LECTURE NOTES

ON

EMBEDDED SYSTEMS

VII-Semester: ECE (AEC016)

(IARE-R16)

Mr. B.Naresh, Assistant Professor

Mr.Mohd.Khadir, Assistant Professor

Mrs.Anusha.N, Assistant Professor

Mrs. M.Lavanya, Assistant Professor

ELECTRONICS AND COMMUNICATION ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(AUTONOMOUS)

DUNDIGAL, HYDERABAD - 500 043

SYALLABUS	
Unit-I	**EMBEDDED COMPUTING**
Definition of embedded system, embedded systems vs. general computing systems, history of embedded systems, complex systems and microprocessor, classification, major application areas, the embedded system design process, characteristics and quality attributes of embedded systems, formalisms for system design, design examples	
Unit-II	**INTRODUCTION TO EMBEDDED C AND APPLICATIONS**
C looping structures, register allocation, function calls, pointer aliasing, structure arrangement, bit fields, unaligned data and endianness, inline functions and inline assembly, portability issues; Embedded systems programming in C, binding and running embedded C program in Keil IDE, dissecting the program, building the hardware; Basic techniques for reading and writing from I/O port pins, switch bounce; Applications: Switch bounce, LED interfacing, interfacing with keyboards, displays, D/A and A/D conversions, multiple interrupts, serial data communication using embedded C interfacing	
Unit-III	**RTOS FUNDAMENTALS AND PROGRAMMING**
Operating system basics, types of operating systems, tasks and task states, process and threads, multiprocessing and multitasking, how to choose an RTOS ,task scheduling, semaphores and queues, hard real-time scheduling considerations, saving memory and power. Task communication: Shared memory, message passing, remote procedure call and sockets; Task synchronization: Task communication synchronization issues, task synchronization techniques, device drivers.	
Unit-IV	**EMBEDDED SOFTWARE DEVELOPMENT TOOLS**
Host and target machines, linker/locators for embedded software, getting embedded software into the target system; Debugging techniques: Testing on host machine, using laboratory tools, an example system.	
Unit-V	**INTRODUCTION TO ADVANCED PROCESSORS**
Introduction to advanced architectures: ARM and SHARC, processor and memory organization and instruction level parallelism; Networked embedded systems: Bus protocols, I2C bus and CAN bus; Internet-EnAnalyzed systems, design example-Elevator controller.	
Text Books:	
1. Shibu K.V, ―Introduction to Embedded Systems‖, Tata McGraw Hill Education Private Limited, 2 nd Edition, 2009. 2. Raj Kamal, ―Embedded Systems: Architecture, Programming and Design‖, Tata McGraw-Hill Education, 2 nd Edition, 2011. 3. Andrew Sloss, Dominic Symes,Wright, ―ARM System Developer's Guide Designing and Optimizing System Software‖, 1st Edition, 2004.	
Reference Books:	
1. Wayne Wolf, ― Computers as Components, Principles of Embedded Computing Systems Design‖, Elsevier, 2 nd Edition, 2009. 2. Dr. K. V. K. K. Prasad, ― Embedded / Real-Time Systems: Concepts, Design & Programming‖, dreamtech publishers, 1 st Edition, 2003. 3. Frank Vahid, Tony Givargis, ―Embedded System Design‖, John Wiley & Sons, 3 rd Edition, 2006. 4. Lyla B Das, ―Embedded Systems‖ , Pearson Education, 1 st Edition, 2012. 5. David E. Simon, ―An Embedded Software Primer‖, Addison-Wesley, 1 st Edition, 1999. 6. Michael J. Pont, ―Embedded C‖, Pearson Education, 2nd Edition, 2008.	

UNIT-I
EMBEDDED COMPUTING

INTRODUCTION

This chapter introduces the reader to the world of embedded systems. Everything that we look around us today is electronic. The days are gone where almost everything was manual. Now even the food that we eat is cooked with the assistance of a microchip (oven) and the ease at which we wash our clothes is due to the washing machine. This world of electronic items is made up of embedded system. In this chapter we will understand the basics of embedded system right from its definition.

DEFINITION OF AN EMBEDDED SYSTEM

- An embedded system is a combination of 3things:
    a. Hardware
    b. Software
    c. Mechanical Components

  And it is supposed to do one specific task only.

- **Example 1: Washing Machine**

  A washing machine from an embedded systems point of view has:

  a. Hardware: Buttons, Display & buzzer, electroniccircuitry.
  b. Software: It has a chip on the circuit that holds the software which drives controls & monitors the various operations possible.
  c. Mechanical Components: the internals of a washing machine which actually wash the clothes control the input and output of water, the chassis itself.

- **Example 2: Air Conditioner**

  An Air Conditioner from an embedded systems point of view has:

  a. Hardware: Remote, Display & buzzer, Infrared Sensors, electronic circuitry.
  b. Software: It has a chip on the circuit that holds the software which drives controls & monitors the various operations possible. The software monitors the external temperature through the sensors and then releases the coolant or suppresses it.
  c. Mechanical Components: the internals of an air conditioner the motor, the chassis, the outlet, etc

- An embedded system is designed to do a specific job only. Example: a washing machine can only wash clothes, an air conditioner can control the temperature in the room in which it is placed.

- The hardware & mechanical components will consist all the physically visible things that are used for input, output, etc.
- An embedded system will always have a chip (either microprocessor or microcontroller) that has the code or software which drives the system.

HISTORY OF EMBEDDED SYSTEM

- The first recognised embedded system is the Apollo Guidance Computer(AGC) developed by MIT lab.
- AGC was designed on 4K words of ROM & 256 words of RAM.
- The clock frequency of first microchip used in AGC was 1.024 MHz.
- The computing unit of AGC consists of 11 instructions and 16 bit word logic.
- It used 5000 ICs.
- The UI of AGC is known DSKY(display/keyboard) which resembles a calculator type keypad with array of numerals.
- The first mass-produced embedded system was guidance computer for the Minuteman-I missile in 1961.
- In the year 1971 Intel introduced the world's first microprocessor chip called the 4004, was designed for use in business calculators. It was produced by the Japanese company Busicom.

EMBEDDEDSYSTEM & GENERAL PURPOSE COMPUTER

The Embedded System and the General purpose computer are at two extremes. The embedded system is designed to perform a specific task whereas as per definition the general purpose computer is meant for general use. It can be used for playing games, watching movies, creating software, work on documents or spreadsheets etc.

Following are certain specific points of difference between embedded systems and general purpose computers:

Criteria	**General Computer Purpose**	**Embedded system**
Contents	It is combination of generic hardware and a general purpose OS for executing a variety of	It is combination of special purpose hardware and embedded OS for executing specific set of applications
Operating System	It contains general purpose operating system	It may or may not contain operating system.
Alterations	Applications are alterable by the user.	Applications are non-alterable by the user.
Key factor	Performance is key factor.	Application specific requirements are key factors.
Power Consumption	More	Less
Response Time	Not Critical	Critical for some applications

CLASSIFICATION OF EMBEDDEDSYSTEM

The classification of embedded system is based on following criteria's:

- On generation
- On complexity & performance
- On deterministic behaviour
- On triggering

On generation

1. **First generation(1G):**
   - Built around 8bit microprocessor & microcontroller.
   - Simple in hardware circuit & firmwaredeveloped.
   - Examples: Digital telephone keypads.

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

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

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

On complexity & performance

1. **Small-scale:**
   - Simple in application need
   - Performance not time-critical.
   - Built around low performance & low cost 8 or 16 bit µp/µc.
   - Example: an electronic toy

2. **Medium-scale:**
   - Slightly complex in hardware & firmwarerequirement.
   - Built around medium performance & low cost 16 or 32 bit µp/µc.
   - Usually contain operating system.
   - Examples: Industrial machines.

3. **Large-scale:**
- Highly complex hardware & firmware.
- Built around 32 or 64 bit RISC µp/µc or PLDs or Multicore Processors.
- Response is time-critical.
- Examples: Mission critical applications.

On deterministic behavior

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

On triggering

- Embedded systems which are –Reactive‖ in nature can be based on triggering.
- Reactive systems can be:
  - ✓ Event triggered
  - ✓ Time triggered

APPLICATION OF EMBEDDED SYSTEM

The application areas and the products in the embedded domain are countless.

1. Consumer Electronics: Camcorders, Cameras.
2. Household appliances: Washing machine, Refrigerator.
3. Automotive industry: Anti-lock breaking system(ABS), engine control.
4. Home automation & security systems: Air conditioners, sprinklers, fire alarms.
5. Telecom: Cellular phones, telephone switches.
6. Computer peripherals: Printers, scanners.
7. Computer networking systems: Network routers and switches.
8. Healthcare: EEG, ECG machines.
9. Banking & Retail: Automatic teller machines, point ofsales.
10. Card Readers: Barcode, smart card readers.

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 [Sto95].

A microprocessor is a single-chip CPU. Very large scale integration (VLSI) the acronym is the name technology has allowed us to put a complete CPU on a single chip since 1970s, but those CPUs were very simple. The first microprocessor, the Intel 4004, was designed for an embedded application, namely, a calculator. The calculator was not a general-purpose computer—it merely provided basic arithmetic functions.

However, Ted Hoff of Intel realized that a general-purpose computer programmed properly could implement the required function, and that the computer-on-a-chip could then be reprogrammed for use in other products as well. Since integrated circuit design was (and still is) an expensive and time consuming process, the ability to reuse the hardware design by changing the software was a key breakthrough. The HP-35 was the first handheld calculator to perform transcendental functions [Whi72]. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor.

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. Application Example describes some of the microprocessors used in the BMW 850i.

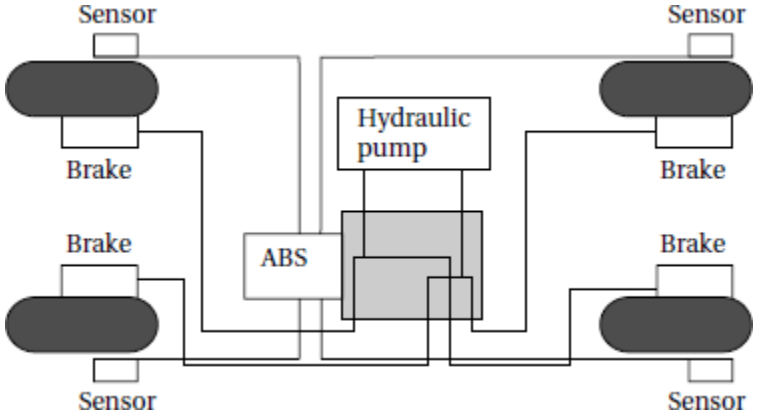
Application Example

BMW 850i brake and stability control system

The BMW 850i was introduced with a sophisticated system for controlling the wheels of the car. An antilock brake system (ABS) reduces skidding by pumping the brakes. An automatic stability control (ASC_T) system intervenes with the engine during maneuvering to improve the car's stability. These systems actively control critical systems of the car; as control systems, they require inputs from and output to the automobile.

Let's first look at the ABS. The purpose of an ABS is to temporarily release the brake on a wheel when it rotates too slowly—when a wheel stops turning, the car starts skidding and becomes hard to control. It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves as seen in the 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.

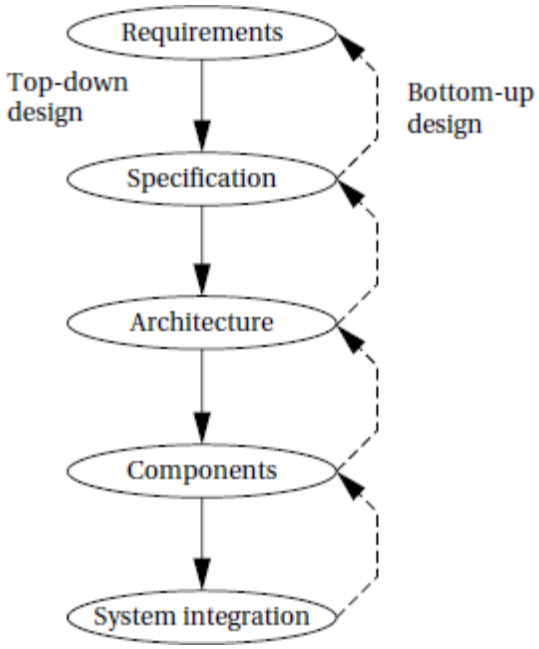
The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent the wheels from skidding. The ASC _ T system's job is to control the engine power and the brake to improve the car's stability during maneuvers. The ASC _ T controls four different systems: throttle, 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.



THE 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 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); and
- power consumption.

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

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

Requirements

Clearly, before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components. We generally proceed in two phases: First, we gather an informal description from the customers known as requirements, and we refine the requirements into a specification that contains enough information to begin designing the system architecture.

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 goodway 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.

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

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

■ *Name:* This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people but can also crystallize the purpose of the machine.

■ *Purpose:* This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.

■ *Inputs and outputs:* These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail: — *Types of data:* Analog electronic signals? Digital data? Mechanical inputs? — *Data characteristics:* Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element? — *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?

■ *Functions:* This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?

■ *Performance:* Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time frame. It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.

■ *Manufacturing cost:* This includes primarily the cost of the hardware 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.

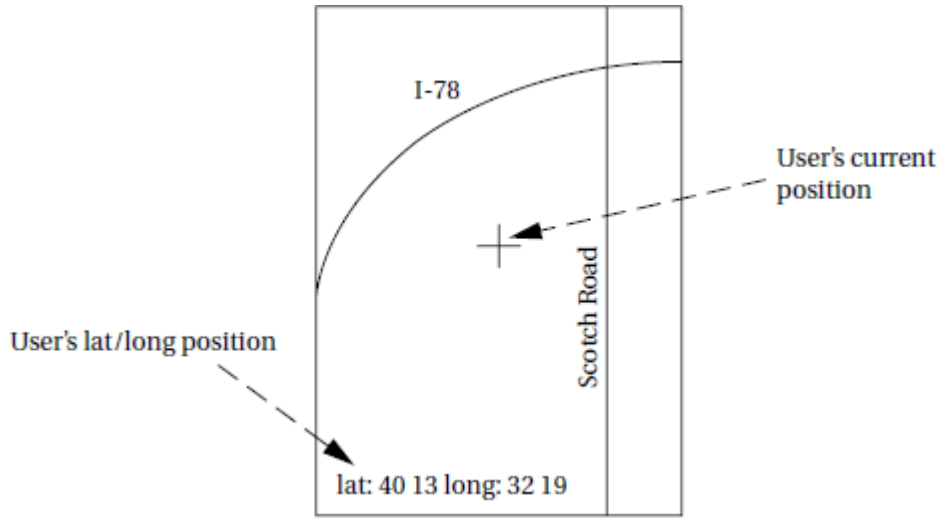
A more thorough requirements analysis for a large system might use a form similar to Figure as a summary of the longer requirements document. After an introductory section containing this form, a longer requirements document could include details on each of the items mentioned in the introduction. For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

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

Example

Requirements analysis of a GPS moving map

The moving map is a handheld device that displays for the user a map of the terrain around the user's current position; the map display changes as the user and the map device change position. The moving map obtains its position from the GPS, a satellite-based navigation system. The moving map display might look something like the following figure.



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

- *Functionality:* This system is designed for highway driving and similar uses, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.
- *User interface:* The screen should have at least 400_600 pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed to allow the user to make selections to control the system.

■ *Performance:* The map should scroll smoothly. Upon power-up, a display should take no more than one second to appear, and the system should be able to verify its position and display the current map within 15 s.

■ *Cost:* The selling cost (street price) of the unit should be no more than $100.

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

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

Note that many of these requirements are not specified in engineering units—for example, physical size is measured relative to a hand, not in centimeters. Although these requirements must ultimately be translated into something that can be used by the designers, keeping a record of what the customer wants can help to resolve questions about the specification that may crop up later during design. Based on this discussion, let's write a requirements chart for our moving map system:

Name	GPS moving map
Purpose	Consumer-grade moving map for driving use
Inputs	Power button, two control buttons
Outputs	Back-lit LCD display 400 × 600
Functions	Uses 5-receiver GPS system; three user-selectable resolutions; always displays current latitude and longitude
Performance	Updates screen within 0.25 seconds upon movement
Manufacturing cost	$30
Power	100 mW
Physical size and weight	No more than 2" × 6, " 12 ounces

Specification

The specification is more precise—it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it accurately reflects the customer's requirements and does so in a way that can be clearly followed during design. Specification is probably the least familiar phase of this methodology for neophyte designers, but it is essential to creating working systems with a minimum of designer effort.

Designers who lack a clear idea of what they want to build when they begin typically make faulty assumptions early in the process that aren't obvious until they have a working system. At that point, the only solution is to take the machine apart, throw away some of it, and start again. The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer. It should also be unambiguous enough that designers know what they need to build.

Designers can run into several different types of problems caused by unclear specifications. If the behavior of some feature in a particular situation is unclear from the specification, the designer may implement the wrong functionality. If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

A specification of the GPS system would include several components:

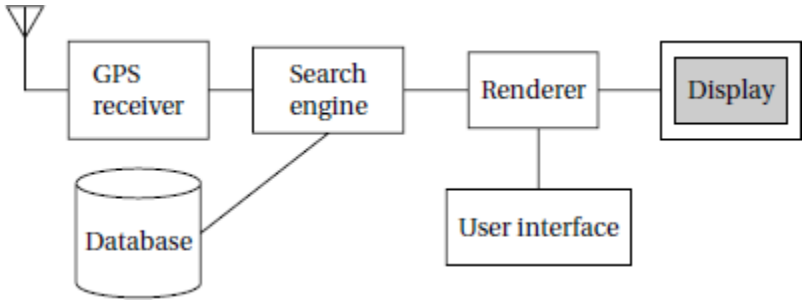
- Data received from the GPS satellite constellation.
- Map data.
- User interface.
- Operations that must be performed to satisfy customer requests.
- Background actions required to keep the system running, such as operating the GPS receiver.

UML, a language for describing specifications, will be introduced later and we will use it to write a specification. We will practice writing specifications in each chapter as we work through example system designs. We will also study specification techniques in more later.

Architecture Design

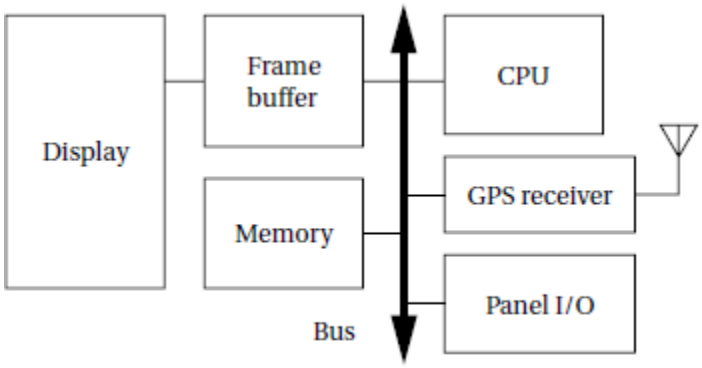
The specification does not say how the system does things, only what the system does. Describing how the system implements those functions is the purpose of the architecture. The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture. The creation of the architecture is the first phase of what many designers think of as design. To understand what an architectural description is, let's look at sample architecture for the moving map of Example Figure shows sample system architecture in the form of a ***block diagram*** that shows major operations and data flows among them.

This block diagram is still quite abstract—we have not yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on. The diagram does, however, go a long way toward describing how to implement the functions described in the specification. We clearly see, for example, that we need to search the topographic database and to render (i.e., draw) the results for the display. We have chosen to separate those functions so that we can potentially do them in parallel—performing rendering separately from searching the database may help us update the screen more fluidly.

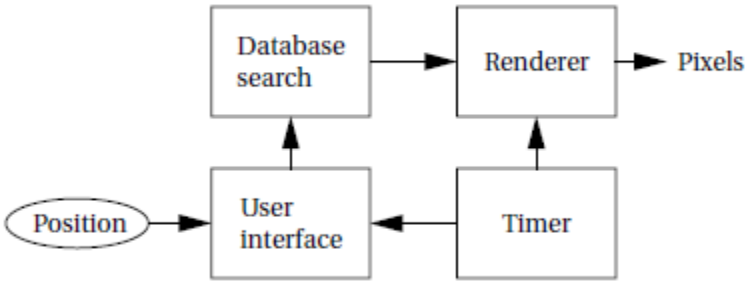


Only after we have designed an initial architecture that is not biased toward too many implementation details should we refine that system block diagram into two block diagrams: one for hardware and another for software. These two more refined block diagrams are shown in Figure 1.4.The hardware block diagram clearly shows that we have one central CPU surrounded by memory and I/O devices. In particular, we have chosen to use two memories: a frame buffer for the pixels to be displayed and a separate program/data memory for general use by the CPU. The software block diagram fairly closely follows the system block diagram, but we have added

a timer to control when we read the buttons on the user interface and render data onto the screen. To have a truly complete architectural description, we require more detail, such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed in time. Architectural descriptions must be designed to satisfy both functional and nonfunctional requirements. Not only must all the required functions be present, but we must meet cost, speed, power, and other nonfunctional constraints.



Hardware



Software

Starting out with a system architecture and refining that to hardware and software architectures is one good way to ensure that we meet all specifications: We can concentrate on the functional elements in the system block diagram, and then consider the nonfunctional constraints when creating the hardware and software architectures. How do we know that our hardware and software architectures in fact meet constraints on speed, cost, and so on? We must somehow be able to estimate the properties of the components of the block diagrams, such as the search and rendering functions in the moving map system.

Accurate estimation derives in part from experience, both general design experience and particular experience with similar systems. However, we can sometimes create simplified models to help us make more accurate estimates. Sound estimates of all nonfunctional constraints during the architecture phase are crucial, since decisions based on bad data will show up during the final phases of design, indicating that we did not, in fact, meet the specification.

Designing Hardware and Software Components

The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification. The components will in general include both hardware—FPGAs, boards, and so on—and software modules. Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other components. In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component.

We can also make use of standard software modules. One good example is the topographic database. Standard topographic databases exist, and you probably want to use standard routines to access the database—not only is the data in a predefined format, but it is highly compressed to save storage. Using standard software for these access functions not only saves us design time, but it may give us a faster implementation for specialized functions such as the data decompression phase. You will have to design some components yourself. Even if you are using only standard integrated circuits, you may have to design the printed circuit board that connects them. You will probably have to do a lot of custom programming as well.

When creating these embedded software modules, you must of course make use of your expertise to ensure that the system runs properly in real time and that it does not take up more memory space than is allowed. The power consumption of the moving map software example is particularly important. You may need to be very careful about how you read and write memory to minimize power—for example, since memory accesses are a major source of power consumption, memory transactions must be carefully planned to avoid reading the same data several times.

System Integration

Only after the components are built do we have the satisfaction of putting them together and seeing a working system. Of course, this phase usually consists of a lot more than just plugging everything together and standing back. Bugs are typically found during system integration, and good planning can help us find the bugs quickly. By building up the system in phases and running properly chosen tests, we can often find bugs more easily. If we debug only a few modules at a time, we are more likely to uncover the simple bugs and able to easily recognize them.

Only by fixing the simple bugs early will we be able to uncover the more complex or obscure bugs that can be identified only by giving the system a hard workout. We need to ensure during the architectural and component design phases that we make it as easy as possible to assemble the system in phases and test functions relatively independently.

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

FORMALISMS FOR SYSTEM DESIGN

As mentioned in the last section, we perform a number of different design tasks at different levels of abstraction throughout this book: creating requirements and specifications, architecting the system, designing code, and designing tests. It is often helpful to conceptualize these tasks in diagrams. Luckily, there is a visual language that can be used to capture all these design tasks: the ***Unified Modeling Language (UML).***

UML was designed to be useful at many levels of abstraction in the design process. UML is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction. UML is an ***object-oriented*** modeling language. We will see precisely what we mean by an object in just a moment, but object-oriented design emphasizes two concepts of importance:

■ It encourages the design to be described as a number of interacting objects, rather than a few large monolithic blocks of code.

■ At least some of those objects will correspond to real pieces of software or hardware in the system. We can also use UML to model the outside world that interacts with our system, in which case the objects may correspond to people or other machines. It is sometimes important to implement something we think of at a high level as a single object using several distinct pieces of code or to otherwise break up the object correspondence in the implementation. However, thinking of the design in terms of actual objects helps us understand the natural structure of the system. Object-oriented (often abbreviated OO) specification can be seen in two complementary ways:

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

■ Object-oriented specification provides a basic set of primitives that can be used to describe systems with particular attributes, irrespective of the relationships of those systems' components to real-world objects. Both views are useful. At a minimum, object-oriented specification is a set of linguistic mechanisms. In many cases, it is useful to describe a system in terms of real-world analogs. However, performance, cost, and so on may dictate that we change the specification to be different in some ways from the real-world elements we are trying to model and implement. In this case, the object-oriented specification mechanisms are still useful. What is the relationship between an object-oriented specification and an object oriented programming

language (such as C++)? A specification language may not be executable. But both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

Unified Modeling Language (UML)—the acronym is the name is a large language, and covering all of it is beyond the scope of this book. In this section, we introduce only a few basic concepts. In later chapters, as we need a few more UML concepts, we introduce them to the basic modeling elements introduced here. Because UML is so rich, there are many graphical elements in a UML diagram. It is important to be careful to use the correct drawing to describe something—for instance, UML distinguishes between arrows with open and filled-in arrowheads, and solid and broken lines. As you become more familiar with the language, uses of the graphical primitives will become more natural to you. We also won't take a strict object-oriented approach. We may not always use objects for certain elements of a design—in some cases, such as when taking particular aspects of the implementation into account, it may make sense to use another design style. However, object-oriented design is widely applicable, and no designer can consider himself or herself design literate without understanding it.

Structural Description

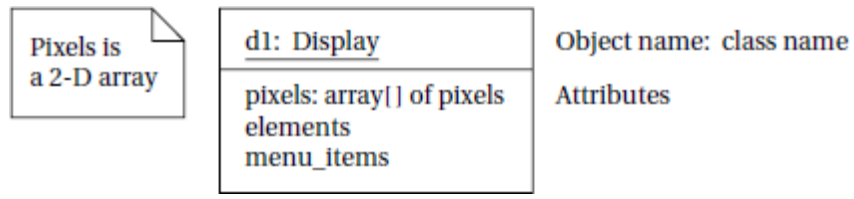
By ***structural description***, we mean the basic components of the system; we will learn how to describe how these components act in the next section. The principal component of an object-oriented design is, naturally enough, the ***object***. An object includes a set of ***attributes*** that define its internal state. When implemented in a programming language, these attributes usually become variables or constants held in a data structure.

In some cases, we will add the type of the attribute after the attribute name for clarity, but we do not always have to specify a type for an attribute. An object describing a display (such as a CRT screen) is shown in UML notation in Figure. The text in the folded-corner page icon is a ***note***; it does not correspond to an object in the system and only serves as a comment. The attribute is, in this case, an array of pixels that holds the contents of the display.

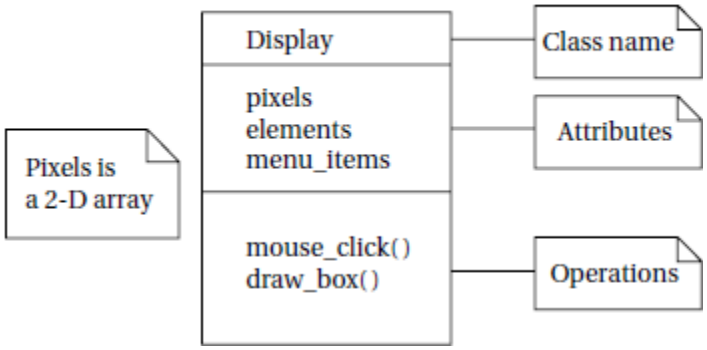
The object is identified in two ways: It has a unique name, and it is a member of a ***class***. The name is underlined to show that this is a description of an object and not of a class. A class is a form of type definition—all objects derived from the same class have the same characteristics, although their attributes may have different values. A class defines the attributes that an object may have. It also defines the ***operations*** that determine how the object interacts with the rest of the world. In a programming language, the operations would become pieces of code used to manipulate the object.

The UML description of the *Display* class is shown in Figure. The class has the name that we saw used in the *d*1 object since *d*1 is an instance of class *Display*. The *Display* class defines the *pixels* attribute seen in the object; remember that when we instantiate the class an object, that

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



An object in UML notation.



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

Clearly, the choice of an interface is a very important decision in object-oriented design. The proper interface must provide ways to access the object's state (since we cannot directly see the attributes) as well as ways to update the state. We need to make the object's interface general enough so that we can make full use of its capabilities. However, excessive generality often makes the object large and slow. Big, complex interfaces also make the class definition difficult for designers to understand and use properly. There are several types of ***relationships*** that can exist between objects and classes:

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

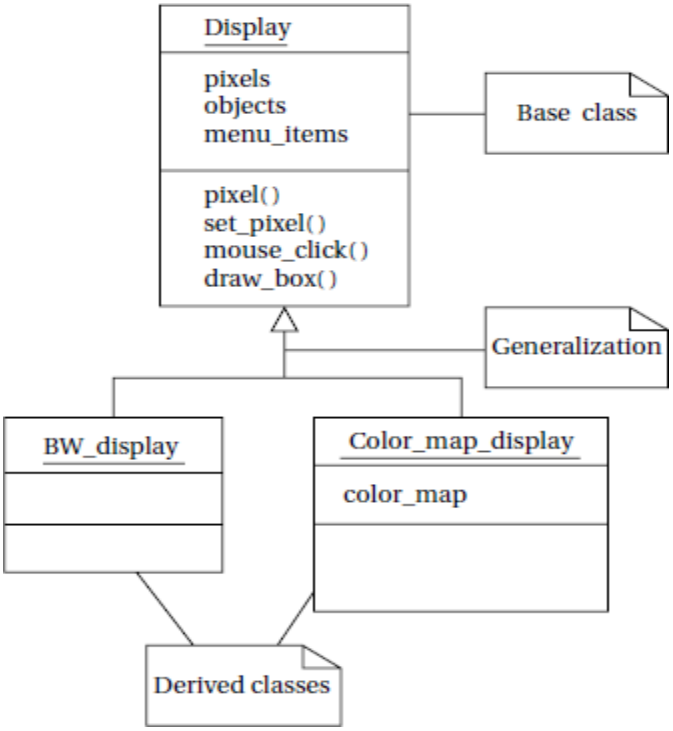
■ ***Aggregation*** describes a complex object made of smaller objects.

■ ***Composition*** is a type of aggregation in which the owner does not allow access to the component objects.

■ ***Generalization*** allows us to define one class in terms of another.

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

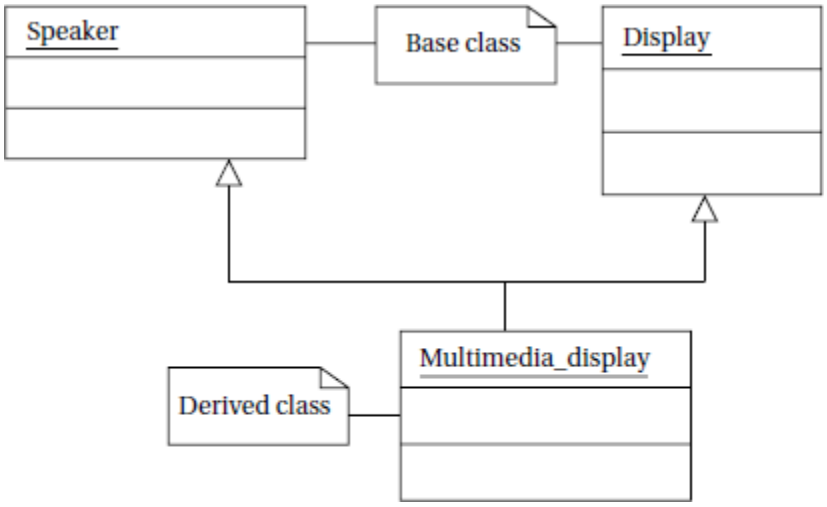
Unified Modeling Language, like most object-oriented languages, allows us to define one class in terms of another. An example is shown in Figure, where we ***derive*** two particular types of displays. The first, *BW_ display*, describes a black and- white display. This does not require us to add new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



select from a large number of available colors even with a small number of bits per pixel. This class defines a *color_map* attribute that determines how pixel values are mapped onto display colors. A ***derived class*** inherits all the attributes and operations from its ***base class***. In this class, *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class.

This relation is transitive—if *Display* were derived from another class, both *BW_display* and *Color_map_display* would inherit all the attributes and operations of *Display's* base class as well. Inheritance has two purposes. It of course allows us to succinctly describe one class that shares some characteristics with another class. Even more important, it captures those relationships between classes and documents them. If we ever need to change any of the classes, knowledge of the class structure helps us determine the reach of changes—for example, should the change affect only *Color_map_display* objects or should it change all Display objects?

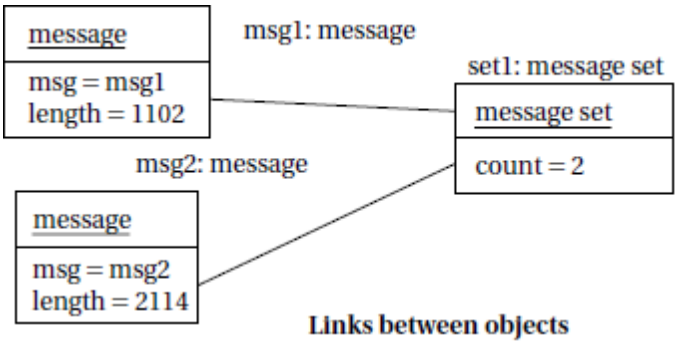
Unified Modeling Language considers inheritance to be one form of generalization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead. Both *BW_display* and *Color_map_display* are specific versions of *Display*, so *Display* generalizes both of them. UML also allows us to define ***multiple inheritance***, in which a class is derived from more than one base class. (Most object-oriented programming languages support multiple inheritance as well.) An example of multiple inheritance is shown in Figure;



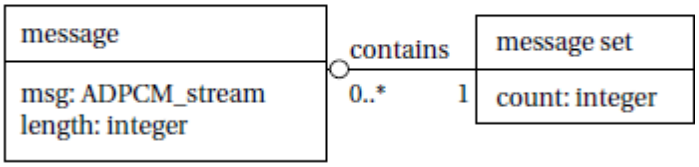
we have omitted the details of the classes' attributes and operations for simplicity. In this case, we have created a *Multimedia_display* class by combining the *Display* class with a *Speaker* class for sound. The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*. Because multiple inheritance causes the sizes of the attribute set and operations to expand so quickly, it should be used with care.

