the analog knobs and hardware to interface to the digital parts of the system.
- The *Formatter* class includes behaviors that know how to read the panel knobs and creates a bit stream for the required message.
- The *Transmitter* class interfaces to analog electronics to send the message along the track.

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

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

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

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

- The *Receiver* class knows how to turn the analog signals on the track into digital form.
- The *Controller* class includes behaviors that interpret the commands and 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.

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

DESIGN METHODOLOGY

This section considers the complete design methodology a design process for embedded computing systems. We will start with the rationale for design methodologies, then look at several different methodologies. Process is important because without it, we can't reliably deliver the products we want to create. Thinking about the sequence of steps necessary to build something may seem superfluous. But the fact is that everyone has their own design process, even if they don't articulate it. If you are designing embedded systems in your basement by yourself, having your own work habits is fine. But when several people work together on a project, they need to agree on who will do things and how they will get done. Being explicit about process is important when people work together. Therefore, since many embedded computing systems are too complex to be designed and built by one person, we have to think about design processes.

The obvious goal of a design process is to create a product that does something useful. Typical specifications for a product will include functionality (e.g., cell phone), manufacturing cost (must have a retail price below $200), performance (must power up within 3 s), power consumption (must run for 12 h on two AA batteries), or other properties. Of course, a design process has several important goals beyond function, performance, and power. Three of these goals are summarized below.

- ***Time-to-market:*** Customers always want new features. The product that comes out first can win the market, even setting customer preferences for future generations of the product. The profitable market life for some products is 3-6 months. If you are 3 months late, you will never make money. In some categories, the competition is against the calendar, not just competitors. Calculators, for example, are disproportionately sold just before school starts in the fall. If you miss your market window, you have to wait a year for another sales season.
- ***Design cost:*** Many consumer products are very cost sensitive. Industrial buyers are also increasingly concerned about cost. The costs of designing the system is distinct from manufacturing cost, the cost of engineers' salaries, computers used in design, and so on must be spread across the units sold. In some cases, only one or a few copies of an embedded system may be built, so design costs can dominate manufacturing costs.

Design costs can also be important for high-volume consumer devices when time-to-market pressures cause teams to swell in size.

- ***Quality:*** Customers not only want their products fast and cheap, they also want them to be right. A design methodology that cranks out shoddy products will soon be forced out of the marketplace. Correctness, reliability, and usability must be explicitly addressed from the beginning of the design job to obtain a high-quality product at the end.

Processes evolve over time. They change due to external and internal forces. Customers may change, requirements change, products change, and available components change. Internally, people learn how to do things better, people move on to other projects and others come in, and companies are bought and sold to merge and shape corporate cultures. Software engineers have spent a great deal of time thinking about software design processes. Much of this thinking has been motivated by mainframe software such as databases. But embedded applications have also inspired some important thinking about software design processes.

A good methodology is critical to building systems that work properly. Delivering buggy systems to customers always causes dissatisfaction. But in some applications, such as medical and automotive systems, bugs create serious safety problems that can endanger the lives of users.

DESIGN FLOWS

A ***design flow*** is a sequence of steps to be followed during a design. Some of the steps can be performed by tools, such as compilers or CAD systems; other steps can be performed by hand. In this section we look at the basic characteristics of design flows.

Figure 1.4.1 shows the ***waterfall model***, the first model proposed for the software development process. The waterfall development model consists of five major phases: *requirements* analysis determines the basic characteristics of the system; *architecture* design decomposes the functionality into major components; *coding* implements the pieces and integrates them; *testing* uncovers bugs; and *maintenance* entails deployment in the field, bug fixes, and upgrades. The waterfall model gets its name from the largely one-way flow of work and information from higher levels of abstraction to more detailed design steps (with a limited amount of feedback to the next-higher level of abstraction). Although top-down design is ideal

since it implies good foreknowledge of the implementation during early design phases, most designs are clearly not quite so top-down. Most design projects entail experimentation and changes that require bottom-up feedback. As a result, the waterfall model is today cited as an unrealistic design process. However, it is important to know what the waterfall model is to be able to understand and how others are reacting against it.

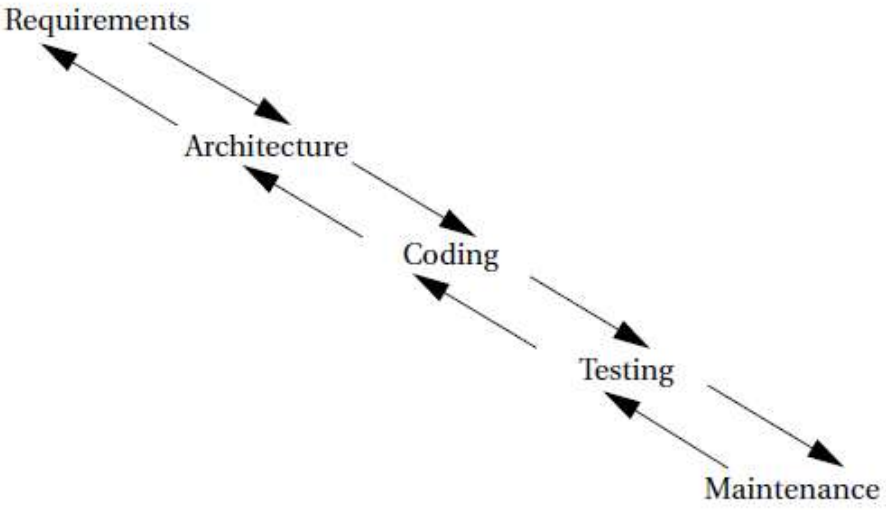


Figure 1.4.1 The waterfall model of software development

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

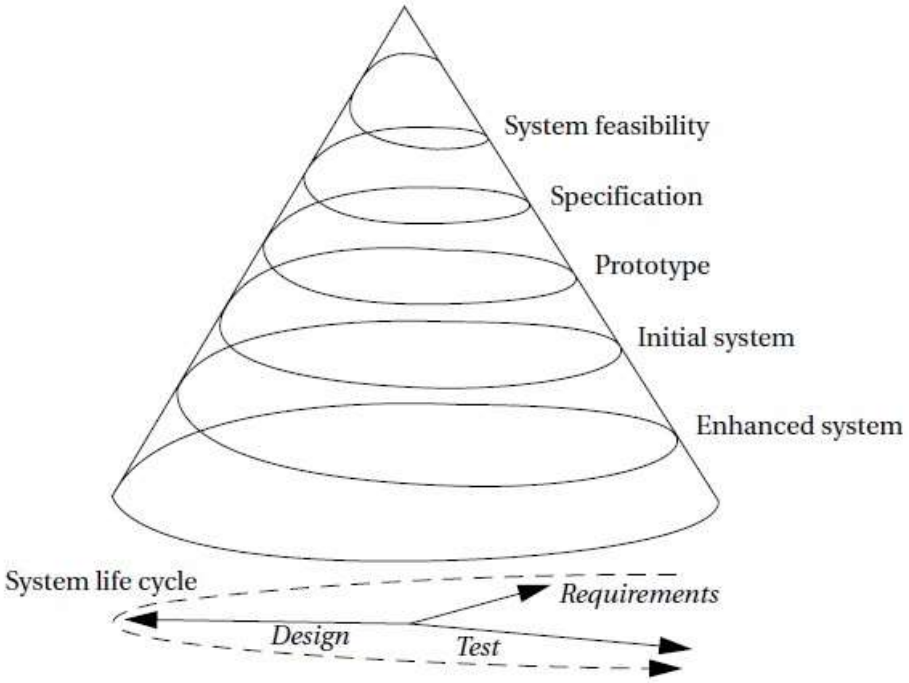


Figure 1.4.2 The spiral model of software design

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Figure 1.4.2 illustrates an alternative model of software development called the ***spiral model***. While the waterfall model assumes that the system is built once in its entirety, the spiral model assumes that several versions of the system will be built. Early systems will be simple mock-ups constructed to aid designer's intuition and to build experience with the system. As design progresses, more complex systems will be constructed. At each level of design, the designers go through requirements, construction, and testing phases. At later stages when more complete versions of the system are constructed, each phase requires more work, widening the design spiral. This successive refinement approach helps the designers understand the system they are working on through a series of design cycles.

The first cycles at the top of the spiral are very small and short, while the final cycles at the spiral's bottom add detail learned from the earlier cycles of the spiral. The spiral model is more realistic than the waterfall model because multiple iterations are often necessary to add enough detail to complete a design. However, a spiral methodology with too many spirals may take too long when design time is a major requirement.

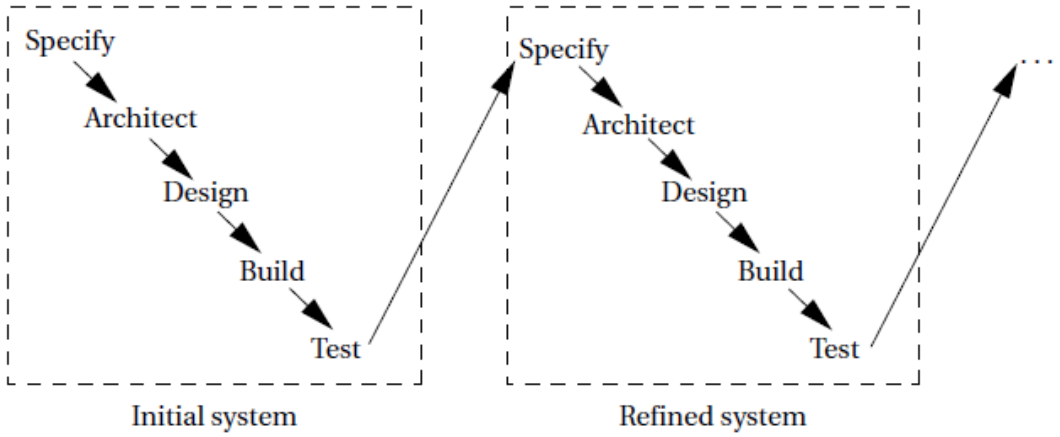


Figure 1.4.3 A successive refinement development mode

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Figure 1.4.3 shows a ***successive refinement*** design methodology. In this approach, the system is built several times. A first system is used as a rough prototype, and successive models of the system are further refined. This methodology makes sense when you are relatively unfamiliar with the application domain for which you are building the system. Refining the system by building several increasingly complex systems allows you to test out architecture and design techniques. The various iterations may also be only partially completed; for

example, continuing an initial system only through the detailed design phase may teach you enough to help you avoid many mistakes in a second design iteration that is carried through to completion.