A ***link*** describes a relationship between objects; association is to link as class is to object. We need links because objects often do not stand alone; associations let us capture type information about these links. Figure 1.9 shows examples of links and an association. When we consider the actual objects in the system, there is a set of messages that keeps track of the current number of active messages (two in this example) and points to the active messages. In this case, the link defines the *contains* relation. When generalized into classes, we define an association between the message set class and the message class. The association is drawn as a line between the two labeled with the name of the association, namely, *contains*. The ball and the number at the message class end indicate that the message set may include zero or more message objects. Sometimes we may want to attach data to the links themselves; we can specify this in the association by attaching a class-like box to the association's edge, which holds the association's data.

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

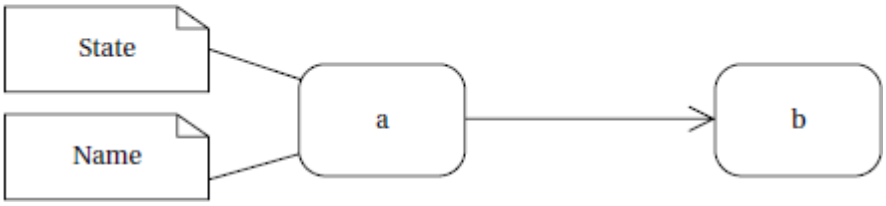


Links between objects



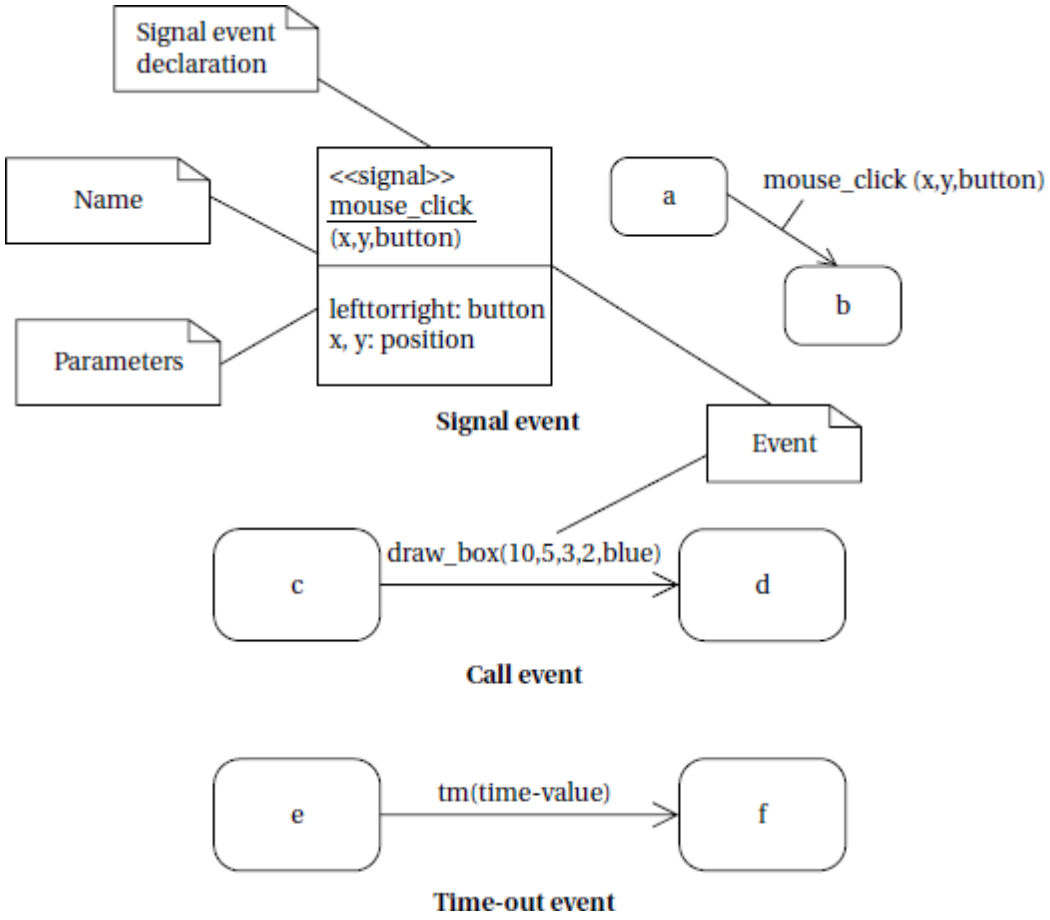
Association between classes

Links and association.



Behavioral Description

We have to specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a ***state machine***. Figure shows UML states; the transition between two states is shown by a skeleton arrow. These state machines will not rely on the operation of a clock, as in hardware; rather, changes from one state to another are triggered by the occurrence of ***events***.

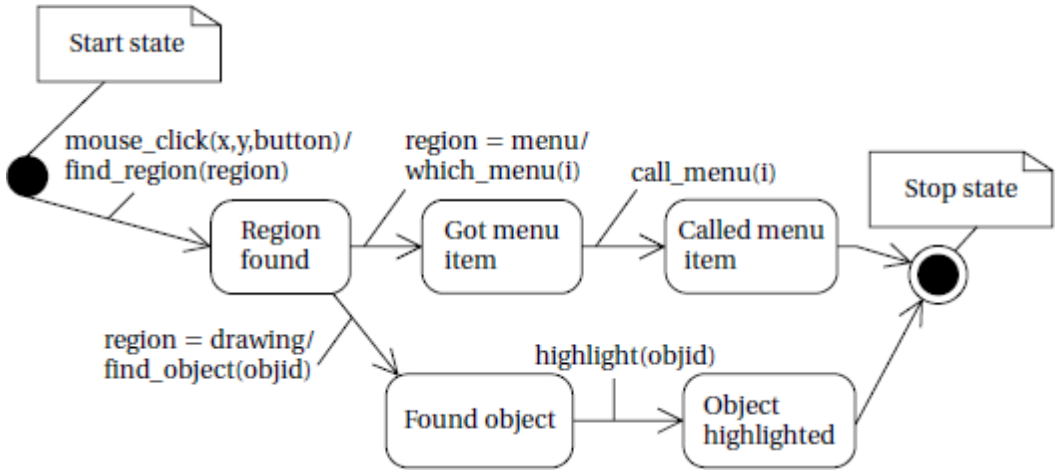


An event is some type of action. The event may originate outside the system, such as a user pressing a button. It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine. We will concentrate on the following three types of events defined by UML, as illustrated in Figure.

■ A ***signal*** is an asynchronous occurrence. It is defined in UML by an object that is labeled as a <<*signal*>>. The object in the diagram serves as a declaration of the event's existence. Because it is an object, a signal may have parameters that are passed to the signal's receiver.

■ A ***call event*** follows the model of a procedure call in a programming language.

■ A ***time-out event*** causes the machine to leave a state after a certain amount of time. The label *tm(time-value)* on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an

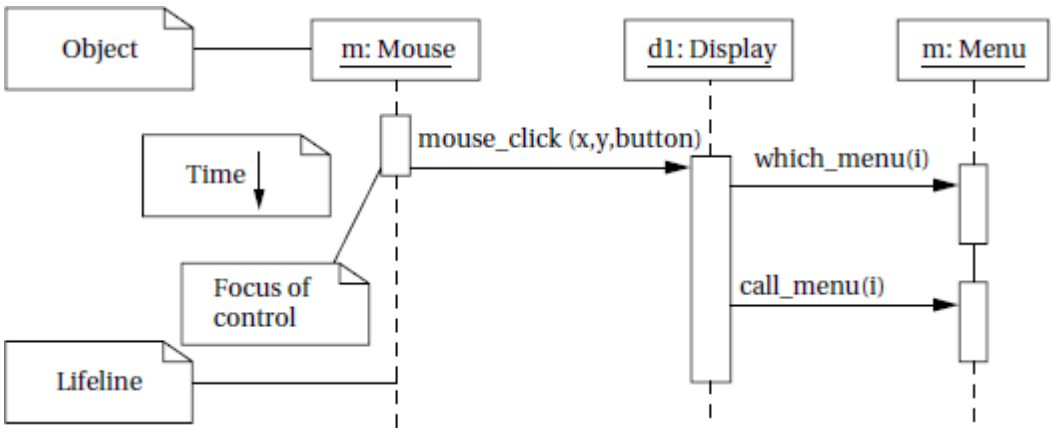


external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism. We show the occurrence of all types of signals in a UML diagram in the same way— as a label on a transition.

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

In some cases, we take conditional transitions out of states based on inputs or the results of some computation done in the state. In other cases, we make an unconditional transition to the next state. Both the unconditional and conditional transitions make use of the call event. Splitting a complex operation into several states helps document the required steps, much as subroutines can be used to structure code. It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved.

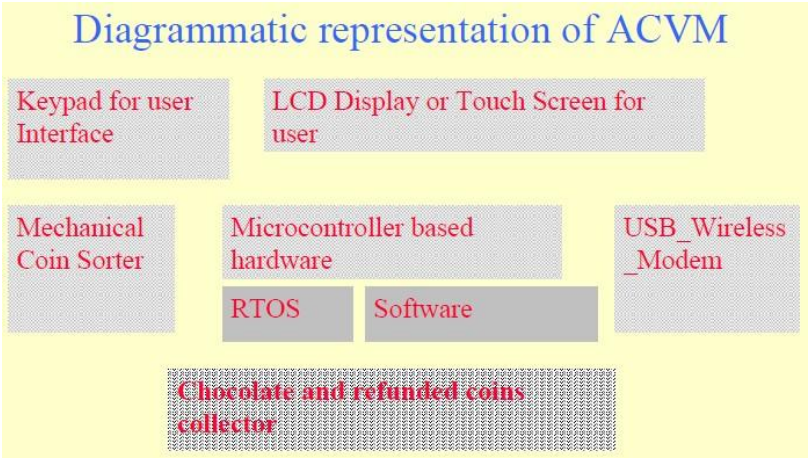
In this case, we can create a sequence diagram, like the one for a mouse click scenario shown in Figure. A ***sequence diagram*** is somewhat similar to a hardware timing diagram, although the time flows vertically in a sequence diagram, whereas time typically flows horizontally in a timing diagram. The sequence diagram is designed to show a particular scenario or choice of events—it is not convenient for showing a number of mutually exclusive possibilities. In this case, the sequence shows what happens when a mouse click is on the menu region. Processing includes three objects shown at the top of the diagram. Extending below each object is its *lifeline*, a dashed line that shows how long the object is alive. In this case, all the objects remain alive for the entire sequence, but in other cases objects may be created or destroyed during processing. The boxes



along the lifelines show the *focus of control* in the sequence, that is, when the object is actively processing. In this case, the mouse object is active only long enough to create the *mouse_click* event. The display object remains in play longer; it in turn uses call events to invoke the menu object twice: once to determine which menu item was selected and again to actually execute the menu call. The find_region( ) call is internal to the display object, so it does not appear as an event in the diagram.

DESIGN PROCESS EXAMPLES

Automatic Chocolate vending machine



Keypad on the top of the machine. LCD display unit on the top of the machine. It displays menus, text entered into the ACVM and pictograms, welcome, thank and other messages. Graphic interactions with the machine. Displays time and date. Delivery slot so that child can collect the chocolate and coins, if refunded. Internet connection port so that owner can know status of the ACVM sales from remote.

ACVM Hardware units

Microcontroller or ASIP (Application Specific Instruction Set Processor). RAM for storing temporary variables and stack. ROM for application codes and RTOS codes for scheduling the tasks. Flash memory for storing user preferences, contact data, user address, user date of birth,

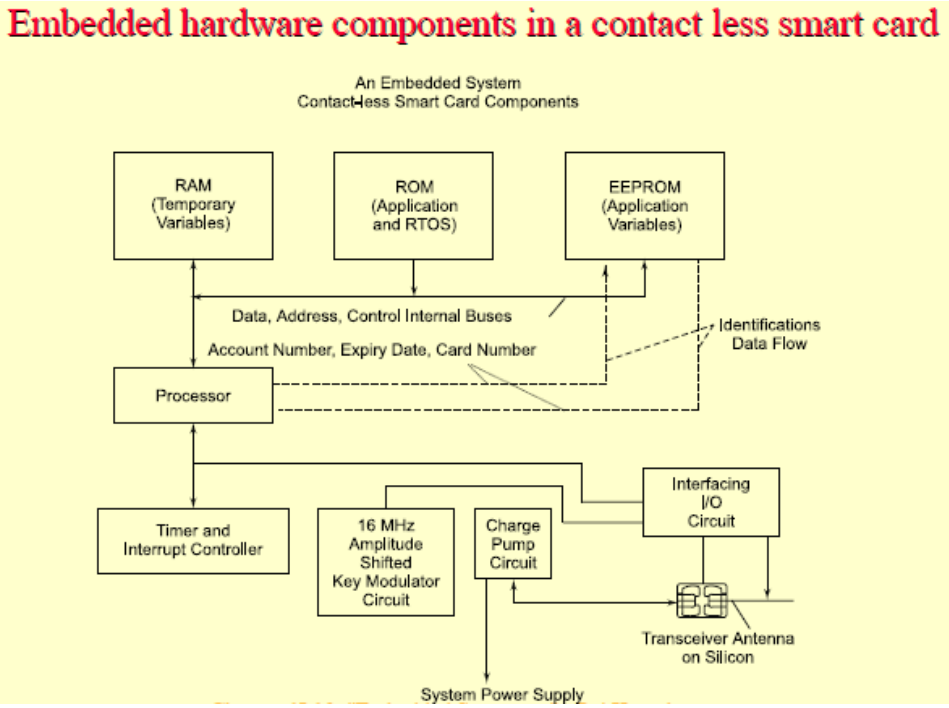
user identification code, answers of FAQs. Timer and Interrupt controller. A TCP/IP port (Internet broadband connection) to the ACVM for remote control and for getting ACVM status reports by owner. ACVM specific hardware. Power supply.

ACVM Software components

_ Keypad input read
_ Display
_ Read coins
_ Deliver chocolate
_ TCP/IP stack processing
_ TCP/IP stack communication

Smart Card

Smart card– a plastic card in ISO standard dimensions, 85.60 mm x 53.98 x 0.80 mm.
_ Embedded system on a card.
_ SoC (System-On-Chip).
_ ISO recommended standards are ISO7816 (1 to 4) for host-machine contact based cards and ISO14443 (Part A or B) for the contact-less cards.
_ Silicon chip is just a few mm in size and is concealed in-between the layers. Its very small size protects the card from bending



Embedded hardware components

_ Microcontroller or ASIP (Application Specific Instruction Set Processor)
_ RAM for temporary variables and stack
_ ROM for application codes and RTOS codes for scheduling the tasks
_ EEPROM for storing user data, user address, user identification codes, card number and expiry date
_ Timer and Interrupt controller
_ A carrier frequency ~16 MHz generating circuit and Amplitude Shifted Key (ASK)
_ Interfacing circuit for the I/Os
_ Charge pump

ROM

Fabrication key, Personalization key An utilization lock.
_ RTOS and application using only the logical addresses

Embedded Software

_ Boot-up, Initialisation and OS programs
_ Smart card secure file system
_ Connection establishment and termination
_ Communication with host
_ Cryptography
_ Host authentication
_ Card authentication
_ Addition parameters or recent new data sent by the host (for example, present balance left).

Smart Card OS Special features

_ Protected environment.
_ Every method, class and run time libraryshould be scalable.
_ Code-size generated be optimum.
_ Memory should not exceed 64 kB memory.
_ Limiting uses of specific data types; multidimensional arrays, long 64-bit integer and floating points

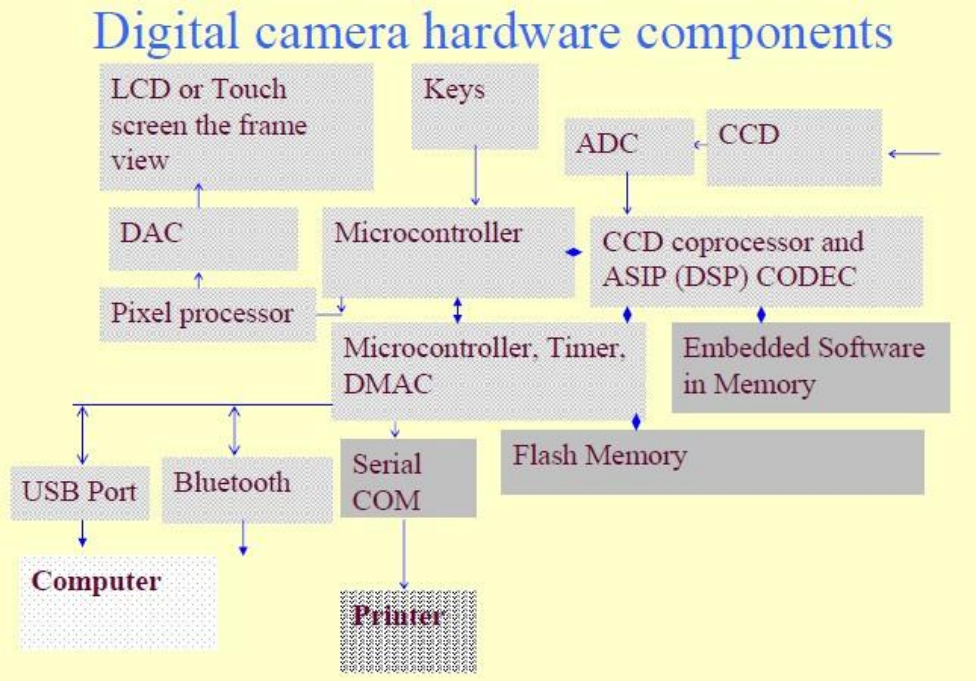
Smart Card OS Limiting features

_ Limiting uses of the error handlers, exceptions, signals, serialization, debugging and profiling. [Serialization means process of converting an object is converted into a data stream for transferring it to network or from one process to another. At receiver end there is de-serialization Smart Card OS File System and Classes
_ Three-layered file system for the data.
_ Master file to store all file headers.
_ Dedicated file to hold a file grouping and headers of the immediate successor elementary files of the group.
_ Elementary file to hold the file header and its file data.

_ Fixed-length or variable-file length management

_ Classes for the network, sockets, connections, data grams, character-input output and streams, security management, digital-certification, symmetric and asymmetric keys-based cryptography and digital signatures..

Digital Camera



A typical Camera

_ 4 M pixel/6 M pixel still images, clear visual display (ClearVid) CMOS sensor, 7 cm wide LCD photo display screen, enhanced imaging processor, double anti blur solution and high-speed processing engine, 10X optical and 20X digital zooms

_ Record high definition video-clips. It therefore has speaker microphone(s) for high quality recorded sound.

_ Audio/video Out Port for connecting to a TV/DVD player.

Arrangements

_ Keys on the camera.

_ Shutter, lens and charge coupled device (CCD) array sensors

_ Good resolution photo quality LCD display unit

_ Displays text such as image-title, shooting data and time and serial number. It displays messages. It displays the GUI menu when user interacts with the camera.

_ Self-timer lamp for flash.

Internal units

_ Internal memory flash to store OS and embedded software and limited number of image files

_ Flash memory stick of 2 GB or more for large storage.

_ Universal Serial Bus (USB), Bluetooth and serial COM port for connecting it to computer, mobile and printer. LCD screen to display frame view.

_ Saved images display using the navigation keys.

_ Frame light falls on the CCD array, which through an ADC transmits the bits for each pixel in each row in the frame and for the dark area pixels in each row for offset correction in CCD signaled light intensities for each row.

_ The CCD bits of each pixel in each row and column are offset corrected by CCD signal processor (CCDSP).

ASIP and Single purpose processors

_ For Signals compression using a JPEG CODEC and saved in one jpg file for each frame.

_ For DSP for compression using the discrete cosine transformations (DCTs) and decompression.

_ For DCT Huffman coding for the JPEG compression.

_ For decompression by inverse DCT before the DAC sends input for display unit through pixel processor.

_ Pixel processor (for example, image contrast, brightness, rotation, translation, color adjustment)

Digital Camera Hardware units

_ Microcontroller or ASIP (Application Specific Instruction Set Processor)

_ Multiple processors (CCDSP, DSP, Pixel Processor and others)

_ RAM for storing temporary variables and stack

_ ROM for application codes and RTOS codes for scheduling the tasks Timer, Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, ADC, DAC and Interrupt controller

_ The DAC gets the input from pixel processor, which gets the inputs from JPEG file for the saved images and also gets input directly from the CCDSP through pixel processor or the frame in present view

_ USB controller Direct Memory Access controller

_ LCD controller

_ Battery and external charging circuit

Digital Camera Software components

_ CCD signal processing for off-set correction

_ JPEG coding

_ JPEG decoding

_ Pixel processing before display

_ Memory and file systems
_ Light, flash and display device drivers
_ LCD, USB and Bluetooth Port device- drivers for port operations for display, printer and Computer communication control

Light, flash and display device drivers
CCD signal processing
JPEG coding
JPEG decoding
Pixel co-processing
LCD and USB Port device drivers
LCD, Bluetooth COM and USB Port device drivers

Characteristics of Embedded systems:

Embedded systems possess certain specific characteristics and these are unique to each Embedded system.

1. Application and domain specific
2. Reactive and Real Time
3. Operates in harsh environments
4. Distributed
5. Small Size and weight
6. Power concerns
7. Single-functioned
8. Complex functionality
9. Tightly-constrained
10. Safety-critical

1. Application and Domain Specific:-

- Each E.S has certain functions to perform and they are developed in such a manner to do the intended functions only.
- They cannot be used for any other purpose.

- Ex – The embedded control units of the microwave oven cannot be replaced with AC"S embedded control unit because the embedded control units of microwave oven and AC are specifically designed to perform certain specific tasks.

2. **Reactive and Real Time:-**
   - E.S are in constant interaction with the real world through sensors and user-defined input devices which are connected to the input port of the system.
   - Any changes in the real world are captured by the sensors or input devices in real time and the control algorithm running inside the unit reacts in a designed manner to bring the controlled output variables to the desired level.
   - E.S produce changes in output in response to the changes in the input, so they are referred as reactive systems.
   - Real Time system operation means the timing behavior of the system should be deterministic ie the system should respond to requests in a known amount of time.

   - Example – E.S which are mission critical like flight control systems, Antilock Brake Systems (**ABS**) etc are Real Time systems.

3. **Operates in Harsh Environment :–**
   - The design of E.S should take care of the operating conditions of the area where the system is going to implement.
   - Ex – If the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade.
   - Also proper shock absorption techniques should be provided to systems which are going to be commissioned in places subject to high shock.

4. **Distributed: –**

   - It means that embedded systems may be a part of a larger system.
   - Many numbers of such distributed embedded systems form a single large embedded control unit.
   - Ex – Automatic vending machine. It contains a card reader, a vending unit etc. Each of them are independent embedded units but they work together to perform the overall vending function.

5. **Small Size and Weight:-**

   - Product aesthetics (size, weight, shape, style, etc) is an important factor in choosing a product.
   - It is convenient to handle a compact device than a bulky product.

6. **Power Concerns:-**
   - Power management is another important factor that needs to be considered in designing embedded systems.
   - E.S should be designed in such a way as to minimize the heat dissipation by the system.

7. **Single-functioned:-** Dedicated to perform a single function

8. **Complex functionality: -**
   - We have to run sophisticated algorithms or multiple algorithms in some applications.

9. **Tightly-constrained:-**
   - Low cost, low power, small, fast, etc

10. **Safety-critical:-**
    - Must not endanger human life and the environment

Quality Attributes of Embedded System:

Quality attributes are the non-functional requirements that need to be documented properly in any system design**.** Quality attributes can be classified as

I. Operational quality attributes

II. Non-operational quality attributes.

I. Operational Quality Attributes: The operational quality attributes represent the relevant quality attributes related to the embedded system when it is in the operational mode or online mode.

Operational Quality Attributes are:

1. Response :-

It is the measure of quickness of the system.

It tells ow fast the system is tracking the changes in input variables. Most of the E.S demands fast response which should be almost real time.

Ex – Fl.ht control application.

2. Throughput :-

It deals ith the efficiency of a system.

It can be defined as the rate of production or operation of a defined process over a stated period of time.

The rat can be expressed in terms of products, batches produced or any other meaningful measurements.

Ex – In case of card reader throughput means how many transactions the reader can perform in a minute or in an hour or in a day.

Throughput is generally measured in terms of –Benchmark‖.

A Benchmark is a reference point by which something can be measured

3. Reliability :-

It is a measure of how much we can rely upon the proper functioning of the system.

- Mean Time Between Failure (MTBF) and Mean Time To Repair (MTTR) are the terms used in determining system reliability.
- MTBF gives the frequency of failures in hours/weeks/months.
- MTTR specifies how long the system is allowed to be out of order following a failure.
- For embedded system with critical application need, it should be of the order of minutes.

4. Maintainability:-

- It deals with support and maintenance to the end user or client in case of technical issues and product failure or on the basis of a routine system checkup.
- Reliability and maintainability are complementary to each other.
- A more reliable system means a system with less corrective maintainability requirements and vice versa.
- Maintainability can be broadly classified into two categories

1. Scheduled or Periodic maintenance (Preventive maintenance)
2. Corrective maintenance to unexpected failures

5. Security:-

- Confidentiality, Integrity and availability are the three major measures of information security.
- Confidentiality deals with protection of data and application from unauthorized disclosure.

- Integrity deals with the protection of data and application from unauthorized modification.

- Availability deals with protection of data and application from unauthorized users.

6. Safety :-

Safety deals with the possible damages that can happen to the operator, public and the environment due to the breakdown of an Embedded System.

The breakdown of an embedded system may occur due to a hardware failure or a firmware failure.

Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of damage to an acceptable level.

II. Non-Operational Quality Attributes: The quality attributes that needs to be addressed for the product not on the basis of operational aspects are grouped under this category.

1. Testability and Debug-ability:-

- Testability deals with how easily one can test the design, application and by which means it can be done.

- For an E.S testability is applicable to both the embedded hardware and firmware.

- Embedded hardware testing ensures that the peripherals and total hardware functions in the desired manner, whereas firmware testing ensures that the firmware is functioning in the expected way**.**

- Debug-ability is a means of debugging the product from unexpected behavior in the system

- Debug-ability is two level process

- 1.Hardware level 2.software level

- **1. Hardware level:** It is used for finding the issues created by hardware problems.

- **2. Software level:** It is employed for finding the errors created by the flaws in the software.

2. Evolvability :-

- It is a term which is closely related to Biology.
- It is referred as the non-heritable variation.
- For an embedded system evolvability refers to the ease with which the embedded product can be modified to take advantage of new firmware or hardware technologies.

3. Portability:-

- It is the measure of system independence.
- An embedded product is said to be portable if the product is capable of functioning in various environments, target processors and embedded operating systems.
- „Porting" represents the migration of embedded firmware written for one target processor to a different target processor.

4. Time-to-Prototype and Market:-

- It is the time elapsed between the conceptualization of a product and the time at which the product is ready for selling.
- The commercial embedded product market is highly competitive and time to market the product is critical factor in the success of commercial embedded product.
- There may be multiple players in embedded industry who develop products of the same category (like mobile phone).

5. Per Unit Cost and Revenue:-

- Cost is a factor which is closely monitored by both end user and product manufacturer.
- Cost is highly sensitive factor for commercial products
- Any failure to position the cost of a commercial product at a nominal rate may lead to the failure of the product in the market.

- Proper market study and cost benefit analysis should be carried out before taking a decision on the per-unit cost of the embedded product.

- The ultimate aim of the product is to generate marginal profit so the budget and total cost should be properly balanced to provide a marginal profit.

FORMALISMS FOR SYSTEM DESIGN:

Visual language that can be used to capture all these design tasks: the ***Unified Modeling Language (UML).***UML was designed to be useful at many levels of abstraction in the design process. UML is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction.

UML is an ***object-oriented*** modeling language. We will see precisely what we mean by an object in just a moment, but object-oriented design emphasizes two concepts of importance:

It encourages the design to be described as a number of interacting objects, rather than a few large monolithic blocks of code.

At least some of those object will correspond to real pieces of software or hardware in the system. We can also use UML to model the outside world that interacts with our system, in which case the objects may correspond to people or other machines. It is sometimes important to implement something we think of at a high level as a single object using several distinct pieces of code or to otherwise break up the object correspondence in the implementation However, thinking of the design in terms of actual objects helps us understand the natural structure of the system. Object-oriented (often abbreviated OO) specification can be seen in two complementary ways:

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

Object-oriented specification provides a basic set of primitives that can be used to describe systems with particular attributes, irrespective of the relationships of those systems' components to real-world objects.

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

A specification language may not be executable. But both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

Unified Modeling Language (UML)—the acronym is the name is a large language, and covering all of it is beyond the scope of this book. In this section, we introduce only a few basic concepts. In later chapters, as we need a few more UML concepts, we introduce them to the basic modeling elements introduced here.

Because UML is so rich, there are many graphical elements in a UML diagram. It is important to be careful to use the correct drawing to describe something for instance; UML distinguishes between arrows with open and filled-in arrowheads, and solid and broken lines. As you become more familiar with the language, uses of the graphical primitives will become more natural to you.

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

1. Structural Description:

By ***structural description***, we mean the basic components of the system; we will learn how to describe how these components act in the next section. The principal component of an object-oriented design is, naturally enough, the ***object***. An object includes a set of ***attributes*** that define its internal state.

When implemented in a programming language, these attributes usually become variables or constants held in a data structure. In some cases, we will add the type of the attribute after the attribute name for clarity, but we do not always have to specify a type for an attribute. An object describing a display (such as a CRT screen) is shown in UML notation in Figure a).

The text in the folded-corner page icon is a ***note***; it does not correspond to an object in the system and only serves as a comment. The attribute is, in this case, an array of pixels that holds the contents of the display. The object is identified in two ways: It has a unique name, and it is a member of a ***class***. The name is underlined to show that this is a description of an object and not of a class.

A class is a form of type definition—all objects derived from the same class have the same characteristics, although their attributes may have different values. A class defines the attributes that an object may have. It also defines the ***operations*** that determine how the object interacts with the rest of the world. In a programming language, the operations would become pieces of code used to manipulate the object.

The UML description of the *Display* class is shown in Figure b). The class has the name that we saw used in the *d*1 object since *d*1 is an instance of class *Display*.

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

A class defines both the ***interface*** for a particular type of object and that object's ***implementation***. When we use an object, we do not directly manipulate its attributes—we can only read or modify the object's state through the operations that define the interface to the object.

As long as we do not change the behavior of the object seen at the interface, we can change the implementation as much as we want. This lets us improve the system by, for example, speeding up an operation or reducing the amount of memory required without requiring changes to anything else that uses the object.

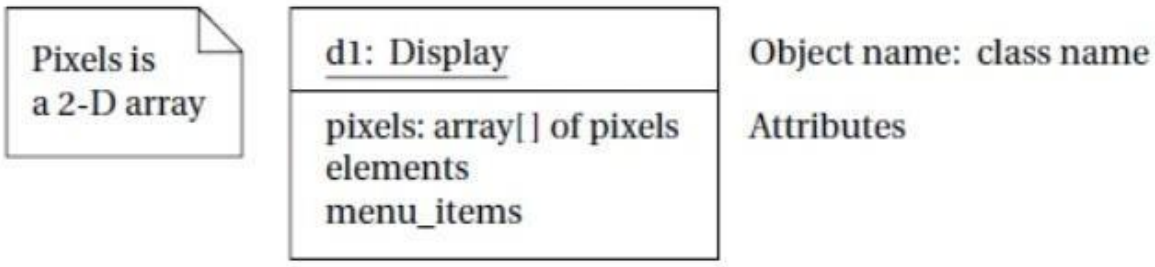


Fig a) An object in UML notation

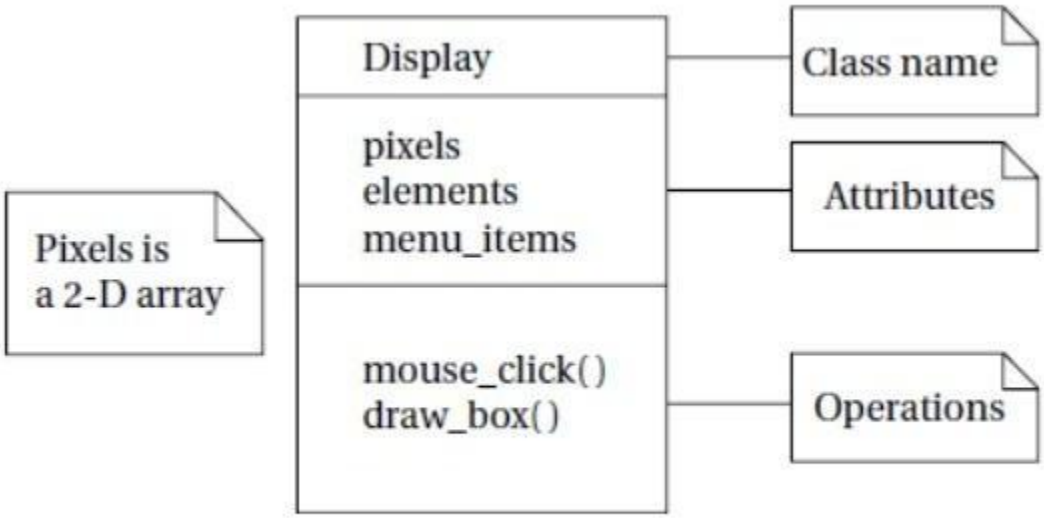


Fig b) A class in UML notation

Clearly, the choice of an interface is a very important decision in object-oriented design. The proper interface must provide ways to access the object's state (since we cannot directly see the attributes) as well as ways to update the state.