Embedded computing systems often involve the design of hardware as well as software. Even if you aren't designing a board, you may be selecting boards and plugging together multiple hardware components as well as writing code.

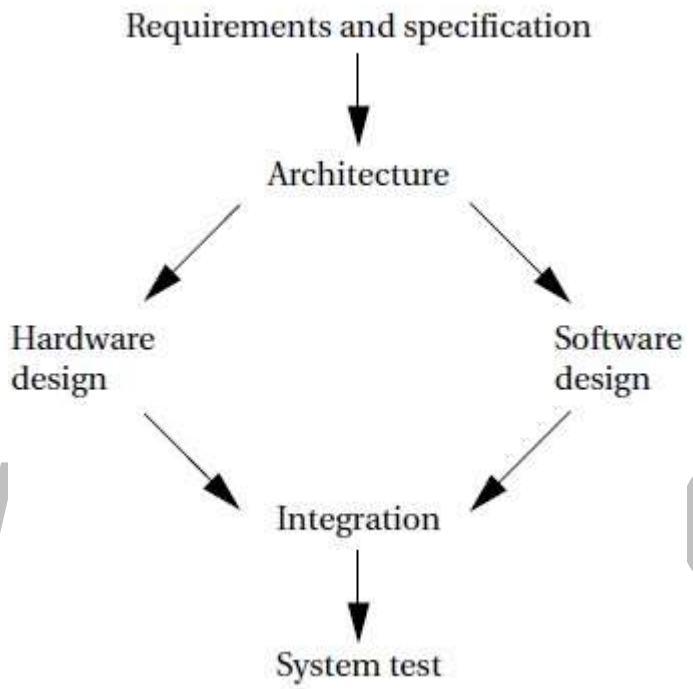


Figure 1.4.4 A simple hardware/software design methodology

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Figure 1.4.4 shows a design methodology for a combined hardware/software project. Front-end activities such as specification and architecture simultaneously consider hardware and software aspects. Similarly, back-end integration and testing consider the entire system. In the middle, however, development of hardware and software components can go on relatively independently, while testing of one will require stubs of the other, most of the hardware and software work can proceed relatively independently. In fact, many complex embedded systems are themselves built of smaller designs.

The complete system may require the design of significant software components, custom logic, and so on, and these in turn may be built from smaller components that need to be designed. The design flow follows the levels of abstraction in the system, from complete system design flows at the most abstract to design flows for individual components.

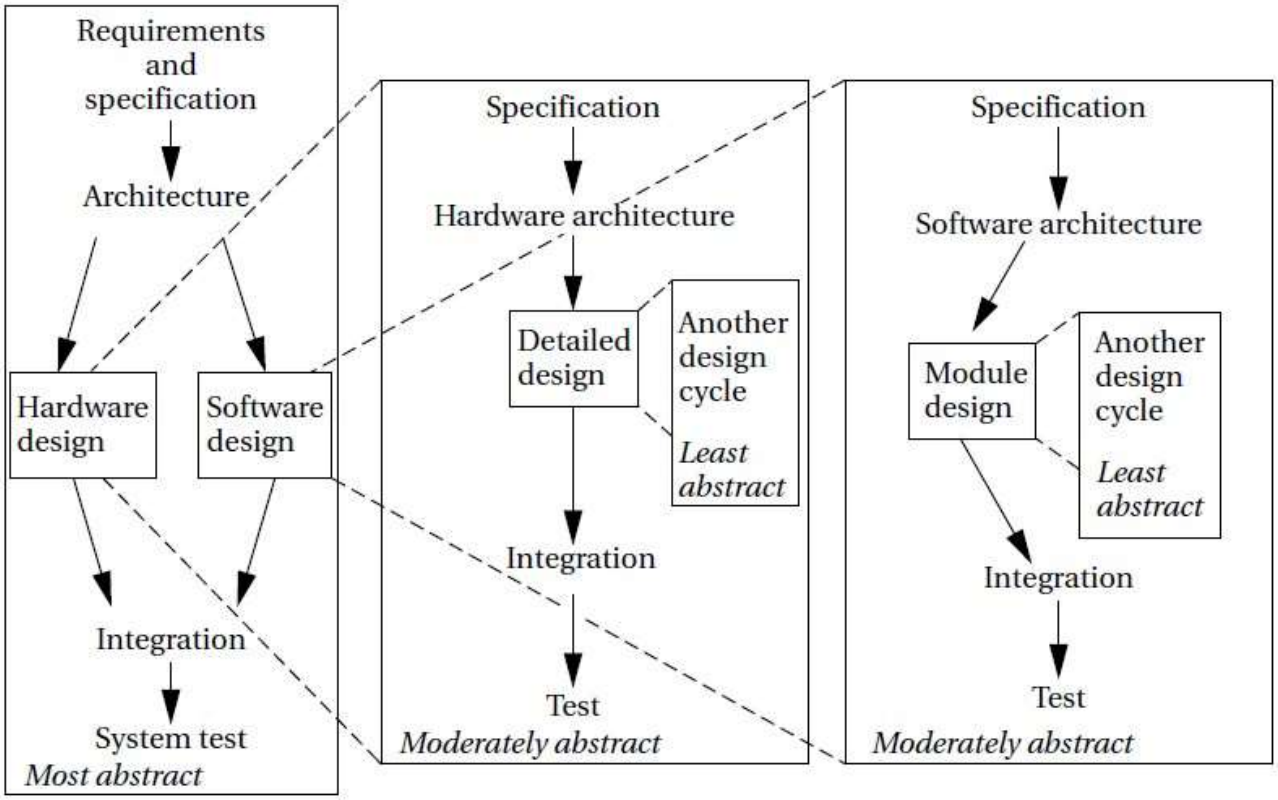


Figure 1.4.5 A hierarchical design flow for an embedded system

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

The design flow for these complex systems resembles the flow shown in Figure 1.4.5. The implementation phase of a flow is itself a complete flow from specification through testing. In such a large project, each flow will probably be handled by separate people or teams. The teams must rely on each other's results. The component teams take their requirements from the team handling the next higher level of abstraction, and the higher-level team relies on the quality of design and testing performed by the component team. Good communication is vital in such large projects. When designing a large system along with many people, it is easy to lose track of the complete design flow and have each designer take a narrow view of his or her role in the design flow.

Concurrent engineering attempts to take a broader approach and optimize the total flow. Reduced design time is an important goal for concurrent engineering, but it can help with

any aspect of the design that cuts across the design flow, such as reliability, performance, power consumption, and so on. It tries to eliminate "over-the-wall" design steps, in which one designer performs an isolated task and then throws the result over the wall to the next designer, with little interaction between the two. In particular, reaping the most benefits from concurrent engineering usually requires eliminating the wall between design and manufacturing. Concurrent engineering efforts are comprised of several elements:

- *Cross-functional teams* include members from various disciplines involved in the process, including manufacturing, hardware and software design, marketing, and so forth.
- *Concurrent product realization* process activities are at the heart of concurrent engineering. Doing several things at once, such as designing various subsystems simultaneously, is critical to reducing design time.
- *Incremental information sharing* and use helps minimize the chance that concurrent product realization will lead to surprises. As soon as new information becomes available, it is shared and integrated into the design. Cross functional teams are important to the effective sharing of information in a timely fashion.
- *Integrated project management* ensures that someone is responsible for the entire project, and that responsibility is not abdicated once one aspect of the work is done.
- *Early and continual supplier involvement* helps make the best use of suppliers' capabilities.
- *Early and continual customer focus* helps ensure that the product best meets customers' needs.

REQUIREMENTS ANALYSIS

Before designing a system, we need to know what we are designing. The terms "requirements" and "specifications" are used in a variety of ways some people use them as synonyms, while others use them as distinct phases. We use them to mean related but distinct steps in the design process. ***Requirements*** are informal descriptions of what the customer wants, while ***specifications*** are more detailed, precise, and consistent descriptions of the

system that can be used to create the architecture. Both requirements and specifications are, however, directed to the outward behavior of the system, not its internal structure.

The overall goal of creating a requirements document is effective communication between the customers and the designers. The designers should know what they are expected to design for the customers; the customers, whether they are known in advance or represented by marketing, should understand what they will get. We have two types of requirements: ***functional*** and ***nonfunctional***. A functional requirement states what the system must do, such as compute an FFT. A nonfunctional requirement can be any number of other attributes, including physical size, cost, power consumption, design time, reliability, and so on.

A good set of requirements should meet several tests:

- *Correctness:* The requirements should not mistakenly describe what the customer wants. Part of correctness is avoiding over-requiring the requirements should not add conditions that are not really necessary.
- *Unambiguousness:* The requirements document should be clear and have only one plain language interpretation.
- *Completeness:* All requirements should be included.
- *Verifiability:* There should be a cost-effective way to ensure that each requirement is satisfied in the final product. For example, a requirement that the system package be “attractive” would be hard to verify without some agreed upon definition of attractiveness.
- *Consistency:* One requirement should not contradict another requirement.
- *Modifiability:* The requirements document should be structured so that it can be modified to meet changing requirements without losing consistency, verifiability, and so forth.
- *Traceability:* Each requirement should be traceable in the following ways:
  - ✓ We should be able to trace backward from the requirements to know why each requirement exists.

- ✓ We should also be able to trace forward from documents created before the requirements (e.g., marketing memos) to understand how they relate to the final requirements.
- ✓ We should be able to trace forward to understand how each requirement is satisfied in the implementation.
- ✓ We should also be able to trace backward from the implementation to know which requirements they were intended to satisfy.

How do you determine requirements? If the product is a continuation of a series, then many of the requirements are well understood. But even in the most modest upgrade, talking to the customer is valuable. In a large company, marketing or sales departments may do most of the work of asking customers what they want, but a surprising number of companies have designers talk directly with customers. Direct customer contact gives the designer an unfiltered sample of what the customer says. It also helps build empathy with the customer, which often pays off in cleaner, easier-to-use customer interfaces. Talking to the customer may also include conducting surveys, organizing focus groups, or asking selected customers to test a mock-up or prototype.

SPECIFICATIONS

In this section we take a look at some advanced techniques for specification and how they can be used.

Control-Oriented Specification Languages

We have already seen how to use state machines to specify control in UML. An example of a widely used state machine specification language is the ***SDL language***, which was developed by the communications industry for specifying communication protocols, telephone systems, and so forth. As illustrated in Figure 1.4.6, SDL specifications include states, actions, and both conditional and unconditional transitions between states. SDL is an event-oriented state machine model since transitions between states are caused by internal and external events.

Other techniques can be used to eliminate clutter and clarify the important structure of a state-based specification. The ***Statechart*** is one well-known technique for state-based

specification that introduced some important concepts. The Statechart notation uses an event-driven model. Statecharts allow states to be grouped together to show common functionality. There are two basic groupings: OR and AND. Figure 1.4.7 shows an example of an OR state by comparing a traditional state transition diagram with a Statechart described via an OR state. The state machine specifies that the machine goes to state *s*4 from any of *s*1, *s*2, or *s*3 when they receive the input *i*2.

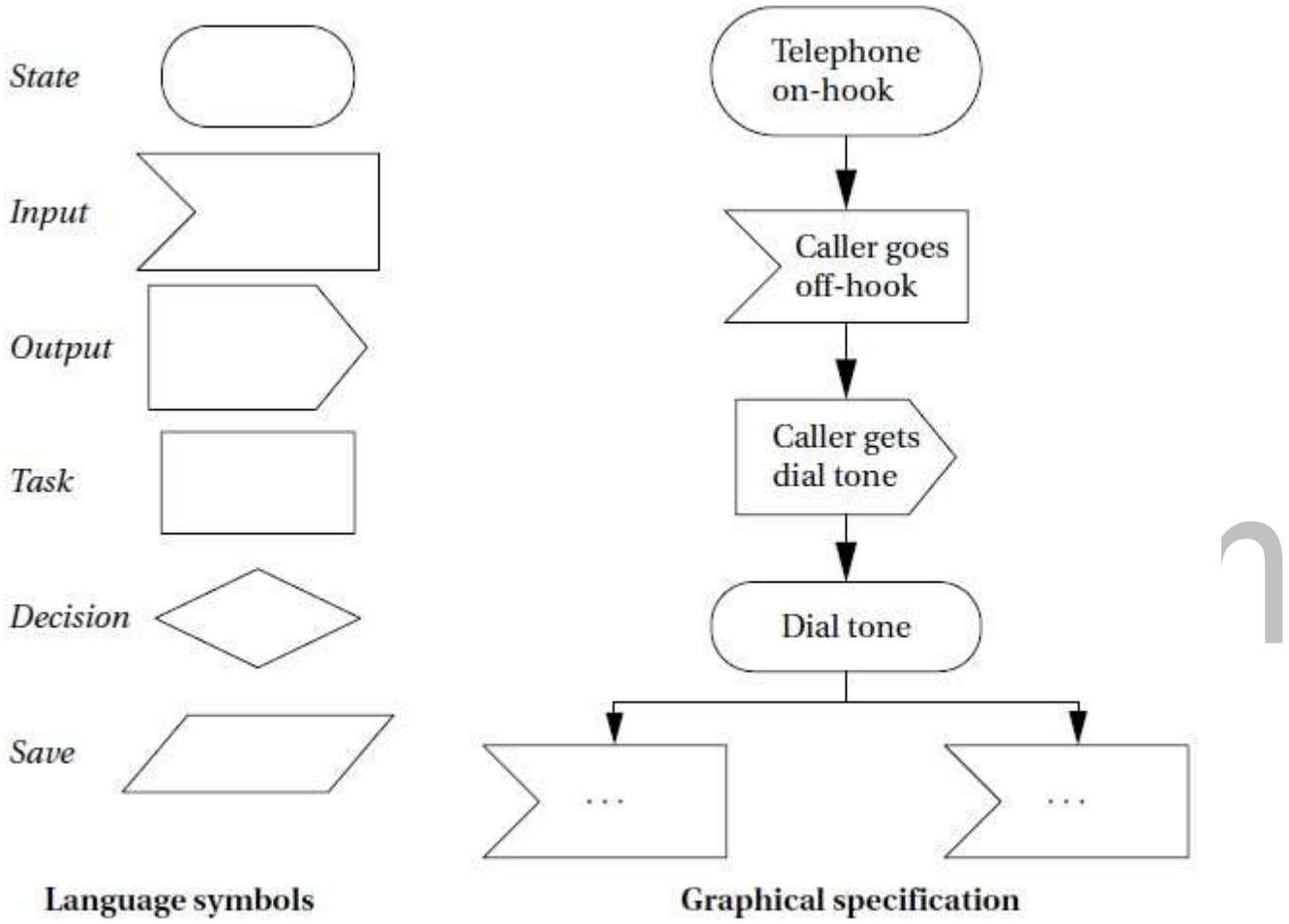


Figure 1.4.6 The SDL specification language

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

The Statechart denotes this commonality by drawing an OR state around *s*1, *s*2, and *s*3 (the name of the OR state is given in the small box at the top of the state). A single transition out of the OR state *s*123 specifies that the machine goes into state *s*4 when it receives the *i*2 input while in any state included in *s*123.The OR state still allows interesting transitions between its member states. There can be multiple ways to get into *s*123 (via *s*1 or *s*2), and

there can be transitions between states within the OR state (such as from *s*1 to *s*3 or *s*2 to *s*3*)*. The OR state is simply a tool for specifying some of the transitions relating to these states.