We need to make the object's interface general enough so that we can make full use of its capabilities. However, excessive generality often makes the object large and slow. Big, complex interfaces also make the class definition difficult for designers to understand and use properly.

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

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

■ ***Aggregation*** describes a complex object made of smaller objects.

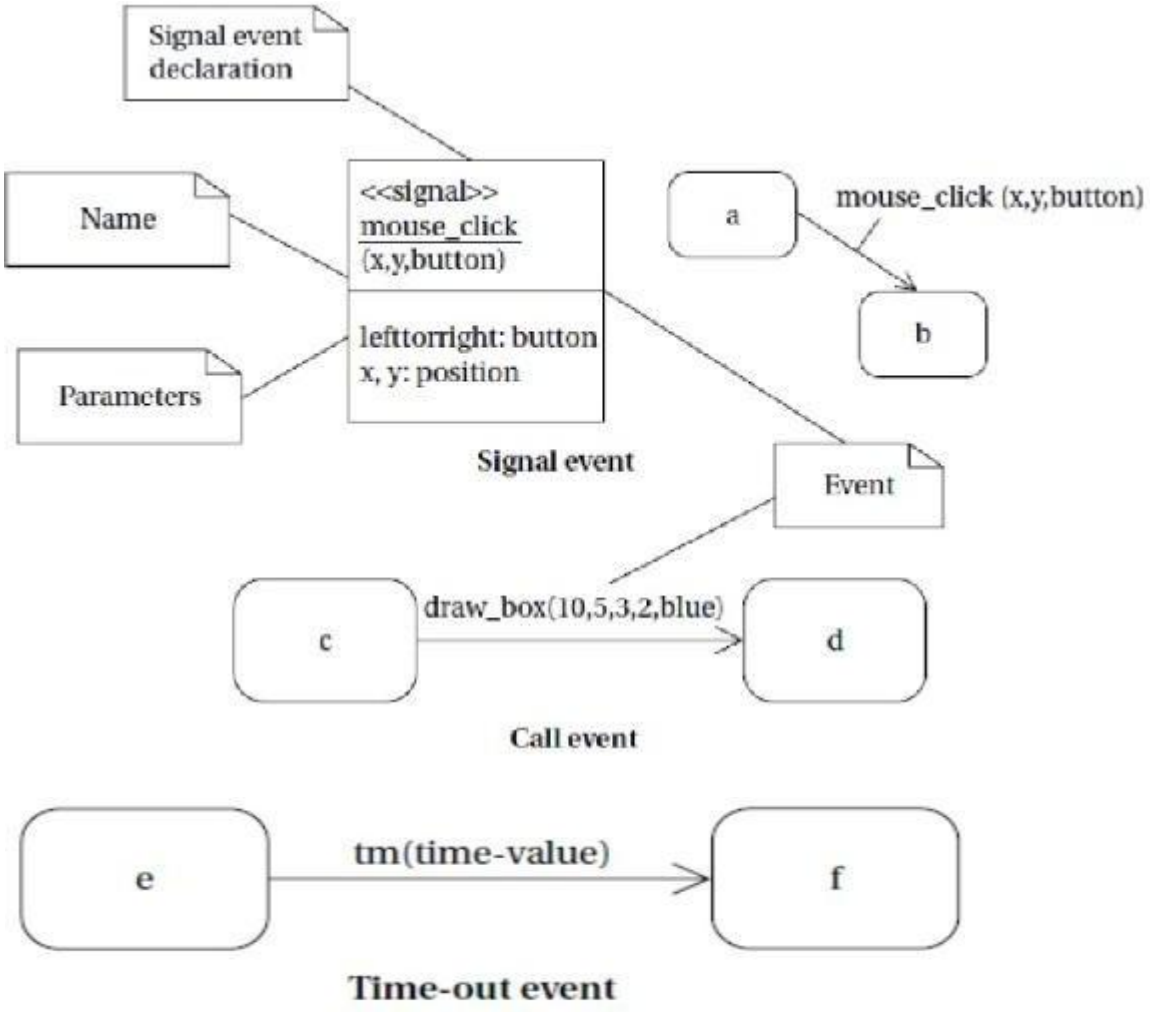
■***Composition*** is a type of aggregation in which the owner does not allow access to the component objects.

■ ***Generalization*** allows us to define one class in terms of another.

2. Behavioral Description:

We have to specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a ***state machine***.

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



An event is some type of action. The event may originate outside the system, such as a user pressing a button. It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine.We will concentrate on the following three types of events defined by UML, as illustrated in Figure 1.8 c):

A ***signal*** is an asynchronous occurrence. It is defined in UML by an object that is labeled as a <<*signal*>>. The object in the diagram serves as a declaration of the event's existence. Because it is an object, a signal may have parameters that are passed to the signal's receiver.

A ***call event*** follows the model of a procedure call in a programming language.

■A ***time-out event*** causes the machine to leave a state after a certain amount of time. The label *tm(time-value)* on the edge gives the amount of time after which the transition occurs. A

time-out is generallyimplemented with an external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism.

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

The control box may have familiar controls such as a throttle, emergency stop button, and so on. Since the train receives its electrical power from the two rails of the track, the control box can send signals to the train over the tracks by modulating the power supply voltage. As shown in the figure, the control panel sends packets over the tracks to the receiver on the train.

The train includes analog electronics to sense the bits being transmitted and a control system to set the train motor's speed and direction based on those commands.

Each packet includes an address so that the console can control several trains on the same track; the packet also includes an error correction code (ECC) to guard against transmission errors. This is a one-way communication system the model train cannot send commands back to the user.

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:

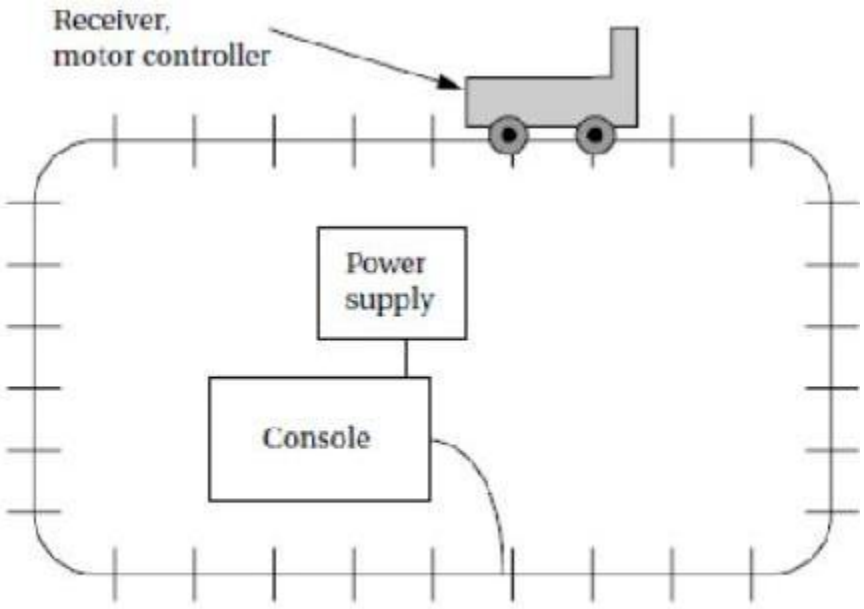
The console shall be able to control up to eight trains on a single track.

The speed of each train shall be controllable by a throttle to at least 63 different levels in each direction (forward and reverse).

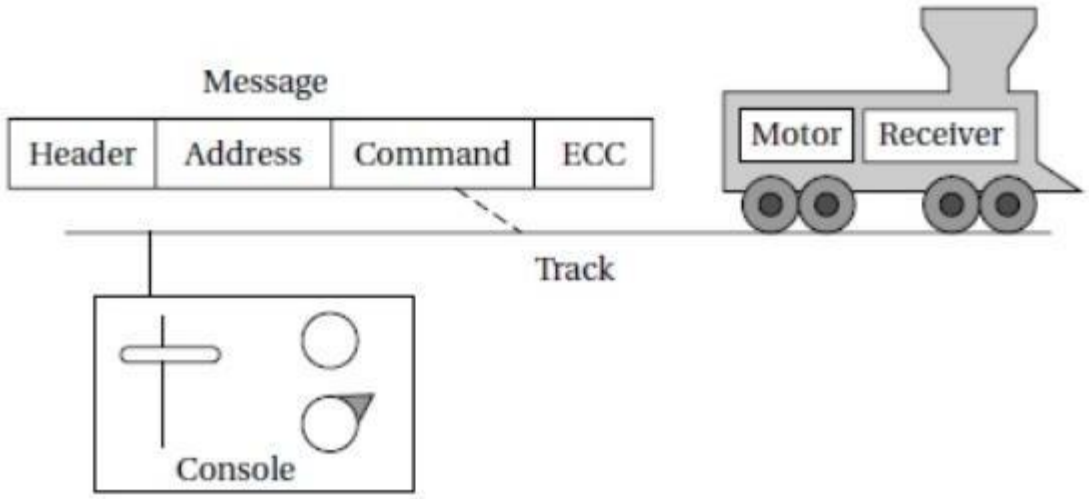
There shall be an inertia control that shall allow the user to adjust the responsiveness of the train to commanded changes in speed. Higher inertia means that the train responds more slowly to a change in the throttle, simulating the inertia of a large train. The inertia control will provide at least 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

Fig A Model train control system

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

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.

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, 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.

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)

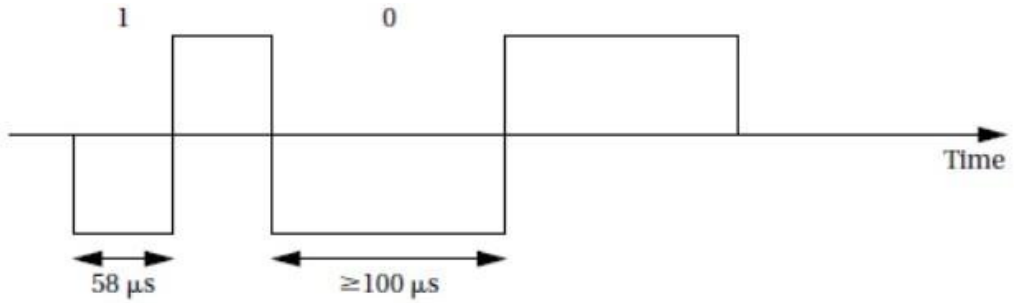


Fig Bit encoding in DCC.

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 addresstransmitted 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 1.4. Each of these subsystems has its own internal structure.

The basic relationship between them is illustrated in Figure 1.5. 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.

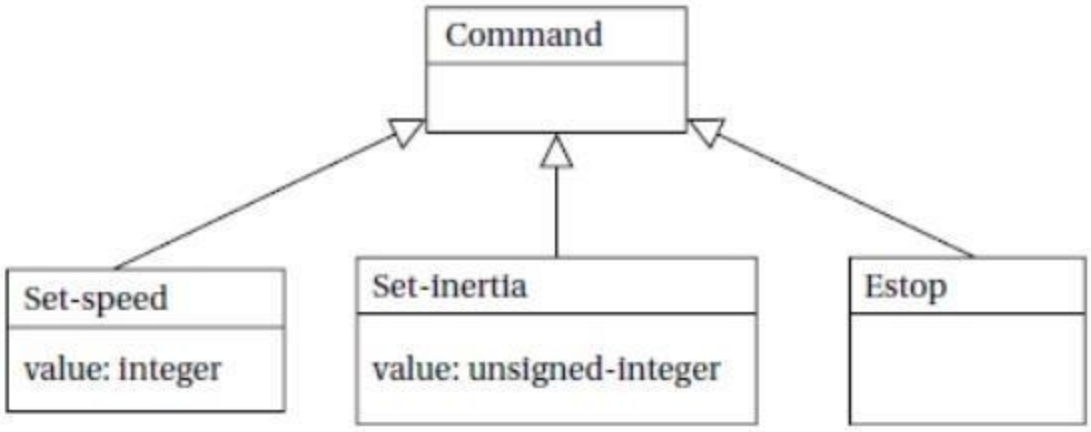


Fig Class diagram for the train controller messages.

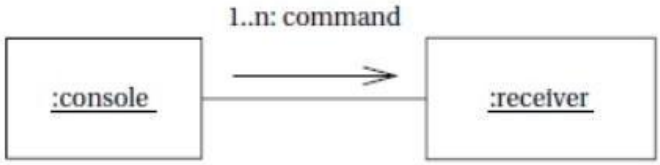


Fig UML collaboration diagram for major subsystems of the train controller system.

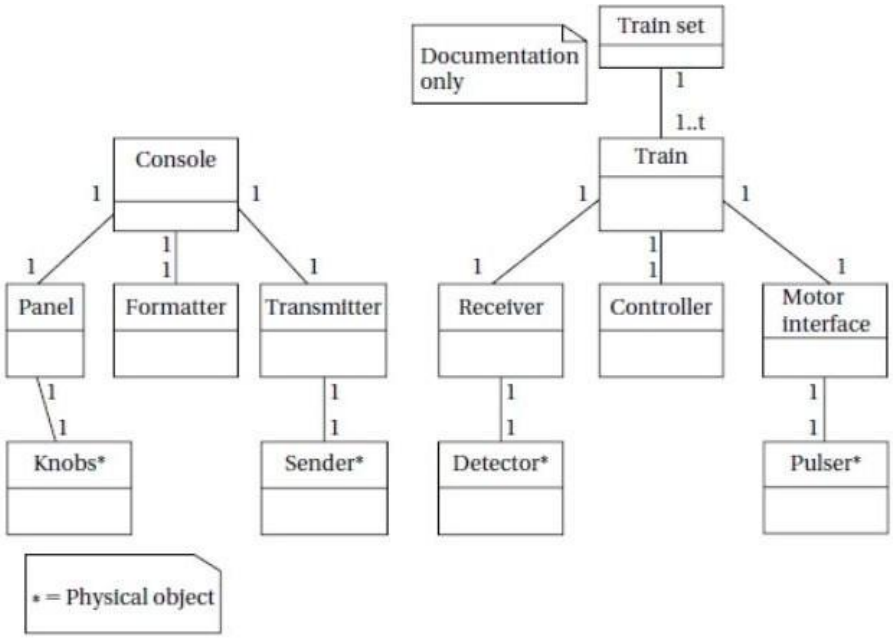


Fig A UML class diagram for the train controller showing the composition of the subsystems.

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. The UML class diagram is shown in Figure 1.6. 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.

UNIT II
INTRODUCTION TO EMBEDDED C AND APPLICATIONS

BASIC C DATA TYPES

Let's start by looking at how ARM compilers handle the basic C data types. We will see that some of these types are more efficient to use for local variables than others. There are also differences between the addressing modes available when loading and storing data of each type.

ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is a RISC load/store architecture. In other words you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

Early versions of the ARM architecture (ARMv1 to ARMv3) provided hardware support for loading and storing unsigned 8-bit and unsigned or signed 32-bit values.

Table5.1 Load and store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMv4	LDRB	load an unsigned 8-bit value
	STRB	store a signed or unsigned 8-bit value
	LDR	load a signed or unsigned 32-bit value
	STR	store asignedorunsigned 32-bit value
ARMv4	LDRSB	load a signed 8-bit value
	LDRH	load an unsigned 16-bit value
	LDRSH	load a signed 16-bit value
	STRH	store asignedorunsigned 16-bit value
ARMv5	LDRD	load a signed or unsigned 64-bit value
	STRD	store asignedorunsigned 64-bit value

These architectures were used on processors prior to the ARM7TDMI. Table 5.1 shows the load/store instruction classes available by ARM architecture.

InTable 5.1loadsthatacton8- or16-bitvalues extendthevalue to32bitsbefore writing to an ARM register. Unsigned values are zero-extended, and signed values sign-extended. This means that the cast of a loaded value to an inttype does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an intto smaller type does notcost extra instructions on a store.

The ARMv4 architecture and above support signed 8-bit and 16-bit loads and stores directly, through new instructions. Since these instructions are a later addition, they do not support as many addressing modes as the pre-ARMv4 instructions. (See Section 3.3

fordetailsofthedifferentaddressingmodes.)Wewillseetheeffectofthisintheexample checksum_v3in Section 5.2.1.

Finally, ARMv5 adds instruction support for 64-bit load and stores. This is available in ARM9E and latercores.

Prior to ARMv4, ARM processors were not good at handling signed 8-bit or any 16-bit values. Therefore ARM C compilers define charto be an unsigned 8-bit value, rather than a signed 8-bitvalueasistypicalinmanyothercompilers.

Compilers *armcc* and *gcc* use the datatype mappings in Table 5.2 foranARM target. The exceptional case for type charis worth noting as it can cause problems when you are porting code from another processor architecture. A common example is using a chartype variable ias a loop counter, with loop continuation condition i 0. As iis unsigned for the ARM compilers, the loop will never terminate. Fortunately *armcc* produces a warning inthis situation: *unsigned comparison with 0*.

Compilers also provide an override switch to make charsigned. For example, the command line option -fsigned-charwill make char signedon*gcc*.Thecommandlineoption-zcwillhavethesameeffectwith*armcc*.

For the rest of this book we assume that you are using an ARMv4 processor or above. This includes ARM7TDMI and all later processors.

Table5.2 C compiler datatypemappings.

CDataType	Implementation
char	unsigned 8-bitbyte
short	signed 16-bithalfword
int	signed32-bitword
long	signed32-bitword
long long	signed 64-bit double word

LOCAL VARIABLE TYPES

ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit datatype, intor long, for local variables wherever possible. Avoid using charand shortaslocalvariabletypes,evenifyouaremanipulatingan8-or16-bitvalue.The one exception is when you want wrap- around to occur. If you require modulo arithmetic of the form 255 1 0, then use the chartype.

To see the effectoflocal variable types, let's considera simple example. We'll look in detail at a checksum function that sums the values in a data packet. Most communication protocols (such as TCP/IP) have a checksum or cyclic redundancy check (CRC) routine to check for errors in a data packet.

The following code checksums a data packet containing 64 words. It shows why you should avoid using char for local variables.

```
int checksum_v1(int *data)
{
char i; int sum=0;

for (i=0; i<64; i++)
{
sum += data[i];
}
return sum;
}
```

Atfirstsightitlooksasthoughdeclaringiasachariseffi cient.Youmaybethinking that a charuses less register spaceorless space on the ARM stack than an int. On the ARM, both these assumptions are wrong. All ARM registers are 32-bit and all stack entries are at least 32-bit. Furthermore, to implement the i++ exactly, the compiler must account for the case when i = 255. Any attempt to

increment255shouldproducetheanswer0.

Consider the compiler output for this function. We've added labels and comments to make the assembly clear.

```
check
s
um_v
1
MOV         r2,r0               ; r2 = data
MOV         r0,#0               ; sum = 0
MOV         r1,#0               ; i = 0
checksum_v1_loop
LDR         r3,[r2,r1,LSL       ;     r3     =
            #2]                 data[i]
ADD         r1,r1,#1            ; r1 = i+1
```

```
AND     r1,r1,#0xff     ; i = (char)r1
CMP     r1,#0x40        ; compare i, 64
ADD     r0,r3,r0        ; sum += r3
BCC     checksum_v1_loop ; if (i<64) loop
MOV     pc,r14          ; return sum
```

Now compare this to the compiler output where instead we declare ias an unsigned int.

```
checksum_v2
MOV     r2,r0           ; r2 = data
MOV     r0,#0           ; sum = 0
MOV     r1,#0           ; i = 0
checksum_v2_loop
LDR     r3,[r2,r1,LSL #2] ; r3 = data[i]
ADD     r1,r1,#1        ; r1++
CMP     r1,#0x40        ; compare i, 64
ADD     r0,r3,r0        ; sum += r3
BCC     checksum_v2_loop ; if (i<64) goto loop
MOV     pc,r14          ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce ito the range 0 to 255beforethecomparisonwith64. Thisinstructiondisappearsinthesecondcase.
Next, suppose the data packet contains 16-bit values and we need a 16-bit checksum. It is tempting to write the following C code:

```
short checksum_v3(short *data)
{
unsigned int i; short sum=0;

for (i=0; i<64; i++)
{
sum = (short)(sum + data[i]);

}
return sum;
}
```

You may wonder why the forloop body doesn't contain the code

```
sum += data[i];
```

With *armcc* this code will produce a warning if you enable implicit narrowing cast warnings using the compiler switch -W+ n. The expression sum+data[i]is an integer and so can onlybeassignedtoashortusingan(implicitorexplicit) narrowingcast. Asyou cansee inthefollowingassemblyoutput,thecompilermustinsertextrainstructionstoimplement the narrowing cast:

```
checksu
m
_v3
MOV         r2,r0       ; r2 = data
MOV         r0,#0       ; sum = 0
MOV         r1,#0       ; i = 0
checksum_v3_loop
ADD         r3,r2,r1,LSL        ; r3 = &data[i]
            #1
LDRH        r3,[r3,#0]          ; r3 = data[i]
ADD         r1,r1,#1            ; i++
CMP         r1,#0x40            ; compare i, 64
ADD         r0,r3,r0            ; r0 = sum + r3
MOV         r0,r0,LSL #16
MOV         r0,r0,ASR #16       ; sum = (short)r0
BCC         checksum_v          ; if (i<64) goto
            3_l oop             loop
MOV         pc,r14              ; return sum
```

The loop is now three instructions longer than the loop for example checksum_v2 earlier! There are two reasons for the extra instructions:

- The LDRH instruction does not allow for a shifted address offset as the LDR instruction did in checksum_v2. Therefore the first ADD in the loop calculates the address of item i in the array. The LDRH loads from an address with no offset. LDRH has fewer addressing modes than LDR as it was a later addition to the ARM instruction set. (See Table 5.1.)
- The cast reducing total+array[i] to a short requires two MOV instructions. The compiler shifts left by 16 and then right by 16 to implement a 16-bit sign extend. The shift right is a sign-extending shift so it replicates the sign bit to fill the upper 16 bits.

We can avoid the second problem by using an int type variable to hold the partial sum. We only reduce the sum to a short type at the function exit.

However, the first problem is a new issue. We can solve it by accessing the array by incrementing the pointer *data* rather than using an index as in data[i]. This is efficient regardless of array type size or element size. All ARM load and store instructions have a post increment addressing mode.

Example:

The checksum_v4 code fixes all the problems we have discussed in this section. It uses int type local variables to avoid unnecessary casts. It increments the pointer data instead of using an index offset data[i].

```
short checksum_v4(short *data)
{
```

```
unsigned int i; int sum=0;

for (i=0; i<64; i++)
{
sum += *(data++);
}
return (short)sum;
}
```

The compiler is still performing one cast to a 16-bit range, on the function return. You could remove this also by returning an int result as discussed in Section 5.2.2.

FUNCTION ARGUMENT TYPES

We saw in Section 5.2.1 that converting local variables from types char or short to type int increases performance and reduces code size. The same holds for function arguments. Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{
return a + (b>>1);
}
```

This function is a little artificial, but it is a useful test case to illustrate the problems faced by the compiler. The input values a, b, and the return value will be passed in 32-bit ARM registers. Should the compiler assume that these 32-bit values are in the range of a short type, that is, 32,768 to 32,767? Or should the compiler force values to be in this range by sign- extending the lowest 16 bits to fill the 32-bit register? The compiler must make compatible decisions for the function caller and callee. Either the caller or callee must perform the cast to a short type.

We say that function arguments are passed *wide* if they are not reduced to the range of the type and *narrow* if they are. You can tell which decision the compiler has made by looking at the assembly output for add_v1. If the compiler passes arguments wide, then the callee must reduce function arguments to the correct range. If the compiler passes arguments narrow, then the caller must reduce the range. If the compiler returns values wide, then the caller must reduce the return value to the correct range. If the compiler returns values narrow, then the callee must reduce the range before returning the value.

For *armcc* in ADS, function arguments are passed narrow and values returned narrow. In other words, the caller casts argument values and the callee casts return values. The compiler uses the ANSI prototype of the function to determine the datatypesofthefunction arguments.
The *armcc* outputforadd_v1showsthatthecompilercaststhereturnvaluetoashort type,butdoesnotcasttheinputvalues. It assumes that the caller has already ensured that the 32-bitvalues r0and r1are in the range of the shorttype. This shows narrowpassing of arguments and return value.

```
add_v1
ADD             r0,r0,r1,ASR #1     ; r0 = (int)a + ((int)b>>1)
MOV             r0,r0,LSL #16
MOV             r0,r0,ASR #16       ; r0 = (short)r0
MOV             pc,r14              ; return r0
```

The *gcc* compiler we used is more cautious and makes no assumptions about the range of argument value. This version of the compiler reduces the input arguments to the rangeofashortin both the callerandthe callee. Italsocasts thereturnvalueto ashorttype. Here is the compiled code for add_v1:

```
add_v
1_gcc
MOV     r0, r0, LSL #16
MOV     r1, r1, LSL #16
MOV     r1, r1, ASR #17     ; r1 = (int)b>>1
ADD     r1, r1, r0, ASR #16 ; r1 += (int)a
MOV     r1, r1, LSL #16
MOV     r0, r1, ASR #16     ; r0 = (short)r1
MOV     pc, lr              ; return r0
```

Whateverthemeritsofdifferentnarrowandwidecallingprotocols, youcanseethat charor shorttype function arguments and return values introduce extra casts. These increase code size and decrease performance. It is more efficient to use the inttypefor functionargumentsandreturnvalues,evenifyouareonlypassingan8-bitvalue.

SIGNED VERSUS UNSIGNED TYPES

The previous sections demonstrate the advantages of using intrather than a charor short type for local variables and function arguments. This section compares the efficiencies of signedintand unsignedint.
Ifyourcodeusesaddition, subtraction, andmultiplication, thenthereisnoperformance difference between signed and unsigned operations. However, there is a difference when it comes to division. Consider the following short example thataveragestwointegers:

```
int average_v1(int a, int b)
{
return (a+b)/2;
}
```

This compiles to

average_v1

```
ADD     r0,r0,r1            ; r0 = a + b
ADD     r0,r0,r0,LSR #31    ; if (r0<0) r0++
MOV     r0,r0,ASR #1        ; r0 = r0>>1
MOV     pc,r14              ; return r0
```

Notice that the compiler adds one to the sum before shifting by right if the sum is negative. In other words it replaces *x*/2 by the statement:

(x<0) ? ((x+1)>>1): (x>>1)

It must do this because *x* is signed. In C on an ARM target, a divide by two is not a right shift if *x* is negative. For example, 3 1 2 but 3/2 1. Division rounds towards zero, but arithmetic right shift rounds towards –.

It is more efficient to use unsigned types for divisions. The compiler converts unsigned power of two divisions directly to right shifts. For general divisions, the divide routine in the C library is faster for unsigned types. See Section 5.10 for discussion on avoiding divisions completely.

SUMMARY **The Efficient Use of C Types**

- For local variables held in registers, don't use a char or short type unless 8-bit or 16-bit modular arithmetic is necessary. Use the signed or unsigned int types instead. Unsigned types are faster when you use divisions.
- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint. The ARMv4 architecture is efficient at loading and storing all data widths provided you traverse arrays by incrementing the array pointer. Avoid using offsets from the base of the array with short type arrays, as LDRH does not support this.
- Use explicit casts when reading array entries or global variables into local variables, or writing local variables out to array entries. The casts make it clear that for fast operation you are taking a narrow width type stored in memory and expanding it to a wider type in the registers. Switch on *implicit narrowing cast* warnings in the compiler to detect implicit casts.
- Avoid implicit or explicit narrowing casts in expressions because they usually cost extra cycles. Casts on loads or stores are usually free because the load or store instruction performs the cast for you.
- Avoid char and short types for function arguments or return values. Instead use the int type even if the range of the parameter is smaller. This prevents the compiler performing unnecessary casts.

C LOOPING STRUCTURES

Thissectionlooksatthemostefficientwaystocodeforandwhileloopsonthe ARM.Westartbylookingatloopswithafixed numberofiterationsand then move on to loop swith avariable number of iterations. Finally we look at loop unrolling.

LOOPS WITH A FIXED NUMBER OF ITERATIONS

What is the most efficient way to write a for loop on the ARM? Let's return to our checksum example and look at the looping structure.

Here is the last version of the 64-word packet checksum routine we studied in

Thisshowshowthecompilertreatsaloopwith incrementingcounti++.

```
int checksum_v5(int *data)
{
unsigned int i; int sum=0;

for (i=0; i<64; i++)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checks
um_v5
MOV         r2,r0                ; r2 = data
MOV         r0,#0                ; sum = 0
MOV         r1,#0                ; i = 0
checksum_v5_loop
LDR         r3,[r2],#4           ; r3 = *(data++)
ADD         r1,r1,#1             ; i++
CMP         r1,#0x40             ; compare i, 64
ADD         r0,r3,r0             ; sum += r3
BCC         checksum_v5_l        ; if (i<64) goto loop
            oop
MOV         pc,r14               ; return sum
```

It takes three instructions to implement the forloop structure:

- An ADDto increment i
- A compare to check if iis less than 64
- A conditional branch tocontinuetheloop if i< 64

This is not efficient. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result
- A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored in the condition flags. Since we are no longer using i as an array index, there is no problem in counting down rather than up.

EXAMPLE 2

This example shows the improvement if we switch to a decrementing loop rather than an incrementing loop.

```
int checksum_v6(int *data)
{
unsigned int i; int sum=0;

for (i=64; i!=0; i--)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum
_v6
MOV             r2,r0               ; r2 = data
MOV             r0,#0               ; sum = 0
MOV             r1,#0x40            ; i = 64
checksum_v6_loop
LDR             r3,[r2],#4          ; r3 = *(data++)
SUBS            r1,r1,#1            ; i-- and set flags
ADD             r0,r3,r0            ; sum += r3
BNE             checksum_v6_l       ; if (i!=0) goto loop
                oop
MOV             pc,r14              ; return sum
```

The SUBS and BNE instructions implement the loop. Our checksum example now has the minimum number of four instructions per loop. This is much better than six for checksum_v1 and eight for checksum_v3.

For an unsigned loop counter i we can use either of the loop continuation conditions i!=0 or i>0. As i can't be negative, they are the same condition. For a signed loop counter, it is tempting to use the condition i>0 to continue the loop. You might expect the compiler to generate the following two instructions to implement the loop:

```
SUBS            r1,r1,#1    ; compare i with 1, i=i-1
BGT             loop        ; if (i+1>1) goto loop
```

In fact, the compiler will generate

```
r1,r1,#1        ; i--

r1,#0           ; compare i with 0

loop            ; if (i>0) goto loop
```

The compiler is not being inefficient. It must be careful about the case when i=-0x80000000 because the two sections of code generate different answers in this case. For the first piece of code the SUBS instruction compares i with 1 and then decrements i. Since -0x80000000 < 1, the loop terminates. For the second piece of code, we decrement i and then compare with 0. Modulo arithmetic means that i now has the value +0x7fffffff, which is greater than zero. Thus the loop continues for many iterations.
Of course, in practice, i rarely takes the value -0x80000000. The compiler can't usually determine this, especially if the loop starts with a variable number of iterations (see Section 5.3.2).
Therefore you should use the termination condition i!=0 for signed or unsigned loop counters. It saves one instruction over the condition i>0 for signed i.

LOOPS USING A VARIABLE NUMBER OF ITERATIONS

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable N giving the number of words in the data packet. Using the lessons from the last section we count down until N 0 and don't require an extra loop counter i.
The checksum_v7 example shows how the compiler handles a for loop with a variable number of iterations N.

```
int checksum_v7(int *data, unsigned int N)
{
int sum=0;

for (; N!=0; N--)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v7
MOV     r2,#0           ; sum = 0
CMP     r1,#0           ; compare N, 0
BEQ     checksum_v7_end ; if (N==0) goto end
```

```
checksum_v7_loop
                LDR     r3,[r0],#4      ; r3 = *(data++)
                SUBS    r1,r1,#1        ; N-- and set flags
                ADD     r2,r3,r2        ; sum += r3
                BNE     checksum_v7_l   ; if (N!=0) goto loop
                        oop
checksum_v7_end
                    M   r0,r2       ; r0 = sum

                  O
                V
                    M   pc,r14      ; return r0

                  O
                V
```

Notice thatthe compiler checks that Nis nonzeroon entry to the function. Often this check is unnecessary since you know thatthearraywon'tbeempty.Inthiscase ado-whileloopgivesbetterperformanceandcodedensitythanaforloop.

EXAMPLE 3 This exampleshowshowtouseado-whileloop toremovethetestforNbeingzero that occursina forloop.

```
int checksum_v8(int *data, unsigned int N)
{
```

```
int sum=0;

    do
    {
    sum += *(data++);
    } while (--N!=0); return sum;
    }
```

The compiler output is now

```
checksum_v8
MOV                 r2,#0           ; sum = 0
checksum_v8_loop
                    LDR     r3,[r0],#4      ; r3 = *(data++)
                    SUBS    r1,r1,#1        ; N-- and set flags
                    ADD     r2,r3,r2        ; sum += r3
                    BNE     checksum_v8_l   ; if (N!=0) goto loop
                            oop
                    MOV     r0,r2           ; r0 = sum
                    MOV     pc,r14          ; return r0
```

Comparethiswiththeoutputforchecksum_v7toseethetwo-cyclesaving.

LOOP UNROLLING

We saw in Section 5.3.1 that each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrementtheloopcountandaconditionalbranch.