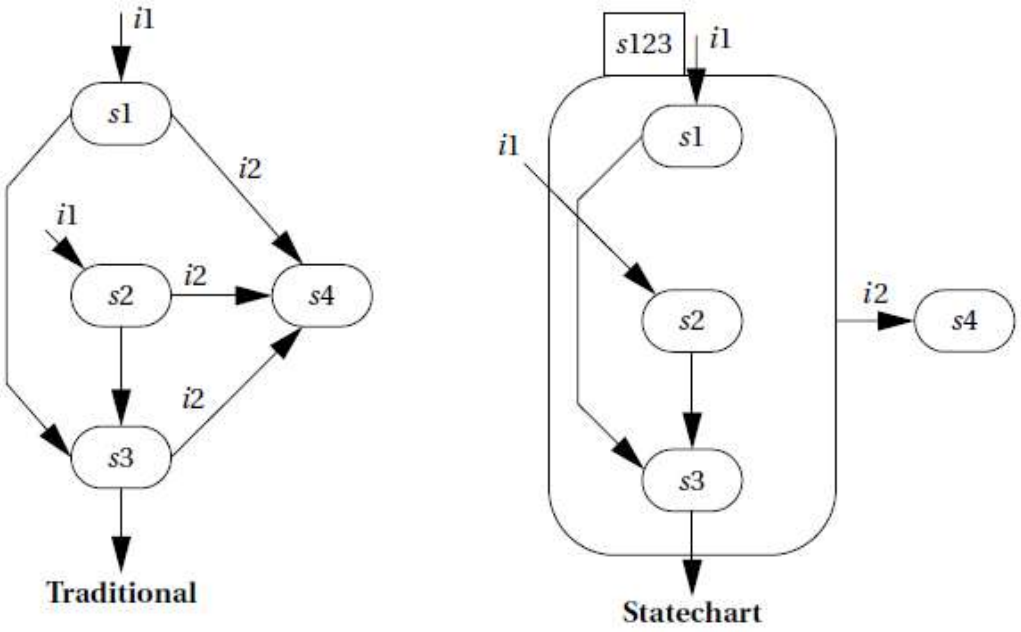


Figure 1.4.7 An OR state in Statecharts

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

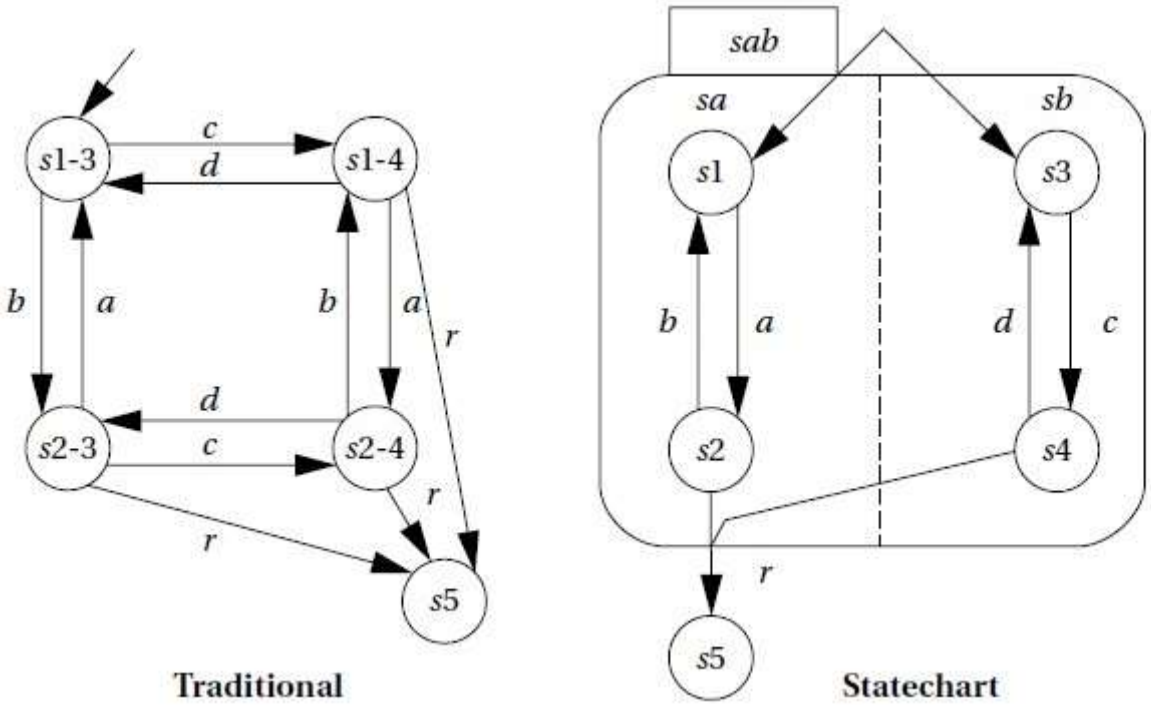


Figure 1.4.8 An AND state in Statecharts

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Figure 1.4.8 shows an example of an AND state specified in Statechart notation as compared to the equivalent in the traditional state machine model. In the traditional model, there are numerous transitions between the states; there is also one entry point into this cluster of states and one exit transition out of the cluster.

In the *Statechart*, the AND state *sab* is decomposed into two components, *sa* and *sb*. When the machine enters the AND state, it simultaneously inhabits the state *s*1 of component *sa* and the state *s*3 of component *sb*. We can think of the system's state as multidimensional. When it enters *sab*, knowing the complete state of the machine requires examining both *sa* and *sb*. The names of the states in the traditional state machine reveal their relationship to the AND state components. Thus, state *s*1-3 corresponds to the Statechart machine having its *sa* component in *s*1 and its *sb* component in *s*3, and so forth. We can exit this cluster of states to go to state *s*5 only when, in the traditional specification, we are in state *s*2-4 and receive input *r*. In the AND state, this corresponds to *sa* in state *s*2, *sb* in state *s*4, and the machine receiving the *r* input while in this composite state. Although the *traditional* and *Statechart* models describe the same behavior, each component has only two states, and the relationships between these states are much simpler to see.

cond1 or (cond2 and !cond3)

Expression

		OR	
A	cond1	T	—
N	cond2	—	T
D	cond3	—	F

Figure 1.4.9 An AND/OR Table

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

The ***AND/OR table***, to describe similar relationships between states. An example AND/OR table and the Boolean expression it describes are shown in Figure 1.4.9. The rows in the AND/OR table are labeled with the basic variables in the expression. Each column corresponds to an AND term in the expression. For example, the AND term (*cond*2 and not *cond*3) is represented in the second column with a *T* for *cond*2, an *F* for *cond*3, and a dash

(don't-care) for *cond*1; this corresponds to the fact that *cond*2 must be *T* and *cond*3 *F* for the AND term to be true. We use the table to evaluate whether a given condition holds in the system. The current states of the variables are compared to the table elements.

A column evaluates to true if all the current variable values correspond to the requirements given in the column. If any one of the columns evaluates to true, then the table's expression evaluates to true, as we would expect for an AND/OR expression. The most important difference between this notation and Statecharts is that don't-cares are explicitly represented in the table, which was found to be of great help in identifying problems in a specification table.

DESIGNING WITH COMPUTING PLATFORMS

System Architecture

We know that architecture is a set of elements and the relationships between them that together form a single unit. The architecture of an embedded computing system is the blueprint for implementing that system it tells you what components you need and how you put them together. The architecture of an embedded computing system includes both hardware and software elements. Let's consider each in turn. The hardware architecture of an embedded computing system is the more obvious manifestation of the architecture since you can touch it and feel it. It includes several elements, some of which may be less obvious than others.

CPU An embedded computing system clearly contains a microprocessor. But which one? There are many different architectures, and even within an architecture we can select between models that vary in clock speed, bus data width, integrated peripherals, and so on. The choice of the CPU is one of the most important, but it cannot be made without considering the software that will execute on the machine.

Bus The choice of a bus is closely tied to that of a CPU, since the bus is an integral part of the microprocessor. But in applications that make intensive use of the bus due to I/O or other data traffic, the bus may be more of a limiting factor than the CPU. Attention must be paid to the required data bandwidths to be sure that the bus can handle the traffic.

Memory Once again, the question is not whether the system will have memory but the characteristics of that memory. The most obvious characteristic is total size, which depends on both the required data volume and the size of the program instructions. The ratio of ROM to RAM and selection of DRAM versus SRAM can have a significant influence on the cost of the system. The speed of the memory will play a large part in determining system performance.

Input and output devices The user's view of the input and output mechanisms may not correspond to the devices connected to the microprocessor.

For example, a set of switches and knobs on a front panel may all be controlled by a single microcontroller, which is in turn connected to the main CPU. For a given function, there may be several different devices of varying sophistication and cost that can do the job. The

difficulty of using a particular device, such as the amount of glue logic required to interface it, may also play a role in final device selection.

Hardware Design

The design complexity of the hardware platform can vary greatly, from a totally off-the-shelf solution to a highly customized design. At the board level, the first step is to consider **evaluation boards** supplied by the microprocessor manufacturer or another company working in collaboration with the manufacturer. Evaluation boards are sold for many microprocessor systems; they typically include the CPU, some memory, a serial link for downloading programs, and some minimal number of I/O devices. Figure 1.7.1 shows an ARM evaluation board manufactured by Sharp.

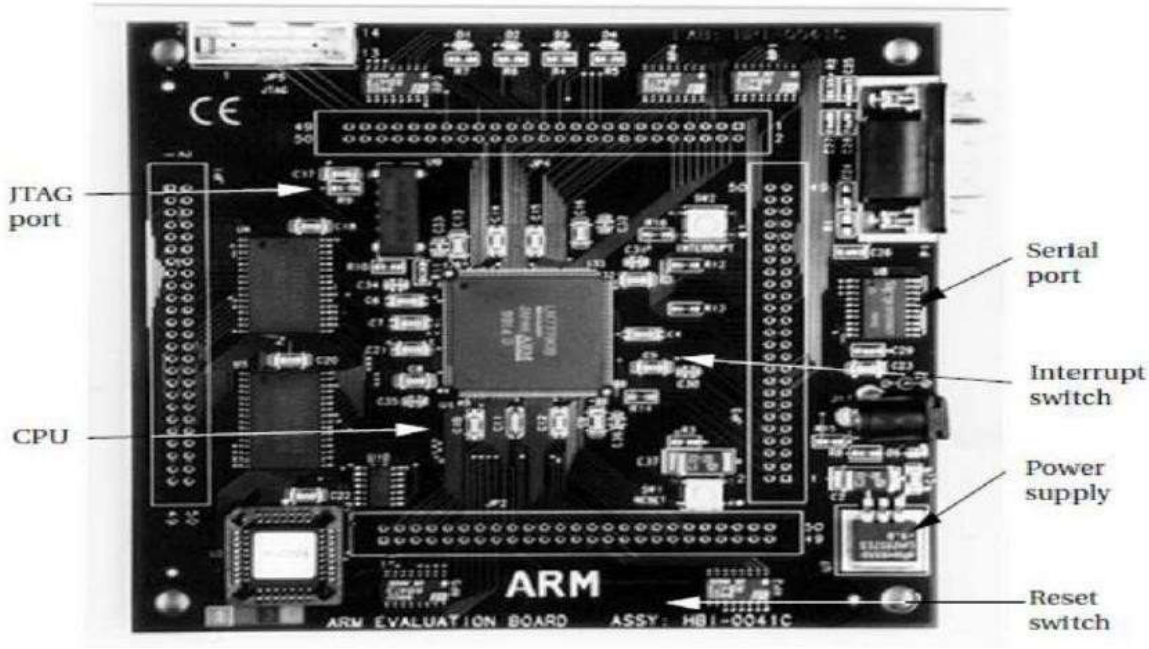


Figure 1.7.1 An ARM evaluation board

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

The evaluation board may be a complete solution or provide what you need with only slight modifications. If the evaluation board is supplied by the microprocessor vendor, its design (netlist, board layout, etc.) may be available from the vendor; companies provide such information to make it easy for customers to use their microprocessors. If the evaluation board comes from a third party, it may be possible to contract them to design a new board with your required modifications, or you can start from scratch on a new board design. The other major task is the choice of memory and peripheral components. In the case of I/O devices, there are

two alternatives for each device: selecting a component from a catalog or designing one yourself. When shopping for devices from a catalog, it is important to read data sheets carefully it may not be trivial to figure out whether the device does what you need it to do.

Development Environments

A typical embedded computing system has a relatively small amount of everything, including CPU horsepower, memory, I/O devices, and so forth. As a result, it is common to do at least part of the software development on a PC or workstation known as a **host** as illustrated in Figure The hardware on which the code will finally run is known as the **target**. The host and target are frequently connected by a USB link, but a higher-speed link such as Ethernet can also be used.