We call these instructions the *loop overhead*. On ARM7 or ARM9 processors thesubtracttakesonecycleand thebranchthreecycles,givinganoverheadoffourcycles perloop.
You can save some of these cycles by *unrolling* a loop—repeating the loop body several times, and reducing the number of loop iterations by the same proportion. For example, let's unrollour packet checksum example four times.

EXAMPLE 4 **The following code unrolls our packet checksum loop by four times. We assume that the number ofwords in the packet Nis amultiple offour.**

```
int checksum_v9(int *data, unsigned int N)
{
int sum=0;

do
{
sum += *(data++); sum += *(data++); sum += *(data++); sum += *(data++); N -= 4;
```

```
} while ( N!=0); return sum;
}
```

This compiles to

```
checksum_v9
MOV r2,#0                       ; sum = 0
checksum_v9_loop
                        LDR     r3,[r0],#4          r3 = *(data++)
                        SUBS    r1,r1,#4            N -= 4 & set flags
                        ADD     r2,r3,r2            sum += r3
                        LDR     r3,[r0],#4          r3 = *(data++)
                        ADD     r2,r3,r2            sum += r3
                        LDR     r3,[r0],#4          r3 = *(data++)
                        ADD     r2,r3,r2            sum += r3
                        LDR     r3,[r0],#4          r3 = *(data++)
                        ADD     r2,r3,r2            sum += r3
                        BNE     checksum_v9_loop    if (N!=0) goto loop
                        MOV     r0,r2               r0 = sum
                        MOV     pc,r14              return r0
```

We have reduced the loop overhead from 4N cycles to (4N)/4 N cycles. On the ARM7TDMI, this accelerates the loop from 8 cycles per accumulate to 20/4 5 cycles per accumulate, nearly doubling the speed! For the ARM9TDMI, which has a faster load instruction, the benefit is even higher.

There are two questions you need to ask when unrolling a loop:

- How many times should I unroll the loop?
- What if the number of loop iterations is not a multiple of the unroll amount? For example, what if N is not a multiple of four in checksum_v9?

To start with the first question, only unroll loops that are important for the overall performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important code from the cache.
Suppose the loop is important, for example, 30% of the entire application. Suppose you unroll the loop until it is 0.5 KB in code size (128 instructions). Then the loop overhead is at most 4 cycles compared to a loop body of around 128 cycles. The loop overhead cost is 3/128, roughly 3%. Recalling that the loop is 30% of the entire application, overall the loop overhead is only 1%. Unrolling the code further gains little extra performance, but has a significant impact on the cache contents. It is usually not worth unrolling further when the gain is less than 1%.
For the second question, try to arrange it so that array sizes are multiples of your unroll amount. If this isn't possible, then you must add extra code to take care of the leftover cases. This increases the code size a little but keeps the performance high.

EXAMPLE 5 **This example handles the checksum of any size of data packet using a loop that has been unrolled four times.**

```
int checksum_v10(int *data, unsigned int N)
{
```

```
unsigned int i; int sum=0;

for (i=N/4; i!=0; i--)
{
sum += *(data++); sum += *(data++); sum +=
*(data++); sum +=*(data++);
}
for (i=N&3; i!=0; i--)
{
                    sum += *(data++);
}
return sum;
}
```

The second forloophandles the remainingcases when Nis not a multiple offour. Note that both N/4and N&3can be zero, so wecan't use do-whileloops.

SUMMARY **Writing Loops Efficiently**

- Use loops that count down to zero. Then the compiler does not need to allocate aregistertohold thetermination value,andthecomparisonwithzeroisfree.
- Useunsignedloopcountersbydefaultandthecontinuationconditioni!=0ratherthan **i>0. This will ensure that the loop overhead is only two instructions.**
- Usedo-whileloops ratherthanforloopswhenyouknowtheloopwilliterateatleast once. Thissaves thecompilercheckingtoseeiftheloopcountiszero.
- Unroll important loops to reduce the loop overhead. Do not overunroll. If the loop overhead is small asaproportion ofthetotal, then unrollingwillincrease codesize and hurt the performance of the cache.
- Try to arrange that the number of elements in arrays are multiples of four or eight. You can then unroll loopseasilybytwo,four,oreighttimeswithoutworryingaboutthe leftover arrayelements.

REGISTER ALLOCATION

The compiler attempts to allocate aprocessorregister to each localvariable you use in aCfunction.Itwilltrytouse the same register for different local variables if the use of the variables do not overlap. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called *spilled* or *swapped out* variables since they are written out to memory (in a similar way virtual memory is swapped outtodisk).Spilledvariablesareslowtoaccesscomparedtovariables allocated toregisters.
To implement a function efficiently, you need to

- minimize the numberof spilled variables
- ensurethatthemostimportantandfrequentlyaccessedvariablesarestoredinregisters

First let's look at the number of processor registers the ARM C compilers have avail- able for allocating variables. Table 5.3shows thestandard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by Ccompilers.

Table 5.3 C compiler register usage.

Alternate Register number	register names	ATPCS register usage
r0	*a1*	**Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.**
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	**General variable registers. The function must preserve the callee values of these registers.**
r5	*v2*	
r6	*v3*	
r7	*v4*	
r8	*v5*	

r9	***v6sb***	**General variable register. The function must preserve the callee** value of this register except when compiling for *read-write position independence* (RWPI). Then *r9* holds the *static base* address. This is the address of the read-write data.
r10	***v7sl***	**General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then *r10* holds the stack limit address.**
r11	***v8fp***	**General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.**
r12	***ip***	**A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intra-procedure call requirements.**
r13	***sp***	**The stack pointer, pointing to the full descending stack.**
r14	***lr***	**The link register. On a function call this holds the return address.**
r15	*pc*	The program counter.

Provided the compiler is not using software stack checking or a frame pointer, then the C compiler can use registers *r0* to *r12* and *r14* to hold variables. It must save the callee values of *r4* to *r11* and *r14* on the stack if using these registers. In theory, the C compiler can assign 14 variables to registers without spillage. In practice, some compilers use a fixed register such as *r12* for intermediate scratch working and do not assign variables to this register. Also, complex expressions require intermediate working registers to evaluate. Therefore, to ensure good assignment to registers, you should try to limit the internal loop of functions to using at most 12 local variables.

If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use. A variable used inside a loop counts multiple times. You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

The register keyword in C hints that a compiler should allocate the given variable to a register. However, different compilers treat this keyword in different ways, and different architectures have a different number of available registers (for example, Thumb and ARM). Therefore we recommend that you avoid using register and rely on the compiler's normal register allocation routine.

SUMMARY **Efficient Register Allocation**

- Try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.
- You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

FUNCTION CALLS

The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers. The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well. The first four integer arguments are passed in the first four ARM registers: *r0,r1,r2,* and *r3*. Subsequent integer arguments are placed on the full descending stack, ascending in memory as in Figure 5.1. Function return integer values are passed in *r0*. This description covers only integer or pointer arguments. Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in *r0,r1*. The compiler may pass structures in registers or by reference according to command line compiler options.

The first point to note about the procedure call standard is the *four-register rule*. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions with more arguments, both the caller and callee must access the stack for some arguments. Note that for C++ the first argument to an object method is the *this* pointer. This argument is implicit and additional to the explicit arguments.

If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than mul- tiple arguments. Which arguments are related will depend on the structure of your software.

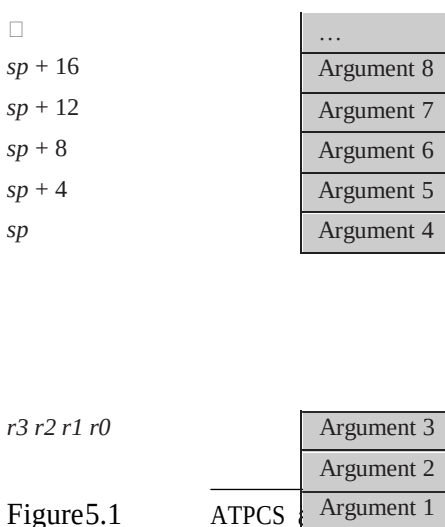


Figure 5.1 ATPCS argument passing.

The next example illustrates the benefits of using a structure pointer. First we show a typical routine to insert N bytes from array data into a queue. We implement the queue using a cyclic buffer with start address Q_start (inclusive) and end address Q_end (exclusive).

```
char *queue_bytes_v1(
char *Q_start,          /* Queue buffer start address */
char *Q_end,            /* Queue buffer end address */
char *Q_ptr,            /* Current queue pointer position */
char *data,             /* Data to insert into the queue   */
unsigned int N)         /* Number of bytes to insert */
```

```
{
do
{
*(Q_ptr++) = *(data++);

if (Q_ptr == Q_end)
{
Q_ptr = Q_start;
}
} while (--N); return Q_ptr;
}
```

This compiles to

```
queue_bytes_v1
                    STR         r14,[r13,#-4]!      save lr on the
                                                    stack
                    LDR         r12,[r13,#4]        r12 = N
queue_v1_loop
                    LDRB            r14,[r3],#1         r14 = *(data++)
                    STRB            r14,[r2],#1         *(Q_ptr++) = r14
                    CMP             r2,r1               if (Q_ptr ==
                                                        Q_end)
                    MOVEQ           r2,r0               {Q_ptr = Q_start;}
                    SUBS            r12,r12,#1          --N and set flags
                    BNE             queue_v1_l          if (N!=0) goto loop
                                    oop
                    MOV             r0,r2               r0 = Q_ptr
                    LDR             pc,[r13],#4         return r0
```

Compare this with a more structured approach using three function arguments.

EXAMPLE

The following code creates a Queuestructure and passes this to the function to reduce the number of functionarguments.

```
typedef struct {
char *Q_start;          /* Queue buffer start address */ char
*Q_end;                 /* Queue buffer end address */
char *Q_ptr;            /* Current queue pointer position */
} Queue;

void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
```

```
char *Q_ptr = queue->Q_ptr; char *Q_end = queue-
>Q_end;

do
{
*(Q_ptr++) = *(data++);

if (Q_ptr == Q_end)
{
Q_ptr = queue->Q_start;
}
} while (--N);
queue->Q_ptr = Q_ptr;
}
```

This compiles to

```
queue_bytes_v2
                    STR     r14,[r13,#-4]!      save lr on the stack
                    LDR     r3,[r0,#8]          r3 = queue->Q_ptr
                    LDR     r14,[r0,#4]         r14 = queue->Q_end
queue_v2_loop
                LDRB    r12,[r1],#1     ;   r12 = *(data++)
                STRB    r12,[r3],#1     ;   *(Q_ptr++) = r12
                CMP     r3,r14          ;   if (Q_ptr == Q_end)
                LDREQ   r3,[r0,#0]      ;       Q_ptr = queue->Q_start
                SUBS    r2,r2,#1        ;   --N and set flags
                BNE     queue_v2_loop   ;   if (N!=0) goto loop
                STR     r3,[r0,#8]      ;   queue->Q_ptr = r3
```

The queue_bytes_v2is one instruction longer than queue_bytes_v1, but it is in fact more efficient overall. The second version has only three function arguments rather than five. Each call to the function requires only three register setups. This compares with four register setups, a stack push, and a stack pull for the first version. There is a net saving of two instructions in function call overhead. There are likely further savings in the callee function, as it only needs to assign a single register to the Queuestructure pointer, rather than three registers in the nonstructured case.
There are other ways of reducing function call overhead if your function is very small and corrupts few registers (uses few local variables). Put the C function in the same C file as the functions that will call it. The C compiler then knows the code generated for the callee functionandcanmakeoptimizations in thecaller function:

- The caller function need not preserve registers that it can see the callee doesn't corrupt. Therefore the callerfunctionneednotsavealltheATPCScorruptibleregisters.
- If the callee function is very small, then the compiler can inline the code in the caller function. This removesthefunctioncalloverheadcompletely.

EXAMPLE

The function uint_to_hexconverts a 32-bitunsigned integer into an array ofeight hexa- decimal digits. It uses a helper functionnybble_to_hex, whichconverts a digitdin the range 0 to 15 to a hexadecimal digit.

```
unsigned int nybble_to_hex(unsigned int d)
{
if (d<10)
{
return d + '0';

return d - 10 + 'A';
}

void uint_to_hex(char *out, unsigned int in)
{
unsigned int i;

for (i=8; i!=0; i--)
{
in = (in<<4) | (in>>28); /* rotate in left by 4 bits */
*(out++) = (char)nybble_to_hex(in & 15);
}
}
```

When we compile this, we see that uint_to_hexdoesn't call nybble_to_hexatall! In the following compiled code, the compilerhasinlinedtheuint_to_hexcode. Thisismoreefficientthangeneratingafunctioncall.

```
uint_to_hex
MOV             r3,#8           ; i = 8
uint_to_hex_loop
                MOV     r1,r1,ROR #28   ; in = (in<<4)|(in>>28)
                AND     r2,r1,#0xf      ; r2 = in & 15
                CMP     r2,#0xa         ; if (r2>=10)
                ADDCS   r2,r2,#0x37     ;       r2 +='A'-10
                ADDCC   r2,r2,#0x30     ; else r2 +='0'
                STRB    r2,[r0],#1      ; *(out++) = r2
                SUBS    r3,r3,#1        ; i-- and set flags
                BNE     uint_to_hex_loop ; if (i!=0) goto loop
                MOV     pc,r14          ; return
```

The compiler will only inline small functions. You can ask the compiler to inline a functionusing the_inline keyword, although this keyword is only a hint and the compiler may ignore it (see Section 5.12 for more on inline functions). Inlining large functions can lead to big increases in code size without much performance improvement.

POINTER ALIASING

Two pointers are said to *alias* when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer. In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.
Let'sstartwithaverysimpleexample.Thefollowingfunctionincrementstwotimer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
*timer1 += *step;
*timer2 += *step;
}
```

This compiles to

```
timers_v1
        LD R            r3,[r0,#0]      ; r3 = *timer1

        LD R            r12,[r2,#0]     ; r12 = *step

        AD D            r3,r3,r12       ; r3 += r12

        ST R            r3,[r0,#0]      ; *timer1 = r3

        LD R            r0,[r1,#0]      ; r0 = *timer2

        LD R            r2,[r2,#0]      ; r2 = *step

        AD D            r0,r0,r2        ; r0 += r2

        ST R            r0,[r1,#0]      ; *timer2 = t0

        M O V           pc,r14          ; return
```

Note that the compiler loads from steptwice. Usually a compiler optimization called *common subexpression elimination* would kick in so that *stepwas only evaluated once, and the value reused for the second occurrence. However, the compiler can't use this optimization here. The pointers timer1and stepmight alias one another. In other words, the compiler cannot be sure that the write to timer1doesn't affect the read from step.

Inthiscasethesecondvalueof*stepisdifferentfromthefirstandhasthevalue*timer1. This forces the compiler to insert an extra load instruction.
The same problem occurs if you use structure accesses rather than direct pointer access. The following code also compiles inefficiently:

```
typedef struct {int step;} State;
typedef struct {int timer1, timer2;} Timers;

void timers_v2(State *state, Timers *timers)
{
timers->timer1 += state->step; timers-
>timer2 += state->step;
}
```

The compiler evaluates state->step twice in case state->step and timers->timer1 are at the same memory address. The fix is easy: Create a new local variable to hold the value of state->step so the compiler only performs a single load.

EXAMPLE **Inthecodefortimers_v3weusealocalvariablesteptoholdthevalueofstate->step. Nowthe compilerdoesnotneed toworrythatstatemayaliaswith timers.**

```
void timers_v3(State *state, Timers *timers)
{
```

```
int step = state->step;
timers->timer1 += step; timers->timer2
+= step;
}
```

You must also be careful of other, less obvious situations where aliasing may occur. When you call another function, thisfunction mayalterthestateofmemoryandsochange the values of any expressions involving memory reads. The compiler will evaluate the expressions again. For example suppose you read state->step, call a function and then read state->step again. The compiler must assume that the function could change the value of state->step in memory.Thereforeitwillperformtworeads,ratherthanreusing the first value it read for state->step.

Another pitfall is to take the address of a local variable. Once you do this, the variable is referenced by a pointer and so aliasing can occur with otherpointers. The compiler is likely to keep reading the variable from the stack in case aliasing occurs. Consider the following example, which reads and then checksums a data packet:

```
int checksum_next_packet(void)
{
int *data; int N, sum=0;
data =
get_next_packet(&N);

do
{
sum += *(data++);
} while (--N);

return sum;
}
```

Here get_next_packetis a function returning the address and size of the next data packet. The previous code compiles to

```
checksum_next_packet
        STMFD   r13!,{r4,r14}   ; save r4, lr on the stack
        SUB     r13,r13,#8      create two stacked variables
        ADD     r0,r13,#4       ; r0 = &N, N stacked
```

```
                MOV     r4,#0           ; sum = 0
                BL      get_next_packet ; r0 = data
checksum_loop
                LDR     r1,[r0],#4      ; r1 = *(data++)
                ADD     r4,r1,r4        ; sum += r1
                LDR     r1,[r13,#4]     ; r1 = N (read from stack)
                SUBS    r1,r1,#1        ; r1-- & set flags
                STR     r1,[r13,#4]     ; N = r1 (write to stack)
                BNE     checksum_loop   ; if (N!=0) goto loop
                MOV     r0,r4           ; r0 = sum
                ADD     r13,r13,#8      ; delete stacked variables
                LDMFD   r13!,{r4,pc}    ; return r0
```

Note how the compiler reads and writes N from the stack for every N--. Once you take the address of N and pass it to get_next_packet, the compiler needs to worry about aliasing because the pointers data and &N may alias. To avoid this, don't take the address of local variables. If you must do this, then copy the value into another local variable before use.

You may wonder why the compiler makes room for two stacked variables when it only uses one. This is to keep the stack eight-byte aligned, which is required for LDRD instructions available in ARMv5TE. The example above doesn't actually use an LDRD, but the compiler does not know whether get_next_packet will use this instruction.

SUMMARY Avoiding Pointer Aliasing

- **Do not rely on the compiler to eliminate common subexpressions involving memory accesses. Instead create new local variables to hold the expression. This ensures the expression is evaluated only once.**
- **Avoid taking the address of local variables. The variable may be inefficient to access from then on.**

STRUCTURE ARRANGEMENT

The way you lay out a frequently used structure can havea significant impact onits performanceandcodedensity.Therearetwo issuesconcerningstructuresontheARM:alignment ofthestructureentriesandtheoverallsizeofthestructure.

For architectures up to and including ARMv5TE, load and store instructions are only guaranteed to load and store values with address aligned to the size of the access width. Table 5.4 summarizes these restrictions.

For this reason, ARM compilers will automatically align the start address ofa structure toamultiple ofthelargestaccesswidth used within the structure (usually four or eight bytes) and alignentries within structures to their access width by inserting padding.

For example, consider the structure

```
struct { char a; int
b; char c; short d;
}
```

For a little-endian memory system the compiler will lay this out adding padding to ensure that the next object is aligned to the sizeofthatobject:

Address	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

Table5.4 Load and store alignment restrictions for ARMv5TE.

Transfer size	Instruction	Byte address
1 byte	LDRB, LDRSB, STRB	any byte address alignment
2 bytes	LDRH, LDRSH, STRH	multiple of 2 bytes
4 bytes	LDR, STR	multiple of4 bytes
8 bytes	LDRD, STRD	multiple of 8 bytes

To improve the memory usage, you should reorder the elements

```
struct { char a; char c;
short d; int b;
}
```

This reduces the structure size from 12 bytes to 8 bytes, with the following new layout:

Address	**+3**	**+2**	**+1**	**+0**
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

Therefore, it is a good idea to group structure elements of the same size, so that the structure layout doesn't contain unnecessary padding. The *armcc* compiler does include akeyword packedthatremovesallpadding.Forexample,the structure

```
__packed struct { char
a; int b; char c; short d;
}
```

will be laid out in memory as

Address	**+3**	**+2**	**+1**	**+0**
+0	b[23,16]	b[15,8]	b[7,0]	a
+4	d[15,8]	d[7,0]	c	b[31,24]

Instructions	**Offset available from the base register**
LDRB, LDRSB, STRB	**0 to 31 bytes**
LDRH, LDRSH, STRH	**0 to 31 halfwords (0 to 62 bytes)**
LDR, STR	**0 to 31 words (0 to 124 bytes)**

```
void dostageA(void); void
dostageB(void); void dostageC(void);

typedef struct {
unsigned int stageA : 1; unsigned int stageB : 1;
unsigned int stageC : 1;
} Stages_v1;

void dostages_v1(Stages_v1 *stages)
{
if (stages->stageA)
{
dostageA();
}


if (stages->stageB)
{
dostageB();
}
if (stages->stageC)
{
dostageC();
}
}
```

Here,weusethreebit-fieldflagstoenablethreepossiblestagesofprocessing. Theexample compiles to

```
dostages_v1
        'T MF D   r13!,{r4,r14}     stack r4, lr
         M OV     r4,r0              move stages to r4


          LDR     r0,[r0,#0]         r0 = stages bitfield

          TST     r0,#1              if (stages->stageA)
          BL      dostageA           {dostageA();}
          NE
          LDR     r0,[r4,#0]         r0 = stages bitfield

          MOV     r0,r0,LSL          shift bit 1 to bit 31
                  #30
          CMP     r0,#0              if (bit31)
```

```
BLL T     dostageB          {dostageB();}

LD R      r0,[r4,#0]        r0 = stages bitfield

MOV       r0,r0,LSL         shift bit 2 to bit 31
          #29
CMP       r0,#0             if (!bit31)

LDML      r13!,{r4,r14}     return
TFD


BLT       dostageC          dostageC();
LD        r13!,{r4,pc}      return
MFD
```

Note that the compiler accesses the memory location containing the bit-field three times. Because the bit-field is stored in memory, the dostage functions could change the value. Also, the compiler uses two instructions to test bit 1 and bit 2 of the bit-field, rather than a single instruction.
You can generate far more efficient code by using an integer rather than a bit-field. Use enumor #define masks to divide the integer type into different fields.

EXAMPLE The following code implements the dostages function using logical operations rather than bit-fields:

```
typedef unsigned long
Stages_v2; #define STAGEA (1ul
<<0)


#define STAGEB (1ul << 1)#define
STAGEC (1ul<<2)

void dostages_v2(Stages_v2 *stages_v2)
{
Stages_v2 stages = *stages_v2;

if (stages & STAGEA)
{
dostageA();
}
if (stages & STAGEB)
{
dostageB();
}
if (stages & STAGEC)
{
dostageC();
}
}
```

Unaligned data and endianness are two issues that can complicate memory accesses and portability. Is the array pointer aligned? Is the ARM configured for a big-endian or little-endian memorysystem?

The ARM load and store instructions assume that the address is a multiple of the type youareloadingorstoring.Ifyou loadorstoretoanaddressthatisnotalignedtoits type, then the behavior depends on the particular implementation. Thecoremaygenerateadata abort or load a rotated value. For well-written, portable code you should avoid unaligned accesses.

C compilers assume that a pointer is aligned unless you say otherwise. If a pointer isn't aligned, thentheprogrammay giveunexpectedresults.Thisissometimesanissuewhenyou are porting code to the ARM from processors that do allow unaligned accesses. For *armcc*, the_packeddirective tells the compiler that a data item can be positioned at any byte alignment.Thisisusefulforportingcode,butusing packedwillimpactperformance.

To illustrate this, look at the following simple routine, readint. It returns the integer at the address pointed to by data. We'veused_packedtotellthecompiler thatthe integer may possibly not be aligned.

```
int readint(_packed int *data)
{
return *data;
}
```

This compiles to

```
readint
        B IC    r3,r0,#3        ; r3 = data & 0xFFFFFFFC
        AND     r0,r0,#3        ; r0 = data & 0x00000003
        M O V   r0,r0,LSL #3    ; r0 = bit offset of data word
        D M     r3,{r3,r12}     ; r3, r12 = 8 bytes read from r3
        O       r3,r3,LSR r0    ; These three instructions
        S B     r0,r0,#0x20     ; shift the 64 bit value r12.r3
        R R     r0,r3,r12,LSL   ; right by r0 bits
                r0
        M O V   pc,r14          ; return r0
```

Table5.6 Little-endian configuration.

Instruction	Width (bits)	b31..b24	b23..b16	b15..b8	0

LDRB	8	0	0	0	B (A)
LDRS B	8	S(A)	S(A)	S(A)	B (A)
STRB	8	X	X	X	B (A)
LDRH	16	0	0	B(A+)	B (A)
LDRS H	16	S(A +1)	S(A +1)	B(A +1)	B (A)
STRH	16	X	X	B(A +1)	B (A)
LDR/ STR	32	B(A +3)	B(A +2)	B(A +1)	B (A

EXAMPLE

Thesefunctionsread a32-bitintegerfromabytestreampointedtobydata. Thebytestream contains little- or big endian data, respectively. These functions are independent of the ARM memory system byte order since they only use byte accesses.

```
int readint_little(char *data)
                        {
int a0,a1,a2,a3;

a0      =      *(data++);          a1      =
*(data++); a2 = *(data++); a3
= *(data++);
return a0 | (a1<<8) | (a2<<16) | (a3<<24);
}

int readint_big(char *data)

{
int a0,a1,a2,a3;



a0 = *(data++); a1 = *(data++); a2
= *(data++); a3 = *(data++);
return (((((a0<<8) | a1)<<8) | a2)<<8) | a3;
}
```

If speed is critical, then the fastest approach is to write several variants of the critical routine. Foreachpossiblealignment andARMendiannessconfiguration,youcallaseparate routine optimized for that situation.

EXAMPLE **The read_samplesroutine takes an array of N16-bit sound samples at address in. The sound samples are little-endian (for example from a.wavfile) and can be at any byte alignment. The routine copiesthe samples to an aligned array of shorttype valuespointed tobyout.**

Thesampleswill bestoredaccordingtotheconfiguredARMmemoryendianness. The routine handles all cases in an efficient manner, regardless of input alignment andofARM endianness configuration.

```
void read_samples(short *out, char *in, unsigned int N)
{
unsigned short *data; /* aligned input pointer */ unsigned int sample, next;

switch ((unsigned int)in & 1)
{
case 0: /* the input pointer is aligned */ data = (unsigned  short
*
)
i
n
;

d
o
{
sample = *(data++); #ifdef
BIG_ENDIAN sample = (sample >> 8) |
(sample<<8); #endif
                                        *(out++) = (short)sample;
                                        }  while
                                        (--N);
                                        break;
case 1: /* the input pointer is not aligned */ data = (unsigned short
*)(in-1);
sample = *(data++);

#ifdef_BIG_ENDIAN
sample = sample & 0xFF; /* get first byte of sample */
#else #endif
                sample = sample >> 8; /* get first byte of sample */



#ifdef_BIG_ENDIAN
do
{
next = *(data++);
/* complete one sample and start the next */


out++ = (short)((next & 0xFF00) | sample); sample = next & 0xFF;
#else
#endif


}
*out++ = (short)((next << 8) | sample); sample = next>>8;


}
}
```

The routine works by having different code for each endianness and alignment. Endianness isdealtwithatcompiletimeusingthe_BIG_ENDIANcompilerflag.Alignment must be dealt with at run time using the switchstatement.
You can make the routine even more efficient by using 32-bit reads and writes rather than 16-bit reads and writes, whichleadstofourelementsintheswitchstatement,onefor eachpossibleaddressalignmentmodulofour.

- Avoid using unaligned data if you can.
- Use the type char* for data that can be at any byte alignment. Access the data by reading bytes and combining with logical operations. Then the code won't depend on alignment or ARM endianness configuration.
- For fast access to unaligned structures, write different variants according to pointer alignment and processor endianness.

DIVISION

The ARM does not have a divide instruction in hardware. Instead the compiler implements divisions by calling softwareroutinesintheClibrary.Therearemanydifferenttypesofdivision routine thatyou can tailor to a specific range of numerator and denominator values. We look at assembly division routines in detail in Chapter 7. The standard integer division routine provided in the C library can take between 20 and 100 cycles, depending on implementation,earlytermination,andtherangesoftheinputoperands.
Division and modulus (/ and %) are such slow operations that you should avoid them as much as possible. However, division by a constant and repeated division by the same denominator can be handled efficiently. This section describes how to replacecertain divisionsbymultiplicationsandhowtominimizethenumberofdivisioncalls.
Circular buffers are one area where programmers often use division, but you can avoid these divisions completely. Suppose you have a circular buffer of size buffer_size bytes and a position indicated by a buffer offset. To advance the offset by incrementbytesyou could write

```
offset = (offset + increment) % buffer_size;
```

Instead it is far more efficient to write

```
offset += increment;
if (offset>=buffer_size)
{
offset -= buffer_size;
}
```

The first version may take 50 cycles; the second will take 3 cycles because it does not involve a division. We've assumed that increment < buffer_size; you can always arrange this in practice.
If you can't avoid a division, then try to arrange that the numerator and denominator are unsigned integers. Signed division routines are slower since they take the absolute values of the numerator and denominator and then call the unsigned division routine. They fix the sign of the result afterwards.

Many C library division routines return the quotient and remainder from the division. In other words a free remainder operation is available to you with each division operation and vice versa. For example, to find the (x, y) position of a locationatoffsetbytes into a screenbuffer, it is tempting towrite

```
typedef struct { int
x; int y;
} point;

point getxy_v1(unsigned int offset, unsigned int bytes_per_line)
{
point p;
p.y = offset / bytes_per_line;
```

```
p.x = offset - p.y * bytes_per_line; return p;
}
```

It appears that we have saved a division by using a subtract and multiply to calculate p.x, butin fact, it is often more efficienttowritethefunctionwiththemodulusorremainder operation.

EXAMPLE

Ingetxy_v2,thequotientandremainderoperationonlyrequireasinglecalltoadivision routine:

```
point getxy_v2(unsigned int offset, unsigned int bytes_per_line)
{
point p;

p.x = offset % bytes_per_line;
p.y = offset / bytes_per_line; return p;
}
```

There is only one division call here, as you can see in the following compiler output. In fact, this version is four instructions shorter than getxy_v1. Note that this may not be the case for all compilersandClibraries.

```
getxy_v2
        STMFD   r13!,{r4, r14}   ; stack r4, lr
        MOV     r4,r0            ; move p to r4
        MOV     r0,r2            ; r0 = bytes_per_line
        BL      _rt_udiv         ; (r0,r1) = (r1/r0, r1%r0)
        STR     r0,[r4,#4]       ; p.y = offset / bytes_per_line
        STR     r1,[r4,#0]       ; p.x = offset % bytes_per_line
```

REPEATED UNSIGNED Division WITH REMAINDER

Often the same denominator occurs several times in code. In the previous example, bytes_per_line will probably be fixed throughout the program. If we project from three to two cartesian coordinates, then we use thedenominatortwice:

$$(x, y, z) \rightarrow (x/z, y/z)$$

In these situations it is more efficient to cache the value of 1/zin some way and use a multiplication by 1/zinstead of a division. We will show how to do this in the next subsection.

Wealsowanttosticktointegerarithmeticandavoidfloatingpoint(seeSection5.11).The next description is rather mathematical and covers the theory behind this con- version of repeated divisions into multiplications. If you are not interested in the theory, then don't worry. You can jump directly to Example 5.13, which follows.

UNIT-III

RTOS Fundamentals and Programming

Introduction -Operating system (OS): An Operating system (OS) is a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create, print, copy, delete and display files, read data from files ,write data to files , control the I/O operations, allocate memory locations and process the interrupts etc. It provides the users an interface to the hardware resources. In a multiuser system it allows several users to share the CPU time, share the other system resources and provide inter task communication, Timers, clocks, memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

So, the Operating system can also be defined as a collection of system calls or functions which provide an interface between hardware and application program.

It manages the hardware resources of a computer and hosting applications that run on the computer. Hence it is also called a resource Manager.

An OS typically provides multitasking, synchronization, Interrupt and Event Handling, Input/Output, Inter-task Communication, Timers and Clocks and Memory Management. The core of the OS is the Kernel which is typically a small, highly optimized set of libraries.

The Kernel is a program that constitutes the central core of an operating system. It has complete control over everything that occurs in the system. The Kernel is the first part of the operating system to load into memory during booting (i.e., system startup), and it remains there for the entire duration of the session because its services are required continuously.

The kernel provides basic services for all other parts of the operating system, typically including memory management, process management, file management and I/O (input/output) management (i.e., accessing the peripheral devices). These services are requested by other parts of the operating system or by application programs through a specified set of program interfaces referred to as system calls.

Popular Operating Systems: Windows (from Microsoft), MacOS, MS-Dos, Linux(Open source), Unix (Multi user-Bell Labs), Xenix (Microsoft), Android (Mobile).

Types of operating systems:

An Operating system (OS) is nothing but a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create ,print ,copy , delete and display files , read data from files ,write data to files ,control the I/O operations , allocate memory locations and process the interrupts etc. It provides the users an interface to the hardware resources. In a multiuser system it allows several users to share the CPU time, share the other system resources and provide inter task communication, Timers, clocks, memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

There are three important types of operating systems .They are (i).Embedded Operating System (ii). Real time operating system and (iii).Handheld operating system.

(i).Embedded Operating System

The operating system used for embedded computer systems is known as embedded operating system. These operating systems are designed to be compact, efficient, and reliable.

The embedded operating system uses a preemptive priority based kernel. But this kernel do not meet the strict deadlines. By removing the unnecessary components from the kernel of desktop operating system, the embedded operating can be obtained. This OS occupies less memory space.

The popularly known embedded operating systems are (a).Embedded NT (b) Windows XP Embedded (c) Embedded Linux

The Embedded NT for its minimal operation without any network support occupies nearly 9MB of RAM and 8 MB of Flash .It is a preemptive, multitasking operating system. Generally Embedded NT is preferred to other OSs because of its ease in developing the applications. It is suitable for embedded systems built around single board computers for applications, like Internet Kiosks, Automatic Teller Machines (ATM) etc..

Microsoft Windows XP Embedded is the successor to Embedded NT. It is also pre- emptive multitasking operating system like Embedded NT. This OS is widely used in set top boxes, point of sale terminals and Internet Kiosks etc.

Embedded Linux is a open source software and it is covered by GNU General Public License (GPL) and hence the complete source code is available at free of cost. The important features of

Embedded Linux are POSIX support and availability of large software resources.

Embedded Linux is used in embedded computer systems such as mobile phones, personal digital assistants, media players, set-top boxes, and other consumer electronics devices, networking equipment, machine control, industrial automation, navigation equipment and medical instruments.

Real-Time Operating System:

A real-time operating system (RTOS) is an operating system (OS) intended to serve the real time application requests .A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task .

A hard real-time operating system has less jitter than a soft real-time operating system. The main objective is not the high throughput, but a guarantee of meeting the deadlines. An RTOS that can usually or generally meet a deadline is a soft real- time OS, but if it can meet a deadline deterministically it is a hard real-time OS.

A real-time OS has an advanced algorithm for scheduling Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency, but a real- time OS is valued more for how quickly or how predictably it can respond.

There are various Real-Time operating systems both commercial and open source in the market

(i).QNX Neutrino (ii)VxWorks (iii) microC/OS-II (iv).RTLinux.

QNX Neutrino is a real time operating system from QNX Software systems limited . It is supported by ARM, MIPS, Power PC, Strong ARM,X86 and Pentium.

This OS supports multiple scheduling algorithms and up to 65535 tasks and can create embedded data base applications.

microC/OS-II is a real time operating system used mainly in academic institutions. It is available in source code form for non-commercial applications. This do not support the Round Robin scheduling algorithm.

RT Linux is a hard real time RTOS microkernel that runs the entire Linux operating system as a fully preemptive process. It was commercialized at FSM Labs. RT Linux runs underneath the Linux OS. The Linux is an idle task for RT Linux. The real-time software running under RT Linux

is given priority as compared to non-real- time threads running under Linux. This OS is an excellent choice for 32-bitprocessor based embedded systems.

REAL TIME SYSTEMS: Real-time systems are those systems in which the correctness of the system depends not only on the Output, but also on the time at which the results are produced (Time constraints must be strictly followed).

Real time systems are two types. (i) Soft real time systems and (ii) Hard real time systems. A Soft real time system is one in which the performance of the system is only degraded but, not destroyed if the timing deadlines are not met.

For Ex: Air conditioner, TV remote or music player, Bus reservation ,automated teller machine in a bank , A Lift etc.

A hard Real time system is one in which the failure to meet the time dead lines may lead to a complete catastrophe or damage to the system.

For Ex: Air navigation system, Nuclear power plant , Failure of car brakes , Gas leakage system ,RADAR operation ,Air traffic control system etc.

Typical Real Time Applications: Real Time systems find applications in various fields of science and technology. The prominent applications are (i) Digital Control (ii) command and control, (iii) Signal processing (iv) Telecommunication systems and (v) Defense etc.

Examples:

- In automobile engineering, the real time systems control the engine and brakes of the vehicle and regulate traffic lights for smooth travel.
- In air craft monitoring, the real time systems schedule and monitor the takeoff and landing of the planes, make it fly, maintain the flight path, and avoid accidents.
- The real time patient care system monitor and regulate the blood pressure and heart beats of the patient and also, they can entertain people with electronic games, TV and music.
- The real time systems are found in Air Traffic Control system also. The Air Traffic Control (ATC) system regulates the flow of flights to each destination airport. It does so by assigning to each aircraft an arrival time and en route to the destination

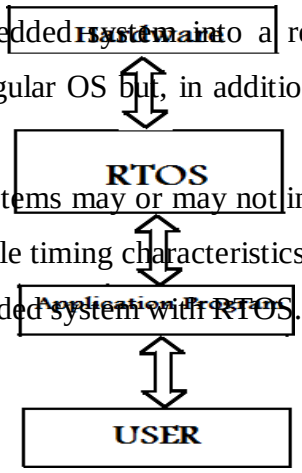
- The real time systems are important in industries also. For example a system of robots perform assembly tasks and repairs in a factory or chemical industries where human beings cannot enter.

- An avionics system for a military aircraft, the real time systems perform the tracking and ballistic computations and coordinates the RADAR and weapon control systems.
- Digital filtering, video and voice compressing/decompression, and radar signal processing are the major applications of real time systems in signal processing.
- Another interesting application is the real-time database systems that refers to a diverse spectrum of information systems, ranging from stock price quotation systems, to track records databases, to real-time file systems.
- Real time systems are also found in Supervisory Control and Data Acquisition (SCADA). In SCADA systems the sensors are placed at different geographical points to collect the raw data and this data are processed and stored in a Real time data base.
- Robots used in nuclear power stations, to handle the radioactive material and other dangerous materials.
- Real time system applications are also found in office automation where
- LASER printers and FAX machines are used.

REAL TIME OPERATING SYSTEM (RTOS): It is an operating system that supports real-time applications by providing logically correct result within the deadline set by the user. A real time operating system makes the embedded system into a real time embedded system. The basic structure of RTOS is similar to regular OS but, in addition, it provides mechanisms to allow real time scheduling of tasks.

Though the real-time operating systems may or may not increase the speed of execution, but they provide more precise and predictable timing characteristics than general-purpose OS.

The figure below shows the embedded system with RTOS.



All the embedded systems are not designed with RTOS. Low end application systems do not require the RTOS but only High end application oriented embedded systems which require scheduling alone need the RTOS.

For example an embedded system which measures Temperature or Humidity etc. do not require any operating system. Whereas a Mobile phone , RADAR or Satellite system used for high end applications require an operating system.

Popular Real-Time Operating Systems:

RTOS	Applications/Features
Windows CE (Microsoft Widows)	Used small foot print mobile and connected devices Supported by ARM,MIPS, SH4 & x86 architectures
LynxOS	Complex, hard real-time applications
	·POSIX- compatible, multiprocess, multithreaded OS. ·Supported by x86, ARM, PowerPC architectures
VxWorks (Wind river)	· Most widely adopted RTOS in the embedded industry. · Used in famous NASA rover robots Spirit and Opportunity ·Certified by several agencies and international standards for real time systems, reliability and security-critical applications.
Micrium µC/OS-II	· Ported to more than a hundred architectures including x86, mainly used in microcontrollers with low resources. · Certified by rigorous standards, such as RTCADO-178B
QNX	· Most traditional RTOS in the market. · Microkernel architecture; completely compatible with the POSIX · Certified by FAADO-278 and MIL-STD-1553 standards.
Symbian	Designed for Smartphones Supported by ARM, x86 architecture

VRTX	· Suitable for traditional board based embedded systems and SoC architectures ·Supported by ARM, MIPS, PowerPC & other RISC architectures
RTLINUX	Open source

Differences between RTOS and General purpose OS:

The key difference between general-computing operating systems and real-time operating systems is the "deterministic " timing behavior in the real- time operating systems. "Deterministic" timing means that OS consume only known and expected amounts of time. RTOS have their worst case latency defined. Latency is not of a concern for General Purpose OS.

Task Scheduling: General purpose operating systems are optimized to run a variety of applications and processes simultaneously, thereby ensuring that all tasks receive at least some processing time. As a consequence, low-priority tasks may have their priority boosted above other higher priority tasks, which the designer may not want. However, RTOS uses priority-based preemptive scheduling, which allows high- priority threads to meet their deadlines consistently. All system calls are deterministic, implying time bounded operation for all operations and ISRs. This is important for embedded systems where delay could cause a safety hazard. The scheduling in RTOS is time based. In case of General purpose OS, like Windows/Linux, scheduling is process based.

- **Preemptive kernel** - In RTOS, all kernel operations are pre-emptiable
- **Priority Inversion** - RTOS have mechanisms to prevent priority inversion
- **Usage** - RTOS are typically used for embedded applications, while General Purpose OS are used for Desktop PCs or other generally purpose PCs.

Note: **Jitter**: The Timing error of a task over subsequent iterations of a program or loop is referred to as jitter. RTOS are optimized to minimize jitter.

There are four broad categories of kernels.

i. **Monolithic** kernels: provide rich and powerful abstractions of the underlying hardware.

ii. **Microkernels** provide a small set of simple hardware abstractions and use applications called servers to provide more functionality

iii. **Hybrid** (modified Micro kernels) Kernels are much like pure Microkernels, except that they include some additional code in kernel space to increase performance.

iv. **Exo-kernels** provide minimal abstractions, allowing low-level hardware access. In Exo-kernel systems, library operating systems provide the abstractions typically present in monolithic kernels.

Pre-Emptive and Non-Pre-Emptive: In a normal operating system ,if a task is running ,it will continue to run until its completion .It cannot be stopped by the OS in the middle due to any reason

.Such concept is known as non-preemptive.

In real time OS, a running task can be stopped due to a high priority task at any time with-out the willing of present running task. This is known as pre-emptiveness.

So, Preemptive scheduling involves scheduling based on the highest priority. The highest priority will always be given chance. Non-preemptive scheduling is a process is not interrupted once started until it is finished.

Initialization of RTOS:

RTOS is initialized using the following code. Void main(void)

{

Init RTOS( ); /*Initialize the RTOS*/ Start task (v respond to Button, High _priority); Start task (v calculate task levels , low_priority); Start_RTOS ( ); /*start RTOS*/

}

Architecture of the RTOS:

The heart or nucleus of any RTOS is the kernel. Inside the kernel is the scheduler. It is basically a set of algorithms which manage the task running order. Multitasking definition comes from the ability of the kernel to control multiple tasks that must run within time deadlines. Multitasking may give the impression that multiple threads are running concurrently, as a matter of fact the processer runs task by task, according to the task scheduling.

General Architecture of RTOS Architecture of the Kernel

The kernel is the core of an operating system. It is a piece of software responsible for providing secure access to the system's hardware and to running the programs. Kernel is common to every operating system either a real time or non-real time .The major difference lies in its architecture .Since there are many programs, and hardware access is limited, the kernel also decides when and how long a program should run. This is called scheduling. Kernels has various functions such as file management, data transfer between the file system, hardware management, memory

management and also the control of CPU time. The kernel also handles the Interrupts.

Kernel Objects: The various kernel objects are Tasks, Task Scheduler, Interrupt Service Routines, Semaphores, Mutexes, Mailboxes, Message Queues, Pipes, Event Registers, Signals and Timers **(i).Task:**

A task is a basic unit or atomic unit of execution that can be scheduled by an RTOS to use the system resources like CPU, Memory, I/O devices etc. It starts with reading of the input data and of the internal state of the task, and terminates with the production of the results and updating the internal state. The control signal that initiates the execution of a task is provided by the operating system.

There are two types of tasks. (i)Simple Task(S-Task) and (ii) Complex Task(C-Task).

Simple Task (S-task): A simple task is one which has no synchronization point i.e., whenever an S -task is started, it continues until its termination point is reached. Because an S-task cannot be blocked within the body of the task the execution time of an S-task is not directly dependent on the progress of the other tasks in the node. S- task is mainly used for single user systems.

Complex Task (C-Task): A task is called a complex task (C-Task) if it contains a blocking synchronization statement (e.g., a semaphore operation "wait") within the task body. Such a "wait" operation may be required because the task must wait until a condition outside the task is satisfied, e.g., until another task has finished updating a common data structure, or until input from a terminal has arrived.

Task States:

At any instant of time a task can be in one of the following states:

(i) Dormant (ii). Ready (iii). Running and (iv).Blocked.

When a task is first created, it is in the dormant task. When it is added to RTOS for scheduling, it is a ready task. If the input or a resource is not available, the task gets blocked.

If no task is ready to run and all of the tasks are blocked, the RTOS will usually run the Idle Task. An Idle Task does nothing .The idle task has the lowest priority.

void Idle task(void)

```
{
While(1);
}
```

Creation of a Task:

A task is characterized by the parameters like task name , its priority , stack size and operating system options .To create a task these parameters must be specified .A simple program to create a task is given below.

```
result = task-create("Tx Task",  100,0x4000,OS_Pre-emptiable);        /*task create*/ if (result =
= os_success)
{                          /*task successfully created*/
}
```

Task Scheduler:

Task scheduler is one of the important component of the Kernel .Basically it is a set of algorithms that manage the multiple tasks in an embedded system. The various tasks are handled by the scheduler in an orderly manner.

This produces the effect of simple multitasking with a single processor. The advantage of using a scheduler is the ease of implementing the sleep mode in microcontrollers which will reduce the power consumption considerably (from mA to µA). This is important in battery operated embedded systems.

The task scheduler establishes task time slots. Time slot width and activation depends on the available resources and priorities.

A scheduler decides which task will run next in a multitasking system. Every RTOS provides three specific functions.

(i).Scheduling (ii) Dispatching and (iii). Inter-process communication and synchronization.

e scheduling determines ,which task ,will run next in a multitasking system and the dispatches perform the necessary book keeping to start the task and Inter-process communication and synchronization assumes that each task cooperate with others.

Scheduling Algorithms: In Multitasking system to schedule the various tasks, different scheduling algorithms are used. They are (a).First in First out (b).Round Robin algorithm (c).Round Robin with priority (d) Non-preemptive (e)Pre-emptive.

In FIFO scheduling algorithm, the tasks which are ready-to-run are kept in a queue and the CPU serves the tasks on first-come-first served basis.

In Round-Robin Algorithm the kernel allocates a certain amount of time for each task waiting in the queue. For example, if three tasks 1, 2 and 3 are waiting in the queue, the CPU first executes task1 then task2 then task3 and then again task1.

The round-robin algorithm can be slightly modified by assigning priority levels to the tasks. A high priority task can interrupt the CPU so that it can be executed. This scheduling algorithm can meet the desired response time for a high priority task. This is the Round Robin with priority.

In Shortest-Job First scheduling algorithm, the task that will take minimum time to be executed will be given priority. The disadvantage of this is that as this approach satisfies the maximum number of tasks, some tasks may have to wait forever.

In preemptive multitasking, the highest priority task is always executed by the CPU, by preempting the lower priority task. All real-time operating systems implement this scheduling algorithm.

The various function calls provided by the OS API for task management are given below.

- Create a task
- Delete a task
- Suspend a task
- Resume a task
- Change priority of a task
- Query a task

Process or Task:

Embedded program (a static entity) = a collection of firmware modules. When a firmware module is executing, it is called a process or task . A task is usually implemented in C by writing a function. A task or process simply identifies a job that is to be done within an embedded application.

When a process is created, it is allocated a number of resources by the OS, which may include: – Process stack – Memory address space – Registers (through the CPU) – A program counter (PC) – I/O ports, network connections, file descriptors, etc.

Threads: A process or task is characterized by a collection of resources that are utilized to execute a program. The smallest subset of these resources (a copy of the CPU registers including the PC and a stack) that is necessary for the execution of the program is called a thread. A thread is a unit of computation with code and context, but no private data.

Multitasking:

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications. Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilization of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power, silicon area and other constraints. In a multi-tasking operating system it is assumed that the various tasks are to cooperate to serve the requirements of the overall system. Co-operation will require that the tasks communicate with each other and share common data in an orderly an disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed.

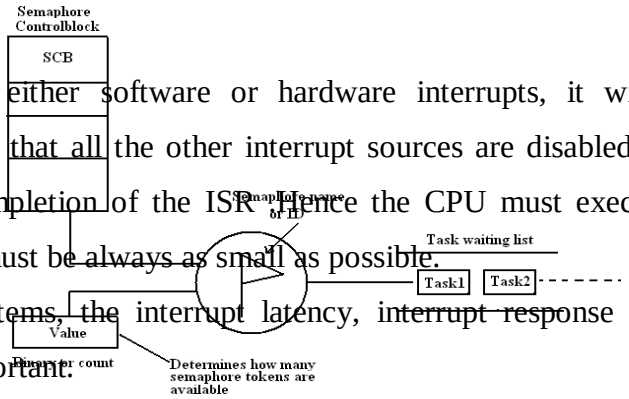
Interrupt Service Routines:

An interrupt service routine (ISR), also known as an interrupt handler, is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt.
In a real-time embedded system, there are two possible interrupts. One is the Hardware Interrupt and the other is the software Interrupt.
Hardware Interrupts are asynchronous interrupts which are triggered by an electric pulse, whereas software interrupts are synchronous interrupts and these are triggered by a command or instruction. In hardware driven scheduling, mostly timers, keyboard devices, I/O ports will take part.

ISR is a small program, which is executed to develop an interface between the user and the hardware. The CPU will execute the ISR subroutine when it receives either a hardware or software interrupt.

The synchronization mechanism cannot be used in an ISR, because it is not possible in an ISR to wait indefinitely for a resource to be available.

The faster the ISR can do its job, the better the real time performance of the RTOS. Hence the ISR should be always as small as possible.



When the CPU receives either software or hardware interrupts, it will try to execute the corresponding ISR. Before that all the other interrupt sources are disabled and the interrupts are enabled only after the completion of the ISR. Hence the CPU must execute the ISR as fast as possible and also the ISR must be always as small as possible.

In real-time operating systems, the interrupt latency, interrupt response time and the interrupt recovery time are very important.

Interrupt Latency: It is the time between the generation of an interrupt by a device and the servicing of the device which generated the interrupt.

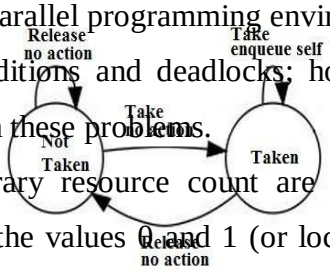
For many operating systems, devices are serviced as soon as the device's interrupt handler is executed. Interrupt latency may be affected by interrupt controllers, interrupt masking, and the operating system's (OS) interrupt handling methods.

Interrupt Response Time: Time between receipt of interrupt signal and starting the code that handles the interrupt is called interrupt response time.

Interrupt Recovery Time: Time required for CPU to return to the interrupted code/highest priority task is called interrupt recovery time.

Semaphores:

A semaphore is nothing but a value or variable or data which can control the allocation of a resource among different tasks in a parallel programming environment. So, Semaphores are a useful tool in the prevention of race conditions and deadlocks; however, their use is by no means a guarantee that a program is free from these problems.



Semaphores which allow an arbitrary resource count are called counting semaphores, whilst semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

The operation of a semaphore can be understood from the following diagram.

Types of Semaphores: There are three types of semaphores

1. Binary Semaphores,
2. Counting Semaphores and
3. Mutexes.

A binary semaphore is a synchronization object that can have only two states 0 or 1.

Take: Taking a binary semaphore brings it in the "taken" state, trying to take a semaphore that is already taken enters the invoking thread into a waiting queue.

Release: Releasing a binary semaphore brings it in the "not taken" state if there are not queued threads. If there are queued threads then a thread is removed from the queue and resumed, the binary semaphore remains in the "taken" state. Releasing a semaphore that is already in its "not taken" state has no effect.

Binary semaphores have no ownership attribute and can be released by any thread or interrupt handler regardless of who performed the last take operation. Because of this binary semaphores are often used to synchronize threads with external events implemented as ISRs, for example waiting for a packet from a network or waiting that a button is pressed. Because there is no ownership concept a binary semaphore object can be created to be either in the "taken" or "not taken" state initially.

Counting Semaphores:

A counting semaphore is a synchronization object that can have an arbitrarily large number of states. The internal state is defined by a signed integer variable, the counter.

The counter value (N) has a precise meaning: The Negative value indicates that, there are exactly -N threads queued on the semaphore.

The Zero value indicates that no waiting threads, a wait operation would put in queue the invoking thread.

The Positive value indicates that no waiting threads, a wait operation would not put in queue the invoking thread.

Two operations are defined for counting the semaphores.

Wait: This operation decreases the semaphore counter .If the result is negative then the invoking thread is queued.

Signal: This operation increases the semaphore counter .If the result is nonnegative then a waiting thread is removed from the queue and resumed.

Counting semaphores have no ownership attribute and can be signaled by any thread or interrupt handler regardless of who performed the last wait operation .Because there is no ownership concept a counting semaphore object can be created with any initial counter value as long it is non-negative. The counting semaphores are usually used as guards of resources available in a discrete quantity. For example the counter may represent the number of used slots into a circular queue, producer threads would "signal" the semaphores when inserting items in the queue, consumer threads would "wait" for an item to appear in queue, this would ensure that no consumer would be able to fetch an item from the queue if there are no items available.

The OS function calls provided for Semaphore management are

- Create a semaphore
- Delete a semaphore
- Acquire a semaphore
- Release a semaphore
- Query a semaphore

Mutexes:

Mutex means mutual exclusion A mutex is a synchronization object that can have only two states. They are not-owned and owned. Two operations are defined for mutexes.

Lock: This operation attempts to take ownership of a mutex, if the mutex is already owned by another thread then the invoking thread is queued.

Unlock: This operation relinquishes ownership of a mutex. If there are queued threads then a thread is removed from the queue and resumed, ownership is implicitly assigned to the thread.

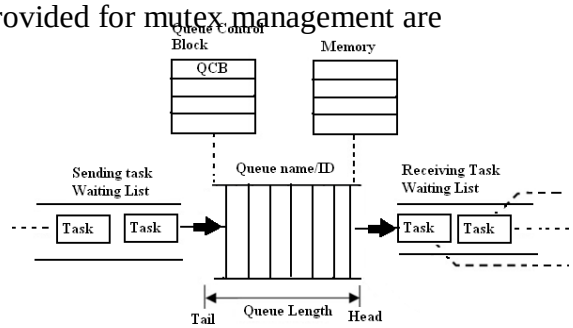
Mutex is basically a locking mechanism where a process locks a resource using mutex. As long as the process has mutex, no other process can use the same resource. (Mutual exclusion). Once process is done with resource, it releases the mutex. Here comes the concept of ownership. Mutex is locked and released by the same process/thread. It cannot happen that mutex is acquired by one process and released by other.

So, unlike semaphores, mutexes have owners. A mutex can be unlocked only by the thread that owns it. Most RTOSs implement this protocol in order to address the Priority Inversion problem.

Semaphores can also handle mutual exclusion problems but are best used as a communication mechanism between threads or between ISRs and threads.

The OS functions calls provided for mutex management are

- Create a mutex
- Delete a mutex
- Acquire a mutex
- Release a mutex
- Query a mutex
- Wait on a mutex



Difference between Mutex & Semaphore: Mutexes are typically used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).

Mailboxes:

One of the important Kernel services used to send the Messages to a task is the message mailbox. A Mailbox is basically a pointer size variable. Tasks or ISRs can deposit and receive messages (the pointer) through the mailbox.

A task looking for a message from an empty mailbox is blocked and placed on waiting list for a time (time out specified by the task) or until a message is received. When a message is sent to the

mail box, the highest priority task waiting for the message is given the message in priority-based mailbox or the first task to request the message is given the message in FIFO based mailbox.

The operation of a mailbox object is similar to our postal mailbox. When someone posts a message in our mailbox, we take out the message.

A task can have a mailbox into which others can post a mail. A task or ISR sends the message to the mailbox.

To manage the mailbox object, the following function calls are provided in the OS API:

- Create a mailbox
- Delete a mailbox
- Query a mailbox
- Post a message in a mailbox
- Read a message form a mailbox.

Message Queues:

The Message Queues, are used to send one or more messages to a task i.e. the message queues are used to establish the Inter task communication. Basically Queue is an array of mailboxes. Tasks and ISRs can send and receive messages to the Queue through services provided by the kernel. Extraction of messages from a queue follow FIFO or LIFO structure.

Applications of message queue are

- Taking the input from a keyboard
- To display output
- Reading voltages from sensors or transducers
- Data packet transmission in a network

In each of these applications, a task or an ISR deposits the message in the message queue. Other tasks can take the messages. Based on our application, the highest priority task or the first task waiting in the queue can take the message. At the time of creating a queue, the queue is given

name or ID, queue length, sending task waiting list and receiving task waiting list.

To use a message queue, first it must be created. The creation of a Queue return a queue ID. So, if any task wish to post some message to a task ,it should use its queue ID.

q_{id} = queue_create("MyQueue" , Queue_options) ; //*Queue name and OS specification options*//

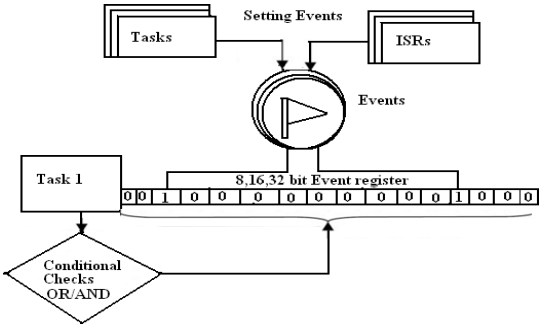
Each queue can be configured as a fixed size/variable size.

The following function calls are provided to manage message queues

- Create a queue
- Delete a queue
- Flush a queue
- Post a message in queue
- Post a message in front of queue
- Read message from queue
- Broadcast a message
- Show queue information
- Show queue waiting list.

Event Registers:

Some kernels provide a special register as part of each tasks control block .This register, called an event register. It consists of a group of binary event flags used to track the occurrence of specific events. Depending on a given kernel's implementation of this mechanism, an event register can be 8 or 16 or 32 bits wide, may be even more.



Each bit in the event register treated like a binary flag and can be either set or cleared.

Through the event register, a task can check for the presence of particular events that can control its execution. An external source, such as a task or an ISR, can set bits in the event register to inform

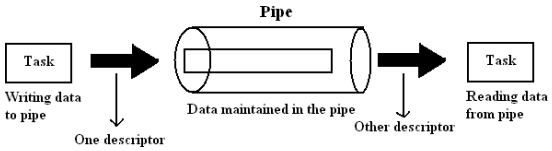
the task that a particular event has occurred.

For managing the event registers, the following function calls are provided:

- Create an event register
- Delete an event register
- Query an event register
- Set an event register
- Clear an event flag

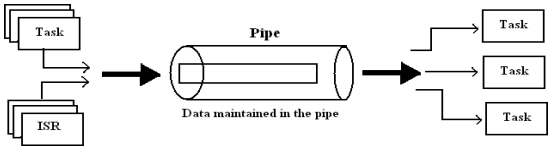
Pipes:

Pipes are kernel objects that are used to exchange unstructured data and facilitate synchronization among tasks. In a traditional implementation, a pipe is a unidirectional data exchange facility, as shown in below Figure.



Two descriptors, one for each end of the pipe (one end for reading and one for writing), are returned when the pipe is created. Data is written via one descriptor and read via the other. The data remains in the pipe as an unstructured byte stream.

Data is read from the pipe in FIFO order. A pipe provides a simple data flow facility so that the reader becomes blocked when the pipe is empty, and the writer becomes blocked when the pipe is full. Typically, a pipe is used to exchange data between a data-producing task and a data-consuming task, as shown in the below Figure. It is also permissible to have several writers for the pipe with multiple readers on it.

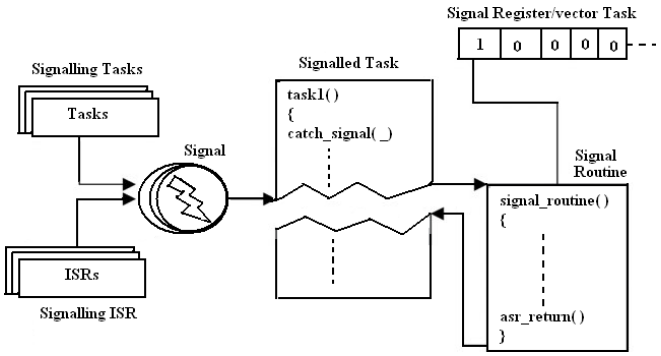


The function calls in the OS API to manage the pipes are:

- Create a pipe
- Open a pipe
- Close a pipe
- Read from the pipe
- Write to the pipe

Signals-Signal Handler

A signal is an event indicator. It is a software interrupt that is generated when an event occurs. It diverts the signal receiver from its normal execution path and triggers the associated asynchronous processing. Mainly the, signals notify tasks of events that occurred during the execution of other tasks or ISRs. The difference between a signal and a normal interrupt is that signals are so-called software interrupts, which are generated via the execution of some software within the system. By contrast, normal interrupts are usually generated by the arrival of an interrupt signal on one of the CPU's external pins. They are not generated by software within the system but by external devices. The number and type of signals defined is both system-dependent and RTOS- dependent. An easy way to understand signals is to remember that each signal is associated with an event. The event can be either unintentional, such as an illegal instruction encountered during program execution, or the event may be intentional, such as a notification to one task from another that it is about to terminate. While a task can specify the particular actions to undertake when a signal arrives, the task has no control over when it receives signals. Consequently, the signal arrivals often appear quite random,



When a signal arrives, the task is diverted from its normal execution path, and the corresponding signal routine is invoked. The terms signal routine, signal handler, asynchronous event handler, and asynchronous signal routine are inter-changeable. Each signal is identified by an integer value, which is the signal number or vector number.

The function calls to manage a signal are

- Install a signal handler
- Remove an installed signal handler
- Send a signal to another task
- Block a signal from being delivered
- Unblock a blocked signal
- Ignore a signal.

Timers:

A timer is the scheduling of an event according to a predefined time value in the future, like setting an alarm clock. For instance, the kernel has to keep track of different times.

- A particular task may need to be executed periodically, say, every 10ms. A timer is used to keep track of this periodicity.
- A task may be waiting in a queue for an event to occur. If the event does not occur for a specified time, it has to take appropriate action.
- A task may be waiting in a queue for a shared resource. If the resource is not available for a specified time, an appropriate action has to be taken.

The following function calls are provided to manage the timer:

- Get time
- Set time
- Time delay (in system clock ticks)
- Time delay (in seconds)
- Reset timer

Memory Management:

It is a service provided by a kernel which allots the memory needed, either static or dynamic for various processes. The manager optimizes the memory needs and memory utilization. The memory manager allocates memory to the processes and manages it with appropriate protection. There may be static and dynamic allocations of memory. The manager optimizes the memory needs and memory utilization. An RTOS may disable the support to the dynamic block allocation, MMU support to the dynamic page allocation and dynamic binding as this increases the latency of servicing the tasks and ISRs.

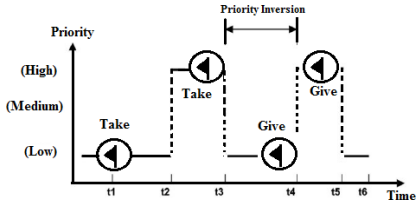
Hence, the two instructions “Malloc” and “free”, although available in C language , are not used by the embedded engineer ,because of the latency problem.

So, an RTOS may or may not support memory protection in order to reduce the latency and memory needs of the processes.

The API provides the following function calls to manage memory

- Create a memory block
- Get data from memory
- Post data in the memory
- Query a memory block
- Free the memory block.

Priority Inversion Problem: In any real time embedded system, if a high priority task is blocked or waiting and a low priority task is running or under execution ,this situation is called Priority Inversion. This priority Inversion is shown in the diagram below.



In Scheduling, priority inversion is the scenario where a low priority Task holds a shared resource that is required by a high priority task. This causes the execution of the high priority task to be blocked until the low priority task releases the resource, effectively "inverting" the relative priorities of the two tasks.

Suppose some other medium priority task, one that does not depend on the shared resource, attempts to run in the interim, it will take precedence over both the low priority task and the high priority task.

The consequences of the priority Inversion are

(i) Reduce the performance of the system

(ii) May reduce the system responsiveness which leads to the violation of response time guarantees

(iii) Create problems in real-time systems.

There are two types of priority inversions.(i) Bounded and (ii).Unbounded.

For example let us consider two tasks T_A and T_B in a real time system. Task A has higher priority than Task B. Initially Task A is under execution. But Task A is blocked after some time due to interruption and Task B is scheduled next. The Task B acquires a mutex corresponding to a resource common to both Task A and Task B. After some time Task A acquire mutex before the completion of Task B. But Task A cannot acquire mutex and it is blocked because already Task B has acquired the mutex. So, the Task A, though has the higher priority, is blocked until Task B releases the mutex for the resource. This is called the bounded Priority inversion.

If the time over which the higher priority is blocked is unknown, then it is called unbounded priority inversion. The Priority Inversion is avoided by using two protocols, namely

Priority Inheritance Protocol (PIP), Priority Ceiling Protocol (PCP).

The Priority Inheritance Protocol is a resource access control protocol that raises the priority of a task, if that task holds a resource being requested by a higher priority task, to the same priority level as the higher priority task.

The **priority ceiling protocol** is a synchronization protocol for shared resources to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections .In this protocol each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource.

Saving Memory and Power:

Saving memory:

- Embedded systems often have limited memory.
- RTOS: each task needs memory space for its stack.
- The first method for determining how much stack space a task needs is to examine your code
- The second method is experimental. Fill each stack with some recognizable data pattern at startup, run the system for a period of time

Program Memory:

- Limit the number of functions used
- Check the automatic inclusions by your linker: may consider writing own functions
- Include only needed functions in RTOS
- Consider using assembly language for large routines

Data Memory:

- Consider using more static variables instead of stack variables
- On 8-bit processors, use char instead of int when possible
- Few ways to save code space:
- Make sure that you are not using two functions to do the same thing.
- Check that your development tools are not sabotaging you.
- Configure your RTOS to contain only those functions that you need.
- Look at the assembly language listings created by your cross-compiler to see if certain of your C statements translate into huge numbers of instructions.

Saving power:

- The primary method for preserving battery power is to turn off parts or all of the system whenever possible.
- Most embedded-system microprocessors have at least one power-saving mode; many have several.
- The modes have names such as sleep mode, low-power mode, idle mode, standby mode, and so on.
- A very common power-saving mode is one in which the microprocessor stops executing instructions, stops any built-in peripherals, and stops its clock circuit. This saves a lot of power, but the drawback typically is that the only way to start the microprocessor up again is to reset it.
- Static RAM uses very little power when the microprocessor isn't executing instructions
- Another typical power-saving mode is one in which the microprocessor stops executing instructions but the on-board peripherals continue to operate.
- Another common method for saving power is to turn off the entire system and have the user turn it back on when it is needed.

Shared memory:

In this model stored information in a shared region of memory is processed, possibly under the control of a supervisor process.

An example might be a single node with

- multiple cores

- share a global memory space
- cores can efficiently exchange/share data

Message Passing:

In this model, data is shared by sending and receiving messages between co-operating processes, using system calls. Message Passing is particularly useful in a distributed environment where the communicating processes may reside on different, network connected, systems. Message passing architectures are usually easier to implement but are also usually slower than shared memory architectures.

An example might be a networked cluster of nodes

- nodes are networked together.
- each with multiple cores.
- each node using its own local memory. /li>
- communicate between nodes and cores via messages.

A message might contain:

1. Header of message that identifies the sending and receiving processes
2. A block of data
3. Process control information

Typically Inter-Process Communication is built on two operations, send() and receive() involving communication links created between co-operating processes.

Remote Procedure Call (RPC):

RPC allows programs to call procedures located on other machines. When a process on machine A calls' a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. This method is known as Remote Procedure Call, or often just RPC.

It can be said as the special case of message-passing model. It has become widely accepted because of the following features: Simple call syntax and similarity to local procedure calls. Its ease of use, efficiency and generality. It can be used as an IPC mechanism between processes on different machines and also between different processes on the same machine.

Sockets:

Sockets (Berkley sockets) are one of the most widely used communication APIs. A socket is an object from which messages and are sent and received. A socket is a network communication endpoint.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here: Data transfer operations on sockets work just like read and write operations on files. A socket is closed, just like a file, when communications is finished.

Network communications are conducted through a pair of cooperating sockets, each known as the peer of the other.

Processes connected by sockets can be on different computers (known as a heterogeneous environment) that may use different data representations. Data is serialized into a sequence of bytes by the local sender and deserialized into a local data format at the receiving end.

Task Synchronization:

All the tasks in the multitasking operating systems work together to solve a larger problem and to synchronize their activities, they occasionally communicate with one another.

For example, in the printer sharing device the printer task doesn't have any work to do until new data is supplied to it by one of the computer tasks. So the printer and the computer tasks must communicate with one another to coordinate their access to common data buffers. One way to do this is to use a data structure called a mutex. Mutexes are mechanisms provided by many operating systems to assist with task synchronization.

A mutex is a multitasking-aware binary flag. It is because the processes of setting and clearing the binary flag are atomic (i.e. these operations cannot be interrupted). When this binary flag is set, the shared data buffer is assumed to be in use by one of the tasks. All other tasks must wait until that flag is cleared before reading or writing any of the data within that buffer.

The atomicity of the mutex set and clear operations is enforced by the operating system, which disables interrupts before reading or modifying the state of the binary flag.

Device drivers:

Simplify the access to devices – Hide device specific details as much as possible – Provide a consistent way to access different devices.

A device driver USER only needs to know (standard) interface functions without knowledge of physical properties of the device .

A device driver DEVELOPER needs to know physical details and provides the interface functions as specified.

UNIT-IV
EMBEDDED SOFTWARE DEVELOPMENT TOOLS

Contents at a glance:

I. Host and target machines
II. linker/locators for embedded software
III. getting embedded software into the target system

DEBUGGING TECHNIQUES

IV. Testing on host machine
V. using laboratory tools
VI. an example system

I. HOST AND TARGET MACHINES:

- **Host:**
  - A computer system on which all the programming tools run
  - Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device.
- **Target:**
  - After writing the program, compiled, assembled and linked, it is moved to target
  - After development, the code is cross-compiled, translated – cross-assembled, linked into target processor instruction set and located into the target.

Host System	Target Computer System
Writing, editing a program, compiling it, linking it, debugging it are done on host system	After the completion of programming work, it is moved from host system to target system.
It is also referred as Work Station	No other name
Software development is done in host system for embedded system	Developed software is shifted to customer from host
Compiler, linker, assembler, debugger are used	Cross compiler is also used
Unit testing on host system ensures software is working properly	By using cross compiler, unit testing allows to recompile code ,execute, test on target system
Stubs are used	Real libraries
Programming centric	Customer centric

Cross Compilers:

- A **cross compiler that runs on host system** and produces the binary instructions that will be understood by your target microprocessor.

- *A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on aWindows 7 PC but generates code that runs on Android smartphone is a cross compiler.*

- Most desktop systems are used as hosts come with compilers, assemblers, linkers that will run on the host. These tools are called native tools.

- Suppose the native compiler on a Windows NT system is based on Intel Pentium. This compiler may possible if target microprocessor is also Intel Pentium. This is not possible if the target microprocessor is other than Intel i.e. like MOTOROLA, Zilog etc.

- A cross compiler that runs on host system and produces the binary instructions that will be understood by your target microprocessor. This cross compiler is a program which will do the above task. If we write C/C++ source code that could compile on native compiler and run on host, we could compile the same source code through cross compiler and make run it run on target also.

- That may not possible in all the cases since there is no problem with if, switch and loops statements for both compilers but there may be an error with respect to the following:

    - In Function declarations
    - The size may be different in host and target
    - Data structures may be different in two machines.
    - Ability to access 16 and 32 bit entries reside at two machines.

  Sometimes cross compiler may warn an error which may not be warned by native complier.

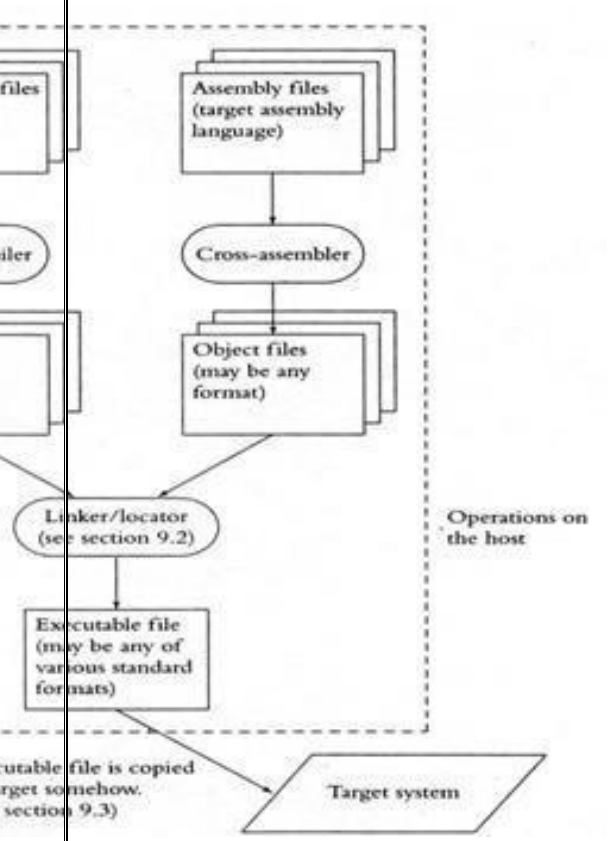
Cross Assemblers and Tool Chains:

- Cross assembling is necessary if target system cannot run an assembler itself.

- **A cross assembler is a program that runs on host produces binary instructions appropriate for the target.** The input to the cross assembler is assembly language file (.asm file) and output is binary file.

- A cross-assembler is just like any other assembler except that it runs on some CPU other than the one for which it assembles code.

Tool chain for building embedded software shown below:



The figure shows the process of building software for an embedded system.

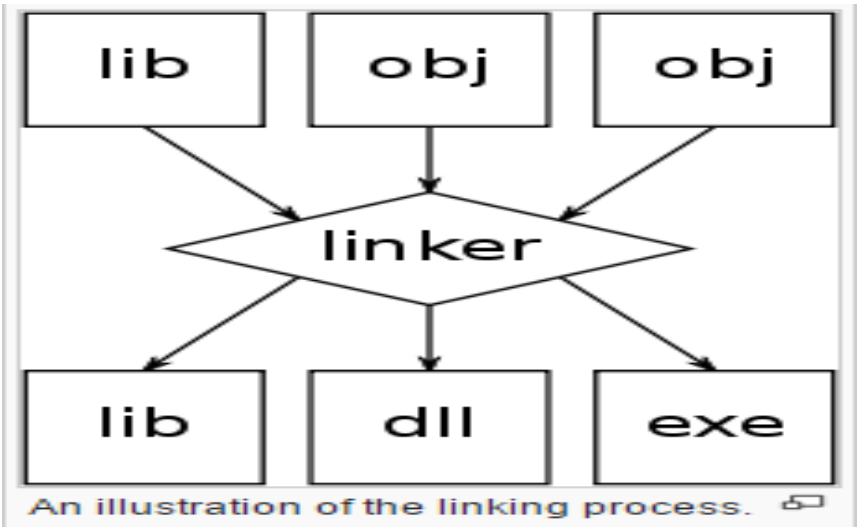
As you can see in figure the output files from each tool become the input files for the next. Because of this the tools must be compatible with each other.

A set of tools that is compatible in this way is called tool chain. Tool chains that run on various hosts and builds programs for various targets.

II. LINKER/LOCATORS FOR EMBEDDED SOFTWARE:

- **Linker:**
  - a linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.
- **Locator:**
  - *locate* embedded binary code into target processors
  - produces target machine code (which the locator glues into the RTOS) and the combined code (called map) gets copied into the target ROM

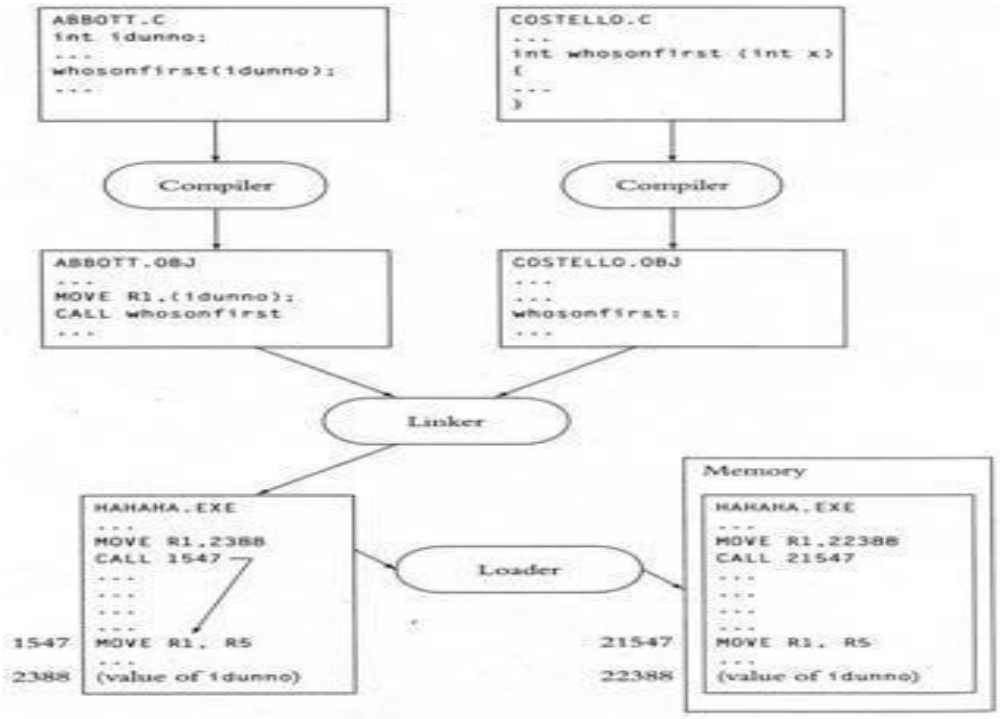
Linking Process shown below:



An illustration of the linking process.

- The **native linker** creates a file on the disk drive of the host system that is read by a part of operating system called the loader whenever the user requests to run the programs.
- **The loader** finds memory into which to load the program, copies the program from the disk into the memory
- Address Resolution:

Native Tool Chain:



Explanation for above native tool chain figure:

- Above Figure shows the process of building application software with native tools. One problem in the tool chain must solve is that many microprocessor instructions contain the addresses of their operands.

- the above figure MOVE instruction in ABBOTT.C will load the value of variable idunno into register R1 must contain the address of the variable. Similarly CALL instruction must contain the address of the whosonfirst. This process of solving problem is called address resolution.

- When abbott.c file compiling,the compiler does not have any idea what the address of idunno and whosonfirst() just it compiles both separately and leave them as object files for linker.

- Now linker will decide that the address of idunno must be patched to whosonfirst() call instructoin. When linker puts the two object files together, it figures out idunno and whosonfirst() are in relation for execution and places in executable files.

- After loader copies the program into memory and exactly knows where idunno and whosonfirst() are in memory. This whole process called as **address resolution.**

Output File Formats:

In most embedded systems there is no loader, when the locator is done then output will be copied

to target.

Therefore the locator must know where the program resides and fix up all memories.

Locators have mechanism that allows you to tell them where the program will be in the target system. Locators use any number of different output file formats.

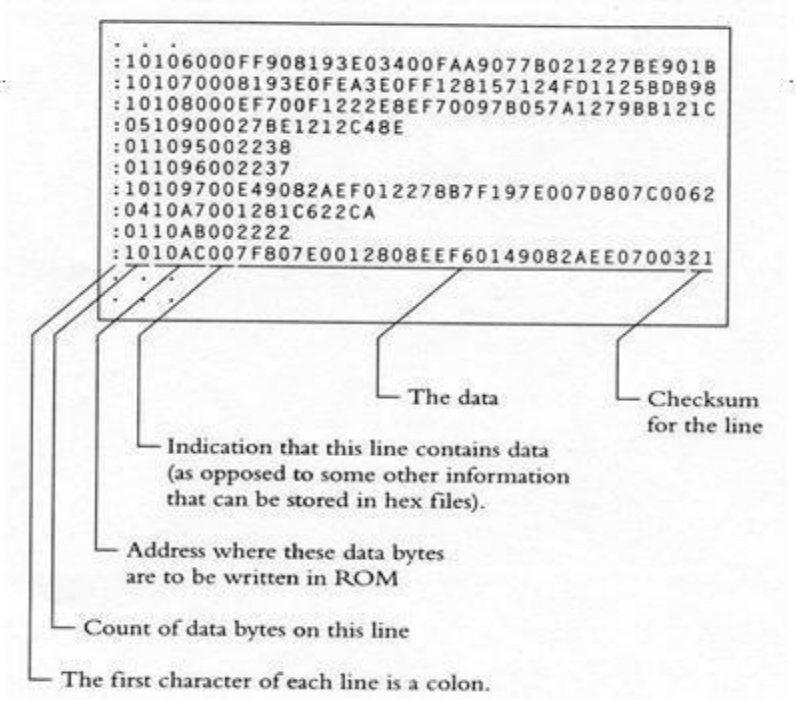
The tools you are using to load your program into target must understand whatever file format your locator produces.

1. intel Hex file format

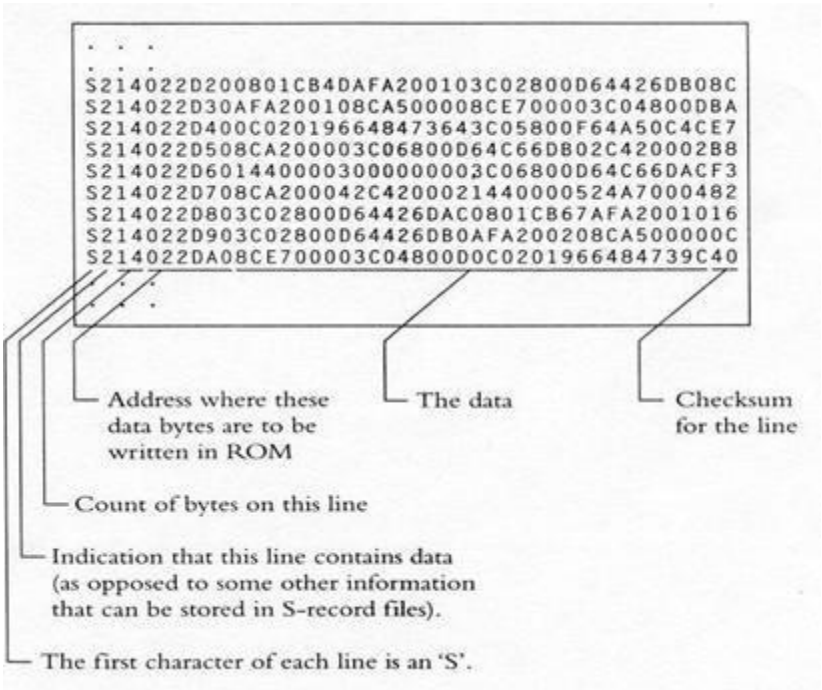
2. Motorola S-Record format

1. Intel Hex file format:

below figure shows Intel Hex file format



2. Motorola S-Record format



Loading program components properly:

Another issue that locators must resolve in the embedded environment is that some parts of the program need to end up in the ROM and some parts need to end up in RAM.

For example whosonfirst() end up in ROM and must be remembered even power is off. The variable idunno would have to be in RAM, since it data may be changed.

This issue does not arise with application programming, because the loader copies the entire program into RAM.

Most tools chains deal with this problem by **dividing the programs into segments**. Each **segment is a piece of program that the locator can place it in memory independently of other segments.**

Segments solve other problems like when processor power on, embedded system programmer must ensure where the first instruction is at particular place with the help of segments.

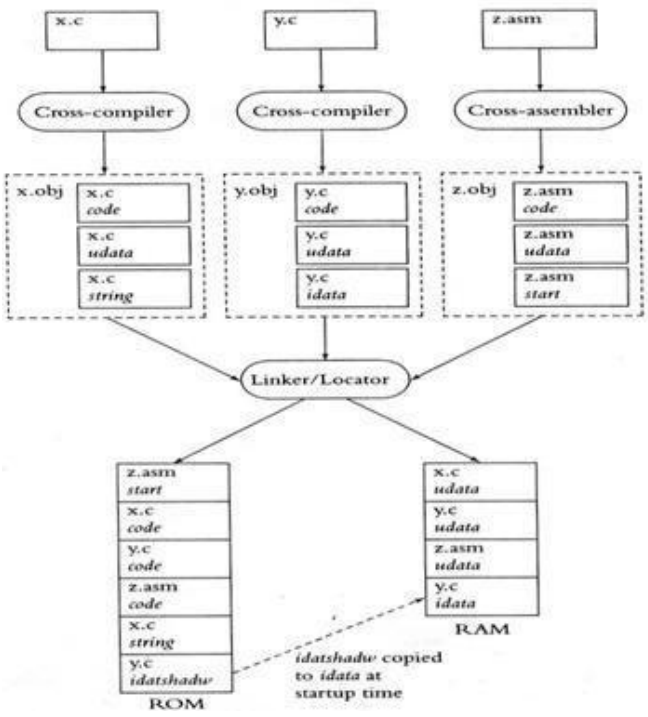


figure: How the tool chain uses segments

Figure shows how a tool chain might work in a system in hypothetical system that contains three modules X.c, Y.c and Z.asm.The code X.c contains some instructions, some uninitialized data and some constant strings. The Y. c contains some instructions, some uninitialized and some initialized data. The Z.asm contains some assembly language function, start up code and uninitialized code

The cross compiler will divide X.c into 3 segments in the object file

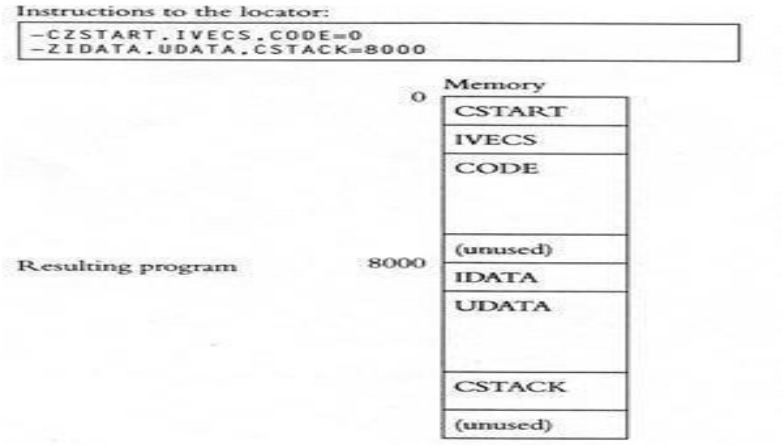
First segment: code
Second segment:
udata
Third segment: constant strings

- The cross compiler will divide Y.c into 3 segments in the object file First segment: code
  Second segment:
  udata Third segment:
  idata
- The cross compiler Z.asm divides the segments into First Segment: assembly language functions Second Segment: start up code
  Third Segment t: udata

The linker/ Locator reshuffle these segments and places Z.asm start up code at where processor begins its execution, it places code segment in ROM and data segment in RAM. Most compilers automatically divide the module into two or more segments: The instructions (code), uninitialized code, Initialized, Constant strings. Cross assemblers also allow you to specify the segment or segments into which the output from the assembler should be placed. Locator places the segments in memory. The following two lines of instructions tells one commercial locator how to build the program.

Fig 6: Locator places segments in memory

- The –Z at the beginning of each line indicates that this line is a list of segments. At the end



of each line is the address where the segment should be placed.

- The locator places the segments one after other in memory, starting with the given address.
- The segments CSTART, IVECS, CODE one after other must be placed at address 0.
- The segments IDATA, UDATA AND CTACK at address at 8000.

Some other features of locators are:

- We can specify the address ranges of RAM and ROM, the locator will warn you if program does not fit within those functions.
- We can specify the address at which the segment is to end, then it will place the segment below that address which is useful for stack memory.
- We can assign each segment into group, and then tell the locator where the group go and deal with individual segments.

Initialized data and constant strings:

Let us see the following code about initialized