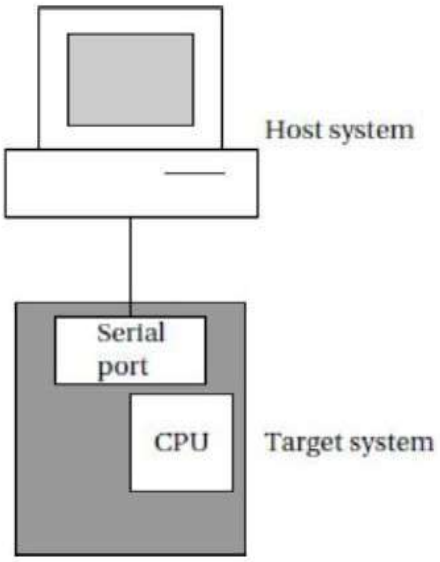


Figure 1.7.2 Connecting a host and a target system

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

The target must include a small amount of software to talk to the host system. That software will take up some memory, interrupt vectors, and so on, but it should generally leave the smallest possible footprint in the target to avoid interfering with the application software. The host should be able to do the following:

- load programs into the target,
- start and stop program execution on the target, and
- examine memory and CPU registers

Debugging Techniques:

A good deal of software debugging can be done by compiling and executing the code on a PC or workstation. But at some point it inevitably becomes necessary to run code on the embedded hardware platform. Embedded systems are usually less friendly programming environments than PCs. Nonetheless, the resourceful designer has several options available for debugging the system. The serial port found on most evaluation boards is one of the most important debugging tools. In fact, it is often a good idea to design a serial port into an embedded system even if it will not be used in the final product; the serial port can be used not only for development debugging but also for diagnosing problems in the field. Another very important debugging tool is the **breakpoint**.

The simplest form of a breakpoint is for the user to specify an address at which the program's execution is to break. When the PC reaches that address, control is returned to the monitor program. From the monitor program, the user can examine and/or modify CPU registers, after which execution can be continued. Implementing breakpoints does not require using exceptions or external devices.

Debugging Challenges

Logical errors in software can be hard to track down, but errors in real-time code can create problems that are even harder to diagnose. Real-time programs are required to finish their work within a certain amount of time; if they run too long, they can create very unexpected behavior.

The exact results of missing real-time deadlines depend on the detailed characteristics of the I/O devices and the nature of the timing violation. This makes debugging real-time problems especially difficult. Unfortunately, the best advice is that if a system exhibits truly unusual behavior, missed deadlines should be suspected. In-circuit emulators, logic analyzers, and even LEDs can be useful tools in checking the execution time of real-time code to determine whether it in fact meets its deadline.

EMBEDDED SYSTEM DESIGN PROCESS

This section provides an overview of the embedded system design process aimed at two objectives. First, it will give us an introduction to the various steps in embedded system design before we delve into them in more detail. Second, it will allow us to consider the design ***methodology*** itself. A design methodology is important for three reasons.

First, it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as optimizing ***performance*** or performing 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 semi automating) 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.

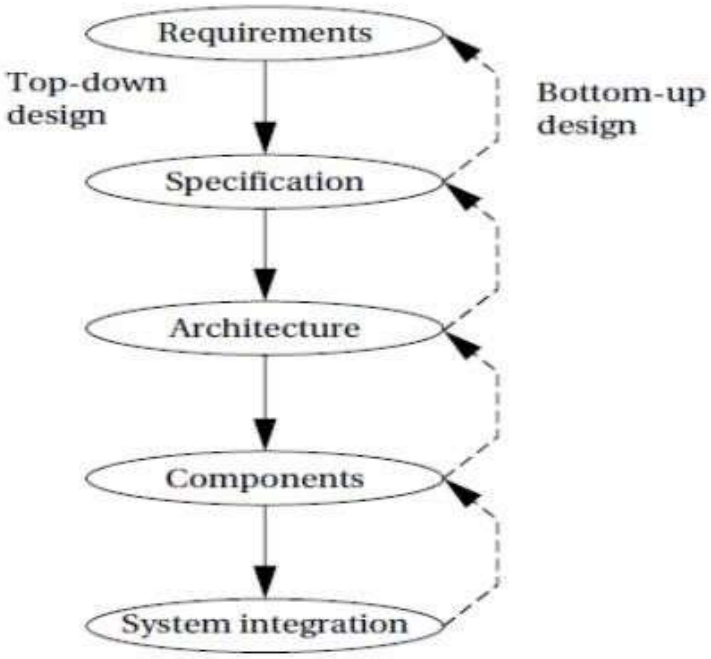


Figure 1.2.1 Embedded System Design Process

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Figure 1.2.1 summarizes the major steps in the embedded system design process. In this top-down view, we start with the system ***requirements***. In the next step, ***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)
- Power consumption

We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail: We must *analyze* the design at each step to determine how we can meet the specifications. We must then *refine* the design to add detail. We must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

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.

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.

Consumers of embedded systems are usually not themselves embedded system designers or even product designers. Their understanding of the system is based on how they envision users' interactions with the system. They may have unrealistic expectations as to what can be done within their budgets; and they may also express their desires in a language very different from system architects' jargon.

Capturing a consistent set of requirements from the customer and then massaging those requirements into a more formal specification is a structured way to manage the process of translating from the consumer's language to the designer's.

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

- ***Performance:*** The speed of the system is often a major consideration both for the usability of the system and 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 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.

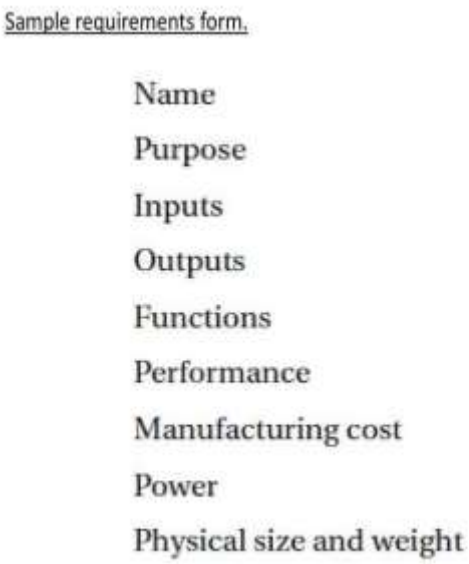


Figure 1.2.2 Sample Requirements Form

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

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

Figure 1.2.2 shows a sample ***requirements form*** that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of 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:
    a. Types *of data:* Analog electronic signals? Digital data? Mechanical inputs?
    b. *Data characteristics:* Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?
    c. *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 components. Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range. Cost has a substantial influence on architecture: A machine that is meant to sell at $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.

PLATFORM-LEVEL PERFORMANCE ANALYSIS

Bus based systems add another layer of complication to performance analysis platform level performance involve much more than the CPU we often focus on the CPU because it processes instructions but any part of the system can affect total system performance. More precisely the CPU provides an upper bound on performance but any other part of the system can slow down the CPU merely counting instruction execution times is not enough.

Consider the simple system we want to move data from memory to the CPU to process it to get the data from memory to the CPU we must:

- Read from the memory
- Transfer over the bus to the cache
- Transfer from the cache to the CPU

The time required to transfer from the cache to the CPU is included in the instruction execution time, but the other two times are not.

Bandwidth as performance

The most basic measure of performance we are interested in is bandwidth the rate at which we can move data ultimately if we are interested in real time performance we are interested in real time performance measured in seconds but often the simplest way to measure performance is in units of clock cycles however different parts of the system will run at different clock rates. We have to make sure that we apply the right clock rate to each part of the performance estimate when we convert from clock cycles to seconds

Bus bandwidth

Bandwidth questions often come up when we are transferring large blocks of data for simplicity let’s start by considering the bandwidth provided by only one system component the bus consider and image of 320 pixels with each pixel composed of 3 bytes of data this gives a grand total of 230400 bytes of data if these images are video frames, we want to check if we can push one frame through the system within the 1/30 sec that we have to process a frame before the next one arrives.

Let us assume that we can transfer one byte of data every microsecond which implies a bus speed of 1 MHz. In this case we would require 230400 µs = 0.23 sec to transfer one frame that is more than the 0.033 sec allotted to the data transfer we would have to increase the transfer rate by 7xto satisfy our performance requirement.

We can increase bandwidth in two ways we can increase the clock rate of the bus or we can increase the amount of data transferred per clock cycle for example if we increased the bus to carry four bytes or 32 bits per transfer we would reduce the transfer time to 0.058 sec if we could also increase the bus clock rate to 2 Mhz. then we would reduce the transfer time to 0.029sec, which is within our time budget for the transfer.

Analyze the Program-Level Performance

Because embedded systems must perform functions in real time we often need to know how fast a program runs. The techniques we use to analyze program execution time are also helpful in analyzing properties such as power consumption. In this section, we study how to analyze programs to estimate their run times. We also examine how to optimize programs to improve their execution times; of course, optimization relies on analysis.

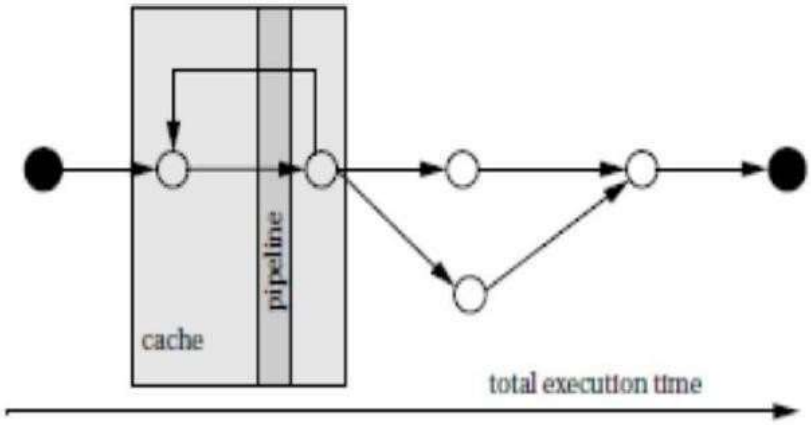


Figure 1.9.1 The CPU Pipeline and Cache act as Windows into Program

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

It is important to keep in mind that CPU performance is not judged in the same way as program performance. Certainly, CPU clock rate is a very unreliable metric for program performance. But more importantly, the fact that the CPU executes part of our program quickly does not mean that it will execute the entire program at the rate we desire. As illustrated in Figure 1.9.1, the CPU pipeline and cache act as windows into our program. In order to

understand the total execution time of our program, we must look at execution paths, which in general are far longer than the pipeline and cache windows.