```
data: #define FREQ 200
Static int ifreq=
FREQ; void
setfreq(int freq)
{

        int ifreq;
        ifreq =
        freq;
}
```

Where the variable ifreq must be stored. In the above code, in the first case ifreq the initial value must reside in the ROM (this is the only memory that stores the data while the power is off).In the second case the ifreq must be in RAM, because setfreq () changes it frequently.

The only solution to the problem is to store the variable in RAM and store the initial value in ROM and copy the initial value into the variable at startup. Loader sees that each initialized variable has the correct initial value when it loads the program. But there is no loader in embedded system, so that the application must itself arrange for initial values to be copied into variables.

The locator deals with this is to create a shadow segment in ROM that contains all of the initial values, a segment that is copied to the real initialized - data segment at start up. When an embedded system is powdered on the contents of the RAM are garbage. They only become all zeros if some start up code in the embedded system sets them as zeros.

Locator Maps:

- Most **locators will create an output file, called map**, that lists where the locator placed each of the segments in memory.
- A **map** consists of **address of all public functions and global variables**.
- These are useful for debugging an _advanced' locator is capable of running a startup code in ROM, which load the embedded code from ROM into RAM to execute quickly since RAM is faster

Locator MAP IS SHOWN BELOW:

```
LINK MAP OF MODULE:  XYZ

   TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME

-----------------------------------------------------

   * * * * * * *  X D A T A   M E M O R Y  * * * * * * *

             0000H     8100H                   *** GAP ***
   XDATA   8100H     0001H     UNIT          ?XD?PROGFLSH
   XDATA   8101H     000CH     UNIT          ?XD?VPROG?PROGFLSH
   XDATA   810DH     0006H     UNIT          ?XD?CHKSM?PROGFLSH
   XDATA   8113H     0080H     UNIT          ?C_LIB_XDATA
   XDATA   8193H     0002H     UNIT          ?XD?MAIN?PAD
   XDATA   8195H     0002H     UNIT          ?XD?RXCALLBACK?PAD
    :
    :

   * * * * * * *  C O D E   M E M O R Y  * * * * * * *

           0000H     0017H                   *** GAP ***
   CODE    0080H     000FH     UNIT          PROGFLSTSTA
   CODE    008FH     0055H     UNIT          PROGFLSA
   CODE    00E4H     01ADH     UNIT          ?PR?VPROG?PROGFLSH
   CODE    0291H     0073H     UNIT          ?PR?SEND?PROGFLSH
   CODE    0304H     001DH     UNIT          ?PR?RX?PROGFLSH
   CODE    0321H     0072H     UNIT          ?PR?CHKSM?PROGFLSH
   CODE    0393H     007EH     INBLOCK       SCC_INIT
   CODE    0411H     082EH     UNIT          ?C_LIB_CODE
    :
    :
```

Executing out of RAM:

RAM is faster than ROM and other kinds of memory like flash. The fast microprocessors (RISC) execute programs rapidly if the program is in RAM than ROM. But they store the programs in ROM, copy them in RAM when system starts up.

The start-up code runs directly from ROM slowly. It copies rest of the code in RAM for fast processing. The code is compressed before storing into the ROM and start up code decompresses when it copies to RAM.

The system will do all this things by locator, locator must build program can be stored at one collection of address ROM and execute at other collection of addresses at RAM.

Getting embedded software into the target system:

- The locator will build a file as an image for the target software. There are few ways to getting the embedded software file into target system.
  - PROM programmers
  - ROM emulators
  - In circuit emulators
  - Flash
  - Monitors

PROM Programmers:

- The classic way to get the software from the locator output file into target system by creating file in ROM or PROM.
- Creating ROM is appropriate when software development has been completed, since cost to build ROMs is quite high. Putting the program into PROM requires a device called PROM programmer device.
- PROM is appropriate if software is small enough, if you plan to make changes to the software and debug. To do this, place PROM in socket on the Target than being soldered directly in the circuit (the following figure shows). When we find bug, you can remove the PROM containing the software with the bug from target and put it into the eraser (if it is an erasable PROM) or into the waste basket. Otherwise program a new PROM with software which is bug fixed and free, and put that PROM in the socket. We need small tool called chip puller (inexpensive) to remove PROM from the socket. We can insert the PROM into socket without any tool than thumb (see figure8). If PROM programmer and the locator are from different vendors, its upto us to make them compatible.

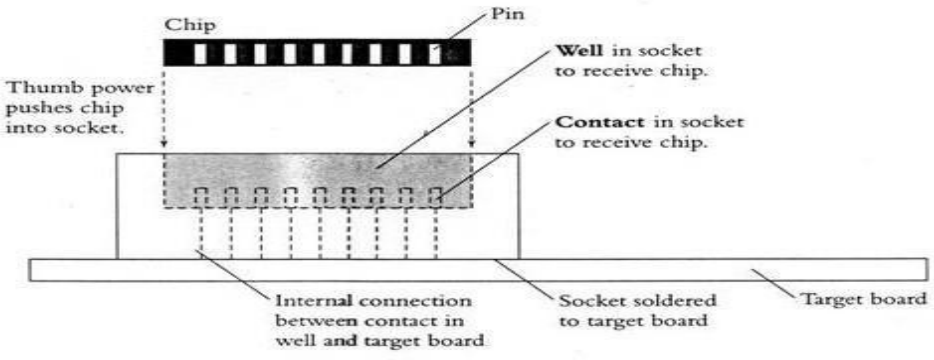


Fig : Semantic edge view of socket

ROM Emulators:

Other mechanism is ROM emulator which is used to get software into target. ROM emulator is a device that replaces the ROM into target system. It just looks like ROM, as shown figure9; ROM emulator consists of large box of electronics and a serial port or a network connection through which it can be connected to your host. Software running on your host can send files created by the locator to the ROM emulator. Ensure the ROM emulator understands the file format which the locator creates.

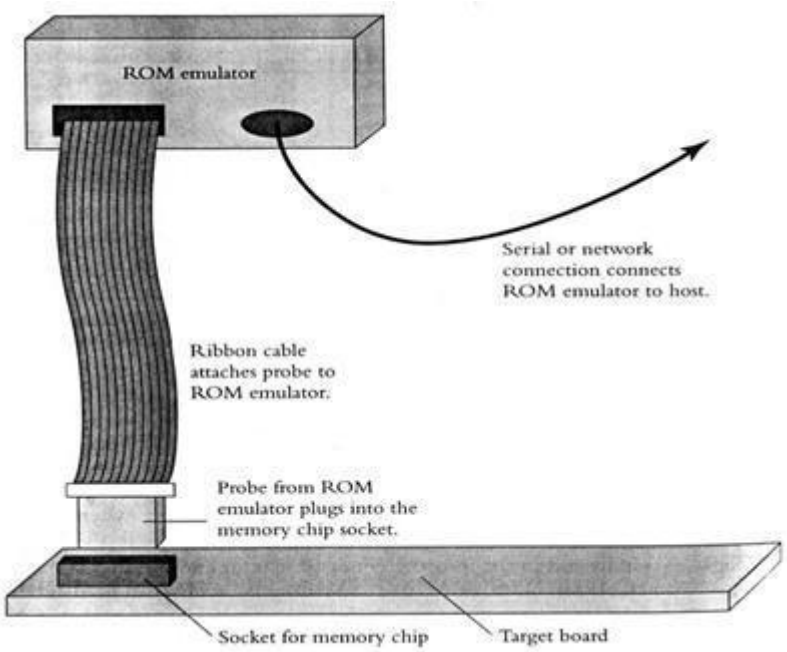


Fig: ROM emulator

In circuit emulators:

If we want to debug the software, then we can use overlay memory which is a common feature of in-circuit emulators. In-circuit emulator is a mechanism to get software into target for debugging purposes.

Flash:

If your target stores its program in flash memory, then one option you always have is to place flash memory in socket and treat it like an EPROM .However, If target has a serial port, a network connection, or some other mechanism for communicating with the outside world, link then target can communicate with outside world, flash memories open up another possibility: you can write a piece of software to receive new programs from your host across the communication link and write them into the flash memory. Although this may seem like difficult

The reasons for new programs from host:

- You can load new software into your system for debugging, without pulling chip out of socket and replacing.
- Downloading new software is fast process than taking out of socket, programming and returning into the socket.
- If customers want to load new versions of the software onto your product.

The following are some issues with this approach:

- Here microprocessor cannot fetch the instructions from flash.
- The flash programming software must copy itself into the RAM, locator has to take care all these activities how those flash memory instructions are executing.
- We must arrange a foolproof way for the system to get flash programming software into the target i.e target system must be able to download properly even if earlier download crashes in the middle.
- To modify the flash programming software, we need to do this in RAM and then copy to flash.

Monitors:

It is a program that resides in target ROM and knows how to load new programs onto the system. A typical monitor allows you to send the data across a serial port, stores the software in the target RAM, and then runs it. Sometimes monitors will act as locator also, offers few debugging services like setting break points, display memory and register values. You can write your own monitor program.

DEBUGGING TECHNIQUES

I. Testing on host machine
II. using laboratory tools
III. an example system

Introduction:

While developing the embedded system software, the developer will develop the code with the lots of bugs in it. The testing and quality assurance process may reduce the number of bugs by some factor. But only the way to ship the product with fewer bugs is to write software with few fewer bugs. The world extremely intolerant of buggy embedded systems. The testing and debugging will play a very important role in embedded system software development process.

Testing on host machine :

- **Goals of Testing process are**
  - **Find bugs early in the development process**
  - **Exercise all of the code**
  - **Develop repeatable , reusable tests**
  - **Leave an audit trail of test results**

Find the bugs early in the development process:

This saves time and money. Early testing gives an idea of how many bugs you have and then how much trouble you are in.

BUT: the target system is available early in the process, or the hardware may be buggy and unstable, because hardware engineers are still working on it.

Exercise all of the code:

Exercise all exceptional cases, even though, we hope that they will never happen, exercise them and get experience how it works.

BUT: It is impossible to exercise all the code in the target. For example, a laser printer may have code to deal with the situation that arise when the user presses the one of the buttons just as a paper jams, but in the real time to test this case. We have to make paper to jam and then press the button within a millisecond, this is not very easy to do.

Develop reusable, repeatable tests:

It is frustrating to see the bug once but not able to find it. To make refuse to happen again, we need to repeatable tests.

BUT: It is difficult to create repeatable tests at target environment.

Example: In bar code scanner, while scanning it will show the pervious scan results every time, the bug will be difficult to find and fix.

Leave an "Audit trail" of test result:

Like telegraph –seems to work‖ in the network environment as it what it sends and receives is not easy as knowing, but valuable of storing what it is sending and receiving.

BUT: It is difficult to keep track of what results we got always, because embedded systems do not have a disk drive.

Conclusion: Don't test on the target, because it is difficult to achieve the goals by testing software on target system. The alternative is to test your code on the host system.

Basic Technique to Test:

The following figure shows the basic method for testing the embedded software on the development host. The left hand side of the figure shows the target system and the right hand side shows how the test will be conducted on the host. The hardware independent code on the two sides of the figure is compiled from the same source.

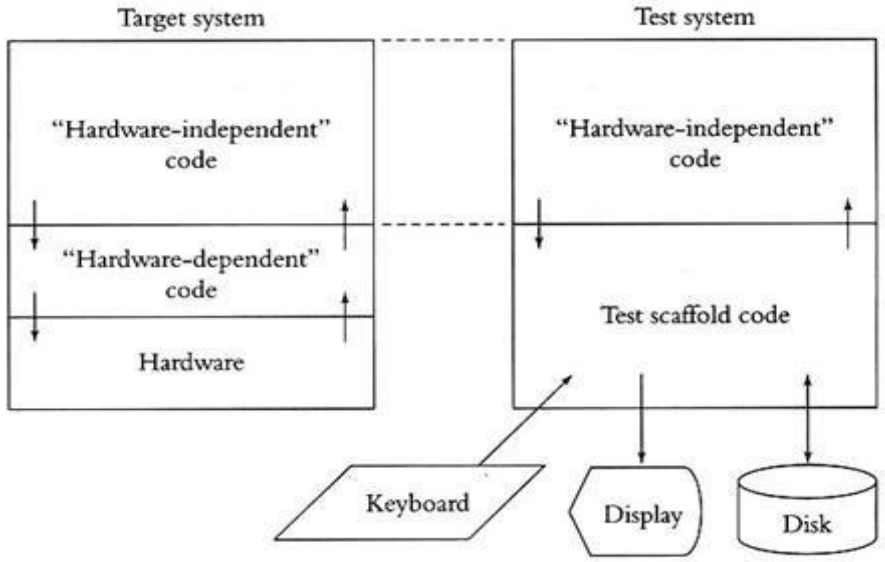


figure: Test System

The hardware and hardware dependent code has been replaced with test scaffold software on the right side. The scaffold software provides the same entry points as does the hardware dependent code on the target system, and it calls the same functions in the hardware independent code. The scaffold software takes its instructions from the keyboard or from a file; it produces output onto the display or into the log file.

Conclusion: Using this technique you can design clean interface between hardware independent software and rest of the code.

Calling Interrupt Routines by scaffold code:

Based on the occurrence of interrupts tasks will be executed. Therefore, to make the system do anything in the test environment, the test scaffold must execute the interrupt routines. Interrupts have two parts one which deals with hardware (by hardware dependent interrupt calls) and other deals rest of the system (hardware independent interrupt calls).

Calling the timer interrupt routine:

One interrupt routine your test scaffold should call is the timer interrupt routine. In most embedded systems initiated the passage of time and timer interrupt at least for some of the activity. You could have the passage of time in your host system call the timer interrupt routine automatically. So time goes by your test system without the test scaffold software participation. It causes your test scaffold to lose control of the timer interrupt routine. So your test scaffold must call Timer interrupt routine directly.

Script files and Output files:

A test scaffold that calls the various interrupt routines in a certain sequence and with certain data. A test scaffold that reads a script from the keyboard or from a file and then makes calls as directed by the script. Script file may not be a project, but must be simple one.

Example: script file to test the bar code scanner