The pipeline and cache influence execution time, but execution time is a global property of the program. While we might hope that the execution time of programs could be precisely determined, this is in fact difficult to do in practice:

- The execution time of a program often varies with the input data values because those values select different execution paths in the program. For example, loops may be executed a varying number of times, and different branches may execute blocks of varying complexity.
- The cache has a major effect on program performance, and once again, the cache's behavior depends in part on the data values input to the program.
- Execution times may vary even at the instruction level. Floating-point operations are the most sensitive to data values, but the normal integer execution pipeline can also introduce data-dependent variations. In general, the execution time of an instruction in a pipeline depends not only on that instruction but on the instructions around it in the pipeline.

We can measure program performance in several ways:

- Some microprocessor manufacturers supply simulators for their CPUs: The simulator runs on a workstation or PC, takes as input an executable for the microprocessor along with input data, and simulates the execution of that program. Some of these simulators go beyond functional simulation to measure the execution time of the program. Simulation is clearly slower than executing the program on the actual microprocessor, but it also provides much greater visibility during execution. Be careful some microprocessor performance simulators are not 100% accurate, and simulation of I/O-intensive code may be difficult.
- A timer connected to the microprocessor bus can be used to measure performance of executing sections of code. The code to be measured would reset and start the timer at its start and stop the timer at the end of execution. The length of the program that can be measured is limited by the accuracy of the timer.

- A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment. This technique relies on the code being able to produce identifiable events on the bus to identify the start and stop of execution. The length of code that can be measured is limited by the size of the logic analyzers buffer. We are interested in the following three different types of performance measures on programs:
  - ✓ **Average-case execution time** This is the typical execution time we would expect for typical data. Clearly, the first challenge is defining typical inputs.
  - ✓ **Worst-case execution time** The longest time that the program can spend on any input sequence is clearly important for systems that must meet deadlines. In some cases, the input set that causes the worst-case execution time is obvious, but in many cases it is not.
  - ✓ **Best-case execution time** This measure can be important in Multirate real-time systems.

First, we look at the fundamentals of program performance in more detail. We then consider trace driven performance based on executing the program and observing its behavior.

Elements of Program Performance

The path is the sequence of instructions executed by the program (or its equivalent in the high-level language representation of the program). The instruction timing is determined based on the sequence of instructions traced by the program path, which takes into account data dependencies, pipeline behaviour, and caching. Luckily, these two problems can be solved relatively independently.

Although we can trace the execution path of a program through its high-level language specification, it is hard to get accurate estimates of total execution time from a high-level language program. The number of memory locations and variables must be estimated, and results may be either saved for reuse or recomputed on the fly, among other effects. These problems become more challenging as the compiler puts more and more effort into optimizing the program. However, some aspects of program performance can be estimated by looking directly at the C program. For example, if a program contains a loop with a large, fixed iteration

bound or if one branch of a conditional is much longer than another, we can get at least a rough idea that these are more time-consuming.

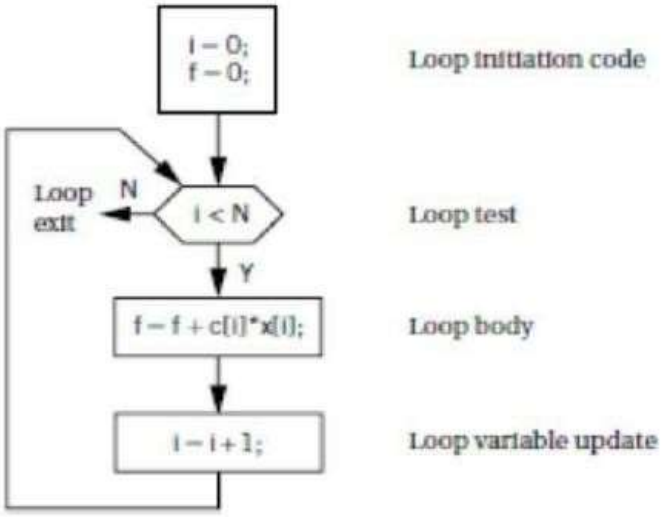


Figure 1.9.2 Segments of the program

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

A precise estimate of performance also relies on the instructions to be executed, since different instructions take different amounts of time. (In addition, to make life even more difficult, the execution time of one instruction can depend on the instructions executed before and after it. To measure the longest path length, we must find the longest path through the optimized CDFG since the compiler may change the structure of the control and data flow to optimize the program's implementation. It is important to keep in mind that choosing the longest path through a CDFG as measured by the number of nodes or edges touched may not correspond to the longest execution time. Since the execution time of a node in the CDFG will vary greatly depending on the instructions represented by that node, we must keep in mind that the longest path through the CDFG depends on the execution times of the nodes.

In general, it is good policy to choose several of what we estimate are the longest paths through the program and measure the lengths of all of them in sufficient detail to be sure that we have in fact captured the longest path. Once we know the execution path of the program, we have to measure the execution time of the instructions executed along that path. The simplest estimate is to assume that every instruction takes the same number of clock cycles, which means we need only count the instructions and multiply by the per-instruction execution

time to obtain n the program's total execution time. However, even ignoring cache effects, this technique is simplistic for the reasons summarized below.

- Not all instructions take the same amount of time. Although RISC architectures tend to provide uniform instruction execution times in order to keep the CPU's pipeline full, even many RISC architectures take different amounts of time to execute certain instructions. Multiple load-store instructions are examples of longer-executing instructions in the ARM architecture. Floating point instructions show especially wide variations in execution time—while basic multiply and add operations are fast, some transcendental functions can take thousands of cycles to execute.
- Execution times of instructions are not independent. The execution time of one instruction depends on the instructions around it. For example, many CPUs use register bypassing to speed up instruction sequences when the result of one instruction is used in the next instruction. As a result, the execution time of an instruction may depend on whether its destination register is used as a source for the next operation (or vice versa).
- The execution time of an instruction may depend on operand values. This is clearly true of floating-point instructions in which a different number of iterations may be required to calculate the result. Other specialized instructions can, for example, perform a data-dependent number of integer operations.

Measurement-Driven Performance Analysis

Most methods of measuring program performance combine the determination of the execution path and the timing of that path: as the program executes, it chooses a path and we observe the execution time along that path. We refer to the record of the execution path of a program as a **program trace** (or more succinctly, a **trace**). Traces can be valuable for other purposes, such as analyzing the cache behavior of the program. Perhaps the biggest problem in measuring program performance is figuring out a useful set of inputs to provide to the program. This problem has two aspects. First, we have to determine the actual input values. We may be able to use benchmark data sets or data captured from a running system to help us generate typical values. For simple programs, we may be able to analyze the algorithm to determine the inputs that cause the worst-case execution time.

The other problem with input data is the **software scaffolding** that we may need to feed data into the program and get data out. When we are designing a large system, it may be difficult to extract out part of the software and test it independently of the other parts of the system. We may need to add new testing modules to the system software to help us introduce testing values and to observe testing outputs.

We can measure program performance either directly on the hardware or by using a simulator. Each method has its advantages and disadvantages. Physical measurement requires some sort of hardware instrumentation. The most direct method of measuring the performance of a program would be to watch the program counter's value: start a timer when the PC reaches the program's start, stop the timer when it reaches the program's end. Unfortunately, it generally isn't possible to directly observe the program counter.

However, it is possible in many cases to modify the program so that it starts a timer at the beginning of execution and stops the timer at the end. While this doesn't give us direct information about the program trace, it does give us execution time. If we have several timers available, we can use them to measure the execution time of different parts of the program.

A logic analyzer or an oscilloscope can be used to watch for signals that mark various points in the execution of the program. However, because logic analyzers have a limited amount of memory, this approach doesn't work well for programs with extremely long execution times.

Some CPUs have hardware facilities for automatically generating trace information. For example, the Pentium family microprocessors generate a special bus cycle, a branch trace message, that shows the source and/or destination address of a branch [Col97]. If we record only traces, we can reconstruct the instructions executed within the basic blocks while greatly reducing the amount of memory required to hold the trace. The alternative to physical measurement of execution time is simulation. A CPU simulator is a program that takes as input a memory image for a CPU and performs the operations on that memory image that the actual CPU would perform, leaving the results in the modified memory image.

For purposes of performance analysis, the most important type of CPU simulator is the **cycle-accurate simulator**, which performs a sufficiently detailed simulation of the processor's internals so that it can determine the exact number of clock cycles required for execution. A cycle-accurate simulator is built with detailed knowledge of how the processor

works, so that it can take into account all the possible behaviours of the micro architecture that may affect execution time. Cycle-accurate simulators are slower than the processor itself, but a variety of techniques can be used to make them surprisingly fast, running only hundreds of times slower than the hardware itself.

A cycle-accurate simulator has a complete model of the processor, including the cache. It can therefore provide valuable information about why the program runs too slowly. The next example discusses a simulator that can be used to model many different processors.

Performance Optimization

Loop Optimizations:

Loops are important targets for optimization because programs with loops tend to spend a lot of time executing those loops. There are three important techniques in optimizing loops: **code motion, induction variable elimination,** and **strength reduction**. Code motion lets us move unnecessary code out of a loop. If a computation's result does not depend on operations performed in the loop body, then we can safely move it out of the loop. Code motion opportunities can arise because programmers may find some computations clearer and more concise when put in the loop body, even though they are not strictly dependent on the loop iterations.