```
#frame arrives
# Dst          Src
               Ctrl
mr/56 ab
#Backoff timeout
expires Kt0
#timeout expires again
Kt0
#sometime pass
Kn2
Kn2
#Another beacon frame arrives
```

Each command in this script file causes the test scaffold to call one of the interrupts in the hardware independent part.

In response to the kt0 command the test scaffold calls one of the timer interrupt routines. In response to the command kn followed by number, the test scaffold calls a different timer interrupt routine the

indicated number of times. In response to the command mr causes the test scaffold to write the data into memory.

Features of script files:

- The commands are simple two or three letter commands and we could write the parser more quickly.
- Comments are allowed, comments script file indicate what is being tested, indicate what results you expect, and gives version control information etc.
- Data can be entered in ASCII or in Hexadecimal.

Most advanced Techniques:

These are few additional techniques for testing on the host. **It is useful to have the test scaffold software do something automatically**. For example, when the hardware independent code for the underground tank monitoring system sends a line of data to the printer, the test scaffold software must capture the line, and it must call the printer interrupt routine to tell the hardware independent code that the printer is ready for the next line.

There may be a need that test scaffold a switch control because there may be button interrupt routine, so that the test scaffold must be able to delay printer interrupt routine. **There may be low, medium, high priority hardware independent requests, then scaffold switches as they appear**. Some Numerical examples of test scaffold software: In Cordless bar code scanner, when H/W independent code sends a frame the scaffold S/W calls the interrupt routine to indicate that the frame has been sent. When H/W independent code sets the timer, then test scaffold code call the timer interrupt after some period. The scaffold software acts as communication medium, which contains multiple instances of H/W independent code with respect to multiple systems in the project.

Bar code scanner Example:

Here the scaffold software generate an interrupts when ever frame send and receive. Bar code Scanner A send data frame, captures by test scaffold and calls frame sent interrupt. The test scaffold software calls receive frame interrupt when it receives frame. When any one of the H/W independent code calls the function to control radio, the scaffold knows which instances have turned their radios, and at what frequencies.

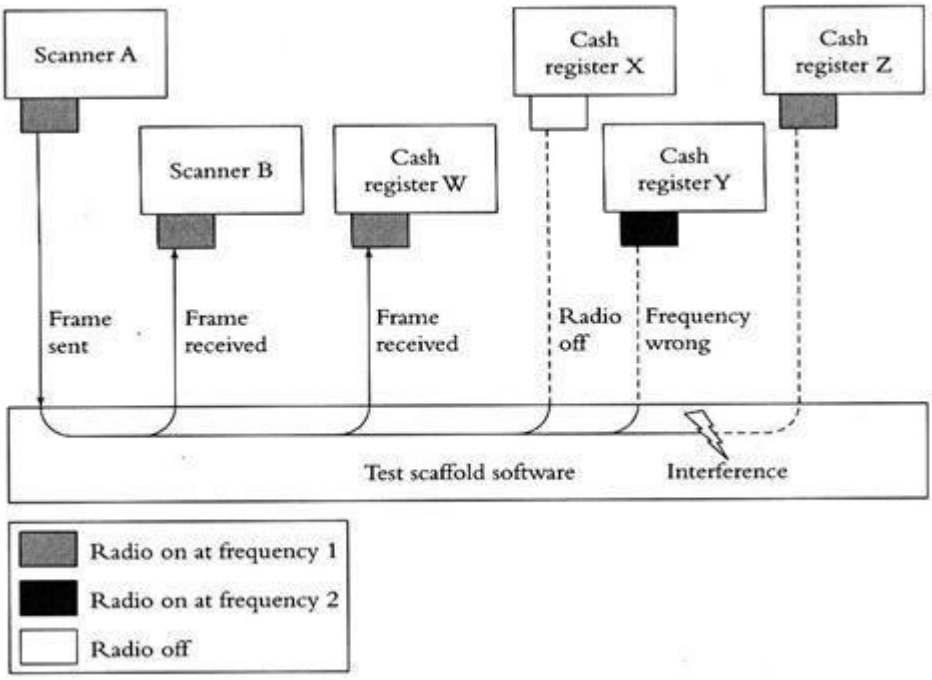


Fig2: Test scaffold for the bar- code scanner software

Targets that have their radios turned off and tuned to different frequencies do not receive the frame.
The scaffold simulates the interference that prevents one or more stations from receiving the data. In this way the scaffold tests various pieces of software communication properly with each other or not.(see the above figure).

OBJECTIONS, LIMITATIONS AND SHORT COMINGS:

Engineers raise many objections to testing embedded system code on their host system, Because many embedded systems are hardware dependent. Most of the code which is tested at host side is hardware dependent code.
To test at host side embedded systems interacts only with the microprocessor, has no direct contact with the hardware. As an example the Telegraph software huge percentage of software is hardware independent i.e. this can be tested on the host with an appropriate scaffold. There are few objections to scaffold: Building a scaffold is more trouble, making compatible to RTOS is other tedious job.

Using laboratory Tools:

- Volt meters and Ohm Meters
- Oscilloscopes
- Logic Analyzers
- Logic Analyzers in Timing mode
- Logic Analyzers in State Mode
- In-circuit Emulators
- Getting ―Visibility‖ into the Hardware
- Software only Monitors
- Other Monitors

Volt meters:

Volt meter is for measuring the voltage difference between two points. The common use of voltmeter is to determine whether or not chip in the circuit have power. A system can suffer power failure for any number of reasons- broken leads, incorrect wiring, etc. the usual way to use a volt meter It is used to turn on the power, put one of the meter probes on a pin that should be attached to the VCC and the other pin that should be attached to ground. If volt meter does not indicate the correct voltage then we have hardware problem to fix.

Ohm Meters:

Ohm meter is used for measuring the resistance between two points, the most common use of Ohm meter is to check whether the two things are connected or not. If one of the address signals from microprocessors is not connected to the RAM, turn the circuit off, and then put the two probes on the two points to be tested, if ohm meter reads out 0 ohms, it means that there is no resistance between two probes and that the two points on the circuit are therefore connected. The product commonly known as Multimeter functions as both volt and Ohm meters.

Oscilloscopes:

It is a device that graphs voltage versus time, time and voltage are graphed horizontal and vertical axis respectively. It is analog device which signals exact voltage but not low or high.
Features of Oscilloscope:

- You can monitor one or two signals simultaneously.
- You can adjust time and voltage scales fairly wide range.
- You can adjust the vertical level on the oscilloscope screen corresponds to ground.

With the use of trigger, oscilloscope starts graphing. For example we can tell the oscilloscope to start graphing when signal reaches 4.25 volts and is rising.

Oscilloscopes extremely useful for Hardware engineers, but software engineers use them for the following purposes:

1. Oscilloscope used as volt meter, if the voltage on a signal never changes, it will display horizontal line whose location on the screen tells the voltage of the signal.
2. If the line on the Oscilloscope display is flat, then no clocking signal is in Microprocessor and it is not executing any instructions.
3. Use Oscilloscope to see as if the signal is changing as expected.
4. We can observe a digital signal which transition from VCC to ground and vice versa shows there is hardware bug.

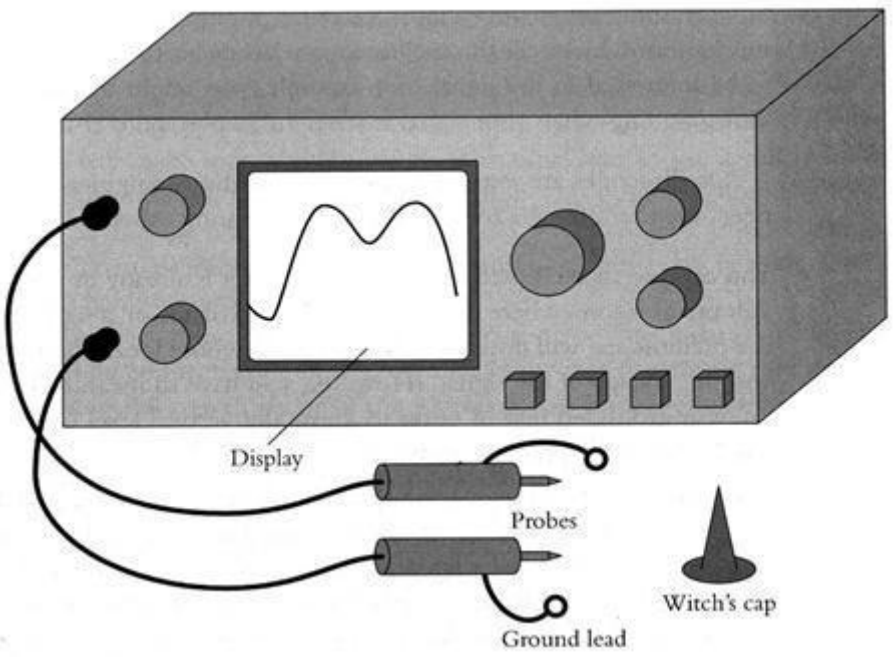
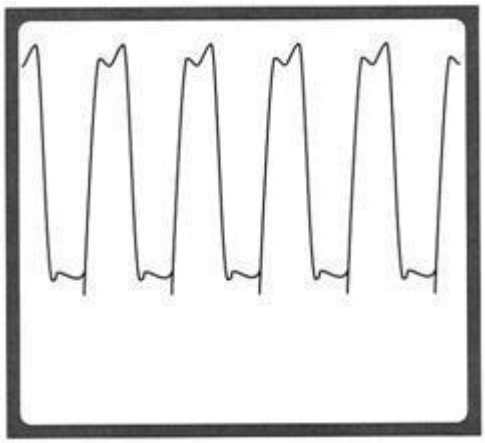
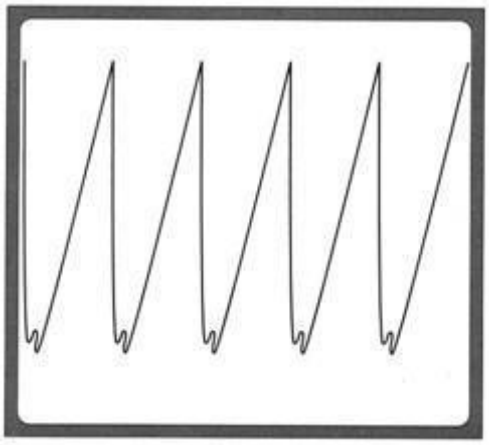


Fig3: Typical Oscilloscope

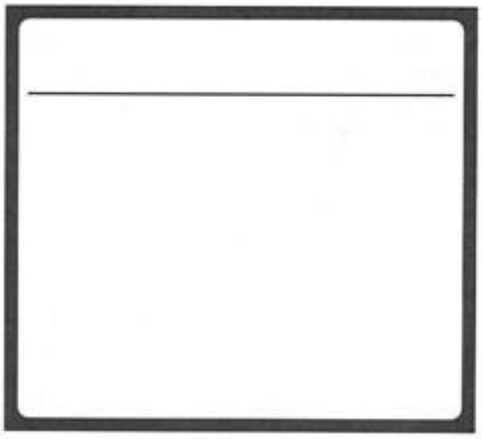
Figure3 is a sketch of a typical oscilloscope, consists of probes used to connect the oscilloscope to the circuit. The probes usually have sharp metal ends holds against the signal on the circuit. Witch's caps fit over the metal points and contain little clip that hold the probe in the circuit. Each probe has ground lead a short wire that extends from the head of the probe, it can easily attach to the circuit. It is having numerous adjustment knobs and buttons allow you to control. Some may have on screen menus and set of function buttons along the side of the screen.



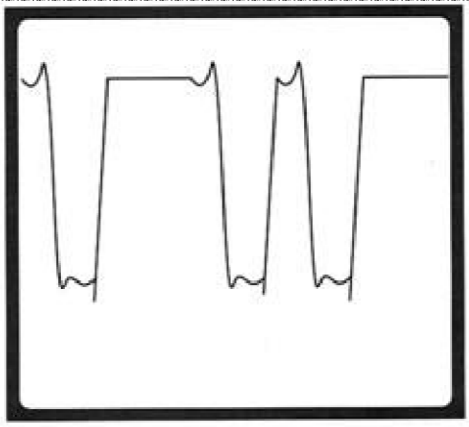
4(a): A Reasonable clock signal



4(b): A Questionable clock signal



4 (c): A dead clock signal



4(d): A ROM chip selection signal

Figure4 (a) to 4(d) shows some typical oscilloscope displays.fig (a) shows a microprocessor input clock signal. Fig (b) shows a questionable clock signal, it differs from 4(a) in that it does not go from lo to high cleanly and stay high for a period of time. Instead it draft from low to high .fig(c) shows a clock circuit that is not working at all.fig(d) shows chip enable signal.

Logic Analyzers:

This tool is similar to oscilloscope, which captures signals and graphs them on its screen. But it differs with oscilloscope in several fundamental ways

- A logic analyzer track many signals simultaneously.
- The logic analyzer only knows 2 voltages, VCC and Ground. If the voltage is in between VCC and ground, then logical analyzer will report it as VCC or Ground but not like exact voltage.
- All logic analyzers are storage devices. They capture signals first and display them later.
- Logic analyzers have much more complex triggering techniques than oscilloscopes.
- Logical analyzers will operate in state mode as well as timing mode.

Logical analyzers in Timing Mode:

Some situations where logical analyzers are working in Timing mode

- If certain events ever occur.
- Example: In bar code scanner software ever turns the radio on, we can attach logic analyzer to the signals that controls the power to the radio.
- We can measure how long it takes for software to respond.
- We can see software puts out appropriate signal patterns to control the hardware. The underground tank monitoring system to find out how long it will takes the software to turn off the bell when you push a button shown in fig5.

Example: After finishing the data transmitting, we can attach the logical analyzer to RTS and its signal to find out if software lowers RTS at right time or early or late. We can also attach the logical analyzer, to ENABLE/ CLK and DATA signals to EEPROM to find if it works correctly or not.(see fig6).

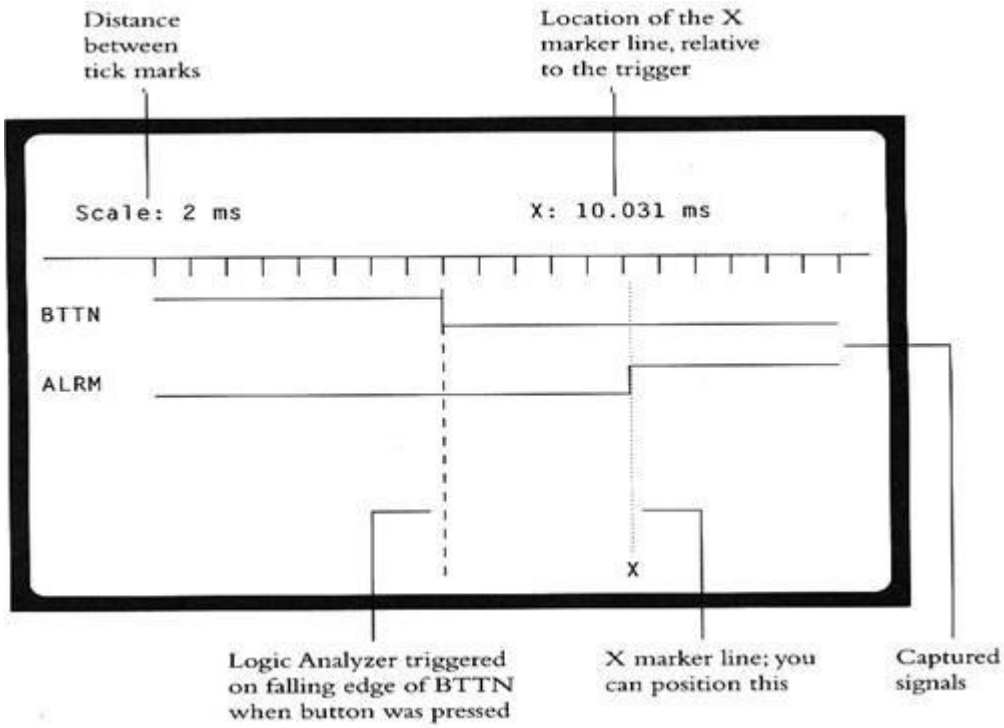


Fig5 : Logic analyzer timing display: Button and Alarm signal

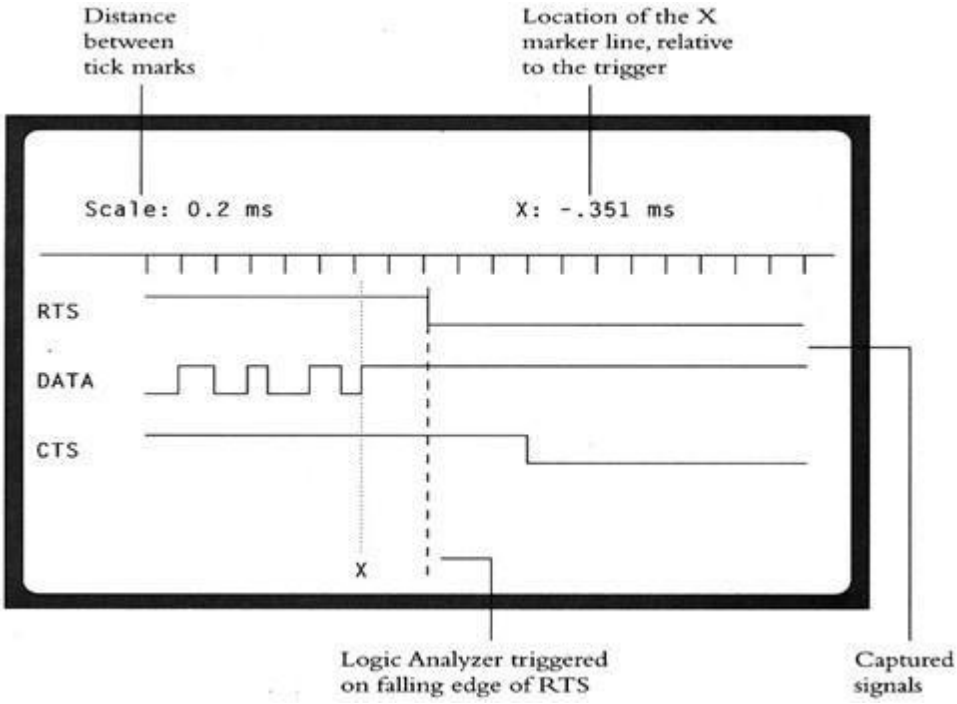


Fig6 : Logic Analyzer timing Display: Data and RTS signal

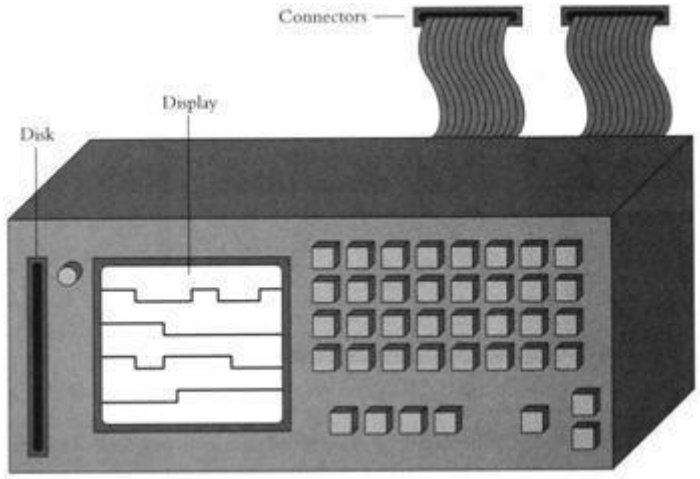


Fig7 : Logic analyzer

Figure7 shows a typical logic analyzer. They have display screens similar to those of oscilloscopes. Most logic analyzers present menus on the screen and give you a keyboard to enter choices, some may have mouse as well as network connections to control from work stations. Logical analyzers include hard disks and diskettes. It can be attached to many signals through ribbons. Since logic analyzer can attach to many signals simultaneously, one or more ribbon cables typically attach to the analyzer.

Logical Analyzer in State Mode:

In the timing mode, logical analyzer is self clocked. That is, it captures data without reference to any events on the circuit. In state mode, they capture data when some particular event occur, called a clock occurs in the system. In this mode the logical analyzer see what instructions the microprocessor fetched and what data it read from and write to its memory and I/O devices. To see what instructions the microprocessor fetched, you connect logical analyzer probes to address and data signals of the system and RE signal on the ROM. Whenever RE signal raise then logical analyzer capture the address and data signals. The captured data is called as trace. The data is valid when RE signal raise. State mode analyzers present a text display as state of signals in row as shown in the below figure.

Fig8 : Typical logic analyzer state mode display

Count	Address	Data	Action	Time
0001	13578	3145	READ	369 ns
0002	1357A	2241	READ	7.44 ns
0003	1357C	1199	WRITE	1.02 ns
0004	1357E	21BC	READ	1.38 ns
0005	02EEA	A1E3	READ	1.78 ns
0006	02EEC	1143	READ	2.01 ns
0007	02EEE	BE45	READ	2.41 ns
0008	02EF0	B1B1	READ	2.73 ns
0009	02EF2	587E	READ	3.04 ns
0010	02EF4	0032	READ	3.44 ns
0011	02EF6	2EEE	READ	4.01 ns
0012	02EEE	BE45	READ	4.41 ns
0013	02EF0	B1B1	READ	4.73 ns
0014	02EF2	587E	READ	5.04 ns
0015	02EF4	0032	READ	5.44 ns
0016	02EF8	143A	READ	6.04 ns
0017	02EFA	318B	READ	6.38 ns

The logical analyzer in state mode extremely useful for the software engineer,

1. Trigger the logical analyzer, if processor never fetch if there is no memory.

2. Trigger the logical analyzer, if processor writes an invalid value to a particular address in RAM.

3. Trigger the logical analyzer, if processor fetches the first instruction of ISR and executed.

4. If we have bug that rarely happens, leave processor and analyzer running overnight and check results in the morning.

5. There is filter to limit what is captured.

Logical analyzers have short comings:

Even though analyzers tell what processor did, we cannot stop, break the processor, even if it did wrong. By the analyzer the processors registers are invisible only we know the contents of memory in which the processors can read or write. If program crashes, we cannot examine anything in the system. We cannot find if the processor executes out of cache. Even if the program crashes, still emulator let make us see the contents of memory and registers. Most emulators capture the trace like analyzers in the state mode. Many emulators have a feature called overlay memory, one or more blocks of memory inside the emulator, emulated microprocessor can use instead of target machine.

In circuit emulators:

In-circuit emulators also called as emulator or ICE replaces the processor in target system.

Ice appears as processor and connects all the signals and drives. It can perform debugging, set break points after break point is hit we can examine the contents of memory, registers, see the source code, resume the execution. Emulators are extremely useful, it is having the power of debugging, acts as logical analyzer. Advantages of logical analyzers over emulators:

- Logical analyzers will have better trace filters, more sophisticated triggering mechanisms.
- Logic analyzers will also run in timing mode.
- Logic analyzers will work with any microprocessor.
- With the logic analyzers you can hook up as many as or few connections as you like. With the emulator you must connect all of the signal.
- Emulators are more invasive than logic analyzers.

Software only Monitors:

One widely available debugging tool often called as Monitor .monitors allow you to run software on the actual target, giving the debugging interface to that of In circuit emulator.
Monitors typically work as follows:

- One part of the monitor is a small program resides in ROM on the target, this knows how to receive software on serial port, across network, copy into the RAM and run on it. Other names for monitor are target agent, monitor, debugging kernel and so on.
- Another part the monitor run on host side, communicates with debugging kernel, provides debugging interface through serial port communication network.
- You write your modules and compile or assemble them.
- The program on the host cooperates with debugging kernel to download compiled module into the target system RAM. Instruct the monitor to set break points, run the system and so on.
- You can then instruct the monitor to set breakpoints.

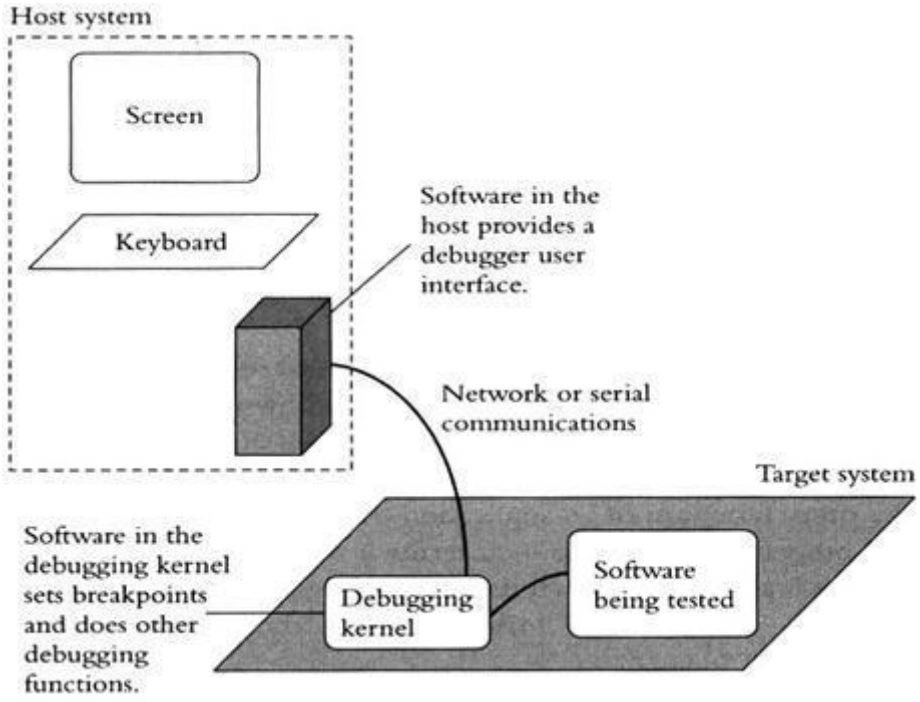


Fig 9: software only the monitor

See the above figure, Monitors are extraordinarily valuable, gives debugging interface without any modifications.

Disadvantages of Monitors:

- The target hardware must have communication port to communicate the debugging kernel with host program. We need to write the communication hardware driver to get the monitor working.
- At some point we have to remove the debugging kernel from your target system and try to run the software without it.
- Most of the monitors are incapable of capturing the traces like of those logic analyzers and emulators.
- Once a breakpoint is hit, stop the execution can disrupt the real time operations so badly.

Other Monitors:

The other two mechanisms are used to construct the monitors, but they differ with normal monitor in how they interact with the target. The first target interface is with through a ROM emulator. This will do the downing programs at target side, allows the host program to set break points and other various debugging techniques.
Advantages of JTAG:

- No need of communication port at target for debugging process.
- This mechanism is not dependent on hardware design.
- No additional software is required in ROM.

UNIT V

Unit V contents at a glance:

I. Introduction to advanced architectures
II. ARM ,
III. SHARC,
IV. processor and memory organization and instruction level parallelism;

networked embedded systems:

I. bus protocols,
II. I2C bus and CAN bus;
III. internet-enabled systems,
IV. design example-elevator controller.

I. INTRODUCTION TO ADVANCED ARCHITECTURES:

Two Computing architectures are available:

1. von Neumann architecture computer
2. Harvard architecture

von Neumann architecture computer:

- The memory holds both data and instructions, and can be read or written when given an address. A computer whose memory holds both data and instructions is known as a von Neumann machine

- The CPU has several internal registers that store values used internally. One of those registers is the
- program counter (PC) ,which holds the address in memory ofaninstruction.

- TheCPUfetchestheinstructionfrommemory,decodestheinstruction, and executes it.

- The program counter does not directly determine what the machine does next,but only indirectly by pointing to an instruction in memory.

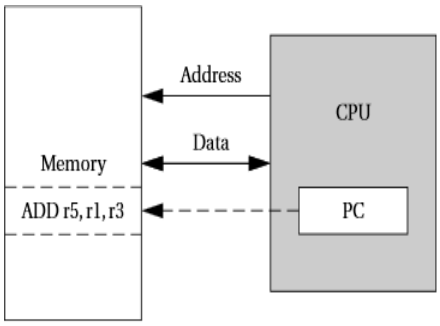


FIGURE 2.1
A von Neumann architecture computer.

2. Harvard architecture:

- Harvard machine has separate memories for data and program.
- The program counter points to program memory, not data memory.
- As a result, it is harder to write self-modifying programs (programs that write data values, then use

  Those values as instructions) on Harvard machines.

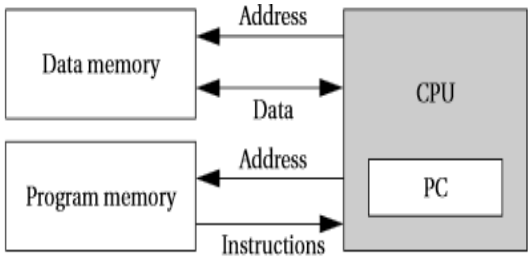


FIGURE 2.2

A Harvard architecture.

Advantage:

- The separation of program and data memories provides higher performance for digital signal processing.

Differences between Von neumann and harvard architecture:

VON NEUMANN	HARVARD ARCHITECTURE
Same memory holds data, instructions	Separate memories for data and instructions
A single set of address/data buses between CPU and memory	Two sets of address/data buses between CPU and memory
Single memory fetch operation	Harvard allows two simultaneous memory fetches
The code is executed serially and takes more clock cycles	The code is executed in parallel
Not exactly suitable for DSP	Most DSPs use Harvard architecture for streaming data: • greater memory bandwidth; • more predictable bandwidth
There is no exclusive Multiplier	It has MAC (Multiply Accumulate)
No Barrel Shifter is there	Barrel Shifterhelp in shifting and rotating operations of the data
The programs can be optimized in lesser size	The program tend to grow big in size
Used in conventional processors found in PCs and Servers, and embedded systems with only control functions.	Used in DSPs and other processors found in latest embedded systems and Mobile communication systems, audio, speech, image processing systems

RISC and CISC Processors:

RISC	CISC
RISC stands for Reduced Instruction Set Computer	CISC stands for Complex Instruction Set Computer
Hardware plays major role in CISC processors	Software plays major role in CISC processors
RISC processors use single clock to execute an instruction	CISC processors use multiple clocks for execution.
Memory-to-memory access is used for data manipulations is RISC processors	intermediate registers are used for data manipulation
In RISC processors, single word instructions are given as inputs	In CISC processors, instructions of variable lengths are given as input, based upon the task to be performed
More lines of code and large memory footprint	High code density
Compact, uniform instructions and hence facilitate pipelining	Many addressing modes and long instructions
Allow effective compiler optimization	Often require manual optimization of assembly code for embedded systems
These machines provided a variety of instructions that may perform very complex tasks, such as string searching	These computers tended to provide somewhat fewer and simpler instructions.

II. ARM(Advanced RISC Machine) Processor:

- **ARM** uses **RISC** architecture
- **ARM** uses **assembly language** for writing programs
- ARM instructions are written one per line, starting after the first column.
- Comments begin with a semicolon and continue to the end of the line.
- A label, which gives a name to a memory location, comes at the beginning of the line, starting in the first column.

Here is an example:

```
        LDR r0,[r8]; a comment
label ADD r4,r0,r1
```

Memory Organization in ARM Processor:

The ARM architecture supports two basic types of data:

- The standard ARM word is 32 bits long.
- The word may be divided into four 8-bit byte
- ARM allows addresses up to 32 bits long
- The ARM processor can be configured at power-up to address the bytes in a word in either **little-endian mode** (with the lowest-order byte residing in the low-order bits of the word) or **big-endian mode**

Data Operations in ARM:

- In the ARM processor, arithmetic and logical operations cannot be performed directly on memory locations.
- ARM is a **load-store architecture**—data operands must first be loaded into the CPU and then stored back to main memory to save the results

ARM Programming Model:

1. Programming model gives information about various registers supported by ARM
2. ARM has 16 general-purpose registers, r0 to r15
3. Except for r15, they are identical—any operation that can be done on one of them can be done on the other one also
4. r15 register is also used as program counter(PC)
5. current program status register (CPSR):
    - This register is set automatically during every arithmetic, logical, or shifting operation.
    - The top four bits of the CPSR hold the following useful information about the results of that arithmetic/logical operation:
        - **The negative (N)** bit is set when the result is negative in two's-complement arithmetic.
        - The **zero (Z)** bit is set when every bit of the result is zero.
        - The **carry (C)** bit is set when there is a carry out of the operation.
        - The **overflow (V)** bit is set when an arithmetic operation results in an overflow.

Types of Instructions supported by ARM Processor:

1. Arithmetic Instructions
2. Logical Instructions
3. shift / rotate Instructions
4. Comparison Instructions
5. move instructions
6. Load store instructions

Instructions examples:

ADD r0,r1,r2

This instruction sets register r0 to the sum of the values stored in r1 and r2.

ADD r0,r1,#2 (immediate operand are allowed during addition)

RSB r0, r1, r2 sets r0 to be r2-r1.

bit clear: BIC r0, r1, r2 sets r0 to r1 and not r2.

Multiplication:

no immediate operand is allowed in multiplication

two source operands must be different registers

MLA: The MLA instruction performs a multiply-accumulate operation, particularly useful in matrix operations and signal processing

MLA r0,r1,r2,r3 sets r0 to the value **r1x r2+r3.**

Shift operations:

Logical shift(LSL, LSR)
Arithmetic shifts (ASL, ASR)

- A **left shift** moves bits up toward the most-significant bits,
- **right shift** moves bits down to the least-significant bit in the word.
- The LSL and LSR modifiers perform left and right logical shifts, filling the least-significant bits of the operand with zeroes.
- The **arithmetic shift left** is equivalent to an LSL, but the ASR copies the sign bit—if the sign is 0, a 0 is copied, while if the sign is 1, a 1 is copied.

Rotate operations: (ROR, RRX)

- The rotate modifiers always rotate right, moving the bits that fall off the least-significant bit up to the most-significant bit in the word.
- The RRX modifier performs a 33-bit rotate, with the CPSR's C bit being inserted above the sign bit of the word; this allows the carry bit to be included in the rotation

Compare instructions: (CMP, CMN)

- **compare instruction modifies flags values (Negative flag, zero flag, carry flag, Overflow flag)**
- **CMP r0, r1 computes r0 – r1**, sets the status bits, and throws away the result of the subtraction.
- CMN uses an addition to set the status bits.
- TST performs a bit-wise AND on the operands,
- while TEQ performs an exclusive-or

Load store instructions:

- ARM uses register-indirect addressing
- The value stored in the register is used as the address to be fetched from memory; the result of that fetch is the desired operand value.
- LDR r0,[r1] sets r0 to the value of memory location 0x100.
- Similarly, STR r0,[r1] would store the contents of r0 in the memory location whose address is given in r1

LDR r0,[r1, – r2]

ARM Register indirect addressing:

LDR r0,[r1, #4] loads r0 from the address r1+ 4.

ARM Base plus offset addressing mode:

The register value is added to another value to form the address.

For instance, **LDR r0,[r1,#16]** loads r0 with the value stored at location r1+16.(r1-base address, 16 is offset)

Auto-indexing updates the base register, such that LDR r0,[r1,#16]!---first adds 16 to the value of r1, and then uses that new value as the address. The ! operator causes the base register to be updated with the computed address so that it can be used again later.

Post-indexing does not perform the offset calculation until after the fetch has been performed. Consequently,

LDR r0,[r1],#

16 will load r0 with the value stored at the memory location whose address is given by r1, and then add 16 to r1 and set r1 to the new value.

FLOW OF CONTROL INSTRUCTIONS

(Branch Instructions):

EQ	Equals zero	Z = 1
NE	Not equal to zero	Z = 0
CS	Carry set	C = 1
CC	Carry clear	C = 0
MI	Minus	N = 1
PL	Nonnegative (plus)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 1 and Z = 0
LS	Unsigned lower or same	C = 0 or Z = 1
GE	Signed greater than or equal	N = V
LT	Signed less than	N V
GT	Signed greater than	Z = 0 and N = V
LE	Signed less than or equal	Z = 1 or N V

FIGURE 2.15

Condition codes in ARM.

Branch Instructions

1. **conditional instructions(BGE-- B is branch, GE is condition)**
2. **unconditional instructions(B)**

the following **branch instruction B #100 will add 400 to the current PC value**

SHARC Processor:

Features of SHARC processor:

1. SHARC stands for **Super Harvard Architecture Computer**
2. The ADSP-21060 SHARC chip is made by Analog Devices, Inc.
3. It is a **32-bit signal processor** made mainly for **sound, speech, graphics, and imaging applications**.
4. It is a high-end digital signal processor designed with **RISC techniques**.
5. Number formats:
   i. 32-bit Fixed Format
   ii. Fractional/Integer Unsigned/Signed
   iii. Floating Point
      32-bit single-precision IEEE floating-point data format
      40-bit version of the IEEE floating-point data format.
      16-bit shortened version of the IEEE floating-point data format.
6. 32 Bit floating point, with 40 bit extended floating point capabilities.
7. Large on-chip memory.
8. Ideal for scalable multi-processing applications.
9. Program memory can store data.
10. Able to simultaneously read or write data at one location and get instructions from another place in memory.
11. 2 buses
      Data memory bus.
      Program bus.
12. Either two separate memories or a single dual-port memory
13. The SHARC incorporates features aimed at optimizing such loops.
14. High-Speed Floating Point Capability
15. Extended Floating Point
16. The SHARC supports floating, extended-floating and non-floating point.
17. No additional clock cycles for floating point computations.
18. Data automatically truncated and zero padded when moved between 32-bit memory and internal registers.

SHARC PROCESSOR PROGRAMMING MODEL:

Programming model gives the registers details. The following registers are used in SHARC processors for various purposes:

- Register files: R0-R15 (aliased as F0-F15 for floating point)
- Status registers.
- Loop registers.
- Data address generator registers(DAG1 and DAG2)
- Interrupt registers.
- 16 primary registers (R0-R15)
- 16 alternate registers (F0-F15)
- each register can hold 40 bits
- R0 – R15 are for Fixed-Point Numbers
- F0 – F15 are for Floating-Point Numbers

Status registers:

ASTAT: arithmetic status.
STKY: sticky.
MODE 1: mode 1.

- The **STKY register** is a sticky version of **ASTAT register**, the STKY bits are set along with ASTAT register bits but not cleared until cleared by an instruction.
- The SHARC perform saturation arithmetic on fixed point values, saturation mode is controlled by ALUSAT bit in MODE1 register.
- All ALU operations set AZ (zero), AN (negative), AV (overflow), AC (fixed-point carry), AI (floating-point invalid) bits in ASTAT.

Data Address Generators(DAG)

- There are two data address generators (DAG1 & DAG2) for addressing memory indirectly (with pre-modify or post-modify). Data address generator 1 (DAG1) generates 32-bit addresses on theData Memory Address Bus.
- Data address generator 2 (DAG2) generates 24-bit addresses on the Program Memory Address Bus.
- Each DAG has four types of registers:
- The Index (I) register acts as a pointer to memory.
- The Modify (M) register contains the increment value for advancing the pointer.
- Base and Limit Registers (More on the next page).

Multifunction computations or instruction level parallel processing:

Can issue some computations in parallel:

- dual add-subtract;
- fixed-point multiply/accumulate and add, subtract, average
- floating-point multiply and ALU operation
- multiplication and dual add/subtract

Pipelining in SHARC processor:

- Instructions are processed in three cycles:
- Fetch instruction from memory
- Decode the opcode and operand
- Execute the instruction
- SHARC supports delayed and non-delayed branches
- Specified by bit in branch instruction
- 2 instruction branch delay slot
- Six Nested Levels of Looping in Hardware Bus Architecture:

Twin Bus Architecture:

1 bus for Fetching Instructions

1 bus for Fetching Data

Improves multiprocessing by allowing more steps to occur during each clock

Addressing modes provided by DAG in SHARC Processor:

1. The Simplest addressing mode
2. Absolute address
3. post modify with update mode
4. base-plus-offset mode
5. Circular Buffers
6. Bit reversal addressing mode

1. The **Simplest addressing mode** provides an immediate value that can represent the address.
   Example : R0=DM(0X200000)
   R0=DM(_a) i.e load R0 with the contents of the variable a
2. An **Absolute address** has entire address in the instruction, space inefficient, address occupies the more space.
3. A **post modify with update mode** allows the program to sweep through a range of address. This uses I register and modifier, I registers shows the address value and modifier (M register value or Immediate value) is update the value.
   For load R0=DM(I3,M1)
   For store : DM(I3,M1)=R0
4. The **base-plus-offset mode** here the address computed as I+M where I is the base and M modifier or offset.
   Example: R0=DM(M1, I0)
   I0=0x2000000 and M0= 4 then the value for R0 is loaded from 0x2000004
5. **Circular Buffers** is an array of n elements is n+1th element is referenced then the location is 0. It is wrapping around from end to beginning of the buffer.
   This mode uses L and B registers, L registers is set with +ve and nonzero value at staring point, B register is stored with same value as the I register is store with base address.
   If I register is used in post modify mode, the incremental value is compared to the sum of L and B registers, if end of the buffer is reached then I register is wrapped around.
6. **Bit reversal addressing mode** : this is used in Fast Fourier Transform (FFT). Bit reversal can be performed only in I0 and I8 and controlled by BR0 and BR8 bits in the MODE1 register. SHARC allows two fetches per cycle.
   F0=DM(M0,I0); FROM DATA MEMORY F1=PM(M8,I8); FROM PROGRAM MEMORY

BASIC addressing:

Immediate value:

```
R0 = DM(0x20000000);
```

Direct load:

```
R0 = DM(_a); ! Loads contents of _a
```

Direct store:

```
DM(_a)= R0; ! Stores R0 at _a
```

SHARC programs examples:

expression:x = (a + b) - c;

program:

```
R0 = DM(_a) ! Load a
R1 = DM(_b); ! Load b
R3 = R0 + R1;
R2 = DM(_c); ! Load c
R3 = R3-R2;
DM(_x) = R3; ! Store result in x
```

expression : y = a*(b+c);

program:

```
R1 = DM(_b) ! Load b
R2 = DM(_c); ! Load c
R2 = R1 + R2;
R0 = DM(_a); ! Load a
R2 = R2*R0;
DM(_y) = R23; ! Store result in y
```

SHARC jump:

Unconditional flow of control change:

```
JUMP foo
```

Three addressing modes:
direct;
indirect;
PC-
relative.

ARM vs. SHARC

• ARM7 is von Neumann architecture
• ARM9 is Harvard architecture
• SHARC is modified Harvard architecture. – On chip memory (> 1Gbit) evenly split between program memory (PM) and data memory (DM) – Program memory can be used to store some data. – Allows data to be fetched from both memory in parallel

The SHARC ALU operations:

1. Fixed point ALU operations
2. Floating point ALU operations
3. Shifter operations in SHARC

Fixed point ALU operations

Rn=Rx+Ry	Add
Rn=Rx- Ry	Subtract
Rn= Rx+Ry+CI	Add with carry
Rn=Rx-Ry+CI-1	Subtract with borrow
Rn=(Rx+Ry)/2	Average
COMP(Rx,Ry)	Compare
Rx=Rx+CI	Add carry
Rn=Rx+CI-1	Add borrow
Rn=Rx+1	Increment
Rn=Rx-1	Decrement
Rn=-Rx	Negate

Rn= ABS Rx	Absolute Value
Rn=PASS Rx	Copy Rx to Rn
Rn=Rx AND Ry	Logical AND
Rn=Rx OR Ry	Logical OR
Rn=Rx XOR Ry	Logical exclusive OR
Rn= NOT Rx	Logical Negate
Rn= MIN (Rx, Ry)	Minimum of Rx, Ry
Rn=MAX (Rx, Ry)	Maximum of Rx, Ry
Rn= CLIP Rx by Ry	Clip Rx within range [-Ry, Ry]

Floating point ALU operations:

Fn=Fx +Fy	Add
Fn=Fx - Fy	Subtract
Fn=ABS (Fx+Fy)	Absolute value of sum
Fn=ABS (Fx-Fy)	Absolute value of difference
Fn=(Fx+Fy)/2	Average
COMP(Fx,Fy)	Compare
Fn=-Fx	Negate
Fn=ABS Fx	Absolute value
Fn=PASS Fx	Copy Fx to Fn
Fn =RND Fx	Round
Fn= SCALB Fx by Ry	Scale exponent of Fx by Ry

Rn = MANT Fx	Extract mantissa of Fx
Rn= LOGB Fx	Convert exponent of Fx into Integer
Rn=FIX Fx, Rn= TRUNC Fx	Convert floating point to integer
Fn= FLOAT Rx by Ry, FLOAT Rx	Convert Integer to Floating Point
Fn= RECIPS Fx	Create seed of Reciprocal
Fn=RSQRTS Fx	Create seed for reciprocal square root
Fn=Fx COPYSIGN Fy	Copy sign of Fy to Fx
Fn=MIN(Fx, Fy)	Minimum of Fx, Fy
Fn=MAX (Fx, Fy)	Maximum of Fx, Fy
Fn= CLIP Fx by Fy	Clip Fx within RANGE [-Fy,Fy]

SHARC Shift operations

Rn= LSHIFT Rx by Ry	Logical shift distance Ry
Rn= Rn OR LSHIFT Rx by Ry	Logical Shift and logical OR
Rn=ASHIFT Rx by Ry	Arithmetic shift
Rn= Rn OR ASHIFT Rx by Ry	Arithmetic shift and logical OR
Rn= ROT Rx by Ry	Rotate distance Ry
Rn= BCLR Rx by Ry	Clear one bit in Rx
Rn=BSET Rx by Ry	Set one bit in Rx
Rn=BTGL Rx by Ry	Toggle one bit in Rx
BTST Rx by Ry	Text one bit in Rx
Rn=FDEP Rx by Ry	Deposit field from Rx into Rn
Rn=Rn OR FDEP Rx by Ry	Deposit field from Rx using OR
Rn=FDEP Rx by Ry	Deposit and sign extend field from Rx
Rn=Rn OR FDEP Rx by Ry	Deposit and sign extend using OR

SHARC Shift operations

Rn=FEXT Rx by Ry	Extract field from Rx
Rn=FEXT Rx by Ry	Extract and sign extend field from Rx
Rn = EXP Rx	Extract exponent field from Rx
Rn = EXP Rx (EX)	Extract exponent field from ALU
Rn = LEFTZ Rx	Extract number of leading zeros
Rn = LEFTO Rx	Extract number of leading ones
Rn = FPACK Rx	Convert 32 bit floating point to 16 bit floating point
Rn = FUNPACK RN	Convert 16 bit floating point to 32 bit floating point

Contents:

Network Embedded System

I. bus protocols,
II. I2 C bus ,
III. CAN bus;
IV. internet enabled systems,
V. design example elevator controller.

I. BUS PROTOCOLS:
For serial data communication between different peripherals components , the following standards are used :

- VME
- PCI
- ISA etc

For distributing embedded applications, the following interconnection network protocols are there:

- I^2C
- CAN etc

I^2C :

- The I 2 C bus is a well-known bus commonly used to link microcontrollers into systems
- I 2C is designed to be low cost, easy to implement, and of moderate speed up to 100 KB/s for the standard bus and up to 400 KB/s for the extended bus
- it uses only two lines: the serial data line (SDL) for data and the serial clock line (SCL), which indicates when valid data are on the data line

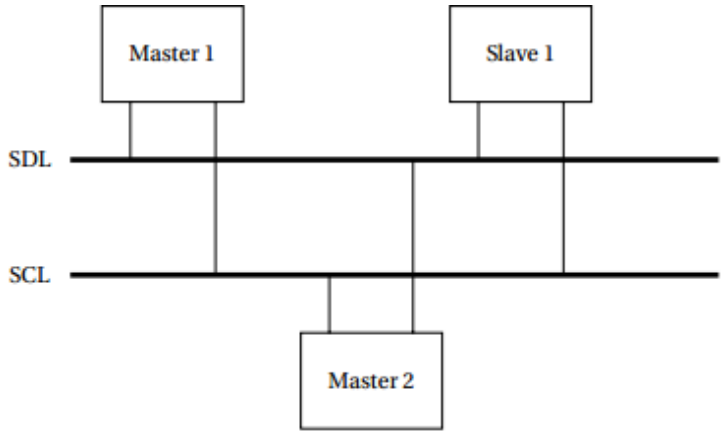


FIGURE 8.7
Structure of an I^2C bus system.

The basic **electrical interface of I2C** to the bus is shown in Figure

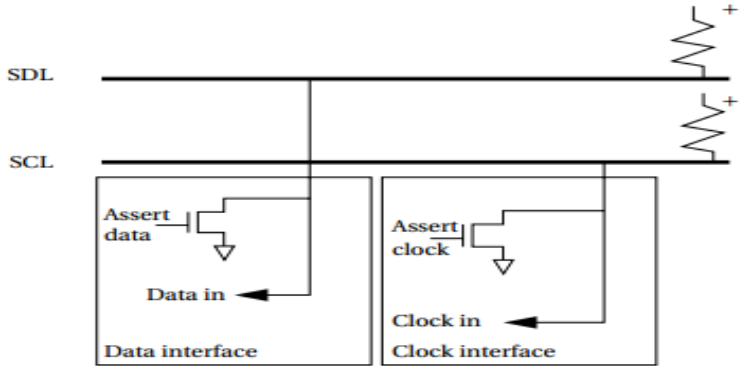


FIGURE 8.8
Electrical interface to the I^2C bus.

- A **pull-up resistor** keeps the default state of the signal high, and transistors are used in each bus device to pull down the signal when a 0 is to be transmitted.
- **Open collector/open drain** signaling allows several devices to simultaneously write the bus without causing electrical damage.
- The open collector/open drain circuitry allows a slave device to stretch a clock signal during a read from a slave.
- The master is responsible for generating the SCL clock, but the slave can stretch the low period of the clock
- The **I2C** bus is designed as a **multimaster bus**—any one of several different devices may act as the master at various times.
- As a result, there is no global master to generate the clock signal on SCL. Instead, a master drives both SCL and SDL when it is sending data. When the bus is idle, both SCL and SDL remain high.
- When **two devices try to drive either SCL or SDL to different values**, the **open collector/ open drain circuitry prevents errors**

Address of devices:

- A device address is 7 bits in the standard I2C definition (the extended I2C allows 10-bit addresses).
- The address 0000000 is used to signal a general call or bus broadcast, which can be used to signal all devices simultaneously**.** A bus transaction comprised a series of 1-byte transmissions and an address followed by one or more data bytes.

data-push programming :

- I2C encourages a data-push programming style. When a master wants to write a slave, it transmits the slave‘s address followed by the data.
- Since a slave cannot initiate a transfer, the master must send a read request with the slave‘s address and let the slave transmit the data.
- Therefore, an address transmission includes the 7-bit address and 1 bit for data direction: 0 for writing from the master to the slave and 1 for reading from the slave to the master

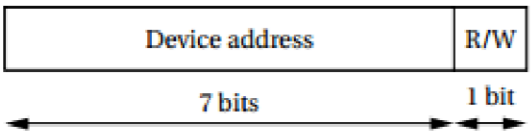


FIGURE 8.9

Format of an I^2C address transmission.

Bus transaction or transmission process:
1) start signal (SCL high and sending 1 to 0 in SDL)
2) followed by device address of 7 bits
3) RW(read / write bit) set to either 0 or 1
4) after address, now the data will be sent
5) after transmitting the complete data, the transmission stops.

The below figure is showing write and read bus transaction:

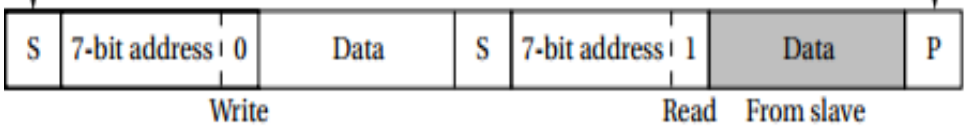


FIGURE 8.11

Typical bus transactions on the I^2C bus.

State transition graph:

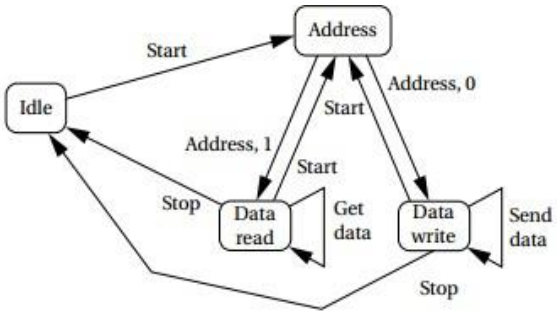


FIGURE 8.10

State transition graph for an I^2C bus master.

Transmitting byte in I2C Bus (Timing Diagram):

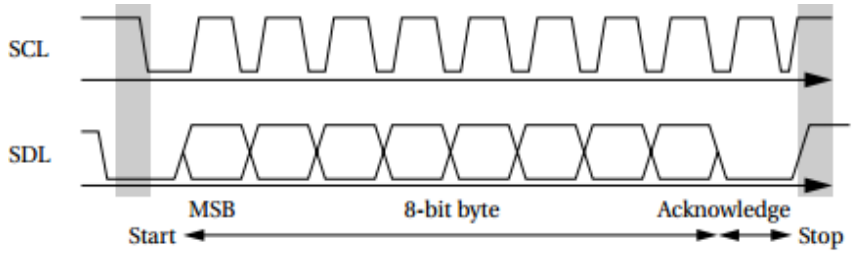


FIGURE 8.12

Transmitting a byte on the I^2C bus.

1. initially, SCL will be high, SDL will be low.
2. data byte will be transmitted.
3. after transmitting every 8 bits, an Acknowledgement will come
4. then stop signal is issued by setting both SCL and SDL high.

I2C interface on a microcontroller:

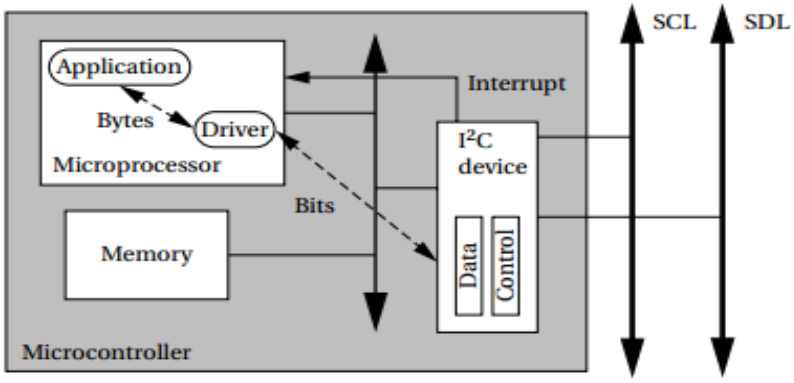


FIGURE 8.13

An I^2C interface in a microcontroller.

Controlled Area Network:

The CAN bus was designed for automotive electronics and was first used in production cars in 1991. The CAN bus uses bit-serial transmission. CAN runs at rates of 1 MB/s over a twisted pair connection of 40 m.

An optical link can also be used. The bus protocol supports multiple masters on the bus.

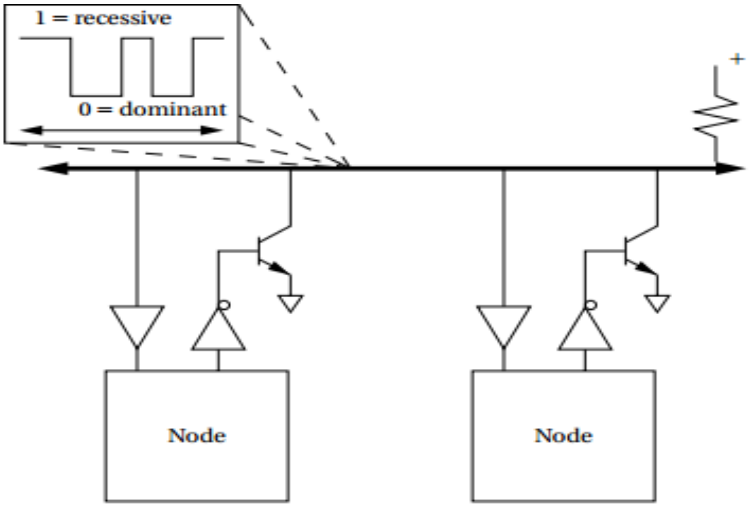


FIGURE 8.22

Physical and electrical organization of a CAN bus.

The above figure shows CAN electrical interface:

- each node in the CAN bus has its own electrical drivers and receivers that connect the node to the bus in wired-AND fashion.
- In CAN terminology, a logical 1 on the bus is called recessive and a logical 0 is **dominant.**
- The driving circuits on the bus cause the bus to be pulled down to 0 if any node on the bus pulls the bus down (making 0 dominant over 1).
- When all nodes are transmitting 1s, the bus is said to be in the recessive state; when a node transmits a 0, the bus is in the dominant state. Data are sent on the network in packets known as **data frames.**

CAN DATA FRAME:

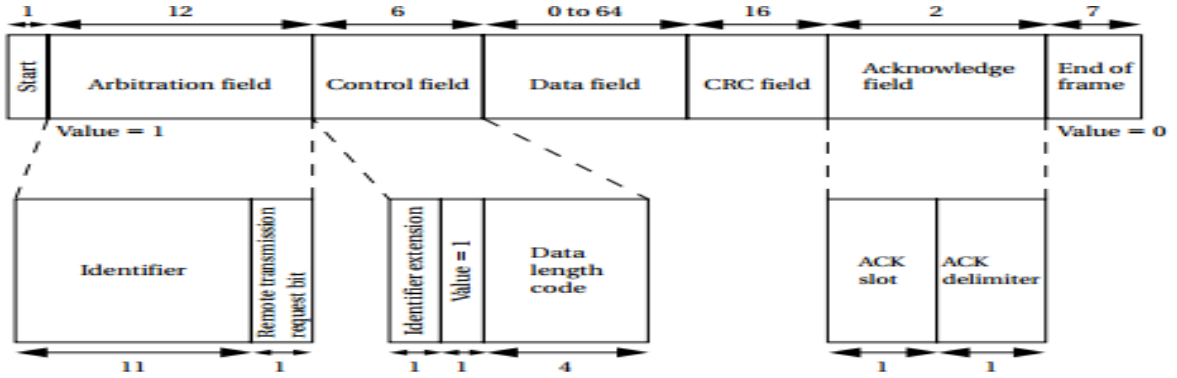


FIGURE 8.23

The CAN data frame format.

Explanation for data frame :

- A data frame starts with a 1 and ends with a string of seven zeroes. (There are at least three bit fields between data frames.)
- The first field in the packet contains the packet‘s destination address and is known as the arbitration field. The destination identifier is 11 bits long.

- The trailing remote transmission request (RTR) bit is set to 0 if the data frame is used to request data from the device specified by the identifier.
- When RTR 1, the packet is used to write data to the destination identifier.
- The control field provides an identifier extension and a 4-bit length for the data field with a 1 in between. The data field is from 0 to 64 bytes, depending on the value given in the control field.
- A cyclic redundancy check (CRC) is sent after the data field for error detection.
- The acknowledge field is used to let the identifier signal whether the frame was correctly received: The sender puts a recessive bit (1) in the ACK slot of the acknowledge field; if the receiver detected an error, it forces the value to a dominant (0) value.
- If the sender sees a 0 on the bus in the ACK slot, it knows that it must retransmit. The ACK slot is followed by a single bit delimiter followed by the end-of-frame field.

Architecture of CAN controller:

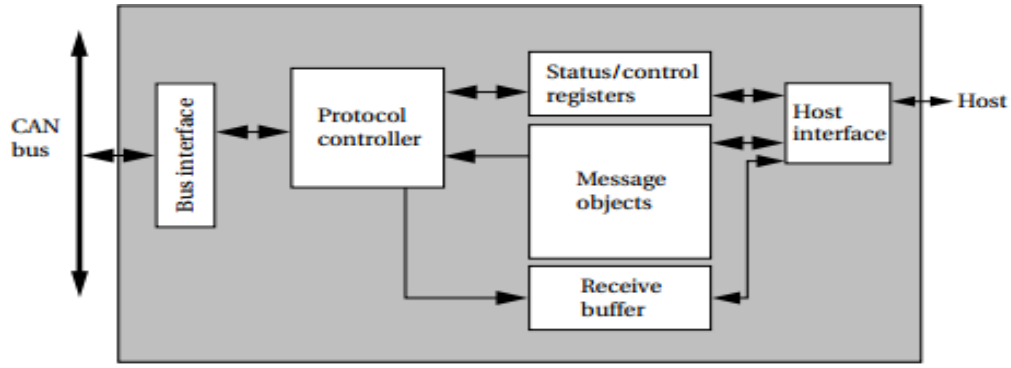


FIGURE 8.24
Architecture of a CAN controller.

- The controller implements the **physical and data link layers**;
- since CAN is a bus, it does not need network layer services to establish end-to-end connections.
- The protocol control block is responsible for determining when to send messages, when a message must be resent due to arbitration losses, and when a message should be received.

INTERNET ENABLED SYSTEMS:

IP Protocol:

- The Internet Protocol (IP) is the fundamental protocol on the Internet.
- It provides connectionless, packet-based communication.
- it is an internetworking standard.
- an Internet packet will travel over several different networks from source to destination.
- The IP allows data to flow seamlessly through these networks from one end user to another

Figure 8.19 explanation:

- IP works at the network layer.
- When node A wants to send data to node B, the application‘s data pass through several layers of the protocol stack to send to the IP.
- IP creates packets for routing to the destination, which are then sent to the data link and physical layers.
- A node that transmits data among different types of networks is known as a router.

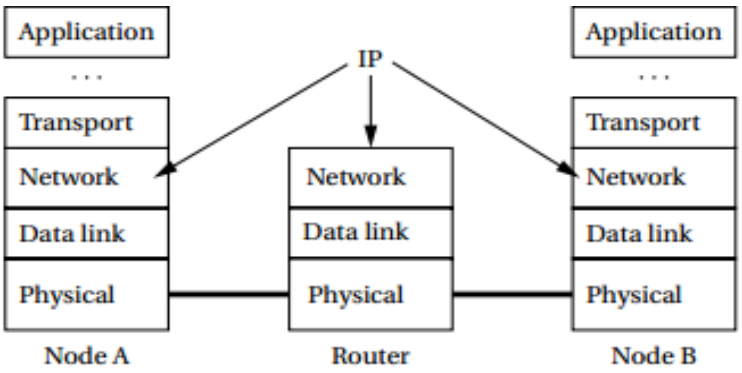


FIGURE 8.19

Protocol utilization in Internet communication.

IP Packet Format:

- The header and data payload are both of variable length.
- The maximum total length of the header and data payload is 65,535 bytes.
- An Internet address is a number (32 bits in early versions of IP, 128 bits in IPv6). The IP address is typically written in the form xxx.xx.xx.xx.
- packets that do arrive may come out of order. This is referred to as **best-effort routing**. Since routes for data may change quickly with subsequent packets being routed along very different paths with different delays, real-time performance of IP can be hard to predict.

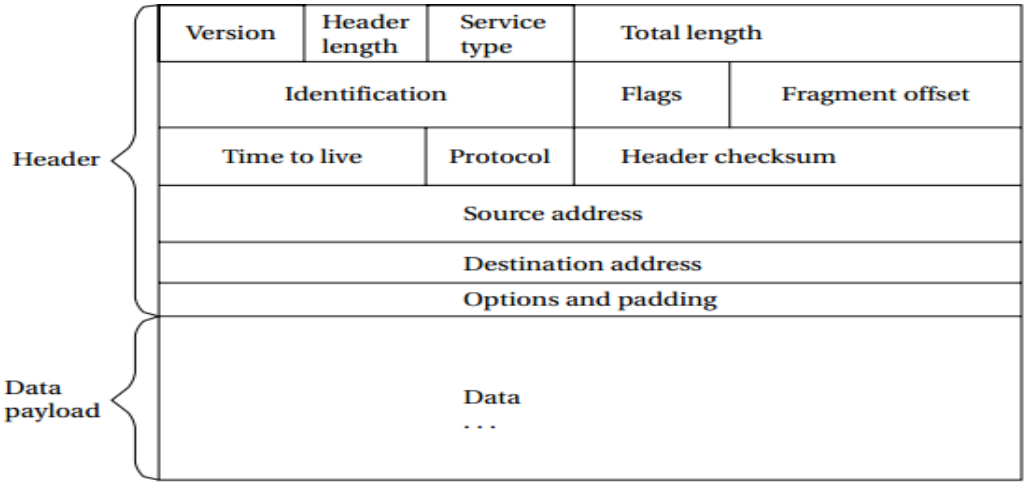


FIGURE 8.20
IP packet structure.

relationships between IP and higher-level Internet services:

Using IP as the foundation, TCP is used to provide File Transport Protocol for batch file transfers, Hypertext Transport Protocol (HTTP) for World Wide Web service, Simple Mail Transfer Protocol for email, and Telnet for virtual terminals. A separate transport protocol, User Datagram Protocol, is used as the basis for the network management services provided by the Simple Network Management Protocol

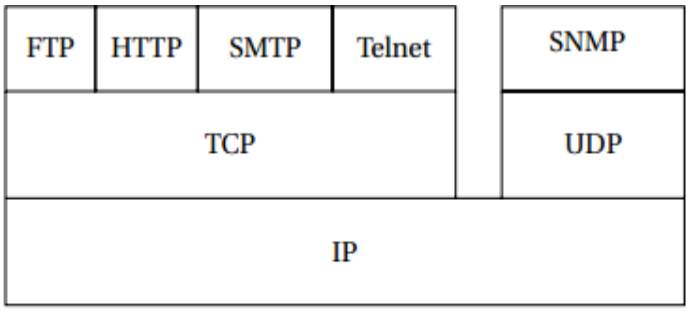


FIGURE 8.21
The Internet service stack.

Design of elevator controller :

- An elevator system is a vertical transport vehicle that efficiently moves people or goods between floors of a building. They are generally powered by electric motors.
- The most popular elevator is the rope elevator. In the rope elevator, the car is raised and lowered by transaction with steel rope.
- Elevators also have electromagnetic brakes that engage, when the car comes to a stop. The electromagnetic actually keeps the brakes in the open position. Instead of closing them with the design, the brakes will automatically clamp shut if the elevator loses power.
- Elevators also have automatic braking systems near the top and the bottom of the elevator shaft.