An **induction variable** is a variable whose value is derived from the loop iteration variable's value. The compiler often introduces induction variables to help it implement the loop. Properly transformed, we may be able to eliminate some variables and apply strength reduction to others. A nested loop is a good example of the use of induction variables.

Cache Optimizations:

A **loop nest** is a set of loops, one inside the other. Loop nests occur when we process arrays. A large body of techniques has been developed for optimizing loop nests. Rewriting a loop nest changes the order in which array elements are accessed. This can expose new parallelism opportunities that can be exploited by later stages of the compiler, and it can also improve cache performance.

QUALITY ASSURANCE TECHNIQUES

Introduction

The quality of a product or service can be judged by how well it satisfies its intended function. A product can be of low quality for several reasons, such as it was shoddily manufactured, its components were improperly designed, its architecture was poorly conceived, and the product's requirements were poorly understood. Quality must be designed in. You can't test out enough bugs to deliver a high-quality product. The ***quality assurance (QA)*** process is vital for the delivery of a satisfactory system. In this section we will concentrate on portions of the methodology particularly aimed at improving the quality of the resulting system. The software testing techniques described earlier in the book constitute one component of quality assurance, but the pursuit of quality extends throughout the design flow.

For example, settling on the proper requirements and specification cannot be overlooked as an important determinant of quality. If the system is too difficult to design, it will probably be difficult to keep it working properly. Customers may desire features that sound nice but in fact don't add much to the overall usefulness of the system. In many cases, having too many features only makes the design more complicated and the final device more prone to breakage. To help us understand the importance of QA, application example serious safety problems in one computer-controlled medical system. Medical equipment, like aviation electronics, is a safety-critical application; unfortunately, this medical equipment caused deaths before its design errors were properly understood. This example also allows us to use specification techniques to understand software design problems. In the rest of the section, we look at several ways of improving quality: design reviews, measurement-based QA, and techniques for debugging large systems.

Quality Assurance Techniques

The International Standards Organization (ISO) has created a set of quality standards known as ***ISO 9000***. ISO 9000 was created to apply to a broad range of industries, including but not limited to embedded hardware and software. A standard developed for a particular product, such as wooden construction beams, could specify criteria particular to that product, such as the load that a beam must be able to carry. However, a wide-ranging standard such as

ISO 9000 cannot specify the detailed standards for every industry. Consequently, ISO 9000 concentrates on processes used to create the product or service. The processes used to satisfy ISO 9000 affect the entire organization as well as the individual steps taken during design and manufacturing. We can, however, make the following observations about quality management based on ISO 9000:

- *Process is crucial:* Haphazard development leads to haphazard products and low quality. Knowing what steps are to be followed to create a high quality product is essential to ensuring that all the necessary steps are in fact followed.
- *Documentation is important:* Documentation has several roles: The creation of the documents describing processes helps those involved understand the processes; documentation helps internal quality monitoring groups to ensure that the required processes are actually being followed; and documentation also helps outside groups (customers, auditors, etc.) understand the processes and how they are being implemented.
- *Communication is important:* Quality ultimately relies on people. Good documentation is an aid for helping people understand the total quality process. The people in the organization should understand not only their specific tasks but also how their jobs can affect overall system quality.

Many types of techniques can be used to verify system designs and ensure quality. Techniques can be either *manual* or *tool based*. Manual techniques are surprisingly effective in practice. Design reviews, which are simply meetings at which the design is discussed and which are very successful in identifying bugs. Many of the software testing techniques can be applied manually by tracing through the program to determine the required tests. Tool-based verification helps considerably in managing large quantities of information that may be generated in a complex design. Test generation programs can automate much of the drudgery of creating test sets for programs. Tracking tools can help ensure that various steps have been performed. Design flow tools automate the process of running design data through other tools. Metrics are important to the quality control process. To know whether we have achieved high levels of quality, we must be able to measure aspects of the system and our design process. We can measure certain aspects of the system itself, such as the execution speed of programs

or the coverage of test patterns. We can also measure aspects of the design process, such as the rate at which bugs are found. Section describes ways in which measurements can be used in the QA process. Tool and manual techniques must fit into an overall process. The details of that process will be determined by several factors, including the type of product being designed (e.g., video game, laser printer, air traffic control system), the number of units to be manufactured and the time allowed for design, the existing practices in the company into which any new processes must be integrated, and many other factors. An important role of ISO 9000 is to help organizations study their total process, not just particular segments that may appear to be important at a particular time.

One well-known way of measuring the quality of an organization's software development process is the ***Capability Maturity Model (CMM)*** developed by Carnegie Mellon University's Software Engineering Institute. The CMM provides a model for judging an organization. It defines the following five levels of maturity:

1. *Initial:* A poorly organized process, with very few well-defined processes. Success of a project depends on the efforts of individuals, not the organization itself.
2. *Repeatable:* This level provides basic tracking mechanisms that allow management to understand cost, scheduling, and how well the systems under development meet their goals.
3. *Defined:* The management and engineering processes are documented and standardized. All projects make use of documented and approved standard methods.
4. *Managed:* This phase makes detailed measurements of the development process and product quality.
5. *Optimizing:* At the highest level, feedback from detailed measurements is used to continually improve the organization's processes.

The Software Engineering Institute has found very few organizations anywhere in the world that meet the highest level of continuous improvement and quite a few organizations that operate under the chaotic processes of the initial level. However, the CMM provides a benchmark by which organizations can judge themselves and use that information for improvement.

Verifying the Specification

The requirements and specification are generated very early in the design process. Verifying the requirements and specification is very important for the simple reason that bugs in the requirements or specification can be extremely expensive to fix later on. Figure 1.6.1 shows how the cost of fixing bugs grows over the course of the design process (we use the waterfall model as a simple example, but the same holds for any design flow). The longer a bug survives in the system, the more expensive it will be to fix. A coding bug, if not found until after system deployment, will cost money to recall and reprogram existing systems, among other things. But a bug introduced earlier in the flow and not discovered until the same point will accrue all those costs and more costs as well.

A bug introduced in the requirements or specification and left until maintenance could force an entire redesign of the product, not just the replacement of a ROM. Discovering bugs early is crucial because it prevents bugs from being released to customers, minimizes design costs, and reduces design time. While some requirements and specification bugs will become apparent in the detailed design stages. For example, as the consequences of certain requirements are better understood, it is possible and desirable to weed out many bugs during the generation of the requirements and spec.

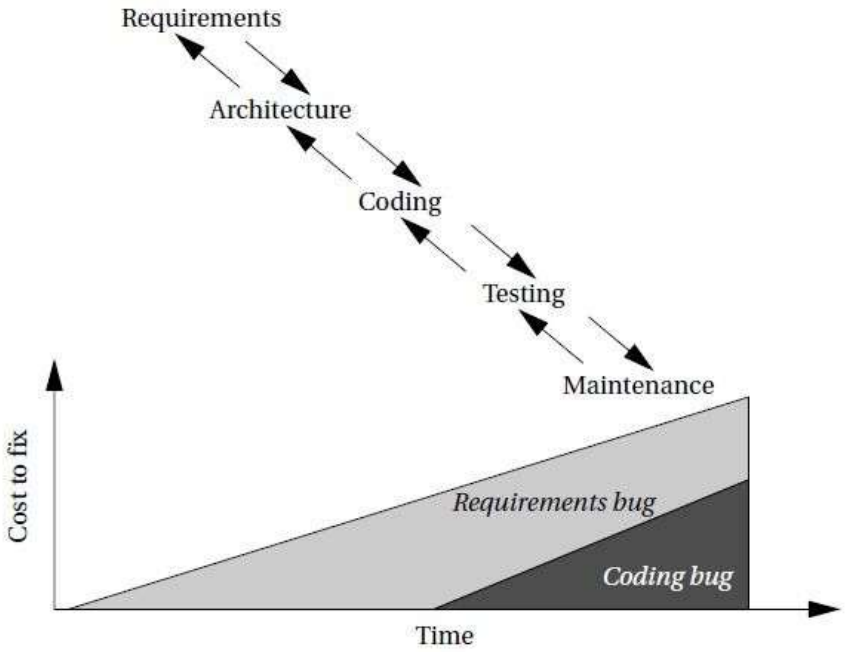


Figure 1.6.1 Long-lived bugs are more expensive to fix

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

The goal of validating the requirements and specification is to ensure that they satisfy the criteria to create the specification, including correctness, completeness, consistency, and so on. Validation is in fact part of the effort of generating the requirements and specification. Some techniques can be applied while they are being created to help you understand the requirements and specifications, while others are applied on a draft, with results used to modify the specs. Since requirements come from the customer and are inherently somewhat informal, it may seem like a challenge to validate them. However, there are many things that can be done to ensure that the customer and the person actually writing the requirements are communicating. ***Prototypes*** are a very useful tool when dealing

with end users rather than simply describe the system to them in broad, technical terms, a prototype can let them see, hear, and touch at least some of the important aspects of the system. Of course, the prototype will not be fully functional since the design work has not yet been done. However, user interfaces in particular are well suited to prototyping and user testing. Canned or randomly generated data can be used to simulate the internal operation of the system. A prototype can help the end user critique numerous functional and nonfunctional requirements, such as data displays, speed of operation, size, weight, and so forth.

Certain programming languages, sometimes called ***prototyping languages*** or ***specification languages***, are especially well suited to prototyping. Very high-level languages (such as MATLAB in the signal processing domain) may be able to perform functional attributes, such as the mathematical function to be performed, but not nonfunctional attributes such as the speed of execution. ***Preexisting systems*** can also be used to help the end user articulate his or her needs. Specifying what someone does or doesn't like about an existing machine is much easier than having them talk about the new system in the abstract. In some cases, it may be possible to construct a prototype of the new system from the preexisting system. Particularly when designing **cyber-physical systems** that use real-time computers for physical control, simulation is an important technique for validating requirements. Requirements for cyber-physical systems depend in part on the physical properties of the plant being controlled. Simulators that model the physical plant can help system designers understand the requirements on the cyber side of the system.

The techniques used to validate requirements are also useful in verifying that the specifications are correct. Building prototypes, specification languages, and comparisons to preexisting systems are as useful to system analysis and designers as they are to end users. Auditing tools may be useful in verifying consistency, completeness, and so forth. Working through ***usage scenarios*** often helps designers fill out the details of a specification and ensure its completeness and correctness. In some cases, ***formal techniques*** (that is, design techniques that make use of mathematical proofs) may be useful. Proofs may be done either manually or automatically. In some cases, proving that a particular condition can or cannot occur according to the specification is important. Automated proofs are particularly useful in certain types of complex systems that can be specified succinctly but whose behavior over time is complex. For example, complex protocols have been successfully formally verified.

Design Reviews

The ***design review*** is a critical component of any QA process. The design review is a simple, low-cost way to catch bugs early in the design process. A design review is simply a meeting in which team members discuss a design,

reviewing how a component of the system works. Some bugs are caught simply by preparing for the meeting, as the designer is forced to think through the design in detail. Other bugs are caught by people attending the meeting, who will notice problems that may not be caught by the unit's designer. By catching bugs early and not allowing them to propagate into the implementation, we reduce the time required to get a working system. We can also use the design review to improve the quality of the implementation and make future changes easier to implement. A design review is held to review a particular component of the system. A design review team has the following members:

- The *designers* of the component being reviewed are, of course, central to the design process. They present their design to the rest of the team for review and analysis.
- The *review leader* coordinates the pre-meeting activities, the design review itself, and the post-meeting follow-up.

- The *review scribe* records the minutes of the meeting so that designers and others know which problems need to be fixed.
- The *review audience* studies the component. Audience members will naturally include other members of the project for which this component is being designed. Audience members from other projects often add valuable perspective and may notice problems that team members have missed.

The design review process begins before the meeting itself. The design team prepares a set of documents (code listings, flowcharts, specifications, etc.) that will be used to describe the component. These documents are distributed to other members of the review team in advance of the meeting, so that everyone has time to become familiar with the material. The review leader coordinates the meeting time, distribution of handouts, and so forth.

During the meeting, the leader is responsible for ensuring that the meeting runs smoothly, while the scribe takes notes about what happens. The designers are responsible for presenting the component design. A top-down presentation often works well, beginning with the requirements and interface description, followed by the overall structure of the component, the details, and then the testing strategy. The audience should look for all types of problems at every level of detail, including the problems listed below.

- Is the design team's view of the component's specification consistent with the overall system specification, or has the team misinterpreted something?
- Is the interface specification correct?
- Does the component's internal architecture work well?
- Are there coding errors in the component?
- Is the testing strategy adequate?

The notes taken by the scribe are used in meeting follow-up. The design team should correct bugs and address concerns raised at the meeting. While doing so, the team should keep notes describing what they did. The design review leader coordinates with the design team, both to make sure that the changes are made and to distribute the change results to the audience. If the changes are straightforward, a written report of them is probably adequate. If the errors found during the review caused a major reworking of the component, a new design review meeting for the new implementation, using as many of the original team members as possible, may be useful.

SYSTEM ANALYSIS AND ARCHITECTURE DESIGN

In this section we consider how to turn a specification into an architecture design. We already have a number of techniques for making specific decisions; in this section we look at how to get a handle on the overall system architecture.

The ***CRC card*** methodology is a well-known and useful way to help analyze a system's structure. It is particularly well suited to object-oriented design since it encourages the encapsulation of data and functions. The acronym *CRC* stands for the following three major items that the methodology tries to identify:

- *Classes* define the logical groupings of data and functionality.
- *Responsibilities* describe what the classes do.
- *Collaborators* are the other classes with which a given class works.

The name *CRC card* comes from the fact that the methodology is practiced by having people write on index cards. (In the United States, the standard size for index cards is 3" X 5", so these cards are often called 3 X 5 cards.) An example card is shown in Figure 1.5.1; it has space to write down the class name, its responsibilities and collaborators, and other information. The essence of the CRC card methodology is to have people write on these cards, talk about them, and update the cards until they are satisfied with the results. This technique may seem like a primitive way to design computer systems.

However, it has several important advantages. First, it is easy to get noncomputer people to create CRC cards. Getting the advice of domain experts (automobile designers for automotive electronics or human factors experts for PDA design, for example) is very important in system design.

The CRC card methodology is informal enough that it will not intimidate non-computer specialists and will allow you to capture their input. Second, it aids even computer specialists by encouraging them to work in a group and analyze scenarios. The walkthrough process used with CRC cards is very useful in scoping out a design and determining what parts of a system are poorly understood. This informal technique is valuable to tool-based design and coding. If you still feel a need to use tools to help you practice the CRC methodology, software engineering tools are available that automate the creation of CRC cards.

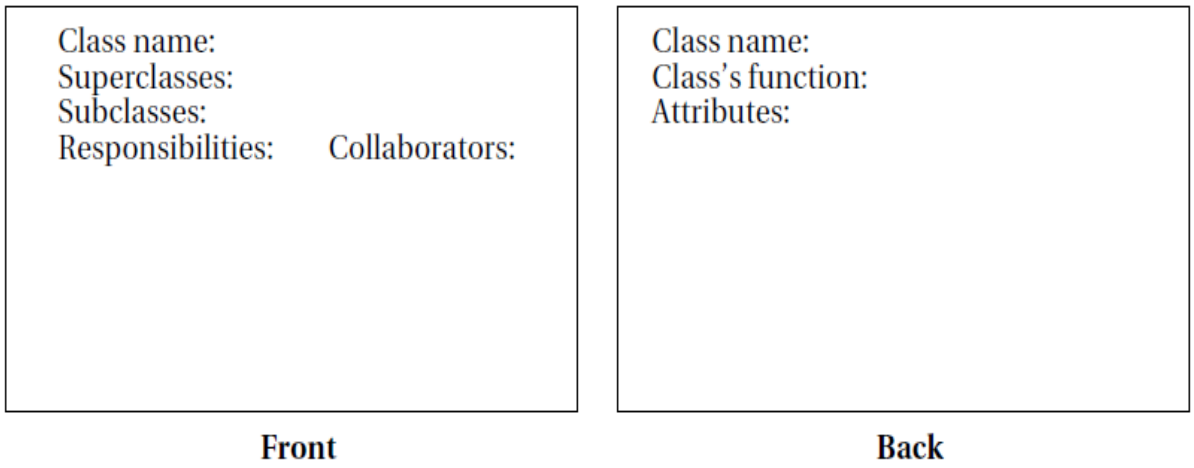


Figure 1.5.1 Layout of a CRC card

[Source: Computers as Components - Principles of Embedded Computing System Design by Marilyn Wolf.]

Before going through the methodology, let's review the CRC concepts in a little more detail. We are familiar with classes they encapsulate functionality. A class may represent a real-world object or it may describe an object that has been created solely to help architect the system. A class has both an internal state and a functional interface; the functional interface describes the class's capabilities. The responsibility set is an informal way of describing that functional interface. The responsibilities provide the class's interface, not its internal implementation. Unlike describing

a class in a programming language, however, the responsibilities may be described informally in English (or your favorite language). The collaborators of a class are simply the classes that it talks to, that is, classes that use its capabilities or that it calls upon to help it do its work. The class terminology is a little misleading when an object oriented programmer looks at CRC cards. In the methodology, a class is actually used more like an object in an OO programming language the CRC card class is used to represent a real actor in the system. However, the CRC card class is easily transformable into a class definition in an object-oriented design.

CRC card analysis is performed by a team of people. It is possible to use it by yourself, but a lot of the benefit of the method comes from talking about the developing classes with others. Before becoming the process, you should create a large number of CRC cards using the basic format shown in Figure 1.5.1. As you are working in your group, you will be writing on these cards; you will probably discard many of them and rewrite them as the system evolves.

The CRC card methodology is informal, but you should go through the following steps when using it to analyze a system:

1. *Develop an initial list of classes:* Write down the class name and perhaps a few words on what it does. A class may represent a real-world object or an architectural object. Identifying which category the class falls into (perhaps by putting a star next to the name of a real-world object) is helpful. Each person can be responsible for handling a part of the system, but team members should talk during this process to be sure that no classes are missed and that

duplicate classes are not created.

2. *Write an initial list of responsibilities and collaborators:* The responsibilities list helps describe in a little more detail what the class does. The collaborators list should be built from obvious relationships between classes. Both the responsibilities and collaborators will be refined in the later stages.

3. *Create some usage scenarios:* These scenarios describe what the system does. Scenarios probably begin with some type of outside stimulus, which is one important reason for identifying the relevant real-world objects.

4. *Walk through the scenarios:* This is the heart of the methodology. During the walk-through, each person on the team represents one or more classes. The scenario should be simulated by acting: people can call out what their class is doing, ask other classes to perform operations, and so on. Moving around, for example, to show the transfer of data, may help you visualize the system's operation. During the walk-through, all of the information created so far is targeted for updating and refinement, including the classes, their responsibilities and collaborators, and the usage scenarios. Classes may be created, destroyed, or modified during this process. You will also probably find many holes in the scenario itself.

5. *Refine the classes, responsibilities, and collaborators:* Some of this will be done during the course of the walkthrough, but making a second pass after the scenarios is a good idea. The longer perspective will help you make more global changes to the CRC cards.

6. *Add class relationships:* Once the CRC cards have been refined, subclass and superclass relationships should become clearer and can be added to the cards.

Once you have the CRC cards, you need to somehow use them to help drive the implementation. In some cases, it may work best to use the CRC cards as direct source material for the implementors; this is particularly true if you can get the designers involved in the CRC card process. In other cases, you may want to write a more formal description, in UML or another language, of the information that was captured during the CRC card analysis, and then use that formal description as the design document for the system implementors.